DAVID BLANK-EDELMAN

practical Perl tools: give me my woobie back

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the book *Perl for System Administration* (O'Reilly, 2000). He has spent the past 20 years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and is one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

EVERYONE I KNOW IN OUR FIELD who has been working with security for any reasonable length of time is walking around these days with their cynicism polished to a mirror sheen. I'm guessing part of this comes from the amount of Keystone Kopp–ish activity taking place every day in the name of "security." I know my eyes roll so much in response to the rules du jour at an airport that I'm starting to look like Cookie Monster.

So in that spirit, let's see what sort of "security theatre" we can perform for others with our Perl code. I want to posit the following question: What can I do to make you feel more secure about running my Perl code?

We're going to look at a set of surface-level changes you can make so that others feel better. They may actually help make the code more secure; they may not. But after all, this is the-a-tre! The notion here is we're trying to reduce fear in the user as she or he runs the code. To that end, here are five ways to address five different kinds of anxiety.

1. When Will It All End? (Fear of Infinite Run-Time)

For anything larger than a trivial script that completes quickly, users want to have a sense of where the code is in its process and how long it will take to complete. There's one reason why installer programs always feature some sort of progress thermometer. Humans are willing to suffer all kinds of pain, including excruciating ennui, if they are constantly reassured that it will pass, and they have some sense as to when that will happen.

There are a number of modules that can make displaying the progress of a process easy. Let's look at two of them. The first is fairly standard. Term::ProgressBar can show a standard progress thermometer:

```
use Term::ProgressBar;

my $endvalue = 500;
my $pbar = Term::ProgressBar->new($endvalue);

foreach my $value (0..$endvalue){
  # do something profound here instead of sleep.
  # (actually, for a new parent, that is pretty
  # profound...)
  sleep 1;
        $pbar->update($value);
}
```

This will produce a lovely thermometer that looks something like this:

```
39% [=========                                   ]
```

Term::ProgressBar has a number of features worth reading the documentation to discover. Especially impressive is its ability to tell you the most efficient way to call its routines. Though we didn't make use of this information in the previous example, update() actually returns the next value that's required to cause the display to change. To see how this might work, imagine a process that has 100,000 steps. Calling update() at the tenth step doesn't do anything because that value isn't large enough to move the thermometer over another notch. It's a wasted subroutine call. If we store the return value of each update() call, we can choose to only call update() again when it matters. This can be a big efficiency win for larger jobs.

A second module worth mentioning came up in the April 2006 column. In that column I demonstrated the use of Smart::Comments. This magic module will actually take specially formatted comments in your code and make them come alive. The example I gave in April was this:

```
use Smart::Comments;

for $i (0 .. 100) { ### Cogitating |===[%]    |
        think_about($i);
}

sub think_about {
        sleep 1; # deep ponder
}
```

The result is a cool progress bar that looks like this as the program runs:

```
Cogitating |[2%]                    |
Cogitating |====[37%]                |   (about 1 minute remaining)
Cogitating |=============[71%]       |   (about 30 seconds remaining)
Cogitating |==========================|
```

2. Hello? Tap . . . Tap . . . Is This Thing On? (Fear That the Script Has Locked Up)

Sometimes your code performs a task of indeterminate length or with an unknowable number of steps before completion. We can't use modules such as Term::ProgressBar in those cases because we have no idea what the end value will be; ten steps could represent 10% completion or .00010% completion. Even though we can't put up a pretty thermometer in cases like this, it is still important to relieve those running the program of their anxiety that things might not actually be progressing.

A simple 'print "Working..."' at the beginning of the process just doesn't cut it. The next best thing (and it's not particularly good) is something like this:

```
while (do_something){ print ".";}
```

This sort of thing works fine for small numbers of operations but it leads to an indistinguishable blizzard of periods marching across the screen when the number of steps is anything but a small amount. Yes, something like this could be used:

```
my $counter;
while (do_something){
        print "." if ($counter++ % 100 == 0); # print every 100 steps
}
```

but we can do better.

I'm old enough to wax nostalgic about Sun2 machine boot sequences, so I have a soft spot in my heart for Term::Twiddle. This module provides a spinner like the one you see in either the Sun or FreeBSD boot process. That's the little animated cursor that prints the following characters in sequence in the same spot on the screen so it appears to spin:

```
\ | / -
```

(For those of you who want to play along with the home-game version, feel free to cut out the previous line, paste each character on its own card, and make a flip-book.)

Term::Twiddle has many customization options available, but my personal favorite is the following (quoted from the documentation):

> probability . . . The purpose of this is to create a random rate of change for the thingy, giving the impression that whatever the user is waiting for is certainly doing a lot of work (e.g., as the rate slows, the computer is working harder, as the rate increases, the computer is working very fast. Either way your computer looks good!).

Using Term::Twiddle is easy:

```
use Term::Twiddle;
my $spinner = new Term::Twiddle;
$spinner->start; # start the spinner a'spinnin'
# do_something code
$spinner->stop;  # fin
```

The one gotcha worth noting for Term::Twiddle is that the do_something section in the this example can't include any sleep() calls, since sleep() potentially messes with the interval timers that allow the module to work.

If you don't like that restriction, you may be interested in another module in that family, Term::Activity, which provides a slightly less disingenuous view of how things are progressing. Term::Twiddle sets off this animated cursor thingy that changes without any direct connection to the actual process it is purporting to show working. In contrast, Term::Activity actually requires your code to call a tick() subroutine every time it wants another step in the process to be registered (and the display to change). The code then looks like this:

```
use Term::Activity;
my $progress = new Term::Activity;
while (things_are_happening){
        # do_something code
        $progress->tick();
}
```

The result is an ASCII wavelike thingy (install the module to see what I mean) that also counts the number of times tick() has been called and the interval between tick()s. All of this increases the warm fuzzy count of the person running the code.

