ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/202165676

The trouble with UNIX: The user interface is horrid

Article *in* Datamation · January 1981

CITATIONS
14

READS
1,315

1 author:

Donald Arthur Norman
University of California, San Diego
337 PUBLICATIONS 42,617 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Design View project

All content following this page was uploaded by Donald Arthur Norman on 30 December 2014.

The user has requested enhancement of the downloaded file.

From Datamation November 1981

The system design is elegant but the user interface is not.

THE TROUBLE WITH UNIX

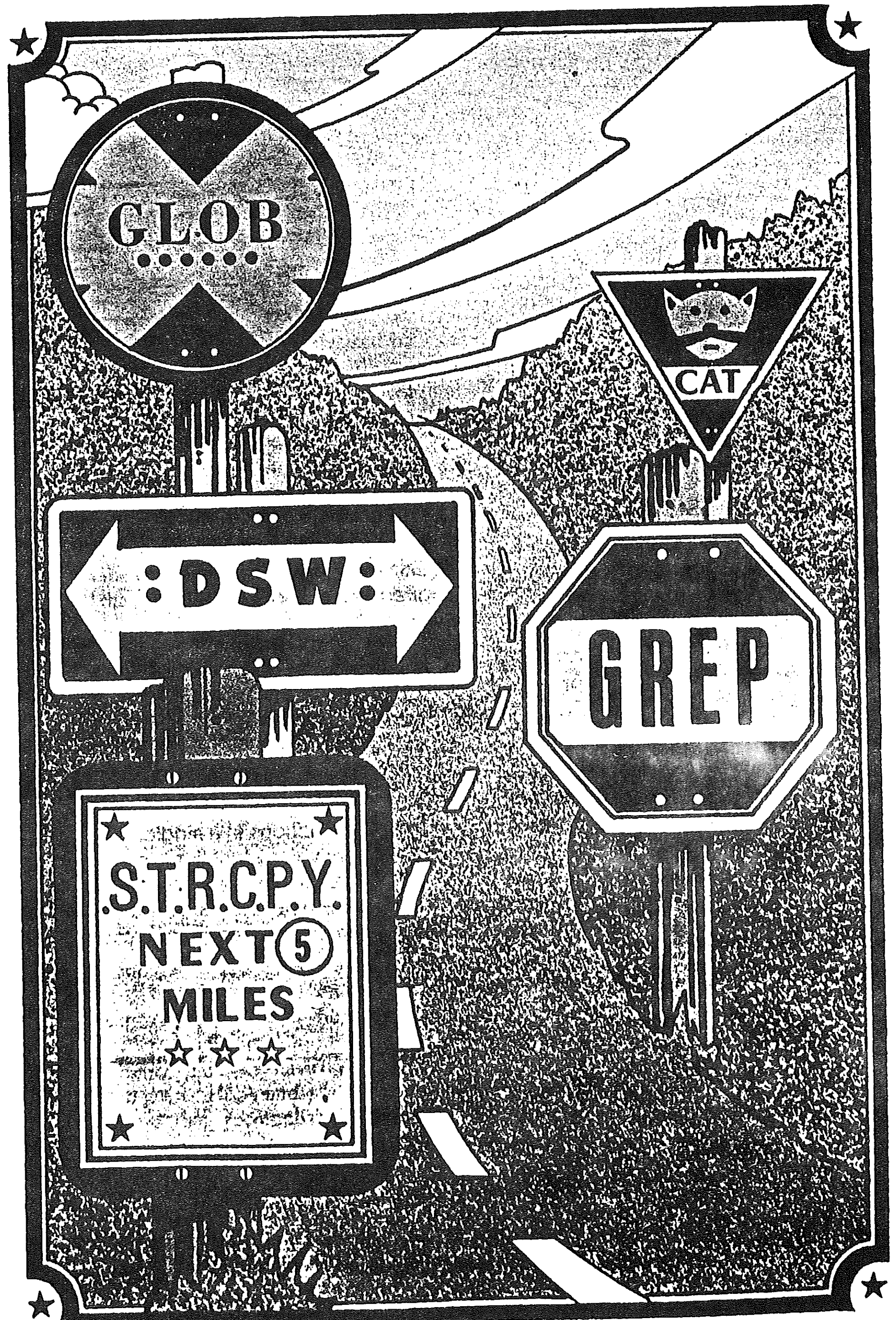

ILLUSTRATION BY CHRIS SPOLLEN

by Donald A. Norman

UNIX is a highly touted operating system. Developed at the Bell Telephone Laboratories and distributed by Western Electric, it has become a standard operating system in universities, and it promises to become a standard for micro and mini systems in homes, small businesses, and schools. But for all of its virtues as a system—and it is indeed an elegant system—UNIX is a disaster for the casual user. It fails both on the scientific principles of human engineering and even in just plain common sense.

