EIFFEL: PROGRAMMING FOR REUSABILITY AND EXTENDIBILITY

Bertrand Meyer

Interactive Software Engineering, Inc.

270 Storke Road, Suite 7, Goleta, CA. 93117 – (805) 685-1006

1 - Introduction

Eiffel is a language and environment intended for the design and implementation of quality software in production environments. The language is based on the principles of object-oriented design, augmented by features enhancing correctness, extendibility and efficiency; the environment includes a basic class library and tools for such tasks as automatic configuration management, documentation and debugging.

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

The present note is a general introduction to Eiffel. More detailed information [1, 2, 3] is available.

2 - Design principles

Software quality is a tradeoff between many factors [4]. In the current state of the software industry, some of these factors appear worth a particular effort. One is *reusability*, or the ability to produce software components that may be used in many different applications; observation of programmers shows that many of the same program patterns frequently recur, but that actual reuse of program modules is much less widespread than it ought to be. Another important factor is *extendibility*: although software is supposed to be ''soft'', it is notoriously hard to modify software systems, especially large ones.

As compared to other quality factors, reusability and extendibility play a special rôle as their satisfaction means that *less* software has to be written, and thus, presumably, that more time may be devoted to the other goals (such as efficiency, ease of use etc.). Thus it may pay off in the long run to concentrate on these two factors, and they were indeed paramount in the design of Eiffel.

Other design goals also played a significant part. Helping programmers ensure *correctness* and *reliability* of their software is of course a prime concern. *Portability* was one of the requirements on the implementation. Finally, *efficiency* cannot be neglected in a tool that is aimed at practical, medium- to large-scale developments.

3 - Object-oriented design

To achieve reusability and extendibility, the principles of object-oriented design seem to provide the best known technical answer to date. This note is not the place to discuss these principles in depth. We shall simply offer a definition of this notion: as discussed here, object-oriented design is **the construction of software systems as structured collections of abstract data type implementations.** The following points are worth noting in this definition:

- the emphasis is on structuring a system around the objects it manipulates rather than the functions it performs on them, and on reusing whole data structures, together with the associated operations, rather than isolated procedures.
- Objects are described as instances of abstract data types – that is to say, data structures known from an official interface rather than through their representation.
- The basic modular unit, called the class, describes the implementation of an abstract data type (not the abstract data type itself, whose specification would not necessarily be executable).
- The word *collection* reflects how classes should be designed: as units which are interesting and useful on their own, independently from the systems to which they belong. Such classes may

then be reused in many different systems. System construction is viewed as the assembly of existing classes, not as a top-down process starting from scratch.

- Finally, the word *structured* reflects the existence of important relationships between classes, particularly the **multiple inheritance** relation.

Following Simula 67 [5], several object-oriented languages have appeared in recent years, such as Smalltalk [6], extensions of C such as C++ [7] and Objective C [8], of Pascal such as Object Pascal [9], of Lisp such as Loops [10], Flavors [11] and Ceyx [12]. Eiffel shares the basic properties of these languages but goes beyond them in several respects, by offering multiple inheritance (which, of the systems quoted, is only available in recent versions of Smalltalk, on an experimental basis, and in the Lisp extensions), generic classes, static type checking (whereas most existing languages are either untyped or offer dynamic checking only) and facilities for redefinition and renaming (see below). Furthermore, Eiffel combines the object-oriented heritage with a strong software engineering influence, through facilities for information hiding, expression of formal program properties (assertions, invariants) and a strict naming policy. Finally the language comes with an environment geared towards the development of sizable software in production environments.

The rest of this presentation introduces the main language features informally, through a set of examples, and describes the principal aspects of the implementation and environment.

One remark is in order on the size of the language. Eiffel includes more than presented in this introduction, but not *much* more; it is a small language, comparable in size (by such an approximate measure as the number of keywords) to Pascal. Thus Eiffel belongs to the class of languages which programmers can master entirely (as opposed to languages of which most programmers know only a subset). Yet it is appropriate for the development of significant software systems, as evidenced by our own experience with such programs as Cépage [13] (which, at the time of this writing, is a 45,000-line Eiffel system) and other examples.

4 - Classes

We first introduce a typical class. As mentioned above, a class represents the implementation of an abstract data type, that is to say a class of data structures known through an official interface. A simple example would be a a bank account. Let *ACCOUNT* be the corresponding class. Before we show the class, we first describe how it would be used; such a use has to be in another class, say *X*, as classes are the only program unit in Eiffel.

