DAVID N. BLANK-EDELMAN

# practical Perl tools: essential techniques

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ 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 one of the LISA '06 Invited Talks co-chairs.

*dnb@ccs.neu.edu*

EVERY ONCE IN A WHILE IN THIS COLUMN I like to get meta. In the past we have talked about ways to become better programmers. We've looked at tools and methodologies like test-first programming which force you into a working style that produces better programs. For this column, I'd like to share three tools that can help you become a better, or at least a more efficient, Perl programmer in particular. So, perhaps this month, we'll go half-meta.

## Being Strict

I know that some percentage of my readership is going to roll their eyes with such vigor that you can hear the noise they make in their sockets, but I must start with this tip. So go ahead, get it out of your system now because I'm going to say it:

```
use strict;
```

The eye rolling comes because the Perl community has been chanting "use strict;! use strict;! use strict;!" to itself like some scene from *Eyes Wide Shut* for many, many years now. When you turn strict on for a program, the Perl interpreter will complain about a whole host of potential issues with your program that go a bit beyond syntax errors. The complaints range from simple things such as

```
Name $blah used only once: possible typo
```

if a variable only appears once in a program (for example, it is set, but never read from—that's often a sign that there's a typo in the name) to more sophisticated warnings such as

```
Global symbol $blah requires explicit package name
```

which are trying to strong-arm you into using local variable scopes (since larger programs that use all global variables are fragile and easily broken).

The reason I mention this tip at all, given how pervasive it is in the community, is that I know it took me a while to get "use strict;" religion. I suspect there are others who have lapsed in the same manner. I think there are two main reasons why people get turned off early in their programming career by this pragma and never really come back to using it as a matter of course (i.e., circumstances don't demand otherwise):

1. It yammers so. Sometimes people new to the language get overwhelmed by the quantity of error messages, especially the more cryptic ones.

This turns them off early in their Perl programming learning curve, and they never really gain a desire to be yelled at by the interpreter ("Thank you, Sir, may I have another error message? Thank you, Sir. May I have another?"). The good news is that Perl developers have worked diligently over the years to make the production of the error messages smarter and the messages themselves more comprehensible. There is also a perldiag documentation section (perldoc perldiag) that provides at least a smidgen more information for every single error message the core Perl interpreter might emit, thus making them more helpful. If you shied away from strict mode before for this reason, I'd encourage you to try it again and see if it works better for you.

2. Some of the error messages that strict mode emits require the spankin' new programmer to understand some programming concepts that may initially be beyond their comprehension. I'm thinking specifically of the scoping-related error message I mentioned before of Global symbol $blah requires explicit package name, which comes up a great deal in first-effort programs. The perldiag reference page I mentioned before says this about it:

   Global symbol "%s" requires explicit package name
   (F) You've said "use strict" or "use strict vars", which indicates that all variables must either be lexically scoped (using "my"), declared beforehand using "our", or explicitly qualified to say which package the global variable is in (using "::").

It is a very direct and pointed explanation with a teaser about how you might fix the problem, but it only describes one or two trees of the forest the programmer is likely to be lost in at that point. If you aren't familiar with lexical and global scoping in programs or are just not clear on Perl's particular way of manifesting these concepts, getting a bunch of these error messages is not going to help much even with this explanation. There's not a lot I can suggest for this case except that the programmer find a text that explains "my", "local", and "our" in a way that makes sense to them before starting to use strict mode.

Before we move on I just want to mention a couple of ways that the "use strict;" idea has been extended:

3. The module Acme::use::strict::with::pride describes itself as performing this service: "enforce bondage and discipline on very naughty modules" and says:

   using Acme::use::strict::with::pride causes all modules to run with use strict; and use warnings;

   Whether they like it or not :-)

In general I don't advocate forcing your choices about how strict a programmer should be on others, but perhaps you have a reason to make sure all of the code you are running passes a "use strict;" test.

4. There are modules like Tie::StrictHash which allow you to subvert the usual auto-vivification nature of hashes (i.e., if you reference a hash key that didn't exist before, perhaps because of a typo, it comes into being whether you wanted that to happen or not). As the docs say:

   Tie::StrictHash is a module for implementing some of the same semantics for hash members that use strict gives to variables. The following constraints are applied to a strict hash:

   - No new keys may be added to the hash except through the add method

of the hash control object.

- No keys may be deleted except through the delete method of the hash control object.
- The hash cannot be re-initialized (cleared) except through the clear method of the hash control object.
- Attempting to retrieve the value for a key that doesn't exist is a fatal error.
- Attempting to store a value for a key that doesn't exist is a fatal error.

