ADAM TUROFF

practical perl

ERROR HANDLING PATTERNS IN PERL

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

HANDLING ERRORS IS THE BANE OF any program. In some programming languages, error handling is tedious to do properly, so we often forget to do it, or we do it improperly. In Perl, there are a few common idioms for handling errors that are both robust and easy to use.

I'm a big fan of design patterns in software development. Through patterns, we can talk intelligently about the code we write, the modules we use, and the behavior (and misbehavior!) of the software we come across on a regular basis. Patterns are a vocabulary that lets us focus on the big picture and ignore the meaningless high-level or low-level details.

In thinking about software design patterns, many people reach for the book *Design Patterns: Elements of Reusable Object Oriented Software*, written by Erich Gamma et al. However, patterns are a deep topic, and there is much more to know about patterns than is found in that book.

The concepts behind patterns did not start with one book describing better ways to build object-oriented software. In fact, the idea started with an alternative view of architecture put forth by Christopher Alexander in the 1970s. Alexander's premise was that we need a common language to discuss architectural principles—something the customer, the engineer, and the architect can all understand. When specialists focus on minutiae or elevate professional fashion over customer needs, we all lose.

Alexander's key insight is that we can work together to build open-ended vocabularies that describe the systems we build—whether they are buildings, towns, cars, airports, compilers, network monitoring software, or Web-based applications. In the realm of software, patterns are about describing the behavior of a module of code or an entire system. Once you start to see a pattern, it is easy to see the pattern repeated. From there, it is easier to repeat the good patterns and avoid the bad ones.

Patterns in Perl

Patterns came to software development through analysis of object-oriented systems. A classic pattern describes how to construct an object with a specific set of known behaviors, and how to combine compatible objects based on the patterns they implement. This school of design is quite prevalent within the Java community. If you have ever come across an iterator or a factory, you've seen some of the behaviors described in *Design Patterns* in use.

But patterns are not restricted to objects or any other domain. Because patterns are an open-ended vocabulary, we can use patterns to describe different levels of software, ranging from a single line of code all the way up to a large, complex project like a database server or a Web-based content management system. Patterns are *everywhere* in software.

For a concrete example, look at the following patterns for error handling. If you are familiar with C, you may have seen this idiom for opening a file:

```
FILE *fp;
fp = fopen("/my/file", "r");
if (fp == NULL) {
    return -1;  // ERROR - could not open file
}
```

In Perl, there's always more than one way to do it. If you learned how to program in C, you can program in Perl in a C-like manner:

```
open(my $fh "/my/file");
if (!$fh) {
    return -1;  ## ERROR - could not open file
}
```

In these brief snippets, there is exactly one operation being performed: opening a file. If there is any problem opening this file, then the operation terminates immediately.

Here is a more natural expression of the same basic intent in Perl:

```
open(my $fh "/my/file") or return -1;
```

In this formulation, there is one operation to perform, and it is expressed all at once in a single statement. Furthermore, the intent of the whole statement reads quite naturally: *do something or recover immediately*. Not only is this statement easier to write, but it is much easier to read. Consequently, this statement is also easier to remember and get right the first time.

In many simple scripts, it is common or even advisable to terminate immediately at the first point of failure. This pattern is known as "open or die," and it is one of the most common patterns in Perl programming:

```
open(my $fh "/my/file") or die "Cannot open '/my/file'\n";
```

Using "open or die" may seem extreme at first, but it provides a simple way to express a set of necessary preconditions for a script. For example, consider a script run periodically by cron that needs to read and write some files. If any of those files are missing or cannot be created, the script cannot proceed. If it does continue to run, it could generate bad output, or, in the worst case, do damage to a running system. Using the "open or die" pattern allows this script to open all of its files and succeed, or gracefully terminate when any one of its files cannot be opened.

Error Handling in Perl

How does the "open or die" pattern work? The key is the ultra-low-precedence `or` operator that connects the two statements. If there is any true value whatsoever on the left half of the expression (in this case, the result of an open operation), it will return that value immediately and not evaluate the right-hand side (`die`). The right-hand side (`die`) will be executed only if the left-hand side (`open`) returns a false value.

