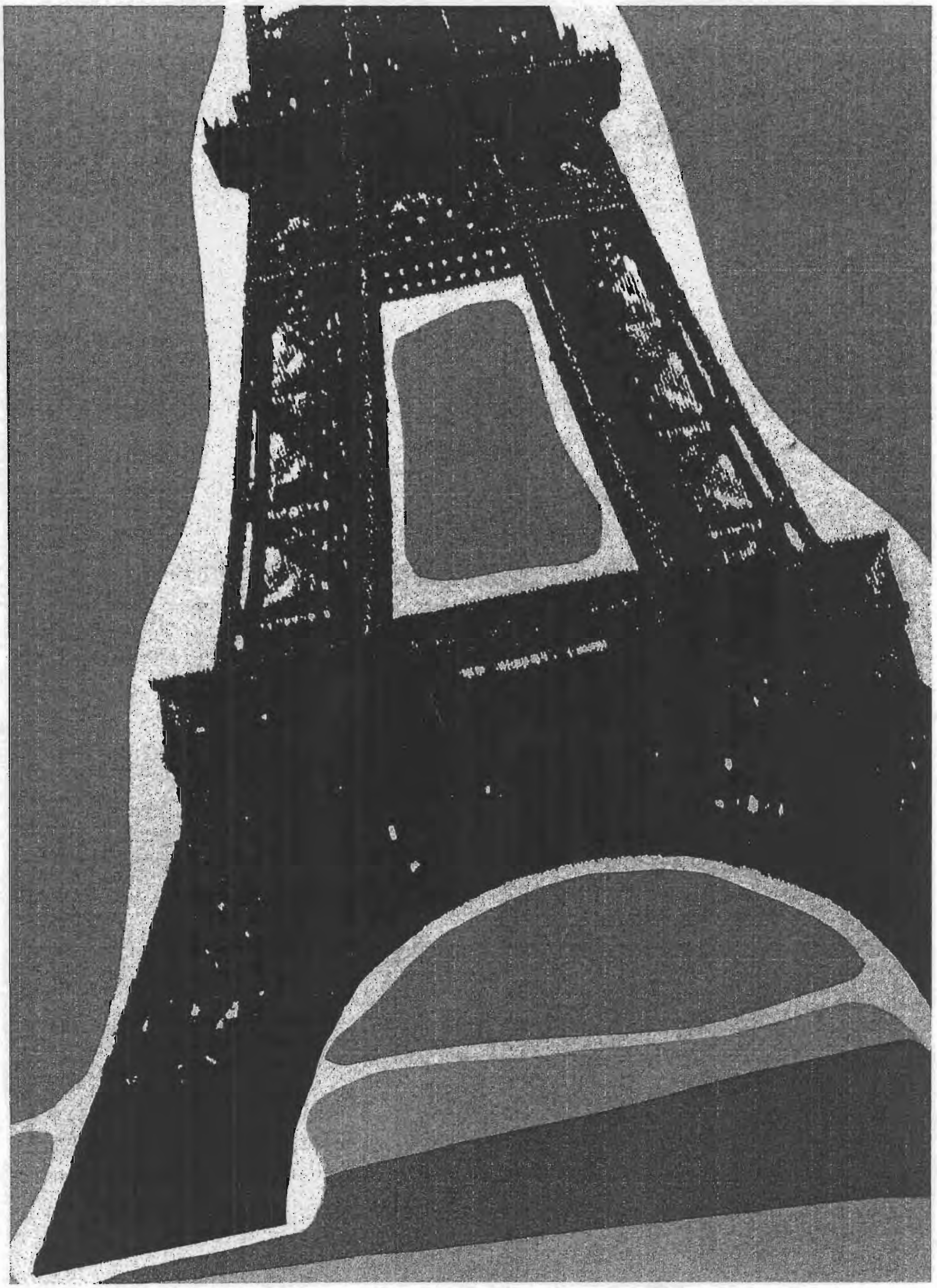


Stephen G. Luker

THE EIFFEL ENVIRONMENT

Because interpreters offer flexibility, most object-oriented environments have been interpreter-implemented. For software development purposes, however, efficiency and reliability are imperatives, and compilation is almost invariably a requirement. This account tells of an attempt to have the best of both of these worlds.

BY BERTRAND MEYER

Object-oriented programming has been hailed as an effective technique for experimenting and prototyping. But another view is also possible—a software engineering view, where efficiency and reliability count for just as much as reusability and extensibility. This article explores the implications of this perspective by reviewing issues that arose in the design of Eiffel [1], an object-oriented environment for the development of high-quality production software.

The traditional freestyle approach to object oriented environments, as exemplified by Smalltalk, is usually implemented by way of interpreters, which yield considerable advantages in terms of flexibility, accessibility to source code, debugging capabilities, and modification ease. By contrast, the goals of software engineering almost invariably require compiling.

But is this to say that a compiled approach necessarily implies an inflexible, hard-to-use environment? This article attempts to show it doesn't have to. To demonstrate this, I'll revisit some of the design choices we made while building the Eiffel environment in an attempt to reconcile some of the best aspects of both the compiler-oriented and the interpreter-implemented worlds. Of course, no compiled environment can hope to give the user the same degree of freedom and direct interaction possible with an interpreter. But certainly it can come close to providing as much comfort, and not sacrifice either reliability or efficiency in the process.

