;login:

THE MAGAZINE OF USENIX & SAGE

June 2002 volume 27 • number 3

inside:

PROGRAMMING

PRACTICAL PERL

by Adam Turoff

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

practical perl

Cleaning Up With Dispatch Tables

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

Editors' Note: Please join us in welcoming Adam Turoff as our new "Practical Perl" columnist.

In this column, we'll be exploring tips and techniques for writing better Perl programs. This month, we examine a very powerful and elegant technique – using dispatch tables to simplify your code.

Recovering from Bad Code

Over time, programmers will inevitably come across a spot of bad code. Perhaps the last rough patch of code was left behind by a former colleague. Maybe it was something that you yourself wrote a few months or years ago. In many ways, the provenance of a piece of ugly code doesn't matter too much – especially when it works in spite of its ugliness. But bad code does make maintenance more difficult, and significantly more frustrating.

This pattern appears to be especially true with Perl programs. Lots of factors contribute to this perception, but the most important one is that Perl is designed to let you, the programmer, solve a problem in the way you feel most comfortable. This means that you are not only expected, but also encouraged to start out writing "baby talk Perl" as a beginner, progressing at your own pace through some of the more advanced features and idioms in the language.

The advantage here is that if you can quickly visualize a quick-and-dirty solution to a problem, very little stands in your way. The disadvantage is that though your brute-force solution may get the job done before the boss fires you, it may cause problems a few months down the road when it is time to extend your program.

Here's an example. The program fragment below needs to perform one of a set of possible operations on a database. The operation is specified through the $mode variable, passed in through the command line, a CGI program, or some other mechanism. The easy way to choose the proper operation is through a cascade of if / elsif / else statements:

```
if ($mode eq "insert") {
    ## insert a record into a database
} elsif ($mode eq "update") {
    ## update an existing record in a database
} elsif ($mode eq "delete") {
    ## delete a record in a database
} elsif ($mode eq "display" or $mode eq undef) {
    ## display the contents of the database
    ## Note: this is the default operation
} else {
    ## Error: no valid operation specified
}
```

Nested Conditionals

For a small series of conditions, this kind of coding is simple, easy to write, and easy to understand and extend. Sometimes, the number of cases to be handled runs into the dozens, making it difficult to keep all of the possibilities in your head. And frequently, each block within this cascade contains dozens of lines of code, making the entire construct take hundreds of lines.

Suppose the requirements for the program change and we need to handle user authentication. If this code were part of a Web log system, we might want a single user to be able to insert and update messages into his own Web log but not other users' Web logs. Similarly, we would want to allow site maintainers to delete inappropriate messages but disallow a regular user from deleting messages. The code might be extended to look something like this:

```
if ($mode eq "insert") {
    if (is_current_user($user)) {

        ## a user is posting into his own Web log

    } else {

        ## Error: attempt to post into someone else's Web log
    }
} elsif ($mode eq "delete") {
    if (is_maintainer($user)) {

        ## site maintainer is deleting an inappropriate posting

    } else {

        ## Error: not a site maintainer; cannot delete messages

    }
}
```

As requirements mount, the simple brute-force design leaves an ever increasing pile of ugly (and difficult to maintain) code in its wake.

Using Dispatch Tables

These types of if / elsif / else cascades tend to inspect the value of a single variable. In the first example code above, we're choosing one of many possible execution paths by inspecting the $mode variable. Because we're using a series of if / elsif statements, only one of statements in this block will execute.

If we turn this problem on its side, then we see that we're associating a set of operations with each possible value for the $mode variable. This behavior should sound familiar, because it describes the behavior of one of Perl's three fundamental datatypes: the associative array, more commonly known as the hash. This insight is the key to using dispatch tables: associating data with code, using references to Perl subroutines (similar to function pointers in C).

A simple dispatch table looks like this:

```
my %dispatch = (
    insert  =>  \&do_insert,
    update  =>  \&do_update,
    delete  =>  \&do_delete,
    display =>  \&do_display,
);
```

Elsewhere in our program, we would find definitions for the do_insert, do_update, do_delete, and do_display subroutines. The bodies of these subs are simply the statements that were previously found within the if / elsif statements.

Next, we refer to the dispatch table to find the piece of code we need to execute when we want to perform one of these operations:

```
my $sub = $dispatch{$mode};

if (defined($sub)) {       # we've found a sub; call it
    $sub->($param1, $param2, ...);
} else {

    ## Error: no sub found; this isn't a known mode

}
```

And that's it. We've clarified our code in a number of important ways:

- Each chunk of code within the (possibly lengthy) if / elsif / else cascade now has a name: do_insert, do_update, etc.
- The dispatch table concisely associates when we want to perform an operation (hash keys) with what operation we want to perform (hash values).
- Extending the dispatch table is as simple as defining a new sub and adding an entry in the dispatch table.
- If necessary, new behaviors can be added or deleted from the dispatch table at runtime.

