Window Systems Should Be Transparent

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Commercial UNIX window systems are unsatisfactory. Because they are cumbersome and complicated, they are unsuitable companions for an operating system that is appreciated for its technical elegance. Their clumsy user interfaces clutter the view of the operating system. A good interface should clarify the view, not obscure it. `Mux` is one window system that is popular and therefore worth studying as an example of good design. (It is not commercially important because it runs only on obsolete hardware.) This paper uses `mux` as a case study to illustrate some principles that can help keep a user interface simple, comfortable, and unobtrusive. When designing their products, the purveyors of commercial window systems should keep these principles in mind.

Introduction

`Mux` is a window system with no icons, no help facility, no customizability, no noise, and only two menus (one with five entries, one with seven). The spareness of its user interface distinguishes it from commercial window systems such as X and SunWindows, yet `mux` is a comfortable and effective system. As the author of `mux`, I am told by many people who have tried other window systems that `mux` is preferable. Why is a window system that is so simple also so popular?

The task of a window system is to provide a multiplexed interface to an operating system. 'Windows' is an apt word because it is through windows that we see the operating system and its programs. A good window system provides an unobstructed view, transparent in two directions. First, programs should be unaffected by the window system under which they run. Second, customers of a window system should be able to ignore it when using those programs. Here, I will concentrate on the viewpoint of the user; the programming issues are distinct enough to be discussed in a separate paper.

`Mux` is successful because it is transparent and unobtrusive. This does not mean that its services are minimal, but that it presents them simply enough that they can be assimilated and used unconsciously. After some practice, a user of `mux` can exploit its facilities without thinking, as automatically as backspacing to correct a typing mistake, or using a steering wheel to guide a car. No amount of practice can make it easy to drive a car by typing compass directions on a keyboard, and no amount of practice can make some window systems feel unobtrusive.

Mux runs only on slow, obsolete hardware that is no longer manufactured, while X runs on almost every modern display. It seems likely I'll soon be asked to turn off my `mux` machine and start using X. I will be reluctant. The window managers I've seen for X are not attractive. It is not enough that X be standard, it should also be good. X needs a much smoother window manager.

`Mux` isn't perfect, but it's good enough to serve as an example of an effective user interface. The principles that make it effective can be distilled to a set of rules of design. These rules are general enough to be applied to programs other than window systems; the user interface of a text editor, a debugger, or any other practical program should never stand in the way of the task at hand. The rules are not tricks such as switchable mouse cursors, but principles that help avoid the mistakes that keep a user interface from becoming subliminal.

`Mux` was not designed for novices, but novices can use it comfortably. I wrote it because I wanted an

effective terminal for programmers, such as myself, who use the UNIX® system. The intent was to make the system more productive. `Mux` also turned out to be easy to use, in part because it was designed with a consistent idea of who would use it. The first rule is therefore:

Design with a specific customer in mind.

The Model

`Mux` multiplexes the keyboard, mouse, and link to a UNIX system among a set of independent virtual terminals on one bitmap terminal screen. The multiplexing is discussed fully elsewhere [Pike 1984]; the following is a summary. A single serial line, running typically at 9600bps, connects a Blit terminal to a ''host'' UNIX time-sharing system. On a traditional full-duplex terminal, characters received from the host are displayed on the screen, and text typed at the keyboard is sent to the host, to be echoed back to the terminal. `Mux` turns the Blit into many terminals by multiplexing the communications resources among a number of windows, each of which acts like a regular terminal. Characters typed on the keyboard appear in the *current window,* which is selected by pointing with the mouse; output from a host process goes to the window associated with its process group.

Each window on a `mux` screen is an independently programmable terminal. Some UNIX programs, such as the text editor `sam` and the debugger `pi` [Pike 1987, Cargill 1986], ''down-load'' code into their windows to customize the terminals' behavior. This can be contrasted with the more common technique of creating a new window for such programs. I don't think that either way is superior; the idea of a programmable terminal is more a byproduct of the way `mux` developed than a conscious design decision.

