Eiffel: Applying the principles of object-oriented design

By Bertrand Meyer

Eiffel is a language based on the principles of object-oriented design, augmented by features enhancing correctness, extendability, and efficiency. Currently available on UNIX, Eiffel includes a library of reusable classes and tools for tasks such as automatic configuration management, documentation, and debugging.

Software quality involves trade-offs among many factors.[1] Reusability, or the ability to produce software components that may be used in many different applications, is one of the more important factors. Many of the same program patterns frequently recur, but actual reuse of program modules is much less widespread than it ought to be. Another important factor is extendability: although software is supposed to be soft, it is notoriously hard to modify software systems, especially large ones.

Reusability and extendability allow for less software to be written. Thus, presumably, more time may be devoted to other goals—efficiency, ease of use, etc. In the long run, it may pay off to concentrate on these two factors; they were indeed paramount in the design of Eiffel.

Other design goals that played a significant part in developing Eiffel include software correctness and robustness, portability, and efficiency in the context of practical, medium-to-large-scale developments.

To achieve reusability and extendability, the principles of object-oriented design provide the best-known technical approach. 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 classes of objects it manipulates rather than the functions it performs.
- Objects are described as instances of abstract data types—data structures known from an interface rather than through their representation.
- Basic modular units, called classes, describe implementations of abstract data types.
- "Collection" reflects how classes should be designed: as units that are interesting and useful on their own, independent of the systems to which they belong, and may be reused by many different systems. Software construction is viewed as the assembly of existing classes, not as a top-down process starting from scratch.
- "Structured" reflects the existence of important relationships between classes, particularly the multiple inheritance relation.

A class represents an implementation of an abstract data type: a set of run-time objects characterized by the operations available on it (which are the same for all instances of a given class) and the properties of these operations. These objects are called the class's instances. Classes and objects should not be confused: classes are a compile-time notion, whereas objects exist only at run time. This difference is similar to the difference in classical programming between a program and one execution of that program.

A simple example of a class is *ACCOUNT*, which describes bank accounts. Before showing the class, we describe how it would be used by another class, such as *X*, called a client. To use *ACCOUNT*, *X* may introduce an entity and declare it of this type:

```
acc1: ACCOUNT
```

The term "entity" is preferred to "variable" as it denotes a more general notion. An entity declared of a class type, such as *acc1*, may at any time during execution refer to an object (Figure 1); since Eiffel is a typed language, this object must be an instance of *ACCOUNT* or (as I will discuss later) a descendant class of *ACCOUNT*. An entity that does not refer to any object is void. By default (at initialization) entities are void; objects must be created explicitly by an instruction:

```
acc1.Create
```

This instruction associates *acc1* with the newly created object. *Create* is a predefined feature of the language.

Once *acc1* has been associated with an object, the features defined in *ACCOUNT* may be applied to it, as in:

```
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, as in *entity_name.feature_name*. The two kinds of features include routines or operations, such as *open*, *deposit*, *may_withdraw*, or *withdraw*, and attributes—data items associated with objects of the class.

Routines are further divided into procedures and functions; only functions return a result. In the previous example, *may_withdraw* is a function with an integer parameter, returning a Boolean result; the other three routines invoked are procedures.

The example of class *X* does not show whether in *ACCOUNT balance* is an attribute or a function without parameters. This ambiguity is intentional. A class such as *X*, a client of *ACCOUNT*, does not need to know how a balance is obtained: it could be stored as an attribute of every account object or recomputed by a function from other attributes such as the list of previous deposits and withdrawals. The choice between these representations is internal to *ACCOUNT* but irrelevant to clients.

The following is a first sketch of how *ACCOUNT* itself might look; line segments beginning with two dashes (--) are comments:

Entity and associated object

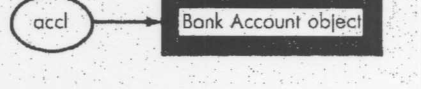


Figure 1.

COMBINE THE RAW POWER OF FORTH WITH THE CONVENIENCE OF CONVENTIONAL LANGUAGES

HS/FORTH

Yes, Forth gives you total control of your computer, but only HS/FORTH gives you implemented functionality so you aren't left hanging with "great possibilities" (and lots of work!) With over 1500 functions you are almost done before you start!

WELCOME TO HS/FORTH, where megabyte programs compile at 10,000 lines per minute, and execute faster than ones built in 64k limited systems. Then use AUTOOPT to reach within a few percent of full assembler performance — not a native code compiler linking only simple code primitives, but a full recursive descent optimizer with almost all of HS/FORTH as a useable resource. Both optimizer and assembler can create independent programs as well as code primitives. The metacompiler creates threaded systems from a few hundred bytes to as large as required, and can produce ANY threading scheme. And the entire system can be saved, sealed, or turnkeyed for distribution either on disk or in ROM (with or without BIOS).

HS/FORTH is a first class application development and implementation system. You can exploit all of DOS (commands, functions, even shelled programs) and link to .OBJ and .LIB files meant for other languages *interactively!*

I/O is easier than in Pascal or Basic, but much more powerful — whether you need parsing, formatting, or random access. Send display output through DOS, BIOS, or direct to video memory. Windows organize both text and graphics display, and greatly enhance both time slice and round robin multitasking utilities. Math facilities include both software and hardware floating point plus an 18 digit integer (finance) extension and fast arrays for all data types. Hardware floating point covers the full range of trig, hyper and transcendental math including complex.

Undeniably the most flexible & complete Forth system available, at any price, with no expensive extras to buy later. Compiles 79 & 83 standard programs. Distribute metacompiled tools, or turnkeyed applications royalty free.

HS/FORTH (complete system):	$395.
HS/FORTH: Tutorial & Ref (500 pg)	$ 59.
Forth: Text & Reference (500 pg)	$ 22.
HS/FORTH Glossary	$ 10.
GIGAFORTH Option (Beta release) (Native Mode from SOFTMILLS, INC.)	$245.

Visa Mastercard

HARVARD SOFTWORKS

PO BOX 69
SPRINGBORO, OH 45066
(513) 748-0390

CIRCLE 47 ON READER SERVICE CARD

