The truth about Unix: The user interface is horrid

Donald A. Norman
Department of Psychology and Program in Cognitive Science
Center for Human Information Processing
University of California, San Diego
La Jolla, California 92093

(to appear in Datamation)
Norman, D. A. The truth about UNIX. Datamation 27, 12 (1981).

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 the large micro- mini-systems for home, small business, and educational setting. But for all 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. The motto of the designers of Unix towards the user seems to be "let the user beware."

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. I have learned to get along with the vagaries of its user interface, but 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. In this article I review both the faults of Unix and also some of the principles of Cognitive Engineering that could improve things, not just for Unix, but for computer systems in general. But first, the conclusion; Unix fails several simple tests.

Consistency: The 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. "No news is good news" is its motto, but as a result, the user can't tell what state the system is in, and essentially, is completely out of touch with things.

Cognitive Engineering: The system does not understand about normal folks, the everyday users of Unix. Cognitive capabilities are strained beyond their limits, the lack of mnemonic structures places large loads of memory, and the lack of interaction puts a strain on one's ability to retain mentally

exactly what state the system is in at any moment. (Get distracted at the wrong time and you lose your place -- and maybe your file.)

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"). To 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 straightforward enough, but in Unix 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. To the designers of Unix, however, such a solution lacks elegance. Unix has no basic listing command, but instead you must use a program meant to do something else.

In Unix, if you wanted to list the contents of a file called "HappyDays", you would use the command named "cat":

```
cat HappyDays
```

Why cat? Why not? After all, said Humpty Dumpty 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 are rather fond of the principle that special purpose functions can be avoided by clever use of a small set of system primitives. Their philosophy is essentially, don't make a special function when the side-effects of other functions will do what you want. But there are several reasons why this philosophy is bad;

1. A psychological principle is that names should reflect function, else the names for the function will be difficult to recall;

2. Side-effects can be used for virtue, but they can also have unwarranted effects. Thus, if cat is used unwisely, it will destroy files (more on this in a moment).

3. Special functions can do nice things for users, such as stop at the end of screens, or put on page headings, or transform non-printing 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 if it did that, why that 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? gee, it seems so obvious to us. Just like

_f_u_n_c_t_i_o_n	U_n_i_x _c_o_m_m_a_n_d _n_a_m_e
c compiler	cc
change working directory	chdir (cd in Berkeley Unix)
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, unless it is the editor which is abbreviated "ed" and concatenate which is "cat" or "date" or "echo" which are not abbreviated at all. Note how useful those 2 letter abbreviations are. See how much time and effort is saved typing only 2 letters instead of -- heaven forbid -- 4 letters. So what is a little inconsistency among friends, especially when you can save almost 400 milliseconds per command.

Similar statements apply to 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 /csl/norman/papers/CogEngineering/unix. The name of the top directory is "/", and csl, 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, "usr" and "mnt." And there are "bin," "lib," and "tmp." (What mere mortals might call 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 doesn't abbreviate everything as 2 or 3 letter commands. It keeps "grep" at its full four letters, when it could have been abbreviated as "gr" or "gp". (What does grep mean, you may ask. "Global REgular expression, Print" -- at least that's the best we can invent, the

manual doesn't even try to say. 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. But that takes me from my topic.)

Do I dare tell you about "dsw"? This also turns out to be an important routine. Suppose you accidentally create a file whose name has a non-printing character in it. How can you remove it? The command that lists the files on your directory won't show non-printing characters. And if the character is a space (or worse, a "*"), "rm" (the program that removes files) won't accept it. "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 (or is it to keep it or delete it -- which action does "yes" mean?). How do you remember dsw? What on earth does the name stand for? The Unix people won't tell; the manual smiles its wry smile of the professional programmer and says "The name dsw is a carryover from the ancient past. Its etymology is amusing." (The implication, I guess, is that true professionals never need to use such a program, but they are allowing it to be released for us novices out in the real world.)

Verification of my charges comes from the experiences of the many users of Unix, and from the modifications that other people have been forced to make to the system. Thus, the system 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). They provide a listing program that provides all the features I claim a user would want (except a sensible name -- but Berkeley Unix even makes it easy to change system names to anything you prefer).