This sort of discipline can be helpful in all sorts of situations.

## Being Tidy

Let's leave all of that kink-themed programming discussion behind for the moment and move on to the question of why your parents were always after you to clean your room. You may have ignored the tool we're going to talk about in this section because it seemed like an aesthetic nicety, but I hope to convince you otherwise. There's a lovely module called Perl::Tidy that comes with a command-line tool called "perltidy." perltidy takes in your code and reformats it to match a set of predefined (by you) stylistic conventions. It's similar to the C program source formatting called "indent" but custom honed for Perl source code. Perl code that has been run through perltidy looks neater and (depending on your stylistic preferences) more readable. Given that Perl is a bit of a punctuation parking lot with a not quite deserved reputation (from those who have never seen APL) of looking like line noise, this can be a considerable improvement.

Let's look at perltidy in action so I can explain why you should be running all of your code through it even during the process of writing it. Here's an example from the perltidy home page (http://perltidy.sourceforge.net/) that shows the most dramatic sort of improvement:

```
%TV=(flintstones=>{series=>"flintstones",nights=>[qw(monday thursday
friday)],
members=>[{name=>"fred",role=>"lead",age=>36,},{name=>"wilma",role=
>"wife",
age=>31,},{name=>"pebbles",role=>"kid",age=>4,},],},jetsons=>{series=>"j
etsons",
nights=>[qw(wednesday saturday)],members=>[{name=>"george",role=>"le
ad",age=>41,
},{name=>"jane",role=>"wife",age=>39,},{name=>"elroy",role=>"kid",ag
e=>9,},],},
simpsons=>{series=>"simpsons",nights=>[qw(monday)],members=>[{name
=>"homer",
role=>"lead",age=>34,},{name=>"marge",role=>"wife",age=>37,},{name=>
"bart",
role=>"kid",age=>11,},],},);
```

run through perltidy becomes this:

```
%TV = (
  flintstones => {
      series  => "flintstones",
      nights  => [qw(monday thursday friday)],
      members => [
          {
              name => "fred",
              role => "lead",
              age  => 36,
```

```
            },
            {
                name => "wilma",
                role => "wife",
                age  => 31,
            },
            {
                name => "pebbles",
                role => "kid",
                age  => 4,
            },
        ],
    },
    jetsons => {
        series  => "jetsons",
        nights  => [qw(wednesday saturday)],
        members => [
            {
                name => "george",
                role => "lead",
                age  => 41,
            },
            {
                name => "jane",
                role => "wife",
                age  => 39,
            },
            {
                name => "elroy",
                role => "kid",
                age  => 9,
            },
        ],
    },
    ...
);
```

Hopefully you don't have to look at code (from other people, right?) that looks like the "before" in this example. But if you do get code from a colleague that isn't that easy to read, perltidy can help.

But this isn't the kind of result that makes perltidy essential. Here is a more interesting example:

```
sub hooberbloober {

  if (test_something()){
     if ($fred == 3){
        check_with_Shiva();
        # ... lots of code
  }}
  # ... lots of code
  do_the_dance_of destruction();
  # ... lots of code
  spin_the_wheel();
}
```

Why is this interesting? It helps demonstrate two reasons why you should hook perltidy into your editor (all of the major ones can do it) so you can run perltidy over code as you write it.

First, there's a class of errors that we have all run into at one time or another having to do with improperly placed closing brackets. It is especially easy to do in cases where the chunks of your code spans multiple screens. We have all had to debug code whose program flow didn't quite work as we anticipated because a section of code was put in or left out of a conditional block by mistake. In that last example, maybe we only wanted to do_the_dance_of_destruction() based on one of the conditional tests. If there was lots more ancillary code in our example, it might not be easy to see that we've closed an if() block prematurely. But if we run it through perltidy, the error jumps right out thanks to the reformatted indentation:

```
sub hooberbloober {

    if ( test_something() ) {
        if ( $fred == 3 ) {
            check_with_Shiva();
            # ... lots of code
        }
    }
    do_the_dance_of destruction();

    # ... lots of code
    spin_the_wheel();
}
```

Second, there's considerable value to always looking at and working with clean-looking code. It has a subtle but powerful effect on how you work. Here's a quote from an invited talk I gave at LISA '07 on what sysadmins could learn from professional cooks and others in the cooking world:

> I worked with a chef who used to step behind the line to a dirty cook's station in the middle of the rush to explain why the offending cook was falling behind. He'd press his palm down on the cutting board, which was littered with peppercorns, spattered sauce, bits of parsley, bread crumbs and the usual flotsam and jetsam that accumulates quickly on a station if not constantly wiped away with a moist side towel. "You see this?" he'd inquire, raising his palm so that the cook could see the bits of dirt and scraps sticking to the chef's palm, "That's what the inside of your head looks like now. Work clean!"
>
> —Anthony Bourdain in *Kitchen Confidential*

perltidy does an excellent job of helping you find small errors not caught by the interpreter's syntax checks and work clean.

