Unix utilities, Part 4

Unix's lessons for component architectures

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

Gain a better understanding of how to achieve successful code reuse.

"Those who don't understand UNIX are doomed to reinvent it, poorly."
--Henry Spencer

Component architects can learn a lot of important design principles from studying Unix; study of Unix principles can help us realize the gains in development speed and reliability that component architecture promises.

Unix provides a beautiful example of an architecture that achieves many of the goals of component architecture, including portability and code reuse. Some of the key benefits include:

- Shell scripts are broadly portable among Unix systems. Programs in C or Perl are generally fairly portable, too.
- No other system has ever had a component used as broadly as `grep`.
- Code reuse is actually *practical* in a Unix environment.
- Ad hoc and scripting capabilities available for Unix support rapid prototyping and testing.
- Time is spent focused on solving problems, not filling out checklists of API features.

Contents:

Related content:

Unix utilities, Part 3
Unix utilities, Part 2
More dW Components resources

Lesson 1: Simple interfaces

If a simple program can't be expressed simply, something is wrong with your environment. Unix's shell interface allows many simple programs to be written in a single line, and indeed, in a single line simple enough that people will actually use this facility. The possibility of ad hoc programming, which is accessible even to people that don't think of themselves as programmers, is one of the things that makes Unix powerful.

Without a genuinely simple interface, you can't have ad hoc programming for naive users. A simple interface *implies* that the data structure at least looks simple, and that brings us to our next lesson.

Lesson 2: Human-readable data

One of the things that makes it practical to quickly develop and test applications based on the Unix tool architecture is that, in general, the intermediate steps of a process can be reviewed quickly and easily by a human being. You don't need to write a special program to display your data in a readable form -- so there's one less place a bug can hide when you can't get the output you expect. At every step in the process, you can check your work.

If you need to do something to your data that is obvious to you, but you can't figure out how to explain it to a computer, you can do it by hand, on the "real" data. You don't need to use a special translation

program to get data in and out of a human-readable format.

Lesson 3: Lots of simple components
The lesson of `uniq` and `sort`, and of `tar` and `gzip`, is that a single monolithic component that solves your whole problem is less useful to you than two or three components that can be combined one way to solve your problem today and can be combined another way to solve a different problem tomorrow.

By keeping each logically distinct task physically separate, Unix encourages users to experiment with new combinations. While you can't change the compression encoding for WinZip or StuffIt without huge changeover costs, it *is* practical to add a new compression algorithm to a Unix system.

Lesson 4: Simple API
Just as a simple front-end interface makes it possible for inexperienced users to write simple shell programs, the simple programming interface of a Unix tool makes it practical for more experienced programmers to write tools for every application they use. The easier it is to write a new component for your library, the more components you will find lying around, waiting to be used again and again.

A Unix tool needs to know about an API that involves a network only if the tool's function is to interact with the network. Otherwise, this complexity is not merely hidden, but entirely removed from the program. If you want to use the network, you use one of the tools for talking to networks.

Lesson 5: Encourage reuse
Unix encourages reuse by making it easy to reuse code, and by providing a huge variety of important pieces to reuse. An elegant and beautiful component architecture that lacks a selection of key building blocks will not encourage the reuse of code. A system where it's less work to teach a component to sort than to have it interface with an existing module for sorting is a system where no one will bother reusing the existing modules.

Lesson 6: Allow evolution
If I don't like one of the standard Unix tools, I am not obliged to use it: I can add a new one. Over time, some of these new utilities have become popular enough to be adopted into one or more of the various standards Unix vendors base their systems on.

Allowing users to evolve their own utilities and tools leads to better components for everyone else to use.

Unix doesn't provide a standard mechanism for verifying that a new feature you've come to like is available on a slightly older system, and it doesn't provide a way to handle the feature's absence. In practice, this is rarely a problem; the behavior you get if you don't specify a new option will be the same as always, and the new utility is probably portable to the old system.

Once again, keeping the logically separate parts of a utility genuinely separate is necessary for this to take place. If your editor can do its own calculation, how can you upgrade the calculator in it? In Unix, the calculator is a separate tool. So if I switch to a smarter or more flexible one, the editor doesn't even notice (let alone protest).

