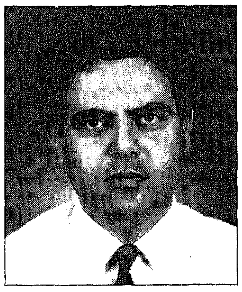
Guest column



What is an object-oriented environment?

Five principles and their application

Bertrand Meyer

ALMOST EVERY SOFTWARE development environment these days claims to be object oriented. But what does the phrase really mean? In some cases it seems that the authors of any tool that has so much as a menu or perhaps a few icons feel they deserve to call it O-O. In other cases the justification is simply that the environment supports an object-oriented language, or perhaps an object-oriented analysis method.

About two years ago, as our group at ISE was starting the development of EiffelBench (the development environment for ISE Eiffel 3), we decided to explore how we could apply the concepts of object orientation to the environment itself—not just to the software developed with it. Impressed as we were by generally accepted ideas, which usually come directly or indirectly from the brilliant example of Smalltalk, we felt that current efforts stopped short of providing the true benefits of object orientation at the environment level.

In this guest column I will share some of the insights that we gained from that effort, describe five key principles of object-oriented environments and explain why, in our opinion, such an environment should have no browser, no debugger, and, in a sense, not even a compiler or an interpreter—while providing all the needed browsing, debugging, compiling, and interpreting facilities, and more. (If you think this is a contradiction, just read on.)

Bertrand Meyer is the author of OBJECT-ORIENTED SOFTWARE CONSTRUCTION, INTRODUCTION TO THE THEORY OF PROGRAMMING LANGUAGES, and other books published by Prentice Hall. He is a member of the JOOP editorial board, the chairman of the TOOLS conference and the editor of the Prentice Hall Object-Oriented Series.

METHOD-ENVIRONMENT CONSISTENCY

What do we want in a set of development tools that claims to be an O-O environment? It should not just help produce object-oriented software but also enforce the object-oriented paradigm throughout the development process.

Underlying this observation is an idea which is hardly new: a good environment will promote a style of user interaction reflecting the style of the underlying language. Lisp environments tend to display the kind of flexible, I-know-what-I-am-doing, no-safety-net allure of Lisp and Lisp developers. Pascal environments usually look like the language—simple and clean. Most FORTRAN environments exhibit the same "rugged simplicity of a Ford Model-T" that was once described (by D.W. Barron) as characterizing the spirit of FORTRAN. Environments for C and derivatives appropriately promote direct access to addresses, pointers, memory blocks, words, bytes, signals handlers, runtime call stacks and other machine-level features, focused on the main task that occupies developers in such environments: discovering the bug du jour. In environments supporting the most popular analysis methods, the little clouds and bubbles which seem to be the main selling points of these tools accurately reflect the vagueness surrounding the methods' theoretical foundations and practical usefulness.

We may express this observation as a principle applicable to many environments:

> Principle 1 (method-environment consistency)—A development environment meant to support a particular method or language must rely on a consistent set of user interaction conventions which closely parallel the concepts promoted by the method or language.

This principle does not just state that the environment must support the method or language—a rather obvious requirement—but that the concepts that prevail in the method or language must also apply, appropriately transposed, to the interaction between developers and the environment.

DATA ABSTRACTION AND DEVELOPMENT OBJECTS

Applied to the specific case of an object-oriented environment, the above principle means that the environment itself should let its users (called *developers* in the rest of this discussion) work in an object-oriented way.

What does that mean? This article is not the place for a lengthy definition or discussion of the O-O paradigm; but on one thing all JOOP readers will (I hope) agree: object orientation means *data abstraction*. The two words in this phrase are equally important:

- The emphasis on *data* means that the basis for our work is object types (classes), not operations. Any operation that we may have to perform is relative to a certain object.
- The use of *abstraction* means that we systematically apply information hiding: an object type is known through the applicable operations, and these operations are described through their abstract properties (assertions), excluding any implementation-related considerations.

In object-oriented software development we apply these concepts to build our software out of classes describing the objects that our software models (at the analysis levels) and implements (at the design and implementation levels). These objects—linked lists, bank accounts, lines of text, airplanes, etc.—may be called *software*

