DAVID BLANK-EDELMAN

practical Perl tools: programming, ho hum

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of *Perl for System Administration* (O'Reilly). 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 chair of the LISA 2005 conference and is an Invited Talks chair for the upcoming LISA '06.

dnb@pobox.com

WELCOME BACK TO THIS LITTLE PERL column. This time the official *;login:* theme is "Programming." Given the subject of the column, that theme is just a walk in the park for this humble columnist. I could simply lean back in my chair, put my boots up on the desk, tilt my hat at a rakish angle, stick a stalk of wheat between my teeth, and say, "Ah yup. Perl's a programming language all right,"[1] and I would have done my part to keep this issue on topic.

But for you, I'll work a little harder. Let's take a look at three programming practices that have cropped up in the Perl world.

Practice #1: Test-First Programming

This first practice isn't actually Perl-specific at all, but we'll look at how it easily can be implemented using Perl. It's not necessarily new either, but the notion has caught on with some of the most respected Perl luminaries and so is receiving more lip service these days than ever before. Plus, this was not something I was taught back when I was a CS major in college (back when you had to learn to whittle your own Turing machine), so it may be new to you.

Simply put, when creating anything greater than a trivial program you need to first write a set of test cases the code is expected to pass. These are written before you write a lick of the actual code. This is the reverse of the standard practice of writing the code and later figuring out how to test it.

This ordering may seem strange, because at first the test cases should completely and unequivocally fail. Since the real code doesn't exist yet (you are just calling stub code at this point), this is to be expected, because there isn't really anything to test. As more and more of the real code is written, ideally more and more of your test cases should begin to pass.

So why write a bunch of test cases that start out failing like this? Perhaps the largest win is that it forces you to think. You are forced to form a clear idea of expected input, output, and (ideally) the possible error-handling the program will exhibit once fully written. This pre-programming pondering can often be pretty difficult, especially for those programmers who like to wander towards their goal, making stuff up as they go along. The

1. Reprising my pivotal, but deleted scene from *Brokeback Mountain*. Look for it in the programming part of the special features when the collector set comes out on DVD.

added discipline may sting a little, but you will find the results are better in the end.

There are other side benefits to this approach as well. It is not uncommon during development to fix one bug and unwittingly introduce one or more new bugs in the process. With test-first development, if your test code is good you should notice those new bugs the very next time you run the tests. This makes debugging at any point easier because it becomes possible to discount many other possible contextual problems when other sections of the code are tested to be known-good.

So now that you are beginning to get religion, let's see how this works in Perl. To Perl's credit, the idea of test cases/code has been present in the community for a very long time. The classic module installation recipe of:

```
perl Makefile.pl
make
make test
make install
```

or the increasingly common:

```
perl Build.pl
./Build          # or just Build (for win32)
./Build test     # or just Build test (for win32)
./Build install  # or just Build install (for win32)
```

both imply that there are tests written that should pass before an installation.

There's a whole good book[2] on writing test code with and for Perl, by Ian Langworth and chromatic, called *Perl Testing: A Developer's Notebook* (O'Reilly), so I won't go into any sort of depth on the subject. We'll just get a quick taste of the process and if it interests you, you can pursue more resources online or buy this book.

2. Bias alert: one of the co-authors of this book is a student of mine here at Northeastern University. Bias aside, it is a really good book.

There are two core concepts:

1. Find ways to encapsulate the question, "If I give this piece of code a specific input (or force a specific error), does it produce the specific result I expect?" If it does, test succeeds; if it doesn't, test fails.

2. Report that success or failure in a consistent manner so testing code can consume the answers and produce an aggregate report. This reporting format is called the TAP (Test Anything Protocol) and is documented in Perl's Test::Harness::TAP documentation.