The Importance of Being Compiled. Reliability and efficiency, of course, are watchwords for those who favor compilers.

Added reliability follows from the numerous checks a compiler makes possible. Eiffel, in particular, is a fully typed language which ensures type consistency through its compiler. This is extremely important since, in an object-oriented context, "type consistency" means that whenever a routine is called on an object (or, in Smalltalk terms, whenever a message is sent to an object), the object must be prepared to deal with the routine. In a dynamically typed, interpreted environment, an object may at runtime receive a message it can't handle, resulting in failure. But in Eiffel, such a case would be detected at compile time, causing the compiler to reject the class containing it. This is essential, given Eiffel's software engineering orientation: who, after all, wants a production system liable to suddenly stop with a printout reading, "*Sorry, I cannot treat the last message I received*"?

Note that static *typing* should not be confused with static *binding*. In Eiffel (as in Smalltalk), all binding is dynamic; that is, when a routine is applied (or, if you prefer, when a message is sent) to an object and more than one version of the routine is available, the version that's applied is chosen on the basis of the object's runtime form. Static typing guarantees that there will be at least one version; dynamic binding guarantees that the "best" version will be selected. Eiffel combines the two.

But enough of reliability; what of efficiency? Eiffel generates C code, the obvious advantage in this being that it makes the environment reasonably portable. Still, it would be inappropriate to call the Eiffel compiler a "pre-processor", given that it performs all the functions of a sophisticated compiler and in no way serves as an extension of C. (We feel it is wrong—and indeed full of risks—to mesh the features of C, a low-level portable language, with advanced object-oriented features intended to improve software quality. It's hard to see how a serious software designer could accept the combination within a single language of such high-level techniques as multiple inheritance with such low-level C notions as pointer arithmetic. Simplicity, consistency, and ease of learning are probably the three foremost requirements for a programming language; under no circumstance can compatibility with past mistakes be regarded as a legitimate excuse for sacrificing these goals.)

One of the Eiffel design aims is to approach the efficiency of straightforward C coding. The current penalty in terms of space as well as time is usually no more than 20 percent, which generally is acceptable—especially considering the production benefits that derive from programming in an object-oriented fashion.

A detailed discussion of the techniques used to reach this performance goal is beyond the scope of this article, but it's worth mentioning that dynamic binding doesn't require a runtime search conditioned by the depth of the relevant inheritance diagrams. Instead, the appropriate routine can always be found in a constant (and small) period of time. This is particularly important given Eiffel's use of multiple inheritance, which without this technique would make the system prohibitively slow.

R*ecompiling Quickly.* One of the major benefits of an interpreted environment is the speed of the change-to-re-execute cycle. That is, when you see some runtime behavior you don't like, you need only stop execution, make a change, and restart the whole thing.

To emulate these results in a compiled environment, it isn't enough to write a compiler capable of quickly compiling a class. What really matters, after all, is not how you can compile an individual class but how you can recreate a working environment (what in Eiffel is called a *system*) after making a set of changes to one or more classes. Here again care must be taken to combine efficiency with reliability: compilation must be quick, but it must also guarantee that the resulting system uses the latest version of each class.

In achieving this end, we wanted to avoid using the **make** approach, under which consistency is ensured by requiring the programmer to manually enter descriptions of intermodule dependencies (*makefiles*). The process of entering these descriptions is not only tedious, but also error-prone; dependencies can easily be forgotten, especially if the software changes a lot—and object-oriented programming, after all, is there to make it easy to change your software. In any case, **make** wouldn't work for Eiffel, since it doesn't support cyclical dependencies. This is an issue because Eiffel classes may depend on each other in two ways: *A* may be a *client* of *B*, meaning it uses some of *B*'s facilities (sends messages to instances of *B*) through *B*'s official interface; the other possibility is that it may inherit from *B*. In Eiffel, inheritance is multiple, which means a class may inherit from an arbitrary number of other classes. While inheritance is an acyclic relation, the client relation may contain cycles; further, the two relations may be freely intertwined, with a class being a client of one of its ancestors or descendants (see Figure 1).

Thus, in the place of **make**, we chose to implement a fully automatic mechanism for achieving both reliability and efficiency. The

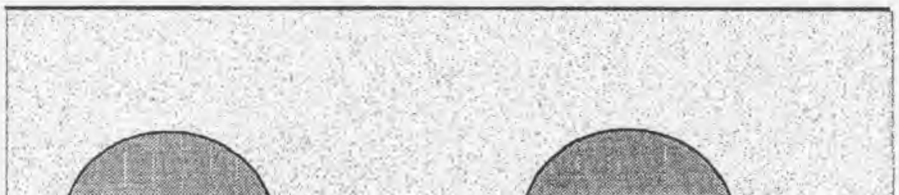


Figure 1 — Possible dependency relations in Eiffel.

magic command (**es**, for Eiffel System) can take a set of classes and find the minimum set of operations needed to recompile it. It needs only the following pieces of information:

- the name of a distinguished class serving as the system's root. A "system" includes all classes on which the root depends either directly or indirectly. Also bear in mind that dependency includes "client" as well as "heir";
- a list of the directories where the classes of the system can be found.

