Program design in the UNIX† environment

Rob Pike

Brian W. Kernighan

ABSTRACT

Much of the power of the UNIX operating system comes from a style of program design that makes programs easy to use and, more important, easy to combine with other programs. This style has been called the use of *software tools*, and depends more on how the programs fit into the programming environment — how they can be used with other programs — than on how they are designed internally. But as the system has become commercially successful and has spread widely, this style has often been compromised, to the detriment of all users. Old programs have become encrusted with dubious features. Newer programs are not always written with attention to proper separation of function and design for interconnection. This paper discusses the elements of program design, showing by example good and bad design, and indicates some possible trends for the future.

The UNIX operating system has become a great commercial success, and is likely to be the standard operating system for microcomputers and some mainframes in the coming years.

There are good reasons for this popularity. One is portability: the operating system kernel and the applications programs are written in the programming language C, and thus can be moved from one type of computer to another with much less effort than would be involved in recreating them in the assembly language of each machine. Essentially the same operating system therefore runs on a wide variety of computers, and users needn't learn a new system when new hardware comes along. Perhaps more important, vendors that sell the UNIX system needn't provide new software for each new machine; instead, their software can be compiled and run without change on any hardware, which makes the system commercially attractive. There is also an element of zealotry: users of the system tend to be enthusiastic and to expect it wherever they go; the students who used the UNIX system in university a few years ago are now in the job market and often demand it as a condition of employment.

But the UNIX system was popular long before it was even portable, let alone a commercial success. The reasons for that are more interesting.

Except for the initial PDP-7 version, the UNIX system was written for the DEC PDP-11, a machine that was (deservedly) very popular. PDP-11's were powerful enough to do real computing, but small enough to be affordable by small organizations such as academic departments in universities.

The early UNIX system was smaller but more effective and technically more interesting than competing systems on the same hardware. It provided a number of innovative applications of computer science, showing the benefits to be obtained by a judicious blend of theory and practice. Examples include the `yacc` parser-generator, the `diff` file comparison program, and the pervasive use of regular expressions to describe string patterns. These led in turn to new programming languages and interesting software for applications like program development, document preparation and circuit design.

Since the system was modest in size, and since essentially everything was written in C, the software was easy to modify, to customize for particular applications or merely to support a view of the world

† UNIX is a trademark of Bell Laboratories.

different from the original. (This ease of change is also a weakness, of course, as evidenced by the plethora of different versions of the system.)

Finally, the UNIX system provided a new style of computing, a new way of thinking of how to attack a problem with a computer. This style was based on the use of *tools*: using programs separately or in combination to get a job done, rather than doing it by hand, by monolithic self-sufficient subsystems, or by special-purpose, one-time programs. This has been much discussed in the literature, so we don't need to repeat it here; see [1], for example.

The style of use and design of the tools on the system are closely related. The style is still evolving, and is the subject of this essay: in particular, how the design and use of a program fit together, how the tools fit into the environment, and how the style influences solutions to new problems. The focus of the discussion is a single example, the program `cat`, which concatenates a set of files onto its standard output. `cat` is a simple program, both in implementation and in use; it is essential to the UNIX system; and it is a good illustration of the kinds of decisions that delight both supporters and critics of the system. (Often a single property of the system will be taken as an asset or as a fault by different audiences; our audience is programmers, because the UNIX environment is designed fundamentally for programming.) Even the name `cat` is typical of UNIX program names: it is short, pronounceable, but not conventional English for the job it does. (For an opposing viewpoint, see [2].) Most important, though, `cat` in its usages and variations exemplifies UNIX program design style and how it has been interpreted by different communities.

```
11/3/71                                                          CAT (I)

NAME          c_a_t_ -- concatenate and print

SYNOPSIS      c_a_t_ f_i_l_e_1_ ...

DESCRIPTION   c_a_t_ reads each file in sequence and writes it on
              the standard output stream.  Thus:

                 c_a_t_ f_i_l_e_

              is about the easiest way to print a file.  Also:

                 c_a_t_ f_i_l_e_1_ f_i_l_e_2_ >f_i_l_e_3_

              is about the easiest way to concatenate files.

              If no input file is given c_a_t_ reads from the
              standard input file.

FILES         --

SEE ALSO      pr, cp

DIAGNOSTICS   none; if a file cannot be found it is ignored.

BUGS          --

OWNER         ken, dmr
```

