THE PERL REVIEW'S COMMENTARY

by Nadim Khemir
Guest
Commentator

Corporate Perl

Why isn't the Perl community "corporate-enterprise" enough? I don't have all of the answers, but I know some of the problems we have to solve.

Nothing could make me more happy than the subject of the next YAPC::Europe, "Corporate Perl", and my expectation level is high. What will the Perl gurus and my peers come up with? People get surprised when I talk about corporate-enterprise Perl and some even tell me it can't be more corporate-enterprise than it is right now, validating their point by telling me companies are making lots of money because of Perl. I have no doubt that all the duct taping we are doing is generating money but what makes, or could make Perl worthy of the "corporate" adjective?

We certainly feel comfortable in the world of Agile development methods, SOA, virtual machines, web services, and enterprise integration. I guess that this is due to the fact that we are flexible and curious. Wild and daring would be another way to put it.

But I believe we are making the biggest possible mistake. Pride is our sin. We *know* we are corporate-enterprise worthy, We *know* that companies that do not embrace our enthusiasm and ways are not worth working for. Our pride is well deserved and it certainly is the most important factor in our success so far, if not the only one, in the duct-tape world where credit is given only when things break. Still, I believe we're making a mistake. We should be judged by the industry not doing the opposite. To be judged, we need to be visible and I'd certainly like the The Perl Foundation to very seriously start doing something about that.

Becoming more corporation-enterprise worthy is going to be a balancing act. The Perl community has a soul and we must keep it while working with the soulless software industry. Or maybe we shouldn't, and accept we just want to be programming pariahs that get things done. In any case, the current limbo must stop.

Since we all more or less know why we are good at what we are doing and why Perl is the best fitted tool to use, I'm not going to list the positive sides of Perl as a development environment (yes, environment) but I'll try to list some of the things that have been hindrances so far.

Scribblers

Perl being used as a scribbling tool and for those who use it in that way, frankly, I intensely dislike them. Not because they are new to the trade or Perl but because they do nothing to get better. Worse than that, the more they learn about Perl the worse code they produce. Out of a hundred Perl beginners, most remain beginners because they are happy with what they have. Perl duct taped their problems to the next problem!

CPAN

CPAN is one of Perl's killer features, but it's not perfect either. Its success is due in part to its low barrier to entry, but that also brings other problems. Mitigating these problems should be a top priority:

- varying code quality of modules
- difficulty finding what I need among many alternatives
- many modules with bad documentation
- poor version management

Handling the industry

No single point of entry for companies to find the needed information to run a modern development environment with Perl. There's a lot to use but it's all bits and pieces. We simply don't care. I, as a developer, like the freedom it gives but I wonder if it's a viable strategy for companies.

Tools

I like the debugger that comes with Perl, but it's as usable as a 1950 telephone switch. The state of the standard Perl tools and integration of those tools is not very good. New projects †want to solve the problem so there is hope but those projects should be helped too.

Your opinion matters! I'm one in a million, hmm, I would like in how many I am in but that information is not available either, and I'd like to see better †integration into existing applications (this is what made python popular), serious work to integrate eclipse (this is what makes things look serious to companies), serious work with Google, Microsoft, IBM and other companies that have the means of putting Perl in the industrial tracks. Work with academia so they produce something better than lobotomized developers that will use Perl to scribble something fast and dirty to save their skin and burn mine in the long run.

What corporations expect of Perl is of primordial interest and will shape the future of Perl as much as our own expectations. The next YAPC::Europe is indeed going to be interesting and I hope that its subject will spark an interest in the other YAPC's too.

by brian d foy

Publisher & Editor

The lore of programming

We're just barely out of the prehistoric age of computing, and our best practices reminds me of Steve Martin in the "Theodoric of Yorc. Medieval Barber" sketch from *Saturday Night Live*:

> *You know, medicine is not an exact science, but we are learning all the time. Why, just fifty years ago, they thought a disease like your daughter's was caused by demonic possession or witchcraft. But nowadays we know that Isabelle is suffering from an imbalance of bodily humors, perhaps caused by a toad or a small dwarf living in her stomach.*

The way that we learn, and pass on knowledge, in programming circles seems frightfully akin to that. Outside of the small circle of the Perl community, bad practices, some of which boggle the mind, seem to live on forever no matter how many FAQs, articles, and forum postings not only show why they are bad but give much better ways of doing things. Matt's Script Archive is still going strong and still infecting new Perl users over a decade later, and several years after London.pm's NMS (Not Matt's Scripts) provided drop-in replacements for most of them.