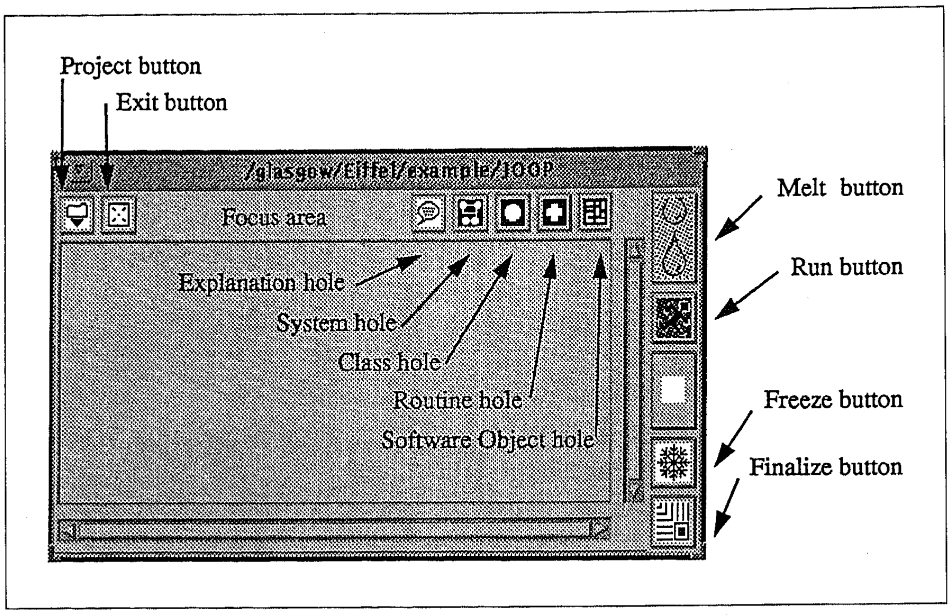
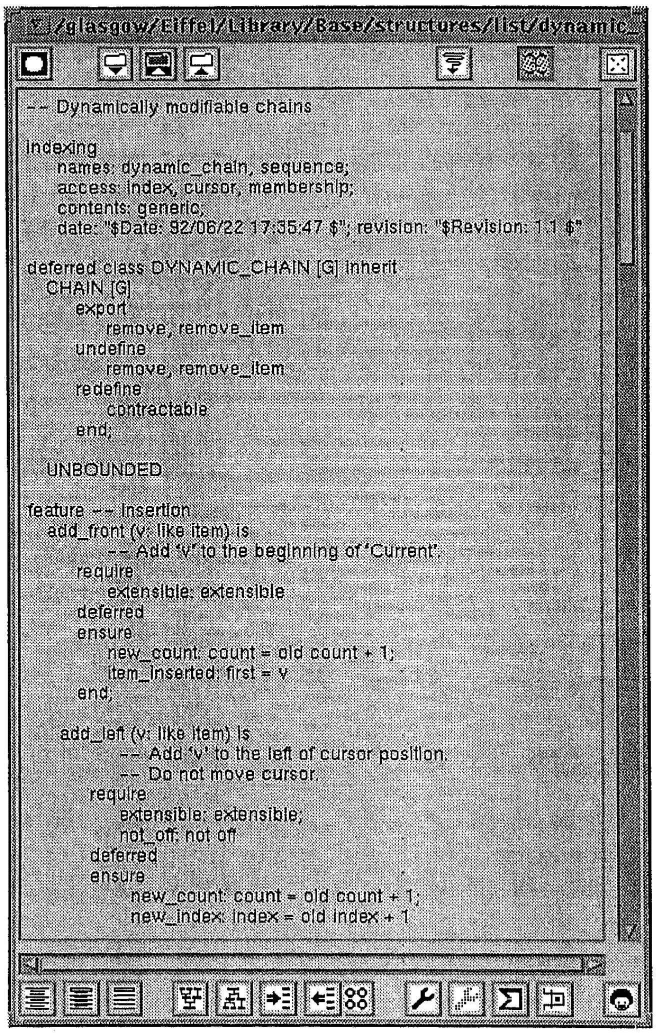


Figure 1. Development object types.



```
-- Dynamically modifiable chains

indexing
    names: dynamic_chain, sequence;
    access: index, cursor, membership;
    contents: generic;
    date: "$Date: 92/06/22 17:35:47 $"; revision: "$Revision: 1.1 $"

deferred class DYNAMIC_CHAIN [G] inherit
  CHAIN [G]
     export
        remove, remove_item
     undefine
        remove, remove_item
     redefine
        contractable
     end;

  UNBOUNDED

feature -- Insertion
  add_front (v: like item) is
        -- Add 'v' to the beginning of 'Current'.
     require
        extensible: extensible
     deferred
     ensure
        new_count: count = old count + 1;
        item_inserted: first = v
     end;

   add_left (v: like item) is
         -- Add 'v' to the left of cursor position.
         -- Do not move cursor.
      require
         extensible: extensible;
         not_off: not off
      deferred
      ensure
         new_count: count = old count + 1;
         new_index: index = old index + 1
```

Figure 2. A Class tool showing the Text format for a class.

objects because they are manipulated by the software.

We may apply exactly the same ideas to the way developers interacts with their development environments. But the objects that we need here are not software objects any more: they correspond to the things that developers (not their software!) manipulate. We may call them *development objects.* The main types of development objects include:

- The class.
- The system. (I am using Eiffel terminology here; a system is an executable assembly of classes. In other O-O approaches the corresponding concept would be *program.*)
- The feature. (The closest concept in other approaches is *method* or *function.*)
- The explanation (if a help facility is desired).
- Software objects—at runtime, for testing and debugging purposes among others, you may want to capture some software objects, see their field values, and follow references to other software objects.