To use *ACCOUNT*, class *X* may introduce an entity and declare it of this type:

acc1: ACCOUNT

The term "entity" is preferred here to "variable" as it corresponds to a more general notion. An entity declared of a class type, such as *acc1*, may at any time during execution refer to an **object**; an entity which does not refer to any object is said to be void. By default (at initialization) entities are void; objects must be created explicitly, by an instruction

acc1.Create

which associates *acc1* with the newly created object (see figure 1). *Create* is a predefined "feature" of the language.

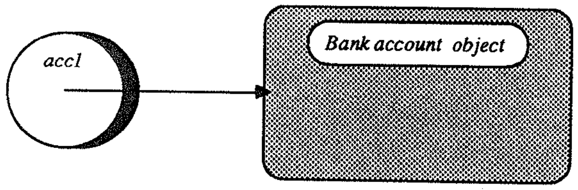

Figure 1: Entity and associated object

Once *acc*1 has been associated with an object, one may apply to it the features defined in the corresponding class. Examples are:

acc1.open ("John");
acc1.deposit (5000);

if *acc1.may_withdraw (3000)* **then**
  acc1.withdraw (3000)
end;
print (acc1.balance)

All feature applications use the dot notation: *entity_name.feature_name*. There are two kinds of features: **routines** (as *open*, *deposit*, *may_withdraw* or *withdraw*), that is to say operations; and **attributes**, that is to say data items associated with objects of the class.

Routines are further divided into procedures (actions, which do not return a value) and functions (which return a value). Here *may_withdraw* is a function with an integer parameter, returning a boolean result; the other three routines invoked are procedures.

The above extract of class *X* does not show whether, in class *ACCOUNT*, *balance* is an attribute or a function without parameters. This ambiguity is intentional. A class such as *X*, said to be a **client** of *ACCOUNT*, does not need to know how a balance is obtained: it could be stored as attribute of every account object, or re-computed by a function, whenever requested, from other attributes such as the list of previous deposits and withdrawals. Deciding on one of these implementations is the business of class *ACCOUNT*, not anybody else's.

Having shown class usage, we now present the definition of the class itself, which might look like the following. Following the Ada convention, line segments beginning with -- are comments.

```
class ACCOUNT export
  open, deposit, may_withdraw,
  withdraw, balance, owner
feature
  balance: INTEGER ;

  minimum_balance: INTEGER is 1000 ;

  owner: STRING ;

  assign_owner (who: STRING) is
      -- Assign the account to owner who
    do
      owner := who
    end ; -- open

  add (sum: INTEGER) is
      -- Add sum to the balance
      -- (Secret procedure)
    do
      balance := balance+sum
    end ; -- deposit

  deposit (sum: INTEGER) is
      -- Deposit sum into the account
    do
      add (sum)
    end ; -- deposit

  withdraw (sum: INTEGER) is
      -- Withdraw sum from the account
    do
      add (-sum)
    end ; -- withdraw

  may_withdraw (sum: INTEGER): BOOLEAN is
      -- Is it permitted to withdraw sum
      -- from the account?
    do
      Result := (balance >= minimum_balance)
    end ; -- deposit
end -- class ACCOUNT
```

This class includes two clauses: **feature**, which describes the features of the class, and **export**, which simply lists those features which are available to clients of the class. Non-exported features are said to be secret. Here procedure *add* is secret, so that *acc1.add (-3000)* would have been illegal in *X*. Similarly, attribute *minimum_balance* is secret. (Selective export of a feature to some classes only is also possible.)

Let us examine the features in sequence. Routines are distinguished from attributes by having a clause of the form **is**...**do**...**end**. Thus *balance* is in fact an attribute. The clause **is** *1000* introduces *minimum_balance* as a constant attribute, which will not occupy any physical space in objects of the class. Non-constant attributes such as *balance* do use space for each object of the class; they are similar to components of a record in Pascal.

Attributes *balance* and *minimum_balance* are declared of type *INTEGER*. Eiffel is strongly typed: every entity is declared of a certain type. A type is either simple, that is to say one of *INTEGER*, *REAL*, *CHARACTER* and *BOOLEAN*, or a class. Arrays and strings belong to the second category; they are described by predefined system classes, *ARRAY* and *STRING*, treated exactly as user-defined classes with one exception: a special notation, as in "*John*", is available to denote literal string constants.

