;login:
THE MAGAZINE OF USENIX & SAGE
August 2003 • volume 28 • number 4

inside:

PROGRAMMING

Turoff: Practical Perl: Cleaning Up with Modules

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

practical perl
Cleaning Up with Modules

by Adam Turoff

Adam is a consultant who specializes in using Perl to manage big data. He is a long-time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

ziggy@panix.com

A friend of mine who occasionally writes some Perl was working on a CGI program to automate data entry into a database. He is not a programmer by trade, but he uses Perl to manage information critical to his job. Programming is a hobby for him, more intellectually stimulating than trainspotting and less important to him than his true calling, studying the English language.

My friend asked me to look at some of his CGI programs. His programs worked, but he felt they were "ugly." A Java programmer colleague of his recommended that he clean them up by creating a series of Java-like objects and classes. A better way to clean up a Perl program is through a mixture of objects, classes, and standard Perl data structures.

Every so often, we all write programs that just "feel wrong." A good indication is when it takes too much effort to do something that should be easy to do. Another indication is when some Perl feature is used in what feels like an "unclean" fashion. Such misuses can increase the cost of maintaining and extending a Perl program over time, and can lead to programs that work yet are difficult to fathom.

Creating Dynamic Web Pages

Perl is a great language for casual programmers who write code occasionally in service of a greater goal. These people often use Perl to write small a CGI program that automates some task. There are very many ways to write Web-based applications, including using a templating system, building XML-based systems, and using the venerable CGI.pm module. Using CGI.pm to generate dynamic HTML pages may not be the best system or the most elegant mechanism, but it works well and is quite easy to use.

Using CGI.pm can have its downsides, though. Its HTML-generation interface is an awkward way to replace HTML syntax with Perl syntax to create static HTML. This technique obscures the content of the document being created by hiding it within nested Perl subroutine calls:

```
#!/usr/bin/perl -Tw
use strict;
use CGI qw(:standard);
print header('text/html');
print start_html("Test Page");
print h1("Test Page");
print table(Tr(td("a"), td("b")), Tr(td("c"), td("d")));
print end_html();
```

A better use of CGI.pm's HTML-generation interface is to focus on wrapping repetitive Perl data inside HTML tags through simple loops. In this fashion, data can be abstracted into one of many places: a configuration file, a module, or a database. The structure of the HTML to be created is less likely to change frequently, so separating the content (data) from the presentation (HTML-generating code) will tend to simplify the program in the long run. After all, it is easier to update a database or change a configuration file than it is to modify and test a program for every little data fix.

This is the approach my friend used with his code. The goal in the following code samples is to create a simple HTML form with a series of fields. The fields are specified in the @fields array, and the code to generate the form simply iterates over all of the fields to be displayed. Each element in the @fields array is a hash reference that defines all of the aspects of an HTML form field to be displayed.

```
#!/usr/bin/perl -Tw
use strict;
use CGI qw(:standard);

my @fields = (
    {
      -name => "name",
      -label=> "Name",
      -size=>50,
      -maxlength=>50,
      -procedure=>\&CGI::textfield,
    },
    {
      -name => "subscribe",
      -label=> "Subscribe",
      -values=>[qw(Yes No)],
      -procedure=>\&CGI::radio_group,
    },
    {
      -name => "availability",
      -label=>"Availability",
      -values=> [qw(Mon Tue Wed Thu Fri)],
      -procedure=>\&CGI::checkbox_group,
    },
    {
      -name => "state",
      -label=>"State",
      -values=> [undef, qw(Maryland DC Virginia)],
```

```
        -procedure=>\&popup_menu,
    },
    …
);
…
```

For added simplicity, each field definition within @fields contains a reference to the CGI.pm function that will display that field. This has the effect of making the @fields array a dispatch table as well as a rudimentary data dictionary. The hash keys in the element definition are taken from the keys used by the CGI.pm HTML-generation subs that create each kind of input field. By combining these two features, creating the HTML form becomes very easy:

```
…
my @rows;
foreach my $row (@fields) {
    my $sub = $row->{'-procedure'};
    push (@rows, Tr(td($row->{-label}), td($sub->(%$row))));
}
print start_form(), table(@rows);
print submit(), reset(), end_form();
…
```

Designing with Modules

After working with this program for a while, my friend felt that the code was starting to get ugly. His colleague, a Java programmer, recommended creating a class called a FieldList object to represent a list of HTML form fields, represented by Field objects. Processing a FieldList would involve creating a FieldListIterator object to examine the elements one by one.

This is an area where Perl programmers and Java programmers have differing opinions. Perl programmers generally feel that creating artificial container objects like FieldList and FieldListIterator are a waste of time. Perl's array variables and the foreach loop exist to hold lists of values and examine them iteratively, thus obviating artificial container and iterator classes.

There is some wisdom in creating a Field class, or rather a hierarchy of field classes to handle different kinds of HTML form fields. The field definitions listed above in @fields are somewhat repetitive. Each field will contain very similar values for the -label and -name keys. The -label key specifies the text to appear to the left of a form field, and the -name key specifies the name of the form variable to appear in the HTML output. Note that there's a very simple relationship between the -name and -label values: In our sample program, the -name value can be derived from the -label value by lowercasing it and removing space characters.

