

Content Considered Harmful

This is Jonathan Rockway's blog, where he talks about Angerwhale, Catalyst, and Everything.

MY BOOK



NAVIGATION

[Home](#) | [Articles](#)

CATEGORIES

[Angerwhale](#)

[Catalyst](#)

[Programming](#)

TAGS

[advocacy](#)

[angerwhale](#)

[apache](#)

[bank](#)

[beer](#)

[bestpractice](#)

[blog](#)

[caffeine](#)

[catalyst](#)

[colors](#)

[cpan](#)

[crypto](#)

[cta](#)

[dbic](#)

[dongs](#)

[doqueue](#)

[emacs](#)

[firefox](#)

[firstpost](#)

[git](#)

[gpg](#)

Catalyst server aliases

No tags [\[+\]](#)

Posted on 2008-9-28 (日) at 5:05 pm by Jonathan Rockway

Posted in: [Catalyst](#)

I generally avoid blog posts that are less than a million words, but today you are in luck; I am just going to share two shell aliases that I find quite helpful for Catalyst development:

```
alias cs="perl script/*_server.pl -d"
alias carpcs="perl -MCarp::Always script/*_server.pl -d"
```

`cs` will start the Catalyst server for the app in the current directory. `carpcs` will make all warnings/dies give a full backtrace.

Maybe a little hackish, but very helpful. I use `bash`, btw.

[1 comment](#) | [Read more...](#)

Error checking

Tags: [perl](#) [\[+\]](#)

Posted on 2008-9-27 (土) at 8:46 am by Jonathan Rockway

Posted in: [Programming](#)

Remember when you were in elementary school and your mom used to tell you to look both ways before crossing the street and to always check the return values of functions that don't throw exceptions? Well, the other day I spent an hour or so familiarizing myself with an interesting perl feature courtesy of a module author not doing that. Hopefully he is more careful around traffic.

Anyway, here is the scenario. I wrote a module like this:

```
package My::Glue;
use Moose; # remember, this turns on strict

has 'id' => ( is => 'ro', isa => 'Int', required => 1 );
has 'some_other_class' => (
    is      => 'ro',
    isa     => 'Some::Other::Class',
    required => 1,
    handles => ['foo'],
);

# the calling package wants $me->id and $me->get_id. whatever
sub get_id { $self->id }
```

For those following along at home, we have four methods, `id`, `some_other_class`, `foo`, and `get_id`. Simple.

That class is loaded by a CPAN module that looks a bit like this:

```
package Glue::Loader;
use strict;

sub load {
    my $self = shift;
    my $module = $self->glue_module;
```

[hate](#)
[html](#)
[humor](#)
[info](#)
[irc](#)
[ircxory](#)
[java](#)
[leak](#)
[lisp](#)
[memory](#)
[metra](#)
[mojomajo](#)
[moose](#)
[mro](#)
[music](#)
[openbsd](#)
[papers](#)
[perl](#)
[photography](#)
[php](#)
[publicserviceannouncement](#)
[purl](#)
[rant](#)
[ruby](#)
[selenium](#)
[social](#)
[software](#)
[spaghetti](#)
[spam](#)
[sucky](#)
[tea](#)
[testing](#)
[twitter](#)
[web](#)
[weewar](#)
[yapc](#)
[モ一娘。](#)
[鉄道](#)

LOGIN

Not logged in. [Log in.](#)

```
eval "require $module";  
my $glue = $module->new( ... this is all fine ... );  
$glue->foo;  
return $glue->get_id;  
}
```

Great. Once `Glue::Loader` is setup such that `glue_module` is the first class we defined above, we call `load`. Try and guess what the output is.

There is none. Instead, an exception is thrown that reads
Can't locate object method "foo" via package "My::Glue".

What? It's right there in the handles.

Okay, so maybe I am using the `handles` syntax wrong. Let's just delegate manually:

```
sub foo {  
    my $self = shift;  
    return $self->some_other_class->foo;  
}  
  
sub get_id ...
```

Now we rerun `load` and everything is fine. Oh wait; no. If everything went fine you wouldn't be reading a blog post about this.

We actually get another exception,
Can't locate object method "some_other_class" via package "My::Glue"

What? So now we are calling `foo` just fine, but `some_other_class` doesn't work?

Well whatever. Maybe I have the `git` version of `Moose` installed and it's broken? Let's just comment out the delegation and add a
`warn "IT IS WORKING"` to see what's going on.

