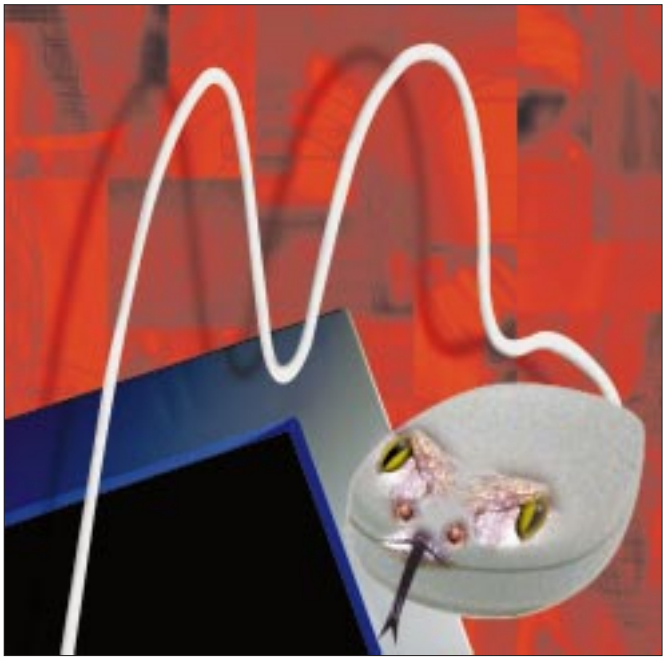
by PETER COLLINSON, Hillside Systems

# The Trouble with Windows

Recently, I've been wondering whether I am slowly turning into a dinosaur. I am not sprouting three horns or acquiring T-Rex teeth, but I am beginning to consider the possible extinction of the UNIX species. More of my work seems to be migrating to various Windows-based systems with their supposedly friendly mouse-driven GUI interfaces. For example, I'm generating images for my Web site using Corel Corp.'s suite of graphics programs. I'm doing more document writing using Microsoft Word because that is what the customer wants.

However, and perhaps here's the dinosaur bit, I keep returning to my UNIX systems to do any substantial job. I return to a command line-based interface, like an old friend, whenever I want to process many files, or perform some repetitive task on a set of files. I return to my UNIX editor for text generation and I often import that text into Word as a final formatting step.



Now, I am not one of those people who resists change at all costs. I like to look at new things that come along. If I don't like something, I usually spend time to make sure that my dislike has better reasons than "the system is different from that which was there before." In computing, it's easy to trundle along using what you know now and never pick up anything new.

What worries me is that the world seems to have totally and unquestioningly embraced the GUI-based Microsoft interface. With NT, Microsoft seems to have understood that to kill off UNIX, it will have to make a system that does not crash at the slightest problem. After all, crashing machines and lost work are tangible problems, problems that people undoubtedly complain about.

What's interesting is that the business world has leapt for Microsoft-based products in the certain knowledge that there is no effective support for the software. Microsoft and PC software vendors are about as approachable and responsive to complaints as the proverbial dead whale that was hard to kick up a beach.

It's less obvious to me whether users ever ask if the human interface to the Windows system is "correct" in some intangible, perhaps academic sense. Microsoft has pushed the Windows GUI at users, who simply accept it. Basically, I think about the interfaces that I use, and I consider that current GUI-driven interfaces leave a lot to be desired.

To be sure, many problems are caused by poorly thought-out design at the application level, rather than intrinsic, fundamental difficulties with the system. But even apart from poor design in applications, I believe that the interfaces don't scale; they only work well for simple operations. Once things start becoming complicated, the interfaces stop being friendly and begin to be battlefields where the user fights the system to get the work done.

To expand a little on what I mean, let's look at something simple. One of the things that we do all the time is move files about the file system. It's one of the fundamental operations that are needed in any operating system. Of course, "moving files" sounds simple but isn't. Do we want the

MICHELLE FRIESENHAHN WILBY

original file left in place? Are we relocating the file or just renaming a file? If we are moving files from one part of the file system to another, then we need to know something about the destination directory. Will we overwrite some important file at the destination?

In Windows, copying files is visual. You open two windows on the screen, one for the source directory and one for the destination. You click on the file that you want to move and drag it into the destination directory. The system realizes what you are doing and copies the data from the source to the destination directory. Sounds easy? Yes, and it should be. Drag and drop is intuitive and works well, or should.