Even inside the small circle of the Perl community that actively participates in CPAN, Perlmonks, and so on, the practices constantly change too. Someone has probably already told you to start using Moose, a new object system for Perl 5, for everything. Moose is only one example, however, of the whiplash speeds that one set of practices falls out of favor and a new one takes its place. This isn't a judgement of Moose, and I could come up with other examples where the Perl community had a new hotness that we don't use anymore.

In both cases, the absurdly long-lived bad practice and the great new thing, people learn by lore—the traditional body of knowledge that people pass on to one another. It's cargo-culting for ideas and concepts. The ideas don't come with a context or any wisdom on how to apply them. They become the traditions that nobody really knows quite how they started or why we do it that way.

This isn't a problem of the velocity of technology, whether too slow or too fast. At any velocity, most people don't have any way to evaluate the merits of what they are doing, so they either follow the latest fad blindly, or follow the first advice they run across on the internet.

I suspect, though, that most people don't have time to properly evaluate major courses of action. The higher-ups need work to start today. If something worked for another company on a completely different project, it should be suitable on their project too. Everyone has a friend or a friend of a friend who knows the right way to do it. Nobody thinks about all of the other people they know that give a different answer.

Tim O'Reilly, in his O'Reilly Radar post "Hard Work and Practice in Programming", goes through the responses of O'Reilly editors to an interview of Bjarne Stroustrup (the creator of C++) talking about the education of programmers. Bjarne says so many interesting things on this subject that it's hard for me to pick out any particular one, but this one will do:

> *The "magic constant" example is indicative. Few students see code as anything but a disposable entity needed to complete the next project and get good grades. Much of the emphasis in teaching encourages that view.*

He goes on to say much more about what he believes is wrong with the educational system for computer science majors, and much of it revolves around the idea that the students should somehow magically pick up good programming skills on their own. Instead, they pick up just what they need to do to satisfy the classroom requirement. If school is still like it was when I went to college, that means students passing ideas amongst them. Those ideas usually shouldn't see the light of day. You have to finish the assignment, turn it in, and move on.

What does that mean for you? If you're just out of college looking for job, you probably don't have the education the employers need. If you're looking to hire employees ready to be good programmers on their first day, you don't have many options. I've heard this from many Perl employers: they always get a lot of applications, but very few suitable candidates. The educational system failed both sides.

I heard from a doctor friend of mine that his medical school professor started off the year with this quote:

> *Half of what I learned in medical school was wrong. Half of what I'll teach you is wrong. Unfortunately, I don't know which half.*

Consider that the next time you get advice or instruction, no matter the source. Is that advice going to be the half that is wrong, or the half that is right? Is it something that you should risk passing on, or quietly evaluate to see how it works out? I hope Moose is in the half that works out to be right because it's looking really nice.

Watch Theodoric of Yorc on NBC's website: *http://www.nbc.com/Saturday_Night_Live/video/clips/theodoric-of-york/2888/*

NMS is available on SourceForge: *http://nms-cgi.sourceforge.net/*

Moose is a postmodern object system for Perl 5: *http://www.iinteractive.com/moose/*

Interview with Bjarne Stroustrup: *http://itmanagement.earthweb.com/features/article.php/3789981*

O'Reilly Media Editors' response to the Stroustrup interview: *http://radar.oreilly.com/2008/12/hard-work-and-practice-in-programming.html* header, and off it goes.

Movie information from IMDB with IMDB::Film

Need to do something more specific, like for instance getting information on a movie from IMDB? All you need to do is use *http://search.cpan.org/* and you'll find a couple of modules to help you:

```
use IMDB::Film;

my $imdbObj = new IMDB::Film(
  crit => 'Troy');

print "Title: " . $imdbObj->title()."\n";
print "Year: " . $imdbObj->year()."\n";
print "Plot Symmary: " .
  $imdbObj->plot()."\n";
```

Create a tag cloud with HTML::Cloud

Want to create a tag cloud to use in your website? Of course CPAN also has that!

```
use HTML::TagCloud;

my $cloud = HTML::TagCloud->new;

$cloud->add(
  'perl',
  'http://example.com/perl', 1 );

$cloud->add(
  'cpan',
  'http://example.com/programming', 3);

$cloud->add(
  'books',
  'http://example.com/books', 2);

my $html = $cloud->html_and_css(50);
```

It's that simple. Also, you'll note that CPAN also shows you some simple (and sometimes more complex) code samples that you can use to get you started. And, if you hit a wall, don't forget to ask for help on PerlMonks.

■ Creating a distribution

If you want to start a distribution properly, you should use the `module-starter` command, which is shipped with the `Module::Starter` package.