There is a simple correspondence in Eiffel between class names and file names: class *XXX* would be stored in file *xxx.e*, for instance. The classes of a system may be scattered over many directories, for which several different programmers can be assigned responsibility. Command **es** will find the corresponding files as needed.

Based on this information, **es** will compile classes, generating files (C, object code, and auxiliary information) in associated directories. Using time stamps, the command will only recompile as necessary; in particular, if a class has been modified but its interface hasn't been, **es** will recognize this and not recompile clients. The point of all this is that programmers neither have to recompile manually nor suffer through the maintenance of *makefile*s.

Of course, even with **es**, recompiling can never be as fast as restarting execution in an interpreted environment. But generally the delays are quite acceptable. Typically, changes in a few classes of a system, even a fairly large system, can be implemented in a few minutes.

The Generated C. C was chosen as an intermediate language for Eiffel because of its portability and its low level. The role it plays for the Eiffel compiler is akin to the role of P-code for Pascal compilers, or of quadruples and other intermediate representations for other compilers. C turned out to be less portable than initially expected, but has enabled us to port Eiffel to close to 30 UNIX platforms and, recently, to VMS.

Although the form of the C code Eiffel generates is such that users can easily trace it back to the corresponding Eiffel code (thanks to simple naming conventions and automatically generated comments), users are not expected to maintain it. (Were that the case, there hardly would be a need for languages of a higher level than C.)

An important practical advantage of using C as intermediate code is that Eiffel programs thus can communicate easily with functions written in C (or even in other languages, assuming the implementations in question follow appropriate passing conventions). This is essential if reusability is a concern, since it must be possible to interface new software with existing code.

C *Package Generation.* In standard Eiffel usage, the generated C is not relevant to programmers. There is, all the same, another **es** option by which users can produce from an Eiffel system a complete C package. This is indispensable for those software developers who need to distribute their products to environments where Eiffel is not supported. A package thus produced will consist of a set of functions corresponding to the exported routines of selected classes. Generated in a directory, the components of the system will include:

- a C translation of all the classes from the system in the form of a set of C files, where only the exported routines (including *Create*) from certain designated classes will be visible (in C form) from other C programs;
- a copy of the Eiffel runtime system, in C form;
- a system-generated *makefile*, which will allow for the automatic recompilation of the package;
- some automatically generated documentation.

Eiffel's C package generation option thus produces entirely self-contained programs, which can be moved to any machine and recompiled there, independent of any Eiffel tools. Packages produced in this way are highly portable, meaning in particular that target systems need not be UNIX-based.

Naming the C Functions When a package is generated, the problem of choosing names for the generated C functions arises. It would be too tedious for the programmer to have to specify all names; but letting the Eiffel compiler fabricate names would not be good for clarity. A natural choice is to use Eiffel names for exported routines, but then—since several classes may use the same routine names—enable programmers to remove ambiguities at the C level. Another issue is the length of identifiers; whereas many C compilers consider only the first seven or eight characters of an identifier, Eiffel has no such restriction. As a consequence, the following naming policy has been adopted for generating packages:

- By default, the original Eiffel name is used, truncated to the appropriate number of characters.
- Whenever the compiler detects a conflict, it fabricates names of its own for the second occurrence of an Eiffel name and for every subsequent occurrence.
- The compiler produces a correspondence file, giving for each Eiffel routine (identified by class name and routine name) the corresponding C function name. The programmer then is given a chance to edit the file and replace any particular C name with another. The compiler will use the names the programmer chooses and will keep and update the correspondence file, meaning the programmer won't be forced to retype choices between successive runs.

A typical correspondence file contains a set of entries such as the following:

Class	Eiffel routine	C name
Window	*display*	*display*
Window	*parent*	*parent*
Window	*height*	*win_height*
Window	*Create*	*win_Create*
...		
Screen	*height*	*scr_height*
Screen	*Create*	*scr_Create*
...		

where the third column has already been edited by the programmer to replace some conflicting C names.

Optimization. Eiffel's C package generator doubles as an optimizer, playing an important role in removing undue overhead from object-oriented techniques. In many implementations of object-oriented languages, three issues contribute serious overhead:

- A routine defined in a class may be redefined in a descendant (in the inheritance sense). Dynamic binding ensures that the version to be applied in any call depends on the runtime form of the object. The Eiffel mechanism for calling a routine takes this possibility into account. But although we've seen that the overhead is kept to a reasonable minimum, it still is desirable to obviate it altogether for routines that are never redefined.
- Object-oriented design encourages the packaging of many facilities in the same class. As a result, a considerable amount of useless code may end up getting loaded into applications, along with the desired routines. This problem can become especially serious with inheritance; designers are tempted to inherit from classes that include only a few needed facilities.
- Finally, object-oriented design, even more than "structured programming", promotes a highly modular style that requires many routine calls. This can lead to inefficiencies.

The Eiffel optimizer addresses these problems in the following ways:

- Calls to never-redefined routines are statically bound to the actual code.
- Unneeded code is removed.
- Routines are automatically expanded in line according to predefined criteria.

These optimizations are critical to obtaining the highest possible efficiency.