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 better keep that one, type in yes, and destroy it forever. Does dsw warn you? Of course not. Does dsw even document itself when it starts, to remind you which way is which? Of course not. That would be talkative, and true Unix programmers are never talkative. (Berkeley Unix, has finally killed dsw, saying "This little known, but indispensible facility has been taken over...". That is a fitting commentary on standard Unix: a system that allows an "indispensible facility" to be "little known.")

The symbol "*" means "glob" (a typical Unix name: the name tells you just what action it does, right?). Let me illustrate with our friend, "cat." Suppose I want to put together 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". After all, this is a pretty logical name, I am combining the separate individual files into a new one that contains "all" the previous ones.

cat paper* > paper.all

Disaster. I will probably blow up the system. 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 nice friendly Unix check against this, or at least give a warning? Oh no, that would be against the policy of Unix. The manual doesn't bother warning against 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 actually only moved to a special directory named "deleted" and they didn't actually get deleted until I logged off. This gave me lots of time for second thoughts and for catching errors. This also 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 for their own personal satisfaction? This also illustrates the evils of 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 standard text editor is called Ed. What a problem that turned out to be. It was so lovely that I spent a whole year using it as an experimental vehicle to see how people dealt with such awful 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. Novice are never sure what that silence means. What did they do wrong, they wonder. Why doesn't Ed say "thank you, here I am" (or at least produce a prompt character)? No, not Unix with the philosophy that silence is golden. No

response means that everything is ok. If something had gone wrong, then it would have told you (unless the system died, of course, but that couldn't happen could it?).

Then there is the famous append mode error. To add text into the buffer, you have to enter "append mode." To do this, one simply types "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, no message, no comment, nothing.) 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 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 knew about it, but their reaction was to laugh: "hah-hah, see Joe cry. He just made the append mode error again." In the tutorial introduction to Ed, written at Bell Labs, the authors joke about it. Even experienced programmers get screwed this way, they say, hah hah, isn't that funny. Well, it may be funny to the experienced programmer, but it is devastating to the beginning secretary or research assistant or student who is trying to use friendly Unix as a word processor, or as an experimental tool, or just to learn about computers. Anyone can use Unix says the programmer, all you need is a sense of humor.

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 would have 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 us normal people (who make errors) had some other choices besides Ed, editors that told you politely when they were working, that would tell you if they were in append or command mode, and that wouldn't let you quit without saving your file unless you were first warned, and then only if you said you really meant it. I could go on.

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 about them, mostly commenting about lack of consistency, about 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 "stringcopy" (strcpy) have reversed order of 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 from program to program. As I said, I could go on.

The good news is that we don't use standard Unix: we use Berkeley Unix. History lists, aliases, a much richer and more intelligent set of system programs, including a list program, an intelligent screen editor, a 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, startup new ones, move things from background to foreground (and vice versa), examine files, and then resume jobs. And the shell has been amplified to be a more powerful programming language, complete with file handling capabilities, if--then--else statements, while, case, and all the other goodies of structured programming (see the accompanying box on Unix).

Aliases are worthy of special comment. Aliases let the user tailor the system to their own needs, naming things in ways they themselves can remember: self-generated names are indeed easier to remember than arbitrary names given 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 file: 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 only patch up the faults: it can't remedy them. Grep is not only still grep, but there is an egrep and an fgrep. But worse, 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. Aliases -- the system for setting them up is not easy to for beginners (who may be the people who need them most). You have to set them up in a file called .cshrc, a name not chosen to inspire 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, that can be used singly or in combinations. The number of special files that must be set up to use all the facilities is horrendus, and they get more complex with each new release from Berkeley. It is vey difficult on new users. The program names are cute rather than systematic. Cuteness is probably better than the lack of meaning of standard Unix, but there are be 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." Apropos! who can remember that? Especially when you need it most. I had to make up an alias called "help" which calls all of the help commands Berkeley provides, but whose names I can never remember (apropos, whatis, whereis, which).

The system is now so wordy and 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 disc). I write this paper on a Vax.

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, etc, 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.'"

