36

An object-oriented environment

Late into Beethoven's Choral Symphony, a baritone breaks the stream of astounding but until then purely instrumental sounds to awake us to something even grander:

> *O my friends! No more of these tunes!*
> *Let us strike up instead*
> *Some more pleasant and friendly songs.*

After reviewing in the preceding chapters some of the common approaches to O-O development, we should similarly end with a perhaps more modern and comprehensive approach (with no intended disparagement of the others; after all the Ninth's first three movements, before it goes vocal, already were pretty decent stuff.)

The diagram is on page 1149.

This chapter presents an environment (ISE's) that relies on the principles developed in the rest of this book, and makes them available concretely to O-O software developers. A complete diagram of the environment appears later in this chapter; some of the principal components are included for trial purposes in the CD-ROM attached to this book.

The purpose of this presentation is to put the final touch to our study of object technology by showing how environment support can make the concepts convenient to use in practice. A caveat: nothing in this discussion suggests that the environment discussed below is perfect (in fact, it is still evolving). It is only one example of a modern O-O environment; others — such as Borland's Delphi to name just one — have met wide and deserved success. But we need to explore one environment in some depth to understand the connection between the method's principles and their day-to-day application by a developer sitting at a terminal. Many of the concepts will, I hope, be useful to readers using other tools.

36.1 COMPONENTS

The environment combines the following elements:

- An underlying *method*: the object-oriented method, as described in this book.
- A *language*, the notation presented in this book, for analysis, design and implementation.
- A set of *tools* for exploiting the method and the language: compiling, browsing, documenting, designing.
- *Libraries* of reusable software components.

The next sections sketch these various elements, except for the first which, of course, has been the subject of the rest of this book.

36.2 THE LANGUAGE

The language is the notation that we have devised in part C and applied throughout the book. We have essentially seen all of it; the only exceptions are a few technical details such as how to represent special characters.

Evolution

The first implementation of the language dates back to late 1986. Only one significant revision has occurred since then (in 1990); it did not change any fundamental concepts but simplified the expression of some of them. Since then there has been a continuous attempt at clarification, simplification and cleanup, affecting only details, and bringing two recent extensions: the concurrency mechanism of chapter 30 (concretely, the addition of a single keyword, **separate**) and the *Precursor* construct to facilitate redefinition. The stability of the language, a rare phenomenon in this field, has been a major benefit to users.

Openness

Although a full-fledged programming language, the notation is also designed to serve as a wrapping mechanism for components that may be written in other languages. The mechanism for including external elements — the **external** clause — was described in an earlier chapter. It is also possible, through the *Cecil* library, for external software to use the O-O mechanisms: create instances of classes, and call features on these objects, through dynamic binding (but of course with only limited static type checking).

Of particular interest are the C and C++ interfaces. For C++, a tool called *Legacy++* is available to produce, out of an existing C++ class, a "wrapper" class that will automatically include the encapsulation of all the exported features of the original. This is particularly useful to developers whose organizations may have used C++ as their first stop on the road to object orientation in the late eighties or early nineties, and now want to move on to a more complete and systematic form of the technology — without sacrificing their investment. Legacy++ smoothes the transition.

36.3 THE COMPILATION TECHNOLOGY

The first task of the environment is, of course, to let us execute our software.

Compilation challenges

Developed over many years and bootstrapped through several iterations, the compilation technology is an answer to a set of challenges:

C1 • The efficiency of the *generated code* must be excellent, comparable to what developers could obtain by using a classical language such as C. There is no reason to pay a significant performance price for O-O techniques.

C2 • The *recompilation time* after a change must be short. More precisely, it should be proportional to the size of the change, not to the size of the entire system. The crucial compilation concern, for developers working on a possibly large system, is the need to perform changes and see the results immediately.

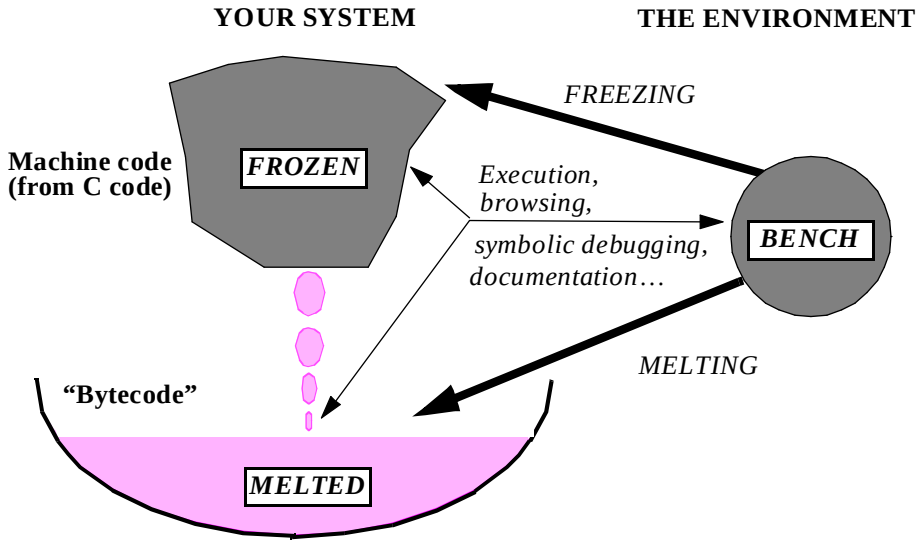
C3 • A third requirement, which appeared more recently, is quickly becoming important: the need to support the fast delivery of applications through the Internet to users or potential users, for immediate execution.

The first two requirements, in particular, are hard to reconcile. C1 is usually addressed through extensive compiler optimizations that make the recompilation and linking process prohibitively long. C2 is well served by interpretive environments, which execute software on-the-fly with little or no processing, but to obtain this result they must sacrifice execution-time performance (C1) and static type checking.

