Palo Alto Research Center

UNIX Needs A True Integrated Environment: CASE Closed

Mark Weiser, L. Peter Deutsch and Peter B. Kessler

XEROX

UNIX Needs A True Integrated Environment: CASE Closed

Mark Weiser, L. Peter Deutsch*, and Peter B. Kessler

CSL-89-4 January 1989 [P89-00004]

Abstract: Long before there was CASE as we now know it, there were integrated programming environments for languages like Smalltalk and Lisp. Why are there no truly integrated environments for UNIX? And why is CASE not enough?

A version of this paper appeared as *Toward a Single Milieu* in *UNIX REVIEW*, 6(11), November, 1988.

CR Categories and Subject Descriptors: D.2.6 **[Programming Environments]:** Interactive; D.2.1 **[Design]:** Methodologies; K.6.3 **[Software Management]:** Software development.

General terms: Design.

Additional Keywords and Phrases: Programming Environments, Computer Aided Software Engineering (CASE), Integration, Interoperability.

*L. Peter Deutsch is now Chief Scientist with ParcPlace Systems, Palo Alto, CA.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Introduction

For the computer programmer in search of a comfortable place to program, UNIX* historically has been not a bad place to settle down. Its long and strong tradition, its well-known functionality, and its growing popularity have helped to make UNIX a productive environment for many a software developer.

As good as many perceive it to be, though, UNIX could afford to be made more comprehensive and functional as a software development environment. Working to this end, some system software engineers have considered how Computer-Aided Software Engineering might apply. To software developers who work on UNIX systems, however, there are at least three general limitations to the CASE approach.

One is that CASE, while not without its virtues, is primarily concerned not with programming per se, but with the *management* of programming--and the distinction between these two is not a trivial one. Second, it's been our experience that existing CASE systems are almost never well integrated into the existing UNIX system, but are rather systems unto themselves. And third, CASE, when it involves executable programs at all, usually covers only a single programming language, and is not in the UNIX spirit of plugging together applications.

There is another tradition, though, distinct from both UNIX and CASE in its growth, that can shed light on ways in which UNIX as a programming environment has been found wanting. "Integrated programming environments" (IPEs) are tools for programmers themselves to use while programming large or small systems [1]. As IPEs share this philosophical attribute with UNIX, there has been activity under way for some time to make UNIX into an IPE. But UNIX presents an unusual challenge for integrated programming environments in that one of the strengths of UNIX is its many languages, and IPEs--bearing at least this one similarity with CASE--are usually single-language systems.

In this article, we'll look at some of the issues involved in supporting integrated environments, and at some of the features they provide. We'll then discuss CASE systems and how, because their goals are distinct from those of integrated environments, they fail to be IPEs. Finally, we'll consider the challenges of creating an integrated environment for all of UNIX and try to guess at what it would take to build such an environment.

What is an Integrated Programming Environment?

The major feature of an integrated programming environment is that any program within it can call (and be called by) any other program within it. In the design of such environments, the decision to ensure such interoperation has the effect of placing certain responsibilities on programmers, but in

* UNIX is a trademark of AT&T Bell Laboratories.

return gives them advantages over other systems in which programs are only used by non-programming users. First and foremost, then, an IPE must provide tools that encourage this kind of *integration.*

The most dramatic effect of integration is the resulting ubiquity of the best of each kind of tool a system has to offer. For example, in the Cedar system--developed at the Xerox Corp. Palo Alto Research Center (Xerox PARC)--the same fully-fonted WYSIWYG editor is used everywhere in the environment that typing is allowed [2]. This pervasiveness has the advantage of giving users the ability to issue any editing command from any context in which they are typing (including the fill-in fields of the user's interface to the editor!). The availability of the editor throughout the system means that editing operations that work in one context will work in every context, so a user spends less time remembering tool-specific editing commands and more time doing useful work.