Lesson 7: Everything is a tool
Programmers tend to see a strong division between *components* used to build an application, and *applications*, which are in some way final products. Unix doesn't do this. Editors are tools, just like any other tool. The command-line interface is a tool. Everything can be used as a tool.

The MH mail system is a particularly stunning example of this. It provides a set of tools, each of which handles some portion of the process of reading, writing, sorting, and responding to e-mail. Since each tool is separate, and each tool is designed to allow the use of other tools with it, users can create their own customized mail processing environments -- all very different, and all well-integrated with the basic capabilities of the OS.

Lesson 8: Eschew safety
"UNIX was not designed to stop you from doing stupid things, because that would also stop you from doing clever things."
--Doug Gwyn

Unix utilities and applications should not try to anticipate all possible needs, and they should never try to

keep you from doing something that might be dangerous. Your program may be so useful that it outlives you. Even if it doesn't, it should most certainly outlive your current project whenever possible: Don't try to guess at how people -- even you yourself -- *might* want to use it in future.

This feeds back into the simplicity argument. Trying to prevent undesirable consequences can be a disaster. (For instance, imagine how much less convenient it would be if programs that used $EDITOR tried to enforce the assumption that it will be an interactive program.)

Lesson 9: 10% of the code does 90% of the work
When writing a component, don't try to anticipate every need; figure out what your core functionality is, and stay focused. This isn't to say you should avoid generality when it's easy to add, but if it's really hard to handle a special case, don't handle it: Other programs that handle that case and nothing else will show up, and they won't clutter your design. The resulting component will be smaller, faster, easier to use, and it will be done on time.

Ideological purity has never been part of the Unix model. There are exceptions to every rule. Don't try to anticipate them all. Don't try to include them all. Pick a problem domain, point to everything outside it, and say "here there be dragons." Recognize that there *are* special cases, but don't try to accommodate them all, or fit them all into your framework. A flexible and robust framework can accommodate special cases; a rickety framework designed to handle everything without special cases will be destroyed when a new special case comes along.

Lesson 10: Structured data is more important than structured code
Try to focus on simpler input and output formats. Put data in streams. And please, please, *make the data human readable*. XML is a great step forward in this regard: It allows you to look at the output of a failing utility and read it directly.

Unix pipes marshal structured data between applications in a transparent way, which allows a great deal of flexibility for other applications and tools to operate on this data in ways that the original designers would not even have anticipated. In practical terms, this ability to repurpose the data is much more likely to save development effort than even the best structured code methodologies. XML is one tremendous example of a technology that has learned this lesson. One of the reasons for XML's meteoric rise is that XML processing brings about flexibility and extensibility similar to that afforded by Unix pipes. At the same time, XML processing is designed to fit into more "mainstream" environments.

XML pretty much brings us full circle past component technology, and back to the idea of synthesizing data from independent processing modes. Certainly, XML has adopted many of the lessons of Unix pipes, especially with the XSLT language.

Taking these lessons with you
Of course, not everything is Unix. (More's the pity!) Not everything is even like Unix. Can these lessons be applied to other environments? Other tasks? Other APIs?

OF COURSE they can!

Even if it's a little more expensive in your environment of choice, take the time to separate out parts of a program. Make sure you're decomposing them as far as you reasonably can. Remember why `uniq` doesn't sort its input; make sure that you keep separate tasks separate. This principle will serve you well whether you are developing or using components, or if you are involved in any other sort of software development process.

Keep an eye out for fundamental operations; build components that provide these services *and do nothing else*. Then use them, regularly.

Finally, above all else: Have fun. Unix has survived, more than anything else, because it is a delightful environment to work in. Always take pride in your work. Don't cut corners. (Narrowing your problem domain isn't cutting corners; failing to handle a domain you never narrowed is cutting corners.)

Resources

- This is the final article in a four-part series. The other parts are:
  - Unix utilities, Part 1: Yet another new component architecture

- Unix utilities, Part 2: Simple tools, complex problems
- Unix utilities, Part 3: The easiest component you'll ever write

- *prettyprint*, *prettyread*, and the associated `pretty.pl`, are available at http://www.plethora.net/~seebs/comp/. *unsort.c* is also available for download so that you may examine the source.
- *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 @server 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.

e-mail it!

What do you think of this article?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Comments?

Submit feedback