The default terminal program provided by `mux` allows ''cut and paste'' editing of the text in the window [Goldberg 1984]. In some sense, this is an extension of the editing any operating system provides for correcting typing mistakes. But because any text on the screen may be edited—not just the current input line—some difficult questions must be faced. When is input sent to the program reading from the terminal? When does a program learn that its screen has been edited?

Ordinarily, the operating system handles line-at-a-time input. When a user types newline (carriage return), the preceding line of text is passed to the program that is reading from the terminal. Until the newline is typed, the line may be edited (typically by backspacing); after the newline is typed, no changes can be made. `Mux` generalizes this editing to apply to any text on the screen, not just the current line of input. `Mux` has no current line; instead there is a division between characters the host has seen and characters not yet transmitted. The location of this division is the *output point:* the position in the text that separates output from input. Characters before the output point have (unless edited) been seen by the host, either as output or input; characters after the output point have not been seen by the host. When characters are sent from the host, they are inserted into the window at the output point, and the output point is advanced beyond the characters. When a newline is typed at the terminal, characters between the output point and the newline are sent to the host, and the output point is advanced beyond the newline. Finally, changes made to text appearing before the output point are maintained in the terminal only; the host system never sees them. Thus, a user may make changes to the screen at will, but only those changes made after the output point will be seen by the host, and even then only when a newline is typed. The output point moves forward monotonically, but the typing cursor may be anywhere in the window.

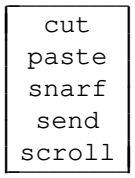
It is impossible on a traditional terminal to correct a typing mistake on an earlier line, so some UNIX programs provide an escape to a conventional text editor, for example, to compose mail messages. `Mux` obviates such escapes because *all* text in `mux` windows can be edited. Unfortunately, this means the contents of the screen are not necessarily the output of any UNIX program; a user's edits can make the screen disagree with what the program printed. 'Visual' programs that maintain their screens by cursor addressing cannot tolerate this inconsistency, so many window systems disallow general editing of the display. `Mux` avoids the problem differently; it does not support cursor addressing. To run a visual program on a Blit, you must down-load a program that emulates a traditional terminal. The mouse-based programs that followed `mux` have largely displaced visual software on Blits.

A Manual

This section describes `mux`'s complete user interface.

The mouse on the Blit has three buttons. The left button (button 1) is used for pointing at another window to make it the current window, or at text within a window to select it. By pointing at some text, holding button 1 down, and releasing it somewhere else, an arbitrary substring may be selected. A double click on a word or the end of a line selects the word or line. The middle button (button 2) provides a pop-up menu of editing facilities; the right button (button 3) provides a pop-up menu of window management facilities.

The editing features affect two objects: the *current text,* selected with the mouse and indicated by highlighting; and an invisible storage place called the *snarf buffer.* The button 2 menu looks like:



`Cut` copies the selected text to the snarf buffer (overwriting its contents) and deletes it from the screen. `Paste` replaces the selected text with a copy of the snarf buffer. `Snarf` copies the selected text to the snarf buffer without affecting the screen. `Send` is described below, and `scroll` toggles to determine whether the window scrolls automatically to reveal new text received from the host. `Cut`, `paste`, and `snarf` do nothing if the source of the text for the edit is empty.

The `send` ''button'' is central to `mux`'s convenience. When `send` is operated, if any text is selected, it is automatically snarfed. Without remembering it, `send` then deletes any text after the output point. It then appends the snarf buffer, after the output point, exactly as if it had been typed. If the text does not end with a newline, one is added automatically. Finally, the characters are sent to the host as if they'd been typed. Thus, in the simplest case, one may select a previous command on the screen, activate `send`, and thereby cause the command to be re-executed.

Typing is easy. Characters typed on the keyboard are inserted into the text, replacing the current selection. If the selection is non-empty, it is `cut` first. After a character is typed, the current text is the null string after the character. Typing a backspace deletes the previous character, even a newline, but not across the output point.