Automatic initialization is ensured by the language definition, so that the initial *balance* of an account object will be zero after a *Create*. Numeric attributes are initialized to zero, booleans to false, characters to the null character; those of class types are initially void.

The other five features are straightforward routines. The first four are procedures, the last one (*may_withdraw*) a function returning a boolean value; note that the special variable *Result* denotes the function result. It is initialized on function entry to the default value of the appropriate type, as defined above.

To properly understand the routines, it is necessary to remember that in an object-oriented languages any operation is relative to a certain object. In an external client invoking the operation, this object is specified by writing the corresponding entity on the left of the dot, as *acc1* in *acc1.open ("John")*. Within the class, however, the "current" object to which operations apply usually remains implicit, so that unqualified references, such as *owner* in procedure *assign_owner* or *add* in *deposit*, mean "the *owner* attribute or *add* routine relative to the current object". The special variable *Current* may be used, if needed, to explicitly denote this object. Thus the unqualified occurrences of *add* appearing in the above class are equivalent to *Current.add*.

In summary, this simple example shows the basic structuring mechanism of the language, the class. A class describes a data structure, accessible to clients through an official interface comprising some of the class features. Features are implemented as attributes or routines; the implementation of exported features may rely on other, secret ones.

5 - Assertions

It was said at the outset that classes are abstract data type implementations. However an abstract data type is defined not just by a list of available operations, but also by the formal properties of these operations, which do not appear in the above example.

Eiffel indeed enables and encourages programmers to express formal properties of classes by writing **assertions**, which may in particular appear in the following positions:

- routine **preconditions** express conditions that must be satisfied whenever a routine is called. For example withdrawal might only be permitted if it keeps the account's balance on or above the minimum. Preconditions are introduced by the keyword **require**.
- Routine **postconditions**, introduced by the keyword **ensure**, express conditions that are guaranteed to be true on routine return (if the precondition was satisfied on entry).
- Class **invariants** must be satisfied by objects of the class at all times, or more precisely after object creation and after any call to a routine of the class. They are described in the **invariant** clause of the class and represent general consistency constraints that are imposed on all routines of the class.

Our *ACCOUNT* class may be rewritten with appropriate assertions:

```
class ACCOUNT export .... (as before)
feature
  -- Attributes as before:
  --balance, minimum_balance, owner

  assign_owner ..... -- as before ;

  add ..... -- as before ;

  deposit (sum: INTEGER) is
      -- Deposit sum into the account
    require
      sum >= 0
    do
      add (sum)
    ensure
      balance = old balance + sum
    end ; -- deposit

  withdraw (sum: INTEGER) is
      -- Withdraw sum from the account
    require
      sum >= 0 ;
      sum <= balance - minimum_balance
    do
      add (-sum)
    ensure
      balance = old balance - sum
    end ; -- withdraw

  may_withdraw ..... -- as before

  Create (initial: INTEGER) is
    require
      initial >= minimum_balance
    do
      balance := initial
    end -- Create

invariant
  balance >= minimum_balance
end -- class ACCOUNT
```

The **old**... notation may be used in an **ensure** clause, with a self-explanatory meaning.

The reader will have noted that a *Create* procedure has been added to the features of the class. The reason is the following. Under the previous scheme, an account was created by, say, *acc1.Create*. Because of the initialization rules, *balance* is then zero and the invariant is violated. If a different initialization is required or, as here, an initialization depending on a parameter supplied by the client, then the class should include a procedure called *Create*. Object creation will be obtained by writing, say

```
acc1.Create (5500)
```

whose effect is to allocate the object (as in the default *Create*) and then to execute the procedure called *Create* in the class, with the given actual parameter. This example call is correct as the precondition is satisfied and the invariant will hold after the call.

Note that procedure *Create*, when explicitly provided, is recognized by the compiler as special; it is automatically exported and should not be included in the **export** clause.

Assertions, as they appear in preconditions, postconditions and invariants, should primarily be viewed as powerful tools for documenting correctness arguments: they serve to make explicit the assumptions on which programmers rely when they write program fragments that they believe are correct. In this respect, assertions are formalized comments. For debugging purposes, the Eiffel environment also makes it possible to monitor assertions at run-time, producing a message if an assertion is found to be violated.