OK, time to run the program again. (BTW, aren't you glad you are using tests and aren't debugging this in your web browser. I am.)

The result? It prints "IT IS WORKING". YAY! IT IS WORKING!

Oh... but then it dies on the next line with
Can't locate object method "get_id" via package "My::Glue".

What? So the `sub foo` declared directly above the `sub get_id` works fine, but `get_id` doesn't work? WHAT THE FUCK. I AM SWITCHING TO RUBY!11

All in all, this is one strange debugging session. I can see the code right there! Why doesn't perl see it?

Well, it is an exciting combination of a number of features. The biggest problem becomes obvious when we remove the working code and just look at the stuff that's broken:

```
use strict;  
sub get_id { $self->id }
```

Does the phrase

Global symbol "\$self" requires explicit package name ring a bell? It should, because you can't just make up variable names and expect your code to compile under `strict`.

So that's one problem. But normally when you forget `my`, perl just dies and your program never gets past the `require` or `use` statement. In this case, though, the author of `Glue::Loader` forgot to check for errors after his `eval "require ..."` statement. Even though `My::Glue` failed to load, he uses it anyway.

The next piece of the bug is how Perl compiles subroutines and how it uses `strict`. I'm sure you're familiar with how `BEGIN{}` blocks work. Perl runs them, in order, at compile time. As it turns out, `sub foo {}` means the same thing to perl as `BEGIN{ *foo = sub { } }`. So what perl really does when it is asked to `require My::Glue` is:

```
BEGIN { require Moose; Moose->import }
BEGIN { use strict; *foo = sub { warn "IT WORKS" } }
BEGIN { use strict; *get_id = sub { $self->id } }
...
```

We die on the last line because strict kills us. Even though other stuff is in the module, it never has the chance to run, because the bad `get_id` definition unwinds the stack before we are even done processing `BEGIN`, much less processing the at-runtime stuff.

The interesting part is that perl doesn't "roll back" the partially-loaded module state; everything that worked, works. You can call `My::Glue::foo` just fine. The correct sub is in the symbol table.

This is why you normally kill the program after a bad module load -- you are now in a somewhat random inconsistent state.

And that's why I wasted an hour debugging my application. Someone forgot a simple `die if $@`. I couldn't see the error with my program, and some of the stuff perl managed to compile worked just fine.

(So why didn't `has` and `handles` work? Simple; all the Moose sugar runs at runtime; after the all of the `BEGIN` blocks. `has` is just a regular old function call.)

What you should take away from this is that you shouldn't roll your own module loading function. Just call `Class::MOP::load_class` and everyone will be a lot happier.

[no comments](#) | [Read more...](#)

The "function class" pattern

Tags: [lisp](#) [perl](#) [+]

Posted on 2008-9-21 (日) at 9:18 am by Jonathan Rockway 

Posted in: [Programming](#)

Here's some code that seems to turn up more and more frequently:

```
package Useful::Function;
use Moose;

has 'args' => (
    is      => 'ro',
    isa     => 'ArrayRef[Str]',
    required => 1,
    auto_deref => 1,
);

sub function {
    my $self = shift;
    my @args = $self->args;
    return ...;
}
```

Then it's used like this:

```
my $useful_function = Useful::Function->new( args => [1,2,3]
my $result = $useful_function->function();
...
```

I think people do this because they like type constraints and coercions, but it sure seems like an abuse of the class system.

Clearly we could rewrite the above class as a small function:

```
sub function {
    my @args = @_;
    return ...;
}

function(1,2,3);
```

Or, if we want to reuse the same arguments repeatedly:

```
sub make_function {
    my %params = @_;
    return sub {
        my @args = @{$params{args}} || [];
        return ...;
    }
}

my $func = make_function( args => [1,2,3] );
$func->();
$func->();
```

For validating the arguments, you can use `Params::Validate` or `MooseX::Method`. (And if you are writing a function-based module, you should also look at `Sub::Exporter`. It provides a lot of sugar, and can even automate the "currying" in the second example for you.)

It's not just Perl programmers that do this. I was reading some Lisp code last weekend and noticed this:

```
(defclass arguments-for-foo ()
  ((arg1 :accessor arg1 :initform 10 :initarg :arg1)
   (arg2 :accessor arg2 :initform 20 :initarg :arg2)))

(defmethod foo ((args arguments-for-foo))
  (+ (arg1 args) (arg2 args)))

(foo (make-instance 'arguments-for-foo :arg2 42))
```

I found it amusingly verbose, especially with CL's built-in `&key` lambda list keyword:

```
(defun foo (&key (arg1 10) (arg2 20))
  (+ arg1 arg2))

(foo :arg2 42)
```


Anyway, the next time you are writing a class, ask yourself "Isn't this just a function!?" If the answer is "yes", then just write a function!