The Melting Ice Technology

The compilation technology that deals with the preceding issues, known as the *Melting Ice Technology*, uses a mix of complementary techniques. Once you have compiled a system, it is said to be *frozen*, like a block of ice stored in the freezer. As you take out the system to start working on it (so the metaphor goes), you produce some heat; the melted elements represent the changes. Those elements will **not** cause a compile-link cycle, which would defeat the goal of fast recompilation (C2); the melted code is, instead, directly executable by the environment's execution engine.

The Frozen and the Melted



The tricky part (for the compiler implementers) is of course to make sure that the various components can work together, in particular that frozen code can call melted elements — even though it was not known, at freezing time, that they would later be melted! But the result is definitely worthwhile:

- Recompilation is fast. The waiting time is typically a few seconds.
- This is still a compilation approach: any recompilation will perform full type checking (without undue penalty on recompilation time because the checking, like the recompilation in general, is incremental: only the changed parts are rechecked).

- Run-time performance remains acceptable because for a non-trivial system a typical modification will only affect a small percentage of the code; everything else will be executed in its compiled form. (For maximum efficiency, you will use the *finalization* form of compilation, as explained below.)

As you perform more and more changes, the proportion of melted code will grow; after a while the effect on performance, time and space, may become perceptible. So it is wise to re-freeze every few days. Because freezing implies a C-compilation and linking, the time it takes is typically more on the order of minutes (or even an hour after several days of extensive changes). You can start this task in the background, or at night.

Dependency analysis

As should be the case in any modern development environment, the recompilation process is automatic; you will simply click on the Melt button of the Project Tool, in the interface described below, and the compiling mechanisms will silently determine the smallest set of elements that need to be recompiled; there is no need for "Make files" and the notation has no notion of "include file".

To compute what needs to be recompiled, the environment's tools first find out what you have changed, either from within the environment, using its own class editor, or through outside tools such as text editors (each class text being stored in a file, the time stamps provide the basic information). Then they use the two dependency relations, client and inheritance, to determine what else may have been affected and needs recompilation. In the client case, information hiding is an important help to minimize propagation: if a change to a class only affects secret features, its clients do not need recompilation.

To reduce melting time further, the grain of recompilation is not the class but the individual routine.

> Note that if you add an external element, for example a C function, a freeze will be required. Again this will be determined automatically.

Precompilation

In accordance with the method's emphasis on reusability, it is essential to allow software developers to put together carefully crafted sets of components — libraries —, compile them once and for all, and distribute them to other developers who will simply include them in their systems without having to know anything about their internal organization.

The precompilation mechanism achieves this goal. A special compilation option generates a compiled form of a set of classes; then it is possible (through the Ace file) to include a precompiled library in a new system.

On Ace files see "Assembling a system", page 198.

There is no limit to the number of precompiled libraries that you may include in a new system. The mechanism that combines precompiled libraries supports sharing: if two precompiled libraries B and C both rely on a third one A (as with the Vision graphical library and the Net client-server library, discussed later, which both rely on the Base libraries for data structures and fundamental algorithms), only one copy of A will be included provided both B and C use the same version of A.

"Formats for reusable component distribution", page 79.

The author of a precompiled library may want to prevent his customers from having access to the source code of the library (an early chapter discussed the pros and cons of this policy). It is indeed possible, when precompiling, to make the source code inaccessible. In that case users of the environment will be able, through the visual tools described later in this chapter, to browse the *short form* and the *flat-short form* of the library's classes, that is to say their interface (public) properties; but they will not be able to see their full text, let alone their flat form.

Remote execution

At the time of writing the plug-in mechanism has not yet been released.

The interpretive code generated by melting — conventionally known as *bytecode* and identified as such on the preceding figure — is platform-independent. To execute bytecode, it suffices to have a copy of the environment's Execution Engine, known as 3E and freely downloadable through the Internet.

By adding 3E as a **plug-in** to a Web browser, it will be possible to make code directly executable: if a browser's user clicks on a hyperlink corresponding to bytecode, 3E will automatically execute the corresponding code. This is the remote execution mechanism first popularized by Java.

3E actually comes in two flavors, distinguished by the accompanying precompiled libraries. The first, secure, is meant for Internet usage; to avoid security risks it only allows input and output to the terminal. The second, meant for Intranet (corporate network) usage, supports general I/O and other precompiled libraries.

> An effort is also in progress to translate the bytecode into Java bytecode, to offer the supplementary possibility of executing the result of a development using a Java virtual machine.

Optimization

To generate the best possible code — goal C1 of the earlier discussion — frozen mode is not sufficient. Some crucial optimizations require having a complete, stable system:

- *Dead code removal* removes any routines that can never be called, directly or indirectly, from the system's root creation procedure. This is particularly important if you rely on many precompiled libraries, of which your system may only need a subset; a space gain of 50% is not uncommon.

"Static binding as an optimization", page 511 (also discusses inlining).

- *Static binding* which, as we studied in detail in the discussion of inheritance, should be applied by the compiler for features that are not redefined, or non-polymorphic entities.
- *Routine inlining*, also subject to compiler algorithms.

When you are still changing your system, these optimizations are not applicable, since your next editing move could invalidate the compiler's work. For example by adding just one call you may resuscitate a supposedly dead routine; by adding a routine redefinition, you cause a statically bound routine to require dynamic binding. Besides, such optimizations may require a complete pass through a system, for example to determine that no class redefines a certain routine; this makes them incompatible with incremental development.

As a result, these optimizations are part of a third form of compilation, **finalization**, complementing the other two (melting and freezing). For a large system finalization can take a few hours; but it leaves no stone unturned in removing anything that will not be needed and speeding up everything that is not optimal. The result is the most efficient executable form of the system.

The obvious opportunity for finalization is the delivery of a system, for a final or intermediate release. But many project leaders like to finalize once a week, at the time of the latest integration.

36.4 TOOLS

The figure on the facing page shows the general organization of the environment. The environment is of course used to bootstrap itself, and is written in the O-O notation (except for some elements of the runtime system, discussed next); this makes it an excellent testbed of the technology, and a living proof that it does scale up to large, ambitious systems (which, of course, we would not want to develop in any other way!).