The first characteristic of an object-oriented environment, then, is defined by the second principle, which is, as the others in this discussion, a consequence of the first:

> **Principle 2 (data abstraction)**—In an object-oriented environment, the basic way of working must be through direct manipulation of visual representations of developer abstractions.

With an object-oriented environment, then, the screen will show various development objects—classes, routines, systems, software objects—under the appropriate representations, and enable developers to work with them using the principle of direct manipulation (as introduced originally by Shneidermann[1] and now widely accepted).

The following control panel from EiffelBench shows some of the major pictorial representations for the fundamental development object types (Fig. 1).

OBJECT-ORIENTED TOOLS

The data abstraction principle has more implications than are apparent at first. If you look at most of today's environments, including those for object-oriented languages, they have tools corresponding to operations: a brows*er*, compil*er*, debugg*er*, test*er*. This is wrong! In an O-O environment, we will have none of this "*er*" stuff. What we want are *object* tools: the Class tool, Routine tool, System tool, Software Object tool. This is our third principle:

> **Principle 3 (object-oriented tools)**—In an object-oriented environment, each tool must be based on an object type (not on a type of operation).

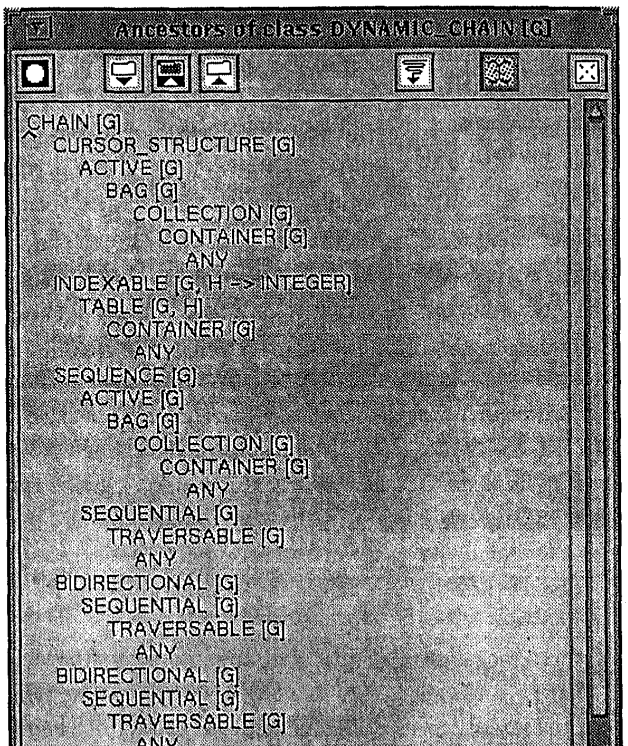
Figure 2 is an example of what is perhaps the most important tool in EiffelBench: the Class tool. It shows a class text users can edit using normal editing facilities. The Class tool shows part of the text of class DYNAMIC_CHAIN, a class from the standard data structures and algorithms library, EiffelBase. (DYNAMIC_CHAIN is a deferred, or abstract, class, which serves as ancestors to classes describing extensible linear structures such as LINKED_LIST and TWO_WAY_LIST.) We say that class DYNAMIC_CHAIN is the current target of the tool; this mirrors the notions of target of a call and of current object in object-oriented computation (the feature call ao+f has a as its target, and during its execution a is the current object of the computation).[2]

At the bottom of the Class tool window, a number of format buttons appear. They make it possible to show information on the class in various ways. (Note that icons have no associated text, to avoid bothering knowledgeable users by taking up precious screen space. To know what an icon stands for, just bring the cursor to it and the "Focus" area at the top of the control panel will display its meaning, for example "Flat form" or "ancestors") Here are some of the formats for a class:

- **Text**—the default, showing the class text.
- **Flat**—the developed version of the class, with all inherited features put at the same level as the immediate features defined in the class itself.
- **Flat-short**—the class interface, which keeps exported features and their assertions ("contracts") but removes all implementation information.
- **Ancestors**—The inheritance hierarchy leading to a class.
- **Descendants.**