The button 3 menu presents a short list of window-manipulation functions:

`New` creates a new window and attaches to it a new process group and command interpreter on the host. The size and position of the window is defined by sweeping with the mouse. After `New` is selected, the mouse cursor changes to a box and arrow, and by pressing button 3 at an arbitrary corner of the new window and releasing it at the diagonally opposite corner, a rectangle is defined. `Mux` provides feedback by displaying dynamically the rectangle defined by the current mouse position. `Reshape` and `Move` change the shape or position of a window. Both provide a gunsight cursor with which to indicate the window to be changed. `Move` then moves the window to the position indicated when the button is released (again, `mux` draws a rectangle for feedback); `Reshape` executes the same mouse protocol as `New` to define the new shape of the window. `Top` and `Bottom` change the stacking order of the indicated window; `Top` makes it fully visible, in front of the other windows; `Bottom` pushes it to the rear of the stack. `Current` and `Delete` provide the named service for the indicated window. Any button 3 operation may be aborted by clicking button 1 or 2. Finally, by pointing at a non-current window and clicking button 1, the window is

made current and top simultaneously. This is the most common way to change the current window.

The final element of the user interface is a scroll bar in each window to control access to text scrolled off the top of the window. (To keep things manageable, `mux` maintains only the most recent 10,000 or so characters for each window.) The scroll bar is activated by depressing a mouse button over the bar, which is a vertical rectangle at the left of the window. Different buttons provide different actions in the scroll bar. The scroll bar is probably the least satisfactory part of the user interface; a user can't intuit how to use it. Part of the problem is that the behavior of the buttons depends on where the mouse is pointing, but `mux` doesn't change the cursor to reflect this change. It should. In `mux`'s defense, though, scroll bars in other systems do not seem particularly better. They are a topic for further research.

Some examples

Consider the actions required to make a new window: pressing button 3 anywhere on the screen, moving the mouse over `New` on the menu, releasing button 3, moving to one corner of the desired window, pressing button 3 again, moving to the diagonally opposite corner, and releasing button 3. Unless windows are created with default size or position, this sequence cannot reasonably be made simpler. Pop-up menus are ideal for selecting from small lists of possibilities because they're at your fingertips: no mouse motion is required to activate them. The Macintosh user soon tires of the hand waving necessary to reach the pull-down menus (and begrudges the screen area they consume). In summary,

Minimize mouse activity.

Some systems try to be helpful by placing windows automatically. Whether windows should tile or overlap is an issue I don't wish to debate, but suffice it to say that if you implement overlapping windows, you should let the user decide where to put them. If you make the window system choose, the user will end up rearranging the screen anyway.

Don't second-guess the user.

The `Reshape` button uses the same mechanism as does `New` to define the new position and size of the window. Devices such as sliding corners or edges are convenient but are not worth the trouble to implement or explain. They take up screen space and require parts of the window to have special properties that are either activated unintentionally or are hard to select. (Evidence for this is that users complain that the `mux` scroll bar is easy to activate accidentally and requires too much precision to activate intentionally. I don't know how to solve the two problems simultaneously.) Once a `mux` user knows how to make a new window, the `Reshape` button is trivial to use and requires no extra explanation; the changing mouse cursor provides the necessary cue.

Use consistent mechanisms for related actions.

One property that distinguishes `mux` from some other systems is that the current window is selected by clicking a mouse button; merely placing the mouse over the desired window is insufficient. Although it probably makes little practical difference, `mux` requires the click because there are (rare) times when it is helpful to type at a window that is largely obscured, and because it is sometimes convenient to move the mouse cursor out of the way altogether without switching windows.

Selecting text is one of the commonest editing activities, so it should be as easy as possible. `Mux` requires one button push, with the position of the press and release defining the ends of the selection. SunWindows uses two clicks (and two buttons), one at each end. This makes it easy to select more than a windowful of text (which cannot be done in `mux`), but makes the overwhelmingly more common smaller selections twice as hard to do. Moreover, between clicks in SunWindows there is an invisible state; the state in `mux` is always clear because the button must be held down and `mux` keeps the selection highlighted. Sometimes generality should be sacrificed to convenience.