Figure 1: Manual page for **cat**, UNIX 1st Edition, November, 1971

Figure 1 is the manual page for `cat` from the UNIX 1st Edition manual. Evidently, `cat` copies its input to its output. The input is normally taken from a sequence of one or more files, but it can come from the standard input. The output is the standard output. The manual suggests two uses, the general file copy:

```
cat file1 file2 >file3
```

and printing a file on the terminal:

```
cat file
```

The general case is certainly what was intended in the design of the program. Output redirection (provided by the > operator, implemented by the UNIX shell) makes `cat` a fine general-purpose file concatenator and a valuable adjunct for other programs, which can use `cat` to process filenames, as in:

```
cat file file2 ... | other-program
```

The fact that `cat` will also print on the terminal is a special case. Perhaps surprisingly, in practice it turns out that the special case is the main use of the program.†

The design of `cat` is typical of most UNIX programs: it implements one simple but general function that can be used in many different applications (including many not envisioned by the original author). Other commands are used for other functions. For example, there are separate commands for file system tasks like renaming files, deleting them or telling how big they are. Other systems instead lump these into a single ‘‘file system’’ command with an internal structure and command language of its own. (The PIP file copy program found on operating systems like CP/M or RSX-11 is an example.) That approach is not necessarily worse or better, but it is certainly against the UNIX philosophy. Unfortunately, such programs are not completely alien to the UNIX system — some mail-reading programs and text editors, for example, are large self-contained ‘‘subsystems’’ that provide their own complete environments and mesh poorly with the rest of the system. Most such subsystems, however, are usually imported from or inspired by programs on other operating systems with markedly different programming environments.

There are some significant advantages to the traditional UNIX system approach. The most important is that the surrounding environment — the shell and the programs it can invoke — provides a uniform access to system facilities. Filename argument patterns are expanded by the shell for all programs, without prearrangement in each command. The same is true of input and output redirection. Pipes are a natural outgrowth of redirection. Rather than decorate each command with options for all relevant pre- and post-processing, each program expects as input, and produces as output, concise and header-free textual data that connects well with other programs to do the rest of the task at hand. It takes some programming discipline to build a program that works well in this environment — primarily, to avoid the temptation to add features that conflict with or duplicate services provided by other commands — but it’s well worthwhile.

Growth is easy when the functions are well separated. For example, the 7th Edition shell was augmented with a backquote operator that converts the output of one program into the arguments to another, as in

```
cat ‘cat filelist‘
```

No changes were made in any other program when this operator was invented; because the backquote is interpreted by the shell, all programs called by the shell acquire the feature transparently and uniformly. If special characters like backquotes were instead interpreted, even by calling a standard subroutine, by each program that found the feature appropriate, every program would require (at least) recompilation whenever someone had a new idea. Not only would uniformity be hard to enforce, but experimentation would be harder because of the effort of installing any changes.

The UNIX 7th Edition system introduced two changes in `cat`. First, files that could not be read, either because of denied permissions or simple non-existence, were reported rather than ignored. Second, and less desirable, was the addition of a single optional argument `-u`, which forced `cat` to unbuffer its output (the reasons for this option, which has disappeared again in the 8th Edition of the system, are technical and irrelevant here.)

But the existence of one argument was enough to suggest more, and other versions of the system

---

† The use of `cat` to feed a single input file to a program has to some degree superseded the shell’s < operator, which illustrates that general-purpose constructs — like `cat` and pipes — are often more natural than convenient special-purpose ones.

soon embellished `cat` with features. This list comes from `cat` on the Berkeley distribution of the UNIX system:

- `-s` strip multiple blank lines to a single instance
- `-n` number the output lines
- `-b` number only the non-blank lines
- `-v` make non-printing characters visible
  - `-ve` mark ends of lines
  - `-vt` change representation of tab

In System V, there are similar options and even a clash of naming: `-s` instructs `cat` to be silent about non-existent files. But none of these options are appropriate additions to `cat`; the reasons get to the heart of how UNIX programs are designed and why they work well together.

It's easy to dispose of (Berkeley) `-s`, `-n` and `-b`: all of these jobs are readily done with existing tools like `sed` and `awk`. For example, to number lines, this `awk` invocation suffices:

```
awk '{ print NR "\t" $0 }' filenames
```

If line-numbering is needed often, this command can be packaged under a name like `linenumber` and put in a convenient public place. Another possibility is to modify the `pr` command, whose job is to format text such as program source for output on a line printer. Numbering lines is an appropriate feature in `pr`; in fact UNIX System V `pr` has a `-n` option to do so. There never was a need to modify `cat`; these options are gratuitous tinkering.

But what about `-v`? That prints non-printing characters in a visible representation. Making strange characters visible is a genuinely new function, for which no existing program is suitable. (''`sed -n l`'', the closest standard possibility, aborts when given very long input lines, which are more likely to occur in files containing non-printing characters.) So isn't it appropriate to add the `-v` option to `cat` to make strange characters visible when a file is printed?