Syntactically, expressions are boolean expressions, with a few extensions (like the **old** notation). The semicolon (see the precondition to *withdraw*) is equivalent to an "and", but permits individual identification of the components, useful for producing informative error messages at run-time.

It must be pointed out that assertions are **not** a technique for exception handling. An exception (in the CLU or Ada sense) is a run-time situation which the programmer prefers to handle in code separate from the main flow of control, but which (for the very reason that the program is prepared to deal with it) falls within the scope of the specification. In contrast, an assertion denotes a condition that should always be satisfied at the given control point; for example, a precondition describes the requirements for a routine body to be applicable. Assertion violation reflects a programming error, not an exceptional but planned situation from which it is possible to recover gracefully.

Eiffel offers no specific mechanism for exception handling. In our experience, standard algorithmic techniques are appropriate for dealing with exceptional conditions. Although we accept differing views on the question, we considered this facility to be of secondary importance and preferred to keep it out of the language in the interest of design simplicity. Encouraging programmers to express correctness arguments precisely through well-placed assertions appeared much more relevant.

6 - Generic classes

Building software components (classes) as implementations of abstract data types yields systems with a solid architecture but does not in itself suffice to ensure reusability and extendibility. This section and the next two describe Eiffel techniques for making the components as general and flexible as possible.

The first technique is genericity, which exists under different forms in Ada and CLU but is fairly new to object-oriented languages. Classes may be declared with generic parameters representing types. For example, the following classes are part of the basic Eiffel library:

ARRAY [*T*]
LIST [*T*]
LINKED_LIST [*T*]

They respectively describe one-dimensional arrays, general lists (without commitment as to a specific representation) and lists in linked representation. All have a formal generic parameter *T* representing an arbitrary type. To actually use these classes, one provides actual generic parameters, which may be either simple or class types, as in the following declarations:

il: LIST [*INTEGER*]*;*
aa: ARRAY [*ACCOUNT*]*;*
aal: LIST [[*ARRAY* [*ACCOUNT*]] --etc.

An earlier article [2] discussed the rôle of genericity in comparison to inheritance and explained their combination in Eiffel, especially in the context of strict type checking. The solution involves the notion of "declaration by association", not addressed here.

7 - Multiple inheritance

Multiple inheritance is a key technique for reusability. The basic idea is simple: when defining a new class, it is often fruitful to introduce it by combination and specialization of existing classes rather than as a new entity defined from scratch.

A simple example is provided by the classes of the basic library mentioned above. *LIST*, as indicated, describes lists of any representation. One possible representation for lists with a fixed number of elements uses an array. Such a class will be defined by combination of *LIST* and *ARRAY*, as follows:

```
class FIXED_LIST [T] export ....
inherit
    LIST [T];
    ARRAY [T]
feature
    ... Specific features of fixed-size lists ...
end -- class FIXED_LIST
```

The **inherit**... clause lists all the "parents" of the new class (which is said to be their "heir"). Thanks to this clause, all the features and properties of lists and arrays are applicable to fixed lists as well. This simple idea makes it possible to achieve remarkable economies of programming, of which we feel the effects daily in our experience of Eiffel software development. (Inheritance was introduced by Simula 67. In Simula, however, as in many other object-oriented languages, inheritance is not multiple: classes may have at most one parent.)

The very power of the mechanism demands adequate means to control its effects. In Eiffel, no name conflict is permitted between inherited features. Since such conflicts will inevitably arise in practice, especially in the case of software components brought in by independent developers, the language provides a technique to remove them: explicit **renaming**, as in

```
class C export... inherit
    A rename x as x1, y as y1;
    B rename x as x2, y as y2
feature...
```

Here the **inherit** clause would be illegal without renaming, since the example assumes that both *A* and *B* have features named *x* and *y*.

Renaming also serves to provide more appropriate feature names in heirs. In another example from the basic library, class *TREE* [*T*] is defined by inheritance from *LINKED_LIST* [*T*] and *LINKABLE* [*T*], the latter describing elements of linked lists. The idea is that a tree is a list (as it has subtrees, to which the usual list operations of insertion, change, deletion, traversal etc. apply), as well as a list element, as it may itself be inserted as subtree into another tree. (The class definition thus obtained is simple and compact, while covering a rich set of tree manipulation primitives which have proved sufficient in such different contexts as a multiple-windowing screen handler and a syntax-directed editor.) In the inheritance clause, the feature *empty* of linked lists, a boolean-valued function which determines whether a list is empty, itself inherited from *LIST*, is renamed *is_leaf* to conform to standard tree terminology.

