EIFFEL:

OBJECT-ORIENTED DESIGN

FOR SOFTWARE ENGINEERING

Bertrand Meyer

Jean-Marc Nerson

Interactive Software Engineering, Inc.
270 Storke Road, Suite 7, Goleta, California 93117 – (805) 685-1006
and *Société des Outils du Logiciel*
3 bis Cité d'Hauteville, 75010 Paris

Masanobu Matsuo

Sumitomo Electric Industries, Ltd.
Osaka, Japan and Los Angeles, California

1 - Introduction

Eiffel is an object-oriented language and environment combining the advances in reusable, flexible programming brought by object-oriented techniques with the concerns introduced by modern software engineering, particularly correctness, reliability and efficiency.

As a language, Eiffel offers an extensive combination of facilities: multiple and repeated inheritance, polymorphism and dynamic binding, deferred features, export controls, generically parameterized classes, assertions and invariants expressing correctness arguments, full static type checking.

The implementation, currently available on Unix (System V and BSD), provides automatic configuration management, efficient resolution of dynamic binding (in constant time), and a set of tools for automatic documentation and debugging. It is complemented by a library of carefully designed software components.

Eiffel is designed as a portable and open system. Eiffel classes may be combined with code written in other languages, and portable C packages may be produced from Eiffel text. Care has been taken, however, to preserve the integrity of the language; the interfaces between Eiffel classes and external code are strictly delimited.

Eiffel is not an experimental tool but a production system which has been installed at a number of user sites in Europe, Japan and North America, both in companies and universities, and has been applied to a number of significant developments since it was made commercially available in September of 1986.

Beyond the language and environment aspects, Eiffel promotes a method of software construction by combination of reusable and flexible modules.

2 - Object-oriented design

The design of Eiffel was an attempt to provide quality-conscious software developers – including ourselves, since Eiffel was an internal tool before it was made publicly available – with a environment to produce high-quality software. The software qualities factors that were particularly sought in this endeavor were correctness, reliability, reusability, extendibility and reusability.

In our opinion, the principles of object-oriented design provide the best known techniques for attaining these qualities. The definition of object-oriented design which serves as the basis for this discussion is the following: object-oriented design is **the construction of software systems as structured collections of abstract data type implementations.**

Eiffel classes are abstract data type implementations not only in principle, but more concretely through the use of explicit assertions (preconditions, postconditions, invariants) which reflect the axioms of the abstract data type specifications.

The word *structured*, as used in the definition, refers to the mechanisms used for combining modules. The

Eiffel structuring mechanisms are multiple and repeated inheritance, direct use (the client relation) and genericity (type parameterization).

Eiffel shares the basic properties of object-oriented languages such as Simula 67 [1, 10], Smalltalk [6], C++ [19] Objective C, [4], Object Pascal [20], Trellis/Owl [17], Loops [2], Flavors [3] and Ceyx [7]. But we think that Eiffel goes further than any existing tool by the combination of features it offers, and by its emphasis on correctness and reliability as well as reusability and flexibility. Many concepts are original: the particular combination of genericity and inheritance, the approach to type checking, the use of assertions in an object-oriented world, the renaming policy applied to multiple inheritance, the provisions for constant and shared objects, the notion of repeated inheritance, etc. Another distinctive feature of Eiffel, beyond the language aspects, is the set of supporting tools which make it a powerful development environment.

In spite of its advanced features, the language is small; although obviously not a complete description, this article introduces its essential constructs. Great care has been taken to keep any non-essential features out in the interest of simplicity, ease of use, and ease of learning.

Other references on Eiffel include an informal overview [16], a detailed description of the language and library [12], a study of the relationship between genericity and inheritance [13] and an Eiffel-based analysis of software reusability issues [14]. A book on object-oriented design and the Eiffel approach is in preparation [15].

3 - An example class: stack

We will introduce the language through a set of example classes providing different implementations of stacks. These classes are given verbatim from the basic Eiffel library [8].