Unless, of course, you are using Java.

[1 comment](#) | [Read more...](#)

Template::Refine

No tags [+]

Posted on 2008-9-7 (日) at 10:59 am by Jonathan Rockway 

Posted in: [Programming](#)

I released the first version of `Template::Refine` today. `Template::Refine` is my attempt to resolve the eternal conflict between developers and web designers.

I'm sure you've heard of this problem before. A developer and web designer want to work on a project together. The web designer only knows HTML, not programming. The developer only knows programming, not design.

This is a problem, because often times both things need to be in the same file. Templating systems usually try to make the programming part easy for the designer to avoid while still allowing the programmer some flexibility in controlling the page rendering. Unfortunately, the balance never seems to be quite right -- if the programmer abstracts too much away, the designer won't know which file to edit and won't be able to see her changes in the browser unless she has an instance of the application running. So although current templating systems try to make things easy for both the designer and the programmer, they usually make things easy for the programmer and hard for the designer. They were all written by programmers, after all. :)

I decided to solve this problem by writing a templating system that's easy for designers *and* programmers. Instead of demanding that the designer learn to

code (as with Mason or PHP) or that the programmer dumb his code way down (like Template::Toolkit or Clearsilver), I let the designer produce HTML and the programmer produce Perl to *refine* that HTML into a final product.

No more compromises! The designer only has to deal with HTML; HTML that is viewable in a browser with no additional effort. The coder can easily write complex code (and test that code easily) without the designer caring at all. Perfect! (BTW, I don't claim to have invented this idea. XSLT and HTML::Seamstress are prior art.)

Anyway, enough hype. Let's see how it works. We'll start with a very simple task -- variable interpolation. In Template::Refine, we look at document regions, not "control structures", so the input HTML can look like anything. The designer doesn't need to specifically mention in the template where \$foo is being interpolated.

In reality, the designer is probably going to give you a page with some test data where the interpolation should be, like this:

```
<p>Hello, <span class="username">world</span>.</p>
```

The username class is there because the designer always wants usernames to show up in a yellow-on-green font, but we can use this tag to know where to interpolate the \$username variable.

So let's do that.

We start by loading Template::Refine. The main entry point is currently Template::Refine::Fragment. This class represents a document fragment that can be refined.

```
use Template::Refine::Fragment;
use Template::Refine::Utils qw(simple_replace replace_text);
```

We also load some sugar, Template::Refine::Utils. Continuing, we need to load the document:

```
my $frag = Template::Refine::Fragment->new_from_file('welcome
```

Here we load the contents of welcome.html; you could also load from a string with the new_from_string constructor. (An XML::LibXML DOM tree is another option; see the POD for details.)

Now we need to write the rule to replace the test data with the real username in the right places. Template::Refine::Utils contains a function, simple_replace, that will return a Template::Refine::Processor::Rule object to do this. (It covers up a verbose-but-complete API for writing rules. You can read the POD or source if you want to know how the whole rule processing system works.)

simple_replace takes two arguments, a coderef that generates the replacement DOM node given the current DOM node, and an XPath expression that finds relevant DOM nodes. In this case, we want to find nodes that look like /*[@class="username"]. The code we want to run on each of those nodes will remove all the text inside the node and replace it with the username:

```
sub {
    my $node = shift;
    return replace_text $node, $username;
}
```

replace_text is another utility function. It will take \$node, make a copy, remove the copy's children, add a simple text node (using \$username as the text) as a child, and then return the whole thing. This has the effect of transforming a node like <p>Hello, world</p> to <p>Your text goes here</p>, which is what we want to do here. (You can build your own nodes with the XML::LibXML API if you want.)

Anyway, the whole rule will look like this:

```
my $username = 'Test User';
my $rule = simple_replace {
```

```
my $node = shift;
return replace_text $node, $username;
} '/*[@class="username"]';
```

Once we have the rule, we just need to apply it to the document fragment \$frag:

```
$frag = $frag->process($rule);
```

We assign the result of the process call to the original variable since `process` returns a copy; it doesn't modify the original document. I was in a functional-programming mood when I wrote the module. (Copying minimizes confusing side-effects, anyway.)