What is required to permit the widespread use of the editor? From a client programmer's point of view, it must be at least as easy to get input from the editor as it is to get it directly by using I/O primitives. In a UNIX environment, for instance, if you want to get the editing features embedded in the *tty* driver, it's easier to work through *pty*s than to try to recreate all the input processing code. From the point of view of the programmer of the editor, an interface must be provided so that the editor can be driven by programs at least as well as it can be driven by its user interfaces. This is what IPE's do. Every Cedar program can easily access multiple copies of the common editor.

A benefit of programming interfaces is that code gets reused, and everyone wins. In the example of the integrated editor, performance improvements of extensions to the editor are available to all clients. It's untenable to think that programs which copied the input editing code from the *tty* driver should all be responsible for tracking changes in order to present a uniform appearance to the user.

An orthogonal aspect of code reuse is whether the environment is seamless--that is, whether a program can call on fine-grained components of other applications and the kernel, or if only the large-grained facilities of other applications and the kernel can be invoked. For example, in Smalltalk [3], you can invoke methods that in UNIX would involve pieces of the kernel that are not accessible to application programs.

If a programming environment is to support integration and universal program access, it must also provide services in support of programming--and provide them to programmers and programs alike. Programmers must be able to browse through code to find functions to call (rather than constantly being required to reinvent them). The programming environment must be able to answer questions such as: "Given an existing program that does something similar to what a prospective program needs to do, how does the existing one do it?" Such a map from executable images back to source code is also necessary for reporting runtime errors. Similarly, a programmer may want to know what programs call a given piece of code so that changes can be made transparently. The Masterscope facility of Interlisp [4] is an inspiration for all such facilities. In UNIX, the way one finds all callers of a function is to **grep** over the sources for an application. That won't work when code is shared among (or available to) lots of programs.

An integrated programming environment should be able to map from source location to execution location to set breakpoints. In UNIX, the compilers and debuggers can answer such questions about C code [5] (with more or less accuracy, depending on whether you compiled with **-g** and installed without **-s**), but not about **awk** programs or shell scripts--and even the questions about C are complicated by such things as dynamic linking and shared libraries.

This variety of services included in an integrated environment are certainly easier to provide in single-language systems. Multi-language environments, in which the registered services of different languages are provided to browsers, are still a research topic. We'll return later to the issue of multi-language environments, but first we address in more detail how it is that IPEs and CASE are different animals.

Differences between CASE and IPE

The primary difference between CASE and IPEs is their difference in goals. Both concepts can be viewed as tools: CASE, although the claims made about it are often broader, is essentially a tool for management and the handling of paperwork; an IPE is primarily a tool to improve an individual programmer's productivity. In this sense, an IPE is more consistent with the UNIX tradition of making things easy for programmers; UNIX itself, in fact, could be considered a rudimentary IPE. CASE, on the other hand, seems designed to increase the productivity and success of large, paperwork-dominated projects at the expense of the individual programmer.

CASE and IPEs have three primary differences. The first has to do with the degree of integration of the environment's components. The second regards the degree of power granted the individual programmer. And the third relates to the extent of restrictions placed on the programmer. Let's look at each of these in turn.

Most CASE systems claim to be integrated--and so they are, in the sense that they connect various tools together and transform data formats within those tools. But IPEs are integrated in a different sense: everything is inside them and integrated into them. Any tool that works for analyzing an application can also be used for analyzing the tool itself--or for analyzing any part of the given machine's operating system.

For example, you might want to find all the procedures in your application that do I/O. This is a reasonable request of a CASE system, and many will help you answer it. But which CASE systems let you ask this question of the CASE system itself ("which CASE modules do I/O?")? And which let you ask it of the computer system on which the CASE software is itself running ("which parts of the UNIX kernel do I/O?")? No CASE system has this degree of integration. Nor do CASE packages necessarily claim this as a goal. Yet this important characteristic of IPEs is reminiscent of some of the important features in the spread of UNIX--the software is written in a high-level, portable language, and the source code of the system is easily browsed.