Make the common cases easy.

The UNIX system allows arbitrary type-ahead, so normal terminals often interleave prompts from the command interpreter with typed input. In `mux`, however, the prompt always appears at the output point, which is always at the beginning of a typed line of input. If a prompt is printed while the user is typing a

line, it is inserted automatically at the output point, to the left of the line. Typed input is directly usable in a subsequent `send`, because it is never corrupted by output in the middle of a line. Output may appear *between* complete lines of input, but that is a lesser problem. In practice the output point keeps output and input largely disentangled.

Because the character cursor may be placed anywhere, input may be assembled in convenient order. After typing most of a command, a user may reach back near the beginning of the line and correct a spelling mistake or add an argument, or may type a newline, causing characters between the output point and the newline to be sent to the host. For example, after typing a long command the user may realize that some other command must be run first to make the long command work correctly; this can be done by putting the cursor back at the output point, typing the other command and hitting newline to run it, then moving the cursor to the end of the remaining input text and typing a second newline.

The output point is an invisible spot in the window. Proposals to make it visible, either by drawing a second cursor in the window or by discreetly differentiating text after the output point, were entertained but never implemented. In practice the output point is usually self-evident, and the effort required to make it visible didn't seem worthwhile. If I were doing `mux` over, however, I might consider doing something to mark the output point.

Some shells (command interpreters) have a ''history'' feature that remembers input to the shell and allows commands to be rerun by typing some abbreviation. This feature is convenient (and even handy in `mux` if the desired command is no longer visible) but restrictive. For one thing, it only applies to the shell, not to the programs it runs. Since `mux` windows are terminals, the equivalent of history (snarfing and sending) applies to any program without prearrangement. Also, with `mux` the source of input may be output from other programs. For instance, many programs when given incorrect arguments will print a ''usage'' template that shows how to use them correctly. With `mux`, the template may be edited into the right form and sent to the host.

Put features where they only need to be implemented once.

A slightly different usage makes `mux` a sort of interactive pipe: the output of one program may be edited by hand (even just by selection) and passed as the input to another program. For example, file sizes printed by the directory listing program may be sent to the desk calculator. Also, output from commands may be used to construct new commands. The best example is editing the output of `make -n`, perhaps to change the flags temporarily on a particular compilation. Another is prepending a command to a list of file names (such as the output of `echo *`) and then selecting and sending the entire string. `Send` appends a newline if needed; it is never necessary to touch the keyboard to run a command already on the screen.

One change brought about by such editing is a new style of output. Interactive programs such as `sh` and `adb` now produce output that may be used directly as input to themselves. This is an issue of syntax rather than format. On the UNIX system, commands already produce output whose format—free of headers, with one item per line—is expected as input by other programs. For example, the old command interpreter printed the value of a variable as

```
var is /usr/rob/paper.ms
```

but the new one prints

```
var=/usr/rob/paper.ms
```

which is exactly the input necessary to recreate the value of `var`. The value is easily changed by editing this text and sending it back to the shell. When all programs provide output usable as input, the system is more convenient to use.

Speak the language you understand.

The most common editing action is just grabbing an old command and sending it. Because `send` uses the contents of the snarf buffer if nothing is selected, and because the menus pop up with the previous selection under the mouse cursor, a command may be run repeatedly just by clicking button 2. After its source has been changed using the regular file editor, a program may be recompiled by going to another window with the compilation command in the snarf buffer and clicking button 2. Until the contents of the

snarf buffer change, nothing else is necessary. (The file editor, `sam` [Pike 1987], which is also based on the cut and paste model — indeed, it is almost identical to `mux` at that level — uses a separate snarf buffer for unrelated technical reasons, and it may be best to keep it separate.)

