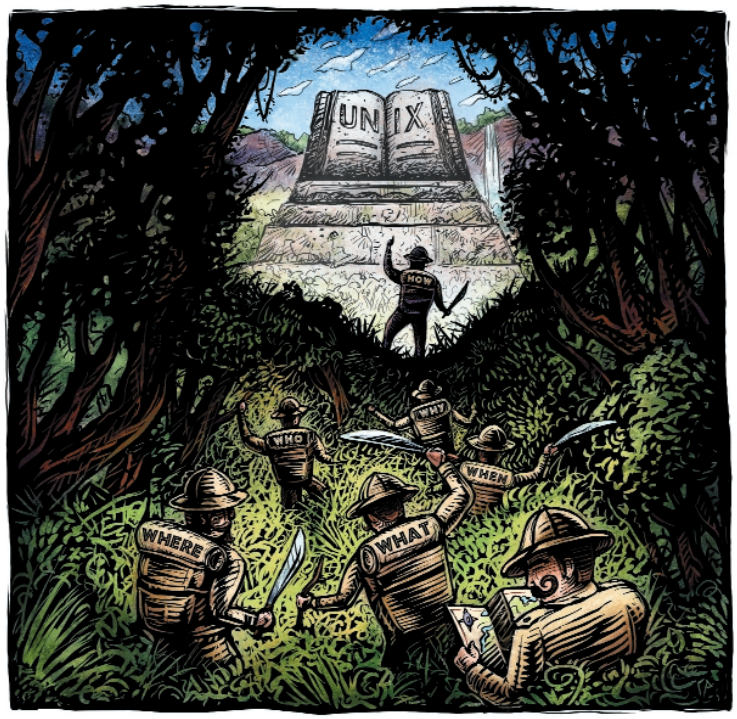
by Jeffreys Copeland and Haemer

When you get up in
the morning when
that big bell rings,
You go marching
to the table,
you see the same
old thing,
Knife and fork are
on the table,
ain't nothin' in my pan,
And you say a word
about it you get in
trouble with the man.
–Hudie Ledbetter,
"Midnight Special"



CHET PHILLIPS

Ken Thompson has an automobile which he helped design. Unlike most automobiles, it has neither speedometer, nor gas gauge, nor any of the other numerous idiot lights which plague the modern driver. Rather, if the driver makes any mistake, a giant "?" lights up in the center of the dashboard. "The experienced driver," says Thompson, "will usually know what's wrong."
–Anonymous

Documentation

"The problem with UNIX is that there's no documentation." This is so widely known that it appears in the book *UNIX Has No Documentation: The Colossal Book of Modern Urban UNIX Legends,* by Jan Harold Brunvand, Thomas J. Craughwell, and David Holt. (Bob's Bait and Publishing, 1999, ISBN 0-945-6QRZP-X.)

Earlier this year, we spent a month in Romania doing volunteer computer work. Among other things, we helped design and give a weeklong UNIX course. The target was people who would be running ISP points-of-presence, on their own, in cities around the country. We had to get them to a point where they could largely teach themselves what they needed to know. We had to teach them how to use the UNIX documentation.

Where would you start?

We took guidance from a poem in *The Second Jungle Book* by Rudyard Kipling:

I keep six honest serving-men
(They taught me all I knew);
Their names are What and Why and When,
And How and Where and Who.

Let's walk through these, though not in that order.

What

What commands are there? Look in `$PATH`. This can be a lot. I see more than 2,000 commands in my path. The word-list file used by the spell-checker was named `w2001` on old UNIX systems, because it contained 2,001 words.

In fact, if you think of these as new vocabulary words–new verbs–how does learning UNIX stack up against natural languages? English has about 13,000 verbs. But what does "to besom" mean? Perhaps we should ask how many common verbs there are? We don't know, but `w2001` suggests that there aren't too many.

Ogden's Basic English has only 850 words, total, about 200 of which are verbs. After adding a bushel of proper names (like "God"), this is enough to write the Bible (`http://www.o-bible.com/bbe.html`).