3. Perl the Destroyer (Fear That the Script Could Be Doing Damage)

If I take a BB gun, aim it at my foot, and pull the trigger I can roughly approximate the process many Perl neophytes go through when they first learn how to write filesystem walking/changing code. Modules such as File::Find or the even spiffier File::Find::Rule make it super easy to write code that will nuke large chunks of your filesystem with very little effort. For this reason it is crucial that your code make it fairly hard to take destructive actions. If your code mass-removes files by default, someone who stumbles on your "cleanup.pl" script will be in a for a rude surprise when he or she decides to run it to see how it works. Granted, this is perhaps the last time that person will do something so ill-advised, but still, this isn't exactly the best pedagogical technique.

One easy way to avoid this situation is to force the user to call the script with a large and slightly unwieldy command-line switch if deletions are really desired. Something such as --deleteFilesAtWill could be used. A quick warning is called for here: If you are using a module such as Getopt::Long that handles abbreviated switches automatically and by default, be sure to pick something that doesn't abbreviate to a common switch name. This means that arguments such as --debugByDeletion, --verboseMakeGoByeBye, or --helpMeNukeMyFilesystem are probably right out.

4. Check What Condition Your Condition Is In (Fear That the Script Will Run for a Long Time and Then Fail)

As tempted as I am, I'm not going to harp more on the test-first methodology we discussed back in the April column. Instead let me harp on a variation of that idea. It can inspire confidence if your program can run a brief self-test before it runs. This test can check necessary conditions the program needs before running. For example:

- Is the database it needs hot and ready to go?
- Are the file permissions on key configuration files still ok?
- Do the working directories used by the program exist, and are they writeable?
- Is DNS reverse-lookup working fine?
- Are the TLS/SSL certificates that are going to be used still valid?

Code like this:

```
use Test::Simple tests => 5;
ok(configs_owned_by_user(), 'config files are fine');
ok(-w $tempdir, '$tempdir ready for writing');
ok(test_database(), 'database ready to go');
ok(reverse_lookup('192.168.0.1') eq 'router.example.com', 'DNS ok');
ok(check_certs($certdir/$certname),'TLS cert is valid');
```

produces comforting output like this when everything is going smoothly:

```
1..5
ok 1 - config files are fine
ok 2 - /var/tmp ready for writing
ok 3 - database ready to go
ok 4 - DNS ok
ok 5 - TLS cert is valid
```

so you know Thunderbirds are Go!

Here's a tip: If the program itself is due to have a long runtime there's a little more leeway for how long this self-test should take, but beware of letting it drag on too long. At a certain point it becomes like a folk singer who spends more time tuning the guitar than playing it. Eventually the audience revolts and starts throwing berets and bongos.

5. Breakfast. Lunch. I Said Lunch, Not Launch! (Fear of Using the Script Incorrectly)

For our last anxiety-reducing tip of this issue we're going to look at an easy way to embed the documentation for a script in that script. This method will allow the documentation to travel with the script (versus a separate manual page) without getting in the way of the code itself.

Pod::Usage makes it easy to provide two types of documentation on demand: the standard short "USAGE" message for a summary of script purpose and options or a full-blown manual page. It can actually produce something in between, but in practice I only use it for these two.

Here's how it works: Code outside of Pod::Usage is responsible for the parsing of the program options that choose how to call Pod::Usage. In practice, this means the usual option-parsing code that probably looks something like this:

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

# I prefer to store the options I receive in a hash
my %options;
GetOptions(\%options, 'help', 'man', {more options...});
```

Now we dispatch based on the switches the script receives:

```
# handle help or man page request
pod2usage(-exitstatus => 0, -verbose=> 0) if (exists $options{help});
pod2usage(-exitstatus => 0, -verbose=> 2) if (exists $options{man});
```

In this case we're calling Pod::Usage with two parameters:

1. Exit status: When Pod::Usage exits after doing its job, what should the exit value of script be as a whole? In the preceding code, we say it should be 0 (i.e., success), since the script will have successfully done its job of printing out documentation upon request. In some cases (e.g., if the script detects that the user hasn't called an option with the right arguments) we'll want to set this exit status to indicate failure. That way the program can exit with an error message and set the status accordingly. For example:

```
# abort if we don't get some key info about our ice cream cone
die "--flavor <name> not specified, aborting... (try --man)\n"
        unless (exists $options{flavor});
```

2. Verbosity: How verbose (i.e., USAGE message only or full manual page) should Pod::Usage be? Verbose level 0 prints the former; level 2 prints the latter.

Are we done? Well, almost. All we have to do now is arrange for there to be some documentation to display. (You did write documentation in parallel with the code, right?) The documentation typically lives at the bottom of the script in POD format (see "perldoc perlpod" and "perldoc perlsyn" for more info) after an __END__ token:

```
# lovely script here above this point in the file ...

__END__

=head1 NAME

    makecone - construct an ice cream cone

=head1 SYNOPSIS

    makecone [options]

        Options:
            --flavor <name of flavor>   specify flavor for ice cream (required)
            --help                      print usage message only
            --man                       show entire man page for this script

=head1 DESCRIPTION
```

. . . and so on.

Now if a user calls your script with --help or --man he or she will receive enough documentation to gain some sense of whether your script is being used correctly.

And with those warm cockles, I'm afraid it is time to end this issue's column. Take care, and I'll see you next time.