The general notion of stack is given by the following class.

```
capsule class STACK [T]
      -- Stacks defined independently of
      -- any specific representation.

export
   nb_elements, empty, full,
   top, push, pop, change_top
   wipe_out

feature

   nb_elements: INTEGER is
         -- Number of elements inserted.
      deferred
      end; -- nb_elements

   empty: BOOLEAN is
         -- Is stack empty?
      do
         Result := (nb_elements = 0)
      ensure
         Result = (nb_elements = 0)
      end; -- empty

   full: BOOLEAN is
         -- Is stack full?
      deferred
      end; -- full

   top: T is
         -- Last element pushed.
      require
         not empty
      deferred
      end; -- top

   push (x: T) is
         -- Push x onto stack.
      require
         not full
      deferred
      ensure
         not empty; top = x;
         nb_elements = old nb_elements + 1
      end; -- push

   pop is
         -- Remove top element.
      require
         not empty
      deferred
      ensure
         not full;
         nb_elements = old nb_elements - 1
      end; -- pop

   change_top (x: T) is
         -- Replace top element by x
      require
         not empty
      do
         pop; push (x)
      ensure
         not empty; top = x
         nb_elements = old nb_elements
      end; -- change_top

   wipe_out is
         -- Remove all elements.
      deferred
      ensure
         empty
      end; -- wipe_out
invariant
   nb_elements >= 0
end -- class STACK
```

This class describes the behavior of a certain category of data structures: stacks. It has a generic parameter, *T*, representing the type of elements pushed onto a stack; genericity provides flexibility in a manner compatible with strict typing.

Any class is characterized by a number of **features**, describing properties of stacks: the number of elements, the ability to push an element, etc. Features include **attributes**, or data fields of instances of the class (stacks), and **routines**, or operations. Routines are further divided into procedures and functions.

As we shall see below, other classes, called **clients** of this class, may use it to manipulate stacks. Clients will access stacks through the given features; only those features which are listed in the "export" clause are available to clients. A class may thus keep for itself its implementation secrets.

This particular class describes rather abstract stacks, independent of any particular representation. This is why the body of many routines (see *push*, *pop* etc.) is given not by actual code but by the keyword **deferred**, which means: "I am telling you that this routine must eventually be somehow implemented, but I can't give you the details of its implementation(s) here". (This is a generalization of the Simula "virtual" concept.) As we shall see, specific implementations may be given in other classes, called **descendants** of this class. A descendant provides a specialized or extended version of a previously defined class.

Any class which has at least one deferred routine is called a "capsule" class and marked accordingly. Not all routines of a capsule class need to be deferred. Some routines are expressed in terms of deferred procedures but are not themselves deferred (see *empty* and *change_top*).

A very important component of the Eiffel approach is the ability to characterize a routine, whether deferred or not, by a precondition (**require**), or a postcondition (**ensure**), or both. Such assertions (generalized boolean expressions) enable precise characterization of the routine's effect; they are useful for correct program writing, for documentation and for debugging (as assertions may be checked, on option, at run-time). The same holds of **class invariants**, which must be satisfied after the creation of each instance of the class and maintained by every routine. The reader is invited to verify that the assertions given in the above class do reflect the intuitive properties of classes.

Thanks to assertions, the above capsule class almost gives a full abstract data type specification (although certain axioms may not be easily expressed in this framework). This use of capsule classes, where the abstract properties of routines are specified by assertions even though no implementation is given, is a key technique in using Eiffel as a **design language**. Eiffel is in fact of a higher-level than most existing PDLs (Program Design Languages). A considerable advantage over a standard PDL is that Eiffel is also a programming language: a capsule class may be refined smoothly, through inheritance (see below), into an executable class. This makes it possible to go from design to implementation in a continuous process, avoiding the gap that characterizes traditional methods.

What can we do with a class such as *STACK*? Among other things, a class may be used as a type. Other classes (the clients of *STACK*) may declare and manipulate entities of that type, representing stacks. The operations available on such entities are given by the features of the class. Thus a client might contain code like

```
s1: STACK [INTEGER]; s2: STACK [FIGURE];
f: FIGURE;

 ............
s1.push (3); if s1.empty then ...
s2.push (f); s2.push (f); s2.pop;
```

Note how actual stack entities use specific types (*INTEGER* and *FIGURE*) for the generic parameter *T*; *INTEGER* is one of the predefined language types, and we assume that a class *FIGURE* has been defined, representing geometrical figures.