To further ensure that the multiple inheritance mechanism is not misused, the invariants of all parent classes automatically apply to a newly defined class. Thus classes may not be combined if their invariants are incompatible.

8 - Polymorphism

One important aspect of inheritance is that it enables the definition of flexible program entities that may refer to objects of various forms at run-time (hence the name "polymorphic").

This possibility is one of the distinguishing features of object-oriented languages. In Eiffel, it is reconciled with static typing. The underlying language convention is simple: an assignment of the form $a := b$ is permitted not only if a and b are of the same type, but more generally if a and b are of class types A and B such that B is a descendant of A, where the descendants of a class include itself, its heirs, the heirs of its heirs etc. (The inverse notion is "ancestor".)

This corresponds to the intuitive idea that a value of a more specialized type may be assigned to an entity of a less specialized type – but not the reverse. (As an analogy, consider the fact that if I request vegetables, getting green vegetables is fine, but if I ask for green vegetables, receiving a dish specified as just "vegetables" is not acceptable, as it could include, say, cauliflower.)

What makes this possibility particularly powerful is the complementary facility: **feature redefinition.** A feature of a class may be redefined in any descendant class; the type of the redefined feature (if an attribute or a function) may be redefined as a descendant type of the original feature, and, in the case of a routine, its body may also be replaced by a new one.

Assume for example that a class *POLYGON*, describing polygons, has among its features an array of points representing the vertices and a function *perimeter* returning a real result, the perimeter of the current polygon, obtained by summing the successive distances between vertices. An heir of *POLYGON* may be:

```
class RECTANGLE export ... inherit
  POLYGON redefine perimeter
feature
  -- Specific features of rectangles, such as:
  side1: REAL; side2: REAL;

  perimeter: REAL is
      -- Rectangle-specific version
    do
      Result := 2 * (side1 + side2)
    end; -- perimeter
  ... other RECTANGLE features ...
```

Here it is appropriate to redefine *perimeter* for rectangles as there is a simpler and more efficient algorithm. Note the explicit **redefine** subclause (which would come after the **rename** if present).

Other descendants of *POLYGON* may also have their own redefinitions of *perimeter*. The version to use in any call is determined by the run-time form of the parameter. Consider the following class fragment:

```
p: POLYGON; r: RECTANGLE;
........ p.Create; r.Create; ........
if c then p := r end;
print (p.perimeter)
```

The assignment *p := r* is valid because of the above rule. If condition *c* is false, *p* will refer to an object of type *POLYGON* when *p.perimeter* is evaluated, so the polygon algorithm will be used; in the opposite case, however, *p* will dynamically refer to a rectangle, so that the redefined version of the feature will be applied.

This technique provides a high degree of flexibility and generality. The remarkable advantage for clients is the ability to request an operation (here the computation of a figure's perimeter) without knowing what version of the operation will be selected; the selection only occurs at run-time. This is crucial in large systems, where many variants of operations may be available, and each component of the system needs to be protected against variant changes in other components.

There is no equivalent to this possibility in non-object-oriented languages. Note for example that discrimination on records with variants, as permitted by Pascal or Ada, is of a much more restrictive nature, as the list of variants of a record type is fixed: any extension may invalidate existing code. In contrast, inheritance is open and incremental: an existing class may always be given a new heir (with new and/or redefined features) without itself being changed. This is crucial in the development of practical software systems which (whether by design or circumstance) is invariably incremental.

Neither do the generic and overloading facilities of Ada offer the kind of polymorphism shown here, as they do not support a programming style in which a client module may issue a request meaning: "compute the perimeter of *p*, using the algorithm appropriate for whatever form *p* happens to have when the request is executed".

This mechanism is more disciplined in Eiffel than in most other object-oriented languages. First, feature redefinition, as seen above, is explicit. Second, because the language is typed, the compiler may always check statically whether a feature application *a.f* is correct; in contrast, languages such as Smalltalk and its descendants (such as Objective-C) defer checks until run-time and hope for the best: if an object "sends a message" to another (Smalltalk terminology for calling a routine), one just expects that the class of the receiving object, or one of its ancestor classes, will happen to include an appropriate "method" (Smalltalk term for routine); if not, a run-time error will occur. Such errors may not happen during the execution of a correctly compiled Eiffel system.