If the user cannot remember what the snarf buffer contains, he or she can `paste` it in, and, if it's right, hit `send`. `Paste` leaves the text selected, so `send` will send it again. This sort of interplay between the various commands is vital to a successful interface, and is worth thinking about in advance and perfecting after usage patterns emerge. The `send` in the earliest `mux` did not send the snarf buffer when there was no selected text. The feature was added later because it seemed like a good idea and added no clutter to the user interface. It has turned out to be one of `mux`'s most useful attributes.

Be prepared to redesign as experience accumulates.

Simplicity

Some of the examples in the previous section are idioms. None of the individual commands in `mux` is particularly useful in isolation (except possibly for `send`), but combinations are remarkably expressive. Idioms develop easily from simple components of a user interface. Many designers of user interfaces try to keep ahead of the users by filling the menus with ''helpful'' commands; what they are doing is inventing idioms on their own, without the users' direct involvement. The commands that result are individually more interesting but less comfortable taken as a whole. The menus are larger and harder to handle; the commands require the user to remember more details; and they are harder to combine. Another tack sometimes taken is to make the system customizable, so idioms may be turned into brief command sequences, but this is false economy. What must be added to an interface to make it customizable often outweighs the gain in effectiveness.

Simple systems are more expressive.

The most obvious way to achieve simplicity is by leaving things out. Indeed, `mux` gets by without several things other systems consider necessary.

`Mux` windows have no ''title bars.'' Each window is a rectangle with a plain border, with the current window's border heavier than the other windows'. Without a title bar across the top, in principle, windows may be hard to tell apart. But, in practice, programs provide different appearances (such as different prompts), so they are easily distinguished. Title bars aren't worth the screen area they consume or the trouble of defining how to use them. Also, title bars have little interesting to say: they announce that every window is a 'Shell Window', or they report the program's version number. Worse, title bars provide a place on the screen to announce features, states, and modes that wouldn't be permitted if they couldn't be announced, and often shouldn't be admitted at all.

Don't clutter the display.

The scroll bars are just vertical rectangles; a dark rectangle down the left edge of the window represents the entire text in the window, and a superimposed light rectangle represents what's visible. There are no top and bottom bumpers or buttons. In `sam`, which was written a couple of years after `mux`, the code that handles the mouse causes the cursor itself to stick to the ends of the bar, as though there were invisible bumpers, so it's easy to go to either end of the text. This is easy to implement, natural to use, and requires no documentation or explanation.

Program the inputs, not just the outputs.

`Mux` doesn't ask for confirmation to do risky things. It always does what the user asks. The most obvious case is deleting a window. The protocol for deleting a window is a two-step process: select `Delete`, then indicate which window. The action can be aborted before picking the window, so there is an element of safety built in; selecting `Delete` does not immediately endanger any window. This is one reason why a window must be selected by a button click rather than just by the position of the mouse. Once in a while a window is accidentally deleted, but this will happen whatever precautions are taken, and the pain of recovering it must be weighed against the bother of dealing with a complicated protocol for deletion. (Presumably software has some safeguards; editors will save their contents, for example.) Confirmations

are ignored in practice. Once a user is at all familiar with a system, the confirmations are executed without thinking about them; the act of deletion becomes ‘click, click’ instead of ‘click, read the message, think about it, click.’

Confirmations don’t work.

`Mux` has no icons. (I’m referring to little pictorial representations of programs, not programmable mouse cursors.) When `mux` was being written, I had never heard of icons, but even if I were writing it today, I would leave them out. It’s so easy to create, delete, and rearrange windows in `mux` that it’s not cumbersome to have lots of windows open, and therefore not necessary to clean up the display occasionally.

