Unix utilities, Part 1

Yet another new component architecture

Peter Seebach (unixcomponents@seebs.plethora.net)
Freelance writer
May 2001

Sometimes it happens that the more things change, the more they shouldn't. Peter Seebach introduces us to a fascinating component architecture that gets back to the basics.

Contents:

Unix as a component architecture
Easy to implement
Why uniq(1) can't sort(1)
Resources
About the author
Rate this article

Imagine a component architecture that has hundreds of widely used, standardized components. These components have a single, standardized interface. A new component doesn't need to "announce" its interface. The standard interface is simple and universal. Components don't need to include any code to handle network connections or other interface issues. There's a separate "wrapper" component which handles distributed work for you.

Programs in this system can often, even *typically*, be written in a single line of code. Inexperienced users can be taught to write simple programs in a matter of minutes. Beginners often produce simple programs without effort within their first day using it.

This system exists, and has for more than 20 years. It's called Unix.

The elements of the Unix environment have been called *software tools*. A software tool is very much like a component, but a bit different. Throughout this piece, we'll refer to components as "tools," but remember that you can almost always substitute "component" for "tool." In Unix, the standard shell utilities are all tools. So is the shell. A tool takes a standard data format -- lines of textual input -- and produces output in the same format. No special code is needed to handle data types; everyone agrees on a simple format and does a small amount of translation. For most programs, there really isn't any translation involved.

In general, a tool takes a stream of lines on input -- this is called *standard input* -- and writes a stream of lines to output -- this is called *standard output*. (Some tools don't really consume the text one line at a time, and some don't even treat the streams merely as text, but they still generally work on input and output streams.)

Unix users learn, generally in their first few minutes on the system, to combine tools. For instance, a user may quickly learn to use

```
$ more file
```

to paginate a file. Soon, the user is using the more command as a tool:

```
$ ls | more
```

This doesn't look like programming to us -- we're used to programs being big and hard to write. And yet, it *is* programming: The user has combined two tools to produce a program to perform a task exactly to

specifications. Isn't that one of the reasons for component architectures in the first place -- ad-hoc programming and code reuse?

The Unix environment allows users to combine pieces to build functional programs with an ease unmatched by other systems. In this series of articles, I'll show how the Unix environment captures the key elements of a good component architecture, how to build tools for it, and how to use these tools.

Unix as a component architecture

Really, when it comes down to it, component architecture is all about code reuse. As a corollary, we hope to see programs written easily and quickly. Simple programs should be simple to write; they shouldn't require a lot of time to be spent copying templates and filling them in.

Unix achieves this goal (better, I think, than any other system in widespread use) through a focus on simplicity. Putting tools together is simply trivial. A program can be a simple line, and the interactive environment in which many programs are written is sufficiently expressive to be a useful interface to the system. Many systems provide some kind of limited programmability, but no more than they want you to have. Unix would never dream of limiting the programmability of a tool, and the Unix shell is a powerful and flexible programming language.

Because of this, simple programs don't need to be "built." A user who wishes to see a sorted list of records matching a given pattern doesn't need to look up interfaces to do this; the tools fall together without effort. You might have to look up usage -- as with cron, which is notorious for its lack of intuitiveness. But you never need to ask the question "how do I feed input to this?" because every one takes stdin.

Easy to implement

More interestingly, the development of a new tool takes very little effort. Unlike systems which specify powerful (and complicated) APIs for new tools, Unix gives you a simple, general interface; process input, produce output. You don't need to have separate mechanisms for different data types, and ways to identify them -- you just read input until you find a newline, process it, and write output ending with a newline, then repeat. This means that time you might have had to spend handling I/O in a more complicated API is available for getting real work done.

In some cases, of course, this isn't flexible enough, and you need a more structured format. Still, the developer and user convenience factor almost always makes up for the slightly limited interface. In practice, you can perform whatever tasks you want.

Of course, this interface isn't completely general. Sometimes you want to process things that don't look exactly like a flat file with one record per line. At this point, the designers of Unix made a crucial decision, which has turned out to be a good one: Instead of requiring every program to develop a way to handle other formats, provide tools that translate one format into another. Most utilities will accept a list of files on the command line. For example, say that you want to send a list of files on standard input. There's a tool (it's called xargs) that takes a list of files on standard input, passes them to some tool, and lets that tool produce whatever output it wants.