If UNIX is really to become a general system, then it has got to be fixed. I urge correction to make the elegance of the system design be reflected as friendliness towards the user, especially the casual user. Although I have learned to get along with the vagaries of UNIX's user interface, our secretarial staff persists only because we insist.

And even I, a heavy user of computer systems for 20 years, have had difficulties: copying the old file over the new, transferring a file into itself until the system collapsed, and removing all the files from a directory simply because an extra space was typed in the argument string. The problem is that UNIX fails several simple tests.

Consistency: Command names, language, functions, and syntax are inconsistent.

Functionality: The command names, formats, and syntax seem to have no relationship to their functions.

Friendliness: UNIX is a recluse, hidden from the user, silent in operation. The lack of interaction makes it hard to tell what state the system is in, and the absence of mnemonic structures puts a burden on the user's memory.

What is good about UNIX? The system design, the generality of programs, the file structure, the job structure, the powerful operating system command language (the "shell"). Too bad the concern for system design was not matched by an equal concern for the human interface.

One of the first things you learn when you start to decipher UNIX is how to list the contents of a file onto your terminal. Now this sounds straight-forward enough, but in UNIX

WHAT IS UNIX?

UNIX is an operating system developed by Dennis Ritchie and Ken Thompson of Bell Laboratories. UNIX is trademarked by Bell Labs and is available under license from Western Electric. Although UNIX is a relatively small operating system, it is quite powerful and general. It has found considerable favor among programming groups, especially in universities, where it is primarily used with DEC computers—various versions of the DEC PDP-11 and the VAX. The operating system and its software are written in a high level programming language called C, and most of the source code and documentation is available on-line. For programmers, UNIX is easy to understand and to modify.

For the nonexpert programmer, the important aspect of UNIX is that it is constructed out of a small, basic set of concepts and programming modules, with a flexible method for interconnecting existing modules to make new functions. All system objects—including all I/O channels—look like files. Thus, it is possible to cause input and output for almost any program to be taken from or to go to files, terminals, or other devices, at any time, without any particular planning on the part of the module writer. UNIX has a hierarchical file structure. Users can add and delete file directories at will and then "position" themselves at different locations in the resulting hierarchy to make it easy to manipulate the files in the neighborhood.

The command interpreter of the operating system interface (called the "shell") can take its input from a file, which means that it is possible to put frequently used sequences of commands into a file and then invoke that file (just by typing its name), thereby executing the command strings. In this way, the user can extend the range of commands that are readily available. Many users end up with a large set of specialized shell command files. Because the shell includes facilities for passing arguments, for iterations, and for conditional operations, these "shell programs" can do quite a lot, essentially calling upon all system resources (including the editors) as subroutines. Many nonprogrammers have discovered that they can write powerful shell programs, thus significantly enhancing the power of the overall system.

By means of a communication channel known as a pipe, the output from one program can easily be directed (piped) to the input of another, allowing a sequence of programming modules to be strung together to do some task that in other systems would have to be done by a special purpose program. UNIX does not provide special purpose programs. Instead, it attempts to provide a set of basic software tools that can be strung together in flexible ways using I/O redirection, pipes, and shell programs. Technically, UNIX is just the operating system. However, because of the way the system has been packaged, many people use the name to include all of the programs that come on the distribution tape. Many people have found it easy to modify the UNIX system and have done so, which has resulted in hordes of variations on various kinds of computers. The "standard UNIX" discussed in the article is BTL UNIX Version 6 (May 1975). The Fourth Berkeley Edition of UNIX is more or less derived from BTL UNIX Version 7 (September 1978), with considerable parallel development at the University of California, Berkeley and some input from other BTL UNIX versions. I am told that some of the complaints in the article have been fixed; however, Version 6 is still used by many people.