Type checking is total here and allows maintaining the integrity of each structure; in other words, an attempt to push a figure onto a stack of integers would be rejected at compile-time. Note that the implementation of genericity nevers entails any code duplication.

The reader may have been surprised, when reading the text of *STACK*, by the absence of a *STACK* parameter to routines such as *push*, *pop*, *top* etc. Where is the stack to which these operations apply? The answer is in the client code given above: dot notation, as in *s1.push (3)*, shows that any of the routines in class *STACK* is always applied to a distinguished stack argument written before the dot. This is one of the key aspects of object-oriented programming: every operation applies to a special target object. Within the class, this object (the "typical instance" of the class) remains implicit most of the time; this is why *top*, for example, has no parameter (it returns a property of the typical stack), and *push* has only one (the value being pushed).

4 - Multiple inheritance: fixed stacks

Using inheritance, a class may be defined on the basis of one or more previously defined classes, called its parents. The new class, called an heir, will possess all the features of its parents, to which it can add its own; some of the parents features may also be overridden, as we shall see below. The following example, an heir of *STACK* called *FIXED_STACK*, will serve to evidence many of the properties of this remarkable technique.

```
class FIXED_STACK [T]
      -- Stacks with a fixed physical size,
      -- implemented as arrays

export
   max_size,
   nb_elements, empty, full,
   top, push, pop, change_top,
   wipe_out

inherit

   STACK [T]
      redefine change_top;

   ARRAY [T]
      rename
         Create as array_Create, size as max_size
feature

   nb_elements: INTEGER;
         -- Note redefinition as attribute

   Create (n: INTEGER) is
         -- Allocate stack with maximum size n
      do
         array_Create (1, n)
      ensure
         max_size = n
      end; -- Create

   full: BOOLEAN is
         -- Is stack full?
      do
         Result := (nb_elements = max_size)
      ensure
         Result = (nb_elements = max_size)
      end; -- full

   top: T is
         -- Last element pushed.
      require
         not empty
      do
         Result := entry (nb_elements)
      end; -- top

   push (x: T) is
         -- Push x onto stack
      require
         not full
      do
         nb_elements := nb_elements + 1;
         enter (nb_elements, x)
      ensure
         not empty; top = x;
         entry (nb_elements) = x;
         nb_elements = old nb_elements + 1
      end; -- push

   pop is
         -- Remove top element.
      require
         not empty
      do
         nb_elements := nb_elements + 1
      ensure
         not full;
         nb_elements := old nb_elements - 1
      end; -- pop

   change_top (x: T) is
         -- Replace top element by x
      require
         not empty
      do
         enter (nb_elements, x)
      ensure
         not empty; top = x
         nb_elements = old nb_elements
      end; -- change_top

   wipe_out is
         -- Remove all elements.
      do
         nb_elements := 0
      ensure
         empty
      end; -- wipe_out
invariant
      nb_elements >= 0; nb_elements < max_size
end -- class FIXED_STACK
```

Fixed stacks are stacks of fixed size; they are simply obtained by inheriting from the basic class *ARRAY* as well as *STACK*. Class *ARRAY* describe arrays of a given type, characterized by the following routines:

- *enter (i, x)* to assign value x to the i-th element of an array;
- *entry (i)*: the value of the i-th element.
- *Create*: array allocation.

The mechanism of multiple inheritance is one of the keys to the power of Eiffel programming. Inheritance makes it possible to design systems by successive extension and specialization, starting from simple classes. Many available object-oriented environments permit only single inheritance (every class may inherit from at most one parent); this is in our view an unacceptable limitation to the power of the method. Our own software developments make a considerable use of multiple inheritance; inheriting from five or six parent classes is a common occurrence.

This style of program design is a radical departure from traditional approaches. Instead of building every new system from scratch (a method also known as top-down design), the designer works by reuse, combination, extension and specialization.

In the above example, the marriage is one between a "noble" family (*STACK*, offering the functionality) and a "rich" family (*ARRAY*, offering the implementation). This is a common occurrence in practical uses of multiple inheritance.