Creating a distribution with it is as simple as typing the information you need to start a module:

```
$ module-starter --module=My::Application
--author="My Name" --email=myself@example.
com
```

Afterwards, browse the files that were created to get a feeling of what should be inside a distribution.

For further information, check also "José's guide for creating Perl modules" at *http://perlmonks.org/index.pl?node_id=431702*.

■ Testing

Perl is very good for testing. While this article is not about testing, I would like (because it has feelings too) to tell you very briefly that test modules in Perl do exist.

Here's a very simple example in which we test that the function `sqr`, invoked with a value of `2`, returns a true value:

```
use Test::More qw(no_plan);

ok ( sqr(2), "sqr is working");
```

And here we're testing whether the result of `sqr(2)` is 4:

```
is ( sqr(2), 4, "sqr(2) is 4");
```

Here's one more tri[illegible]sting whether the result of `random_phone_number` matches a regular expression that only accepts a string comprised of 9 numbers (yes, I know that's not how you validate phone numbers; this is just an example):

```
like (
  random_phone_number(),
  qr/^\d{9}$/,
  "random phone number has nine digits"
);
```

■ Handy references

No one knows everything, and even if you do, it's hard being able to recall everything at any given time; hence, print these three reference cards and keep them handy:

Perl Quick Reference Card
http://johnbokma.com/perl/perl-quick-reference-card.html

Perl Regular Expressions Card
http://refcards.com/refcard/perl-regexp-trusketti

Perl Testing Reference Card
http://langworth.com/pub/notes/perltestref.html

■ About the author

José Castro (*cog@cpan.org*) was an organizer of YAPC::EU 2006 in Braga, Portugal, and is one of the organizers of YAPC::EU 2009 in Lisbon, Portugal. He's the president of the Associação Portuguesa de Programadores Perl (APPP), the Community Relations Manager for The Perl Foundation, and the author of "José's Guide for Creating Perl Modules" (*http://perlmonks.org/index.pl?node_id=431702*) and "Perl White Magic" (*http://perlmonks.org/index.pl?node_id=431511*). You can blame him for several `Acme` modules.

Programming with CPAN.pm

by brian d foy
brian.d.foy@gmail.com

Anyone can easily make their own tools to deal with the Comprehensive Perl Archive Network (CPAN). Instead of relying on the tools that make other people's life easier, I can make a tool that does exactly what I need to do. Not every tool has to install modules, either.

I'm the author of the `cpan` script that comes with `CPAN.pm` (and hence with `perl` too), and examples in this article are also features of `cpan`. However, if my tool doesn't do the job you need, I'm not going to feel bad if you create your own. And, even though I use `CPAN.pm`, it's just as easy to create tools with `CPANPLUS`.

■ A note on terms

In talking about CPAN and its ecosystem has a problem: everything is called something that is pronounced alike. For this article, CPAN in all capital letters and no special formatting refers to the archive network itself. I'll always say `CPAN.pm` with the code font to refer to the module, and when I want to talk about my script that comes with `CPAN.pm`, I'll use the lowercase `cpan` in the code font. If you're reading this aloud to your kids at bedtime, make up special voices for each (a feature I want for my `Pod::Speakit::MacSpeech` module)..

Also note, that certain filesystems have made odd choices. I'm not going to single out the case-insensitive *but case preserving* Mac OS X filesystem because I am sure there are others, but its something I have to remember when I use `perldoc`. When I tell `perldoc` to find something, it stops at the first thing it finds. For modules, it allows an extra *.pm* or *.pod* at the end of my term. However, in this case, `cpan` can match different things. Try these variations of perldoc to see what you get:

```
% perldoc cpan

% perldoc CPAN

% perldoc cPaN

% perldoc CPAN.pm
```

On my Mac, the first three bring up the documentation for my `cpan` script, although the various other BSDs I use handle the case as most people would expect. The filesystem is mostly to blame for this. I could choose to use UFS on my Mac, but then I'd have to do work. Just remember that you might have to tack on the extra `.pm` to look at the actual CPAN module.