The accompanying article is written with heavy hand, and it may be difficult to discern that I am a friend of UNIX. The negative tone should not obscure the beauty and power of the operating system, file structure, and the shell. UNIX is indeed a superior operating system. I would not use any other. Some of the difficulties detailed result from the fact that many of the system modules were written by the early users of UNIX, not by the system designers; a lot of individual idiosyncrasies have gotten into the system. It is my hope that the positive aspects of the article will not be overlooked. They can be used by all system designers, not just by those working on UNIX. Some other systems need these comments a lot more than does UNIX.

—D.A.N.

even this simple operation has its drawbacks. Suppose I have a file called "testfile." I want to see what is inside of it. How would you design a system to do it? I would have written a program that listed the contents onto the terminal, perhaps stopping every 24 lines if you had signified that you were on a display terminal with only a 24-line display. UNIX, however, has no basic listing command, and instead uses a program meant to do something else.

Thus if you want to list the contents of a file called "HappyDays," you use the command named "cat":

cat HappyDays

Why cat? Why not? After all, as Humpty Dumpty said to Alice, who is to be the boss, words or us? "Cat," short for "concatenate" as in, take file1 and concatenate it with file2 (yielding one file, with the first part file1, the second file2) and put the result on the "standard output" (which is usually the terminal):

cat file1 file2

Obvious, right? And if you have only one file, why cat will put it on the standard output—the terminal—and that accomplishes the goal (except for those of us with video terminals, who watch helplessly as the text goes streaming off the display).

The UNIX designers believe in the principle that special-purpose functions can be avoided by clever use of a small set of system primitives. Why make a special function when the side effects of other functions will do what you want? Well, for several reasons:

- Meaningful terms are considerably easier to learn than nonmeaningful ones. In computer systems, this means that names should reflect function, else the names for the function will be difficult to recall.
- Making use of the side effects of system primitives can be risky. If cat is used unwisely, it will destroy files (more on this in a moment).
- Special functions can do nice things for users, such as stop at the end of screens, or put on page headings, or transform nonprinting characters into printing ones, or get rid of underlines for terminals that can't do that.

Cat, of course, won't stop at terminal or page boundaries, because doing so would disrupt the concatenation feature. But still, isn't it elegant to use cat for listing? Who needs a print or a list command? You mean "cat" isn't how you would abbreviate concatenate? It seems so obvious, just like:

FUNCTION	UNIX COMMAND NAME
c compiler	cc
change working directory	chdir
change password	passwd
concatenate	cat
copy	cp
date	date
echo	echo
editor	ed
link	ln
move	mv
remove	rm
search file for pattern	grep

Notice the lack of consistency in forming the command name from the function. Some names are formed by using the first two consonants of the function name. Editor, however, is "ed," concatenate is "cat," and "date" and "echo" are not abbreviated at all. Note how useful those two-letter abbreviations are. They save almost 400 milliseconds per command.

Similar problems exist with the names of the file directories. UNIX is a file-oriented system, with hierarchical directory structures, so the directory names are very important. Thus, this paper is being written on a file named "unix" and whose "path" is /cs1/norman/papers/CogEngineering/unix. The name of the top directory is "/", and cs1, norman, papers, and CogEngineering are the names of directories hierarchically placed beneath "/". Note that the symbol "/" has two meanings: the name of the top level directory and the symbol that separates levels of the directories. This is very difficult to justify to new users. And those names: the directory for "users" and "mount" are called, of course,

After all, as Humpty Dumpty said to Alice, who is to be the boss, words or us?

"usr" and "mnt." And there are "bin," "lib," and "tmp" (binary, library, and temp). UNIX loves abbreviations, even when the original name is already very short. To write "user" as "usr" or "temp" as "tmp" saves an entire letter: a letter a day must keep the service person away. But UNIX is inconsistent; it keeps "grep" at its full four letters, when it could have been abbreviated as "gr" or "gp." (What does grep mean? "Global REgular expression, Print"—at least that's the best we can invent; the manual doesn't even try. The name wouldn't matter if grep were something obscure, hardly ever used, but in fact it is one of the more powerful, frequently used string processing commands.)

LIKE CAT? THEN TRY DSW

Another important routine goes by the name of "dsw." Suppose you accidentally create a file whose name has a nonprinting character in it. How can you remove it? The command that lists the files on your directory won't show nonprinting characters. And if the character is a space (or worse, a "*"), "rm" (the program that removes files) won't accept it. The name "dsw" was evidently written by someone at Bell Labs who felt frustrated by this problem and hacked up a quick solution. Dsw goes to each file in your directory and asks you to respond "yes" or "no," whether to delete the file or keep it.