However, on my NT 4 system, brain-dead design decisions kick in. My view of human interfaces is that it's important to be consistent. If the system acts one way while presenting some image to the user, then the system should always act in that way. So drag and drop should always be consistent in the way that it handles files. However, on NT, if I drag and drop files within a single disk, then the system assumes that I wanted to relocate the file and will delete the original. If I am copying files to another disk or another machine, then the system thinks that I want a copy and leaves the original file alone.

There are undoubtedly good reasons why the drag-and-drop action has different results depending on context, but I think that it should not. To be fair, there is an attempt to show the change in state because the cursor changes shape when I enter the different windows. However, I had not noticed this change until it was pointed out to me by a seasoned Windows user. So the effect on the GUI was not as pronounced as it might be. Now that I have noticed the visible indication of what will happen, I expect to use drag and drop more often.

Up to now, I have always copied files using the `File` menu, which is predictable but fiddly. I select a file that I want to copy from the source window, use `Cut` and `Copy`, depending on what I want to happen, find the destination window and `Paste` it in. When I first used this mechanism, I thought it was a tad counterintuitive, and although I have become used to the metaphor, I don't necessarily feel it's a valid one.

Let's add a further complication. If I want to selectively copy several files from a directory, then I need to select those files before `Cut`ting or `Copy`ing them. This is done visually, using one of the things that I hate about GUIs: secret key combinations. To select more than one object you hold down the Control key and click with the mouse on several files. Alternatively, you can hold down the Shift key and select a range of files. It's quite hard to find out about the Shift option. I only discovered it by accident because it's next to the Control key.

I detest these secret options because the whole point of the Windows GUI is that everything a user can do is visible. A menu option may not be available and be grayed out, but the choice remains visible. I don't agree with this modus operandi, but that's another issue.

Since the fundamental theory behind the interface design is that everything is visible, users don't expect to look for the secret parts of the interface that the mode-changing keys represent. Also, when you discover the keys, there is nothing in the interface that tells you that your mouse clicks will do something different because you have a key pressed on the keyboard. I am not sure what should change, but it would be simple to use a new cursor shape to indicate the selection mode.

Incidentally, while writing this column, I've discovered another feature about these file lists. You can type letters into the keyboard and the selection box will jump to an appropriately named file. I've also learned from the help files that in Windows Explorer, and only in Windows Explorer, you can group files by capturing the files inside a box controlled by holding down the mouse button.

## Command Lines

Now I am going to talk about UNIX command line input, and you are all going to think that I am saying it's better. Your hands are poised to send me mail saying you are convinced that I am a UNIX junkie and I don't want to alter. Well, that's not true. I make considerable use of the file-copying systems on NT and rarely use its command line interface. In fact, my NT keyboard usually lives on top of the monitor so I have to reach up for the Control or Shift keys. I don't intend to say that UNIX is better, but I am using UNIX as an example of a well-developed human interface to illustrate where I believe there are shortcomings in the GUI-based systems.

If we return to our file copying example, you know that we copy single files on UNIX by using either the `cp` or `mv` commands. The basis of using a command line is that we supply arguments to a program that does the work for us. To use the system, we have to learn that the commands exist and know their names. Certainly, UNIX command names can be extremely counterintuitive. Like Windows' hidden keys, UNIX hides its features and we have to seek them out. However, UNIX doesn't hide this fact from its users. People are supposed to Read The Fine Manual.

We don't have to learn anything particularly special about how commands are used. After a short time of using UNIX, people acquire an expectation of how commands on the system will behave, how they are controlled, and how they are used in combination with other commands.

It's true that users sometimes don't appreciate why certain aspects of command invocation operate the way they do, for example, why does

```
$ cat afile bfile > afile
```

not work intuitively? It's reasonable to expect that the command adds the contents of `afile` and `bfile` and places the result back into `afile`. It doesn't work because the shell opens the output file `afile` before it executes the `cat` command, so the contents of `afile` have been set to zero before the `cat` command has a chance to execute. On UNIX there is a consistency of result, even if the result is sometimes bad.

