practical perl

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

Using Object Factories

In my last column, I demonstrated how to clean up a CGI form-building program by refactoring it and using a hierarchy of modules. Each module inherited from a common application-specific Field module and implemented specialized behaviors to produce a specific kind of CGI form field. This time, I revisit the same problem and examine a different solution – object factories and factory methods.

One of the best features of Perl is the principle of TMTOWTDI: "There's More Than One Way to Do It." Even if you have only a passing familiarity with Perl, you can use it to automate a tedious task or write a small program to get your job done. You can approach Perl as a shell programmer, or as a C or Java programmer, and still get your job done.

However, Perl is a rich language unlike any other. While you are free to use idioms from shell, C, or Java to accomplish your task, using Perl idioms can help you do it faster and with less effort. My last column took a Java-flavored approach to cleaning up a CGI program. In this column, I'll look at a more Perl-flavored approach that's easier to write, maintain, and extend.

Many Ways to Do It

Consider the problem from the last column: a CGI program that needs to create an HTML form. The simple and straightforward approach is to use the CGI.pm HTML-building functions to build a Web page one piece at a time:

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

use strict;
use CGI qw(:standard);

print header('text/html');
print start_html("Test Page");

print start_form();
print popup_menu(...);
...
print submit(), reset();
print end_form();

print end_html();
```

While that approach is easy to write and easy to understand, it is also awkward and cumbersome. It is a simple translation of HTML syntax into Perl statements for building a static Web page. Modifying and debugging this program will be more difficult than necessary – programmers will need to keep a mental model of the HTML page in mind while modifying code that uses Perl syntax. Adding dynamic features to selectively display some components will turn this simple program into something complex very quickly.

The above fragment deals with two primary tasks: building the Web page, and building the form components. In my experience, the first part of this program is static and unchanging, while the second part is more dynamic. Therefore, the program can be simplified by separating the static HTML-building parts from the more dynamic form-building parts.

One way to simplify the form-building part of this program is to describe an HTML form with a list of hashes. Each hash in this list represents a single form field. Building an HTML form involves processing these field descriptions and converting them into HTML form fields as necessary. The full Web page is then created by combining the static HTML elements with these dynamically generated form fields. A program written this way might look something like this:

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

my @fields = (
    {
       -name => "name",
       -label=> "Name",
       -size=>50,
       -maxlength=>50,
       -procedure=>\&CGI::textfield,
    },
    ## more fields here
);

...
my @rows;
foreach my $row (@fields) {
        my $sub = $row->{'-procedure'};
        push (@rows, Tr(td($row->{-label}),
                td($sub->(%$row))));
}
```

```
print start_form(),
        table(@rows),
        submit(), reset(),
        end_form();
...
```

While this approach is a step forward, it does have problems. The format for the field descriptions found in the @fields array are undocumented. They are values that will be passed to a CGI.pm function like CGI::textfield, but that knowledge is hidden dozens or hundreds of lines away in the body of the foreach loop.

This approach also leads to a lot of duplicated information. The -name and -label components contain similar values. Instead, one could easily be derived from the other, reducing an opportunity for bugs to creep in.

Ideally, these field objects should be simple to create and use. One way to do that is to create a group of Field modules to ease the process of defining fields to build an HTML form. Here is an example of what that might look like, taken from the last column:

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

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

my @fields = (
     new Field::Text('Name'),
     ## more fields here
);

...
my @rows;
foreach my $field (@fields) {
     push (@rows, $field->display_row());
}

print start_form(),
        table(@rows),
        submit(), reset(),
        end_form();
...
```

In this example, the interface for building a Web page is much cleaner. Creating the @fields list is done by creating a series of objects that are defined by the Field module. Each object constructor uses sensible defaults and requires a minimal amount of information. Later on, the dynamic HTML form field generation is accomplished by calling the display_row method on each field object in turn.

The Problem with Inheritance

The interface provided by the Field::* modules certainly simplifies the job of creating a dynamic Web form. It works by using a core Field class and subclasses like Field::Text to construct specific field types:

```
package Field;
use strict;

use CGI qw(:standard);

sub init_field {
     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)));
}

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_field(-label     => $label,
                       -size      => 50,
                       -maxlength => 50,
                       -procedure => \&CGI::textfield);
}
```

Field types that display multiple values share similar behaviors. The Field::Group module helps to define field types like radio groups and checkbox groups:

```
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);
}

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, @_);
}
```

Although this module hierarchy does aid in creating dynamic HTML forms, it has a Java-flavored design that leads to overly complex Perl code. In order to define three types of fields, five classes are required in a hierarchy that is three levels deep.

Extending this library isn't difficult, but it is cumbersome. Each HTML form field type requires a new class definition. Each class definition contains a package declaration, a use base declaration, and a constructor method. While none of these requirements are particularly odious, they obscure the intent: identifying the differences between textboxes, radio groups, checkbox groups, and other HTML form fields.

Using Object Factories

Looking at the code for the Field modules, there are two primary tasks that need to be solved: creating and displaying field objects. The process of creating fields is handled by a series of constructor functions, and the process of displaying fields is handled by the display_row() method in the Field class.

Each type of field object is created by a different method. Textbox objects are created by the constructor of the Field::Text class, radio group objects are created by the constructor of the Field::RadioGroup class, and so on. But there's very little difference between these objects. In fact, the only real differences between these objects are in the data they store.

Because there are no behavioral differences between these objects, there's no necessity to create multiple class definitions. In fact, all of these objects could be created through different methods in a single class. After all, there's nothing special about object constructors in Perl – they're just class methods that happen to create objects.

Refactoring the code to take advantage of this observation, we can replace the entire class hierarchy with two classes: one to display fields and one to create field objects. The new Field class is very easy to write; it contains all of the behaviors shared across field objects. Currently, this is only the display_row() method, and a basic constructor:

```
package Field;
use CGI qw(:standard);

sub new {
    return bless {}, __PACKAGE__;
}

sub display_row {
    my $self    = shift;
    my $sub     = $self->{-procedure};
    return Tr(td($self->{-label}), td($sub->(%$self)));
}
```

The rest of the magic is in an object factory class, a class that exists to create other objects. This class, FieldFactory, contains the methods for creating and customizing new Field objects, init_field() and init_group(). The init_field() method handles the bulk of the initialization and customization of a new Field object, while the init_group() method handles tasks common to initializing group fields.

Here's the start of the FieldFactory class. These two methods are almost exactly the same as the previous versions. The main difference is that the init_field() method customizes a new object, $obj, not the current object, $self (now a FieldFactory object):

```
package FieldFactory;
use CGI;

sub new {
    return bless {}, __PACKAGE__;
}

sub init_field {
    my $self = shift;
    my %params = @_;

    my $obj = new Field;

    ## Assign the key/value pairs for this object
    while(my ($key, $value) = each %params) {
```

```
        $obj->{$key} = $value;
    }

    ## Create the field name from the text label
    my $name = "\L$self->{-label}";
    $name =~ s/ /_/g;
    $obj->{-name} = $name;

    return $obj;
}

sub init_group {
    my $self        = shift;
    my $procedure   = shift;
    my $label       = shift;
    my @values      = @_;

    $self->init_field(
        -label      => $label,
        -procedure  => $procedure,
        -values     => \@values);
}
```

The remainder of the FieldFactory class is composed of factory methods which call these two init methods to create Field objects. Here is the factory method that creates textbox fields. It is almost identical to the old Field::Text constructor:

```
sub textbox {
    my $self = shift;
    my $label = shift;

    return $self->init_field(
        -label      => $label,
        -size       => 50,
        -maxlength  => 50,
        -procedure  => \&CGI::textfield);
}
```

The real benefit comes from adding new factory methods. Here are a few more which create radio groups, checkbox groups, and pop-up menus. Note that all we need here is the code. The extraneous package declarations and other magic incantations are no longer necessary:

```
sub radio_group{
    my $self = shift;
    $self->init_field_group(\&CGI::radio_group, @_);
}

sub checkbox_group{
    my $self = shift;
    $self->init_field_group(\&CGI::checkbox_group, @_);
}

sub popup_menu {
    my $self = shift;
    $self->init_field_group(\&CGI::popup_menu, @_);
}
```

With these changes to the Field module, the CGI program needs some minor changes. First, we have to create a FieldFactory object. Next, the constructor calls to create form fields need to be replaced with method calls on the factory object. The updated code looks something like this:

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

use strict;
use Field;
use FieldFactory;

my $factory = new FieldFactory;

my @fields = (
    $factory->textbox('Name'),
    ...
);

my @rows;
foreach my $field (@fields) {
    push (@rows, $field->display_row());
}

print start_form(),
        table(@rows),
        submit(), reset(),
        end_form();
...
```

Conclusion

This program demonstrates there's always more than one way to do it. Simple and straightforward programs may be easy to write initially, but they can lead to readability and maintainability problems later on as they grow. Cleaning up with ad hoc data structures can help some areas of a program and hurt others.

Restructuring programs to use modules is a very big win, and there's more than one way to factor out common code into modules. One common technique is to create a hierarchy of classes to solve a problem. Another is to create an object factory instead of a class hierarchy to describe differences between objects. Each technique has its benefits and its uses. In this example, using an object factory helped simplify the implementation of an application-specific module with very little impact on the main of the program.