Documentation Tools. Good tools for exploring software that's already developed are essential in a powerful development environment. Users in interpreted environments, for example, often like to use source code as a form of documentation. The famous Smalltalk "browser" enables this use of the code, taking advantage of a multiple-windowing interface to give users access to useful classes, explored in source form.

```
class interface TABLE [T] exported features

        insert, delete, search, value, nb_elements, max_elements

feature specification

        nb_elements: INTEGER;

        max_elements: INTEGER;
        insert (element: T; key: STRING)
                -- Insert element with key key
        require
                nb_elements < max_elements
        ensure
                nb_elements <= max_elements
                value (key) = element;
                nb_elements = old nb_elements + 1

        . . .Similar interface descriptions for other routines. . .

end interface -- class TABLE
```

Figure 2 — Part of a class interface generated by short.

The Eiffel approach is somewhat different. Although library classes are made available in source form, we've tried to provide tools that also offer more abstract views of the software. These include a class abstracter (**short**), a class flattener (**flat**), and a system for generating and exploring system structures (**good**).

The Class Abstracter. The tool **short** offers a simple but useful means for taking a class and automatically producing interface documentation. For example, the command **short c** will yield the interface of class *C*. Included in this output will be the text of the class, exclusive of any implementation detail and any non-exported feature—a bit like the specification part for an Ada or Modula-2 module, but produced by the computer rather than the programmer.

An extract from sample output of **short** is shown in Figure 2 (obtained using the **-t** option, which produces **troff** code). Note that the use of Eiffel assertions is essential to abstractly express the semantics of classes (preconditions are introduced by the *require* keyword, while postconditions are introduced by *ensure*).

Flattening a Class. When a class has been defined through one or more levels of inheritance, it will often be difficult to understand the class by merely viewing it in isolation. This is because the definitions for many of the class's components can be found only by examining its ancestors. Such detailed information cannot be obtained just by using the **short** command.

The Eiffel class flattener addresses the problem by producing as output a "flat" version of the class in which it's shown as a self-contained module, without reference to any ancestor. This means that all inherited components are copied into the output class (taking renaming and redefinition into account), and that the inheritance clause is removed. For clients of the class, the resulting flat version is functionally equivalent to the original. The flat version of class *C* is obtained by executing **flat c**. By injecting this output into **short**, as in **flat c | short**, one can obtain a complete interface for *C* providing the same type of information for inherited features as for features declared in the class itself. The result is a complete interface description of *C*.

Used independently of **short**, the flattener may serve to produce a standalone version of a class (for distribution to customers, for instance).

G*ood: Graphical Exploration of System Structures.* Complementing the commands **short** and **flat** (which apply to individual classes) is a set of tools useful for controlling the design and analysis of a system's overall structure. The set goes by the name **good** (for "Graphics for Object-Oriented Design") and is used for the graphical creation and exploration of class relationships.

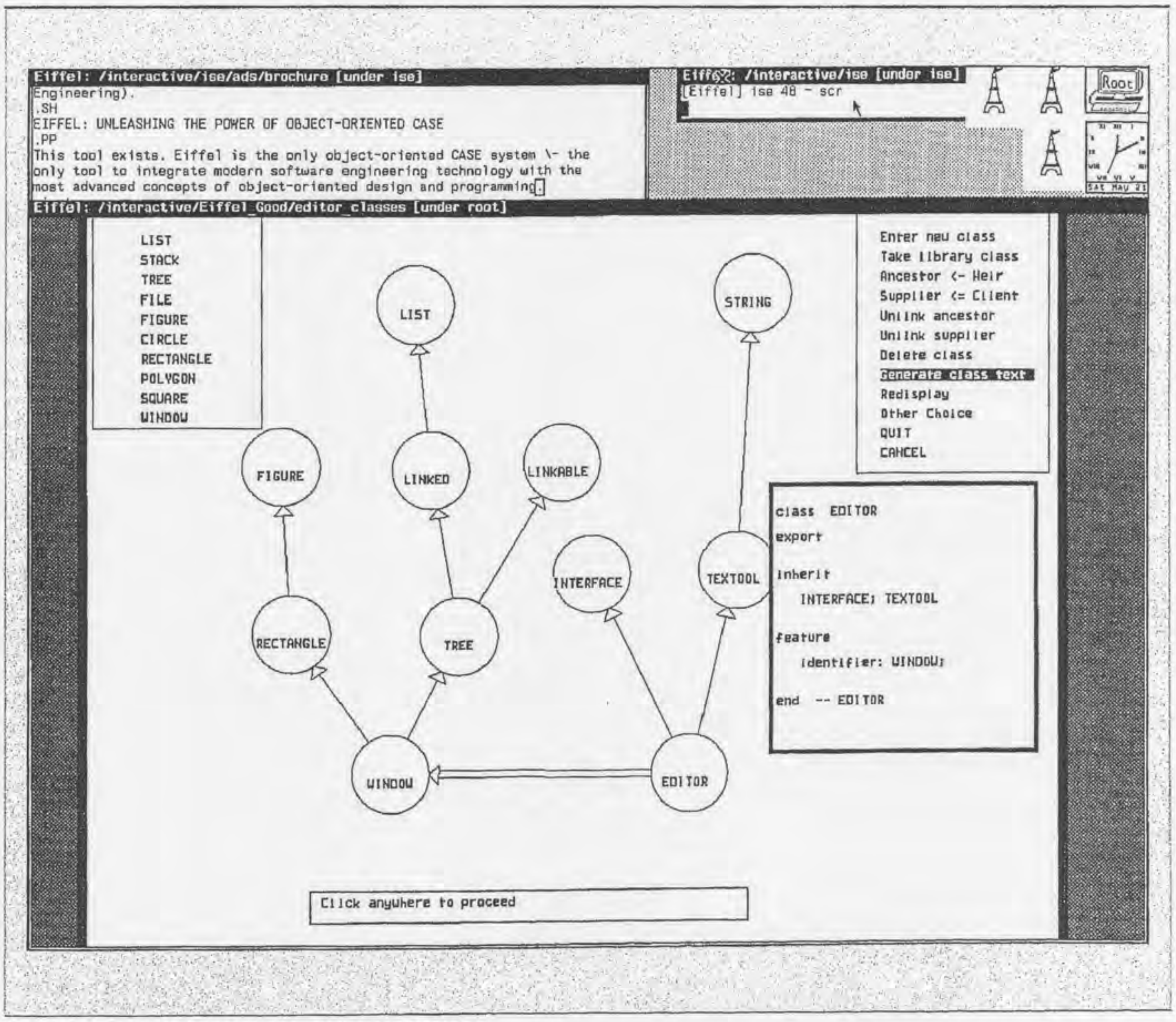
The tools of **good** are based on the graphical conventions Eiffel uses to describe classes and their connections [1]: bubbles represent classes, single arrows show inheritance, and double arrows denote client relationships. The user interface for **good** was built with the Eiffel graphics library, providing classes for windows, pop-up menus, and the like; internally, this library relies on the X Windows graphics package.