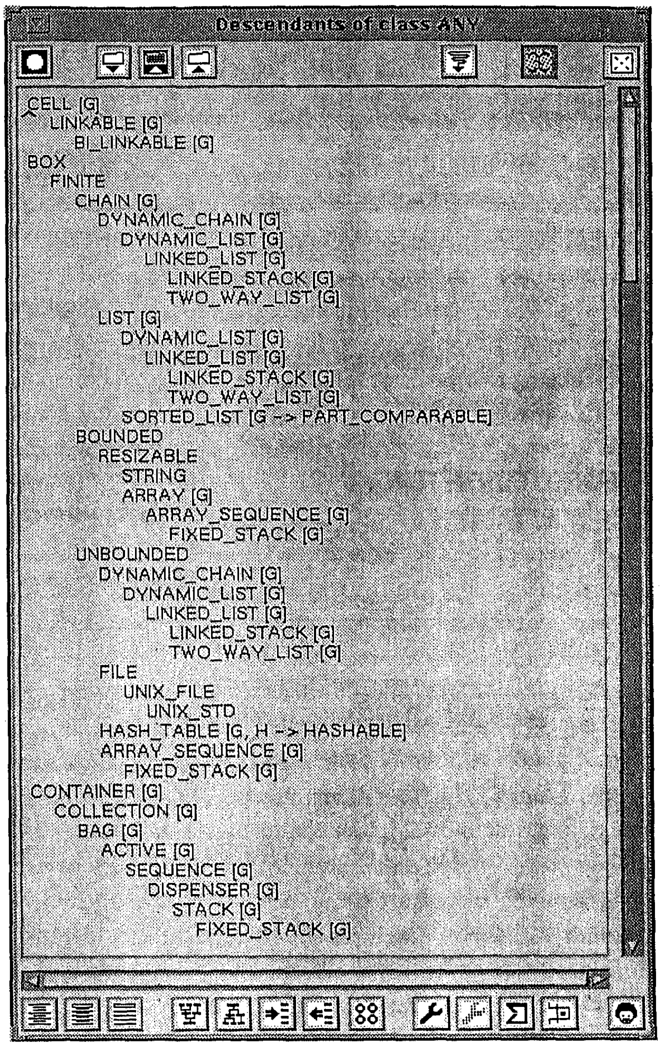
Other formats include clients, suppliers, attributes, routines, deferred routines, once routines, and "custom" (through which you can devise a specific format and set of selection criteria).

Figure 3 shows the Ancestors format for class DYNAMIC_CHAIN. Note the systematic use of multiple inheritance, which is a central property of the Eiffel method and makes it possible to build powerful classes with little effort. The EiffelBase library is the result of a long-term effort to produce a multi-criterion taxonomy of the fundamental structures of computer science[3]; some aspects of the taxonomy are visible in the inheritance structure shown.



Ancestors of class DYNAMIC_CHAIN [G]

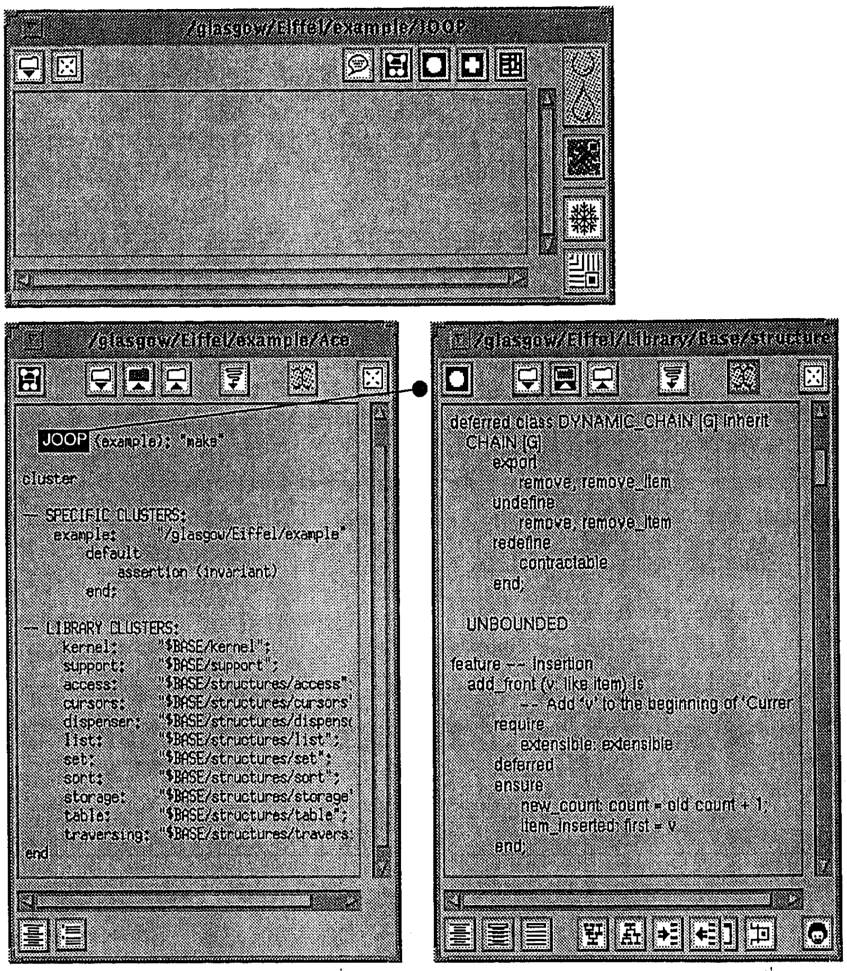
```
CHAIN [G]
  CURSOR_STRUCTURE [G]
    ACTIVE [G]
      BAG [G]
        COLLECTION [G]
          CONTAINER [G]
            ANY
  INDEXABLE [G, H -> INTEGER]
    TABLE [G, H]
      CONTAINER [G]
        ANY
  SEQUENCE [G]
    ACTIVE [G]
      BAG [G]
        COLLECTION [G]
          CONTAINER [G]
            ANY
    SEQUENTIAL [G]
      TRAVERSABLE [G]
        ANY
  BIDIRECTIONAL [G]
    SEQUENTIAL [G]
      TRAVERSABLE [G]
        ANY
  BIDIRECTIONAL [G]
    SEQUENTIAL [G]
      TRAVERSABLE [G]
        ANY
  FINITE
    BOX
      ANY
```