This pattern uses the `or` operator instead of the higher-precedence || (Boolean or) operator, for a couple of reasons. First, it is clearer when reading the code. Second, because `or` is an ultra-low-precedence operator, there is no ambiguity:

```
## The first statement *must* be "open" with two parameters
## The second statement *must* be "die" with one parameter
open FH, "/my/file" or die "Cannot open '/my/file'\n";
```

Using the higher-precedence || operator would be ambiguous to Perl:

```
## Is the second parameter "/my/file" || ....
## or is it an open followed by || die?
open FH, "/my/file" || die "Cannot open '/my/file'\n";
```

Another important characteristic of "open or die" is the behavior of the left-hand side of the expression, open. If open encounters *any* error whatsoever, it will return a false value. For any other result, it will return *some* true value. Therefore, the die clause in this statement will execute only when there is an open failure. (The actual cause of the failure can be found elsewhere, in the special variable $!.)

The true power in this small pattern is not that it is a concise expression of the proper behavior for opening files, but in its general utility in similar contexts. Most Perl functions that handle system interaction provide the same behavior—return false on error, true on success. This includes functions like chdir, mkdir, unlink, and so on. For conciseness, Perl programmers generally call this overall pattern "open or die."

In C, the pattern is just the opposite—return zero (false) on success, and a non-zero error code (true) on error. This leads to cumbersome idioms like the example above with fopen. In Perl, the system built-in function behaves in a C-like manner, returning false (zero) on success, and a true (non-zero) value on failure. The result is similar to the cumbersome "system and die" pattern, since the system built-in adheres to this C-like behavior:

```
system "/bin/date" and die "Can't find '/bin/date'";
```

(The "system and die" pattern is generally regarded as broken. Larry Wall has declared that this unfortunate misfeature will be fixed in Perl 6.)

Recovering from Errors in Perl

Using the "open or die" pattern is a great way to terminate your script at the first sign of error. But sometimes you do not want to terminate your script. Rather, you need to exit immediately from a subroutine, break out of a loop, or just display a warning message.

Even though this pattern is commonly called "open or die," the right-hand side doesn't need to call die. Any recovery action fits the pattern, including return, warn, or even print.

Below is a sub that takes a filename and returns all non-blank lines that do not start with a hash character (i.e., comments). If it cannot open a file, it exits immediately and returns false:

```
sub get_lines {
   my $filename = shift;
   my @lines;

   open(my $in, $filename) or return;
   while (<$in>) {
      next if m/^$/;  ## Skip blank lines
      next if m/^#/;  ## Skip comment lines
      push (@lines, $_);
   }

   return @lines;
}
```

Carp, Croak, and Friends

Functions like die and warn can report exactly where an error occurred. That may work for scripts, where the cause of the error is likely nearby. But this behavior does not work very well when using modules. Although your program may have issued a warning or terminated at line 135 of SomeModule.pm, that message may not mean anything to you, especially if you installed SomeModule from CPAN.

It makes more sense to identify the location of the code that led to the error in SomeModule.pm. This is more likely the cause of the problem, especially when using well-tested modules. This is the problem that the Carp module solves. Carp is a standard module that is bundled with Perl which provides the error-reporting functions carp and croak, which can be used in place of warn and die. When these functions display a warning or a termination message, they report the location where the current sub was called, not the location where an error was encountered (i.e., the location of the call to carp or croak).

Here is a simple program that demonstrates the difference between these two sets of functions:

```
1: package Testing;
2: use Carp;
3:
4: sub test_carp {
5:     carp "Testing carp";
6: }
7:
8: sub test_warn {
9:    warn "Testing warn";
10: }
11:
12: package main;
13:
14: Testing::test_carp();
15: Testing::test_warn();
```

And here is the result:

```
Testing carp at test.pl line 14
Testing warn at test.pl line 9
```

Note that carp focuses attention on where the sub test_carp was called, while warn focuses attention within the body of test_warn. This is why module authors should prefer the functions provided by Carp over the standard built-in functions.

Returning Errors in Perl

Patterns for handling errors are important. By understanding when and how functions like open return errors, we can use a clear and concise pattern for handling errors when they occur. The beauty behind this pattern lies in its extensibility. Not only can it be used with many recovery strategies, but it can be used with any sub that signals errors the same way open does.