How do you remember dsw? What on earth does the name stand for? The UNIX people won't tell; the manual smiles the wry smile of the professional programmer and says, "The name dsw is a carryover from the ancient past. Its etymology is amusing." Which operation takes place if you say "yes"? Why, the file is deleted of course. So if you go through your files and see important-file, you nod to yourself and say, yes, I had better keep that one. You type in "yes," and destroy it forever. There's no warning; dsw doesn't even document itself when it starts, to remind you of which way is which. Berkeley UNIX has finally killed dsw, saying "This little known, but indispensable facility has been taken over . . ." That is a fitting commentary on standard UNIX: a system that allows an "indispensable facility" to be "little known."

The symbol "*" means "glob" (a typical UNIX name: the name tells you just what it does, right?). Let me illustrate with our friend, "cat." Suppose I want to collect a set of files named paper.1 paper.2 paper.3 and paper.4 into one file. I can do this with cat:

cat paper.1 paper.2 paper.3 paper.4> newfilename

UNIX provides "glob" to make the job even easier. Glob means to expand the filename by examining all files in the directory to find all that fit. Thus, I can redo my command as

cat paper*>newfilename

where paper* expands to {paper.1 paper.2 paper.3 paper.4}. This is one of the typical virtues of UNIX; there are a number of quite helpful functions. But suppose I had decided to name this new file "paper.all"—pretty logical name.

cat paper*>paper.all

Disaster. In this case, paper* expands to paper.1 paper.2 paper.3 paper.4 paper.all, and so I am filling up a file from itself:

cat paper.1 paper.2 paper.3 paper.4 paper.all>paper.all

Eventually the file will burst. Does UNIX check against this, or at least give a warning? No such luck. The manual doesn't alert users to this either, although it does warn of another, related infelicity: "Beware of 'cat a b > a' and 'cat b a > a', which destroy the input files before reading them." Nice of them to tell us.

The command to remove all files that start with the word "paper"

rm paper*

becomes a disaster if a space gets inserted by accident:

rm paper *

for now the file "paper" is removed, as well as every file in the entire directory (the power of glob). Why is there not a check against such things? I finally had to alter my version of rm so that when I said to remove files, they were moved to a special directory named "deleted" and preserved there until I logged off, leaving me lots of time for second thoughts and catching errors. This illustrates the power of UNIX: what other operating system would make it so easy for someone to completely change the operation of a system command? It also illustrates the trouble with UNIX: what other operating system would make it so necessary to do so? (This is no longer necessary now that we use Berkeley UNIX—more on this in a moment.)

THE SHY TEXT EDITOR

The standard text editor is called Ed. I spent a year using it as an experimental vehicle to see how people deal with such confusing things. Ed's major property is his shyness; he doesn't like to talk. You invoke Ed by saying, reasonably enough, "ed." The result is silence: no response, no prompt, no message, just silence. Novices are never sure what that silence means. Ed would be a bit more likable if he answered, "thank you, here I am," or at least produced a prompt character, but in UNIX silence is golden. No response means that everything is okay; if something had gone wrong, it would have told you.

Then there is the famous append mode error. To add text into the buffer, you have to enter "append mode." To do this, you simply type "a," followed by RETURN. Now everything that is typed on the terminal goes into the buffer. (Ed, true to form, does not inform you that it is now in append mode: when you type "a" followed by "RETURN" the result is silence.) When you are finished adding text, you are supposed to type a line that "contains only a . on it." This gets you out of append mode.

Want to bet on how many extra periods got inserted into text files, or how many commands got inserted into texts, because the users thought that they were in command mode and forgot that they had not left append mode? Does Ed tell you when you have left append mode? Hah! This problem is so obvious that even the designers recognized it, but their reaction, in the tutorial introduction to Ed, was merely to note wryly that even experienced programmers make this mistake. While they may be able to see humor in the problem, it is devastating to the beginning secretary, research assistant or student trying to use UNIX as a word processor, an experimental tool, or just to learn about computers.