Bench and the development process

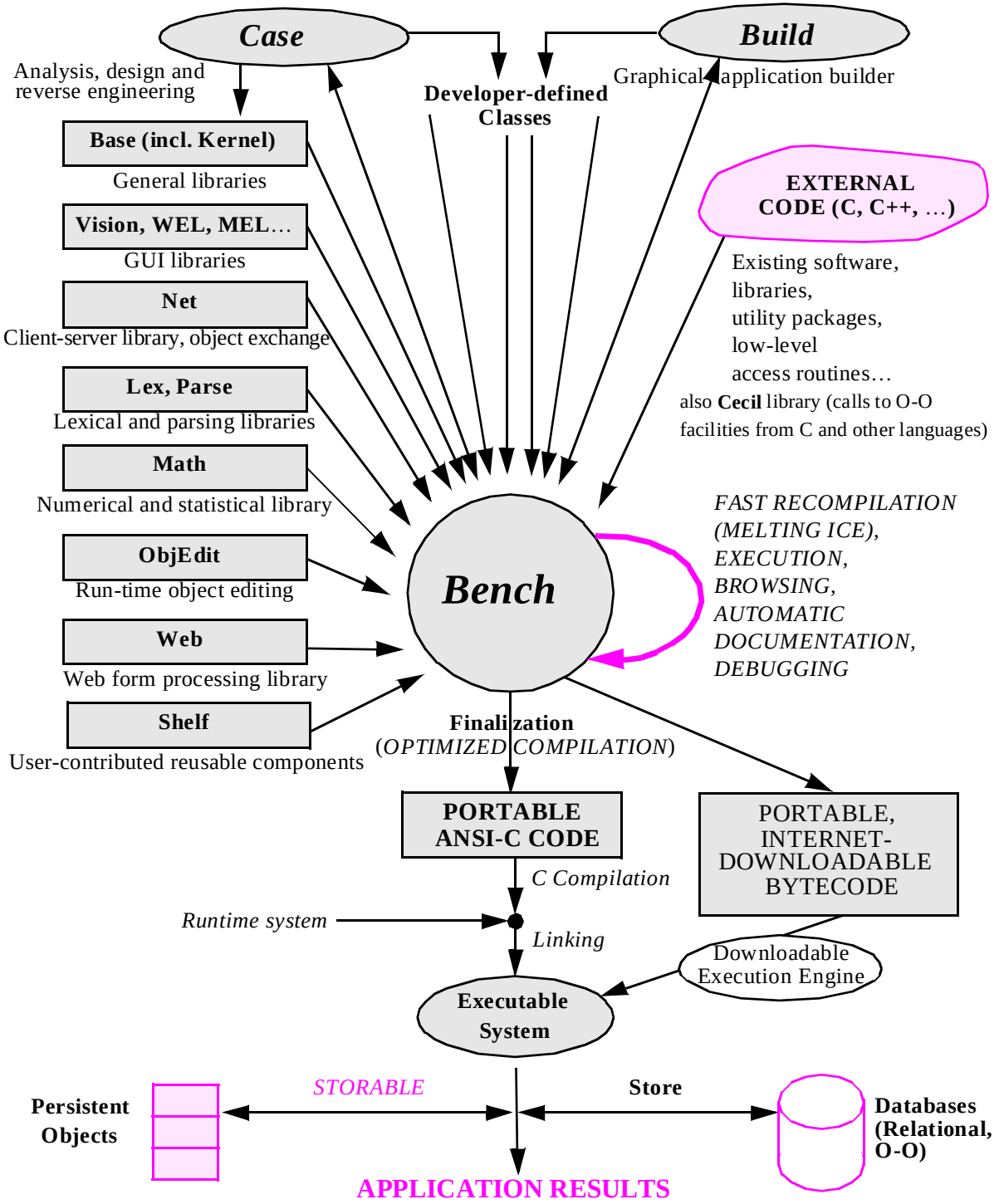
The centerpiece is *Bench*, the graphical workbench for compilation, browsing (exploring classes and features), documentation, execution, debugging. When developing a system you will constantly interact with Bench. For example you can melt the latest version by clicking on the Melt button of the Bench's Project Tool, shown below.

As long as you are melting and freezing you can stay within Bench. When you finalize a system — also by clicking on a button, although for this operation and many others non-graphical commands are also available — the outcome will be a C program, which the environment will compile to machine code for your platform by calling the appropriate C compiler. Freezing too relies on C as intermediate code. The use of C has several benefits: C is available on just about every platform; the language is sufficiently low level to provide a good target format for a compiler; C compilers perform their own extensive optimizations. Two further advantages deserve emphasis:

- Thanks to C generation you can use the environment as a **cross-development** platform, by compiling the generated C on another platform. This is particularly useful for the production of embedded systems development, which typically uses a different platform for development and for final execution.
- The use of C as compilation technology helps implement the openness mechanisms discussed earlier, in particular the interfaces to and from existing software written in C and C++.

Finalized C code, once compiled, must be linked; at this stage it uses the **runtime system**, a set of routines providing the interface with the operating system: file access, signal handling, basic memory allocation.

> In the case of cross development of embedded systems, it is possible to provide a minimum form of the runtime, which, for example, does not include any I/O.


Case
Analysis, design and reverse engineering
Build
Graphical application builder
Developer-defined Classes
Base (incl. Kernel)
General libraries
Vision, WEL, MEL…
GUI libraries
Net
Client-server library, object exchange
Lex, Parse
Lexical and parsing libraries
Math
Numerical and statistical library
ObjEdit
Run-time object editing
Web
Web form processing library
Shelf
User-contributed reusable components
EXTERNAL CODE (C, C++, …)
Existing software, libraries, utility packages, low-level access routines…
also Cecil library (calls to O-O facilities from C and other languages)
Bench
FAST RECOMPILATION (MELTING ICE), EXECUTION, BROWSING, AUTOMATIC DOCUMENTATION, DEBUGGING
Finalization
(OPTIMIZED COMPILATION)
PORTABLE ANSI-C CODE
C Compilation
Runtime system
Linking
PORTABLE, INTERNET-DOWNLOADABLE BYTECODE
Downloadable Execution Engine
Executable System
Persistent Objects
STORABLE
Store
Databases (Relational, O-O)
APPLICATION RESULTS

High-level tools

At the top of the figure on the preceding page, two high-level generation tools appear.

