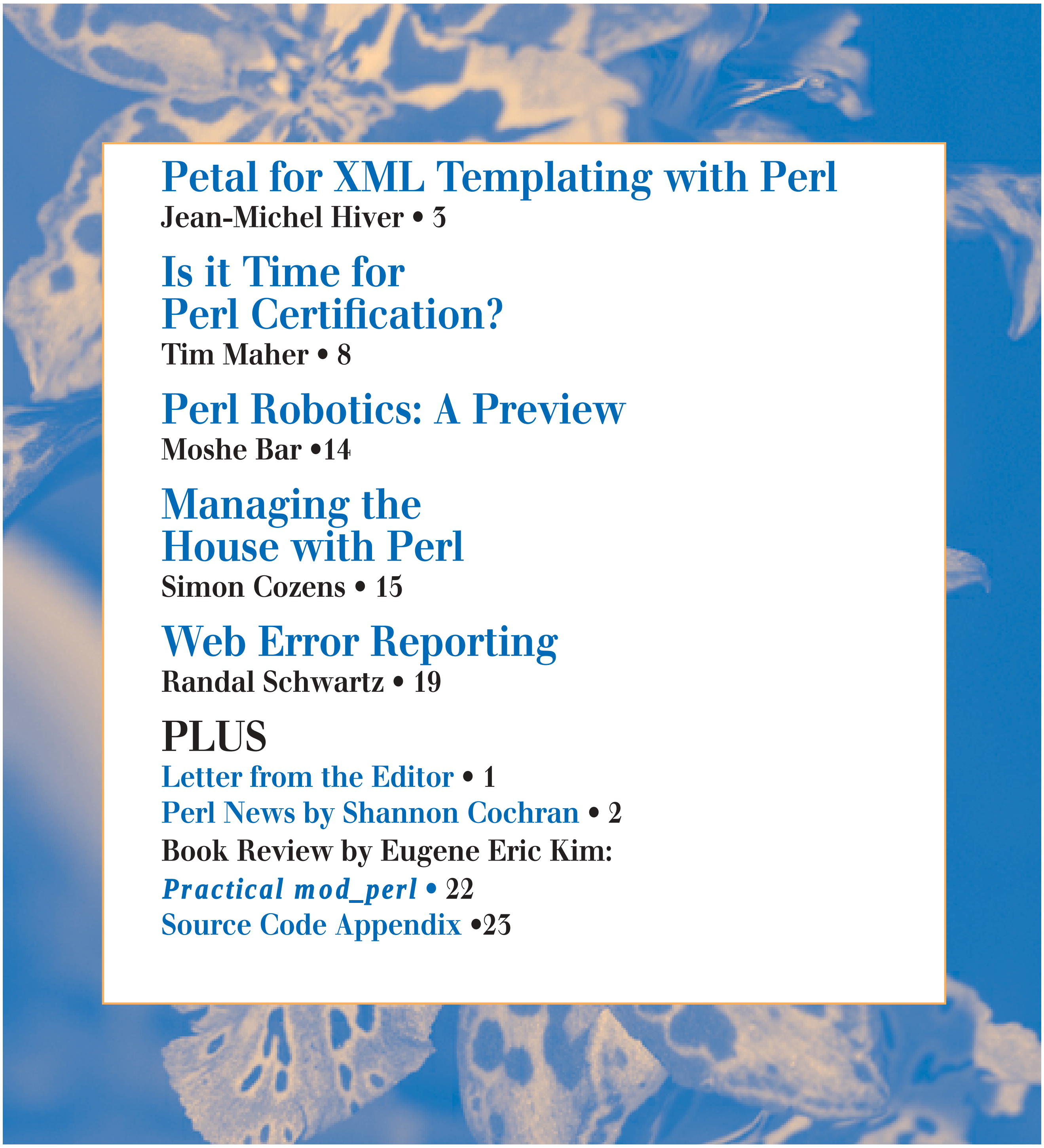
CMP
United Business Media

October 2003

The Perl Journal



LETTER FROM THE EDITOR

Certification

This month, *TPJ* brings you a call for Perl certification by Perl teacher and consultant Tim Maher. It's been a subject of some debate in the Perl world over the years. This month, we've given space to Tim to make his case. It's not the first time Tim has made this argument: You may be familiar with his position if you attended OSCON this year.

Why would we want a certification program? The general argument goes like this: Perl programmers are hard to hire because corporate HR folks don't know how to screen them for qualifications. HR departments rely (for better or worse) on devices like certification to cull the résumé pile. This is symptomatic of a larger lack of understanding of Perl in the corporate world. Project leaders who don't know Perl, aren't inclined to encourage Perl use in projects because Perl seems a bit untamed: perhaps a bit unknowable. Hence, Perl is underutilized, and Perl programmers are underemployed. Certification would give Perl gravitas and measurability, making it—and, therefore, Perl programmers—more attractive.

But the argument against Perl certification generally claims that such a certification couldn't possibly have any meaning. There are simply too many ways in Perl to solve any particular problem to be able to codify a clear "best practice." How would you test for one best way (or even several best ways) to do a task in Perl? And who decides what those "best ways" are? Perl was designed to deliberately make the answers to these questions ambiguous. Frequently, the proponents of this argument then point out that certifications of other languages are also bogus and serve no purpose but to line someone's pockets. Generally, the argument is that certification, in any form, is a boondoggle.

Why worry about it now? Lots of Perlies are unemployed. And they seem to be losing out to folks with Java and C++ credentials. And I agree with Tim when he says that's not because there's more Java and C++ development to be done than Perl development. In fact, I suspect there's a lot of semiclandestine Perl work going on in the big projects that are officially supposed to be using just the corporate-sanctioned Java and C++. Perl is just too handy for prototyping and testing for it to be truly banished from these environments.

But maybe it's a cultural difference that goes deeper than that. Maybe the managers who shy away from Perl just aren't comfortable with its ambiguities, certified or otherwise. For them, maybe TMTOWTDI is actually TATMWTDI (There Are Too Many Ways To Do It).

We know there's a divide in the Perl community over this issue. What do you think? Would a real certification program hurt Perl or help it? In the interest of a fair debate, we're eager to know what *TPJ* readers think about this issue. You can e-mail us at editors@tpj.com with your opinions. If *TPJ* readers speak their minds, we'll print some of those opinions in a "Letters" section in a future issue. What's more, if you have a different take on the question of Perl certification, and want to write an article as a counterpoint to Tim's, drop us a line.

Kevin Carlson

Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@tpj.com, faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.
THE PERL JOURNAL (ISSN 1545-7567) is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: $18.00 for one year. Payment may be made via Mastercard or Visa; see http://www.tpj.com/. Entire contents (c) 2003 by CMP Media LLC, unless otherwise noted. All rights reserved.

Shannon Cochran

Perl News

This month, Andy Lester fills us in on the plans for Phalanx, a test suite for Ponie. Do you have Perl news you'd like to share? Send it in to editors@tpj.com.

— Editors

Phalanx

Phalanx is a Perl QA project aimed at providing a solid testing base for Ponie, the next version of Perl 5 that will be based on the Parrot virtual machine. By increasing the test coverage of Perl modules and Perl itself, we will make Ponie the best-tested version of Perl ever.

The first phase of Phalanx will update the tests in 100 of the most widely used modules on CPAN. A Phalanx team member, or hoplite, will pick a distribution and, with the agreement and cooperation of the module's author, start working on the improvements. There may be one hoplite, or many, if the lead hoplite wants to bring others into that part of the project. Along the way, the hoplites will verify the accuracy of the documentation, explore the depth and breadth of the module, and make sure that everything that can be tested is tested. Once changes are made, the lead hoplite will feed patches back to the author, who will update the distribution. All changes are voluntary, and the author still retains full control of the module.

There are three goals for Phalanx. The first is to provide an excellent set of tests for the next version of Perl. Perhaps even more important, we want to encourage participation from members of the community who have never contributed back to Perl. It will be easy to get involved, and your involvement can be as much or as little as necessary. Prospective hoplites need not be part of perl5-porters or any perceived "Perl cabal," or even know about Perl internals. Finally, we're certain that we will uncover undiscovered bugs, and we'll identify them for the author, if not eliminate them.

After we've had some success in the first phase of Phalanx, we'll expand the process to Perl 5 itself and the core modules. Please visit the Phalanx web site at http://qa.perl.org/phalanx/. If you'd like to help out, join the perl-qa mailing list, or e-mail me at phalanx@petdance.com.

— Andy Lester

Pieces of Eight! Pieces of Eight!

The 0.0.11 release of Parrot, dubbed "Doubloon," is now available from CPAN and http://dev.perl.org/cvs. The new version features a long list of enhancements, including executable output, dynamic PMC registration, a trial exception system, the beginnings of an object system, iterators; ordered hashes, and I/O system improvements. Steve Fink wrote on use.perl.org: "Start from parrotcode.org for information on all Parroty things. Our immediate future plans include exceptions and objects, so now's a good time to jump in if those things grab you."

— Shannon Cochran

Perl 5.8.1 Imminent

As of press time, Perl 5.8.1 RC5 has been distributed, and the official 5.8.1 is expected within a few days, unless a nest of unexpected bugs is discovered burrowing within the release candidate. RC4 was tested for nearly two months, so the software would seem to be reasonably arthropod-proof. One update from the 5.8.0 release that may affect applications is the improved hash randomization. From the perldelta: "Previously, while the order of hash elements from *keys()*, *values()*, and *each()* was essentially random, it was still repeatable. Now, however, the order varies between different runs of Perl…For example, if you have used the *Data::Dumper* module to dump data into different files, and then compared the files to see whether the data has changed, now you will have false positives since the order in which hashes are dumped will vary."

Perl 5.8.1 RC5 is available at http://www.cpan.org/authors/id/J/JH/JHI/perl-5.8.1-RC5.tar.gz.

— Shannon Cochran

Kiel Oni Diras "Nyah-Hah" en Esperanto?

This year, under the auspices of YAPC::Europe, the London and Paris Perl Mongers held an auction to benefit The Perl Foundation; each agreed to rewrite the front page of their web site in the auction winner's language of choice. As it happened, however, neither the Anglophile nor Francophone contingents emerged victorious in the auction. Both were outbid by a group of Esperantists, who presumably saw in this bitter international rivalry a chance to promote the language of peace—and who, in the process, contributed 1300 Euros to the The Perl Foundation. Now both web sites must convert to Esperanto for a full month. Visitors to http://www.london.pm.org/ can already view the latest news of the "Londonaj Perl-Manipulistoj," but paris .mongueurs.net is asking for help in the translation project. To join the Esperanto effort, send e-mail with a blank subject line to majordomo@mongueurs.net, with "subscribe esperanto" in the body.

— Shannon Cochran

Jean-Michel Hiver

Petal for XML Templating with Perl

There are a lot of templating modules on the Perl scene: *HTML::Template*, *Text::Template*, *CGI::FastTemplate*, and of course, the well known and formidable Template Toolkit. Why create yet another one?

All of these systems suffer from serious drawbacks, not least of which is loop management: Graphic designers have to know about the programmatic structure of the loops, and make sure the *end* statements are correctly placed.

The Zope Corporation came up with a very elegant solution—the Template Attribute Language (TAL). TAL is an open specification that has been designed to make templates more compliant with WYSIWYG tools such as Adobe GoLive or Dreamweaver. There are many advantages to TAL, as described in Revuen Lerner's "At the Forge" article (http://www.lerner.co.il/atf/atf_98/).

In a nutshell, TAL lets you use extra XML attributes in a clever way so that you can control your templates. While TAL statements might look a bit alien at first, they integrate so nicely with the XML/XHTML syntax that template designers can simply ignore them—even for loop and conditional statements.

Clearly, TAL is a good idea. There was no Perl module for it. Stealing a good idea is often also a good idea. So I started hacking around with *XML::Parser* and *HTML::TreeBuilder*.

Eventually, I released something on CPAN and created a mailing list. A small yet dedicated community of Perl hackers joined in, submitting bug reports, patches, benchmarks, and tests. Very quickly, Petal (PErl TAL) became a full-featured, robust module.