One of the applications of **good** is system analysis: starting with a class, one can see ancestors and clients. The tools also work in generation mode, meaning you can enter a new class and graphically indicate its ancestors and clients. The tools then generate the skeletons of the corresponding Eiffel class texts.

As an example, Figure 3 shows a screen from a representative **good** session. Notice how the system has generated a class superstructure from the diagram entered by a user employing a mouse.

D*ebugging and Testing.* Perhaps the most visible advantage of interpreted environments is that they make it easy to trace and control execution during searches for bugs. But compiled environments can offer similar facilities, as shown by what's available in Eiffel. The intent is to enable Eiffel programmers to forget a compiler is involved when they're trying to interactively debug an application.

It's already been noted that the Eiffel compiler is not a preproces-


```
Eiffel: /interactive/ise/ads/brochure [under ise]
Engineering).
.SH
EIFFEL: UNLEASHING THE POWER OF OBJECT-ORIENTED CASE
.PP
This tool exists. Eiffel is the only object-oriented CASE system \- the
only tool to integrate modern software engineering technology with the
most advanced concepts of object-oriented design and programming.
```

```
Eiffel: /interactive/ise [under ise]
[Eiffel] ise 48 - scr
```

```
Eiffel: /interactive/Eiffel_Good/editor_classes [under root]
```

```
class  EDITOR
export

inherit
   INTERFACE; TEXTOOL

feature
   identifier: WINDOW;

end  -- EDITOR
```

```
Click anywhere to proceed
```

Figure 3 — A screen from a representative good session.

sor. This in particular means that Eiffel programmers need not know C since they should be able to handle all debugging without venturing outside the Eiffel environment. A number of facilities make this possible:

- As previously indicated, Eiffel supports assertions—that is, "require" and "ensure" clauses which express conditions on routines, as well as class invariants which express integrity constraints. These assertions may, as an option, be monitored at runtime, providing much support for debugging if some care is taken to decorate classes and routines with proper assertions. Actually, even application classes that don't include many assertions of their own may benefit from the mechanism because of the heavy reliance typical Eiffel programming places on the basic library classes, which *have* been heavily loaded with assertions. Accordingly, many errors in application classes will give rise to assertions being violated in the basic classes (a good example of this is an attempt to insert an element into a list at an illegal position).
- Routine calls on a given class can be traced.
- The Eiffel "debug" option enables the selective execution of debugging instructions.

All of these options are selected at compile-time but may then be changed dynamically without recompilation. They are particularly useful when used in conjunction with a tool named the *viewer*. The viewer is an interactive tool for runtime exploration and object-based debugging—both of which are especially important in object-oriented systems. When the viewer is called, the user is given a chance to traverse the object structure by following references. Figure 4 includes an extract from a typical session. Among the commands available at any step are:

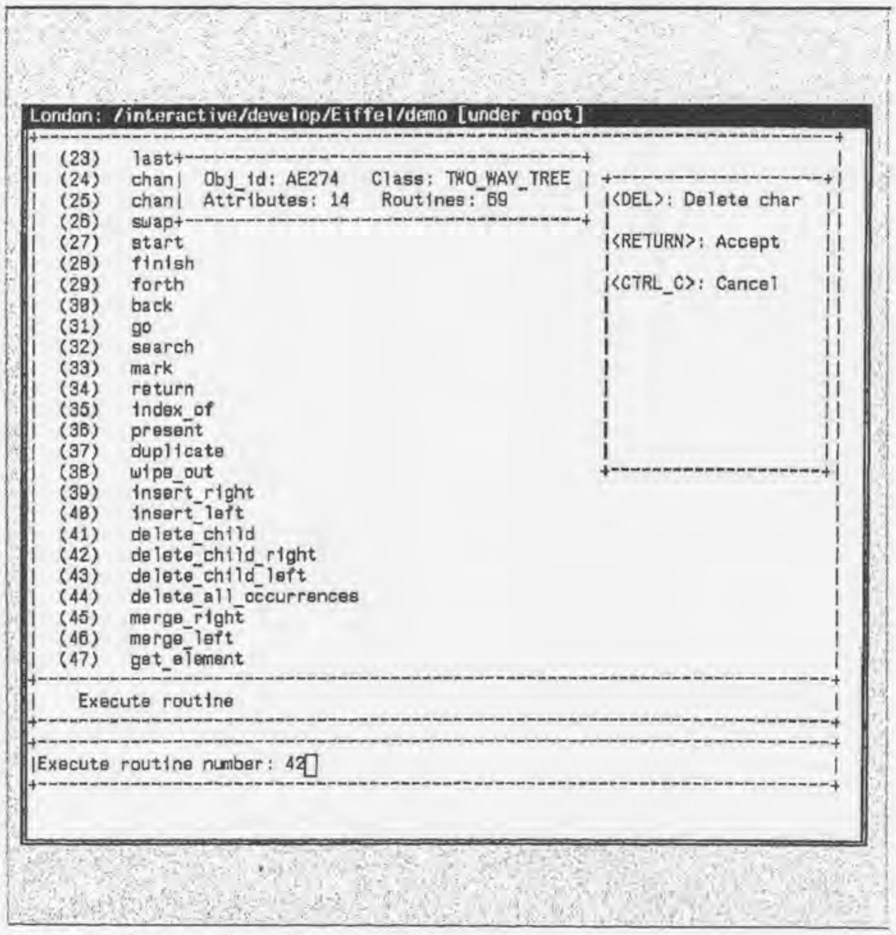
- show the values of the attributes (instance variables) for the current object;
- show the list of routines (methods) applicable to the current object (this is the command shown in Figure 4);
- move to another object referenced by an attribute of the current object;
- execute a routine on the current object;
- change the value of an attribute;
- create a new object;
- check the class invariant.

The viewer thus is not just a means for inspecting objects but a complete tool for interactive debugging.

The viewer also serves as a testing tool: the **et** (Eiffel Test) command creates an instance of a class and sets the viewer to work on it. Accordingly, users can avoid writing a test driver for handling a simple test that needn't be kept for later repetition. Built into the compiler but residing in the library class *VIEWABLE*, the viewer can be started by calling the procedure *view* (*input_file, output_file*) in any class that inherits from *VIEWABLE*. The viewer can also be set so that it will be triggered automatically when a runtime error occurs and isn't caught by the Eiffel exception-handling mechanism.

Garbage Collection. Any decent object-oriented environment must have a garbage collector, lest the task of reclaiming unused object space fall squarely onto the programmer (which, of course, would be unacceptable for all but the smallest programs). In the case of Eiffel, the task falls to a garbage collector that conceptually is a parallel process but is implemented as a co-routine activated whenever the Eiffel runtime system detects that available space is running low. Since the collector is organized as an infinite loop that can be interrupted at any point, the runtime system controls the activation time of each collector burst, using a self-adapting scheme that attempts to maintain an equilibrium in terms of space occupancy, while ensuring fairness in the competition between application and collector for CPU time.

At perfect equilibrium, memory usage will be kept constant as the application "forgets" one object for each new one it creates and the co-routine quickly recovers the space. But when equilibrium is broken and the size of occupied memory grows, the co-routine will



```
London: /interactive/develop/Eiffel/demo [under root]
+-------------------------------------------------------------------------+
|  (23)   last+------------------------------------+                      |
|  (24)   chan|  Obj_id: AE274   Class: TWO_WAY_TREE | +------------------+|
|  (25)   chan|  Attributes: 14   Routines: 69     | |<DEL>: Delete char ||
|  (26)   swap+------------------------------------+ |                  ||
|  (27)   start                                      |<RETURN>: Accept  ||
|  (28)   finish                                     |                  ||
|  (29)   forth                                      |<CTRL_C>: Cancel  ||
|  (30)   back                                       |                  ||
|  (31)   go                                         |                  ||
|  (32)   search                                     |                  ||
|  (33)   mark                                       |                  ||
|  (34)   return                                     |                  ||
|  (35)   index_of                                   |                  ||
|  (36)   present                                    |                  ||
|  (37)   duplicate                                  |                  ||
|  (38)   wipe_out                                   +------------------+|
|  (39)   insert_right                                                    |
|  (40)   insert_left                                                     |
|  (41)   delete_child                                                    |
|  (42)   delete_child_right                                              |
|  (43)   delete_child_left                                               |
|  (44)   delete_all_occurrences                                          |
|  (45)   merge_right                                                     |
|  (46)   merge_left                                                      |
|  (47)   get_element                                                     |
+-------------------------------------------------------------------------+
|    Execute routine                                                      |
+-------------------------------------------------------------------------+
+-------------------------------------------------------------------------+
|Execute routine number: 42[]                                             |
+-------------------------------------------------------------------------+
```

Figure 4 — An extract from a typical session.

Object-Oriented Engineering

Without Objective-C®, you could be missing something important.

Not all object-oriented programming environments are created equal.

Especially when it comes to features and support.

Unfortunately, many software engineers find this out too late.