Build is an interactive application generator based on the Context-Event-Command-State model developed in an earlier chapter. You can use it to develop GUI (Graphical User Interface) applications graphically and interactively.

Chapter 32.

Case is an analysis and design workbench which provides the ability to reason on systems at a high level of abstraction, and through graphical representations. In accordance with the principles of **seamlessness** and **reversibility** introduced in the discussion of the software process, Case allows you both to:

Chapter 28.

- Devise system structures through graphical interaction — to produce visual representations of classes ("bubbles"), specify their relations through client and inheritance arrows, and group them into clusters —, relying on Case to generate the corresponding software texts in the end (**forward engineering**).
- Process existing class texts to produce the corresponding graphical representations, to facilitate exploring and restructuring (**reverse engineering**).

Particular attention has been devoted to making sure that developers can freely alternate between forward and reverse engineering. In particular, you can make changes on either the graphical or the textual form; Case provides a **reconciliation** mechanism which will merge the two sets of changes and, in case of conflicts, take you through a step-by-step decision process in which you will see the conflicting versions of a feature and choose, in each case, the version to be retained. This part of the tool is key to ensuring true reversibility, letting developers decide at each stage the level of abstraction and the notation, graphical or textual, that they find most appropriate.

The conventions of Case are drawn from the Business Object Notation described in an earlier chapter. BON supports in particular the tools' facilities for abstraction and **zooming**: it is essential, for large systems, to enable developers to work on an entire system, on a subsystem, on just a small cluster, choosing the exact level of abstraction they desire.

"THE BUSINESS OBJECT NOTATION", 27.7, page 919.

An example Case screen appears at the top of the facing page, showing a cluster from a chemical plant description, the properties of one of its classes (*VAT*), and the properties of one of the features of that class (*fill*).

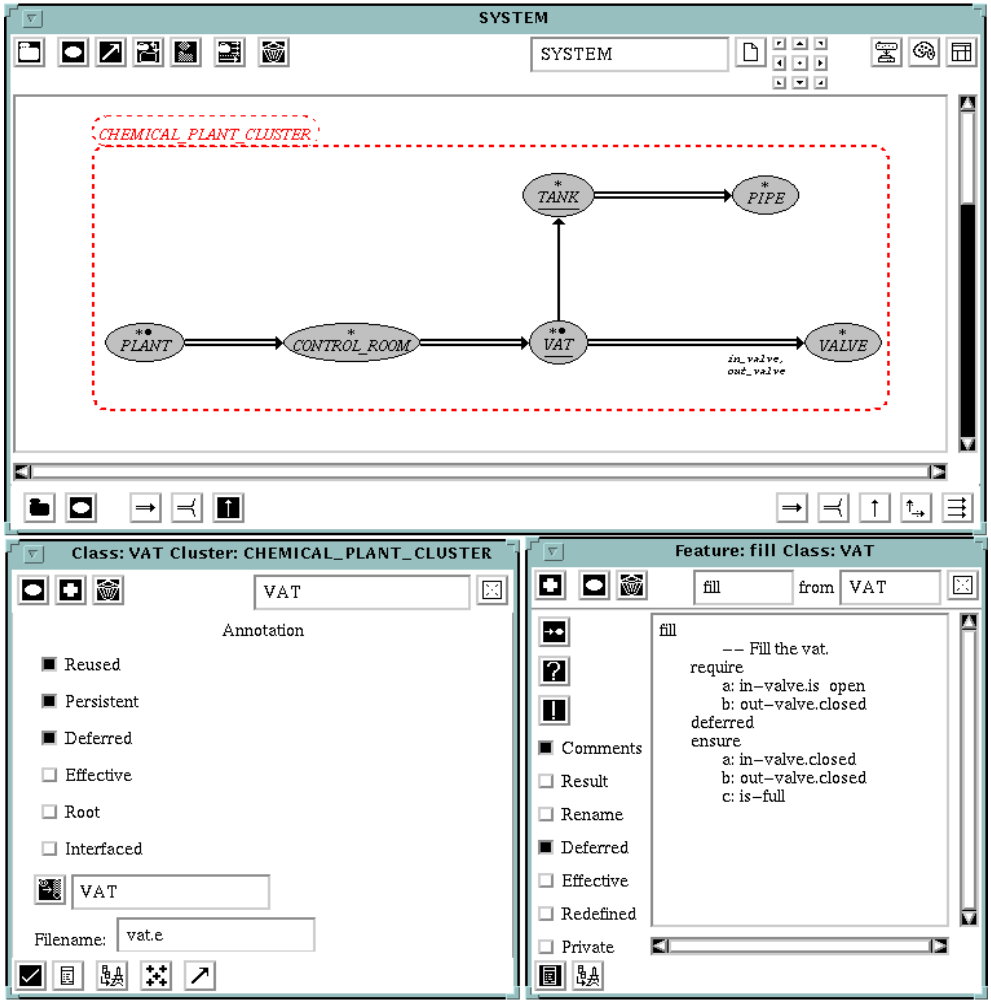
36.5 LIBRARIES

A number of libraries appear on the general environment diagram of the preceding page. They play a considerable role in the software development process, providing developers with a rich set (several thousand classes) of reusable components. They include:

- The *Base* libraries, about 200 classes covering the fundamental data structures (lists, tables, trees, stacks, queues, files and so on). The most fundamental classes make up the *Kernel* library, governed by an international standard (ELKS).
- The graphical libraries: *Vision* for platform-independent GUI development; *WEL* for Windows, *MEL* for Motif, *PEL* for OS/2-Presentation Manager.