Similarly, when people realized that it would be useful to be able to distribute work across multiple machines, rather than forcing every tool to learn about network interfaces, they provided a "wrapper" tool, which connects to a remote machine and runs something there. The software on the other end doesn't know (or care) whether you're running it remotely or locally, it just takes lines of input and produces lines of output.

This simplification means that it's fairly simple to write a new utility to solve a specialized problem. And indeed, Unix users tend to accumulate sets of personalized utilities. One-off utilities are cheap enough to be a reasonable solution to a problem, and you don't need to justify the design time on something you can write in 30 seconds.

Once again, by simplifying all of the ease-of-use features out, we are able to come up with a system that is *really* easy to use. Code reuse isn't all about projects measured in person years; some of it is about whether you can have a program to do what you want *today* -- or even *this minute*.

Why uniq(1) can't sort(1)

This architecture also means that a tool doesn't have to be complete to be useful. For instance, there is a Unix tool called uniq, which does simple tasks involving eliminating or counting duplicate lines. This is all it ever does. It doesn't detect duplicate lines unless they are adjacent. If you want it to give a list of unique occurrences in an unsorted list, you must sort the list first.

In many architectures, the natural response would be to build uniq up until it could sort input. It would then be able to handle various kinds of records that are not lines, sort input, and keep hash tables. It would be large and complicated. It turns out that keeping the interfaces clean and separating the components produces better results.

In Unix, even writing in C (not the world's tersest language for string handling), uniq is about 4k of code, not including the copyright notice. That includes all the argument handling, including special features like "ignore the first N fields on each line."

Once the architecture makes it easy enough for a user to simply combine uniq with sort *when necessary*, there is no need for every utility to be able to sort. Indeed, very few programs sort. Those that do, do so because the data on which they sort may not be visible. (For example, ls can sort because it may sort files by timestamp even when it isn't displaying the timestamp.)

Finally, for a particularly compelling example, look at the way Unix programs that need to allow the user to edit generally solve their problem. The user can set an environment variable -- $EDITOR -- that represents the editor of choice. Almost all standard programs use this when they need editing. When the administrator of a BSD Unix box wishes to edit the password file, he runs a utility called *vipw*. All this program does is:

- Lock the password file
- Run your editor of choice on it
- Check for consistency
- Apply changes

You may be tempted to write this off (it's an administrative task, after all). However, Unix doesn't care that the task is administrative; any task that calls for an editor will let you plug an editor in. Unix doesn't make you think one way to administer, another to use, and another to program; programming is use, and administration is just like any other task.

Note that *vipw* doesn't need to be altered if you get a new editor. Similarly, your editor doesn't need to know how to lock a password file. Each program does the one thing that it's good at, and it does it well. Your editor is just the editing component of the system.

Lots of systems claim to allow you to substitute a new editor widget, a feature Unix has had working for a long time. (Later in the series, we'll see some surprising benefits this has.)

The Unix environment -- by simplifying its interface, using the same interface for everything wherever possible, and eliminating artificial distinctions, such as a distinction between "applications" and "components" -- provides all of the key benefits a component architecture is supposed to provide. In many ways, in day-to-day use, it surpasses anything else out there for ease of code reuse.

In Part 2, I'll discuss how to use Unix tools.

Resources

- *The Mythical Man Month*, Frederick P. Brooks, Addison-Wesley 1995.
- *A Quarter Century of Unix*, Peter H. Salus, Addison-Wesley 1994.
- In *Let's Make Unix Not Suck*, Miguel De Icaza does not agree that Unix utilities are components.
- Chris Browne on The Unix Philosophy.
- Thomas Scoville's well-known *UNIX as Literature* essay.

- IBM Tools and Toys for Unix page, where you can also download and study the code for these IBM open-source Unix utilities.
- An collection of links to excellent Unix resources from the IBM eServer group
- Application development tools for AIX and Unix. For all of those who think Unix is "old," we just had to link to this page, which states "Behind every cutting-edge software development, there's a Unix developer." How true that is ...

About the author

As a small child, Peter Seebach thought that "rm" was a perfectly intuitive name for the command to "ReMove" a file. He has been a Unix bigot ever since. You can reach him at unixcomponents@seebs.plethora.net.