We can obviously call `$process` as many times as we want. When we are done processing, we just need to `render` the document to get our HTML back:

```
say $frag->render;
```

That will then print:

```
<p>Hello, <span class="username">Test User</span>.</p>
```

And that's that. The basic flow is:

```
Template::Refine::Fragment->
  new( ... )->
  process( simple_replace { ... } '//x/path' )->
  render;
```

Simple!

(BTW, in case you're wondering how it's possible to apply XPath expressions to document fragments and where `/` is, we reparent the fragment inside `<html><head /><body>...</body></html>`, yielding a full XML document. So `/` is the `html` node, and everything works.)

Let's do one more example -- this time, a bit more complicated. This is another situation I'm sure you've encountered. You have a form, and some fields are required. You want to annotate the labels of the required fields with a `*` so that the user knows they're required. (You also don't want the designer to do this manually, since you may change the requirements in your code at some point.)

Template::Refine to the rescue!

We'll start by defining a Moose class that is the "result" of submitting the form:

```
package Person;
use Moose;

has 'name' => ( is => 'ro', isa => 'Str', required => 1 );
has 'bio' => ( is => 'ro', isa => 'Str' );
has 'age' => ( is => 'ro', isa => 'Int', required => 1 );
```

Hopefully you're using Moose by now, but if not, I think the code is pretty easy to understand. `Person` has three fields, `name`, `bio`, and `age`. `name` and `age` are required. `bio` is optional.

Now let's see what the designer gave us as HTML:

```
<form>
  <div id="name">
    <span class="label">Name</span>: <input />
  </div>
  <div id="bio">
    <span class="label">Biography</span>: <input />
  </div>
  <div id="age">
    <span class="label">Age</span>: <input />
  </div>
</form>
```

You can see that each field has a region that can be selected with

```
//div[@id='<name>']. Inside that region, the label can be selected with
//span[@class='label'] or similar.
```

Here's how to write a `Template::Refine` rule for a two-stage scheme like this:

```
sub transform {
  my $frag = shift;
  for my $attribute (Person->meta->compute_all_applicable_a

    my $attribute_id = $attribute->name;
    $frag = $frag->process(
      simple_replace {
        my $n = shift;
        my $sub_fragment = Template::Refine::Fragment
          $n->toString,
      );
      return annotate_required_field($sub_fragment,
    } "/*[@id='$attribute_id']"
  );
}
return $frag
}

sub annotate_required_field {
  my ($fragment, $attribute) = @_;
  return $fragment unless $attribute->is_required;
  return $fragment->process(
    simple_replace {
      my $n = shift;
      return replace_text $n, $n->textContent . ' *';
    } q|/*[@class='label']|,
  );
}
```

The `transform` subroutine processes the entire form. For each attribute, it creates a rule that finds that attribute's region (the `div`). In the coderef that generates the replacement, we generate a `Template::Refine::Fragment` that represents only that region. Then we process the "sub fragment", this time looking for the labels (via `annotate_required_field`). For each label found, we get the existing text and add a `*` if the attribute's metaclass indicates that it is required.

The whole script (minus those two functions) looks like this:

```
use Person;
use Template::Refine::Fragment;
use Template::Refine::Utils qw(replace_text simple_replace);

my $frag = Template::Refine::Fragment->new_from_file('person_
print transform($frag)->render;
```

And the output is what you would expect from looking at the HTML and Moose class:

```
<form>
  <div id="name">
    <span class="label">Name *</span>: <input/></div>
  <div id="bio">
    <span class="label">Biography</span>: <input/></div>
  <div id="age">
    <span class="label">Age *</span>: <input/></div>
</form>
```

I think this is pretty cool. If you change the Moose class, the HTML changes. But the designer doesn't need to know code, and you don't need to know design! It Just Works. Yay!

I see a lot of applications for this. One that sounds interesting but I haven't tried yet is document localization. You can read the document in, find the area to translate (via `xpath`), look up the existing text in the English to msgid conversion table, find the string for the language you are looking for, and replace the English with the language you are translating in to. This means you can maintain your HTML in (mostly) English, and translate "out of band". It seems like it could work. (If you want more applications, look in my `Ernst` repository. I

am using `Template::Refine` for interfacing Moose and HTML.)

