DAVID BLANK-EDELMAN

practical Perl tools: impractical Perl tools

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 one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

I'M WRITING THIS COLUMN RIGHT around the unofficial U.S. holiday of April Fool's Day. I realize you won't be reading it until several months after this day has passed, but pretend the U.S. Congress was so enamored with how well the daylight-saving-time rules changes went that they began moving around other dates on the calendar willy-nilly.

Given that you are now reading this column on the new date for April Fool's Day, I now have the license to look at some of the least practical of the Perl modules available. However, despite my best attempts to the contrary, I fear you'll probably learn something practical from this exploration (but shh, don't tell anyone!).

The mother lode of impracticality in the Perl world is the Acme:: namespace. In the gaggle of modules whose name begins with Acme:: you can find modules that extend the language in weird and wacky ways or play around with various programming ideas. These modules range from the exceptionally clever to the downright stupid. But sometimes even stupid code examples can teach us something.

Cleaner, Brighter Code

The progenitor of the Acme:: namespace is the module Acme::Bleach (originally just called Bleach.pm) by Damian Conway. Written in slightly cryptic code (there's a good explanation of it at http://www.perlmonks.org/?node_id=270023), the Acme::Bleach module takes a script that consists of perfectly ordinary looking Perl code like:

```
use Acme::Bleach;

open my $EXAMPLE, '<', 'example.txt' or die
"Can't open example: $!\n";
   while (<$EXAMPLE>){
      print;
   }
   close $EXAMPLE;
```

and transforms the file into:

```
    use Acme::Bleach;

(plus another 107 lines of carefully selected
whitespace characters)
```

That file, believe it or not, actually runs and does the same thing as the original program. This idea of a program that could rewrite itself into an obfuscated form that is self-deobfuscating on the fly quickly caught on in the Perl community as a fun thing to try. Now there are a number of modules like this available. Some of them are actually bordering on being useful.

Let's look at two examples. (1) Acme::PerlTidy runs the Perl::Tidy cleanup process on itself every time it is run, thus assuring your code is always as readable as possible. (2) Acme::PerlML takes your perl code and translates it to an XML representation. For example, the sample code above (substituting Acme::PerlML for Acme::Bleach) becomes:

```
use Acme::PerlML;
```

```
<document><token_whitespace></token_whitespace>
<statement><token_word>open</token_word><token_whitespace></token_whitespace>
<token_word>my</token_word><token_whitespace></token_whitespace>
<token_symbol>$EXAMPLE</token_symbol>
<token_operator>,</token_operator><token_whitespace></token_whitespace>
<token_quote_single>&apos;&lt;&apos;</token_quote_single>
<token_operator>,</token_operator><token_whitespace></token_whitespace>
<token_quote_single>&apos;example.txt&apos;</token_quote_single>
<token_whitespace></token_whitespace>
<token_operator>or</token_operator><token_whitespace></token_whitespace>
<token_word>die</token_word><token_whitespace></token_whitespace>
<token_quote_double>&quot;Can&apos;t open example:$!\n&quot;</token_quote_double>
<token_structure>;</token_structure>
</statement><token_whitespace></token_whitespace>
<statement_compound><token_word>while</token_word><token_whitespace></token_whitespace>
<structure_condition><token_structure>(</token_structure>
<statement_expression><token_quotelike_readline>&lt;$EXAMPLE&gt;</token_quotelike_readline>
</statement_expression><token_structure>)</ token_structure>
</structure_condition>
<structure_block><token_structure>{</token_structure>< token_whitespace>
. . .
```

(plus more lines of a rather ugly XML representation of the Perl code).

You could imagine someone finding this transformation to be actually helpful, perhaps in conjunction with an XML database or XML acceleration appliance.

Before we leave the Acme::Bleach family to look at more useful modules, I think it is worth pointing out that the general concept of messing with the source code of a script right before it is executed is an interesting one that opens up many possibilities.

Perl has had a feature to do this for some time (although it isn't used by the Acme::Bleach module) called "source filtering." With source filtering you can write code that processes the source code being read into the Perl interpreter *before it is executed*. If you can fiddle with source like this before handing it to Perl to interpret it gives you the power to write your source in any form you'd like (just as long as it eventually can be transformed back to basic Perl syntax). Just to show the power of the concept, Damian Conway has written a module that allows you to program in Perl using Latin. See the perlfilter man page if you are interested in this concept.

Not Just Fun and Games

The set of Acme:: modules I tend to respect the most are those that play with various language concepts or attempt to solve real problems while still maintaining a sense of humor. An example of the latter is Acme::RemoteINC, which describes itself as the "Slowest Possible Module Loading." That's being overly modest. What it really does is fetch a module using FTP from some repository in a transparent way if it isn't available when the program runs. Even if this turns out to be a slow operation, you have to admit the concept is cool and ripe for further exploration.

Similarly strange but very clever in its own way is Acme::Scripticide, which allows you to write scripts that delete themselves. Why is this useful? The author explains this in the documentation:

> Believe it or not this is handy if you have a one time job to execute:
>
> ```
> # $script uses Acme::Scripticide
> system $script if -e $script;
> ```
>
> or say to create static files from a database:
>
> ```
> # in flowers.pl (copy this to whatever names you want and execute:)
> use Acme::Scripticide qw(good_bye_cruel_world);
> good_bye_cruel_world('.html', get_html($0));
> ```
>
> now flowers.pl does not exist and flowers.html is there.
>
> You could have a directory full of those types of scripts and glob() them in and execute each one; once that is done, you have a directory of corresponding static html files.