How good is your sense of humor? Suppose you have been working on a file for an hour and then decide to quit work, exiting Ed by saying "q." The problem is that Ed would promptly quit. Woof, there went your last hour's work. Gone forever. Why, if you had wanted to save it you would have said so, right? Thank goodness for all those other people across the country who immediately rewrote the text editor so that we normal people (who make errors) have some other choices besides Ed, editors that tell you politely when they are working, that tell you if they are in append or command mode, and that don't let you quit without saving your file unless you are first warned, and then only if you say you really mean it.

As I wrote this paper I sent out a message on our networked message system and asked my colleagues to tell me of their favorite peeves. I got a lot of responses, but there is no need to go into detail about them; they all have much the same flavor, mostly commenting about the lack of consistency and the lack of interactive feedback. Thus, there is no standardization of means to exit programs (and because the "shell" is just another program as far as the system is concerned, it is very easy to log yourself off the system by accident). There are very useful pattern matching features (such as the "glob" function), but the shell and the different programs use the symbols in inconsistent ways. The UNIX copy command (cp) and the related C programming language "string-copy" (strcpy) reverse the meaning of their arguments, and UNIX move (mv) and copy (cp) operations will destroy existing files without any warning. Many programs take special "argument flags" but the manner of specifying the flags is inconsistent, varying

Ed's major property is his shyness; he doesn't like to talk.

ANOTHER VIEW

Prof. Norman praises the UNIX system design but makes a number of caustic remarks about command names and other aspects of the human interface. These might be ignored, since he has no experimental tests to justify them; or they might even be taken as flattery of UNIX, since he does not name any system he likes better; but some of his comments are worth discussing.

Most of the command names Norman points to are indeed strange; some, such as dsw, were removed several years ago (by the way, to repair the discourtesy of the manual, dsw meant "delete from switches"). However, it is not clear that it makes much difference what the command names are. T. K. Landauer, K. Galotti, and S. Hartwell recently tried teaching people a version of the editor in which "append," "delete," and "substitute" were called "allege," "cypher," and "deliberate." It didn't seem to have much effect on learning time, and afterwards the users would say things like "I alleged three lines and deliberated a comma on the last one" just like subjects who had learned the ordinary version of the editor ("A Computer Command By Any Other Name: A Study of Text Editing Terms," available from the authors at Bell Labs.)

In addition to the amusing but secondary discussion of command names, Prof. Norman does raise some significant issues: (1) whether systems should be verbose or terse; (2) whether they should have a few general commands or many special-purpose ones; and (3) whether they should try to anticipate typical mistakes. Experimental results on these issues would be welcome; meanwhile, the armchair evidence is not all on one side.

UNIX is undoubtedly near an extreme of terseness, partly because it was originally designed for slow hardcopy terminals. However, the terseness is very valuable when connecting processes. If the command that lists the logged-on users prints a heading above the list, you can't tell how many users are on by feeding the command output to a line counter. If the editor types acknowledgments now and then, its output may not be directly usable as input somewhere else. Of course, you could feed it through something which strips off the extra remarks, but presumably that program would add its own chatty messages.

Prof. Norman complains about using "cat" for a command which prints files, rather than having a special-purpose command for the purpose (there is one, by the way: "pg"). Having a few general-purpose commands is a definite aid to system learning. In practice, it is not the novices who use the alternatives to "cat"; it is the experts, who want something better adapted to their special needs and are willing to learn another command. In general, people are quite good at recognizing special uses of commands in context, probably because it is a lot like things they have to do every day in English. To take an analogy from programming languages, one doubts that Prof. Norman would advocate a separate operator for "+" in integer arithmetic and "+" in floating point arithmetic. There are many advantages to a small, general-purpose set of commands. Having only one way to do any given task minimizes software maintenance while maximizing the ability of two users to help each other with advice. But this implies that whenever a general command and a specific command do the same thing, the specific command should be removed. It would be a definite service if the "cognitive engineers" could tell us how many commands are reasonable, to give some guidance on, for example, whether "merge" should be a separate command or an option on "sort" (on UNIX it is a sort option) and whether the terminal drivers should be separate commands or options on a graphics output command (on UNIX they are separate). The best rule of thumb we have today is that designing the system so that the manual will be as short as possible minimizes learning effort.