Avoid missing out by using Objective-C. It's the most advanced object-oriented environment for C programmers available today.

And it will be tomorrow.

Our business is object-oriented technology, so we have the know-how and desire to keep you at the leading edge.

Objective-C is 100% compatible with the new ANSI C standard. So you can freely intermix object oriented code with "straight C." And combine new object-oriented programs with existing C code.

Objective-C provides powerful type checking and compile-time error detection independent of your C compiler. Plus, you can choose between static and dynamic binding.

Even better, you get powerful memory management features ranging from a memory leakage detector to a set of state-of-the-art garbage collectors.

There's also built-in support to automate the complexities of releasing multiply-owned objects. And built-in support for your own runtime memory manager.

Objective-C puts you in control. You determine just how much of the power of object-oriented engineering you want to use for each application

So if you want your software to turn out as efficient and safe as you planned, get Objective-C.

You'll never know what you've been missing till you do.

Stepstone™
The Leader in Object-Oriented Technology

The Stepstone Corporation 75 Glen Road Sandy Hook, CT 06482 203 426 1875 1 800 BUY OBJECT (1 800 289 6253)

Circle No. 27 on Inquiry Card

Clean Elegant Powerful...

. . . Eiffel

Object-oriented programming: Why settle for half a solution?

C++ vs. Eiffel

*"Could exceptions . . . be completely faked in a language such as C++? Unfortunately no."**

Eiffel has full exception handling.

*"Ada and CLU support parameterized types. Unfortunately, C++ does not."**

Eiffel supports parameterized types.

*"C++ does not support garbage collection."**

Eiffel includes a powerful garbage collector.

*"Unfortunately, undisciplined use of explicit type conversion can . . . create hell for innocent victims maintaining a program containing them."***

Eiffel provides full type checking.

B. Stroustrup, C++ language designer:
*IEEE *Software*, May 1988;
***The C++ Programming Language*, Add.-Wesley.

	C++	Eiffel
Parameterized classes	No	Yes
Exception handling	No	Yes
Multiple inheritance	Some versions only	Yes
Strong typing	Partial (explicit conversions)	Full
Assertions for safe program construction	No	Yes
Selective exports	No	Yes
Dynamic binding	Virtual functions only	Any routine
Renaming	No	Yes
Garbage collector	No	Yes
Automatic Make	No	Yes
Class documenter	No	Yes
Graphical design tools	No	Yes
Source-Level debugger	No	Yes

C++ information is based on the above published references and the current AT&T version (1.2.1).

Eiffel: The comprehensive solution.

Interactive
Software Engineering Inc.

270 Storke Road, Suite 7, Goleta, CA 93117 USA
Telephone: (805) 685-1006 FAX: (805) 685-6869

Eiffel is a trademark of Interactive Software Engineering Inc.

Circle No. 28 on Inquiry Card

be called in longer bursts; as it does its job and as the size of occupied memory decreases, the bursts are reduced. Below a certain threshold, the collector won't be activated at all. Conversely, if memory has been exhausted, the co-routine will turn into a full sequential mark-and-sweep garbage collector.

Further collection improvements are obtained by Eiffel through a technique known as *generation scavenging*. Under this approach, collection is accelerated by "tenuring" objects that have survived a large enough number of collector cycles.

Some applications, of course, may require no garbage collection whatsoever, so Eiffel treats collection as a compilation option. If enabled, it can always be turned off and on again at runtime. Also, the collector can be explicitly called at specified points.

The Library. The Eiffel library, which constitutes a key part of the environment, contains a set of carefully written, reusable software components, covering fundamental data structures and algorithms from table searching to trees to parsing. The elements of the library are readily available to users in source form and, besides serving as tools, are meant to serve as examples of Eiffel style. They use all the language's properties extensively, including genericity (most of the library classes have generic parameters representing types), assertions, multiple inheritance, redefinition, renaming, and such.

The library also includes a set of graphic classes, internally based on X Windows, that allow programmers to think in terms of graphical concepts such as windows, menus, polygons, circles, and the like, as opposed to X Window concepts. The already described **good** system, for instance, was built using this library.

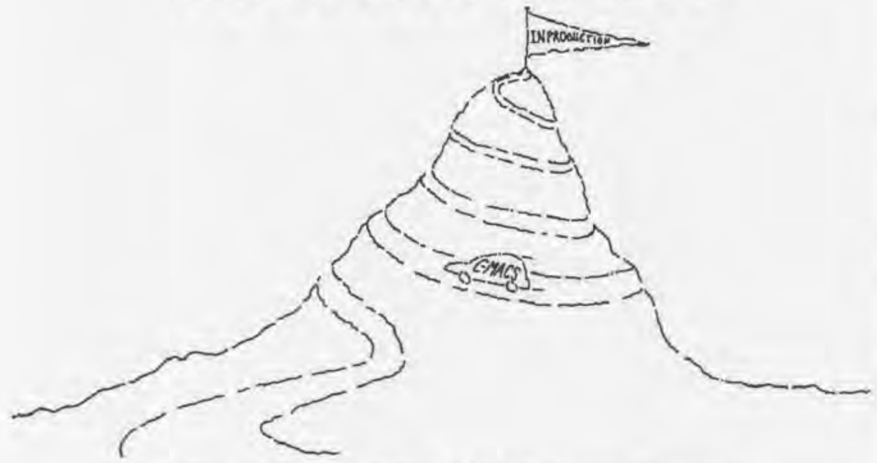
Apart from any other aspect of object-oriented design, the mere presence of the library would suffice to distinguish Eiffel programming from programming with such classical languages as Pascal and C. The effect is that Eiffel functions as a much higher-level language where the basic data types are not limited simply to *integers*, *reals*, and so on, but also include lists, trees, hash tables, and other such types—all of which are open to extension and specialization thanks to genericity and inheritance.