We would find it very interesting to see a study of the size of typical UNIX active and passive "vocabularies"–how many commands users can typically use and how many they recognize.

Where

Try the command `type date`. Try also `whereis date`, `whatis date`, and even `whois date`.

We encourage our Kiplingesque readers to create commands that permit `howis date`, `whyis date`, and `whenis date`.

Why

Why would you want to use a command? You could read the man page, but when we want a quick-and-dirty synopsis, we run `man -k date`, or the frequently available synonym `apropos`

`date`. We've always particularly liked the self-referential `apropos apropos`.

Unfortunately, this sometimes gets us a bit more than we want, but it's trivial to filter the output with something like

```
man date | perl -ne 'print if /^date\s/'
```

When

When was a command used? Could that be useful to ask? Yes.

Have you ever spent time debugging a program only to discover that the version you were debugging wasn't the one you were executing? `ls -ul` tells you the last time a file was read. Invoking an executable requires reading it, and this will tell you. So try

```
$ ls -ul /bin/date
$ date
$ ls -ul /bin/date
```

There are other uses for the `-u` flag, too. We use it to see if data files are actually being read by the programs that should be reading them. Try

```
$ ls -ul /etc/hosts
$ ping foo
$ ls -ul /etc/hosts
```

A more creative application is to ask whether Zipf's Law holds for UNIX commands.

Zipf's law is an observation in human linguistics: there is an inverse correlation between word-use and word-frequency.

Zipf's law is an observation in human linguistics: there is an inverse correlation between word-use and word-frequency. Frequently used words are short. This is a statement about the evolution of language. It's why we create contractions and acronyms. Some of its implications are historical and cultural. For example, "dar" (gallows) is a one-syllable word in Persian, and "whip" is a one-syllable word in English.

But is it why `ls` and `cd` are easy to type? Is it why the AT&T predecessor to `more` was called `l`? Probably.

As it says in K&R, "Since assignment is about twice as frequent as equality testing in typical C programs, it's appropriate that the operator be half as long."

Does Zipf's law hold for UNIX commands as a whole? Collecting long-term statistics on UNIX command-frequency use would require instrumenting the shell or the kernel. But we can use `ls -ut` to get a quick-and-dirty measure, because frequency of use will be correlated with recency. The most frequently used commands will, in general, be ones that were used most recently. The most rarely used commands, in contrast, will not have been used for a long time.

On the systems we measured, command-name length and recency-of-use were essentially uncorrelated. The highest correlation coefficient–only about 0.25–was for `/bin`. For `/usr/local/bin` the two were negatively correlated to the same degree!

Our mnemonic for the too-little-used `-u` flag is "useful."

How

How do you use a command? First, try it out. Invoke the command: `date`.

Second, try a bogus flag. You should get a usage message. The flag `-?` is often a good choice. Also, POSIX.2 guarantees that `:` will never be a valid flag, so `-:` should always not work. `date -?` and `date -:` should offer usage messages. (On our system, neither does, but both tell you how to get a usage message.)

Third, the `--help` flag often gives detailed information about the options.

Next-to-last, you can Read The Friendly Manual. We were amused by this bumper sticker.

> WWJD?
> JWRTFM.

Finally, on open-source systems, you can read the code.

Who

Your path determines the commands you can invoke. For example, root's `$PATH` typically adds a suite of commands from the directories `/sbin` and `/usr/sbin`.

And we all put `~/bin` in our path to let us add homegrown commands–idiosyncratic slang–to our vocabularies.

The shell makes this easy. An early study by Ted Dolotta and co-conspirators found that most programs on the multiuser UNIX systems they examined were shell scripts that users had constructed for themselves. (T. A. Dolotta, R. C. Haight, and J. R. Mashey, "The Programmer's Workbench," *Bell Systems Technical Journal*, volume 57, number 6, pp. 2177-2200, July-August, 1978.)