All the other users of Unix who have read drafts of this paper agreed with me. Indeed, their major reaction was to forward examples of problems that I had not covered. This complaint was unique. I do sympathize with the spirit of the complaint. He is correct, but ... The "but" is that the system is nationally distributed, under strict licensing agreements, with a very high charge to industry, and nominal charges to educational institutes. Western Electric doesn't mind getting a profit, but they have not attempted to worry about the product. If Unix were still what it started to be, a simple experiment on the development of operating systems, then the complaints I list 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 combine system level programs into self-tailored systems of remarkable power with remarkable ease.

In the development of the user interface aspects of Unix, the designers have been failures. They have been difficult and derisive. A common theme runs through the commands: don't be nice to the casual user -- write the system for the dedicated expert. The system is a recluse. It uses weird names, and it won't speak to you, not even if spoken to. For system programmers, Unix is a delight. It is well structured, with a consistent, powerful philosophy of control and structure. My complaint is simple: why was not the same effort put into the design at the level of the user? The answer to my complaint is a bit more complex. There really are no well known principles of design at the level of the user interface. So, to remedy the harm that I may have caused by my heavy-handed sarcasm, let me attempt to provide some positive suggestions based upon the research that has been done by me and by 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 are often sloppy, for they can design for themselves without realizing the difficulties that will be faced by other users. Moreover,

there are different levels of users: people with a large amount of knowledge of the device they are about to use are quite different from those who lack a basic understanding. Experts are different than novices. And the expert who is normally skilled at the use of some systems but who has not used it for awhile is at a peculiar level of knowledge, neither novice nor expert.

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 make up is apt to be wrong. Do not count on the user fully understanding the mechanics of the device. Secretaries and scientists alike will share a lack of knowledge of a computer system. The users are not apt to understand 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, both in helping to establish the appropriate mental model and in letting the user keep its current state in synchrony with the actual system.

3. Provide mnemonic aids. Human memory is a fragile thing. Actually, 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 than this simple two-stage one, but this is still a valid approximation). Short-term memory is, as the name suggests, limited in duration and quantity: about five to seven items is the limit. 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, so that it can be found when needed. Learning is difficult, unless there is a good structure, and it is visible to the learner. The system designer must provide sensible assistance to the user so that the material can be structured. There are lots of sensible memory aids that can be provided, but the most powerful and sensible of all is understanding. Make a system so that it can be understood and the memory follows with ease. Make the command names ones that can be understood, where the names follow from the function that is desired. If abbreviations must be used, adopt a consistent policy of forming the abbreviations. Do not deviate from the policy, even when it appears that it would be easier for a particular command to deviate:

> inconsistency is an evil. Remember the major problem of any large-scale memory is finding the information that is sought, even if the information is there someplace. We retrieve things from memory by starting off with some description of the information we seek, use that description to enter their memory system in an attempt to match against the desired information. If the designer uses cute names and non-standard abbreviations, our ability to generate a valid description is impaired. As a result, the person who is not expert and current in the use of the system is apt to flounder.

There are many ways of formatting information on terminals to provide useful memory and syntax aids for users. With today's modern terminals, it is possible to use menus, multiple screens and windows, highlighted areas, and with full duplex systems, automatic or semi-automatic command completion systems. The principles for these systems are under active study by a number of groups, but none are directly relevant to my critique of the UNIX operating system. UNIX is designed specifically so that it can be used with a wide variety of terminals, including hard copy terminals.

The problem with Unix is more fundamental. Unix does not provide the user with a systematic set of principles; it does not provide a simple, consistent mental model for the user, consistent not only in the shell but in the major system programs and languages; it does not provide the user with simple memory aids that can be used to learn the system structure and then, when one is not completely current in the use of a particular command, still to be able to retrieve (or better, derive) what is needed. There are essentially no user help files, despite the claim that all the documentation is on-line via the command named man (for manual, of course). But "man" requires you to know the name of the command you want information about, although it is the name that is probably just the information you are seeking.

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, varying degrees of experience. Remember, people's short-term memories are limited in size, and they learn and retrieve things best when there is a consistent reason for the name, the function, and the syntax. Friendly systems treat users as intelligent adults who, like normal adults, are forgetful, distracted, thinking of other things, and not quite as knowledgeable about the world as they themselves would like to be. Treat the user with intelligence. There is no need to talk down to the user, nor to explain everything. But give the user a share in understanding by presenting a consistent view of the system.