```
class ACCOUNT export
  open, deposit, may_withdraw,
  withdraw, balance, owner
feature
  balance: INTEGER ;
  minimum_balance: INTEGER is 1000 ;
  owner: STRING ;
  open (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 lists the names of features available to clients of the class. Nonexported features are considered secret. Here procedure *add* is secret, so *acc1.add (−3000)* would be illegal in *X*. Attribute *minimum _balance* is also secret.

Classes are defined as abstract data type implementations. What defines an abstract data type, however, is not just the available operations but also the formal properties of these operations, which do not appear in the previous example.

Eiffel enables programmers to express formal properties of classes by writing assertions, which may 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 guaranteed to be true on routine return.
- Class invariants must be satisfied by instances of a class at all times, or, more precisely, after object creation and after any call to the class's routine. They are described in the *invariant* clause of a class and represent general consistency constraints imposed on all routines of a class.

Assertions in Eiffel reflect the notion of correctness already mentioned. Techniques for producing extendable and reusable components are of little interest unless we convince ourselves these components are also correct and robust.

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

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

Syntactically, assertions are Boolean expressions with a few extensions, such as the *old* notation. The semicolon (see the precondition to *withdraw*) is equivalent to *and* but permits individual identification of the components, which is useful for producing informative error messages when assertions are checked at run time.

Assertions may indeed be monitored at run time; since such monitoring may penalize the performance, it is enabled on option, class by class. (For each class, two levels of monitoring are possible: preconditions only or all assertions). The classes of the basic Eiffel library, widely used in Eiffel programming, are protected by carefully written assertions. A violated assertion will trigger an exception. Unless the programmer has written an appropriate exception handler, the exception will cause an error message and termination.

Independently of any run-time checking, however, assertions are powerful tools for documenting correctness arguments: they make explicit the assumptions on which programmers rely when they write program fragments they believe are correct. Writing assertions, especially preconditions and postconditions, amounts to spelling out the terms of a contract that controls the relationship between a routine and its callers. The precondition binds the callers, the postcondition binds the routine. This metaphor of programming as contracting is a general and fruitful paradigm.[2]

Using contracts involves the risk that they may be broken, which is where exceptions are needed.[3,4] An exception may arise from one of several causes. When assertions are monitored, an assertion violation will raise an exception. Another cause is the occurrence of a hardware-triggered abnormal signal, which might arise from arithmetic overflow or a failure to find the memory needed for allocating an object.