A "Hello World" Example

Say you need a screen for an application that says hello to a logged-in user. The template looks like this:

```
<html>
  <body>
    <p>Hello, World!</p>
  </body>
</html>
```

Jean-Michel is a partner with Webarchitects, a UK-based company that specializes in usable, accessible, and standards-compliant Internet solutions.

Since Petal is an XML processing tool, the first thing you want to do on a template this simplistic is to send it through HTML *tidy* to make it valid XHTML as follows (the template just shown lives in file example1_01.html; all code from this article is available at http://www.tpj.com/source/):

```
$ tidy -asxhtml -asutf8 <example1_01.html
```

Which outputs:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
</head>
<body>
<p>Hello, World!</p>
</body>
</html>
```

Replacing Bits with *tal:replace*

Assuming that you are using HTTP authentication, the user login should be contained in the variable $ENV{REMOTE_USER}. In TAL expression syntax, this corresponds to ENV/REMOTE_USER.

Your template then becomes:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">
<html
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:tal="http://purl.org/petal/1.0/"
>
  <head>
    <title></title>
    </head>
    <body>
      <p>
        Hello,

  <spantal:replace="ENV/REMOTE_USER">World</span>!
      </p>
    </body>
  </html>
```

Petal::CodePerl is a subclass of Petal that compiles your templates to highly optimized Perl code

(This template is *example1_03.html* in the source code for this article.) As you can see, a few things were added. The attribute *xmlns:tal="http://purl.org/petal/1.0/"* means "all the stuff that is prefixed with *tal:* is in the Petal namespace." If you don't specify this, Petal will default to using the *petal:* prefix by default rather than *tal:*.

<span tal:replace="ENV/REMOTE_USER">World</span> means "replace this tag and its contents with the value returned by the expression ENV/REMOTE_USER."

Now, open up your template in your favorite browser: It still looks like the original mockup!

Sample Perl Script

Example 1 shows a sample Perl script to test our template. (This is file example1.pl in the source code.) As you can see, using Petal is pretty similar to other templating systems: You create a template object pointing to the template file you want to process, and then you execute the *process(%args)* method, passing a hash of arguments.

Conditional Display with *tal:condition*

One problem with this template is that if $ENV{REMOTE_USER} is undefined, you will get the following output:

```
<html>
  <body>

    <p>Hello, !</p>

  </body>

</html>
```

Which doesn't exactly look good.

You can add a condition as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">
<html
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:tal="http://purl.org/petal/1.0/"
>
  <head>
    <title></title>
  </head>
  <body>
    <p tal:condition="true: ENV/REMOTE_USER">
      Hello,
                                                              <span
tal:replace="ENV/REMOTE_USER">World</span>!
    </p>
  </body>
</html>
```

In the event that ENV/REMOTE_USER is false, *tal:condition* will remove the *<p>* tag altogether. Now, open this new template in your favorite browser or HTML editor. Despite the added condition, it still looks the same!

As you can see, this template is perfectly well-formed XML. There are many useful things you can do with it. You can edit it in Dreamweaver or GoLive, send it through HTML *tidy*, or check it for well-formed XML with *xmllint*.

But now, let's move on to a more advanced, and more interesting example.

More TAL Statements

Let's say that you want to write a script that takes the URI of an RSS file and turns it into beautifully crafted XHTML. You could try to figure out an XSL stylesheet, which would transform the RSS into the XHTML page you've been given, and then use an XSL processor such as sabolotron to do the job. Eventually.

Or you could use *LWP::Simple* to fetch the file, *XML::RSS* to parse it, and Petal to template it, and leave work much earlier.

Let's look at the script in Example 2. We are going to want Petal to access the following Perl values:

- *$rss->channel ('title')*, the title of the channel
- *$rss->channel ('link')*, the URI of the channel
- each element of the array *@{$rss->{items}}*, for example: *$rss –>{item}->[$index]->{title}* and *$rss->{item}->[$index]->{link}*

Now, let's look at this HTML mockup:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
                                              strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title></title>
  </head>
  <body>
    <h1>
      <a href="#">This is the RSS title</a>
    </h1>
    <ul>
      <li>
```

```
use strict;
use warnings;
use Petal;

$ENV{REMOTE_USER} = 'larry';
my $template_file = 'example1_03.html';
my $template = new Petal ( $template_file );
print $template->process ( ENV => \%ENV );
```

Example 1: Sample script to test the template.

```
use LWP::Simple;
use XML::RSS;
use Petal;
my $uri = 'http://mkdoc.com/rss100headline.rdf';
my $content = get ($uri) || die 'nuthin?';
my $rss = new XML::RSS;
$rss->parse ($content);
print Petal->new ('example2_02.html')->process (rss => $rss);
```

Example 2: Sample script for processing RSS values.

```
        <a href="#">RSS item</a>
      </li>
    </ul>
  </body>
</html>
```

The very first thing we want to do is to add the Petal namespace:

```
<html xmlns:tal="http://purl.org/petal/1.0/">
```

Then, using *tal:define*, we'll define a few aliases to save some typing in the rest of the template:

```
<html
xmlns:tal="http://purl.org/petal/1.0/"
tal:define="rss_title rss/channel --title;
            rss_link  rss/channel --link;
            rss_desc  rss/channel --description;
            rss_items rss/items"
>
```

As you can see:

rss/channel –title	maps to
$rss->channel ('title')	
rss/channel –link	maps to
$rss->channel ('link')	
rss/channel –description	maps to
$rss->channel ('description')	
rss/items	maps to
$rss->{items}	

You can use *some/stuff* and not worry about *stuff* being the key of a hash, an object attribute, or an object method. Petal will just Do The Right Thing.

Now, we want to replace the content of the *<a>* element within the *<h1>* element with the value of the *rss_title* variable. In order to do that, we use a *tal:content* directive.

```
<a
 href="#"
 title="some_title"
 tal:content="rss_title">
```

We also want to replace the *href* attribute with the value of *rss_link*, and the *title* attribute with the value of *rss_desc*. Both are replaced with the *tal:attributes* statement. Our *<a>* element becomes:

```
<a href="#"
  title="some_title"
  tal:content="rss_title"
  tal:attributes="href rss_link; title rss_desc">
```

Now, we need to iterate through each element of the *rss_items* array. In order to do that, we'll use *tal:repeat* on the *<li>* element so that we get one *<li>* per iteration.

```
<li tal:repeat="item rss_items">
```

This statement creates a new variable called *item* for each iteration of the loop. As previously, we use *tal:content* and *tal:attributes* on the *<a>* element, respectively.

```
<a
 href="#"
 tal:attributes="href item/link"
 tal:content="item/title">
```

Our complete template is shown in Figure 1. You can try it with the script in Example 2 (example2.pl in the source code).

So, to summarize the Petal attributes:

- ***tal:define*** – defines variable to use them for later.
- ***tal:condition*** – processes a tag and its contents under a condition.
- ***tal:repeat*** – iterates a tag through an array.
- ***tal:attributes*** – replaces tag attributes values.
- ***tal:content*** – replaces the content of a tag with the value of an expression.
- ***tal:replace*** – same as *tal:content*, but replaces the tag as well.

And that's pretty much what TAL is all about. Using these basic building blocks, you can template almost any kind of XML, while keeping your XML template well formed and compatible with whatever tool you use to edit, fix, or validate it.

Inside Petal

Let's examine what happens behind the scenes. Whenever you process an XML template using Petal, the library does (roughly) the following:

1. Read the source XML template.
2. $INPUT (XML or HTML) throws XML events from the source file.
3. $OUTPUT (XML or HTML) uses these XML events to canonicalize the template.
4. *Petal::CodeGenerator* turns the canonical template into Perl code.
5. *Petal::Cache::Disk* caches the Perl code on disk.
6. Petal turns the Perl code into a subroutine.
7. *Petal::Cache::Memory* caches the subroutine in memory.
8. Petal executes the subroutine.

This big, complicated procedure is really done only once. Subsequent calls to the same template will always resume at step 5 until the template file changes. If you are running a persistent environment a la *mod_perl*, then you get an extra bonus because subsequent calls to the same template in the same process will only involve step 8.

All this makes Petal pretty speedy. If you need even more speed, you can use Fergal Daly's most excellent *Petal::CodePerl*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:tal="http://purl.org/petal/1.0/"
  tal:define="rss_title rss/channel --title;
              rss_link  rss/channel --link;
              rss_desc  rss/channel --description;
              rss_items rss/items"
>
  <head>
    <title tal:content="rss_title">This is the RSS title</title>
  </head>
  <body>
    <h1>
      <a
        href="#"
        tal:content="rss_title"
        tal:attributes="href rss_link; title rss_desc"
      >This is the RSS title</a>
    </h1>
    <ul>
      <li tal:repeat="item rss_items">
        <a
          href="#"
          tal:content="item/title"
          tal:attributes="href item/link"
        >RSS item</a>
      </li>
    </ul>
  </body>
</html>
```

Figure 1: The complete Petal template.

Petal::CodePerl is a subclass of Petal that compiles your templates to highly optimized Perl code. This module is still experimental, yet it seems very sound.

Input Parser

Petal supports multiple parsing mechanisms. By default, Petal can parse your templates using the rather broken *XML::Parser* (I wish I had known that when I started to write Petal).

However, if you find *XML::Parser* too picky for your source templates (because your templates are not well formed), you can also use *HTML::TreeBuilder*, which will do a reasonable job at parsing somewhat broken XHTML.

You can enable this behavior as follows:

```
# use a local variable
local $Petal::INPUT = 'XHTML';

# or pass it as an argument to the constructor
my $template = Petal->new (

          file     => ,"gerbils.html,",

          input    => ,"HTML,",

);
```

Whichever parser you use will fire *XML::Parser* look-alike XML events, which will be used by a canonicalizer.

Output Canonicalizer

At this point, I have to confess: Petal started as just a big hack. It is built on top of a previous twist in which I used a syntax made of XML processing instructions as follows:

```
<?if name="something"?>
   ... do something <?var name="foo"?> etc...
<?end?>
```

Looks Familiar, Huh?

Instead of building a module that directly converts XML events into Perl code, I've built a module that converts XML events into the just described syntax, because this system was already working. I've also made this module turn an *$inline_variable* into a *<?var name="inline_variable"?>*.

The Petal canonicalizer was born. It turns alternate syntaxes (TAL, inline) into the original canonical syntax, which unfortunately happens to look very ugly. A neat effect is that you can use all three syntaxes at the same time, so this is legal:

```
<?if name="true:something"?>
  <p tal:repeat="item something">$item</p>
<?end?>
```

However, you should really stick with the standard TAL syntax as much as possible:

```
<p tal:condition="true:something"
   tal:repeat="item something"
   tal:content="item">Exempli Gratia</p>
```

If you want more information on the canonical syntax, see the perldoc Petal section "UGLY SYNTAX."

There are now two canonicalizer modules. The default canonicalizer produces generic XML. The XHTML canonicalizer has extra logic to deal with XHTML-specific syntaxic annoyances, such as: *<br></br>* needs to be *<br />*. If you want to output HTML, you want to change the output canonicalizer to XHTML:

```
# use a local variable
local $Petal::OUTPUT = 'XHTML';

# or pass it as an argument to the constructor
my $template = Petal->new (
          file     => ,"camels.html,",
          output   => ,"HTML,",
);
```

Code Generator

So your poor template now looks like a terrible mix of XML processing instructions and XML tags. Although this syntax is hard to read for a human, Perl-wise it is much easier to tokenize and parse. As its name implies, the code generator returns a string that can be *eval*ed. When the string is *eval*ed, it returns a CODE reference ready to be executed.

You can even substitute Petal's code generator with another one using the *$Petal::CodeGenerator* variable, but you probably will never need to do that unless you're into serious Petal hacking.

Modifiers

Imagine that you want to replace an attribute of a tag with more than a simple variable. For the sake of the example, let's call the attribute *bar* and its tag *foo*. You want to replace it with the value *"Hello, $user"*.