Prof. Norman seems to think that the computer should try to anticipate user problems, and refuse commands that appear dangerous. The computer world is undoubtedly moving in this direction; strong typing in programming languages is a good example. The "ed" editor has warned for some years if the user tries to quit without writing a file. The "vi" editor has an "undo" feature, regardless of the complexity of the command which has been executed. Such a facility is undoubtedly the best solution. It lets the user recognize his mistakes and back out of them, rather than expecting the system to foresee them. It is really not possible to anticipate the infinite variety of possible user mistakes; as every programmer who has ever debugged anything knows, it is hard enough to deal with the correct inputs to a program. Human hindsight is undoubtedly better than machine foresight.

A large number of Prof. Norman' comments are pleas for consistency. UN has grown more than it has been built, with many people from many places tossing software into the system. The ability of the system to accept commands so easily is one of its main strengths. However, it results in command names like "finger" for what Bell Labs called "whois" (identify a user) and "more," "cat," or "pg" for what Prof. Norman would rather call "list." The thought of a UNIX Command Standardization Committee trying to impose rules on names is a frightening alternative. Much of the attractiveness of UNIX derives from its hospitality to new commands and features. This has also meant a diversity of names and styles. To some of us this diversity is attractive, while to others the diversity is frustrating, but to hope for the hospitality without the diversity is unrealistic.

—Michael Lesk
Bell Labs
Murray Hill, N.J.

from program to program.

The version of UNIX I now use is called the Fourth Berkeley Edition for the VAX, distributed by Joy, Babaoglu, Fabry, and Sklower at the University of California, Berkeley (henceforth, Berkeley UNIX). This is both good and bad.

Among the advantages: History lists, aliases, a richer and more intelligent set of system programs (including a list program, an intelligent screen editor, an intelligent set of routines for interacting with terminals according to their capabilities), and a job control that allows one to stop jobs right in the middle, start up new ones, move things from background to foreground (and vice versa), examine files, and then resume jobs. The shell has been amplified to be a more powerful programming language, complete with file handling capabilities, if—then—else statements, while, case, and other goodies of structured programming (see box, p. 00).

Aliases are worthy of special comment. Aliases let users tailor the system to their own needs, naming things in ways they can remember; names you devise yourself are easier to recall than names provided to you. And aliases allow abbreviations that are meaningful to the individual, without burdening everyone else with your cleverness or difficulties.

To work on this paper, I need only type the word "unix," for I have set up an alias called "unix" that is defined to be equal to the correct command to change directories, combined with a call to the editor (called "vi" for "visual" on this system) on the fi

alias unix "chdir /csl/norman/papers/
CogEngineering; vi unix"

These Berkeley UNIX features have proven to be indispensable: the people in my laboratory would probably refuse to go back to standard UNIX.

The bad news is that Berkeley UNIX is jury-rigged on top of regular UNIX, so it can

There are lots of aids to memory that can be provided, but the most powerful of all is understanding.

only patch up the faults: it can't remedy them. Grep is not only still grep, but there is an egrep and an fgrep.

And the generators of Berkeley UNIX have their problems: if Bell Labs people are smug and lean, Berkeley people are cute and overweight. Programs are wordy. Special features proliferate. The system is now so large that it no longer fits on the smaller machines: our laboratory machine, a DEC 11/45, cannot hold the latest release of Berkeley UNIX (even with a full complement of memory and a reasonable amount of disk). I wrote this paper on a VAX.

LEARNING IS NOT EASY

Learning the system for setting up aliases is not easy for beginners, who may be the people who need them most. You have to set them up in a file called .cshrc, not a name that inspires confidence. The "period" in the filename means that it is invisible—the normal method of directory listing programs won't show it. The directory listing program, ls, comes with 19 possible argument flags, which can be used singly or in combinations. The number of special files that must be set up to use all the facilities is horrendous, and they get more complex with each new release from Berkeley.

It is very difficult for new users. The program names are cute rather than systematic. Cuteness is probably better than standard UNIX's lack of meaning, but there are limits. The listing program is called "more" (as in, "give me more"), the program that tells you who is on the system is called "finger," and a keyword help file—most helpful, by the way—is called "apropos." I used the alias feature to rename it "help."