More Advanced Dispatch Tables

If you look closely, you'll see that we haven't quite replaced our original code yet. Our first dispatch table handles the case where the mode is explicitly specified, but does not handle the default mode, where the value of $mode is undefined. This is actually quite simple to fix with a few lines of setup code:

```
my $key = $mode;

## handle the default mode
$key = "display" unless defined $key;

my $sub = $dispatch{$key};

## continue as before
```

At this point, you may be thinking that dispatch tables are sufficient for replacing simple cascading if / elsif, but not more complex structures like those found in the second code sample above. More complex code blocks can be handled with dispatch tables, but they require a little more ingenuity.

Recall that two embedded conditionals need to be handled: "insert" operations being performed by the "current user" and "delete" operations being performed by a maintainer. We can update our dispatch table to look like this:

```
my %dispatch = (
    insert-user    =>   \&do_insert,
    delete-maint   =>   \&do_delete,
    display        =>   \&do_display,
       ## ...
    );
```

and can update the key for our dispatch tables with some more setup code.

```
my $key = $mode;

$key = "dispatch" unless defined $key;
$key .= "-user"
    if (is_current_user($user) and $key eq "insert");
$key .= "-maint"
    if (is_maintainer($user) and $key eq "delete");

## ...
```

Note that the dispatch table contains no generic "insert" or "delete" action. This is because our setup code filters out inserts that are not being performed by the current user and deletions that are not being performed by the maintainer. As a result, these invalid errors don't map to a valid key in the dispatch table, and raise an error.

Synthetic Keys

This technique for supplying synthetic values can be quite powerful. We started out with simple string equality comparisons, but dispatch tables can be used with more complicated comparisons, including regular expressions. Again, all we need to do is add a little more setup code before evaluating the dispatch table:

```
$key = "music" if $input =~ m/jazz|blues|ragtime/i;
$key = "sport" if substr($game, -4) =~ m/ball/i;
$key = "pair" if @terms == 2;
```

Simplifying Dispatch Tables

At this point, we've seen how to convert simple cascading if / elsif blocks into dispatch tables, as well as how to convert some nested conditionals into dispatch tables. But we haven't seen how to convert a series of equivalent values into keys in a dispatch table. For example, take this test:

```
if ($input eq "baseball"
    or $input eq "football"
    or $input eq "basketball"
    or $input eq "volleyball"
    or $input eq "racquetball"
    or $input eq "squash"
    or $input eq "tennis"
    or $input eq "golf") {
    ## do something with sports
}
```

The simple and obvious solution is to put one entry in the dispatch table for each sport we're trying to match:

```
my %dispatch = (
    baseball    =>  \&do_sports,
    football    =>  \&do_sports,
    basketball  =>  \&do_sports,
    volleyball  =>  \&do_sports,
    racquetball =>  \&do_sports,
    squash      =>  \&do_sports,
    tennis      =>  \&do_sports,
    golf        =>  \&do_sports,
    ## ....
);
```

Alternatively, we could have one entry called "sport", and use some setup code to translate each of these sports to the value "sport" just before we inspect our dispatch table. That would have the advantage of having one definition in the dispatch table that matches "sport".

If all we want to do is remove seven extraneous declarations in our dispatch table, we don't necessarily need to jump through such hoops. We could simply declare one particular sport in the dispatch table, and declare later on that the other seven sports have the same behavior as our first sport:

```
my %dispatch = (
    baseball   => \&do_sports,
    ## ...
);

## These values are equivalent to $dispatch{baseball},
## whatever it may be
my @equivalent = qw (
    football     squash
    basketball   tennis
    volleyball   golf
    racquetball
);

foreach (@equivalent) {
    $dispatch{$_} = $dispatch{baseball};
}
```

Using Closures

Finally, there's one last issue to discuss: why do we need to create subroutines just to take their references? If we have a complicated set of statements, creating a new subroutine and giving it a descriptive name helps clarify the intention of our code. On the other hand, for dispatch behaviors that are just very short subroutines – one or two statements – we can create an anonymous subroutine, also known as a closure.

As the name implies, an anonymous subroutine is a subroutine that simply has no name:

```
my %dispatch = (
    coffee   =>   sub {print "Turning on the coffee maker.\n"},
    tea      =>   \&make_tea,
);
```

In this example, both values in the dispatch table are subroutine references. There is no difference in the way they are invoked. As a result, our code that grabs a value from the dispatch table doesn't care if a value is a reference to a named subroutine or an anonymous subroutine. The difference here is one of describing intent – the layout of this code tells other programmers that the process to make tea is more complicated than the process to make coffee.

Conclusion

Dispatch tables are an excellent way to bring order to disorderly code. Their primary benefit comes from separating when actions take place from what actions take place. While this technique can be used to simplify and refactor poorly written code, it can also be used as a design tool to create maintainable software simply and easily.

Dispatch tables are an excellent way to bring order to disorderly code.