■ The old way

The old `CPAN.pm` way involved starting a shell and issuing interactive commands. From the command line, I could include the `CPAN.pm` module with the `-M` switch:

```
% perl -MCPAN -e shell
```

Once I ran that, I had a prompt where I would have to tell `CPAN.pm` to do something.

```
cpan> install Business::ISBN Tie::Cycle
```

■ The new way

Although the interactive prompt is adequate, it's just a little bit too much work for me. I don't like all of that typing on the command line just to start it, so I created my own bash alias for it and put it in my `.bash_profile`:

```
alias cpan 'perl -MCPAN -e shell'
```

Now I just needed to type five characters (don't forget to count that newline that everyone ignores!) to start the shell:

```
% cpan
cpan[1]>
```

But then, I needed to type a lot more to install a module. The shell expects a command first then the arguments that go with that command:

```
% cpan
cpan[1]> install Set::CrossProduct
```

I got rid of my bash alias and created a Perl program instead. With arguments, it installed the name modules, and without arguments, it started the shell.

```
% cpan HTML::SimpleLinkExtor
% cpan
cpan[1]>
```

From there, I started to add other features to make things more convenient for me, Andreas included it in `CPAN.pm` so other people started to use it, and other people have contributed features they found convenient. Although my initial features

Refactoring Factorial

by Alberto Manuel Simões
ambs@di.uminho.pt

You probably know the factorial function. It is a common mathematical function defined for the integer *n* as the product of all the integers in the range *[1..n]*. Looking common algorithms books it is easy to find one or two different algorithms to compute the factorial.

In this article I present different solutions to compute the factorial function using different aspects of the Perl language, but also talk about the performance.

■ Solution 1: Recursive is beautiful

The factorial function is habitually defined recursively. The idea is easy: if *n = 1*, I return 1 as a special case, otherwise, I return the product of *n* with the factorial of *n-1*:

```
sub factorial {
   my $v = shift;
   if ($v > 1) {
      return $v * factorial($v-1);
   } else {
      return 1;
   }
}
```

Some Perl linguists might prefer a solution that is easier to read in English. Instead of the C-like `if-else`, I use separate statements to handle the special case of 1 and all of the other cases:

```
sub factorial {
   my $v = shift;
   return 1 if $v == 1;
   return $v * factorial($v-1);
   }
```

Being written in Perl I can remove the return statements, and get a more functional-like solution:

```
sub factorial {
   my $v = shift;
   if ($v > 1) {
      $v * factorial($v-1)
   } else {
      1
   }
}
```

Also, the ternary operator can make my life a little easier by removing most of the syntax from the `if-else`:

```
sub factorial {
   my $v = shift;
   return $v > 1
      ?
      $v * factorial($v-1)
      :
      1;
}
```

This can be a very slow operation though. To compute the factorial of *n*, I have to make *n-1* subroutine calls, and I have to do that every time I want to make the computation.

■ Solution 2: Iterative is faster

When discussing the benefits of iterative or functional programming languages, it is commonly said that the recursive solution is slower than the iterative one, so reducing the number of function calls should make it faster:

```
sub factorial {
   my $v = shift;
   my $res = 1;
   while ($v > 1) {
      $res *= $v;
      $v--;
   }
   return $res;
}
```

Other languages might be able to optimize code written in a recursive fashion to make it iterative, which is why recursion appears faster in other languages. The code that we see isn't always exactly what the computer is actually doing.

Again, Perl linguists have their own preferences. Instead of a `while` loop maybe they want to use a `for` loop:

```
sub factorial {
   my $v = shift;
   my $res = 1;
   $res *= $_ for (2 .. $v);
   return $res;
}
```

In the other side, there are Perl golfers and obfuscators who like to iterate using different approaches. They might use grep to control a loop, even though it isn't designed for that:

```
sub factorial {
  my $v = shift;
  my $res = 1;
  grep { $res *= $_ } (2 .. $v);
  return $res;
  }
```

Yes, I can do the same with map in a void context, even though map isn't designed for this either:

```
sub factorial {
  my $v = shift;
  my $res = 1;
  map { $res *= $_ } (2 .. $v);
  return $res;
  }
```

Other people might get rid of the variables to make it a bit shorter. My $v disappears if I access @_ directly to make the list:

```
sub factorial {
  my $res = 1;
  grep { $res *= $_ } (2 .. $_[0]);
  return $res;
  }
```

Even though these are iterative solutions, they still have a performance problem that I will fix later.

■ Solution 3: Haskell programmers learn Perl --------

There are some commands that should be added directly to the Perl core. I can probably convince the Perl pumpkings Nicholas or Rafael to add one of these, but until then I can use the List::Util module, and the reduce method:

```
use List::Util qw(reduce);

sub factorial {
  my $v = shift;
  return reduce { $a * $b } (2 .. $v);
  }
```

The reduce method has the same syntax as Perl's built-in sort. It takes a code block as its first argument, and then a list. It takes the first two values of the list and puts them in $a and $b, then executes the block of code. Whatever the result of the code is goes back onto the list as the first element. reduce does that until it ends up with one element, which is the final result.