I get excited about stuff like this because it opens up a whole new avenue of thinking for solving certain kinds of problems.

Mucking about with language constructs using Acme:: modules has a similar expanding action on one's brain. For example, the Acme::BottomsUp module lets you order your really long compound Perl statements closer to the way you might explain the statement to someone. Once again I'm going to quote from a module's documentation, because it has a superb example. It shows that a code fragment like this:

```
my @arr = (1..10);

   print                  # lastly, display result
       join ":",          # and glue together
       map { $_**3 }      # then cube each one
       grep { $_ % 2 }    # then get the odd ones
       @arr               # first, start with numbers
;
```

"reads better" if you use the Acme::BottomsUp module like so:

```
my @arr = (1..10);

use Acme::BottomsUp;
@arr                      # first, start w/ numbers
    grep { $_ % 2 }       # then get the odd ones
    map { $_**3 }         # then cube each one
    join ":",             # and glue together
    print                 # lastly, display result
;
no Acme::BottomsUp;
```

If you've worked with other programming languages that use a different statement order, you won't find this idea to be particularly revolutionary. But for someone who has only written code like the "before" sample here, this module might provide a welcome ponderable about language design.

Let me give you one last titillating example in the language vein before we move on: Acme::use::strict::with::pride. This module is designed (and I quote) to "enforce bondage and discipline on very naughty modules." As soon as you load this module, all subsequent modules loaded by the script via "use" or "require" will find themselves running with use strict and use warnings turned on (and I quote again) "whether they like it or not."

A::u::s::w::p provides us with two things:

1. The chance to enforce the same level of discipline on the modules you import from someone else that you might impose on yourself when writing code.
2. Another good ponderable about what other sorts of context or manipulation could be applied to these external modules as they are loaded.

Getting More Entertaining

For a column with "impractical" in the name we're drifting perilously close to abject seriousness. Let's get a little lighter by looking at two modules that solve a "problem" that we may not have considered easily solvable.

The first module is actually not necessary as of this writing but it is nice to know it exists. Acme::DNS::Correct was written to correct for a condition that afflicted the Internet for a brief while back in 2003. This was the great VeriSign SiteFinder debacle. At some point VeriSign decided it would help the Internet by making sure that all domain names in the .COM and .NET top-level domains would resolve to something when queried, even the ones that *didn't exist*. This broke all sorts of things and so modules such as Acme::DNS::Correct were developed.

Acme::DNS::Correct lets you do all of the standard Net::DNS resolution stuff but is smart enough to remove all references to the $ROOT_OF_EVIL, VeriSign's SiteFinder server, when it encounters them. Luckily that "service" was quickly run out of town. Earthlink pulled a similar stunt earlier this year (http://kb.earthlink.net/case.asp?article=187117), so it is good to know that modules like this are still available should this idea rear its ugly head in any substantial way again.

A second Acme:: problem-solver module is the Acme::MetaSyntactic breed of modules. Acme::MetaSyntactic is dedicated to the problem of finding good example variable names when "$foo" and "@bar" ceases to cut it. I tend to use "$fred", "$barney", and "@betty" when teaching but thanks to this module it is clear that I've been limiting myself. The module has many, many themes (there are 104 as of this writing) from many different sources. It ships with a helper script called meta that allows you to say:

```
$ meta teletubbies 3 # give 3 example variable names using this theme
Noo_Noo
Laa_Laa
Tinky_Winky

$ meta sins 3
laziness
gluttony
pride
```

```
$ meta thunderbirds 3
Brains
Gordon_Tracy
Parker
```

You'll never be without interesting example variable names again.

OK, I Lied; It Is All Fun and Games

As a way of ending this month's column with a smile, let me stick to my guns and show you three modules that are legitimately of dubious practical value but are amusing nonetheless.

The first is Acme::Test::Weather. Acme::Test::Weather is meant to be like the other testing-oriented modules (Test::More, Test::Simple, etc.) I first wrote about almost precisely a year ago in this column. The difference is that instead of providing testing primitives that perform comparisons such as "Is the result of subroutine() eq to this string?" it provides tests such as:

```
is_cloudy()
isnt_snowing()
eq_fahrenheit()
lt_humidity()
```

The module allows you to write tests based on the current weather for the machine running this test. Seriously. To make this happen it first attempts to look up the IP address's location using CAIDA::NetGeo::Client. With this location it calls Weather::Underground to find the current weather for that location. Why does it do this? The doc says, "Because, you know, it may be important to your Perl module that it's raining outside."

(As a related aside, I have to confess that in one of my classes I show people Perl code that behaves a certain way based on the current phase of the moon. Maybe I need to package this into its own Acme:: module?)

The second of our closing modules will mostly amuse the computer science readers. Let me let it speak for itself:

```
Acme::HaltingProblem - A program to decide whether a given program
halts
```

I would show you some sample code that uses this module, but the documentation lists the following bug:

```
This code does not correctly deal with the case where the machine does
not halt.
```

And finally, there is Acme::Morse::Audible. Like Acme::Bleach, the first time you run it it rewrites the script containing your original source code. In this case the source code becomes a real MIDI file containing the original source code: the original source code translated into Morse code, that is. Once you strip out the leading "use Acme::Morse::Audible;" line anything that can play back MIDI files will let you listen to your Perl code as it would be rendered in dots and dashes. And yes, if you leave the first line intact the obfuscated script still runs fine.

At best the idea of translating your programs into audible representations may inspire some new great ideas (or some nostalgia for the days when listening to relays could help debug programs). At worst this module's very existence tickles me pink. On April Fool's Day that's good enough for me. Take care, and I'll see you next time.