The second difference of CASE and IPEs is found in the degree of power that each gives the

individual programmer. CASE systems excel at supporting the non-coding parts of programming, such as designs and documentation. IPEs instead help to get the code written and working right, including the finding of pre-existing code to reuse and modify. CASE focuses on the large and the long-range matters of the software development problem, most of which do not involve actual programming. The IPE approach, on the other hand, is consistent with the UNIX approach of providing many basic tools to support programming, including (for UNIX) many different programming languages and "little languages" such as **awk** and **sed**.

The third way that CASE and IPEs differ has to do with the attitudes they reflect regarding restrictions. CASE systems are often full of restrictions, and proud of them. Their purpose is to "enforce" a particular design method, or "require" particular tools, or "automatically deliver" certain reports. IPEs are proud of the opposite: that they have no restrictions, that anything is possible, and that most things are easy. You say you need to intercept certain Ethernet packets and send them to a particular monitoring routine? In the Interlisp IPE, no problem: just "advise" the Lisp Ethernet function that when it is called with certain arguments, it is to "do the following...".

Of course, when anything is possible, it is possible to do great damage. This is what CASE tries to avoid (consistent with its focus on the large and the long-term), and what IPEs shrug off. "Power tools for power programmers" would be the IPE motto, and, of course, power tools in unskilled hands can be dangerous. Again we have an analogy with UNIX: what experienced UNIX programmer doesn't know three ways to bring the machine to its knees?

Towards an Integrated UNIX Environment

If IPEs are so great, why isn't there an IPE for UNIX? The short answer is: no one has ever done a multi-lingual IPE, and much of the power of UNIX comes from its multiple languages (especially its "little languages" as noted above). There are IPEs *on* UNIX, which are more or less integrated environments for one (or a few) particular languages that try to bring all of UNIX within that IPE (the accompanying article on the Smalltalk IPE tells of such a system). But these are not IPEs *for* UNIX, which would integrate existing UNIX tools and facilities as we have described above.

There are four things needed in an IPE that are difficult in UNIX. They are: locating things via connections; powerful user interfaces; system seamlessness; and incrementality. We now consider each of these individually.

Locating

Important in any IPE is the power of locating things. This is so for two related reasons. The first is the value of finding things to reuse in programs. Perhaps someone has already written a particular program that's now needed by someone else, or perhaps there are existing subroutines or system functions that would make a program's initial writing that much easier. The second reason that locating is important is the value of finding things related to other things already in hand. For instance,

if some object code is misbehaving, lets locate its source code. Or again, given a call on a system library, one can check against past cases to see if the library is being called correctly. These two reasons and the kinds of searches they imply are quite similar--they differ in that the first involves locating something based on its relationship to an idea in the programmer's mind, and the second involves locating something based on its relationship to something already present in the machine.

UNIX provides very little help for either kind of location. There is no standard way of remembering relations among source, object, derived source and object (which compiler, for example, generated a particular object), documentation, and the like. In many IPEs--such as Cedar--you can automatically deduce from an object module all the source versions that went into its making, including the versions of any functions that it calls and the version of the compiler used for its compilation. Various ad hoc systems for managing versions in UNIX, among them SCCS [6] and Tichy's RCS [7], have been invented, but none of them fully establishes the relationship chain among all parts of a system.

When taking the UNIX approach, if one is interested in locating things based on ideas in one's head, there is little more that one can do than read the complete manual from beginning to end. The **apropos** command available in some UNIX systems, which locates on-line manual pages based on a match of keyword and summary description, is often useful. And **grep**ing the manual page directories, or even the system source code, is certainly not unheard of. But these ad hoc practices merely help demonstrate how UNIX lacks a real IPE.

