Editors: Dave Thomas and Andy Hunt ■ The Pragmatic Programmers
dave@pragmaticprogrammer.com ■ andy@pragmaticprogrammer.com

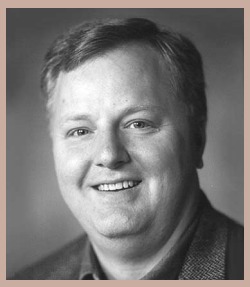
Pipelines

Dave Thomas and Andy Hunt

Pipelines can let you replace complex program logic with a chain of far simpler discrete steps.

One of the underlying philosophies of the Unix command shell is the idea of the pipeline: Take a set of small, specialized programs and let them interact by feeding the output of one into the input of the next. By stringing simple programs together in this way, rich behavior can emerge.

For example, you might want to know



just how many files you have in a directory. In the Unix shell, the `ls` command produces a list of files, and the `wc` command counts the words, lines, and characters in its input. String the two together into a pipeline (using the `-l` option to `wc` to tell it to report the number of lines it has read) and you have your answer:

```
dave%  ls | wc -l
      13
```

The `grep` command reports those lines in its input that match a pattern. Feed the command the directory listing from `ls` and you can find out the files that have a "u" or a "z" in their name:

```
dave%  ls | grep '[uz]'
automation
schedule
zero_tolerance
```

If you instead wanted to know *how many* files have a "u" or "z" in their name, you could add a `wc` command to the end of the pipeline:

```
dave% ls | grep '[uz]' | wc -l
      3
```

I got to thinking about pipelines when I was working on the answers to a programming exercise (see http://pragprog.com/pragdave/Practices/Kata/KataSix.rdoc). The challenge was to find all the sets of anagrams in a list of words. After working through all the obvious solutions in Ruby, Java, and C#, I decided to recode a solution in straight C. It turned out to be ugly: because C doesn't directly support collections more sophisticated than linear arrays and because it has no garbage collection, my code spent a lot of time handling low-level details. As a result, the code was harder to write, and the result was a lot harder to read.

Jon Bentley also has an implementation of the anagram problem in his marvelous book *Programming Pearls* (Addison-Wesley, 1999).

0740-7459/03/$17.00 © 2003 IEEE

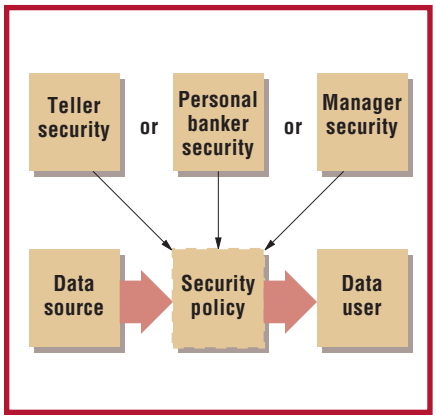


Figure 1. Pluggable security using pipelines.

His solution breaks the problem into two parts, which he calls *sign* and *squish*. In sign, he generates a signature for each word in the original list. This signature is simply all the letters in a word, sorted; all words that are anagrams will share a common signature. The program writes this signature, along with the original word, to its standard output. Given a word list containing the words "ante," "cart," and "neat," the program would write

```
aent    ante
acrt    cart
aent    neat
```

Bentley then feeds this output through a standard system sort program, resulting in

```
acrt    cart
aent    ante
aent    neat
```

This has the effect of bringing all the anagrams together. He then feeds this list through squish to read the file and collapse adjacent lines with the same signature. The result is a grouping of all the words that are anagrams.

You'd invoke this anagram finder using a pipeline, where each step performs a fairly simple piece of processing on its input.

```
cat words.txt | sign |
sort | squish
```

The key thing here is that the combined complexity of the two subprograms, sign and squish, is considerably less that that of a single program that does the same job. By dividing up the problem, we've made our lives a lot simpler. We've also allowed ourselves some potential reuse. The squish program, for example, simply groups words based on some signature. If we had a list of people's first and last names, we could also use squish to also group folks who share a first name. If we had a list of football teams and players, we could use it to produce a list of players grouped by team.

Pipelines and filters

One thing pipelines can do is filter: only a subset of data that enters the program makes it through to the other side. We saw that earlier with the `grep` example; only filenames containing the letters "u" and "z" passed through to the word count program.

We can use the same kind of filtering in real-world applications. Banks and other financial institutions, for example, have complex rules when it comes to data security and access rights. A particular bank might say that

- Tellers may access account information but not customer addresses.
- Personal bankers may access account information and the addresses of customers who are not also bank staff.
- Managers may access the account information and addresses of all customers except headquarters staff.
- And so on ….