An interesting side-effect of using `Template::Refine` is that you have an introspectable DOM for all of your HTML. In times past, HTML was just some garbage that your app passed to the browser in hopes that it would understand it. Now your program actually understands the HTML, and can do intelligent things with it. For example, you will know immediately when your HTML is invalid (and `XML::LibXML` can't auto-fix it); the `Template::Refine::Fragment` constructor will die. You can find all "p" nodes and spell-check them. You can pretty-print the HTML for development, but compress it for production. The list goes on -- I think this is a pretty major change in how applications treat HTML. (Of course, everyone using XSLT has known this for a long time.)

Anyway, `Template::Refine` is a very simple module, so don't hesitate to try it out, read the code, and improve it! I think it's a nice compliment to other templating systems (you don't have to use it for everything), and it has a very high usefulness-to-complexity ratio -- `Template::Refine` is 400 lines of code, including docs. `Template Toolkit` is 26,000 lines!

Have fun.

[4 comments](#) | [Read more...](#)

Why I stick with Perl

Tags: [perl advocacy](#) [+]

Posted on 2008-8-4 (月) at 4:59 pm by Jonathan Rockway 

Posted in: [Programming](#)

I noticed a [discussion about Perl on Hacker News](#) this morning. The linked article is about why the author likes Perl. As with all articles of this type, it attracted the usual "Ruby is better!!" comments. The Hacker News discussion is *slightly* more intelligent, arguing that Perl is ugly, that the article's author "gave up on Ruby too soon", and that regular expressions are slow (as though that has something to do with Perl). Anyway, I thought I would address these points, and share with you *why I think you should use Perl*.

The first thing that you should realize when thinking about any programming language, Perl included, is that it sucks. It has tons of engineering trade-offs that you wouldn't have made, but you're stuck with them anyway. For example, the perl core is filled with legacy crap that doesn't work right and can never be fixed, for compatibility reasons. It trades correctness for compatibility. I would never do that, but I don't get to make that decision, because Perl is not my project. I could go on and on about what I don't like about Perl; trust me, there are hundreds of things that I absolutely hate and that make me want to cry -- but that's not the point of this article.

Every other language I've used has sucked just as badly as Perl. I really like Lisp, I think it's exactly the way a programming language should work. Provide the minimum, and let me build the rest. That way I will always be happy. But why did Common Lisp need to build in at least 3 separate ways to map keys to values (getf, assoc, and hash tables)? The answer is because the language designers wanted all three. Again, not my project... so I don't get to make the decisions. I can either live with them, choose another set of compromises that I don't really like (another language), or make my own language, which is yet another compromise. None are ideal. No language is perfect. No language you build yourself will be perfect, unless you have infinite time to write all the libraries that the non-perfect language already had. Tradeoffs, tradeoffs, tradeoffs. That's what engineering is.

So hopefully by now, you've thought of some reasons why you don't like your language of choice. If you haven't, it's because you haven't used it enough, and you aren't really in a good position to recommend or criticize programming languages. It took me a long time to realize that Perl isn't perfect. Before then, I just told everyone it was the greatest thing, and they probably laughed at me for being a shill fanboi. Now that I've grown up a bit, I can think rationally about programming languages. Hopefully, you can too.

Anyway, think about those features in your language that you hate, that really make you think "how could someone have been so dumb as to do it *this way!*" Think about the features from other languages that you love but are missing from your favorite. Really; do it. Don't you wish that Common Lisp had continuations? Don't you wish that Python executed faster?

With those reasons in mind, I think you'll find that they don't really stop you from

solving problems. If the problems made you seriously unproductive, you would have switched languages long before you gave those problems a second thought. Features that you think are harmful, you can simply not use. Features that don't exist can probably be emulated. (Python doesn't have lexical closures like Perl's, but you can emulate them with a class. Problem solved.) The end result is that the syntax and implementation compromises of your language don't really matter much. You call some functions, make some classes, throw in a dash of builtins, and your problem is solved. Yeah, maybe the syntax is ugly. But now you are rich and famous and can move on to problem number n plus one!

As you can see, real programs are rarely about syntax. If you need to speak HTTP, no awesome language feature is going to do that for you. Even if your language has The Perfect Featureset, writing an HTTP header parser is going to be a waste of your time. Libraries are the key to productivity; ignoring one piece of syntax you don't like is going to be a lot faster than reading the HTTP spec, implementing it, and testing it.

That's why libraries are important, they let you spend your time solving problems that haven't been solved instead of reinventing the wheel. If you don't have good libraries, you can't write good software.