Descendants of class ANY

```
CELL [G]
  LINKABLE [G]
    BI_LINKABLE [G]
BOX
  FINITE
    CHAIN [G]
      DYNAMIC_CHAIN [G]
        DYNAMIC_LIST [G]
          LINKED_LIST [G]
            LINKED_STACK [G]
            TWO_WAY_LIST [G]
      LIST [G]
        DYNAMIC_LIST [G]
          LINKED_LIST [G]
            LINKED_STACK [G]
            TWO_WAY_LIST [G]
        SORTED_LIST [G -> PART_COMPARABLE]
    BOUNDED
      RESIZABLE
        STRING
        ARRAY [G]
          ARRAY_SEQUENCE [G]
            FIXED_STACK [G]
    UNBOUNDED
      DYNAMIC_CHAIN [G]
        DYNAMIC_LIST [G]
          LINKED_LIST [G]
            LINKED_STACK [G]
            TWO_WAY_LIST [G]
      FILE
        UNIX_FILE
          UNIX_STD
      HASH_TABLE [G, H -> HASHABLE]
      ARRAY_SEQUENCE [G]
        FIXED_STACK [G]
CONTAINER [G]
  COLLECTION [G]
    BAG [G]
      ACTIVE [G]
        SEQUENCE [G]
          DISPENSER [G]
            STACK [G]
              FIXED_STACK [G]
```

Figure 3. A Class tool showing the Ancestors format for a class.
Figure 4 (right). A Class tool showing the Descendants format for a class.


/glasgow/Eiffel/example/JOOP
/glasgow/Eiffel/example/Ace
/glasgow/Eiffel/Library/Base/structure

```
JOOP (example); "make"

cluster

-- SPECIFIC CLUSTERS:
   example:      "/glasgow/Eiffel/example"
        default
            assertion (invariant)
        end;

-- LIBRARY CLUSTERS:
    kernel:      "$BASE/kernel";
    support:     "$BASE/support";
    access:      "$BASE/structures/access";
    cursors:     "$BASE/structures/cursors'
    dispenser:   "$BASE/structures/dispens
    list:        "$BASE/structures/list";
    set:         "$BASE/structures/set";
    sort:        "$BASE/structures/sort";
    storage:     "$BASE/structures/storage'
    table:       "$BASE/structures/table";
    traversing:  "$BASE/structures/travers:
end
```

```
deferred class DYNAMIC_CHAIN [G] inherit
   CHAIN [G]
      export
         remove, remove_item
      undefine
         remove, remove_item
      redefine
         contractable
      end;

   UNBOUNDED

feature -- Insertion
   add_front (v: like item) is
         -- Add 'v' to the beginning of 'Currer
      require
         extensible: extensible
      deferred
      ensure
         new_count: count = old count + 1;
         item_inserted: first = v
      end;
```

Figure 5. Dragging a development object.

All branches of the inheritance graph ends at class ANY, the mother of all classes. Figure 4 shows the beginning of the Descendants format for ANY; this is a good way of seeing the inheritance structure of an entire system.

SEMANTIC CONSISTENCY

The next principle is essential if we are to enable users to interact with the environment effectively and consistently. Development objects will appear, under various guises, on different parts of the screen. For example, class DYNAMIC_CHAIN appears as the target of the Class tool of Figure 2, but it also appears, represented by its name, on Figures 3 and 4. The class name may also appear in a System tool under the format displaying the list of all classes in the system.

Assume that as you are using the environment you spot a development object, shown under any suitable visual form—for example, the object's name appearing in a class text, or some icon which has been associated with it. In a flash you decide that you need to perform some operation on the object; depending on the object's type, this may be an operation that obtains more information about the object, modifies its text, compiles it (for a class), executes it (for a routine), or changes a field (for a software object). In any such case you will want to grab the development object right away, at the very place where you have found it. You should be able to do this regardless of the tool in which you have spotted the object (Class tool, Routine tool, System tool, etc.), the format selected for that tool (Text, Descendants, Clients, etc.), the visual representation under which the development object appears in the tool (textual name, graphical icon, etc.) and the place where it appears. In all circumstances you should be able to choose from the same set of applicable operations, determined only by the object's type and properties. Hence the fourth principle, which is perhaps the most important in this discussion:

Principle 4 (semantic consistency)—An object- oriented environment must enable its users, for any symbol (textual, graphical, or otherwise) representing a development object in the user interface, to select the object through its symbol and apply any operation that is semantically valid for the object, regardless of the symbol's context—tool, location, format, representation.