There are times when cleaning up makes sense, such as between problems, or after some interruption. Even then, icons are unnecessary. A simpler scheme (although `mux` doesn’t do this, either) would be to allow windows to be pushed, intact, off-screen, with just an edge poking in to be grabbed by the mouse when needed. This requires no more user interface and, just as important, no more programmer interface. (Incidentally, why are icons necessarily pictorial? There is no picture as evocative as the word ‘copy’.) Bitmap icons first appeared in tiling window systems, which cannot allow many programs to have large windows simultaneously. Icons were therefore invented to collapse large windows in a way that a touch of a button could bring them back. Overlapping window systems achieve this differently.

Icons aren’t useful to overlapping window systems.

Not all of what makes `mux` comfortable is what was left out. It is also important to choose carefully what to put in, and to make the things that are put in interact coherently with one another. The ways in which editing and typing interact were carefully chosen and adjusted (and strongly influenced by Smalltalk [Goldberg 1984]). Each of the editing commands isolates a single function, but performs it in concert with all the other commands, by sharing the interface model and the objects upon which it acts. This is hard to say without making it sound trivial, but experience with other window systems shows it still needs to be said.

Design the elements of the interface together.

Some window systems have options that adjust the behavior to suit individual tastes. `Mux` has no options. I would rather defend, and if necessary extend, a good set of design decisions than attempt to make everyone happy by appealing to their pre-set ideas of how things should be. All-inclusive systems end up pleasing no one; they do nothing particularly well, and are unwieldy to implement and to use.

Choose a design and stick to it.

Convenience

How can a system be made convenient? To answer this question requires the expertise of a human factors researcher, but I can provide a couple of illustrative examples. Consider the layout of the button 3 menu, pictured earlier. Functions related in purpose (such as `Top` and `Bottom`) or in user interface (such as `New` and `Reshape`) are grouped together. `Delete`, the most destructive operator, is at one end of the menu, with the least commonly used operations separating it from the more commonly used ones at the top. `New` and `Delete` are the most common commands to activate, and they can be found quickly by their position at the edges of the menu: `New` is at the top, `Delete` at the bottom, and either can be found in an instant without reading all the entries on the menu.

The shorter the menu, the easier it is to use. When the menu gets long, the user must read all the entries to find the one desired. Railway timetables are marked with lines or distinct colors every few lines to guide the eye. If the marks were every twenty stations, they wouldn’t be helpful. Similarly, a menu with five or so entries can be understood at a glance; a menu with twenty must be examined carefully, slowly. Keeping menus small is an easy way to make a system convenient to use. The middle of the seven-element button 3 menu is a no-man’s-land of infrequently used commands, but all of the five-element button 2 menu is accessible without reading the entries. Making the button 2 menu even one entry longer might make it considerably harder to use.

Keep menus short.

Cascaded menus are no help here. The individual menus stay short, but not all options are visible, and such menus are fussy to use.

Don't use cascading menus.

The cost of adding something to a user interface can be assessed by balancing the worth of the addition against the cost of the added complexity. `Mux` users occasionally ask for a feature to search for text in the window. I have resisted their requests because the extra menu entry, although occasionally useful, would make the menu harder to use, and the balance does not seem in favor of adding the feature. Of course, this is subjective; I could have tried a search button to see if I liked it. The problem with such experiments, though, is that social pressures make features harder to remove from a program than to leave out in the first place.

Balance the benefit of a feature against its overall cost.

Even minor conveniences can have important consequences. The menus in `mux` ''remember'' the last item selected from them. After a user has selected, say, `send`, the next time button 2 is depressed the menu comes up with `send` under the cursor. This means that multiple operations, which are quite common, can be done by clicking the button *without having to look at the menu.* Instead, the user can concentrate on the text being edited, maximizing the convenience of pop-up menus.

Remember what the user did last time.

Some systems instead bring the menu up with the mouse cursor at the top of the menu, and with nothing pre-selected. After a few selections, which take a centimeter or two of mouse motion each, the mouse has drifted off the window and the desk. (This is particularly annoying in systems that select the current window solely by mouse position.) Also, such menus force the user to read the menu, because the selection must be made every time; just clicking isn't good enough.

Don't make the mouse ratchet across the screen.