Unless a routine has made specific provisions to handle exceptions, it will fail if an exception arises during its execution. A routine that fails triggers an exception in its caller. However, a routine may handle an exception through a rescue clause. This optional clause attempts to patch things up by bringing the current object to a stable state (one satisfying the class invariant). Then it can terminate in either of two ways:

- The rescue clause may execute a *retry* instruction. This will cause the routine to restart its execution from the beginning, attempting again to fulfill the routine's contract, usually through another strategy. *Retry* assumes the instructions of the rescue clause—before the *retry*—have attempted to correct the cause of the exception.
- If the rescue clause does not end with *retry*, the routine fails: it returns to its caller, immediately signaling an exception. (The caller's rescue clause will be executed according to the same rules.)

Note that a routine with no rescue clause is considered to have an empty rescue clause, so any exception occurring during the routine's execution will cause the routine to fail immediately. The principle underlying this approach is that a routine must either succeed or fail. If it fails, it must notify its caller by triggering an exception.

The Advanced Programmer's Editor That Doesn't Waste Your Time

For DOS, Microport UNIX, SCO Xenix or **OS/2** Protected Mode

EPSILON

- Fast, EMACS-style commands—completely reconfigurable
- Run other programs without stopping Epsilon—concurrently!
- C Language support—fix errors while your compiler runs
- Powerful extension language
- Multiple windows, files
- Unlimited file size, line length
- 30 day money-back guarantee
- Great on-line help system
- Regular Expression search
- Supports large displays
- Not copy protected

Only $195

Lugaru Software Ltd.
5843 Forbes Avenue
Pittsburgh, PA 15217

Call (412) 421-5911
for IBM PC/XT/AT's or compatibles

CIRCLE 48 ON READER SERVICE CARD

LAHEY SETS NEW FORTRAN STANDARDS

LAHEY PERSONAL FORTRAN 77 **$95**
Low cost, Full 77 Standard, Debugger, Fast Compilation

F77L FORTRAN LANGUAGE SYSTEM **$477**
For porting or developing, this is the critics' choice.
"Editor's Choice" .... ***PC Magazine***
". . .the most robust compiler tested." .... ***Micro/Systems***
". . .the most efficient and productive FORTRAN development tool for the DOS environment" .... ***BYTE***

F77L-EM/32-bit $895 **F77L-EM/16-bit $695**
Break through the DOS 640K barrier. The most powerful PC Fortran Language Systems for downloading or writing large programs.

PRODUCTIVITY TOOLS
Profiler, ULI Mathematical Functions Library, Overlay Linker, Toolkit, Utility Libraries, Windows, Memory Boards, 80386 HummingBoard.

IF YOU DEMAND THE VERY BEST, THEN YOU SHOULD BE USING LAHEY. CALL US TO DISCUSS YOUR PC FORTRAN NEEDS.

CALL FOR NEW FEATURES INCLUDING MATH COPROCESSOR EMULATION

30 DAY MONEY-BACK GUARANTEE

FOR INFORMATION OR TO ORDER:
1-800-548-4778
Lahey Computer Systems, Inc.
P.O. Box 6091, Incline Village, NV 89450
TEL: 702-831-2500 TLX: 9102401256

Lahey Computer Systems Inc.

CIRCLE 49 ON READER SERVICE CARD

Building software components or classes as implementations of abstract data types yields systems with a solid architecture but does not ensure reusability and extendability. However, certain Eiffel techniques help make components as general and flexible as possible.

The first such technique is genericity, which exists in different form in languages such as Ada but is new to object-oriented languages. Classes may have generic parameters representing types. The following examples come from the basic Eiffel library:

```
ARRAY [T]
LIST [T]
LINKED_LIST [T]
```

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

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

Another key reusability technique is multiple inheritance. The basic idea is simple: define a new class by combining and refining existing classes, rather than as a new entity defined from scratch.

The following is a typical example of multiple inheritance from the basic Eiffel library. *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 the parents of the new class, which is their heir. (The ancestors of a class include the class itself, its parents, grandparents, etc. The opposite term is "descendant.") Declaring *FIXED_LIST* as shown ensures all the features and properties of lists and arrays are applicable to fixed lists as well.

Another example of multiple inheritance is extracted from a windowing system based on a class *WINDOW*. Windows have graphical features—height, width, position, etc.—with associated routines to scale them, move them, and so on. The system permits windows to be nested, allowing for hierarchical features: access to subwindows and the parent window, adding a subwindow, deleting a subwindow, attaching to another parent, and so on. Rather than writing a complex class that would contain specific implementations for all these features, it is preferable to inherit all hierarchical features from *TREE* (one of the classes in the basic Eiffel library describing tree implementations) and all graphical features from a class *RECTANGLE*.

Multiple inheritance raises the possibility of name clashes. This problem is solved in Eiffel by a *rename* construct, which is also useful to provide locally well-adapted names for inherited features.