TYPED DRAG-AND-DROP

Now for the next question: given that we can select an object wherever we see it, how are we going to perform operations on it? Here there is no absolute principle, as various user-interface tastes may play their role.

A natural convention (assuming the now common WIMP style of interface—for windows, icons, menus, pointing device) is to let the user click on the symbol representing the developer object and display a menu listing the various applicable operations. When our group was considering that issue, however, we were a bit tired of menus and menu selection, and devised another technique which we (and apparently our users, too!) have come to like very much: the typed drag-and-drop interface technique. With this approach, the basic operation consists of grabbing a software object, identified by its name appearing in one of the tool windows, and dragging it into a matching hole of some tool (the same or another). As you are dragging a software object, its type is represented by a small icon known as a pebble: a disk pebble if the development object is a class, a cross-shaped pebble if it is a routine, a graph-like pebble if it is a system, and so on.

The environment's basic operation, then, is very simple: drag a pebble into the corresponding hole in a tool.

For example, consider the situation in Figure 5. I have selected the name of a class, JOOP, in the System tool on the left; because JOOP is the name of a class, the pebble shows a disk. If I drag this pebble and drop it into the hole of the Class tool on the right, JOOP will become the new target of that Class tool; in other words, the Class tool, which was already in Text format, will now show the text of JOOP rather than the text of DYNAMIC_CHAIN. Rather than overwriting an existing Class tool, I could get a new one (in a new window) by dropping the pebble into the class hole of the control panel.

Now you have probably guessed the rule behind typed drag and drop. Eiffel is a typed language; in accordance with Principle 1 and with the rest of this discussion, the environment should be typed, too. The pebble shapes serve to visualize the types of development objects; and a pebble of a certain shape can only be dropped into a hole of a matching shape. For example, you cannot drop a class pebble into a routine hole. (If you try it, nothing will happen.) Although, as noted, the exact choice of interface conventions is in part a matter of taste, there is one more general principle at work here:

> **Principle 5 (typed environment)**—In an object-oriented environment supporting a statically typed language and method, the environment's visual conventions should display and enforce the type constraints on development objects.

In the same way that a typed object-oriented language such as Eiffel offers some flexibility (thanks to inheritance) as to what kinds of values can be assigned to each other, the environment lets pebbles match holes in some cases where they are not of identical shapes. For example, you may drop a feature pebble into a class hole; the new target of the class hole will then be the class in which the feature appears. This is a nice and easy way to obtain the context of a feature: you select the feature at a place where its name appears (e.g., in a call to the feature) and bring it to a class hole. From then on you can move to ancestors of the feature's class, to its ancestors, and so on.

We have applied the Typed Drag and Drop model of interaction to all the components of ISE Eiffel 3—not just EiffelBench but also EiffelBuild (the interactive application builder), EiffelCase (the analysis and design environment), etc.; we find that it provides a convenient and general mechanism, which is both easy for novices to learn and still pleasant for experienced users to use.

ERRORS: PREVENTION RATHER THAN CURE

Typed Drag and Drop has an important effect on the general style of user interaction with the environment. One of the annoying features of most WIMP-style tools is the number of times an "alert panel" pops up, forcing you to click on an "OK" button. This usually occurs as the result of an error, but sometimes it is just because the tool authors want to make sure that you read a certain message before proceeding. Alert panels are irritating, and fail to meet what I believe to be an important principle of interface design: it is always better to prevent errors than to detect them after the fact.

We have not yet found it possible to remove all alert panels altogether. But we have found that applying the following rules yields a much improved user interface:

- Use Typed Drag and Drop to implement consistency rules whenever possible. In this way many potential errors disappear: they simply correspond to cases in which a pebble does not fit a hole.
- With this rule, the application will have very few alert panels if any. Make sure every remaining case is justified and cannot be handled by less obtrusive methods.
- Never use an error panel requiring a single possible action (such as clicking "OK"). If you are requiring the user to act, you should—if only out of politeness!—give him a choice.