A cluster, class and feature under Case

(*Here on a Sparc-station with Motif, but versions exist for Windows and other look-and-feel variants.*)



"Storable format variants", page 1038.

- *Net*, for client-server development, allowing the transferral of arbitrarily complex object structures over a network; the platforms may be the same or different (under *independent_store* the format is platform-independent).

- *Lex*, *Parse* for language analysis. Parse, in particular, provides an interesting approach to parsing, based on a systematic application of object-oriented concepts to parsing (each production modeled by a class; see the bibliographical notes). A supporting public-domain tool, YOOCC, serves as front-end for Parse.

"Object-oriented re-architecturing", page 441.

- *Math* is a numerical library providing an object-oriented view of the fundamental techniques of numerical computation. It is based internally on the NAG library and covers a large set of facilities. Some of its concepts were presented in an earlier chapter as an example of O-O re-architecturing of non-O-O mechanisms.

- *ObjEdit* provides facilities for editing objects interactively during execution.
- *Web* supports the processing of forms submitted by visitors to a Web site, advantageously replacing the Perl or C "CGI scripts" sometimes used for this purpose.

The bottom part of the environment diagram shows libraries used for taking care of persistence needs during execution: the *STORABLE* class and a few complementary tools, discussed in earlier chapters, support storage, retrieval and network transmission of object structures, self-contained through the application of the Persistence Closure principle; and the *Store* library is the database interface, providing mechanisms for accessing and storing data in relational databases (such as Oracle, Ingres, Sybase) and object-oriented databases.

This list is not exhaustive; other components are under development, and users of the environment have provided their own libraries, either free or commercial.

A particularly interesting combination is the use of *Net*, *Vision* and *Store* for building client-server systems: a server can take care of the database aspects through Store, and of the heaviest part the computation (possibly using Base, Math etc.); lean clients that only handle the user interface part can rely on Vision (or just one of the platform-specific libraries), and include little else.

36.6 INTERFACE MECHANISMS

To support the preceding concepts, the environment provides a visual interface, based on an analysis of the needs of developers and of the requirements of various platforms.

This brief presentation will only mention some of the most original aspects of the environment. Ample literature (see the bibliographic notes) is available on its other facilities; the reader familiar with other modern development environments will have no difficulty guessing some of the tools and possibilities not described here.

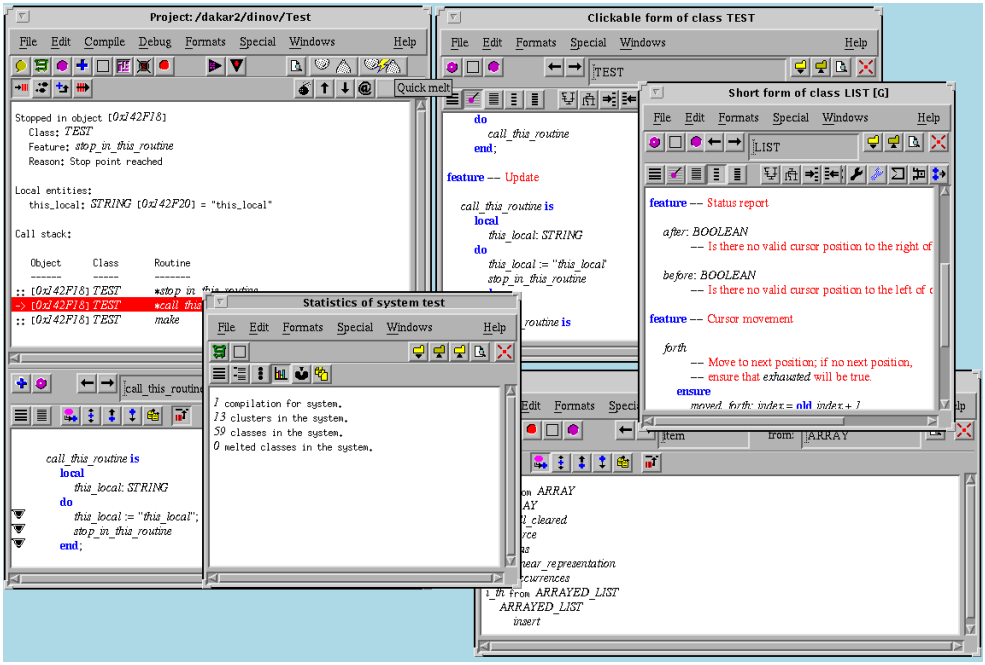
Platforms

The screenshots that follow were drawn from a session on a Sun Sparcstation, for no other reason than convenience. Other platforms supported at the time of writing include Windows 95 and Windows NT, Windows 3.1, OS/2, Digital's VMS (Alpha and Vax) and all major brands of Unix (SunOS, Solaris, Silicon Graphics, IBM RS/6000, Unixware, Linux, Hewlett-Packard 9000 Series etc.).

Although the general concepts are the same on every platform, and the environment supports source-code compatibility, the exact look-and-feel adapts to the conventions of each platform, especially for Windows which has its own distinctive culture.

The following screenshot shows a set of environment windows during a session. Although printed in black and white in this book, the display makes extensive use of colors, especially to distinguish the various parts of class texts (the default conventions, user-changeable, are keywords in blue, identifiers in black, comments in red).