There are other common behaviors shared by all field objects. In this program, all fields are displayed within a two-element HTML table row, with the text label appearing in the left cell and the HTML form input field appearing in the right cell. And then there are differences between individual field types. Setting the maxlength makes sense only with text fields.

A better way to structure this code would be to create a simple Field package (or class) that handles the common behaviors and then create specialized packages that describe the actual differences between text fields, radio groups, checkboxes, and other HTML form input fields. Here's the Field class that describes the common behaviors of HTML form input fields for this application. Note that the init will create a default CGI variable name from a field's label, a technique that reduces redundancy found in the older version of the program.

```
#!/usr/bin/perl -w
use strict;
package Field;
use CGI qw(:standard);
sub init {
    my $self = shift;
    my %params = @_;

    ## Assign the key/value pairs for this object
    while(my ($key, $value) = each %params) {
        $self->{$key} = $value;
    }
    ## Create the field name from the text label
    my $name = "\L$self->{-label}";
    $name =~ s/ /_/g;
    $self->{-name} = $name;
    return $self;
}
sub display_row {
    my $self = shift;
    my $sub  = $self->{-procedure};

    return Tr(td($self->{-label}), td($sub->(%$self)));
}
```

Creating a package to describe text fields is not difficult. The first step is to declare that this text field package inherits from the Field package (via a use base; declaration). This package needs a constructor (a method named new) that creates new text field objects with a minimal amount of information specified by the user. The only information that is absolutely necessary is the label for this input field. The CGI field name for this input field can be derived using the Field::init method. Attributes like the field size and the maximum input length can be defaulted to sensible values. Finally, all text fields will use the CGI::textfield subroutine to display themselves.

Here is the complete package definition this application needs to create CGI text input fields. The new method receives the label for a text field and uses sensible default values for the rest of the field's attributes. The size and maxlength methods exist to override the default values for the -size and -maxlength attributes.

```
package Field::Text;
use base 'Field';
use CGI;
sub new {
    my $class = shift;
    my $label = shift;
    ## Create an object
    my $self = bless {}, $class;
    ## Finish initialization
    $self->init(-label => $label,
        -size               => 50,
        -maxlength          => 50,
        -procedure          => \&CGI::textfield);
}
sub size {
    my $self = shift;
    $self->{-size} = shift;
}
sub maxlength {
    my $self = shift;
    $self->{-maxlength} = shift;
}
```

The other input field types (checkbox groups, pop-up menus, radio groups) are all similar in that they have multiple values. A good way to describe these similarities is to define a Field::Group package and derive specializations of that package for the specific input field types.

```
package Field::Group;
use base 'Field';

sub init_group {
    my $self = shift;
    my $procedure = shift;
    my $label = shift;
    my @values = @_;
    $self->init(-label => $label,
        -procedure          => $procedure,
        -values             => \@values);
}
```

The last few packages handle creation of radio groups, checkbox groups and pop-up menus. They are all very similar; only the procedure to display them varies:

```
package Field::RadioGroup;
use base 'Field::Group';
use CGI;

sub new {
    my $class = shift;

    my $self = bless {}, $class;
    $self->init_group(\&CGI::radio_group, @_);
}
package Field::CheckboxGroup;
use base 'Field::Group';
use CGI;

sub new {
    my $class = shift;

    my $self = bless {}, $class;
    $self->init_group(\&CGI::checkbox_group, @_);
}
package Field::PopupMenu;
use base 'Field::Group';
use CGI;

sub new {
    my $class = shift;

    my $self = bless {}, $class;
    $self->init_group(\&CGI::popup_menu, @_);
}
```

Cleaning Up with Modules

Now that we have a set of packages for creating CGI input fields, it's time to revisit the application. Recall that the first step was to create a list of field definitions, then create the HTML form. With the Field modules created for this application, it is easy to simply and succinctly declare what fields are used in this CGI application. There is no repetition in defining -label and -name attributes, and the common attributes of each type of field are defined once and only once.

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

use strict;
use Field;        ## pull in all the Field packages
use CGI qw(:standard);

my @fields = (
    new Field::Text('Name'),
    new Field::RadioGroup('Subscribe', 'Yes', 'No'),
    new Field::CheckboxGroup('Availability', 'Mon',
            'Tue', 'Wed', 'Thu', 'Fri'),
    new Field::PopupMenu('State', undef,
        'Maryland', 'DC', 'Virginia'),
);
...
my @rows;
foreach my $field (@fields) {
    push (@rows, $field->display_row());
}
print start_form(), table(@rows);
print submit(), reset(), end_form();
...
```

Conclusion

Over time, all software has a tendency to be extended beyond its original design and "get ugly." One technique for improving code that "feels wrong" is to restructure it and extract common behaviors into modules. In this example, the code for constructing CGI form fields was abstracted into a Field package and its subpackages. The main CGI program was simplified and now focuses on *what* fields need to be created, not the arcane details of *how* to create those fields.