An 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, a phenomenon called polymorphism. This capability is one of the distinctive 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*.

This convention 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 ordering vegetables: asking for green vegetables and receiving a dish labeled "vegetables" is not acceptable as it could include carrots or another nongreen vegetable.)

What makes polymorphism particularly powerful is two complementary facilities: redefinition and dynamic binding. A class's feature 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 the 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...
```

It is appropriate to redefine *perimeter* for rectangles since a simpler and more efficient algorithm exists.

Other descendants of *POLYGON* may also have their own redefinitions of *perimeter*. Dynamic binding means that 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 rule in the previous example. 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 the redefined version of the feature will be applied. This capability is known as dynamic binding.

Dynamic binding provides a high degree of flexibility and generality. Its advantages include 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 occurs only at run time. This is essential in large systems, where many variants of operations may be available and each component of the system should be protected against variant changes in other components.

Assertions are another tool in Eiffel for controlling the power of the redefinition mechanism. If no precautions are taken, redefinition may be dangerous: how can a user be sure evaluation of *p.perimeter* will not in some cases return the area, for instance?

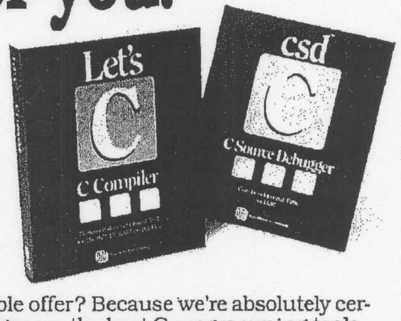
One way to maintain the semantic consistency of routines throughout their redefinitions is to use preconditions and postconditions, which are binding on redefinitions. More precisely, any redefined version must satisfy a weaker or equal precondition and ensure a stronger or equal postcondition than in the original. Thus, by making the semantic constraints explicit, routine writers may limit the amount of freedom granted to eventual redefiners.

These rules should be understood in light of the contracting metaphor described earlier. Redefinition and dynamic binding introduce subcontracting: for example, *POLYGON* subcontracts the implementation of *perimeter* to *RECTANGLE* when applied to any entity that refers at run-time to a rectangle object. An honest subcontractor is bound by the contract accepted by the prime contractor: it may not impose stronger requirements on the clients but may accept more general requests; hence the possibility for the precondition to be weaker. Also, it must achieve at least as much as promised by the original contractor, but may achieve more; hence the possibility for the postcondition to be stronger.

An important extension of the inheritance mechanism is provided by deferred classes. A deferred class is a class containing at least one deferred routine; a routine is declared as deferred to express that implementations of the routines will be provided only in descendants. For example, a system used by the Department of Motor Vehicles to register vehicles could include a class of the form:

```
deferred class VEHICLE export
  dues_paid, valid_plate, register,...
feature
  dues_paid (year:INTEGER):
    BOOLEAN is...end;
```

If you think you can buy a better C compiler, don't. We'll buy it for you.



Buy Let's C® with csd™ for just $75. If you're not satisfied, get Turbo C or QuickC. Free.*

Why are we making this incredible offer? Because we're absolutely certain Let's C and *csd* C Source Debugger are the best C programming tools you can own.

Rest assured that, like its competition, Let's C features incredibly fast in-memory compilation and produces extremely tight, high quality code. The differences lie in how much faster you can perform other programming chores.

Our debugger, for example, can cut development time in half. But that's not all:

"csd *is close to the ideal debugging environment... a definite aid to learning C and an indispensable tool for program development."*
—William G. Wong, BYTE

And comparatively speaking: *"No debugger is included in the Turbo C package... a serious shortcoming."*
—Michael Abrash, Programmer's Journal

Unlike our competition, Let's C *includes* its own full-featured assembler, features documentation with *complete* examples and technical support with complete answers—the *first* time you call.

LET'S C AND *csd* FEATURES

LET'S C:
- Now compiles twice as fast
- Integrated edit-compile cycle: editor automatically points to errors
- Includes both small and large memory model
- Integrated environment or command line interface
- 8087 sensing and support
- Full UNIX compatibility and complete libraries
- Many powerful utilities including make, assembler, archiver
- MicroEMACS full screen editor with source code included
- Supported by dozens of third party libraries

csd:
- Debug in C source code, not assembler
- Provides separate source, evaluation, program and history windows
- Ability to set trace points and monitor variables
- Can interactively evaluate any C expression
- Can execute any C function in your program
- Trace back function

So if you're thinking about buying any other C compiler, think again. But this offer is only available for a limited time. So think fast. And see your software dealer or call 1-800-MWC-1700 soon. (1-312-472-6659 in Illinois.)

To exchange Let's C and *csd* for Turbo C or QuickC, return registration card within 15 days of purchase date, notify Mark Williams Company that you are returning products and receive a return authorization number. Products must be returned within 30 days of purchase date or within 30 days of shipping date on telephone orders.

Mark Williams Company

1430 West Wrightwood, Chicago, Illinois 60614, 312-472-6659

© 1988 Mark Williams Company

CIRCLE 51 ON READER SERVICE CARD