Tools



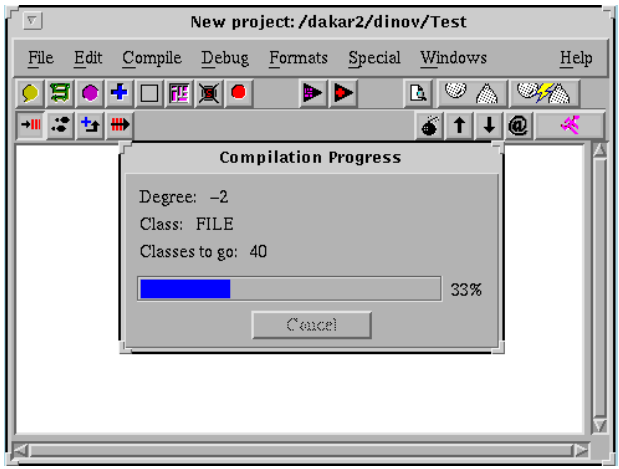
Tools

An environment consists of tools. In many cases those tools are *functional* tools, in the sense of being devoted to functions: a browser tool to browse, a debugger tool to debug, a pretty-printer tool to produce formated versions of software texts. A recent environment such as Sun's Java Workshop (as demonstrated in September of 1996) still conforms to this traditional pattern; to find the ancestors of a class (its parent, grandparent etc.) you start a special "browser" tool.

The disadvantage of this approach is that it is *modal*: it forces you to select first what you want to do, then what you want to do it to. The practice of software development is different. During the course of a debugging session, you may suddenly need a browsing facility: for example you discover that a routine causing trouble is a redefined version, and you want to see the original. If you see that original you may next want to see the enclosing class, its short form, and so on. Modal environments do not let you do this: you will have to go away from the "debugger tool" to a "browser tool" and restart from scratch to look for the item of interest (the routine) even though you had it in the other window.

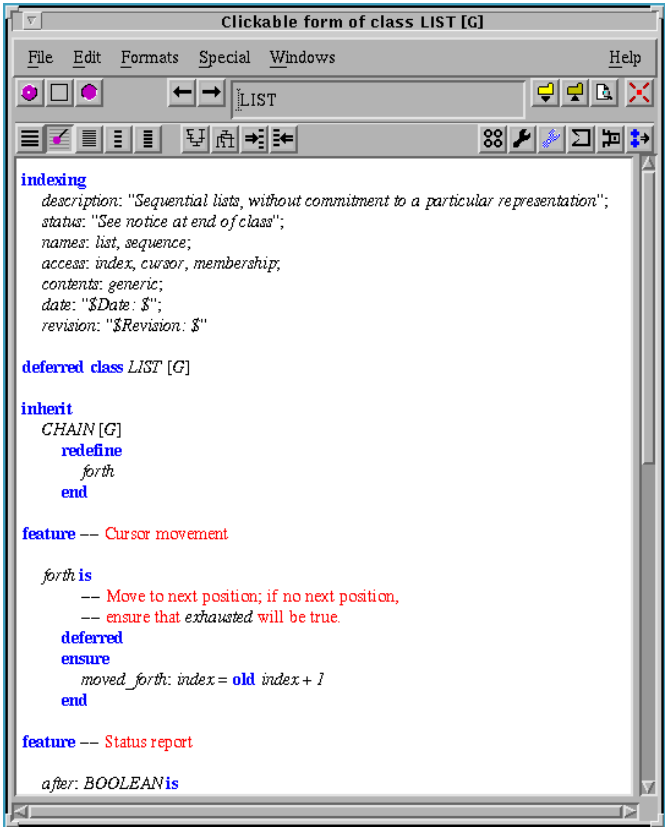
Here too the object-oriented method provides a better approach. In the same way that we learned to trust object types rather than functions to define our software architectures, we can base our tools on the type of **development objects** that developers manipulate. So we will have no debugger or browser window, but instead a Class Tool, a Feature Tool, a System Tool, a Project Tool, an Object Tool, corresponding to the abstractions that O-O software developers deal with day in and day out: classes, features, systems (assemblies of classes), projects, and, at run-time, class instances ("objects" in the strict sense).

A Project Tool, for example, will keep track of your overall project. You use it among other applications to perform a Melt, a Freeze or a Finalize; here is a Project Tool captured during a compilation, with a progress bar showing the percentage done:



Project Tool during a compilation

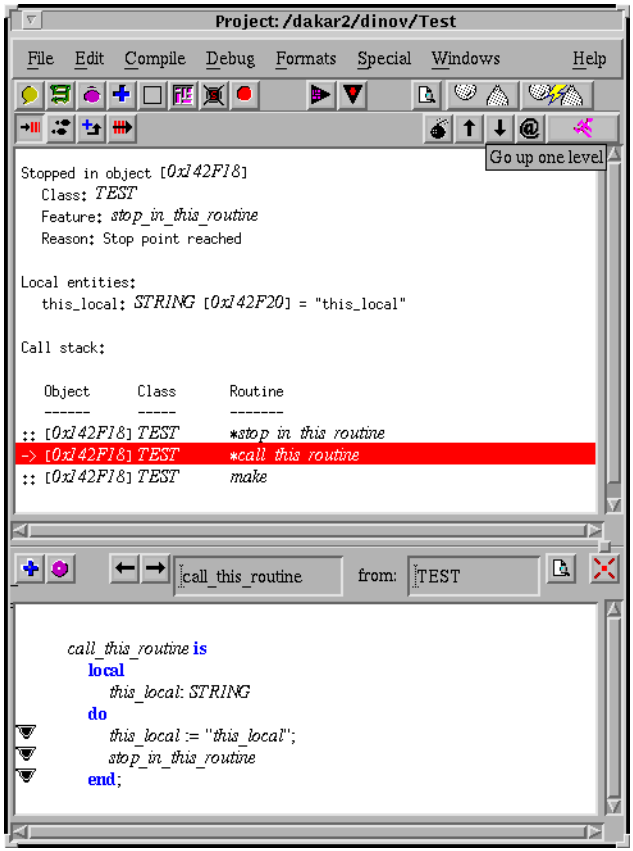
A Class Tool will be **targeted** to a particular class such as *LIST*:



A Class Tool in default format

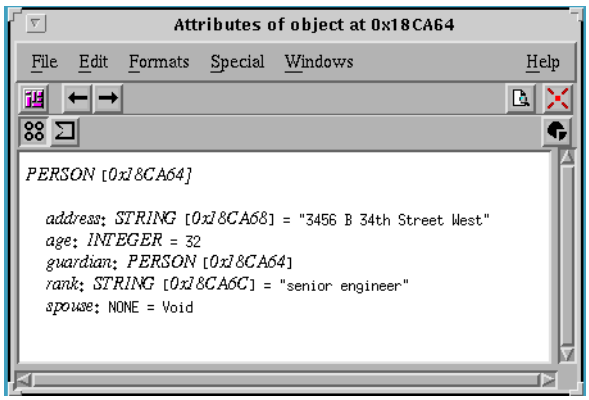
A Feature Tool, here attached to a Project Tool as part of a debugging session, shows both a feature and the progress of the execution, with mechanisms for step-by-step execution, displaying the call stacks (see the local entities' values in the Project Tool). The Feature Tool is targeted to feature *call_this_routine* of class *TEST*.