See Perldoc's Test::Tutorial in the Test::Simple package as a first step toward writing tests. Here's an utterly trivial example, just so you can see the bare-bones ideas made real:

```
use Scalar::Util ('looks_like_number');
use Test::Simple tests => 4;

# adds 2 to argument and return result (or undef if arg not numeric)
sub AddTwo {
    my $arg = shift;
    if (! looks_like_number $arg) { return undef; }
    return ($arg + 2);
}

ok ( AddTwo(2) == 4,               'testing simple addition');
ok ( AddTwo(AddTwo(2)) == 6,       'testing recursive call');
ok ( AddTwo('zoinks') eq undef,    'testing non-numeric call');
ok ( AddTwo(AddTwo('zoinks')) eq 'bogus test',
                                   'testing recursive non-numeric call');
```

Running the code, we get very pretty output that describes the number of tests run and their result (including the last, broken test):

```
1..3
ok 1 - testing simple addition
ok 2 - testing recursive call
ok 3 - testing non-numeric call
not ok 4 - testing recursive non-numeric call
# Failed test 'testing recursive non-numeric call'
# in untitled 1.pl at line 16.
# Looks like you failed 1 test of 4.
```

Test::Simple makes it easy to write quick tests like this. It provides an ok() routine which essentially performs an if-then-else comparison along the lines of "if (your test here) { print "ok" } else { print "not ok" }"; that simple construct is at the heart of most of the more complex testing that can take place. Don't be fooled by how trivial the ok() construct looks. The complexity of the code being called in the ok() is in your hands. If you want to write something that takes eons to compute like:

```
ok(compute_meaning($life) == 42, 'life, the universe, and everything');
```

you can do that.

There are a whole slew of other modules that allow for more advanced tests with more sophisticated comparisons (e.g., Test::Deep will compare two entire data structures), data sources (e.g., Test::DatabaseRow can access a SQL database), control flow items (e.g., Test::Exception for testing exception-based code), and other program components (e.g., Test::Pod to test the code's documentation).

Once you've written a gaggle of individual tests you'll probably want to bring something like Test::Harness into the picture to allow you to run all of the tests and report back the aggregate results. You've probably used Test::Harness before without even knowing it. It is the module called by most modules during the "make test" or "build test" install phase.

If your test scripts output the right TAP protocol, using Test::Harness is super-simple:

```
use Test::Harness;

my @test_scripts = qw( test1.pl test2.pl test3.pl );

runtests(@test_scripts);
```

The three scripts will be run and the results reported at the end. Test::Harness also provides a prove command which can be used to run a set of tests from the command line. See *Perl Testing* for more details on all of these test-related ideas.

Practice #2: Write the Code in Another Language

Oh, the heresy, the sacrilege, the gumption! I hate to be the one to introduce a little existential truth into this issue of *;login:* (usually I'd save that for the philosophy-themed issue), but sometimes you need to program in another language besides Perl to get the job done. Perhaps the vendor of a product you are using only provides C libraries and header files or you've found a really cool Python library that doesn't have a Perl equivalent. The bad news is that sometimes these situations occur; the good news is that you don't necessarily have to write your entire program in that foreign language. You may be able to create a tiny island of strange code surrounded by a sea of Perl.

One easy way to include foreign languages within a Perl program is through the Inline family of modules. Here's a quick example of embedding Python in Perl code (oh, the impiety!) taken from the man page for Inline::Python:

```
print "9 + 16 = ", add(9, 16), "\n";
print "9 - 16 = ", subtract(9, 16), "\n";

use Inline Python => <<'END_OF_PYTHON_CODE';
def add(x,y):
        return x + y

def subtract(x,y):
        return x - y

END_OF_PYTHON_CODE
```

Inline modules exist for a whole bunch of the popular and more obscure programming languages. There's a good chance you'll be able to find what you need to embed that language into your Perl code.