```
valid_plate (year:INTEGER):
  BOOLEAN is...end;
register (year: INTEGER) is
    --Register vehicle for year
  require
    dues_paid (year)
  deferred
  ensure
    valid_plate (year)
  end;--register
 ...Other features...
end
```

This example assumes that no single registration algorithm is applicable to all types of vehicles; the exact procedure to follow depends on the type of vehicle considered: passenger car, motorcycle, truck, etc. However, the same precondition and postcondition are applicable to all types of vehicles. The solution is to treat *register* as a deferred routine, making *VEHICLE* a deferred class. Effective versions of this routine are given in descendants of class *VEHICLE*: for example *CAR*, *TRUCK*, etc.

A deferred class may not be instantiated: *v.Create* is illegal if *v* is an entity declared of type *VEHICLE*. But such an entity may be assigned a reference to an instance of a nondeferred descendant of *VEHICLE*.

For example, assuming *CAR* and *TRUCK* provide effective definitions for all deferred routines of *VEHICLE*, the following will be correct:

```
...
v:VEHICLE; c:CAR; t:TRUCK;
...
c.Create (...); t.Create (...);
...
if "some test" then v := c
  else v := end;
v.register (1988)
```

Deferred classes are particularly useful for applying Eiffel as a design language rather than just for implementation. The first version of a system may be given as a set of deferred classes whose purpose may be described by assertions. This is achieved with dynamic binding. Depending on the outcome of a test, the appropriate version or *register* will be used at run time.

Eiffel runs on UNIX System V, 4.2BSD, and XENIX, and has been ported to about 15 different architectures. The compiler uses C as an intermediate language, giving Eiffel the potential to be portable to any environment supporting C. (The task of making Eiffel portable to VAX/VMS is under way.)

The openness of Eiffel's implementation deserves mention. Eiffel's classes are meant to be interfaced with code written in other languages. This is reflected by the optional external clause that, in a routine declaration, lists external subprograms used by the routine. For example, a square root routine might rely on an external function:

```
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 languages.

The construction of systems in Eiffel is supported by a set of development tools. Most important are the facilities for automatic configuration management integrated in the compilation command *es* (Eiffel System). 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-

REALIA COBOL DRIVES THE HIGH-SPEED DESKTOP

Our competitors will tell you that performance is secondary—it's not. Or they'll want you to buy faster hardware—don't.

Thousands of programmers know how much difference Realia COBOL makes every day, in their own productivity and their users' satisfaction. Realia COBOL is not only the fastest-compiling COBOL for the PC or PS/2 (2 to 10 times faster than the other PC COBOLs). It also produces the tightest, fastest executable programs (4 to 70 times faster than the others).

Call for your copy of the Realia COBOL 3.00 Benchmark Report. Or ask for a free evaluation copy of the product—we'll also send you the benchmark program, so you can compare the compilers for yourself.

REALIA®

10 South Riverside Plaza, Chicago, IL 60606 • (312) 346-0642 • Telex 332979

CIRCLE 46 ON READER SERVICE CARD

compiles those whose compiled versions
class may be a client of one of its descendants) and, in the case of the client relation, may involve cycles. But Eiffel's solution frees programmers from having to keep track of changed modules to maintain the consistency of their systems. An algorithm avoids many unneeded recompilations by detecting modifications that do not impact class interfaces. In practice, this algorithm prevents a chain reaction of recompilations in a large system when a feature implementation is changed in a low-level class.

Eiffel's environment also contains debugging tools: tools for run-time assertion checking, a tracer and symbolic debugger, and a viewer for interactive exploration of the object structure at run time.

A documentation tool, short, produces a summary version of a class that shows the interface as available to clients: the exported features and, in the case of routines, the header, precondition, and postcondition. The manual for the basic Eiffel library contains Eiffel documentation produced almost entirely from output generated by short.[5] Such documentation is essentially obtained for free and, even more importantly, is guaranteed to be consistent with the documented software as the documentation is extracted from it. This should be contrasted with classical approaches, where software and documentation are viewed as separate products.