Project and Feature Tool for debugging



During an execution, you can also see an individual object through an Object Tool:

An object and its fields captured during execution



This shows the various fields of the objects. One of them, *guardian*, denotes a non-void reference; you can see the corresponding object by following the link, as we will shortly see.

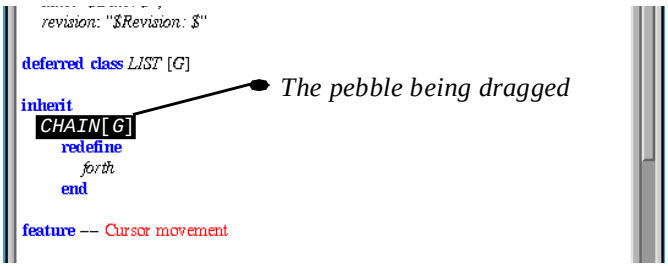
You can of course have as many Class Tools, Feature Tools and Object Tools as you like, although there is only one System Tool and one Project Tool during a session.

Retargeting and browsing

Various techniques are available to change the target of a tool, for example to retarget the preceding Class Tool from *LIST* to *ARRAY*. One way is simply to type the new class name in the corresponding field (possibly with wild card characters as in *ARR**, to get a menu of matching names if you do not exactly remember).

But you can also used the **pick-and-throw** mechanism briefly introduced in an earlier chapter. If you right-click on a class name, such as *CHAIN* in the Class Tool targeted to *LIST*

See the figure entitled "Pick-and-throw", page 534



Typed pick-and-throw

the cursor changes into a **pebble** of elliptical form, indicating that what you have picked is a class. The ellipse corresponds to the form of the class **hole**  ; find the Class Tool that you want to retarget (the same or another), and drop the pebble into the hole, by right-clicking into it, to retarget the tool to the chosen class. For convenience you can actually drop it more or less anywhere into the tool, globally considered as a big hole. Rather than pick, drag and throw, you can control-right-click on an object — class, feature… — to start a new tool of the appropriate type, targeted to the object.

The pick-and-throw mechanism is a generalization of common drag-and-drop. Instead of having to maintain the button pressed, however, you work in three steps: the first right-click selects the object; you release the button immediately. Then you are in drag mode, where moving the mouse will cause the line attached to the original element (as on the above figure) to follow the pebble. Finally, you right-click again in the destination hole. This has three advantages over common drag-and-drop:

- Having to keep the mouse button pressed during the whole process, although acceptable for occasional drag-and-drop operations such as moving an element in an interface builder, can cause considerable muscle fatigue at the end of the day when you use it frequently.

- It is all too easy to slacken off the pressure for a split second and drop on the wrong place, often with unpleasant or even catastrophic consequences. (This has happened to me on Windows 95 while drag-and-dropping an icon representing a file; I involuntarily dropped it at a quite unintended place and had a hard time finding out what the operating system had done with the file.)

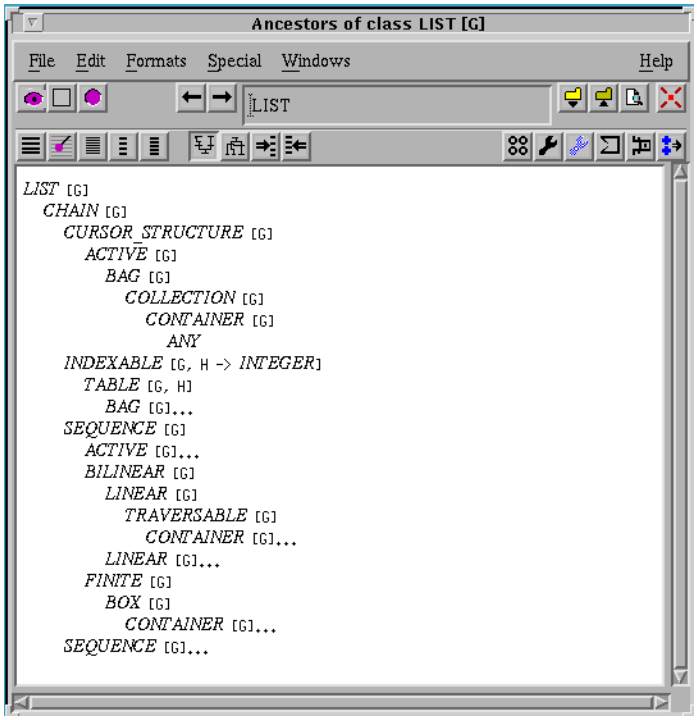
- Common drag-and-drop does not let you cancel the operation! Once you have picked an object, you *must* drop it somewhere; but there may not be such an acceptable somewhere if you have changed your mind. With the pick-and-throw mechanism, a left-click will cancel the entire operation at any time before throwing.

- Also note that the mechanism is **typed**: it will only let you drop a pebble into a matching hole. There is some tolerance: in the same way that polymorphism lets you attach a *RECTANGLE* object to a *POLYGON* entity, you can drop a feature pebble into a class hole (and see the enclosing class, with the feature highlighted). Again the environment's interaction mechanisms directly apply, for convenience and consistency, the concepts of the method. (Here the difference with common drag-and-drop mechanisms is not crucial, as some of them do have a limited form of typing.)