A class may, of course, have an arbitrary number of direct or indirect heirs (or "descendants"; the reverse notion is "ancestor"). The Eiffel library, for example, includes another implementation of stacks: class *LINKED_STACK* which, as the reader will have guessed, is obtained by multiple inheritance from *STACK* and the library class *LINKED_LIST*. This class is similar in form to *FIXED_STACK*, with appropriate routine implementations, using linked list operations, substituted for the array implementations.

5 - Renaming, redefinition and dynamic binding

The power of inheritance is domesticated in Eiffel through explicit renaming and redefinition.

The problem of renaming arises because of multiple inheritance, which carries a serious risk of confusion if two parent classes happen to include features with identical names. Such name clashes are prohibited. They must be removed in the inheritance clause, using the **rename** construct, which will make inherited features available locally under a different name. In the above example, renaming is in fact applied for other purposes: making features from a parent class available in their original form even though they are redefined for the new class (see *Create* from arrays); or simply to using a locally more appropriate name (see *max_size*).

Another key property of inheritance is the ability to override a routine implementation in a descendant. At the *STACK* level, for example, since the representation is not known, routine *change_top* is implemented as a *pop* followed by a *push*. For fixed stacks, a more efficient implementation is possible; hence the new version of *change_top* in class *FIXED_STACK*, which simply changes the value of an array element. For stacks of type *FIXED_STACK*, this version will be used instead of the general *STACK* implementation.

A special form of redefinition is the introduction of an actual definition for a routine which was deferred in an ancestor. Eiffel permits a routine to be implemented as an attribute: for example, *nb_elements*, which was a deferred function at the *STACK* level, is implemented as an attribute in *FIXED_STACK*. This possibility is important for information hiding: when a client requests the number of elements of a stack *s* through the notation *s.nb_elements*, whether *nb_elements* is an attribute or a function (that is to say, stored or computed) is irrelevant for the client.

The redefinition mechanism allows full **polymorphism** and **dynamic binding**. Polymorphism is the ability for an entity to refer at run-time to objects of various types; for example, the following is permitted:

```
s: STACK [X];
f: FIXED_STACK [X]; l: LINKED_STACK [X];
.........
s := f; ........; s := l
```

in other words, a *STACK* entity may at run-time refer to a *FIXED_STACK* or a *LINKED_STACK* object. Eiffel is, however, strictly typed: here, typing means that the type of the source (here *FIXED_STACK* or *LINKED_STACK*) can only be a descendant class of the type of the target (*STACK*). The reverse assignment would not be permitted. The motivation is clear: a stack is a more general concept than a fixed stack; so a stack entity may be assigned a fixed stack value, but not conversely.

Dynamic binding, in this context, means that if different descendants of a class provide alternative versions of a routine such as *push*, the version to be applied in a call of the form

s.push

depends on the run-time form of *s*. This dynamic adaptation of operations to the form of the objects to which they apply is one of the keys to the modular, decentralized programming style permitted by object-oriented languages. Instead of one *push* routine which must know about every variety of stack in the world, we have autonomous variants of the *STACK* notion, each of which comes with its own version of *push* and other operations. The binding between the operation name and the appropriate variant is done only at run-time. (As we shall see below, the Eiffel implementation techniques achieve this result with very little run-time overhead.)

The very power of the mechanisms for multiple inheritance, redefinition, polymorphism and dynamic binding implies risks to the consistency and correctness of programs. How do we make sure that a call to *push* does not trigger a completely different routine because of dynamic binding? Eiffel offers a series of safeguards:

1 • As we have seen, polymorphism is limited by the type rules. As a result, a run-time type error may **never** occur: the type system is such that any incorrect routine application *entity.routine* is rejected by the compiler. In a correctly compiled program, *routine* is guaranteed to be defined for *entity* whenever the application is executed, even though *entity* may refer to instances of widely different classes.

2 • Redefinition may not occur by accident: to override a routine from a parent class, the class author must list the routine in the **redefine** clause, as with *change_top* in *FIXED_STACK*. Inadvertantly reusing an ancestor's routine name for a new routine will result in a compile-time error, not in unwanted redefinition.

3 • Finally, the redefinition mechanism is controlled by assertions. The language rule is that a redefinition of a routine must obey its initial specification: the precondition may be kept or weakened, and the postcondition may be kept or

strengthened. Although in the absence of a built-in theorem prover this rule can only be enforced at run-time, it is a powerful methodological guideline.