A further disadvantage of the Smalltalk approach is that it may imply costly run-time searches, as a requested method may not be defined in the class of the receiving object but inherited from a possibly remote ancestor. What, on the other hand, may be said in favor of the interpretive, dynamic Smalltalk approach is that it makes it easy, in the Lisp tradition, to modify the software while it is running, for example by providing a missing method; such facilities, useful for exploratory programming, are harder to provide in a compiler-oriented system which also includes correctness, reliability and efficiency among its design goals.

Another tool for controlling the power of the redefinition mechanism is provided in Eiffel by assertions. If no precautions are taken, redefinition may be dangerous: how can a client be sure that evaluation of *p.perimeter* will not in some cases return, say, the area? One way to maintain the semantic consistency of routines throughout their redefinitions is to use preconditions and postconditions, which are binding on redefinitions. More precisely, any redefined version must satisfy a weaker or equal precondition and ensure a stronger or equal postcondition than in the original. Thus, by making the semantic constraints explicit, routine writers may limit the amount of freedom granted to eventual redefiners.

This concludes the overview of the language. As it is well known that a programming language is in practice no better than its implementation, we conclude with a brief description of the Eiffel compiler and associated environment.

9 - The implementation

The current implementation of Eiffel runs on various versions of Unix (System V, 4.2BSD, Xenix). 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.

Note that (as the above discussion should suffice to show) 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.

Great care has been taken to provide efficient compilation and execution, so that the environment would support the development of serious software. The following points are particularly worth noting.

- As was seen above, the potential for redefinition implies that a qualified routine reference, say *p.perimeter*, may have many different interpretations depending on the value of *p* at run-time. As mentioned, this powerful facility may result in serious inefficiencies if the search for the appropriate routine is made at run-time, as seems to be necessary. The maximum length of such a search grows with the depth of the inheritance graph, putting reusability (which tends to promote the addition of new levels of inheritance and feature redefinition) in direct conflict with efficient performance. Note that the situation becomes hopeless with multiple inheritance, as not only a linear list but a complete graph of ancestor classes must be searched at run-time.

  The Eiffel implementation always finds the appropriate routine in constant time; the space overhead associated with this technique is negligible. We found this result rather difficult to achieve in the presence of multiple inheritance – but essential in light of the previous discussion. Whether you have one or one million routines in your system, *a.x* always takes the same time.

- There is almost never any duplication of code. Again this was difficult to achieve with multiple inheritance and genericity: the implementation of multiple inheritance included in recent versions of Smalltalk [14] duplicates codes for parent classes other than the first, precisely to avoid the graph traversal mentioned above; most implementations of generic packages in Ada systems also duplicate code for various generic instances of a program unit. The only case in which we do duplicate code is a rather rare and special occurrence, "repeated" inheritance with feature duplication, not described here.

- Memory management is handled by a run-time system that takes care of object creation and (system-controlled) de-allocation. Optionally, the memory management system includes its own virtual memory subsystem, which may be turned off if the facilities of the underlying operating system are sufficient. Garbage collection is also provided; it is performed incrementally by a parallel process which steals as little time as it can from application programs [15]. (At the moment garbage collection is parallel on the Unix System V version only.)

- 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 usually (in our experience) between 50% and 100% of the time for the next step, translation from C to machine code.

One more property of the implementation deserves a mention: its openness. 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. For example, an Eiffel routine for computing square roots might rely on an external function, as follows:

```
sqrt (x: REAL, eps: REAL): REAL is
      -- Square root of x with precision eps
   require
      x >= 0 ; eps > 0
   external
      csqrt (x: REAL, eps: REAL): REAL
         name "sqrt" language "C"
   do
      Result := csqrt (x, eps)
   ensure
      abs (Result ^ 2 - x) <= eps
   end -- sqrt
```

The optional name... subclause caters to the various naming conventions of other programming languages.

This mechanism makes it possible to use external routines without impacting the conceptual consistency of the Eiffel classes. Note in particular how the C function *sqrt* is granted a more dignified status as Eiffel routine with the addition of a precondition and a postcondition.