Recall for a moment how open communicates errors: It returns a false value on failure, and some true value on success; any error message will be returned through a pre-defined variable ($! in this case). Any other sub that behaves in this manner can be used with the “open or die” pattern.

This process sounds simple enough, except that there is some subtlety involved. Remember that there are precisely five false values in Perl:

- The number 0
- The string “0”
- The empty string
- The empty list
- The undefined value (undef)

There is another, subtle wrinkle in Perl behavior. Subroutines can be called in one of two possible ways: in scalar context and in list context. In list context, there is precisely one false value, the empty list. All other values produce a list of one element, which is true. The following program illustrates the differences:

```
sub return_empty  {return;}
sub return_string {return "";}
sub return_zero   {return "0";}
sub return_0      {return 0;}
sub return_undef  {return undef;}

## Test scalar return values
($_ = return_empty) and print "True (scalar empty)";
($_ = return_string) and print "True (scalar string)";
($_ = return_zero) and print "True (scalar zero)";
($_ = return_0) and print "True (scalar 0)";
($_ = return_undef) and print "True (scalar undef)";

## Test list return values
(@_ = return_empty) and print "True (list empty)";
(@_ = return_string) and print "True (list string)";
(@_ = return_zero) and print "True (list zero)";
(@_ = return_0) and print "True (list 0)";
(@_ = return_undef) and print "True (list undef)";
```

As described above, this program produces the following output:

```
True (list string)
True (list zero)
True (list 0)
True (list undef)
```

These rules may sound complicated, but they really aren’t. They help Perl do the right thing in a variety of common circumstances. This program demonstrates that if you want to return a false value in all circumstances, just use a simple `return` statement—it will return false whether you, the caller, uses list or scalar context. Therefore, any sub that uses this behavior can plug right into the “open or die” pattern with no extra effort.

Alternative Error Mechanisms in Perl

Returning a false value is often sufficient for signaling an error. But sometimes there are legitimate values returned that happen to be false but do not signal an error. Consider a sub that returns a series of numbers that could include zero, or a sub that returns a series of strings which could include the empty string. In these situations, it may not be feasible to say that “any false value” signals an error. In these cases, it is generally better to say that “the undefined value” signals an error.

A common example of this pattern is reading lines from a file:

```
while (<>) {
  ...
}
```

The purpose of looping over a file line by line is to process one line at a time. However, sometimes it is possible to read an empty string from a file, or a line containing the single character 0. These values should not signal end-of-loop.

What is actually happening here is that Perl sees that construct and interprets it as this instead:

```
while (defined($_ = <>)) {
  ...
}
```

This behavior allows Perl to act as we expect it should. The construct will read every line from the file, including blank lines and lines that contain the number zero. The undefined value will be returned when there is an error reading from the file, such as when trying to read past the end-of-file. Thus, Perl can read each and every line from a file (regardless of whether that line is "true" or not) and terminate when reaching end of file.

You can reuse this pattern in your programs as well. If you need to return false values (like zero) from a sub but still want to plug into the "open or die" pattern, just remember to check to see whether the return value is defined. If it is not, then some error must have occurred:

```
defined(add_user()) or die "Cannot add user";
```

Conclusion

Handling errors is a key aspect of any program. In Perl, there are many patterns for handling errors. If you are comfortable programming in a C-like manner, you can use the error-handling patterns that feel comfortable to you. However, native Perl patterns for handling errors are simpler to use and easier to get right the first time.

Corrections

In my last column on Class::DBI, I used this idiom to edit a temporary file:

```
sub get_input {
  open (my $fh, ">/tmp/library.data.$$");
  ....
}
```

Jeremy Mates wrote in, identifying this as a security flaw. I want to thank him profusely for pointing this out. Jeremy goes on to say:

> Insecure temporary file handling problems are unfortunately far too common in code still being written and used, despite being trivial to eradicate through the use of secure alternatives such as mktemp(1) and various modules in Perl, such as File::Temp.
>
> For soliciting input from an external editor, I recommend the use of my Term::CallEditor module, which uses File::Temp to create a secure temporary file that an editor can be run on.
>
> Thanks, Jeremy.