The last remark is worth some more explanations. Although assertions have been associated with programming languages before [9, 18], we have found their systematic use in conjunction with object-oriented techniques to bring a completely new perspective to program design. Underlying their use in Eiffel is the notion of *programming as a contractual activity*: assertions spell out the terms of the contract between a client and a supplier. Inheritance and dynamic binding, in this context, are **subcontracting** mechanisms; the above rules, governing adaptation of assertions in redefined routines, guarantee that the subcontractor does not violate the terms of the original contract. These ideas have been thoroughly applied to the design and validation of the components of the basic Eiffel library. They are explained in detail in the forthcoming book [15].

6 - The implementation

The current implementation of Eiffel runs on various versions of Unix. It is based on translation from Eiffel to C, then C to machine code using the resident C compiler. It may potentially be ported to any environment including a C compiler and some basic operating system capabilities.

Eiffel is in no way an extension of C; C is only used as a vehicle for implementation and has had no influence on the language design. Other compilation techniques would be possible, but the use of a portable assembly language such as C as intermediate code has obvious advantages for transferability of the implementation.

We feel very strongly about the independence of Eiffel from C. The addition of some object-oriented concepts to a language stem which (independently of any value judgment on C) is not object-oriented by any reasonable definition can only result in inconsistent constructions, difficult to teach and to use. The complex hybrid scope system of C++ [5] is a case in point. In our view, simplicity and consistency are princely qualities in the design of a programming language.

Although Eiffel is not an extension of an existing language, this does not mean that Eiffel programmers are isolated from the rest of the world. To the contrary, the implementation is highly open. Eiffel classes are meant to be interfaced with code written in other languages. This concern is reflected in the language by the optional **external** clause which, in a routine declaration, lists external subprograms used by the routine.

This mechanism makes it possible to use external routines without impacting the conceptual consistency of the Eiffel classes. In particular, the external routines are used by clients not directly but through Eiffel routines, which may include a precondition and postcondition.

A good external facility was essential in view of the design objectives listed in section 2: when you introduce an environment which claims to promote reusability, you can hardly tell prospective users to throw away all the software they have written so far.

One application of the "external" construct is for using Eiffel as an integrating mechanism for components written in other languages. Eiffel is in fact a useful tool for combining software written in various languages, and as a tool for package writers who want to distribute their software in C form.

Great care has been taken to provide efficient compilation and execution. In particular, the resolution of dynamic binding takes constant time: in *entity.routine*, the time to find the appropriate version of *routine*, which depends on the run-time form of *entity*, is bounded by a small constant. In many object-oriented systems, the resolution implies a potentially costly run-time search; the problem is particularly acute with multiple inheritance.

Other features of the implementation are worth noting:

- There is never any duplication of code (except in a rather rare and special occurrence, "repeated" inheritance with feature duplication, not described in this paper, where duplication of some features is conceptually necessary).
- Memory management is handled by a run-time system that takes care of object creation and (system-controlled) de-allocation.
- A systemwide optimizer cleans up the generated code, removes all unneeded routines (a potentially serious problem for very large applications in an approach encouraging the reuse of powerful software components) and substitutes direct calls for routines that are never redefined (thus removing *any* dynamic binding overhead in this case).
- Compilation is performed on a class-by-class basis, so that large systems can be changed and extended incrementally. The translation time from Eiffel to C is typically about 50% of the time for the next step, translation from C to machine code.

7 - The environment

The construction of systems in Eiffel is supported by a set of development tools.

Most important are the facilities for automatic configuration management integrated into the compilation commands. When a class *C* is compiled, the system automatically looks for all classes on which *C* depends directly or indirectly (as client or heir), and re-compiles those whose compiled versions have become obsolete. Unix programmers will recognize this facility as giving the power of Make without programmer-written makefiles.

This is far from being a trivial problem, as the dependency relations are complex (a class may be, say, a client of one of its descendants) and, in the case of the client relation, may involve cycles. The problem is made even more difficult by the presence of remote feature calls of the form *a.b.c.d....* We felt it essential, however, to find a solution, as this completely frees programmers from having to keep track of changed modules to maintain the consistency of their systems. Note that the algorithm used is able to avoid many unneeded recompilations by detecting that a modification made to a class has not impacted its interface; this has proved very important in practice, as it avoids triggering a chain reaction of re-compilations in a large system when the implementation of a feature is changed in a low-level class.