This facility is essential in view of the design objectives listed in section 2: an environment promoting reusability should enable reuse of software written prior to its introduction, not just of future software to be developed with it. Beyond this remark, the "external" construct lies at the basis of one of the applications of Eiffel: as an integrating mechanism for components written in other languages. An example might be scientific software: although numerical programs will benefit just as much as others from such structuring mechanisms as classes, multiple inheritance, genericity, export controls, assertions etc., programmers may prefer to code the actual bodies of numerical routines in a language specifically designed for this purpose, and package them cleanly in Eiffel classes.

This application is one of the three main intended uses of Eiffel. The second is the most obvious, namely as a tool covering the whole design and implementation process; this is how we apply Eiffel at Interactive Software Engineering. The third may appeal to users who, because of external requirements, must produce final programs in some other language (such as Ada); it consists in employing Eiffel as an object-oriented PDL (Program Design Language) covering the part of the process extending from global to detailed design, and translating the result into the required implementation language. Eiffel may even be used earlier, at the specification stage, thanks in particular to a facility not described in this article, deferred routines. (A deferred routine has no body: implementations must be provided in descendants of its class; semantics may, however, be specified by a precondition and a postcondition.)

10 - 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. However the solution to this problem 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.

The environment also contains debugging tools: tools for run-time checking of assertions; a tracer and symbolic debugger; a viewer for 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.

A postprocessor, **ep**, 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, "obscuring" of the C code (by default, the generated C code is fairly readable, and it is not too difficult to find the correspondence with the original Eiffel; "obscured" code will make it very hard to understand the algorithms). The last two options are particularly interesting for software implementors who use Eiffel for the development of packages that will be delivered to their customers in standard C form.

Finally the environment includes a set of basic classes covering some of the most important data structure implementations. Our goal is to extend this library so as to cover the most common patterns of everyday programming.

11 - Conclusion

A number of individual concepts embodied in Eiffel were present in previous languages; our most important conscious debts are (in order of decreasing influence) to Simula 67, Ada and Alphard. Among the new contributions are, from the language standpoint, 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 multiple inheritance, a clean interface with external routines, and the introduction of full static typing into an object-oriented language. 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 documentation, support for the preparation of deliverable software packages.

More generally, we think that Eiffel is the first language to combine the powerful ideas of object-oriented languages with the modern concepts of software engineering; these results have been made available to practicing software developers in an environment offering the facilities that are required to develop serious software.

References

1. Bertrand Meyer, *Eiffel: a Language for Software Engineering,* Technical Report TRCS85-19, University of California, Santa Barbara, Computer Science Department, November 1985 (Revised, August 1986).
2. Bertrand Meyer, "Genericity versus inheritance," in *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 391-405, Portland (Oregon), September 29 - October 2, 1986.
3. Bertrand Meyer, "Software Reusability: the Case for Object-Oriented Design," Report TR-86-06, Interactive Software Engineering, Santa Barbara (California), 1986.
4. James McCall, (Ed.) *Factors in Software Quality,* General Electric, 1977.
5. Graham Birtwistle, Ole-Johan Dahl, Bjorn Myrhaug, and Kristen Nygaard, *Simula Begin,* Studentliteratur and Auerbach Publishers, 1973.
6. Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, Reading (Massachusets), 1983.
7. Bjarne Stroustrup, *The C++ Programming Language,* Addison-Wesley, Menlo Park (California), 1986.
8. Brad J. Cox, *Object-Oriented Programming: An Evolutionary Approach,* Addison-Wesley, Reading (Massachusetts), 1986.
9. Larry Tesler, "Object Pascal Report," *Structured Language World*, vol. 9, no. 3, 1985.
10. Daniel G. Bobrow and M.J. Stefik, *LOOPS: an Object-Oriented Programming System for Interlisp,* Xerox PARC, 1982.
11. H.I. Cannon, "Flavors," Technical Report, MIT Artificial Intelligence Laboratory, Cambridge (Massachussets), 1980.
12. Jean-Marie Hullot, "Ceyx, Version 15: I - une Initiation," Rapport Technique no. 44, INRIA, Rocquencourt, Eté 1984.
13. Bertrand Meyer, "Cépage: Towards Computer-Aided Design of Software," *Computer Language*, vol. 3, no. 9, pp. 43-53, September 1986.
14. Alan H. Borning and Daniel H. H. Ingalls, "Multiple Inheritance in Smalltalk-80," in *Proceedings of AAAI-82*, pp. 234-237, 1982.
15. Edsger W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Communications of the ACM*, vol. 21, no. 11, pp. 966-975, November 1978.