What is the fundamental limitation in current UNIX systems that makes locating things difficult? It is that there are no *things*, only *containers*. Suppose you have some source code in a file called *foobar.c*. Is *foobar.c* the name of the source? Well, no, it's the name of the container of the source, namely the file. If the source code is edited, the filename remains the same and does not reflect the change to the source. There is in UNIX no way to refer to the source code itself. Essentially, the source code has no real existence in UNIX. And this is true for all other important UNIX objects: object code, system data structures, documentation, and so on.

Naturally, names of containers serve a purpose. But there are times when it would be helpful to name the contents of containers as well. In the Cedar file system--among others--a filename refers to the contents of the file, and if the contents change, the filename changes as well (by changing a version number or a file timestamp). (Alternatively, you can refer to the container by using the filename without version number or timestamp.)

A usual objection to naming things instead of containers is simply that there are too many things; hence, the thinking goes, you will sooner or later run out of storage. But naming something and keeping it forever are two different things. If you have a long-lost cousin named Sue Perwillicker, that remains her name whether you ever find her again or not. Similarly, having the name of the source belonging to some object does not guarantee that the source is still around, but it does keep you from thinking the source is around when it isn't.

You can imagine a UNIX system that did allow the naming of things. In most cases, it would be

file contents that would cause us a problem, so let's just consider naming them (and leave the naming of system data structures for another time). Suppose every UNIX file had an extra unique number associated with it--a 64-bit hash of the contents with the last *write* time, say, such that this hash was always different from other hashes whenever the file was different from other files. The hash, together with the filename, would name the contents of the file.

Existing UNIX programs would not care about--or see--the hash: its presence would be upward-compatible. New programs that wanted to name contents--not containers--would use the filename and the hash, and so have a way to build up relationships among objects, not just containers. Just making the **-p** flag (for "preserve dates") the default for the **cp** command would go a long way toward establishing names for things. A system like this could still be called "UNIX". And until UNIX is able to name things, not just containers, it will be hard to build a true IPE that provides powerful locating.

User Interface

Associated with IPEs are powerful user interfaces. And synonymous with powerful user interfaces these days are window systems--the first of which was developed at Xerox PARC several years ago as part of the Smalltalk IPE. At present, UNIX has its own standard window system--in fact, it has several. Window systems have an importance all their own, but to have an IPE requires going beyond just windows to an *integration of function* within the windows. And for certain forms of such integration, we have not the barest beginnings in UNIX.

Editors in IPEs are one of their distinguishing--and sometimes frustrating--features. The structure editor in Interlisp is a gem once you get to know it, and a complete mystery to the novice. It's representative of a fairly familiar notion these days of a "structured" or "intelligent" editor--that is, an editor that itself provides the programmer with help regarding the syntax, and sometimes semantics, of the programming language being edited. It's hard enough to design such an editor in a single-language IPE, especially when you want to have the structure editor help you with the proper parameters for calling functions that may have only been entered into the system a minute ago. But when you want to use the same editor for Lisp, Smalltalk, C, Cedar, FORTRAN, Pascal, **yacc**, **awk**, **sed**, Bourne shell scripts, C-shell scripts, and on and on, true integration is, well, more difficult.

Some UNIX editors, such as the various Emacs products, do provide some help with simple syntax and "pretty-printing" for a few of the more common languages. But for locating proper parameters for a procedure to be called, or even for glancing at the source, there is no user interface system that seamlessly applies across the spectrum of UNIX code. Hence, even if we solved the problem of the preceding section--that of locating things--we still would have no way of invoking those locating procedures from our user interfaces.

There are two UNIX features that make this a hard problem. One is the proliferation of editors, the other the proliferation of languages. Both are features you would not want to do without, and they are both part of what makes UNIX the powerful system it is. But it would be even better if your editor were able to "ask" your programming language how it wants to be edited (with defaults when it doesn't

answer)--and better still if your programming language could "ask" your editor for help in pretty-printing itself. In a system like Smalltalk, this "asking" is implicit: it simply happens by the inheritance of methods from higher classes, and you need not write any code at all.