Thanks to these facilities, the environment gives Eiffel programmers all the safety and run-time efficiency of a compiler-oriented system, while achieving a try-change-retry cycle time almost as short as in interpretive systems.

The environment also contains debugging tools: tools for run-time checking of assertions; a tracer and symbolic debugger; and a viewer for visual, interactive exploration of the object structure at run-time.

A documentation tool, **short**, produces a summary version of a class showing only its official information: the exported features only and, in the case of routines, only the header, precondition and postcondition. An option of **short** produces formated output in **-troff -ms** form for typesetting; for example, the following is the result of applying **short** to one of the above classes:

```
class interface FIXED_STACK [T] exported features
  max_size, nb_elements, empty, full, top, push, pop,
  change_top, wipe_out
inherit
  STACK [T]
    redefine
      change_top;
  ARRAY [T]
    rename
      size as max_size,
feature specification
  Create (n: INTEGER)
      -- Allocate stack with maximum size n
    ensure
      max_size = n

  nb_elements: INTEGER

  full: BOOLEAN
      -- Is stack full?
    ensure
      Result = (nb_elements = max_size)

  top: T
      -- Last element pushed.
    require
      not empty

  push (x: T)
      -- Push x onto stack
    require
      not full
    ensure
      not empty;
      top = x;
      nb_elements = old nb_elements + 1


  pop
      -- Remove top element.
    require
      not empty
    ensure
      not full;
      nb_elements := old nb_elements - 1


  change_top (x: T)
      -- Replace top element by x
    require
      not empty
    ensure
      not empty;
      top = x
      nb_elements = old nb_elements


  wipe_out
      -- Remove all elements.
    ensure
      empty
invariant
  nb_elements >= 0;
  nb_elements < max_size
end interface -- class FIXED_STACK
```

The ability to automatically produce such high-quality documentation – concise, high-level (thanks to the assertions) and, most importantly, guaranteed to be consistent with the software being documented – has proved to be a key benefit of the Eiffel environment. Consistency is almost impossible to achieve if documentation, design and code are considered as separate products.

The Eiffel library manual [8] is almost entirely produced by **short**.

A postprocessor performs various optional modifications on the generated C code: removal of unneeded routines, simplification of calls to non-polymorphic routines, packaging of the entire generating code as a library of routines callable from other programs. The last option is particularly interesting for software implementors who use Eiffel for the development of packages that will be delivered to their customers in standard C form, or who must do cross-development (program writing on one machine, execution on another which does not necessarily support Eiffel.)

Finally the environment includes a **library** of robust, efficient, carefully crafted basic classes, covering some of the most important data structure implementations. Use of this library is one of the elements that give Eiffel programming its distinctive flavor, enabling programmers to think and write in terms of lists, trees, stacks etc. rather than arrays, pointers, flags and the like. The library is being further refined so as to cover all the common patterns of everyday programming.

8 - Status and further work

Eiffel has been available internally within Interactive Software Engineering since the Spring of 1986, and commercially since September of that year. The system has been ported to various versions of Unix, on such machines as Sun-2 and 3, Apollo workstations, Sumitomo Electric U-station, NCR Power 32, Perkin-Elmer, VAX, Ridge/Bull SPS 9, AT&T 3B2-3B5, Eunice, and others. Porting to VAX-VMS and other non-Unix environments is in progress. A number of systems have been installed in Europe, Japan and North America, both in industrial companies and in universities, and have been applied to the development of various applications.

Based on this experience, we do not plan any significant changes to the language. However more work is being pursued in the area of support for concurrency and event-driven computation. Although some support for persistent objects is already offered by the present system, more advanced facilities are needed in this area. As regards the implementation, the tools for configuration management and documentation, which have proved invaluable in practice, must be complemented by "browsing" facilities to enable exploration and retrieval of existing classes and features. Some primitive exploration tools are available; rather than a Smalltalk-like browser, however, which we feel would not scale up to real reusability on an industrial basis, we are investigating the use of repositories of software components, organized around a data base management system. Finally, we would like to better integrate Eiffel with the formal specification method M [11], based on many of the same principles. This would give us a method covering most of the software lifecycle activities.