The answer is ''No.'' Such a modification confuses what `cat`'s job is — concatenating files — with what it happens to do in a common special case — showing a file on the terminal. A UNIX program should do one thing well, and leave unrelated tasks to other programs. `cat`'s job is to *collect* the data in files. Programs that collect data shouldn't *change* the data; `cat` therefore shouldn't transform its input.

The preferred approach in this case is a separate program that deals with non-printable characters. We called ours `vis` (a suggestive, pronounceable, non-English name) because its job is to make things visible. As usual, the default is to do what most users will want — make strange characters visible — and as necessary include options for variations on that theme. By making `vis` a separate program, related useful functions are easy to provide. For example, the option `-s` strips out (i.e., discards) strange characters, which is handy for dealing with files from other operating systems. Other options control the treatment and format of characters like tabs and backspaces that may or may not be considered strange in different situations. Such options make sense in `vis` because its focus is entirely on the treatment of such characters. In `cat`, they require an entire sub-language within the `-v` option, and thus get even further away from the fundamental purpose of that program. Also, providing the function in a separate program makes convenient options such as `-s` easier to invent, because it isolates the problem as well as the solution.

One possible objection to separate programs for each task is efficiency. For example, if we want numbered lines and visible characters it is probably more efficient to run the one command

```
cat -n -v file
```

than the two-element pipeline

```
linenumber file | vis
```

In practice, however, `cat` is usually used with no options, so it makes sense to have the common cases be the efficient ones. The current research version of the `cat` command is actually about five times faster than the Berkeley and System V versions because it can process data in large blocks instead of the byte-at-time processing that might be required if an option is enabled. Also, and this is perhaps more important, it is hard to imagine any of these examples being the bottleneck of a production program. Most of the real

time is probably taken waiting for the user's terminal to display the characters, or even for the user to read them.

Separate programs are not always better than wider options; which is better depends on the problem. Whenever one needs a way to perform a new function, one faces the choice of whether to add a new option or write a new program (assuming that none of the programmable tools will do the job conveniently). The guiding principle for making the choice should be that each program does one thing. Options are appropriately added to a program that already has the right functionality. If there is no such program, then a new program is called for. In that case, the usual criteria for program design should be used: the program should be as general as possible, its default behavior should match the most common usage, and it should cooperate with other programs.

Let's look at these issues in the context of another problem, dealing with fast terminal lines. The first versions of the UNIX system were written in the days when 150 baud was ''fast,'' and all terminals used paper. Today, 9600 baud is typical, and hard-copy terminals are rare. How should we deal with the fact that output from programs like `cat` scrolls off the top of the screen faster than one can read it?

There are two obvious approaches. One is to tell each program about the properties of terminals, so it does the right thing (whether by option or automatically). The other is to write a command that handles terminals, and leave most programs untouched.

An example of the first approach is Berkeley's version of the `ls` command, which lists the filenames in a directory. Let us call it `lsc` to avoid confusion. The 7th Edition `ls` command lists filenames in a single column, so for a large directory, the list of filenames disappears off the top of the screen at great speed. `lsc` prints in columns across the screen (which is assumed to be 80 columns wide), so there are typically four to eight times as many names on each line, and thus the output usually fits on one screen. The option `-1` can be used to get the old single-column behavior.

Surprisingly, `lsc` operates differently if its output is a file or pipe:

```
lsc
```

produces output different from

```
lsc | cat
```

The reason is that `lsc` begins by examining whether or not its output is a terminal, and prints in columns only if it is. By retaining single-column output to files or pipes, `lsc` ensures compatibility with programs like `grep` or `wc` that expect things to be printed one per line. This *ad hoc* adjustment of the output format depending on the destination is not only distasteful, it is unique — no standard UNIX command has this property.

A more insidious problem with `lsc` is that the columnation facility, which is actually a useful, general function, is built in and thus inaccessible to other programs that could use a similar compression. Programs should not attempt special solutions to general problems. The automatic columnation in `lsc` is reminiscent of the ''wild cards'' found in some systems that provide filename pattern matching only for a particular program. The experience with centralized processing of wild cards in the UNIX shell shows overwhelmingly how important it is to centralize the function where it can be used by all programs.

One solution for the `ls` problem is obvious — a separate program for columnation, so that columnation into say 5 columns is just

```
ls | 5
```

It is easy to build a first-draft version with the multi-column option of `pr`. The commands `2`, `3`, etc., are all links to a single file:

```
pr -$0 -t -l1 $*
```

`$0` is the program name (`2`, `3`, etc.), so `-$0` becomes `-`*n* where *n* is the number of columns that `pr` is to produce. The other options suppress the normal heading, set the page length to 1 line, and pass the arguments on to `pr`. This implementation is typical of the use of tools — it takes only a moment to write, and it serves perfectly well for most applications. If a more general service is desired, such as automatically selecting the number of columns for optimal compaction, a C program is probably required, but the one-

line implementation above satisfies the immediate need and provides a base for experimentation with the design of a fancier program, should one become necessary.