One reader of a draft of this paper—a systems programmer—complained bitterly: "Such whining, hand-wringing, and general bitchiness will cause most people to dismiss it as over-emotional nonsense. . . . The UNIX system was originally designed by systems programmers for their own use and with no intention for others using it. Other hackers liked it so much that eventually a lot of them started using it. Word spread about this wonderful system, and the rest you probably know. I think that Ken Thompson and Dennis Ritchie could easily shrug their shoulders and say 'But we never intended it for other than our personal use.' "

This complaint was unique, and I sympathize with its spirit. It should be remembered, though, that UNIX is nationally distributed under strict licensing agreements. Western Electric's motives are not altogether altruistic. If UNIX had remained a simple experiment on the development of operating systems, then complaints could be made in a more friendly, constructive manner. But UNIX is more than that. It is taken as the very model of a proper operating system. And that is exactly what it is not.

In the development of the system aspects of UNIX, the designers have done a magnificent job. They have been creative, and systematic. A common theme runs through the development of programs, and by means of their file structure, the development of "pipes" and "redirection" of both input and output, plus the power of the iterative "shell" system-level commands, one can easily combine system level programs into self-tailored systems of remarkable power. For system programmers, UNIX is a delight. It is well structured, with a consistent, powerful philosophy of control and structure.

Why was the same effort not put into the design at the level of the user? The answer is complex, but one reason is the fact that there really are no well known principles of design at the level of the user interface. So, to remedy the harm I may have caused with my heavy-handed sarcasm, let me attempt to provide some positive suggestions based upon research conducted by myself and others into the principles of the human information processing system.

Cognitive engineering is a new discipline, so new that it doesn't exist, but it ought to. Quite a bit is known about the human information processing system, enough that we can specify some basic principles for designers. People are complex entities and can adapt to almost anything. As a result, designers often design for themselves, without regard for other kinds of users.

The three most important concepts for system design are these:

1. Be consistent. A fundamental set of principles ought to be evolved and followed consistently throughout all phases of the design.

2. Provide the user with an explicit model. Users develop mental models of the devices with which they interact. If you do not provide them with one, they will make one up themselves, and the one they create is apt to be wrong.

Do not count on the user fully understanding the mechanics of the device. Both secretaries and scientists may be ignorant of the difference between the buffer, the working memory, the working files, and the permanent files of a text editor. They are apt to believe that once they have typed something into the system, it is permanently in their files. They are apt to expect more intelligence from the system than the designer knows is there. And they are apt to read into comments (or the lack of comments) more than you have intended.

Feedback is of critical importance in helping establish the appropriate mental model and in letting the user keep its current state in synchrony with the actual system.

3. Provide mnemonic aids. For most purposes it is convenient to think of human memory as consisting of two parts: a short-term memory and a long-term memory (modern cognitive psychology is developing more sophisticated notions, but this is still a valid approximation). Five to seven items is about the limit for short-term memory. Thus, do not expect a user to remember the contents of a message for much longer than it is visible on the terminal. Long-term memory is robust, but it faces two difficulties: getting stuff in so that it is properly organized, and getting stuff out when it is needed. Learning is difficult, unless there is a good structure and it is visible to the learner.

There are lots of sensible memory aids that can be provided, but the most powerful and sensible of all is understanding. Make the command names describe the function that is desired. If abbreviations must be used, adopt a consistent policy of forming them. Do not deviate from the policy, even when it appears that a particular command warrants doing so.

System designers take note. Design the system for the person, not for the computer, not even for yourself. People are also information processing systems, with varying degrees of knowledge and experience. Friendly systems treat users as normal, intelligent adults who are sometimes forgetful and are rarely as knowledgeable about the world as they would like to be. There is no need to talk down to the user, nor to explain everything. But give the users a share in understanding by presenting a consistent view of the system. Their response will be your reward. *

Partial research support was provided by Contract N00014-79-C-0323, NR 157-437 with the Personnel and Training Research Programs of the Office of Naval Research, and was sponsored by the Office of Naval Research and the Air Force Office of Scientific Research. I thank the members of the LNR research group for their helpful suggestions and descriptions of misery. In particular, I wish to thank Phil Cohen, Tom Erickson, Jonathan Grudin, Henry Halff, Gary Perlman, and Mark Wallen for their analysis of UNIX. Gary Perlman and Mark Wallen provided a number of useful suggestions.

Donald A. Norman is professor of psychology and director of the program in cognitive science at the University of California, San Diego. He has degrees in electrical engineering from MIT and the University of Pennsylvania, and a doctorate in psychology from the University of Pennsylvania. He is the author of seven books, including *Human Information Processing*, Academic Press, N.Y., 1977.