9 - Conclusion

A number of individual concepts embodied in Eiffel were present in previous languages, notably Simula 67, Ada and Alphard. However the design has brought in many new contributions.

From the language standpoint, one may quote the safe treatment of multiple inheritance through renaming, the combination between genericity and inheritance, disciplined polymorphism by explicit redefinition, the integration of the assertion/invariant mechanism with inheritance, a clean interface with external routines, and the introduction of full static typing into an object-oriented language with multiple inheritance.

From the implementation standpoint, a number of our solutions are original: constant-time routine access, separate compilation with automatic configuration management in an object-oriented world, support for debugging and automatic interface documentation, support for the preparation of deliverable software packages.

More generally, we believe quite frankly that Eiffel is the first full-scale effort enabling developers of practical software systems to take advantage of object-oriented techniques in a manner consistent with the newest concepts of software engineering.

We first designed Eiffel for our own needs as software developers and, now that we have used it extensively for small and large developments alike, we wouldn't trade it for anything else. We hope that this article will have inoculated the reader with at least some of our enthusiasm.

References

1. Graham Birtwistle, Ole-Johan Dahl, Bjorn Myrhaug, and Kristen Nygaard, *Simula Begin*, Studentliteratur and Auerbach Publishers, 1973.
2. Daniel G. Bobrow and M.J. Stefik, *LOOPS: an Object-Oriented Programming System for Interlisp*, Xerox PARC, 1982.
3. H.I. Cannon, "Flavors," Technical Report, MIT Artificial Intelligence Laboratory, Cambridge (Massachussets), 1980.
4. Brad J. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading (Massachusetts), 1986.
5. S.C. Dewhurst, "Object Representation of Scope during Translation," in *ECOOP 87: European Conference on Object-Oriented Programming*, Paris, June 1987.
6. Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading (Massachusets), 1983.
7. Jean-Marie Hullot, "Ceyx, Version 15: I - une Initiation," Rapport Technique no. 44, INRIA, Rocquencourt, Eté 1984.
8. Interactive Software Engineering, Inc., "Eiffel Library Manual," Technical Report TR-EI-7/LI, 1986.
9. Bruce W. Lampson, Jim J. Horning, Ralph L. London, J. G. Mitchell, and Gerard L. Popek, "Report on the Programming Language Euclid," *SIGPLAN Notices*, vol. 12, no. 2, pp. 1-79, February

1977.

10.Bertrand Meyer, "Quelques concepts importants des langages de programmation modernes et leur expression en Simula 67," *Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique)*, no. 1, pp. 89-150, Clamart (France), 1979. Also in GROPLAN 9, AFCET, 1979.

11.Bertrand Meyer, "M: A System Description Method," Technical Report TRCS85-15, University of California, Santa Barbara, Computer Science Department, August 1986. Submitted for publication.

12.Bertrand Meyer, "Eiffel: A Language and Environment for Software Engineering," *The Journal of Systems and Software*, 1987. To appear.

13.Bertrand Meyer, "Genericity, static type checking, and inheritance," *The Journal of Pascal, Ada and Modula-2*, 1987. To appear (Revised version of paper in ACM OOPSLA conference proceedings, Portland, Oregon, September 1986, pp. 391-405).

14.Bertrand Meyer, *Reusability: the Case for Object-Oriented Design*, 4, pp. 50-64, IEEE Software, March 1987.

15.Bertrand Meyer, *Object-oriented Software Construction*, 1987. To appear

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

17.Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt, "An Introduction to Trellis-Owl," in *OOPSLA '86 Conference Proceedings, Portland Oregon, Sept. 29-Oct. 2, 1986*, pp. 9-16, 1986. (Published as SIGPLAN Notices, 21, 11, Nov. 1986.)

18.Mary Shaw (Ed.), in *Alphard: Form and Content*, Springer-Verlag, New York, Heidelberg, Berlin, 1981.

19.Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, Menlo Park (California), 1986.

20.Larry Tesler, "Object Pascal Report," *Structured Language World*, vol. 9, no. 3, 1985.