BROWSING WITHOUT A BROWSER

By now you may be beginning to see how one can browse—and browse quite effectively—without a specific "browser" tool. Imagine the following sequence of operations:

- Start from the System tool.
- Grab the name of the root class (the place where execution starts). Assume the root class name is ROOT. Drag-and-drop-it into the class hole of the control panel. This starts a new class tool, with ROOT as a target. The default format is Text, so the class tool shows the text of ROOT.
- Change the format to Suppliers. The clients of ROOT (the classes that ROOT uses through calls) appear.
- In this list of suppliers choose one particular client of ROOT, class C. Grab it and drag-and-drop it to the class hole of the current class tool. Change the format to Text; the text of C appears.
- Change the format to Ancestors. Grab one of the ancestors of C, say, D; drag-and-drop it to the class hole. Select the Routines format; the list of all the routines of D appear, each with its class of origin (the class where it was first introduced). Grab one of these routines and drag-and-drop it to the routine hole of the control panel, and so on.

Rather than explaining all these manipulations on paper I would really prefer to show them to you in real time, or, better yet, let you play with the actual environment, but following a written description is the next best thing. I hope you are getting the knack of this *proximity-based browsing*, which is the nicest way I know to move around a system quickly and effectively.

DEBUGGING WITHOUT A DEBUGGER

So far we have just explored the structure and components of existing software. What about changing class texts, compiling classes, debugging a system?

For the debugging part, I'll leave you the pleasure of guessing some of the details and seeing how it all fits into place. We don't want a debugger tool, of course. Instead, we take the Routine tool and the Software Object tool (both of which are object-oriented tools, not functional ones) and consider such operations as:

- Put in a breakpoint at a specific place in the routine.
- Resume execution of the routine until the next breakpoint.
- Resume execution of the routine until the next call.
- Remove a breakpoint.
- Grab a variable name (the Eiffel term is *entity*) and drag-and-drop it into a software object hole, which will show the contents of the corresponding run-time object.
- Grab an object field representing a reference to another object, and drag-and-drop it into an object hole.

And so on. You must see the picture by now, and understand how one can debug without a debugger.

Something else that you may have guessed is how the Help facility works. Assume that during a compilation step (as

ADVERTISER INDEX

	PAGE #	CIRCLE
Arkhon Technologies	83	Recruitment
Cadre Technologies	C4	176
DataLink	82	Recruitment
Digitalk	C2	34
Excel Software	25	110
Fastrak	13	177
Houser, Martin, Morris, & Assoc.	82	Recruitment
IBM	2	13
Knowledge Systems Corp.	19	54
NeXT, Inc.	20–21	80
Object Design	1	17
The Object People	39	81
Objectivity	5	99
Objectshare Systems	49	175
OOPSLA '93	67–69	63
Poet Software (formerly Bks)	11	90
Rational	7	43
Rogue Wave Software	29	131

	PAGE #	CIRCLE
SAS Institute	23	—
Semaphore	15	12
SES	9	167
Set Labs	37	56
SIGS Books:		
Writers Wanted	82	Recruitment
Object Lessons	84	120
Conference Proceedings	94	178
White Paper	85	124
SIGS Conferences:		
C++ World	17	186
Destination C++	30–31	172
SIGS Publications		
Cumulative Article Index	63	165
SYBASE	83	Recruitment
UniSQL	27	153
Versant Object Technology	C3	88
Walt Disney Imagineering	33	Recruitment

described next) an error occurs. A code and brief explanation will show up on the control panel. To know more about the error, for example, if it is a violation of a language constraint as discussed in EIFFEL: THE LANGUAGE,[2] grab the code with the mouse. The pebble in this case has the form of the "explanation" hole. Drag-and-drop it to the Explanation hole of the control panel; and, voilà, the explanation pops up in an Explanation tool, in the form of the complete language rule straight from the book.

You can of course drag many other objects to an Explanation hole. In fact, the usual way of obtaining information about an object, in EiffelBench as in other environment components, is to drag it into an Explanation hole.

COMPILING: THE MELTING ICE TECHNOLOGY

The final aspect is compiling. You play around with a few classes, change their texts; in Text format, the Class tool doubles as an editor and you can change class texts (other formats are read-only). You save your changes. Then you want to recompile.

EiffelBench offers not one but three compiling mechanisms: melting, freezing, and finalizing. You can trigger them by clicking on one of the three buttons on the right side of the control panel (see Fig. 1).

Why three compiling modes? Compilation should reconcile the following goals:

- **C code generation**—for portability, it is useful to use C in its proper role, that of a portable assembly language rather than a language for programmers to use directly (except in special cases). The final output of a compilation, then, will be a complete C package that can be ported to various platforms.
- **Security and efficiency of the generated code**—traditional compiling techniques for typed languages ensure that compilers can catch many errors before it is too late, and generate more efficient code.
- **Quick turnaround**—interpreter-based environments make it possible to have an almost immediate transition from the time you write or (more commonly) modify software to the time when you can execute the result of what you just wrote.

These goals, especially the last two, have so far tended to be mutually exclusive. A good compiler and linker may perform extensive checking and generate excellent code, but this takes time. An interpreter processes your changes quickly, but performs few checks and usually sacrifices runtime performance.