When telling someone else about a UNIX command, we usually just have to say the name of the command and perhaps give some indication of any odd arguments the

program may have. The user then understands how to use that command. It's a little like language acquisition, the user is given a new verb (*to flipple*) and will know how to cope with the past tense (*I flippled*) or how to create an adverb (*it went flipplingly*).

This preknowledge of use is an aspect of the UNIX user interface that is often forgotten. People hold up the find or dd commands with their odd arguments as examples of why

*It's a very short step from this single file copy statement to one that moves many files. Very early on, UNIX users are taught that they can move many files using shell file name expansion characters.*

UNIX is hard to learn. But once users understand the arguments, they will know how to make dd read from a file, or how to make find output go into the sort program.

If we are copying single files, then we will expect to do some typing. Not only will we type the name of the command, but also the names of the source and destination files. The system helps us by maintaining the current working directory for a process, so file names can be typed relative to that directory. However, if we are copying files across the file system, we will expect to type the full path name for at least one of the file arguments. Over time, UNIX has developed several ancillary mechanisms that help us to input long path names with more ease.

The first of these mechanisms was perhaps the tilde character, allowing the user to specify their home directory trivially and in a position-independent manner. Later, automatic file and command name completion was taken from the Tenex system and implemented.

I make considerable use of the ability of the bash shell to expand path names dynamically. If I type the first few letters of a file name and hit the Tab key, then the shell will look in the directory that the file is in, and, if a unique match is found, will complete the file name automatically. Path name specification has become as easy as clicking with the mouse in a dialog box: You type a character, hit tab for expansion, type another character, hit tab again and so on.

The other feature of which I make considerable use is the character-based cut and paste using the mouse that is supplied by the X Window manager. This lets me sweep out a set of characters in one window with one mouse action and insert the characters into another window using a single mouse click.

However, there is no escaping that even with these aids, it's much easier to use drag-and-drop mechanisms that visual GUIs give us. It's harder on UNIX, which wants us

to build up a text line like

```
% cp srcfile /usr/file/dest/srcfile
```

ready for execution to make a file copy. Typing this line takes considerably more basic knowledge of the system than the equivalent drag-and-drop action.

However, this knowledge is a building block. It's a very short step from this single file copy statement to one that moves many files. Very early on, UNIX users are taught that they can move many files using shell file name expansion characters. The command

```
% cp src* /usr/file/dest
```

copies all the files starting with the string `src` to the destination directory, ignoring any other files. A simple pattern is used to specify a group of source files and perform a command several times using the matched files as arguments.

Again, there's some basic knowledge that is needed about expansion of file names. The expansion is done by the shell, which then calls the command with the expanded list. Placing the expansion in the shell sometimes has odd effects but makes sense because the expansion can be used consistently with all commands. We are not dependent on the application writer to provide correct implementation of this part of the human interface.

Using pattern-matching characters to generate arguments also means that we don't have the possible problem of missing a file that we want to copy. Once we've issued the command we know that all the files that we have specified will be copied. We don't have this certainty on Windows when copying many files via selections in dialog boxes. It's very easy to miss files that we should have copied, and so we have to spend time making sure that the destination directory contains all the files that we want it to hold.

UNIX also offers other types of pattern matches, which helps us to pinpoint the files that we want to copy. For example, the question mark character matches any single character, so `??` will match all the file names that are exactly two characters long. This option is perhaps much less useful than the ability to match character ranges. For example,

```
$ cp [a-ln-z]*.c /usr/dest
```

copies all the files that don't start with the letter "m" to the destination directory.

## Learning Loops

I suppose the next step in complexity that follows on from the ability of the shell to expand file names is the use of the `for` or `foreach` loop that shells support. When this is coupled with the shell's ability to manipulate strings, we move onto a new plane of usefulness. Here's something I type into the machine without thinking twice:

```
$ for name in *.c
> do
>      mv $name $name.old
> done
```

or in `csh`

```
% foreach name (*.c)
?      mv $name $name.old
? end
```

Both are shell loops, the `for` or `foreach` statements supply a shell variable name that is set to different values while the statements in the loop are executed. In this example, the variable name will be successively set to each of the names of the files in the current directory that match the pattern `*.c`. The string `$name` in the `mv` commands is replaced by the contents of the variable before the command is executed. The loop executes several `mv` commands, each one moving one file from its old name of say `program.c` to a new name of `program.c.old`.