## Being Critical

OK, last tool. If you liked how "use strict;" provided feedback about problems with your code, then you are going to love this. Perl::Critic, and its accompanying command-line program perlcritic, goes even further in this direction. The documentation describes it as:

> an extensible framework for creating and applying coding standards to Perl source code. Essentially, it is a static source code analysis engine. Perl::Critic is distributed with a number of Perl::Critic::Policy modules that attempt to enforce various coding guidelines. Most Policy modules are based on Damian Conway's book Perl Best Practices. However, Perl::Critic is not limited to PBP

and will even support Policies that contradict Conway. You can enable, disable, and customize those Polices through the Perl::Critic interface. You can also create new Policy modules that suit your own tastes.

While I wouldn't necessarily run perlcritic over my code as often as I would perltidy, it is definitely helpful to periodically feed your code to perlcritic as you go along. To give you an idea of how it works, here's some output when run over some sample code found in this very column from 2006:

```
$ perlcritic geocode.pl:
Code before strictures are enabled at line 5, column 5.  See page 429 of PBP.
(Severity: 5)
```

The error here is I've not included (for space reasons) "use strict;" in my code. If I pick some other code I wrote back in 2005, it tells me about more interesting errors:

```
  chart.pl: Bareword file handle opened at line 20, column 1.  See pages
202,204 of PBP.  (Severity: 5)
  chart.pl: Two-argument "open" used at line 20, column 1.  See page 207 of
PBP.  (Severity: 5)
```

It's complaining about this line in the code:

```
open (T,">/tmp/t.png") or die "Can't open t.png:$!\n";
```

which is using conventions that have since fallen out of favor. A better way to write that would be:

```
open my $T, '>', '/tmp/t.png' or die "Can't open t.png:$!\n";
```

which passes perlcritic (with the default rules) with flying colors.

But the default settings for perlcritic only show the most flagrant violations. If I crank that up to 11 (or, rather, to a severity level of "brutal"), I get these errors from that one line:

```
Code is not tidy at line 1, column 1.  See page 33 of PBP.  (Severity: 1)
RCS keywords $Id$ not found at line 1, column 1.  See page 441 of PBP.
(Severity: 2)
RCS keywords $Revision$, $HeadURL$, $Date$ not found at line 1, column 1.
See page 441 of PBP.  (Severity: 2)
RCS keywords $Revision$, $Source$, $Date$ not found at line 1, column 1.
See page 441 of PBP.  (Severity: 2)
No "$VERSION" variable found at line 1, column 1.  See page 404 of PBP.
(Severity: 2)
Close filehandles as soon as possible after opening them at line 1, column 4.
See page 209 of PBP.  (Severity: 4)
Module does not end with "1;" at line 1, column 4.  Must end with a recogniz-
able true value.  (Severity: 4)
Code not contained in explicit package at line 1, column 4.  Violates encapsula-
tion.  (Severity: 4)
Code before strictures are enabled at line 1, column 4.  See page 429 of PBP.
(Severity: 5)
Code before warnings are enabled at line 1, column 4.  See page 431 of PBP.
(Severity: 4)
Magic punctuation variable used in interpolated string at line 1, column 41.  See
page 79 of PBP.  (Severity: 2)
Found "\N{SPACE}" at the end of the line at line 1, column 65.  Don't use
whitespace at the end of lines.  (Severity: 1)
```

and that's just with the default module rules. There are many other Perl::Critic::* modules on CPAN that can add even more or different fussi-

ness. Clearly, much of what it is complaining about can be ignored (since I was only testing a single line), but in real life cases perlcritic often offers really helpful criticism. If you want to play with Perl::Critic without installing the module, some people in the Perl community have been kind enough to set up a Web site (http://perlcritic.com) that will audit your code for you remotely.

All three of the tools we've looked at in this column can, in the right measure, really help improve your Perl programming. Enjoy, and I'll see you next time.