It is true that this particular way of integrating is not essential to produce clever user interfaces. But until Unix programs can all work together so that any UNIX code in any language can be seamlessly called at any time from any other UNIX program, no amount of cleverness in user interfaces will get us an IPE for UNIX.

System Seamlessness

System seamlessness refers to the ease with which a programmer in an IPE can look at, and influence, every aspect of the programming environment, including the operating system. This is hard in UNIX for two reasons. First, and most fundamental, is the "kernel seam". Code in the UNIX kernel is different from other code: it runs under different assumptions, for one thing, and it requires reboots to test--and these are just two of many such differences.

Seamlessness in UNIX is also hard because of the "process seam", the division of UNIX work into processes that share almost no state and can communicate with each other only through fairly heavyweight mechanisms like pipes. Most UNIX programmers are so accustomed to these having "seams" in the environment that they may not be perceived as obstacles to creating an IPE. They nonetheless are, as can readily be demonstrated.

Suppose you want to change the way the **sqrt** function works. For instance, say that it returns the square root of the absolute value for negative arguments, and you want it to return *NaN* ("Not a Number"). You want to do that without recoding **sqrt** or finding every place in your code that calls it, but simply by wrapping the system **sqrt** routine with a little code that returns *NaN* for negative arguments, and calls the old routine for positive arguments. Doing this is not easy in UNIX. If you are willing to have all your code call **foosqrt**, you can have **foosqrt** be the wrapper that calls **sqrt**. If you insist my code also be called **sqrt**, and yet call the system *sqrt*, then you must do loader tricks to have the same name appear twice with different meanings for each. And what if you don't want to rebuild, but just dynamically load new programs and program interfaces? Not a casual task under UNIX.

The problem of system seamlessness is not only one of changing existing system functions. Suppose you want to set a breakpoint in a part of the UNIX kernel, a breakpoint to be taken only when that part of the kernel is called from your user program. Or consider an even simpler example: suppose you have a breakpoint in the UNIX kernel and you want to see the complete call stack of the user process that called down into the kernel routines and finally hit the breakpoint. This can be done, but it isn't easy: because of the kernel seam, the user stack might not even be addressable from the kernel.

A similar issue with which to grapple is the one of locating kernel-related parts of UNIX. Even as simple a thing as connecting the running kernel to its object file is hard to do. The kernel seam means that if you run a version of UNIX that's different from the UNIX described in **/vmunix**, a multitude of programs (like **ps**) break.

Of course, seams also have their virtues, particularly in the timeshared multiuser environment common for UNIX these days. Seams keep one misbehaving program or user from causing excessive damage to other programs or users. In a UNIX IPE, though, this benefit could be acquired in one of several alternative ways. First, the IPE could run only on a personal workstation. This is the platform on which most existing IPEs now run, and it certainly serves to limit the damage to your own goods.

Second, the IPE could have a strong notion of which user requested which kind of seamlessness, and only permit seams to be removed where permitted. For instance, if you advised the kernel to use some code in place of its usual *write* code, it would only do so when you called it, and then would be able to recover gracefully if you gave it code that infinitely looped. The Unix compatible Mach operating system [8] offers approximately this capability via its "ports", but it required a complete redesign and rewrite of the kernel.

Incrementality

Incrementality is related to all three previously mentioned problems for IPEs in UNIX. It refers to the ability to access code and make small changes to it in a "lightweight" way. Two examples make this clear. First, IPEs always have interpreters as well as compilers. In a UNIX IPE, if you had a small C program (or a change to a small piece of a large program) to run, you wouldn't bother to compile, but would just give the program to the C interpreter. Second, IPEs generally permit small changes in small amounts of time. In an IPE, if you have a one-line change to a program that is part of a 2 MB object, it would take only moments to implement. Most UNIX systems, though, take quite a while to build a 2 MB object.