As an example, the software that runs this blog (Angerwhale) uses 212 different Perl modules. If I had to write all that code myself, there would be a lot less functionality. And before you make fun of me for needing all those libraries "just for a blog", please think about all the complexity that you don't see. Some things that go on under the covers include working with posts encoded in any character encoding, supporting different rich text formats, computing the MD5 sum of the post for the HTTP ETag header, handing filesystem paths in a cross-platform manner, parsing the configuration file, reading PGP signatures, etc. It adds up fast. Without libraries, I would have a lot less features, or a broken piece of shit that kind-of-sort-of-okay-maybe-not works. (Incidentally, PHP doesn't have very many libraries...)

I am convinced that the Perl community knows libraries better than anyone else. We have the CPAN, with *tens of thousands* of distributions that already exist. Ten thousand is hard to comprehend, but try searching for your favorite programming topic on [search.cpan](#). You will probably find that your problem has been solved in an easily-digestible library form. Clicking a few links just saved you from a week of coding.

The important thing about Perl is that we have a *culture* of writing good libraries. No Perl programmer would write a few lines of code, post it to a blog, and call it a "library". Everyone feels obligated to create a CPAN distribution, with documentation (sometimes a bit on the minimal side, but not everyone is a writer), a test suite, a Makefile, etc. I'm not sure why, but this always happens. I think it's because there is a strong convention, and tools that make following the convention easy.

The existence of libraries make writing more libraries even easier. So the cycle feeds itself; we have a lot of libraries, that makes writing another library easier, now we have more, writing another library is even easier... and before you know it, we have more than anyone can imagine.

I've looked at other languages, and the communities just aren't like Perl. They set up sites for sharing libraries, but nobody contributes. Sure, RubyForge has a some libraries. PEAR has a few. Python has a lot. But there is no culture of writing or using libraries, and most people end up with a library or two next to some wheel reinvention and cut-n-paste code. (Search for "rails". Most of these programmers will depend on one library, Rails, but then get the rest of their code by cutting-and-pasting from blogs.) These people don't really understand libraries, and won't bother writing any themselves. As people continue to not write libraries, there continue to not *be* any libraries. Perl's huge number of libraries makes it easy to write more; the other languages' lack of libraries makes it hard to write more. So I don't think Ruby is going to magically have tens of thousands of libraries in another few years; they are going to keep doing what they're doing now. (Hopefully I don't need to explain why a packaging system and central library repository is a better strategy than cutting and pasting from blogs.)

The C community has also failed to understand libraries. It has a few, like GTK+, but most C programmers don't think about libraries when they are programming. I don't blame them; with no way to declare dependencies in your code, no namespaces, and no out-of-band error signaling, using and writing libraries is a horrific nightmare. I'm surprised that there are as many libraries as there are.

But it irritates me when I need to get at `gpg` from a web application, and can't

just use a "libpgg". I have to fork a gpg process, setup file descriptors just right, write input to a pipe, wait for input on another pipe, and then parse the result -- all for what amounts to a few XOR operations and bit shifts. How could anyone think this is a good idea? (There is libpggme, but this just hides the fork inside a library. There is still tons of totally unnecessary work going on.)

Git is similarly bad. If a programmer that had a culture of using and writing libraries wrote Git, most of Git would be classes that thin command-line scripts instantiated and mutated according to command-line arguments. There would be no important code in the scripts, only in the easily-reusable classes. This would make using git from another program trivial; just create an instance of the class, call a method, and you're done! Instead, since the C programmers never thought that anyone would need a library, you have to fork and exec a binary that returns the data you want in a custom format, that you then have to write a parser for. Great idea, guys. (Even git-fast-import, the easiest possible thing to make into a library, is just a big old monolithic binary. You have to learn a "simple" new language just to talk to it. Who thought that was a good idea!?)

So finally to the conclusion. If you are like me, most programming you do is about gluing things together with libraries. Perl may not be the prettiest language, but we have so many libraries that you won't have much time to see the ugliness. This is the reason why people stick with Perl, even though other languages have prettier syntax. It is so easy to get stuff done, and contribute your "stuff" back to the community, that we never even worry about the ugly syntax. It just goes away, and all we think about are solving our problems.

[45 comments](#) | [Read more...](#)

[Older articles >](#)

PAGE GENERATED BY [ANGERWHALE](#) VERSION 0.062 ON 2008-10-1 (水) AT 7:32 PM.