One thing pipelines can do is filter: only a subset of data that enters the program makes it through to the other side.

We can design a system that implements these rules by encoding them into the various business objects that represent customers, accounts, and staff. This works but can lead to some convoluted logic; it can also spread that logic over many parts of the system, turning development and maintenance into an unpleasant game of bug whack-a-mole.

Alternatively, you might use a pipeline where security is implemented by injecting a filter process between the program that generates the list of accounts and the program that uses that list (see Figure 1). Inserting different filters would implement different security policies.

If we split security into its own filter, we immediately gain some advantages. First, we've encapsulated the security rule into its own self-contained module or program; when we need to change the rules, we know where to look. We've also simplified the code that works with secured entities; this code no longer needs the various hooks and conditional logic to deal with security issues. We've also potentially achieved a degree of reuse; if other parts of the system need this form of security, they can reuse our security filter program.

Pipelines and decorators

The programs in pipelines can also decorate the data that passes through them. The sign program, for example, adds a signature to every word it receives as input.

In the commercial world, we're always decorating data with additional derived information. We might take a set of values and add subtotals and totals, turning it into a report. Or we might take the same data and perform some statistics on the values for risk analysis purposes. If the data represented an order, we might add shipping charges, sales tax, or some cross-sell

message that will appear on the invoice. We can achieve all these things using conventional design and coding, but they might be better implemented in pipelines. Given a standard data source, we can construct a multistage pipeline that includes our shipping calculator, our sales tax calculator, the totaling program, and the cross-sell program before feeding the resulting data into our invoice generator.

Again, this approach gives us many advantages: it's easier to write and test the individual functions as free-standing pipeline programs, and the resulting code could well be reusable in other contexts.

Implementing pipelines

In the Unix world, a pipeline is a command-shell construct, joining together a set of freestanding executable programs. And in some circumstances, we can implement pipelines in our applications the same way, particularly if the applications are batch-oriented.

However, it would be a pain to architect the average Web application as a set of separate executables run by a command shell. Fortunately, we don't have to in order to take advantage of a pipelined architecture.

One possibility is to write our programs using a separate thread for each step in the pipeline and have the threads communicate via queues; each stage in the pipeline would write its results to a queue that the next step reads. This kind of architecture takes a little up-front glue code, but it's simple and effective.

You can get more sophisticated and use a messaging system to connect each stage in the pipeline. Here an off-the-shelf middleware layer lets the pipeline's various components communicate by sending messages. For simple applications this is probably overkill, but it does offer scalability; if you find that sales tax calculation is using up too many resources, you could transparently move it onto another machine.

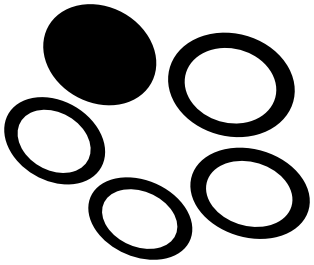
Interestingly, the current lemming-esque swarm to Web services-based architectures makes pipelining easy. We can write a central coordinating application that takes output from one Web service and feeds it as input to another. It then gets the output from that second service and feeds it to a third, and so on until the problem is solved.

The trick to making all this work is choosing a good representation for the data passed along the pipe. For the components to be reusable, the pipeline must pass the data in a generic and easy-to-use format. In the Unix world, this is a stream of bytes. In our brave new Web world, it might well be XML.

Pipelines are a wonderful opportunity for both simplification and reuse. Is there plumbing just waiting to be discovered in your current application?

Dave Thomas and **Andy Hunt** are partners in The Pragmatic Programmers. They feel that software consultants who can't program shouldn't be consulting, so they keep current by developing complex software systems for their clients. Contact them via www.pragmaticprogrammer.com.

One of the top five



for information technology degrees.

The only thing more valuable than having information is knowing how to use it. **Pace University's School of Computer Science and Information Systems** offers an outstanding faculty dedicated to teaching graduate students a unique blend of theory and real-world application. No wonder a recent *Crain's New York Business* survey listed CSIS as one of the top five sources for technology degrees in the New York City metropolitan area.

A wide range of financial aid options are available:
- Graduate assistantships
- Coop Internships
- Scholarships
- Student Loans

Learn more.

PACE UNIVERSITY
A New York Success Story

For more information,
www.pace.edu
or call
1-800-874-PACE ext. 5122

New York City
Pleasantville/Briarcliff
White Plains • Hudson Valley