There are incremental loaders for UNIX, and there are C interpreters. But they are all standalone tools that cannot be used seamlessly with every part of the operating system. If you want to load a new piece of kernel incrementally, or interpret a piece of the system-supplied C library, it ought to be as easy as doing so for your own application code. Existing incremental loaders and interpreters don't make this possible, through no fault of their own: UNIX just doesn't make it possible.

What is the basic reason incrementality is hard in UNIX? It actually comes down to this: not enough information. Without taking very special care in building an object module, there is not enough information around to load things--successfully and incrementally--into the module multiple times, and an interpreter will have difficulty finding existing variable names, locations, and types.

To bring incrementality to a UNIX IPE, there must be cooperation among existing UNIX tools to a much stronger degree than now exists. The best cooperation now comes from the C compiler when using the **-g** option. The resulting object code contains complete information about the code, including filenames containing source, along with names and types of all the variables. But try getting the same information about your **awk** program, or setting breakpoints when a certain pattern appears in a pipeline between two programs. Not in current UNIX.

If a UNIX system was enhanced so that every program could uniformly report about its types and names in a manner sufficient for various incremental tools to work, it would still be UNIX. Old tools

would all continue to run, and new tools would have additional opportunities to cooperate with the environment. Universal **-g**, together with locating objects, user interfaces, and system seamlessness, will go a long way toward eliminating the barriers to a UNIX IPE.

Conclusion: There is Hope

We've painted a dim picture of what it takes to bring IPEs to UNIX. The problems of locating, user interfaces, system seamlessness, and incrementality are hard to solve for current UNIXes--but not impossible. One of the reasons so little attention has been paid to the needs of IPEs in UNIX is that UNIX had not had good examples of IPEs for inspiration. This is changing: for instance, one of this article's authors has helped to develop the Smalltalk IPE for UNIX (see the adjacent story), and two others of us are working to make the Cedar IPE available on UNIX.

What's more, new UNIX facilities, such as shared memory and lightweight processes (threads), go a long way toward enabling seamless integration. Of course, these features don't themselves deliver integration: that takes UNIX programmers shaping UNIX as they always have--in the context of a friendly and cooperative community. As more UNIX programmers come to know IPEs and their power, UNIX itself will inevitably evolve toward being a full IPE. And then UNIX programmers can have what Lisp and Smalltalk and Cedar programmers have had for many years: a truly comfortable place to program.

How an Integrated Environment Integrates with UNIX: The Smalltalk-80 IPE

by L. Peter Deutsch

For an integrated programming environment to provide value to programmers, it naturally has to have raw material to integrate. The Smalltalk-80* IPE, for example, provides integration within the world of the Smalltalk-80 language and class (data type) library. But this IPE has also evolved to a point where it provides integration between this world and the four traditional sources of functionality in UNIX: system calls, services available through sockets, packages available as C source or object code, and self-contained applications designed to be run from the shell (including shell scripts).

To program in the Smalltalk-80 language, one creates a set of *classes.* Each class defines a data type by specifying a set of state variables, and defines the operations on data of that type by specifying the names and implementations of the operations. Classes, as defined here, are fully opaque: any class can be substituted for any other class that has compatible operations. This attribute automatically provides for full data abstraction for *every* class.

For instance, if there is already a well-designed "stream" abstraction implemented by some class,

* Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

then new implementations of the abstraction are automatically compatible with all existing uses of the existing class. Neither existing implementations nor existing clients need be rewritten (in the present Smalltalk-80 system, in fact, they don't even need to be recompiled).

While other object-oriented languages--such as C++ [9] --provide at least some of the linguistic advantages of the Smalltalk-80 language, the Smalltalk-80 IPE adds three characteristics that specifically support rapid software development. The compiler is incremental at the grain of individual operations, and compilation does not discard any execution state (allocated objects or call stack). Second, in place of a file editor, the Smalltalk-80 "browser" provides both hierarchical and query-oriented access to the classes and operations of a program. And third, the Smalltalk-80 IPE contains the source code for the entire IPE itself, including the compiler, browser, and user-interface classes. This allows the programmer to gain greater understanding of--and eventually to enhance--the IPE.