```
<foo bar="dummy value" />
```

Of course, you could directly replace *"dummy value"* with *"Hello, $user"* as follows:

```
<foo bar="Hello, $user" />
```

but this is not compliant with the TAL specification. If you want to keep your *dummy value* intact, you can use the *tal:attributes* statement combined with the *string:* modifier. So, instead of writing:

```
<!-- doesn't do the right thing -->
<foo bar="dummy value" tal:attributes="bar user"/>
```

You write:

```
<!-- does the right thing -->
<foo bar="dummy value" tal:attributes="bar string:Hello,
${user}!"/>
```

Modifiers are implemented using either code references or modules. Petal features the following default modifiers:

- *string:*, which as we have seen earlier interpolates expressions within a string.
- *true:*, which returns True if a value is true unless it's an array reference pointing to an empty list.
- *false:*, which you can work out.

The exciting thing is that it's possible—and dead easy—to write your own modifiers. For example, let's write an *uppercase:* modifier:

```
$Petal::Hash::MODIFIERS->{'uppercase:'} = sub {
      my $hash  = shift;
      my $value = shift;
      return uc ($value);
};
```

We can then write the following template:

```
<span tal:replace="uppercase:string:Hello, World!">hello,
world</span>
```

which will output:

```
HELLO, WORLD!
```

or we could write a *decrement:* modifier, which would decrement a variable:

```
$Petal::Hash::MODIFIERS->{'decrement:'} = sub {
      my $hash  = shift;
```

```
        my $value = shift;
    $hash->{$value}--;
    };
```

And try it on:

```
    <spantal:define="num 5" tal:replace="decrement:num" />
```

If the modifier you're planning to write is too big to be a single subroutine, you can write it as a module. The uppercase example becomes:

```
    package Petal::Hash::UpperCase;
    use warnings;
    use strict;

    sub process
    {
      my $class = shift;
      my $hash  = shift;
      my $expr  = shift;
      return uc ($hash->get ($expr));
    }

    1;