These, however, are just user interface issues. More important is the role of pick-and-throw, combined with other mechanisms of the environment, to provide an integrated set of mechanisms for all tasks of software development. If you look back at the Class Tool targeted to *LIST*, a deferred class from the Base libraries, you will note a row of format buttons (the second row of buttons from the top). They include:

- Class text  .
- Ancestors  .
- Short form  .
- Routines  .
- Deferred routines  .

and so on. Clicking on one of them will display the class text in the corresponding format. For example if you click on Ancestors the Class Tool will display the inheritance structure leading to *LIST* in the Base libraries:



The ancestry of a class

In such a display, as in every other tool display, **everything of importance is clickable**. This means that if for example you notice class *CURSOR_STRUCTURE* and want to learn more about it, you can just right-click on it and use pick-and-throw to retarget this tool, or another, to the chosen class. Then you can choose another format, such as Short Form. If in that format you see the name of an interesting routine, you can again apply pick-and-throw to target a Feature Tool to it. In the Feature Tool, the available format buttons include **history** which shows all the adventures of a feature in the inheritance games: all the versions it has in various classes, after renaming, effecting, redefinition; and whenever it lists a class or a feature in showing this information, the environment will let you pick-and-throw the element.

Similarly, the debugging session shown earlier showed class and feature names in various places; to find out information on any of them, just use pick-and-throw. To see an object, such as *0X142F18* on the previous example (an internal identifier, by itself meaningless but clickable), control-right-click on it to start an Object Tool similar to the one we saw, displaying an instance of *PERSON*. In that tool, all fields are identified by their class names — clickable — and references are also clickable, so that you can easily explore the run-time data structures, however complex.

For each of the available formats, you can produce output in various forms such as HTML, TEX, Microsoft's Rich Text Format, FrameMaker MML, troff and so on (a small descriptive language enables you to define your own output forms or adapt an existing one). The output can be displayed, stored with the class files, or, if you want to produce on-line documentation for an entire project or cluster, stored in a separate directory.

These browsing mechanisms do not make any difference between built-in libraries and developer-defined classes. If an element of your software uses *INTEGER*, you can just control-right-click or use pick-and-throw to see that basic class in a Class Tool, in any available format. (As noted, the author of a precompiled library may elect to make the source unavailable, but you will still have access to the short and flat-short forms, with usual clickability properties.) This is of course in line with this book's general principle of uniformity and seamlessness, attempting as much as possible to use a single set of concepts throughout software development activities.

> In contrast, I tried in the aforementioned demo of Java Workshop to get some information about a redefined feature of a certain class, picked at random, but was told that there was no way the "browser tool" could handle that feature, since it turned out to come from a class of the predefined graphical library. The only way to get any information at all was to go to another tool and bring up the documentation — which had a one-line description of the feature. (*INTEGER* would probably also not be browsable since basic types are not classes in Java.)

The run-time mechanisms, in particular the debugging facilities (single-stepping, stop points and so on) all follow from these basic concepts. For example to put a stop point on an instruction or a routine you just drag-and-drop the chosen stop point location to a Stop Point hole  .

Some holes, known as "buttonholes", double up as buttons. For example clicking on a Stop Point hole, treated as a button, will display in the Project Tool information about all the currently active stop points; such information being again clickable, you can easily remove existing stop points or add new ones to the list.

The systematic application of these techniques makes up a mechanism for proximity browsing where everything of interest is hyperlinked — far preferable, in my experience, to modal environments which force you to ask at each step "Am I browsing? Oh no, I am debugging, so I must start a browser tool. And what tool should I start to get the class documentation?".

You are neither debugging nor browsing nor documenting nor editing; you are using and building software, and the tools should let you do what you want on all the objects you want, at any time you want.

36.7 BIBLIOGRAPHICAL NOTES

For an up-to-date summary of the benefits of the environment see [M 1996b], also available on line [M-Web] along with many other technical documents and descriptions of actual projects.

A collective volume describing a set of industrial applications produced with the environment over the years, whose chapters are written by the project leaders in the companies involved, was published as [M 1993].

Among the publications that have described various aspects of the environments at successive stages of its evolutions are: [M 1985c], [M 1987b], [M 1987c], [M 1988], [M 1988a], [M 1988d], [M 1988f], [M 1989], [M 1993d], [M 1997].

The reference on the language is [M 1992]. The book *Reusable Software* [M 1994a] contains, along with a discussion of library design principles, a detailed description of the Base libraries.

Another book [M 1994] presents the environment as a whole. [M 1995c] describes the Case analysis and design workbench, and [M 1995e] the Build graphical application builder. The interface principles were presented in [M 1993d].

The YOOC compiler generator was developed by Christine Mingins, Jon Avotins, Heinz Schmidt and Glenn Maughan of Monash University [Avotins 1995] and is available from Monash's FTP site. The object-oriented parsing techniques of the underlying Parse library, initially presented in [M 1989d], are covered in [M 1994a].

The Math library was developed by Paul Dubois and is described in [Dubois 1997].

Many people have participated in the development of the environment. Some of the principal contributions are due to Éric Bezault (to whom I am also grateful for proofreading parts of this book), Reynald Bouy, Fred Deramat, Fred Dernbach (who built the original architecture of the current compiler), Sylvain Dufour, Fabrice Franceschi, Dewi Jonker, Patrice Khawam, Vince Kraemer, Philippe Lahire, Frédéric Lalanne, Guus Leeuw, Olivier Mallet, Raphaël Manfredi (who established the basis for the current runtime system), Mario Menger, Joost De Moel, David Morgan, Jean-Marc Nerson (especially for the initial versions), Robin van Ommeren, Jean-Pierre Sarkis, Glen Smith, Philippe Stephan (who originated many of the interface principles), Terry Tang, Dino Valente, Xavier Le Vourch, Deniz Yuksel. It is impossible to cite even a small part of the environment users who also helped through their feedback and suggestions