Has this changed? Most UNIX systems today are single-owner systems with a few thousand standard commands in their distributions. Nevertheless, the workstations we've glanced at still have hundreds of scripts in the users' home directories.

Documenting Your Commands

But how well are the homegrown commands documented? You know–the ones you write yourself and put in your `~/bin` directory?

We looked in a friend's `~/bin` directory that had 244 executable scripts. Seven appeared to have man pages. We were afraid to run them to find how many produced usage messages.

Of course, since they're personal commands, perhaps he remembers what they all do and exactly how to use them. All 244? `ls -ult` says probably not.

Inspection of the `~/bin` directory of another friend, who

has only 71 commands in his bin directory, reveals that the majority are more than three-and-a-half years old.

So, is it hard to add documentation to homegrown code? Not very. Let's show you how, first in Perl, then in the shell.

Pod::Usage Basics

Here's a simple Perl script.

```
#!/usr/bin/perl -w
# $Id: hello0.pl,v 1.1 ... jsh Exp $

print "hello, world\n";
```

Even if you're not a Perl programmer, this one should be clear. We use this example to remind our readers that November 21, Haemer's birthday, is also "World Hello Day," and to complain that, as usual, you forgot to send him a birthday card.

Next, let's handle a few arguments, so we have something to emit a usage message about.

```
#!/usr/bin/perl -w
# $Id: hello1.pl,v 1.2 ... jsh Exp $

$0 =~ s(.*/)();
$usage = "$0 [greetee]\n";

if (@ARGV == 0) {
 print "hello, world\n"
} elsif (@ARGV == 1) {
 print "hello, $ARGV[0]\n";
} else {
 die $usage;
}
```

For the Perl novice, here's how this behaves:

```
$ hello
hello, world
$ hello Gillian
hello, Gillian
$ hello Zoe and Riley
usage: hello [greetee]
```

This isn't good enough.

```
$ hello -?
hello, -?
$ man hello
No manual entry for hello
```

We could write a lot of argument-parsing code, and then a lot of troff for the man page, but Perl offers a better path: POD (plain old documentation), and the Getopt::Long argument-parsing package. Both come in the standard Perl distribution. Using these tools, our third version does this:

```
$ hello
hello, world
$ hello allie
hello, allie
$ hello jj liz
Too many args: 2
Usage:
  hello [--help] [--man] [greetee]

$ hello -?
Unknown option: ?
Try 'hello2.pl --help' for more information
Usage:
  hello [--help] [--man] [greetee]

$ hello --help
Usage:
  hello [--help] [--man] [greetee]

Options and Arguments:
 greetee
  The world to whom salutations should be
offered.

 *-help*
  Print more details about the arguments.

 *-man*
  Print a full man page.
```

It looks like this:

```
#!/usr/bin/perl -w
# $Id: hello2.pl,v 1.3 ... jsh Exp $

use Pod::Usage;
use Getopt::Long;

$0 =~ s(.*/)();

# From the Pod::Usage man page
our ($opt_help, $opt_man);
GetOptions("help", "man")
 or pod2usage("Try '$0 --help' for more\
 information");
pod2usage(-verbose => 1) if $opt_help;
pod2usage(-verbose => 2) if $opt_man;
pod2usage("Too many args: " . @ARGV)
    if @ARGV > 1;

print "hello, " . (@ARGV ? $ARGV[0] :
 "world") . "\n";

__END__

=head1 NAME
```

```
hello - hello, world

=head1 SYNOPSIS

 hello [--help] [--man] [greetee]

=head1 DESCRIPTION

The first program to write is the same for
all languages:

=over 4

=item

Print the words

  hello, world

=back

=head1 OPTIONS AND ARGUMENTS

=over 4

=item greetee

The world to whom salutations should be
offered.

=item I<-help>

Print more details about the arguments.

=item I<-man>

Print a full man page.

=back

=head1 AUTHOR

 Jeffrey Copeland
 <copeland@alumni.caltech.edu>
 Jeffrey S. Haemer <jsh@usenix.org>

=head1 CONFORMING TO

K&R

=cut
```

Notice a few things:

1. The documentation is in the same file as the program.
2. The program messages are extracted from the documentation itself.
3. The argument parsing is relatively short.