One of the most significant practical consequences is that it is not necessary in everyday Eiffel programming to explicitly use pointers. Although necessary for nontrivial data structures, pointer manipulations are quite tricky and prone to error. A look at the internals of the Eiffel library is sufficient evidence of this. Happily, the pointer manipulations in the library are (I hope) correct, and have been encapsulated once and for all. Hence, in standard situations, Eiffel programmers should be able to use lists and trees in the place of pointers and offsets.

Further Facilities. Among the other facilities in the Eiffel environment are:

- *Storing object structures.* To store the state of a system, or part

C-MACS
NO SPEED LIMITS



Finally, the Text Editor for C Programmers.

Today's programmers use sophisticated tools—like C-macs™. They know that each advance in software methodology has made them much more effective.

C-macs is the Newest Advance

C-macs checks and balances parentheses and braces, and inserts user definable C language control flow templates into your program.

C-macs is multi-window. You can edit and view several files or portions of the same file simultaneously to see all your work at once. It even has "shell windows" to run commands in an edit window. No more exiting the editor, running commands and re-invoking the editor!

C-macs will run "make" and collect compiler error messages in a window. C-macs then analyzes these errors and automatically positions the cursor on the erroneous source lines for editing, one-by-one.

C-macs knows C keywords and UNIX™ calls, and displays appropriate definitions and calling sequences at the touch of a button. No more getting up to look in the manual!

C-macs can optionally display pull-down menus to assist the user.

C-macs maintains "tags" of all the components of your system. This allows you to edit functions without knowing their filenames. No more wasting time hunting for your routines!

C-macs includes all UniPress Emacs™ facilities.

C-macs is available for a wide range of computers running UNIX, Ultrix™, Xenix™ and other UNIX derivatives.

Call us at **800-222-0550 (outside NJ)** or 201-985-8000. We'll send you information about C-macs and our other UNIX software products. Ask about our "Try it, Then Buy it" plan.

UniPress Software, Inc.
2025 Lincoln Hwy.
Edison, NJ 08817

Trademarks of: UNIX, AT&T Bell Laboratories; Ultrix, Digital Equipment Corp.; Xenix, Microsoft; C-macs UniPress Emacs, UniPress Software, Inc.

UniPressSoftware

Circle No. 29 on Inquiry Card

of it, into a file, the call *x.store (file_name)* can be used. This will store the whole object structure starting at *x* into *file_name*, using an appropriate external representation for handling arbitrary pointer structures (including cycles). The objects may be retrieved by *x.retrieve (file_name)*. Routines *store* and *retrieve* are part of the library class *STORABLE*, which may be inherited by any class requiring these facilities.

- *Access to internal structures.* Normally, internal object representation is irrelevant to programmers. Special applications, however, may require that the representation be directly accessed. In particular, this need arises in systems that must manipulate Eiffel objects from other languages such as C. A library class, *INTERNAL*, provides the appropriate primitives, enabling programs to obtain such information as the dynamic type of an object and the corresponding class name, the number of fields, the type and value of each field, and the like. Also provided are procedures for dynamically altering the internal structure. As with *VIEWABLE* and *STORABLE*, class *INTERNAL* must be inherited by any class that needs access to its facilities.
- *Syntax-directed editor.* A version of the general syntax-directed editor Cépage [2], which has been specifically tailored to Eiffel, supports the development and modification of Eiffel texts.

Other tools and library additions are also under development.

Being compiler-based, the Eiffel environment is very different in spirit and style from interpreted environments and so will probably never satisfy programmers who are accustomed to programming through informal experimentation and observation. Instead, it is intended to be used by developers concerned with building production-quality software, and has been shown to be applicable to industrial developments [3] as well as to teaching software engineering [4]. Still, we've found that even though the environment is compiler-based, users can be given enough power and flexibility that they needn't be jealous of their interpreter-oriented colleagues. ●

Bertrand Meyer is president of Interactive Software Engineering Inc. (Santa Barbara, CA), a company engaged in the development of tools, methods, and languages (such as Eiffel) for improving software quality. Besides holding various industry and academic positions in the US and in France, Meyer has taught at the University of California in Santa Barbara (1983-86). His latest book, Object-Oriented Software Construction, *was published by Prentice-Hall in March 1988.*

References

[1] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall (1988).
[2] B. Meyer, "Cépage, a Software Design Tool", *Computer Language* (Sept. 1986).
[3] C. Cindre and F. Sada, "A Development in Eiffel: Design and Implementation of a Network Simulator", to appear in *Journal of Object-Oriented Programming* (1988).
[4] R. Rousseau and M. Rueher, "Teaching Software Engineering Using Eiffel", Fourth AFCET Software Engineering Conference, 1988 (in French).

Circle No. 30 on Inquiry Card