We wanted to have the best of all worlds and avoid the limitations of the best traditional answer—incremental compilers. Hence the idea of the *melting ice*, which is based on the following analysis.

Most of the compilation literature studies the problem of compiling an entire program. The practical problem is more that of processing an incremental change to an existing software system. The change may be big or small; the system may be big or small. (By "small" we mean up to a few tens of thousands of lines.) Of the four possible cases shown in Table 1, only one is really interesting:

If the system is small (left column), speed of recompilation with a good compiler such as earlier Eiffel compilers (e.g., ISE Eiffel 2.3) will be acceptable (although it never hurts to make it faster). In the bottom-right box, you have spent (say) six weeks changing dozens of classes in a big system; then, frankly, you can wait a little. Go and reward yourself with a good dinner after starting the recompilation, and come back the next day. The really important case—and the one that can cause most frustration—is the one marked ***: you change only a small part of a big system. Then you will want the result *now*. A few seconds' wait will be tolerable, but not much more.

Hence the melting ice technology. As you start with your system you will do a first compilation—possibly a bit slow, but that does not matter too much, as the system is still small. In melting ice terminology, you have *frozen* your system, as if you had put a block of ice in the freezer.

You come in the next morning, take the system—the ice—out of the freezer, and start working on it. As you work hard with your mind, your forehead produces some heat, and a few drops of water figuratively fall into a bucket. The drops are the software elements that you have changed.

One thing that you would *not* want to do after a few such changes is refreeze the system: that would take far too long. In software terms this means that you will only rarely perform a global recompilation. Instead, the melted part (the changes) will be processed much faster.

At execution time, the frozen part will still be run in compiled mode, but the melted part will be partly interpreted. Of course all this is far from trivial since the melted and frozen elements must be able to talk to each other, but that is the business of the environment's implementers, not of its users. What matters for the users is that interpretation does not have any negative effects,

Table I.

	Small System	Big System
Small Change		***
Big Change		

since typically you will only melt a small part of a system; the impact on efficiency is then negligible. Also, melting still performs all the type checks that you may expect from a serious development environment. But the key property of melting is its speed: the time needed to melt a system after a change depends only on the size of the change and its logical implications, never on the size of a system. This satisfies the major requirement of software developers in this area: small changes to big systems should recompile quickly.

Some gauge on the screen should tell you what proportion of the system you have melted so far. When that proportion becomes a little high, you may begin to experience a decrease in efficiency: time to freeze again. It may be a good idea to get into the habit of freezing before going home every night. (I take that back. Not all software developers go home at night. Some go home in the morning. Some seem never to go home. For some, home is where the computer is.)

By now the meaning of the three compiling buttons on the right of the control panel should be clear:

- Click on the melt button after making changes. After a short while—typically ten seconds or so—your system will be ready to execute again.
- Click on the Freeze button to restart on a clean basis after many changes, or for the first compilation of a system (although you may start with a melt if you prefer).
- Click on the Finalize button at the end of a project. Finalizing produces portable and highly optimized C code. Note that some optimizations can only be done in final mode as they apply to an entire system. For example, dead code removal is impossible as long as you remain under EiffelBench: like Lazarus, a feature that is dead today (because no one calls it) can become alive again tomorrow (if you insert a call to it somewhere). Only when you finalize can you safely play Gogol's Revizor and sort out the legitimate dead souls from the living.

One point I almost forgot to mention, since it is obvious to Eiffel developers: in any compilation mode the analysis of what has changed, of what is still the same, and of the set of software elements impacted by a change through the client and inheritance relations, is entirely automatic. No Make file, no Include file of any kind. These are tedious and error-prone mechanisms, and a thing of the past; developers have better things to do than telling compilers about information which, with a little effort, can be deduced from the text of the software itself.

IN SUMMARY

There is undoubtedly more to the notion of object-oriented environments than has been discussed in this column, and no doubt others will explore further implications of the ideas presented here. But I hope to have shown that it is possible to apply object-oriented principles much more systematically than has been thought possible so far, and that the software engineering principles we use for the software we produce can have fruitful consequences on the very process of producing it.

These observations are representative of an idea that strikes me ever more often as I examine the implications of the object-oriented paradigm—not the somewhat degraded version that one finds in many current technical publications, but the serious view based on abstract data types and other profound ideas. The methodological and epistemological consequences of these principles extend, I believe, far beyond software, to domains such as sociology and economics. But this is the theme for some other article. At least this one may have brought to your attention some non-trivial applications of O-O ideas to the way we interact with our everyday working tools. ■

References

1. Shneiderman, B. Direct manipulation: A step beyond programming languages, IEEE Computer, 16(8):57–69, 1983.
2. Meyer, B. Eiffel: The Language, Prentice Hall, Englewood Cliffs, NJ, 1992.
3. Meyer, B. Eiffel: The Libraries (forthcoming), Prentice Hall, Englewood Cliffs, NJ, 1993.