All these help keep the documentation and code in synch. And for a man page? Either `hello --man` or `perldoc hello` will give it to you instantly, or you can run `pod2man hello` and generate a man page, suitable for installation in `/usr/local/man/manl`.

What would you pay? But wait! There's more. You can run other preprocessors like `pod2html` to generate Web-based man pages, `pod2text` to generate flat text, `podselect` to pull out specific sections to the man page, and on and on. All at today's special, introductory, low price of ... nothing.

Do a `man perlpod` on your system for more details. There's seldom an excuse to have an undocumented Perl script.

Shell Scripts

What about shell scripts, that other workhorse of quick-and-dirty commands? The solution here is–hold on to your hats–the same thing.

Starting with an equivalent, trivial shell script

```
#!/bin/sh
# $Id: hello0.sh,v 1.1 ... jsh Exp $

echo hello, world
```

we first add argument-handling to give us something to parse

```
#!/bin/sh
# $Id: hello1.sh,v 1.1 ... jsh Exp $

die() {
 test -z "$*" || echo $* 1>&2
 exit 1
}

usage="usage: ${0##*/} [greetee]"

case $# in
 0) echo hello, world ;;
 1) echo hello, $1 ;;
 2) die $usage ;;
esac
```

and then add documentation

```
#!/bin/sh
# $Id: hello2.sh,v 1.4 ... jsh Exp $

usage() {
  echo Try ${0##*/} --help for more\
  information 1>&2
  pod2usage $0
  exit 1
}

for o
do
 case "$o" in
```

```
  --man) pod2text $0
    exit 0 ;;
  --help) pod2usage $0
    podselect -s 'OPTIONS AND ARGUMENTS' $0|
     pod2text
    exit 0 ;;
  -*) usage;;
  *) test $# -gt 1 && usage ;;
 esac
done

echo hello, ${1:-world}

exit 0

=head1 NAME

hello - hello, world

=head1 SYNOPSIS

 hello [--help] [--man] [greetee]

=head1 DESCRIPTION

The first program to write is the same for
all languages:

=over 4

=item

Print the words

  hello, world

=back

=head1 OPTIONS AND ARGUMENTS

=over 4

=item greetee

The world to whom salutations should be
offered.

=item I<-help>

Print more details about the arguments.

=item I<-man>

Print a full man page.

=back

=head1 AUTHOR

 Jeffrey Copeland
 <copeland@alumni.caltech.edu>
 Jeffrey S. Haemer <jsh@usenix.org>

=head1 CONFORMING TO

K&R

=cut
```

We could go through this line-by-line but, as you can see, it's basically the same approach as in Perl. The `exit 0` at the end of the code prevents the shell from trying to interpret the pod, and the pod utilities ignore everything outside the pod directives. Three command-line utilities that understand pod do the work of the Perl function calls. *Pod2text(1)* formats pod into printable text–both man page and help message. *Podselect(1)* pulls out specific sections. *Pod2usage(1)* creates a usage message out of a pod man page.

Want more information on how to deal with documentation? It's on your system. Reread this column on how to get it.

We tried using *getopt(1)* to parse the arguments, but decided, in the end, that it was clunkier than a simple `case` statement enclosed in a `for` loop.

Is this overkill for `echo hello, world`? Absolutely. But if you have something that's more than a few lines long, you can use these as models. There's seldom an excuse to have an undocumented shell script. And we haven't even talked about things like `info` or `xman` or the `troff` "man" macros or ... Want more information on how to deal with documentation? It's on your system. Reread this column on how to get it.

Until next time, happy documenting, happy FM reading, and happy trails.

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer *(*`jsh@usenix.org`*) works at Minolta-QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work` *or alternately at* `ftp://ftp.cpg.com/pub/Work`.