Similar reasoning suggests a solution for the general problem of data flowing off screens (columnated or not): a separate program to take any input and print it a screen at a time. Such programs are by now widely available, under names like `pg` and `more`. This solution affects no other programs, but can be used with all of them. As usual, once the basic feature is right, the program can be enhanced with options for specifying screen size, backing up, searching for patterns, and anything else that proves useful within that basic job.

There is still a problem, of course. If the user forgets to pipe output into `pg`, the output that goes off the top of the screen is gone. It would be desirable if the facilities of `pg` were always present without having to be requested explicitly.

There are related useful functions that are typically only available as part of a particular program, not in a central service. One example is the history mechanism provided by some versions of the UNIX shell: commands are remembered, so it's possible to review and repeat them, perhaps with editing. But why should this facility be restricted to the shell? (It's not even general enough to pass input to programs called *by* the shell; it applies to shell commands only.) Certainly other programs could profit as well; any interactive program could benefit from the ability to re-execute commands. More subtly, why should the facility be restricted to program *input*? Pipes have shown that the output from one program is often useful as input to another. With a little editing, the output of commands such as `ls` or `make` can be turned into commands or data for other programs.

Another facility that could be usefully centralized is typified by the editor escape in some mail commands. It is possible to pick up part of a mail message, edit it, and then include it in a reply. But this is all done by special facilities within the `mail` command and so its use is restricted.

Each such service is provided by a different program, which usually has its own syntax and semantics. This is in contrast to features such as pagination, which is always the same because it is only done by one program. The editing of input and output text is more environmental than functional; it is more like the shell's expansion of filename metacharacters than automatic numbering of lines of text. But since the shell does not see the characters sent as input to the programs, it cannot provide such editing. The `emacs` editor[3] provides a limited form of this capability, by processing all UNIX command input and output, but this is expensive, clumsy, and subjects the users to the complexities and vagaries of yet another massive subsystem (which isn't to criticize the inventiveness of the idea).

A potentially simpler solution is to let the terminal or terminal interface do the work, with controlled scrolling, editing and retransmission of visible text, and review of what has gone before. We have used the programmability of the Blit terminal[4] — a programmable bitmap graphics display — to capitalize on this possibility, to good effect.

The Blit uses a mouse to point to characters on the display, which can be edited, rearranged and transmitted back to the UNIX system as though they had been typed on the keyboard. Because the terminal is essentially simulating typed input, the programs are oblivious to how the text was created; all the features discussed above are provided by the general editing capabilities of the terminal, with no changes to the UNIX programs.

There are some obvious direct advantages to the Blit's ability to process text under the user's control. Shell history is trivial: commands can be selected with the mouse, edited if desired, and retransmitted. Since from the terminal's viewpoint all text on the display is equivalent, history is limited neither to the shell nor to command input. Because the Blit provides editing, most of the interactive features of programs like `mail` are unnecessary; they are done easily, transparently and uniformly by the terminal.

The most interesting facet of this work, however, is the way it removes the need for interactive features in programs; instead, the Blit is the place where interaction is provided, much as the shell is the program that interprets filename-matching metacharacters. Unfortunately, of course, programming the terminal demands access to a part of the environment off-limits to most programmers, but the solution meshes well with the environment and is appealing in its simplicity. If the terminal cannot be modified to provide the capabilities, a user-level program or perhaps the UNIX kernel itself could be modified fairly easily to do roughly what the Blit does, with similar results.

The key to problem-solving on the UNIX system is to identify the right primitive operations and to put them at the right place. UNIX programs tend to solve general problems rather than special cases. In a very loose sense, the programs are orthogonal, spanning the space of jobs to be done (although with a fair amount of overlap for reasons of history, convenience or efficiency). Functions are placed where they will do the most good: there shouldn't be a pager in every program that produces output any more than there should be filename pattern matching in every program that uses filenames.

One thing that UNIX does not need is more features. It is successful in part because it has a small number of good ideas that work well together. Merely adding features does not make it easier for users to do things — it just makes the manual thicker. The right solution in the right place is always more effective than haphazard hacking.

1. B. W. Kernighan and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall (1984).
2. D. Norman, ''The Truth about UNIX,'' *Datamation* (November, 1981).
3. James Gosling, ''UNIX Emacs,'' CMU internal memorandum (August, 1982).
4. R. Pike, ''The Blit: A Multiplexed Graphics Terminal,'' *Bell System Technical Journal* (this issue, 1984).