Moving the file names back to their original positions uses a new command and a new concept in the shell:

```
$ for name in *.c.old
> do
>      dest=`basename $name .old`
>      mv $name $dest
> done
```

or in `csh`

```
% foreach name (*.c.old)
?      set dest = `basename $name .old`
?      mv $name $dest
? end
```

The new command is `basename`, and its job is to deconstruct strings. For example,

```
% basename fred.c.old .old
fred.c
```

The command is given an argument containing the file name that we want to take apart, and the string that we expect will occur at the end of that string. It prints its result to standard output. This is where we apply the new concept. The backquote operator takes the result of a command that it executes and reads the data back into the shell. In fact, we then place the result of the command into a shell variable `dest`.

Now, we have not come very far. None of these loops use constructs that are very far removed from typing a single command. The basic knowledge that was learned to copy a file extends seamlessly to give us the ability to express complicated file copies involving the renaming of selective files. Because the shell is a language, we can insert any processing element in the loop, applying a command selectively to a set of files. Also, it's a very short step from here to take one of the loops and create a new command file that can be used to save typing.

This type of complexity is just not easily available with GUI-based interfaces, and the lack of it hurts. To be fair, there are attempts made to provide script-based programming interfaces for various tools. But a huge leap in knowledge is needed to go from using the GUI to using a script.

Many of the systems have internal script languages based on some variant of BASIC. For example, these articles are written using a subset of `troff` markup but are dispatched using RTF format. I automatically generate the RTF using Word. I first pass the source file through a small C program to generate some markup and then into Word for processing by some macros that I have written, translating my limited `troff` constructions into Word markup.

One of the Corel packages provides an event recorder, so you can teach it some action sequence and redo the sequence on new data. But it turns out that not all the actions are recorded, and it's hard to parameterize the actions. I will confess to have given up when I discovered that it would not record all the actions that I needed to take an AVI movie and split it into one file per frame. I ended up making all the key clicks by hand in an error-prone way.

Finally, the Corel suite has a script editor and debugger. Sadly, this is made very hard to learn. The printed manual says, "We can do all these neat things and you'll find all the details in the on-line help." But there is no effective way of progressing sequentially through the on-line help so that you might have some chance of learning how to use the package. Click on the Help button in the package and you are presented with a nice-looking finder. When you select an item in the finder, you get a screenful of text, but the finder goes away. There is no easy way back to the place that caused the text to be displayed. You have to start the finder from the beginning each time.

## In Summary

Again, don't think I am advocating that we should all use the UNIX command line because it's somehow superior to the GUI-based Windows interface. I'm not. I am simply pointing out several deficiencies in the current GUI-based interfaces, deficiencies that seem to have no easy solution.

First, I believe the lack of the notion of a current working directory, or at least the lack of coherent handling of the current directory, to be a serious deficiency. Commands work fine when you read and write data in one directory, but once you start moving data from one point in the file system to another then each application seems to deal with the current directory in its own way. I seem to spend an inordinate amount of time using those file finder dialog boxes.

Second, the fundamental problem is that Windows has no easy way of using its applications in a batch fashion except by using some human as a sequencer, whose job is to sit there and press the buttons. It's not easy to use the applications as part of some other tool, and consequently, there is very little extensibility. We are forced to rely on the designers of the package thinking about all the ways we might want to use their programs.

Third, having a interchangeability of data is a good thing. The common denominator on UNIX is text files. Most of the tools use text as their building blocks, allowing users to create new processing elements, transforming new data in new ways. The power to create new processing elements is firmly in the hands of the users. I will agree that these users have a learning curve to surmount.

Windows tries to achieve the same end with objects. Where objects do fit together, they fit together well. The problem is that the fitting together needs an expert programmer, and the required expertise is far from trivial to learn. With Windows, the power remains firmly in the hands of the developers, and I see this as a problem.

Fourth, scalability is a must. I should be able to process many source files as easily as I can deal with one. This appears to be a serious problem with current Windows systems.

Finally, there really is insufficient attention paid to the way that the GUI works. It's very inconsistent, and it's usually impossible to obtain the reasons why it behaves in the way that it does. I find this is my biggest objection.

---

Peter Collinson runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever… He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email: `pc@cpg.com`.