The Apollo and PERQ window systems provide a one-line buffer at the bottom of the screen into which typed characters are echoed. (These systems don't have the concept of an output point, but their designers wanted to allow mouse editing of input text.) As each line is typed, the user's attention must flick from the bottom of the screen to the destination window and back again. This is needless distraction.

Keep activity localized on the screen.

Responsiveness

`Mux` is not a speedy window system, but it is a responsive one. For instance, when a rectangle is swept to make a new window, the window appears instantly, and the associated shell starts right away. After the rectangle has been swept, less than a second passes before the shell prompt appears.

Compare this to SunWindows or X, which spawn subprocesses that eventually pop up on the screen. After requesting a new window, the user is free to do other work. But at an unpredictable time, usually some seconds later, a window will suddenly appear (in SunWindows) or the mouse will suddenly change state (in X). This can be confusing and troublesome—for instance, the behavior of the mouse changes unpredictably—so in practice it's best just to wait. These systems have insufficient correlation between the action (requesting a window) and its result (acquiring a window); their delays make them as unsettling as a telephone conversation via satellite.

Bind actions directly to their consequences.

A related rule is:

Don't change state unpredictably.

An unresponsive system cannot be used dynamically. Different people work differently, but some

mux users treat windows like sheets of scratch paper, grabbing a fresh one for a quick calculation and then tossing it away soon afterwards. The clumsiness of making and adjusting windows in other systems forces their windows to be created statically rather than as occasion demands. It's as though the number of windows were fixed, and a window system with a fixed number of windows is like a file system with a fixed number of files.

Don't make the user wait.

System issues

This paper is not about the systems programming problem of implementing mux, but some of the systems-related decisions made in the design did influence the user interface. The most evident of these is that mux windows do not support cursor addressing. This was a deliberate omission; I wanted to break cleanly away from 'visual' software to tools that use the mouse. Today this doesn't seem like a radical decision, but when it was made in 1981 it was extremely contentious. Fairly early, one of mux's clients wrote a terminal program, to run in a window, that emulated a traditional 24×80 terminal. Before long, however, mux's editing capabilities attracted people by providing a helpful environment that encouraged them to use the mouse.

Another factor in mux's success was that it does not override the operating system it serves. What it offers is not a new programmer interface, but a better user interface to what was already there. Also, the way mux handles text fits in comfortably with the UNIX system's idea of stream processing. A user interface that instead depended on novel ideas or facilities would not work as well. For example, a UNIX window system employing multiple type fonts would not be as successful, because the operating system does not itself use multiple fonts.

Tie the user interface to the ideas in the system.

The ideal

When a user interface works well, it's invisible. The driver of a car isn't conscious of shifting gears, braking, and steering; the task at hand is driving. Similarly, the user of a computer program shouldn't be conscious of the details of the interaction, but instead should be free to concentrate on the problem being solved. The user interface to correct typing mistakes in a traditional operating system is one interface that succeeds by this criterion; typing backspace becomes second nature.

Make the user interface invisible.

Not all mux users are completely comfortable with a mouse, but for those that are, mux's user interface seems nearly as unintrusive as typing backspace. Occasionally, a mux user sits down in front of a terminal *not* running mux, makes a typing mistake, and reaches for the mouse instinctively. This implies not only that mux makes it possible to use the mouse to edit text, but that mux makes it so easy to use the mouse that people do so subconsciously. That is the mark of a successful interface.

References

[Cargill 1986] T. A. Cargill, 'The feel of Pi,' *Winter USENIX Conference Proceedings, Denver 1986,* 62-71, USENIX Assoc., El Cerrito, CA.

[Goldberg 1984] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley, Reading, Mass., 1984.

[Pike 1984] R. Pike, 'The Blit: a multiplexed graphics terminal,' *AT&T Bell Labs. Tech. J.,* **63**, (8), 1607-1631 (1984).

[Pike 1987] R. Pike, 'The text editor sam,' *Softw. Pract. Exp.,* **17**, (1), 813-845 (1987).