Within this environment, all that is needed to incorporate the four sources of UNIX functionality is to provide classes corresponding to each source. System calls generally appear as operations on a small set of classes, such as *File* and *FileDirectory*. Services available through sockets appear as specialized classes which present a semantically appropriate operation set that's implemented using lower-level facilities (such as *FileStreams* or *RemoteProcedureCalls*). For packages implemented by C code, the Smalltalk-80 IPE provides "glue" facilities that establish the correspondences between C procedures and operations defined in some class, and between C objects and Smalltalk-80 objects. Self-contained applications are treated in a fashion similar to socket-based services: the Smalltalk-80 IPE provides basic stream connections, and each application can be presented through a specialized class that provides higher-level operations by constructing input for--or parsing output from--the existing application.

To observe a simple example of using existing UNIX applications directly, imagine wanting to locate a list of files with **find** and then to use **egrep** to search each of them for a pattern. Doing this through the shell requires manipulating the output of **find** (which produces each filename on a separate line) into the syntactic form required by **egrep** (in which all the files appear on the command line). However, a (hypothetical) Smalltalk-80 program could perform this function with no trouble:

```
(UnixShell execute: 'find . -name \*.text -print') asLines
        collect:
                [:fileName |
                (UnixShell execute: 'egrep smerglitch ' , fileName) asLines]
```

In contrast to the shell (even extended by **awk**), which limits the interactive user to processing strings, the Smalltalk-80 environment allows the user to apply the full power of a programming language to the input and output of existing applications.

At the user-interface level, the Smalltalk-80 system can either provide its own screen management and interaction style, or use the host window system and some of the host interaction toolkit, such as those facilities provided by X Windows environments [10] or AT&T's Open Look user interface. Since the Smalltalk-80 language enforces data abstraction, applications running within the Smalltalk-80

system generally will neither know nor care whether the objects that mediate between them and the user are implemented primarily with the Smalltalk-80 system itself (as is the case currently), or primarily in the underlying operating system and C libraries (as might be the case in the future).

In short, the qualities of the Smalltalk-80 IPE that have distinguished it as the archetype of a flexible, evolvable system--a fully object-oriented language, support for incremental development, fully open source code, and explicit facilities for access to the platform's functionality--make it an excellent integrator for many of the present capabilities of Unix and a good candidate for accommodating Unix's future evolution. Although, like other single language IPE's, the Smalltalk-80 environment does not accept other languages as full citizens in its universe, the Smalltalk-80 programmer does not have to go outside the Smalltalk-80 framework of classes and operations to enjoy access to Unix capabilities.

References

[1] Anthony I. Wasserman, editor, Special Issue on Automated Development Environments, IEEE Computer, Volume 14, No 4, April 1981

[2] Daniel Swinehart, Polle Zellweger, Richard Beach, and Robert Hagmann, "A Structural View of the the Cedar Programming Environment", *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 4, pp. 419-490 (October 1986).

[3] Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation*, McGraw-Hill (1983).

[4] Xerox Corporation, *Interlisp Reference Manual* (October 1983).

[5] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall (1978).

[6] L.E. Bonanni and A. Glasser, *SCCS/PWB Users Manual*, Bell Telephone Labs, Inc. (November 1977).

[7] Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System", *Proceedings of the 6th International Conference on Software Engineering*, pp. 58-67 (September 1982).

[8] Richard F. Rashid, "Threads of a New System", *UNIX REVIEW*, Vol. 4, No. 8, pp.36-49 (August 1986).

[9] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley (1986).

[10] Robert W. Scheifler and Jim Gettys, "The X Window System", *ACM Transactions on Graphics*, Vol. 5, No. 2, pp. 79-100 (April 1986).

UNIX Needs A True Integrated Environment: CASE Closed

Mark Weiser, L. Peter Deutsch and Peter B. Kessler

CSL-89-4