If I do not want to use external modules, I can try to mimic reduce. What reduce does is to multiply all values from 1 to $v. I can do this by creating a string where values are separated by the multiplication operator. Then, I can eval this string to return the value I'm are expecting:

```
sub factorial {
  my $v = shift;
  return eval join "*", (2 .. $v);
  }
```

This form of string eval is only of academic interest though and I shouldn't do this in real code. You might see this sort of thing in Perl golf, though.

■ Better performance --

In Perl, the iterative solution was better than the recursive one because it didn't make a lot of useless function calls. That is only a good solution if I have to call the subroutine once in my program. If I have to call it multiple times, I have to repeat a lot of work. If I want to compute the factorial of 10,000, I do that, but when I want to compute the factorial of 10,001, I have to repeat all the work I just did along with one more multiplication.

Instead of repeating all of the work, I can save the result of all of my previous work so I can reuse it. In @cache I save what I've done before, and if I have to compute something bigger than that I start where I left off:

```
my @cache = (1, 1);

sub factorial {
  my $v = shift;
  return $cache[$v] if defined $cache[$v];

  for my $i (@cache .. $v) {
    $cache[$i] = $cache[$i-1] * $i;
  }
}
```

I don't have to do that work myself though. The Memoize module does it for me, no matter which factorial implementation I want to use. I just have to tell it which subroutine I want to memoize and it handles the rest for me:

```
use Memoize;

memoize('factorial');

sub factorial {
  my $v = shift;
  return $v > 1
    ? $v * factorial($v-1) : 1;
  }
```

Now, when I use this is real code, I don't have to repeat work. Of course, this is a silly example, but it shows that optimizing just one call doesn't mean I've optimized my program.

Really big numbers

For most people, you probably can only calculate up to the factorial of 170, or some other smaller number. After that, 32-bit `perl` gives you `inf`. Even with smaller numbers, I don't get the full number. Here's a program that prints the factorial of the number I give it on the command line:

```
#!/usr/bin/perl

print factorial($ARGV[0]), "\n";

sub factorial {
  my $v = shift;
  my $product = 1;
  foreach my $n (2 .. $v) {
    $product *= $n
  }
  $product;
}
```

Depending on my system I might be different results, but here is some sample output, showing that Perl switches number formats and eventually gives up:

```
$ perl factorial.pl 17
355687428096000
$ perl factorial.pl 18
6.402373705728e+15
$ perl factorial.pl 170
7.25741561530799e+306
$ perl factorial.pl 171
inf
```

To get around this, I can use the `bignum` pragma that comes with Perl 5.8. It lets me calculate much bigger numbers. I just have to add the pragma:

```
#!/usr/bin/perl

use bignum;

print factorial($ARGV[0]), "\n";

sub factorial {
  my $v = shift;
  my $product = 1;
  foreach my $n (2 .. $v) {
    $product *= $n
  }
  $product;
}
```

Now the output is different. I see the whole number, and I can go past 170:

```
$ perl factorial.pl 18
6402373705728000
$ perl factorial.pl 171
12410180702176678234248405241031039926166
05577501693185388951803611996075221691752
99275197812048758557646495950167038705280
98898586907107673312420322184843643104735
77889968548278290754541561964852153468318
04429323959817369689965723590394761615227
85581800611763651084288000000000000000000
00000000000000000000000
```

Putting in all together

Adding all of the parts, I end up with something I can use many time in the same program and get really big numbers:

```
#!/usr/bin/perl

use Memoize;
use bignum;

memoize('factorial');

foreach (1 .. $ARGV[0]) {
  print "$_! = ", factorial($_), "\n";
}

sub factorial {
  # whichever method I choose
}
```

The future of factorial

When Perl 6 is ready for the public, I can use its built-in reduction operator, [ ]. Inside the brackets I put the operator I want applied to each successive pair. It's just like the `reduce` from `List::Util`, but it's part of the language syntax now and I don't even need a function call:

```
# Perl 6
my $factorial = [ * ] 2 .. $n;
```

Even better, Perl 6 has lazy lists, which means it doesn't have to create all of the values in the list to do the operation. It creates the values as it needs them.

Credits

Thanks for contributions from brian d foy, Aristotle Pagaltzis, Bart Lateur, dug, and Peter Sinnott.

About the author

Alberto Simões recently finished his Ph.D. in Machine Translation. He had been working in natural language for five years which explains the large amount of Perl contributions under the `Lingua::PT` namespace. He is one of the YAPC::EU::2009 organizers and a frequent contributor to *The Perl Review*.