A postprocessor integrated in *es* performs various optional modifications on the generated C code: removing of unnecesary routines, simplifying of calls to nonpolymorphic routines, and in-line expansion of simple routines. One of its main options is the generation of a stand-alone C package from an Eiffel system. The package comes with a make file and a copy of the run-time system. It may be ported to any environment supporting C and some primitive system functions. This cross-development facility is particularly interesting for developers who use Eiffel to design and implement their software but deliver it to their customers in C form: Eiffel need not be available on the target environments.

The basic Eiffel library is a repertoire of classes covering many important data structures and algorithms. The library enables programmers to think and write in terms of lists, trees, stacks, hash tables, etc., rather than arrays, pointers, flags, and the like.

Recent Eiffel developments include a
through the network of existing classes. These tools rely on on a set of graphical library classes, based on the X Windows package from Massachusetts Institute of Technology (Cambridge, Mass.).

I believe Eiffel is the first language to combine the powerful ideas of object-oriented languages with the modern concepts of software engineering. These capabilities are available to software developers in an environment offering the facilities required to develop serious software. ▪

References
[illegible] and Inheritance," *The Journal of Pascal, Ada and Modula-2*, 1988 (to appear).
4. Meyer, Bertrand. *Object-Oriented Software Construction.* Englewood Cliffs, N.J.: Prentice-Hall, 1988.
5. Interactive Software Engineering Inc., *Eiffel User's Manual*, Technical Report TREI5/UM, 1986.

Bertrand Meyer is president of Interactive Software Engineering Inc., Santa Barbara, Calif., a company that produces and distributes CASE tools.

Now COBOL Programmers Can Do Formatted Screens Quickly and Easily - with SCREENIO.

Realia COBOL • Micro Focus COBOL • IBM COBOL/2

SCREENIO is a high-performance, COBOL-specific screen manager for the Personal Computer and compatibles. It's powerful, offers an impressive array of features, yet is easy to use. SCREENIO was written by experienced COBOL professionals and represents a truly COBOL approach to screen management on the PC.

Design and Revise Screens Painlessly with Our Panel Editor.

Use our interactive Panel Editor Facility to design your screens with a What-You-See-Is-What-You-Get approach. You can easily modify and experiment with screen layouts. Type titles and descriptive information on the screen, draw lines and boxes using the cursor movement keys, and paint the colors you want using the cursor keys or block functions. Use all 256 color combinations on your screens.

Programming with SCREENIO is Easy.

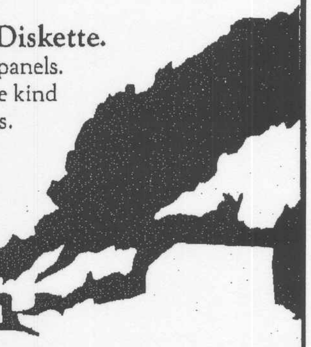
You can display your screen and accept data with as few as two statements; a COPY *panelname* in your WORKING-STORAGE SECTION, and a CALL SCREENIO statement in the PROCEDURE DIVISION. SCREENIO is linked with your application just as any other COBOL subroutine would be.

Yes, we've always done windows. SCREENIO supports Edit Masks, Automatic Error Detection and Editing, High Speed Video, Hot-Fields, Key Redefinition, Foreign Language Capability, Custom Tabbing Order, Color Control at Runtime, PC Speaker Control, and much, much more. A BMS Mapset Import is available. Most programmers are amazed at how easy it is to build flashy applications using SCREENIO and COBOL.

Our Support is Outstanding.

We provide superb telephone support. Upgrades are distributed to licensed users automatically for the first year. There are no Runtime Fees—the code you develop is yours to distribute freely.

We'll Send You a Free Demonstration Diskette.



This limited version of our Panel Editor shows how you design panels. Because it's written with COBOL and SCREENIO, it shows the kind of features and performance you can expect in your applications.

SCREENIO 2.1 is only $400 plus shipping. Ask about our other programming tools and package deals, too. In a hurry? Most orders are *delivered* within 24 hours!

NORCOM

Northern Computing Consultants
P.O. Box 020897 • Juneau, Alaska 99802
(907) 780-6464 • TELEX 510 601 4951 (Norcom)

CIRCLE 53 ON READER SERVICE CARD