    __END__
```

And that's it: Petal will automagically pick up any file in *@ISA* that looks like "Petal/Hash/SomeWeirdPlugin.pm" and install it as "someweirdplugin:".

Conclusion

Fortunately, Petal is *not* "yet another templating module." It is a stable Perl implementation of an open specification from the Zope Corporation team. So far, TAL has been implemented in Python, PHP, and Perl.

Petal has all the bells and whistles you would expect from a good templating module: a modular architecture, proper Unicode support, decent speed, *mod_perl* friendliness, POD documentation, and a test suite. Moreover, the TAL specification lets you amend XML templates in a way that stays compatible with existing XML tools. With TAL, error-prone *<%end-loop%>*-style tags are a nonissue. Petal makes use of TAL to help you produce industry-standard-compliant data.

Finally, and more important, the main strength of Petal has been and will be its mailing list and its fantastic community of friendly and competent users.

So type "perl -MCPAN -e 'install Petal"' and join us!

References

Text::Template: http://search.cpan.org/author/MJD/Text-Template-1.44/lib/Text/Template.pm

HTML::Template:http://search.cpan.org/author/SAMTREGAR/HTML-Template-2.6/Template.pm

CGI::FastTemplate: http://search.cpan.org/author/JMOORE/CGI-FastTemplate-1.09/FastTemplate.pm

Template Toolkit: http://template-toolkit.org/

TAL of ZPT: http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL

TAL specification: http://zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.4

Petal: http://search.cpan.org/author/JHIVER/Petal-1.03/

Petal mailing list: http://lists.webarch.co.uk/mailman/listinfo/petal/

"At the Forge:" http://www.lerner.co.il/atf/atf_98

TPJ

Tim Maher

Is it Time for Perl Certification?

This month, Tim Maher calls for the Perl community to adopt a serious certification program, both to help individual Perl programmers make it through the hiring process, and to increase Perl's standing in the corporate IT market. What do you think? Your comments are welcome at editors@tpj.com.

— Editors

During my six years as a Perl educator and contract programmer, I've often pondered the problems facing the Perl community. One of these is the alarming state of un- or under-employment of many Perl specialists (including some of our brightest stars), while our colleagues who program in more prosaic languages such as Java and C++ enjoy more secure job positions.

To gain a better understanding of how Perl and its programmers are perceived in the industry, I've talked with managers of Information Technology (IT) departments and professionals working in Human Resources (HR) and recruiting agencies. This has led me to some conclusions about how we could improve our situation that I'm eager to share with you.

But first, you should know some of the pertinent aspects of my background. During my 12 years in academia, I obtained a wealth of experience in both taking and constructing examinations, and I studied techniques for computer assisted learning and testing. Later, while working with Sun Microsystems Inc., I had to take and pass the certification exams of the "Solaris System Administrator" series, and I provided feedback to help improve their quality.

These experiences have made me comfortable with testing technologies, but also highly cognizant of the need for testing to be done accurately and responsibly.

Before I turn to the subject of Perl certification, I'll review certain aspects of Perl's current status in the enterprise.

Perl's Image Is Cool, but Strange

It goes without saying that Perl is a marvelously expressive, extensible, and productive language that's fun to use. This has been noticed by many IT departments that value it as a general-purpose scripting language, a language for CGI or DB development, or one for cross-platform system administration.

Tim is the founder and leader of SPUG (Seattle.pm) and the CEO of Consultix, which offers Perl, UNIX, and Linux training. He can be reached at tim@teachmeperl.com.

Unfortunately, from the vantage point of old-school computer science types, Perl can also look like a "toy language" when compared to ones such as C++ and Java. That's partly due to Perl's lack of support for many features that let IT managers sleep better at night, such as strict type checking, compile-time function binding, standardized exception handling, and a conventional OO model.

In consequence, when it comes to critical software development projects — you know, the kinds that retain their participants even during recessions — Perl doesn't make the grade. It's summarily rejected for such projects, because their participants tend to view characteristics such as successful compilations in the face of missing subroutine arguments as *tragic flaws*, rather than charming indulgences of poetic license.

But Perl's historical laxity in these areas is by design, because Whipuptitude and Manipulexity (Larryisms, of course) are largely incompatible with, and have always had higher priority than, those aforementioned sleep-inducing properties.

Perl Should Cater More to Corporate Needs

Besides the intrinsic strengths and weaknesses unique to Perl's design tradeoffs, I believe there's another reason why Perl is often left sitting on the bench in the big games. Put bluntly, we've done a lousy job of pitching our language to the business community.

For instance, there are many IT managers who are willing to consider Open Source solutions, but are unsure how to view Perl relative to the languages they know. And that's perfectly understandable, because Perl's eclecticism is associated with lots of mixed messages, that would confuse any sensible person.

They're told that Perl is a scripting language, but not really, because it's compiled, but then again it's some strange kind of compilation that doesn't produce object code, so it's not really compiled, and it's a procedural language, or sometimes an OO one, but there aren't really classes or exceptions, and there isn't just one OO model—you can roll your own, and it's a *feature* that missing functions won't trigger warnings until they're called, rather than at compile time, and so forth.

Not to mention the fact that Perl programmers are notorious for cultivating idiosyncratic dialects of the language, that are almost as distinctive as handwriting styles, and can prevent one programmer from being able to read or maintain another's programs.

On top of all this, Perl advocates are inclined toward pronouncements like "It's *too hard to parse* to allow beautification," "It's so flexible it can be programmed in *poetry mode* with *autovivification* and *bleached code* or in *Latin* with *Klingon numbers*," and "It's too *multifaceted*, *expressive*, *advanced*, and, well, *artsy* and *esoteric*" to allow meaningful certification tests for its programmers.

(Regarding beautification, see my 1998 TPC presentation on the first "Perl Beautifier" at http://www.teachmeperl.com/perl_beautifier.html.)

Furthermore, as the last nail in the coffin, the only "Perl certification" identifiable with the Perl community itself is a *purposely bogus one*.

Given the technical shortcomings just summarized, and PR like this, is it any wonder that other languages, including the most stodgy, cumbersome, and uninspired ones, are eating Perl's lunch in thousands of corporate cafeterias?

But there's a development in the offing that could help improve our situation.

Perl 6 Will Have More Enterprise Appeal

The wondrously reworked Perl 6, when it arrives, will go a long way toward stemming many of these concerns about the fickleness and eccentricity of our language, because it promises to give programmers the option for more rigor right where the IT managers want it. That means they can have the best of both programming models—the traditional "expressive and intuitive and loose" Perl, and a new, "more rigid and bulletproof" variation. It will still be Perlish, but suddenly no longer a weak sister to C++ or Java.

Perl 6 is justifiably expected to stimulate a widespread reconsideration of the important roles that Perl can play in IT departments, and an increase in new hiring for JAPHs (JAPH means "Just Another Perl Hacker").

But how will hiring managers be able to confirm that the applicants for the new Perl jobs really know Perl 6? An answer that would readily occur to those managers would be "through *certification*."

But while we're waiting for Perl 6, is there anything we could do to improve Perl's image?

How Can We Help Perl Get the Respect it Deserves?

What would be the best way to improve the (somewhat motley) position of Perl in the corporate world, so it would be more frequently chosen for important applications, and Perl programmers could get placed in more secure positions?

How about improving Perl's documentation? That couldn't help; Perl's documentation is already the best in the industry.

How about creating a worldwide network of support groups for Perl programmers? We've already got one. It's called "Perl Mongers."

How about making Perl more robust? Perl 6, which will be more robust in all the right areas, is on the way.

How about launching a PR campaign extolling Perl's virtues? It's probably too late for Perl 5, whose strengths and weaknesses are already well known, for any attempt at "spin" to be very successful. We'll have a lot more to boast about when Perl 6 comes out, so it might be best to delay this kind of effort until then.

How about creating a certification program for Perl programmers? Hmm…that might be just what we need!

Establishing Perl skills as *certifiable*, and the Perl community as willing to comply with accepted hiring protocols, could cast Perl in a totally new light. First of all, hiring managers would be inclined to see Perl as more stable and conventional, because certification (unlike "poetry mode" and "bleached code") is considered a hallmark of serious languages. Second, they'd realize that screening Perl programmers would suddenly be no more difficult than screening Java programmers, Oracle Database Administrators, or Linux System Administrators.

If we don't rise to the challenge to do it properly, some opportunistic corporation might beat us to the punch

And of course, these benefits at the hiring end of the equation would make it easier for managers to consider basing additional projects on Perl, because of the greater ease in staffing them.

Finally, when Perl 6 arrives, managers would realize that the new Perl would be suitable for the most critical enterprise applications, offering additional incentives for increasing Perl development and JAPH hiring.

In a nutshell, these are the conclusions I've come to, and my recommendations to the Perl community. In the remainder of this article, I'll provide some details that will help you see how I've arrived at these conclusions, and help you make up your own mind about this important issue.

There's a Demand for Certification

As a quick search with Google will confirm, there are several vendors currently offering certificates of Perl competency based on online tests, and one of these has reportedly been designated as a requirement for job applicants at certain companies.

The very fact that some have found it necessary to qualify applicants for Perl jobs on the basis of certificates of dubious value (more on this in a moment) indicates a real need for a legitimate Perl certification service that's going unfilled.

Through my involvement with the Perl community, I've made contact with many hiring managers who are also Perl programmers and advocates, and they tell a similar story, which goes like this: They'd like to use Perl more widely in their (old-school, traditional) businesses, but they feel like they're swimming against the corporate tide. Eventually, they get tired of championing an underdog, and they ultimately settle on another language that's easier to defend to colleagues and more amenable to HR screening practices. (This reminds me of the old "Nobody ever got fired for buying IBM" ads that appealed to the CYA demon whispering in every manager's ear.)

One of the major complaints of these managers is the lack of any help from the Perl community in validating the skills of a job applicant. That puts the burden of vetting essential applicant skills directly on the shoulders of the manager (or his staff). And they know that's not a burden shared by managers hiring Java programmers, because the Java language is associated with a series of professionally designed and administered certification tests which are widely respected as evidence of competence.

But what about C++, Perl's other major competitor in the enterprise? Like Perl, it lacks a widely accepted certification program, but that certainly hasn't prevented it from reaching widespread acceptance.

The crucial difference between Java and C++ is that C++ was there first, and had an existing base of corporate interviewing teams that felt comfortable hiring C++ programmers. When Java emerged as an alternative, its proponents had to work harder to make it appealing to the business community. That's why they went to the effort of developing a certification program, to make it easy for HR departments to perform initial screenings of Java job applicants, and to make the new language easier for managers to adopt.

We in the Perl community are in many ways in the same "underdog position" as Java initially was, so we'd be wise to take a page from Java's book, and make it as easy as possible for companies to hire Perl programmers.

Clearing the First Hurdle

Larry Wall's whimsical thoughts on the "Three Virtues of Perl Programmers" are well known in the Perl community. An even more essential bit of knowledge for JAPHs is the first hurdle that must generally be overcome by a job applicant, especially when dealing with a big company. That hurdle, called "screening," is erected by the HR department, and those who surmount it get their résumés onto the hiring manager's desk, while the others have theirs consigned forever to the dreaded *circular file*. (The same process is also commonly used by recruiters working for placement agencies, but for simplicity, I'll refer to this as an HR activity.)

The important thing to understand about HR departments is that their technical knowledge is limited to buzzwords and certifications. Accordingly, when 200 résumés show up on Monday for the single Perl job advertised on Sunday (as can happen in the USA), HR starts the screening process in a frenzy, to reduce that stack to the much shorter one desired by the hiring manager.

During the winnowing process, in the absence of explicit instructions to the contrary, résumés that are missing that prized screening credential—a relevant certification—generally get trashed. In fact, the screeners might in their zeal even trash the résumé that says "Larry Wall" at the top, if they're lucky enough to get it.

In one case reported to me, this ruthless screening process caused 100 percent of the applicants for a Perl-only position to be ruled ineligible for an interview! In other situations, such as positions that invite both Perl and Java applicants, this process has put JAPH contenders at a huge disadvantage to their (more commonly certified) Java competitors.

OSCON Attendees Voted *For* Perl Certification

At the 2003 O'Reilly Open Source Convention (OSCON), I moderated a Panel Discussion on Perl certification (http://conferences.oreillynet.com/cs/os2003/view/e_sess/3747), featuring a diverse and distinguished panel, including Perl 6 designer Damian Conway. The discussion centered around the pros and cons of having a certification program for Perl programmers.

Approximately 200 people attended the session, and during the open discussion period, many posed questions to the panelists, and shared their own experiences and views. As any seasoned observer of the Perl community would guess, passionate arguments were heard on both sides of the issue.

But something happened at the end of the session that surprised all the panelists, and every Perl community "leader" with whom I've since discussed it (but interestingly, not many Perl "followers"). Specifically, Damian called for a show-of-hands vote of attendees *for* and *against* the development of a certification program for Perl programmers, and the *for*s won by a wide margin of approximately 14 to 1 (as agreed by the author, Damian, Nat Torkington, Tim Wilde, and others; for further details see http://teachmeperl.com/perlcert/OSCON_vote.html).

That outcome was a surprise because previous discussions on this subject had not shown a strong majority in favor of certification. But the fact that those prior exchanges were often dominated by influential figures arguing against the idea provides a possible explanation for this discrepancy. Specifically, I suspect that the less-famous members of our community might have been reluctant to go on record in a public forum as expressing disagreement with the more famous ones.

But unlike the environment offered by a newsgroup, a mailing list, a wiki site, or a spirited discussion in a pub, the OSCON show-of-hands vote should have provided an environment where individuals could feel more free to express their views on this controversial topic—after all, their names were not even requested, let alone stored on the Internet for all to see.

We've done a lousy job of pitching our language to the business community

This vote provides the most specific and credible evidence we've ever had of community opinions on this topic, and it tells us that JAPHs are clearly in favor of making certification a reality.

Certifications

What exactly is certification? Professionals, in a wide variety of specialized fields, obtain certification as a way of establishing their knowledge, whether to satisfy licensing conditions imposed by regulatory authorities (CPAs, attorneys, doctors, auto mechanics, and so forth), or as an aid in convincing prospective employers of their skills.

Many software technologies have serious, standardized certification programs including Java, VisualBasic, Visual C++, Oracle, DB2, and various flavors of UNIX and GNU/Linux.

Historically, Perl programmers have had the opportunity to acquire four types of credentials attesting to their knowledge of Perl. These credentials vary widely in price, sophistication, credibility, and value, and they are listed below.

School certificates: In recent years, certain progressive institutions of higher education have established Perl training programs that award certificates to their graduates. I'm on the advisory board for such a program at the University of Washington.

That program covers a fairly comprehensive range of programming topics, and awards a pass/fail grade based partly on the student's ability to submit acceptable source code. Instructors have included Perl specialists from Amazon.com and the Slash project, among others, who are familiar with current Perl programming practices in both corporate and Open-Source development environments.

Because students who earn these types of certificates have not only learned practical uses of the language but have also demonstrated an ability to program in it, these are generally considered the most impressive certificates currently available.

Training vendor certificates: Many training vendors award "class completion certificates" to students who attend training classes. Although there was a distant time when it was not unusual for final exams to be administered on the last day of such classes, in recent times, the industry standard has shifted toward awarding these certificates largely on the basis of *attendance* (can you say "self esteem movement"?).

Naturally, such credentials are of rather limited value in themselves, but they still signify something valuable. That's because the vast majority of students attending such classes actually apply themselves and learn the material.

Dubious certificates: With a little Googling, one can find a few vendors that are currently offering Perl certifications, based exclusively on brief on-line tests, at very low prices ($50 or less).

But even if these tests were masterfully designed and initially validated under controlled conditions by accepted psychometric procedures, the certificates they're now awarding would have to be viewed with circumspection. That's because these vendors are testing individuals with unconfirmed identities under unregulated conditions, which allows:

- Anybody to pose as John Doe to get "him" certified.
- The real John Doe to take the test, while obtaining answers from somebody else.
- The possibility that a prospective future testee could print the test, get somebody to tell him the answers, and then later pass the test—without ever learning anything about Perl! (This assumes the test questions are drawn from a small pool, which is a safe bet for such cheap programs.)

These uncontrolled testing conditions represent an egregious violation of psychometric requirements, including "validity," the property that the test (as administered) is truly assessing the testee's knowledge of the subject, and "reliability," the property that a similar grade would be expected on a retest.

Joke certificates: There are well-known members of the Perl community who are strongly opposed to certification. In fact, in an exhibition of admirable entrepreneurial spirit, a few of them started selling fancy and personalized, but *totally bogus* Perl certifications years ago. Their admitted motivation is to make it difficult for any serious attempt at a Perl-certification program to gain acceptance, by flooding the market with these cheap, fraudulent certificates, which hiring managers will think are legitimate.

It's understandable that anyone who had been offended by a laughingly defective certification test would find the cynicism underlying this prank to be amusing. But despite its Pythonesque appeal, this "disservice" has, thankfully, not caught on.

But I shudder to contemplate the message this is sending about Perl to the IT community, which is one of disrespecting accepted corporate hiring practices, and actively plotting to preserve the current difficulties that HR professionals face in hiring JAPHs.

Serious Perl Certificates

Never in Perl's history has there been a Perl certification program that was widely recognized by employers or endorsed by community leaders. Far from it. I think we in the Perl community should change this, by creating a *serious* certification program.

Its tests should be:

- Designed by subject matter experts.
- Compliant with established psychometric principles (validity, reliability, and so on).
- Administered under regulated conditions.
- Controlled by a respected organization.
- Endorsed by community leaders and leading corporations.
- Sensitive to feedback from testees, to facilitate identification and correction of (inevitable) errors.
- Optional, based on an understanding that one's experience and track record should be recognized as alternative ways of establishing one's qualifications.

Beneficiaries

The beneficiaries of a serious certification program for Perl programmers would include:

- Recruiters, HR departments, and hiring managers who need assistance in screening, classifying, and ranking applicants for Perl jobs.

Establishing Perl skills as certifiable could cast Perl in a totally new light

- Perl programmers, who would have an opportunity to obtain Perl credentials that would be compatible with established corporate hiring practices, and who would have help in identifying the gaps in their knowledge, so they could better their skills.
- Booksellers, publishers, authors, training vendors, colleges, and testing centers, because there would be an increase in demand for Perl-related educational materials and services. (See http://teachmeperl.com/perlcert/beneficiaries.html).
- The whole world of Perl, through a formal definition of the essential components of the basic language and the specialty areas; through the enhanced professionalism of our image that would more accurately reflect the value of our language to the enterprise; and through revenues flowing back into the community, if certification becomes profitable.

Perl Certification as a Rorschach Test

The phrase Perl Certification seems to conjure up a nightmare version of a Rorschach Test for some JAPHs, who imagine the worst possible interpretation of an ambiguous stimulus, and react accordingly. They picture an obligatory, monolithic, multiple-guess trivia test that strives to encompass everything Perlish, takes five hours, and is conspicuously missing correct answers for 100 of its 500 questions.

What's worse, it has to be taken at a special test center, in the next major city. And it must be retaken annually. Oh, and it costs $500 each time. Not including tax. But no checks or charge cards are accepted, only U.S. dollars. Small bills only, $20 max. Must be in mint condition…

A test like this, which is not (altogether) unprecedented in the industry, would function as more of a "programmer tax" and "anger management challenge" than a test of Perl knowledge. But, this nightmare scenario aside, I suspect that most of certification's detractors wouldn't really oppose a well-designed, well-executed, sensible, and fair certification program, especially if it could help our community, and if they had direct input into its design.

Why Rock Perl's Boat?

Okay, so Perl has survived all these years, acquired a large following, made some inroads into IT circles, and accomplished all that without the benefit of a respectable certification program. Why should we make such a dramatic change as to adopt a certification program now?

I'll give you four reasons:

Perl skills are not properly valued. In my capacity as the job-listings liaison for SPUG (aka Seattle.pm), I notice changes in the job market for Perl programmers, and it's obvious that they have suffered disproportionately during this extended recession.

One disturbing trend is that many IT managers have retained their Java and C++ programmers in recent years, while laying off their Perl programmers. Of course, some of this layoff disparity is rightfully due to the "mission critical" nature of the jobs some Java and C++ programmers are doing, versus the less glamorous "glue" jobs performed by many Perl programmers. (But if those managers only knew how much of the useful output from their Java and C++ programmers was really derived from clandestine Perl usage, they might have more respect for their JAPHs!)

These JAPH-dumping IT managers are probably the same ones who practice lunchtime economizing by skipping the previously routine $3.45 latte, while maintaining the tradition of the Whopper or Big Mac. And of course, Perl (and its JAPHs) would be better off in the entrée category!

I've also noticed another aspect of JAPH under-appreciation. In recent years, many smart and capable members of SPUG, after being laid off from their Perl jobs, have found it necessary to earn certifications in Java or C++ in order to gain new employment.

Wouldn't it be better for the Perl community if they could secure employment by obtaining certification in Perl *instead*? With the arrival of the newly robustified Perl 6 that will make Perl a favorable alternative to its competitors, and a "business makeover" to allow Perl's PR to more accurately reflect its newfound capabilities, I think we could change the workplace into one where that could happen.

Perl 6 heralds a clarion call for certification. In my optimistic vision of the future, I see a world where the Perl 6 team members will all have sufficient job security that they can devote more time to their back-burner activities, and finish Perl 6. And it will garner great reviews, and rekindle the interest of IT executives, who will be encouraged by its obvious superiority to give the new Perl a chance for more widespread use in the enterprise.

By then, the U.S. economy will have had plenty of time to turn around, so we might even experience a new Golden Era (or more likely a Bronze Era, but that's good enough) of high-tech corporate hiring. You know, sort of like 1998–2000, but without the unsustainable and unhealthy exponential components.

And guess what the first question will be that recruiters, HR executives, and hiring managers will be asking applicants for the scores of newly allocated Perl 6 development positions? I'll tell you: "Before we go any further, exactly how much do you know about Perl 6?"

The advent of Perl 6, with its greater rigor, robustness, and "conventionality" in certain critical areas could be our golden (okay, *bronze*) opportunity to win market share from other languages. But how easily that will be accomplished will depend on how easy we make it for corporations to come up with confident assessments of our Perl 6 skills, to hire us, and then to be dazzled by the wonders we can achieve with it.

But to make the most of the opportunity provided by Perl 6, we should have a mature and respected certification program already in place when it arrives.

Otherwise, no matter how interested IT managers might be in giving Perl 6 a chance, there will still be significant corporate obstacles blocking its widespread acceptance—such as HR departments that don't know what to do with résumés from JAPHs except to file them in the wastebasket.

The boat is already rocking. The pertinent question is not "Why rock Perl's boat," because in fact it's already rocking. The more appropriate question is "How do we keep it from capsizing?" We're losing market share to other languages, such as Ruby, Python, and PHP, whose main contribution is providing more prosaic and conventional ways to get at Perl-like capabilities.

I've even heard of IT departments hiring nonprogrammers and teaching them to write CGI programs in PHP rather than hiring experienced JAPHs to do those same jobs with Perl, because they don't feel comfortable with Perl's unrivaled peculiarities, complexities, and freedom for individual expression. We need to work on reversing such trends in the marketplace, and soon.

Perl 6 could be our golden opportunity to win market share from other languages

Certification is worth a try. I know of no case where the introduction of a serious and well-managed certification program for a software technology has ever resulted in changes that were largely detrimental to the associated community. Sure, there tends to be some initial grumbling about the testing fees by those who choose to seek certification, but if it helps them obtain and keep employment, and improves the image of their technology, and increases the value attached to their skills, that's all beneficial.

We know we have problems competing for enterprise programming jobs now, and that the advent of Perl 6 will create new demands for certification. Sitting on our laurels surely won't help us face these challenges, but developing a certification program seems likely to help. There are risks involved (I'll get to these in a moment), but I think the odds are in our favor—in large part, because we, in the Perl community, as the developers of the program, would be in charge!

That means we'd have the freedom to devise any kind of unconventional certification regimen we might fancy—so long as its results are reducible to a few words on a résumé, for the convenience of HR departments. For instance, in keeping with the TMTOWTDI principle, credit could be given for knowing *any* correct solution to a programming problem, rather than a particular one, so those who know different dialects of Perl could all obtain certification.

And the "HR-friendly" words associated with testing might include "Perl Certification, Level 1: Passed with Distinction," as well as "Acknowledged Perl Guru" (for those granted testing waivers, on the basis of their code portfolios)—if that's what we want.

The Downside

I've outlined my views about how a serious Perl certification program developed by our community could benefit us. But as any maintenance programmer can tell you, the introduction of any new element into a complex system raises the probability that things will go awry.

My perspective on these concerns is simply this: If we take on the responsibility for this task as a community, we'll be in charge. So if we see things going awry, we can take corrective action. If we do this well, individual JAPHs with good ideas will have a much better chance of effecting changes in Perl certification than they would have with other programs, run by large corporations, that are looking out for their own vested interests.

Certification only needs to yield a net gain to be a success. There will undoubtedly be some undesirable repercussions of introducing such a program so late in the evolution of Perl and its community. But if the results are largely beneficial, especially in the critical employment arena, we'd be foolish not to seize the opportunity to make the world a better place for JAPHs.

And we must not overlook the fact that there are also risks associated with inactivity. Specifically, if we don't start developing our own serious program for Perl certification, and soon, somebody else might do it for us—or perhaps the more appropriate phrasing would be *to us*.

Policy Precedes Implementation

Although I've written and spoken elsewhere about my own ideas for implementing an optional, state-of-the-art multilevel certification program for Perl, and several others have also offered creative ideas, I've purposely avoided matters of implementation here.

That's because the unavoidable nitpicking involved in design and implementation must not be allowed to get in the way of rational decision making, and we have a very important decision to make at this juncture.

Given that a majority of community members expressed a desire at OSCON for Perl to have a certification program, the question we must now collectively answer is: "How should we respond to the demand for Perl Certification?"

If we decide to "make it so," then we should proceed to collectively determine what the properties of that program should be and work on creating it. But there is one formidable obstacle that we'll have to overcome.

The Biggest Obstacle: Us!

We in the Perl community are both our greatest asset and our greatest liability (but life's like that!). If an upstart language as great as Perl had been created by a corporation (think Java, but with more inspiration), the business-friendly infrastructure would have been incorporated from the start, just as surely as The Larry felt the need to provide **a2p** and **s2p** with the first release of Perl. So that hypothetical language would have had an effective PR program and a certification process long ago, and the advantages that accrue from them. (By the way, **a2p** and **s2p** are, respectively, awk-to-Perl and sed-to-Perl translators. Larry provided these to automate conversion of programs written for those UNIX utilities into Perl programs, with Perl's initial release. What a guy!)

However, along with having greater intelligence, creativity, generosity, sociability, and tribal spirit than your average geek, JAPHs also exhibit greater independence and nonconformance. And that (sometimes unfortunately) includes a tendency to flout the established conventions of the corporate world.

But this is to be expected—it's no accident, after all, that we've been brought together under the banner of TMTOWTDI. Nor is it an accident that most of the greatest accomplishments in Perl culture (CPAN, CPANPLUS, DBI, PAR, perltidy, the Perl 6 design, the Parrot interpreter, TPJ, the Camel book, the Perl Cookbook, Damian's OO Perl book, TPF development grants, Perl Mongers, Perlmonks, and YAPC) have been achieved either by individuals working alone, or in very small groups of like-minded colleagues.

In recognition of this, it would seem that our best chance for success would be to have a small group of people, who show evidence of being able to agree on certain core principles, oversee this project.

But we'll have to avoid getting dispirited by those who will warn us of "insurmountable problems" and a "pestilence on all of Perlity" if we dare to establish such a program. Although this kind of input can sometimes freeze people into inaction, we need to remember two things: 1) Many groups have already done what we'd be doing (for example, the Linux community), and 2) We have an unmatched pool of talent, ingenuity, and perspicacity in our community to apply to the effort.

So there should be absolutely no doubt about our ability to make this happen, and make a success of it, if we should choose to take on this task.

Conclusion

Why should we create a certification program?

- HR departments want JAPHs to be certified to facilitate preliminary screening of job applicants.
- Managers want JAPHs to be certified to make their individual skills easier to compare.
- Catering to accepted hiring practices should promote greater hiring of JAPHs and greater acceptance of Perl in the enterprise.
- The advent of Perl 6 should cause renewed interest in Perl, more Perl positions in mission-critical application areas, and therefore, a heightened demand for applicants certified as having the latest Perl skills.
- Invalid "certification programs" that tarnish Perl's image should rightly have to compete with a serious one created by those who really know the language and have its best interests at heart—the Perl community.
- Vendors are already providing this service with various degrees of sincerity and sophistication. If we don't rise to the challenge to do it properly, some opportunistic corporation might beat us to the punch, and wrest control of this important aspect of our culture from us.

It's Time for Perl Certification

My recommendation is that we immediately start working on a certification program for the essential skills of Perl 5. Once that proven testing infrastructure is in place and accepted by the corporate world, we can start creating new tests for Perl 6, while continuing to develop add-on certifications for Perl 5 (which should remain an important language through the rest of the decade).

How to Get Involved

It would seem most natural to develop a community-based Perl certification program under the auspices of a community-wide organization. Although The Perl Foundation is the only one we have that comes close to being appropriate, and its leader is interested in this issue, she has thus far declined to take an active role in it.

But if we're to make any headway on this challenge, somebody will have to act as a coordinator—so I volunteer.

As a first step, I encourage those interested in helping to develop a Perl certification program to subscribe to the Perl Certification Mailing List, at http://perlocity.org/cgi-bin/mailman/listinfo/perlcert. Then post a message announcing any special qualifications you might have (such as experience in academic testing or corporate hiring) and indicating what roles you might be willing to play in this effort (test designer, fund-raiser, business liaison, hosting provider, and so forth).

We'll take it from there—as a community!

Acknowledgment

The author is indebted to the following reviewers for helpful comments on earlier versions of this article: Damian Conway, Terry Nightingale, Rodney Doe, Nancy Corbett, Christie Robertson, and John Michael Mars.

TPJ

Moshe Bar

Perl Robotics: A Preview

I recently received an AIBO robotic dog for review (see http://www.aibo.com/). AIBO is a Sony product geared at the geek market as well as the upscale-toy market. It features dozens of motors to let the robotic dog move around the house, speak, take pictures, and use Wi-Fi to connect to your home network. The heart of the dog (not too literally speaking, though) is the MIPS 64-bit CPU running at 180 Mhz and a real-time, proprietary operating system.

The AIBO is a lot of fun for the first few days, as it roams the house looking for interesting objects (it takes pictures of them and emails them to you). It understands natural language (over 75 commands) and knows how to walk back to the base station to recharge its batteries.

However, all this gets boring soon enough, and it is only then that you discover the usefulness of the AIBO for robotics research and learning purposes. In fact, at least one university (the University of Pennsylvania) has a major research undertaking based on the AIBO platform. It seems they wrote a Perl interpreter to make it easy to program the AIBO, although no information about it could be found yet on the Net.

Last year, there even was an AIBO soccer tournament, called "RoboCup2002," where competing teams of AIBOs had to play a game of soccer (see http://www.openr.org/robocup/index.html). The software to control the servos had to be written in C or C++. An example of the code needed to control the joints of the AIBO is shown in Listing 1 (available from http://www.tpj.com/source/).

Developing a Perl module that uses a bit of C++ to interface with the AIBO at the lowest level shouldn't be difficult. It should be possible to use the h2xs command to convert the header files to Perl extensions, being careful to not exclude external subroutines (that is, don't use the *–X* switch). Then, subroutines in the .pm could serve for the various robot functions, such as:

```
sub get_joint {

my ($joint_name) = @_;

… … … …
return $joint_number;
}
```

and could then be used in the Perl program.

Not much can be found on the Net when it comes to Perl and robotics. But once my geek curiosity had been awakened, I just had to walk that way all the way to the end. So, I started looking for Perl modules, software, and what have you to control robots. At first, one would assume Perl to be the perfect controlling environment for machinery. However, it seems that this is an area generally not very well covered by Perl. Part of that is certainly (sometimes, at least) due to real-time scheduling requirements.

In the CPAN you won't find anything either, but the need for a real servo/robot module is certainly there. The only real Perl system dealing with controllable motors is MisterHouse, the home automation system I use and described a few months ago in *The Perl Journal* (see http://misterhouse.sourceforge.net/).

MisterHouse would make a good robot-controlling framework, especially because a robot usually has a specific purpose and operates in a specific environment. Controlling the environment is easy with MisterHouse. It is already capable of doing things like these:

```
$fountain = new X10_Item 'B1';
  set $fountain ON if time_now '6:00 PM';

  $v_bedroom_curtain = new Voice_Cmd
                 '[open,close] the bedroom curtains';
  curtain('bedroom', $state) if $state = said $v_bedroom_curtain;
```

So, adding to MisterHouse all the necessary modules to be able to do something like this should be easy:

```
$movement_sensor = new Serial_Item 'AIB'O, 'door';
  play(file => 'intrusion.wav') if state_now $movement_
                                           sensor eq 'door';
```

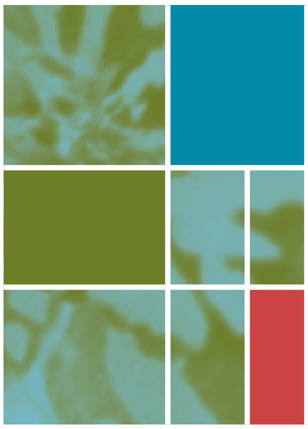
In other words, MisterHouse is the ideal controlling framework for robots, because all the environment data is already present. Additionally, MisterHouse is very extensible and makes it easy to drop in new modules.

I intend to start work on exactly this kind of robot module and will document its development in *TPJ* over the next couple of issues.

Stay tuned!

TPJ

Moshe is a systems administrator and operating-system researcher and has an M.Sc. and a Ph.D. in computer science. He can be contacted at moshe@moelabs.com.



Managing the House with Perl

Simon Cozens

These days, a lot of the code that I write for myself, out of work time, comes as a result of changes in my life situation. When I went to Japan for a month, I wrote some code that helped me maintain a diary and newsletter. Recently, I've moved house, and now have the joy of housemates again.

On top of everything else, this means all sorts of daily tasks require additional administration—bills need to be divided up, the house network needs better organization, there needs to be a shopping list for communally bought items, and so on. Being a lazy hacker, I shun additional administration and code around it. And since there are quite a few overly geeky houseshares around who might benefit from automating their admin, I took the time to write *HouseShare.pm*.

What It Does

HouseShare is a web-based administration system for a shared house. When it's completely finished, it will look after the network, the phone bill, the shopping list, and pass messages and information between housemates. At the moment, it does a reasonable chunk of those things.

When you first connect to HouseShare with your web browser, you'll see a menu like Figure 1.

Here, you see the front page for the Trinity House (that's my geekhouse) installation of HouseShare. At the bottom of the page are the latest blog entries—notes that all housemates should see.

Let's add a new computer to the house network by following the link on the computer icon. This presents us with a list of the currently configured computers, and prompts us for information about a new one. (See Figure 2.)

You'll notice that the system also suggests the next available IP address for us. Hosts on the network can be renamed, reconfigured, or deleted; changes to the network will be reflected in the DNS server, which is controlled by the whole HouseShare application.

HouseShare is a modular system and additional components can easily be added and updated. The phone bill and communal-shopping modules haven't yet been written, although they have been designed and I'll talk about their operation later on, but they will slot in with one additional database table and an additional Perl module each.

Simon is a freelance programmer and author, whose titles include Beginning Perl *(Wrox Press, 2000) and* Extending and Embedding Perl *(Manning Publications, 2002). He's the creator of over 30 CPAN modules and a former Parrot pumpking. Simon can be reached at simon@simon-cozens.org.*

How It Does It

The heart of the HouseShare system is a combination of two of the Perl modules I talk about most in these columns: *Class::DBI* and the Template Toolkit. As Kake Pugh points out (http://www.perl.com/pub/a/2003/07/15/nocode.html), these two modules are almost made for each other, allowing you to go straight from a database to HTML with very little Perl in the middle.

Most of the magic that runs HouseShare is done in the appropriately named *HouseShare::Magic* class. This is a subclass of *Class::DBI*, which all the HouseShare classes inherit from. Its job is to provide all the bridging code necessary to get from the database to HTML output.

One of the most important methods it provides, for instance, is *list*. The various listing pages for computers, users, and so on is all provided by this one *list* method in *HouseShare::Magic*. Even more interestingly, most of the pages are produced by exactly the same Template Toolkit template. This raises an obvious question: How does the template know whether or not it's dealing with a user, a computer, or something else?

Class::DBI helps with some of this, providing methods like *columns* which return a list of a database table's columns. If we tell the template the names of each table's columns, we can write code like this to turn a list of objects into a table:

```
[% FOR item = objects;
   "<tr>";
   FOR col = classmetadata.columns;
      NEXT IF col == "id";
      "<td>";item.$col;"</td>";
   END;
   button(item, "edit");
   button(item, "delete");
   "</tr>";
END %]
```

Another very useful piece of code is *UNIVERSAL::moniker*, which adds two methods to every class: *moniker* and *plural_moniker*. These methods transform a class name like *HouseShare::Computer* into *computer* and *computers*, respectively.

Now code like:

```
<h2> Listing of all [% classmetadata.plural %]</h2>
```

will say "Listing of all computers" and "listing of all users." If we have a class like *HouseShare::PhoneNumber*, which represents numbers that users have registered as having called recently, we can override the *moniker* and *plural_moniker* methods appropriately:

```
package HouseShare::PhoneNumber;
sub plural_moniker { "recently called phone numbers" }
sub moniker { "phone number" }
```

and the same template will still make sense.

HouseShare::Magic contains a do-everything templating method, *process*, which finds the templates, sets up the *Template* object, and creates a default set of arguments for it to use. The more interesting of these are *classmetadata*. We've already seen the use of *columns* and *plural*; here's the *classmetadata* argument in full:

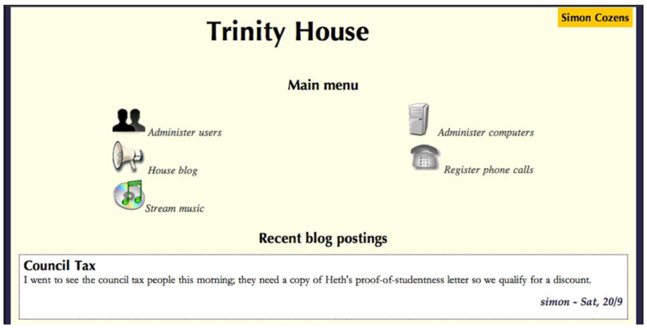
```
$args->{classmetadata} = {
   name => $class,
   columns => [ $class->columns ],
   colnames => { $class->column_names },
   moniker => $class->moniker,
   plural  => $class->plural_moniker,
   description => $class->description
};
```

Two methods in that metadata section may not be immediately recognizable: *description* and *column_names* are provided by *HouseShare::Magic* itself, and are supposed to be overridden in *child* classes. *column_names* maps a database table's columns to names that are sensible for display; the default implementation just uppercases the first character:

```
sub column_names {
   my $class = shift;
   map { $_ => ucfirst $_ } $class->columns
}
```

However, for some classes, you'll want to specify more human-readable column names. For instance, in the computer table, the column for the IP address is *ip*. With the default version of *column_names*, this will come out as "Ip," which is horrific. Instead, we provide a better mapping:

```
sub column_names {
   ip => "IP Address",
   hostname => "Hostname",
   owner => "Owner",
   comment => "Comment"
}
```


Simon Cozens

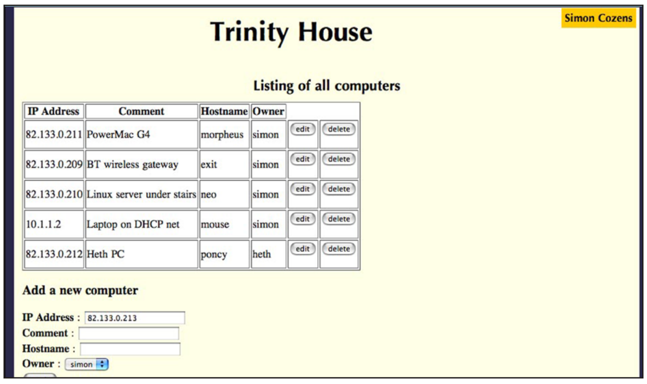
Trinity House

Main menu

Recent blog postings

Council Tax
I went to see the council tax people this morning; they need a copy of Heth's proof-of-studentness letter so we qualify for a discount.
simon - Sat, 20/9

Figure 1: HouseShare main interface screen.


Simon Cozens

Trinity House

Listing of all computers

IP Address	Comment	Hostname	Owner		
82.133.0.211	PowerMac G4	morpheus	simon	edit	delete
82.133.0.209	BT wireless gateway	exit	simon	edit	delete
82.133.0.210	Linux server under stairs	neo	simon	edit	delete
10.1.1.2	Laptop on DHCP net	mouse	simon	edit	delete
82.133.0.212	Heth PC	poncy	heth	edit	delete

Add a new computer

IP Address : 82.133.0.213
Comment :
Hostname :
Owner : simon

Figure 2: Adding a new computer to the house network.

Now our table can have a nice, friendly header:

```
<TR>
[% FOR col = classmetadata.columns.list;
   NEXT IF col == "id";
   "<TH>"; classmetadata.colnames.$col; "</TH>";
END %]
```

Similarly, *description* provides a human-readable description of what the class represents.

This more or less covers what *process* does, and everything else that spits out HTML is implemented in terms of that. For instance, the *list* method that produces the lists of things just looks like this:

```
sub list {
   my $class = shift;
   $class->process("list", { objects => [$class->retrieve_all]
});
}
```

This code looks for a template called "list," and passes as additional arguments to an array called *objects*, which are all the table's rows.

As we've seen with our list template, we then go through all the columns of this class's database table, and ask each object for its details. This works perfectly for things such as comments and IP addresses, but when I asked a computer for its *owner*, I was surprised to see that my computers had an owner of "1", rather than "simon."

This is because, in the database schema, the *owner* is stored as a foreign key into the *users* table. We've set up a *Class::DBI* has-a relationship to say that each computer has an owner, and therefore, quite correctly, calling *owner* on the *HouseShare::Computer* object that returns a *HouseShare::User*.

Unfortunately, this object stringifies to the ID, which is not what we want. (At least it doesn't stringify to *HouseShare::User=HASH(0xgarbage)*, which would be even less useful.) We want to display the actual username.

There are two ways we could do this. I did it first a good way, and this helped me to see a better way. The good way is to allow each class to override the default template. We do this in the magic template processing method by providing a series of template search paths:

```
my ($class, $name, $args) = @_;
my $template = Template->new({ INCLUDE_PATH => [
   File::Spec->catdir($HouseShare::templatehome, $class-
                                                  >moniker),
   File::Spec->catdir($HouseShare::templatehome, "custom"),
   File::Spec->catdir($HouseShare::templatehome, "factory")
]});
```

This means, if we call *HouseShare::Computer->list*, Template Toolkit will first look for templates in the /opt/houseshare/templates/computer directory, then in /opt/houseshare/templates/custom, (where installation-specific customizations can be made), and finally, in /opt/houseshare/templates/factory, where the factory settings are found. This allowed me to put code into templates/computer/list to fiddle with the *owner* column:

```
IF col == "owner"; item.owner.username; ELSE; item.$col;
END;
```

Now we can have our *HouseShare::Computer* class-specific templates in one location, out of the way. That was the good way.

The better way is to realize that *Class::DBI* is only trying to be helpful when it stringifies a *HouseShare::User* object to the ID, and it could easily be persuaded to stringify it to something more useful instead. So, putting the following code in *HouseShare::User*:

```
    __PACKAGE__->columns(Stringify => qw[ username ]);
```

solves the problem without having to mess with specific templates.

Editing Records

So much for displaying things. What about editing them? Well, there's the wonderful *Class::DBI::FromCGI* method, which turns a set of posted CGI form parameters into a *Class::DBI* object in your specified class, handling untainting via *CGI::Untaint*. That solves half the CGI problem, allowing you to create and update objects just by receiving form field values—the other half of the problem involves creating the CGI form in the first place. As it turns out, there's a nice, generic way we can do this, too.

HouseShare is a web-based administration system for a shared house

In the process of writing my HouseShare application, I found myself writing the *Class::DBI::AsForm* module. This provides a *to_cgi* method, returning hash mapping columns to HTML form elements.

If we feed this hash to our template too, we can create a generic form for adding entries to a database table like so:

```
<h3>Add a new [%classmetadata.moniker%]</h3>
<FORM METHOD="post">
  <INPUT TYPE="hidden" NAME="action"
                                VALUE="crate">
  <INPUT TYPE="hidden" NAME="class"
                   VALUE="[%classmetadata.name%]">
   [% FOR col = classmetadata.columns;
     NEXT IF col == "id";
     "<b>";classmetadata.colnames.$col;"</b> : ";
     classmetadata.cgi.$col;
     "<BR>";
     END;
  %]
   <INPUT TYPE="submit" NAME="create" VALUE=
                                      "create">
</FORM>
```

Editing objects is very similar: Just replace the relevant row in the list table with a set of calls to *to_field($col)* on the object. This produces an HTML snippet for the column in question, optionally taking notice of has-a relationships. For instance, when we edit a computer, at some point, our template will do the equivalent of

```
object.to_field("owner")
```

The *owner* of a computer is a *HouseShare::User*, and *to_field* knows this, so it produces a drop-down box of the users, with the current owner selected:

```
<select name="owner">
  <option value=1 selected> simon </option>
  <option value=2> heth </option>
  …
</select>
```

Hence, the add box we used to add a new computer to the network was generated completely generically, using a generic template and no special code in the computer class.

We've mentioned briefly the *FromCGI* module that is used to process these forms when the data is returned. Here's the code which does this, again in the generic *Magic* class:

```
sub do_edit {
  my $class = shift;
  my $r = Apache->request;
  my $obj = $class->retrieve(shift);
  my $h = CGI::Untaint->new(%{Apache::Request->new($r)
                                       ->parms});
  $obj->update_from_cgi($h);
  $class->list;
}
```

I've removed some of the error-checking code for the purposes of clarity: We'll be passed in an object ID by the front-end handler, and then *CGI::Untaint* reads and verifies the CGI form parameters. Sending this *CGI::Untaint* object to the *update_from_cgi* method, as provided by *Class::DBI::FromCGI* does the rest, and we direct the user back to the list page.

Identifying Users

What other information do we feed to our magical *process* method? You'll notice that at the top right-hand corner of our page, there was a little box with our name, demonstrating that the system had recognized the current user. This is done by passing in a *HouseShare::User* instance into the template arguments:

```
$args->{me} = HouseShare::User->me;
```

The *me* method tries to work out which of the housemates the remote user viewing the page actually is. How can we do that? Well, given that we know about all of their computers, and we can determine which IP address their browser is connected from, we can tell who owns the computer making the request. This is obviously a weak form of authentication, but in a house share situation where everyone has physical access to each other's kneecaps—sorry, I mean, computers—there's not much point in having any stronger authentication.

On the other hand, it is important to ensure that this request is actually coming from inside the house's network. The last thing you want is some random stranger out there on the Internet messing with your milk budget. To demonstrate the HouseShare system to the world at large, I added a demo mode which means that people can access and view the web site, but not change anything.

To work out who the user is, we start by knowing their IP address — information we get from the environment:

```
sub my_ip {
  return $ENV{'REMOTE_ADDR'} ||
  inet_ntoa(scalar gethostbyname(hostname() || "localhost"));
}
```

The first line checks the *REMOTE_ADDR* as set by the web server; the second line assures that this function will still work properly when *HouseShare* routines are called from the command line. As well as helping with debugging, we'll see later that this overcomes an interesting problem.

Now we can ask the main *HouseShare* module for the house's network and construct a *NetAddr::IP* representing the network range:

```
sub me {
  my $class = shift;

   my $net = NetAddr::IP->new(HouseShare->config->
                                     {network});
```

Now, if our IP address is not in this range, we switch to demo mode and don't return a user:

```
if (!$net->contains(NetAddr::IP->new($class->my_ip))) {
  $HouseShare::demo = 1;
  return;
}
```

and now we can see if we have a computer in the house registered to this IP:

```
if (my @computer = HouseShare::Computer->search({ip =>
$class->my_ip})) {
    return $computer[0]->owner;
}
```

The *owner* method will, quite properly, return a *HouseShare::User*, so our work is done.

Now comes the interesting problem. Suppose, you've just installed HouseShare, and you go to the web site and want to start adding computers and users. Unfortunately, the computer doesn't currently know who you are—and it can't look you up by computer, because you don't have any computers registered either! To bootstrap the system, the installation program prompts for the first user's information and creates a *HouseShare::User* object for them. From then on, any access from an unregistered IP address inside the network is assumed to be from this first "administrator" user:

```
my @users = $class->retrieve_all;
return $users[0];
```

And that, basically, is the *HouseShare::User* class.

The Front End

Finally, in our tour of the *HouseShare* classes, let's look at the front-end *mod_perl* handler, which ties the whole system together.

Here's the entire handler:

```
sub handler {
    my $r = shift;
    my @pi = split /\//, $r->uri();
    shift @pi while @pi and !$pi[0];
    @pi = qw(user process frontpage) unless @pi;
    my $class = "HouseShare::".ucfirst(shift(@pi));
    my $method = shift @pi || "present";
        return DECLINED if !$class->require || !$class
                                              ->can($method);
    $r->send_http_header("text/html");
    $class->$method(@pi);
    return OK;
}
```

I will admit that this code is a little insecure, although it's not easy to see how to exploit it—you'd have to find a dangerous class method in a *HouseShare* class. This is not the way I would recommend you code, but it was a neat hack. The idea is that the URL http://houseshare.my.house/computer/edit/5 gets turned into *HouseShare::Computer->edit(5);*.

If there isn't a method, we call the generic method *present*; this means that things like the *HouseShare::Blog* class (based on Bryar, the subject of some of my previous articles) can be accessed from http://houseshare.my.house/blog.

And if there isn't even a class, such as when we hit the front page, we're effectively sent to *HouseShare::User->present("frontpage")*, which displays the *frontpage* template. (The choice of the user class to do this is somewhat arbitrary.)

One thing you might notice about that handler routine is that it's simple and compact, a theme that runs through the whole system—in fact, the system currently weighs in at only 250 lines of actual Perl code, and 300 lines in the templates. All the heavy lifting is either done with existing modules or abstracting tasks away to a generic layer, such as the *Magic* class.

Doing Real Work

So far, we've discussed a lot of infrastructure — a framework for doing neat things with databases and the Web. However, it's now very trivial to build on top of this framework to add "real work" functionality to HouseShare.

For instance, one piece of real work we can do with *HouseShare::Computer* is to update the DNS tables when a *Computer* object is added or modified. Thankfully, with *Class::DBI*'s trigger support, this is a very simple matter. Assuming we have a subroutine called *build_zonefile* which does the equivalent of:

```
print $_->hostname, " IN A ", $_->ip, "\n"
    for HouseShare::Computer->retrieve_all;
```

(except, naturally, with a little more smarts…) we can trivially arrange for this to be called when anything changes:

The last thing you want is some random stranger out there on the Internet messing with your milk budget

```
__PACKAGE__->add_trigger(after_update => \&build_zone
                                                file);
__PACKAGE__->add_trigger(after_create => \&build_zone
                                                file);
```

Now new hosts will automatically be added to the DNS server, and updates will automatically be reflected—as usual, with only a tiny bit of code.

What It Will Do Soon

HouseShare currently does around 50 percent of what I would like to see it do. It was very simple to plug in *HouseShare::Blog* as a 20-line subclass of Bryar to add a blog to the site, and an *Apache::MP3* instance to share music around the house; I also plan to add a *CGI::Wiki* wiki to share information about who to call when the power fails, and so on.

The next big step will be integrating Tony Bowden's *Data::BT::PhoneBill* to download and parse phone-bill data. The methodology here is similar to what we've seen so far: A *Phonenumbers* class and database table will register known numbers and associate them with people who are likely to have called them, and then a method on that class will grab the data, share out the cost, and process a template displaying the results.

The final piece (for now) will be something to record purchases of house essentials and share the cost between the housemates; again, this will be a simple class with a database table, and a class method to do the work. This simple approach can be used to extended the system to add all kinds of functionality—in fact, the framework we've drawn up can be used in a huge number of applications, and we use a (admittedly, more complex) variant of the idea at work as a basis for all kinds of e-commerce sites.

HouseShare is available from my CVS repository at http://cvs.simon-cozens.org/viewcvs.cgi/?cvsroot=HouseShare; it's currently a little underdocumented but e-mail me if you want to install it and hack on it and I'll help you through it.

I sometimes think that HouseShare is a little bit overkill for the job it does; it's currently slightly more useful than a whiteboard in the hall. But it's been a lot of fun, which is the main thing, and in the course of writing it, I've learned a lot about *Class::DBI*, Template Toolkit, abstracting functionality away, and making generic templates do a wide range of tasks. I hope through reading this, you have, too.

TPJ

Web Error Reporting

Randal Schwartz

Recently, a discussion occurred on the Perl CGI beginners mailing list about using *CGI::Carp*, presumably in the *fatalsToBrowser* configuration. This core module is a very nice tool for developing Perl CGI applications, as it can be used to redirect errors or at least make them safe to display in a web browser. For example, to make all of your fatal errors turn into a message to your browser instead, you simply add:

```
use CGI::Carp qw(fatalsToBrowser);
```

to the beginning of your program. Even syntax errors occurring later in the same file will show up in the browser, which is especially nice if you're using a shared hosting environment and have difficult or no access to the web server's error logs.

But as great as such setup is for development, it is exactly the wrong thing to do for a web site that will be accessed by the general public. An error message displayed to the browser of a random visitor is generally frowned upon (no matter how infrequently it might appear).

Worse yet, visitors are unlikely to report the problem, or if they do, their report will likely be paraphrased or incomplete, making diagnosis difficult. (Can you tell that I've had experience working in help desks?)

But the biggest problem is the potential leak of security information. I've encountered many ill-designed web sites that (upon the failure of some necessary component) showed me, in intricate detail, the pathnames being used, operating system details, values of internal program variables, and even the SQL and database table names for a query! This is better than dumpster-diving as a means of finding a method of intrusion, because the potential intruder can immediately vary the query and see how it affects the error messages.

But what else should we do? If we don't have any error trapping, we give the user the unpleasant "500 error" experience. If we trap the error, where do we write it?

One solution that has occurred to me is to log the error (even to the standard web server error log), but also generate a virtual "trouble ticket" that will be shown to the user, with a message that is essentially, "Something went wrong, we know about it, and here's our trouble ticket number if you want to reference this problem." A simple *$SIG{__DIE__}* or *eval* handler will take care of catching the error, and then a bit of massaging of the error text will ensure a clean splatting into the error log.

Randal is a coauthor of Programming Perl, Learning Perl, Learning Perl for Win32 Systems, *and* Effective Perl Programming, *as well as a founding board member of the Perl Mongers (perl.org). Randal can be reached at merlyn@stonehenge.com.*

But we can even take this one step further. Instead of simply telling the user that an error has occurred and has been logged, let's give the user a text form to describe what they were trying to do. Using a hidden field, we can integrate the incident number into the form submission to correlate their description of the problem with our known error messages, which we log into a separate file. Later, we can create trouble tickets by correlating the log files using the ticket numbers. We can also give the user the option of including a contact e-mail address or phone number for further information, and log that as well.

The CGI script in Listing 1 has been modified to trap its errors in such a fashion. Although the code is included explicitly in this script in the first *BEGIN* block, you'd generally want to factor that out into a site-wide module for consistency and ease of upgrading, starting every local CGI script with something like:

```
use lib "/my/site/lib";
use MySite::ErrorWrapper;
```

But for simplicity, let's look at the code in place.

Lines 1 through 3 start nearly every CGI program I write, turning on the normal compiler restrictions, taint checking, warnings, and unbuffering *STDOUT*.

Lines 5 through 44 define the *BEGIN* block that will set up our handler. Because it's in a *BEGIN* block, even syntax errors occurring later in the file are handled properly.

Lines 6 through 43 define a *die* handler for all untrapped fatal errors. The text of the error message will be passed to this subroutine as its first parameter.

Line 7 creates an incident number. Here, I'm taking the current epoch time (integer seconds since 1-Jan-1970 midnight UTC) and the process ID, and concatenating them into a 12-character hex string. This is reasonably unique per host (unless you can cycle 30,000 processes in one second), but might collide if several web servers are working in a load-balancing situation. You might consider the *File::CounterFile* module from the CPAN for generating monotonically increasing identifiers, or perhaps a real database connection using *DBI* to get the next incident report number.

Line 9 takes the incoming error message text from the argument list. Line 12 logs this message to the web server error log, ensuring that every line of the error message is prefixed by the incident number in a relatively distinct pattern for later extraction and processing. For example, if *die "one\ntwo\nthree\n"* is triggered, the resulting error messages might look like:

```
crash 3f71df085c4f: one
crash 3f71df085c4f: two
crash 3f71df085c4f: three
```

with the hex string naturally containing the timestamp and process ID.

Lines 15 to 41 present the user with a simple form, consisting of a text area for the description of the problem, a text field for the e-mail address, and a submit button. In addition, a hidden text field includes the incident number so that we may later correlate this submission with the trigger.

Line 42 causes the CGI program to successfully exit after having delivered the error form. This keeps the script from generating a "500 error."

The biggest problem is the potential leak of security information

The URL for the form has to point to a second script to capture the submitted information. A sample implementation for this script is given in Listing 2. But we'll get to that in a minute.

Following the *BEGIN* block, we continue the rest of our program as we normally would. As a sample program for testing, I included some text that had a syntax error (line 49)—the beginning of a regular expression without the matching end.

The error reporting program in Listing 2 should be as simple as possible. In this case, I'm pulling in *CGI.pm* to decode the parameters and generate the response HTML. Again, lines 1 through 3 are like the previous program, and line 5 pulls in *CGI.pm*.

Lines 7 to 14 use the *CGI.pm* shortcut methods to generate a simple response using small soothing words.

Line 18 opens up an error log for appending. I put the error log here in /tmp for simplicity during my coding, but I'd probably move this to a more secure location for production, typically alongside the web error logs in the same directory.

Again, in the "keep it simple" motif, I wanted to store all the environment variables in the response, in case the user agent or source IP address or referrer page might be useful. I'm doing this by using the *CGI.pm*'s *param* function to pretend that additional parameters were passed as the form values. Each param is named like the Perl-style access to that environment variable. For example, a param name of *$ENV{HOME}* stores the value for the *HOME* variable.

Finally, in line 20, the contents of the form fields are dumped in *CGI.pm*'s dump format into the log file. By using this dump format, we'll be able to restore the data very cleanly, regardless of how many odd characters were included in the form submission or environment.

Note that it might be a good idea to *flock* this filehandle before writing to it, but as long as the writes are under 8 KB or so, we probably won't have any problems.

Once we've set up all of our CGI scripts with the *BEGIN* block from Listing 1, and the common CGI crash-reporting script in Listing 2, we're ready to watch for errors. As the errors occur, they'll be logged into the web server error log, and the user will be shown the simple form. If the user fills out and submits the form, the resulting form values (including our added *%ENV* values) are placed into the separate crash reporting log.

All we need to do now is extract the values from the crash reporting log and correlate those with the web server's error log. Luckily, *CGI.pm* can do most of the hard work for us again. In Listing 3, I have a short snippet of a program to extract the data.

Please note that this is not a CGI script and will not be placed into the CGI directories of my web server, even though it contains *use CGI* in line 3. I'm pulling in the CGI module only to parse the CGI dump-format file. Line 5 defines the location of that file, which is opened in line 9.

Line 7 defines a top-level data structure that will hold all of the entries in the file, so that I can dump them in lines 20 and 21.

Lines 10 through 18 process through the file by having the *CGI* module construct new CGI objects from the file until the file is at end-of-file (tested in line 10).

Line 11 assigns the newly constructed CGI object (pulled from the file in the CGI dump format) into the special *$CGI::Q* variable. This variable is the default CGI object for objectless *CGI* calls, so that I can just say *param()* later instead of *$CGI::Q–>param()*.

Line 12 creates a hash that will hold all of the params of this particular form submission.

Lines 13 to 16 go through each param, pulling out its values into *@p* in line 14. If there are fewer than two values, then *$p[0]* is used as the final value.

Line 17 pushes a reference to this param hash onto the *@all* data structure.

Lines 20 and 21 use *Data::Dumper* to show us what the various error reports would have looked like after parsing. Had this been an actual code, the incident values of each report would be keyed into some database correlated with incident values from the web server error log. But that's beyond the scope of what I wanted to show in these examples.

So, there you have it. Don't show error messages to your web users. Put them someplace where you can get to them, but let the user annotate with their own comments to help you diagnose your problem. And as always, until next time, have the appropriate amount of fun!

TPJ

(Listings are also available online at http://www.tpj.com/source/.)

Listing 1

```
#!/usr/bin/perl -Tw
use strict;
$|++;

BEGIN {
  $SIG{__DIE__} = sub {
    my $incident = unpack "H*", pack "NS", time, $$;

    my $message = shift;

    ## record reason for death into the error log
    warn join "", map "crash $incident: $_\n", split /\n+/, $message;

    ## now give the user a place to record their experience
    print <<END_OF_HTML;
Content-type: text/html

<hr>
<h1>An error has occurred</h1>
<p>An error has occurred. Relevant details have already been
logged for us, but if you want to tell us what you were doing at
the time, it might help us determine the cause more accurately
to fix the problem more easily.</p>
<form action="/cgi/crash-reporter">
<input type="hidden" name="incident" value="$incident">
<table>
<tr><th>
  Email address (optional):
</th><td>
  <input type="text" name="email" value="foo\@example.com" size="60"/>
</td></tr>
<tr><th>
  Comments:
</th><td><textarea name="comments" rows="10" cols="60">
Describe your circumstances here.  What were you trying to do?
</textarea></td></tr></table>
<input type="submit" />
</form>
<p>Or, you can contact us at 1-800-555-5555 and reference incident
number $incident.</p>
<hr />
END_OF_HTML
    exit 0;
  };
}

use CGI qw(:all);
print start_header, start_html("Buggy Program");

/ bad syntax

print end_html;
```

Listing 2

```
#!/usr/bin/perl -Tw
use strict;
$|++;

use CGI qw(:all);

print
  header,
  start_html("Incident Reporter"),
  h1("Thank you!"),
  p("Thank you for your incident report.",
    "The appropriate authorities have been notified.",
    "Remain calm."),
  end_html;

## save the parameters and environment

open ERRLOG, ">>/tmp/crash-reporter.log";
param("\$ENV{$_}", $ENV{$_}) for keys %ENV;
$CGI::Q->save(\*ERRLOG);
```

Listing 3

```
#!/usr/bin/perl -w

use CGI qw(param);

my $FILE = "/tmp/crash-reporter.log";

my @all;

open FILE, $FILE or die;
until (eof FILE) {
  $CGI::Q = CGI->new(\*FILE);
  my %p;
  for (param) {
    my @p = param($_);
    $p{$_} = @p < 2 ? $p[0] : \@p;
  }
  push @all, \%p;
}

use Data::Dumper;
print Dumper \@all;
```

TPJ

Practical mod_perl

Eugene Eric Kim

Practical mod_perl is simultaneously an apt and misleading title. On one hand, the book is a massive collection of practical knowledge regarding administering and programming mod_perl. On the other hand, its scope extends well beyond mod_perl, delving into aspects of web server performance and maintenance, Apache configuration, and Perl peculiarities. This is largely by necessity, but at times, the authors go overboard in their digressions. The result is a book that is full of outstanding and useful information, but that is also too large and not as organized as it could have been.

Stas Bekman and Eric Cholet's book is markedly different from O'Reilly & Associates's previous foray into mod_perl documentation—Lincoln Stein and Doug MacEachern's *Writing Apache Modules with Perl and C* (O'Reilly & Associates, 1999). The latter focused mostly on documenting the Apache 1.x API, and introduced some basic information about mod_perl programming. Bekman and Cholet focus their energies on the practical issues that arise when using mod_perl, and there are many, many, many of these.

Unlike CGI, which provides a black box where even poorly written programs will run without problems, mod_perl is closely coupled with the Apache web server. Without awareness of how Apache works, mod_perl programmers are bound to run into trouble.

For example, Apache processes—and by extension, mod_perl processes—are persistent. This enables you to do many things not possible with CGI: preload precompiled code, maintain persistent connections to databases, and so forth. It also means that you have to be much more diligent about your Perl code. You have to know what will happen with your global variables, and how to initialize them properly. You have to worry about closing your filehandles and destroying your objects. That's only the beginning.

Scope and Organization

Only one chapter out of 25 (namely, Chapter 6) focuses exclusively on programming mod_perl. The vast majority of the book is devoted to configuration and performance. Again, this is largely by necessity. Like many open-source projects, mod_perl is poorly documented, and much of the information in this book is not available anywhere else. Practical mod_perl also claims to be the first book to cover mod_perl 2.0, which is still in beta and whose API continues to evolve. This information alone makes the book valuable, as mod_perl 2.0 is essentially undocumented.

Unfortunately, the authors go into too much detail in certain areas, and not enough in others, and the tradeoff is significant. For example, Part 2 discusses mod_perl performance, and contains a tremendous amount of useful knowledge, such as how to configure a mod_perl-enabled web server with an http proxy. However, one of its chapters (Chapter 8) delves into choosing the right platform and includes a substantial section on hardware. Not only was much of this information gratuitous, some of its advice was questionable.

Part 4 focuses on debugging, which would have been a good (and much thinner) book in and of itself. Once again, there was much useful information (for example, running httpd in single-process mode, the *Devel::ptkdb* module, and so on), but there was also a section on debugging using print statements, which was completely unnecessary.

Contrast this with the first part of Chapter 6, which describes some unusual behavior in an innocent-looking mod_perl application. To make a long story short, a subroutine that looks like it is referring to a lexical variable defined outside of its scope is actually a closure, because of the way mod_perl loads applications. That's a mouthful, and it requires a more advanced understanding of how Perl handles scoping. You could argue that this topic is beyond the scope of this book, but I would have found a more extensive chapter on scoping issues more valuable than a section on debugging using print statements.

Practical mod_perl
Stas Bekman and Eric Cholet
O'Reilly & Associates, 2003
$49.95
ISBN 0-596-00227-0

Bigger Is Still Better

I think that *Practical mod_perl* could lose a few hundred pages and be a better (and cheaper) book. That said, the authors faced a difficult balancing act, and in the end, I'm glad that they erred on the side of too much information. It's worth reading the book front-to-back, skimming through some of the less useful parts, because the accumulated knowledge in the book is broad, and you may find tips where you least expect it. For instance, prior to reading this book, I did not know that mod_perl could be used to configure Apache, a technique that I now plan to use on some of my projects. Ultimately, most mod_perl programmers will find *Practical mod_perl* valuable.

TPJ

Eugene is the founder and executive director of Blue Oxen Associates, a think tank devoted to improving collaboration and knowledge management. He can be reached at eekim@blueoxen.org.

Source Code Appendix

Randal Schwartz "Web Error Reporting"

Listing 1

```
#!/usr/bin/perl -Tw
use strict;
$|++;

BEGIN {
  $SIG{__DIE__} = sub {
    my $incident = unpack "H*", pack "NS", time, $$;

    my $message = shift;

    ## record reason for death into the error log
    warn join "", map "crash $incident: $_\n", split /\n+/, $message;

    ## now give the user a place to record their experience
    print <<END_OF_HTML;
Content-type: text/html

<hr>
<h1>An error has occurred</h1>
<p>An error has occurred. Relevant details have already been
logged for us, but if you want to tell us what you were doing at
the time, it might help us determine the cause more accurately
 to fix the problem more easily.</p>
<form action="/cgi/crash-reporter">
<input type="hidden" name="incident" value="$incident">
<table>
<tr><th>
  Email address (optional):
</th><td>
  <input type="text" name="email" value="foo\@example.com" size="60"/>


</td></tr>
<tr><th>

  Comments:
</th><td><textarea name="comments" rows="10" cols="60">
Describe your circumstances here.  What were you trying to do?
</textarea></td></tr></table>
<input type="submit" />
</form>
<p>Or, you can contact us at 1-800-555-5555 and reference incident
number $incident.</p>
<hr />
END_OF_HTML
    exit 0;
  };
}

use CGI qw(:all);
print start_header, start_html("Buggy Program");

/ bad syntax

print end_html;
```

Listing 2

```
#!/usr/bin/perl -Tw
use strict;
$|++;

use CGI qw(:all);

print
  header,
  start_html("Incident Reporter"),
  h1("Thank you!"),
  p("Thank you for your incident report.",
    "The appropriate authorities have been notified.",
    "Remain calm."),
  end_html;

## save the parameters and environment

open ERRLOG, ">>/tmp/crash-reporter.log";
param("\$ENV{$_}", $ENV{$_}) for keys %ENV;
$CGI::Q->save(\*ERRLOG);
```

Listing 3

```
#!/usr/bin/perl -w

use CGI qw(param);

my $FILE = "/tmp/crash-reporter.log";

my @all;

open FILE, $FILE or die;
until (eof FILE) {
  $CGI::Q = CGI->new(\*FILE);
  my %p;
  for (param) {
    my @p = param($_);
    $p{$_} = @p < 2 ? $p[0] : \@p;
  }
  push @all, \%p;
}

use Data::Dumper;
print Dumper \@all;
```

TPJ