Another potentially useful method of programming in another language involves a language that doesn't really exist yet (certainly not in a finished form): Perl 6. There are two ways to begin enjoying some of the nifty and mind-blowing features of Perl 6:

1. PUGS (http://www.pugscode.org)—I don't think I'd use this for any serious tasks yet, but if you want to play around with Perl 6 well ahead of the actual language being ready, you can use a project started by the worship-worthy Autrijus Tang. Tang and some other programmers have basically been working to implement the Perl 6 language as specified to date using the functional programming language Haskell. This lets people kick the tires on the language design by actually using it. See the URL above for more details.

2. Damian Conway had a similar notion about using implementation to test the design, so he led the charge to create Perl 5 modules that offer test implementations for various pieces of the Perl 6 language design. He and a group of other authors have been releasing modules into the Perl6:: namespace on CPAN for quite a while.

For example, if you'd like to use the new Perl 6 slurp command to read the contents of a file into a variable, you could

```
use Perl6:: Slurp;

$data = slurp 'file';
```

Probably the most useful of these modules is the Perl6::Form module, which allows you to use the Perl 6 replacement for Perl 4/5's sub-optimal format built-ins. See the Perl6:: modules on CPAN for the sorts of Perl 6 features available for use in your Perl 5 programs today.

Practice #3: Add a Little Magic to Your Programs

For our final topic we're going to look at a couple of ways to get work done via "magic." Since we just mentioned Damian Conway in the last section, let's show another one of his creations: Smart::Comments. With this module the normally passive comments in a program's listing can spring to life and do interesting things. For instance, if you wrote code that looked like this:

```
use Smart::Comments;

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

}

sub think_about {
        sleep 1; # deep ponder

}
```

the program would print a cool animated progress bar that would look like this at various stages in the program run:

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

We didn't have to write all of the progress bar code (or even the part that attempts to provide an estimate for how long the program will continue to run), all we had to do was add the comment ### Cogitating |===[%] | next to the for() loop. This module can do other spiffy things that help with debugging your code; be sure to consult its documentation for details.

The last piece of magic I want to bring to your attention is the IO::All module by Brian Ingerson. This module is so magical that it is hard to describe. Here's what the docs have to say:

> IO::All combines all of the best Perl IO modules into a single Spiffy object-oriented interface to greatly simplify your everyday Perl IO idioms. It exports a single function called io, which returns a new IO::All object. And that object can do it all!

And when it says "can do it all!" it isn't kidding. Here are some examples to give you a flavor of its capabilities:

```
io('filename') > $data;             # slurps contents of filename into $data
$data = io('filename')->slurp;      # does the same thing

$data >> io('filename');            # appends contents of $data to filename
io('filename')->append($data);      # does the same thing

io('file1') > io('file2');          # copies file1 to file2

$line = io('filename')->getline;    # read a line from filename
io('filename')->println($line);     # write a line to filename

$io = io 'filename';
$line = $io->[@$io /2 ];            # read a line from the middle of filename

@dir = io('dirname/')->all;         # list items found in dirname
@dir = io('dirname/')->all(0);      # recurse all the way down into dirname
```

From these examples you can see that IO::All makes it easy to read and write to files and operate on directories with a minimum of code. It has both a OO-like interface (e.g. ->slurp) and a set of overloaded operators (e.g., >) for these tasks. Many of these methods can be chained together for even quicker results.

But that's only a small part of the magic. Let's see more of the IO::All pixie dust:

```
io('filename')->lock;                    # lock filename
io('filename')->unlock;                  # unlock filename (could also ->close())

io('filename')->{lulu} = 42;             # write to DBM database called filename
print io('filename')->{tubby};           # read from that database

$data < io->http('usenix.org');          # read a web page into $data
io('filename') > io->('ftp://hostname')  # write filename to ftp server

$socket = io(':80')->fork->accept;       # listen on a socket
$socket->print("hi there\n");            # print to the socket
$socket->close;                          # close the connection
```

Easy file locking, database access, and a dash of network operations. Pretty spiffy indeed.

And with that, I'm afraid we have to bring this issue's column to a close. Take care, and I'll see you next time.