A Perl toolbox for regression tests

TESTING TOOLS

With a test suite, you can fix bugs and add new features without ruining the existing codebase.

BY MICHAEL SCHILLI

A program that works right the first time is uncommon enough to make you suspicious. In test-driven development, developers first define a test case before implementing a feature. Inevitably, the test will fail at first, but as soon as the feature gets implemented – presto – the test suite gives a thumbs up. This technique motivates developers throughout the development lifecycle; each new test adds to the test suite and will be executed again and again as the project emerges. Small steps add up to a detailed test suite that no quality management department in the world would be capable of achieving.

In case of ongoing development and refactoring, there is always some danger of a change introducing undesirable side effects. Having the ability to run hundreds of test cases at no cost takes the worries out of replacing parts of a system. Developers can roll out new releases on a regular basis and yet sleep tight and enjoy sweet dreams. You don't need to be an Extreme Programming enthusiast to see the benefits.

Standard Suite

Thank goodness the Perl community is well-mannered enough to add a test suite that checks critical functions to almost every CPAN module. But what happens if you are developing a small script rather than a module? Over the years I have come to the conclusion that scripts should only parse command line parameters and output help texts. I use modules for the remaining functionality. This allows other scripts to leverage the modules. And, of course, each module has accompanying documentation and a test suite, right?

The TAP protocol (Test Anything Protocol) has become the de facto standard for regression testing in Perl. TAP typically outputs a header first to indicate the number of tests to be performed; this

Figure 1: Output from the simple.t test script.

is followed by a line for each test that reads *ok* if the test is successful, and *not ok* otherwise:

```
1..3
ok 1
not ok 2
ok 3
```

Of course, this kind of output isn't exactly easy to read if you are performing hundreds of tests. To change this, an overlying test harness provides a summary telling you if everything checked out, or how many tests failed.

Listing 1 shows an example. Traditionally, Perl test scripts tend to have the file extension **.t* and reside below the *t* directory in the module distribution. As the test cases often check similar things, and initiate similar actions, there are some special test modules – such as

www.sxc.hu

Listing 1: simple.t

```
01 #!/usr/bin/perl
02 use strict;
03 use warnings;
04
05 use Test::More tests
   => 4;
06
07 BEGIN {
08  use_ok("Config::
   Patch");
09 }
10
11 my $p =
12   Config::Patch->new(
13  key => "foo");
14
15   # #1 true
16 ok($p, "New object");
17
18   # #1 eq #2
19 is($p->key,
   "Waaah!",
20  "Retrieve key");
21
22   # #1 matches #2
23 like($p->key(),
   qr/^f/,
24  "Key starts with
   'f'");
```

Test::More – that simplify the process and help to avoid writing redundant test code. After all, the same design principles apply to test code and application code.

The sample script tests the *Config::Patch* CPAN module, which patches configuration files in a reversible way. *Test::More* first sets *tests => 4* to specify that exactly four tests will be performed. This is important: if the test suite quits beforehand, you will want to know all about it. Some developers water down this test by stipulating *use Test::More qw(no_plan);* while they are busy extending the test suite, but best practices dictate a fixed number of tests.

The *simple.t* test script first calls *use_ok()* (exported by *Test::More*) to check if the *Config::Patch* module actually loads. The *new* constructor then (hopefully) returns an object. The following function, *ok()*, also from *Test::More*, writes *ok 2* to standard output if the object has a true value, and *not ok 2* if not. A second parameter, which can be passed to *ok()* optionally, sets a comment that gets printed next to the test result. Figure 1 shows the output from the script.

The third test case shows how useful it can be to use the *is()* function from *Test::More* rather than *ok()* if you need to compare things. If

Table 1: Test Utilities

Modul	Function
Test::Simple	Common test utility, includes Test::More
Test::Deep	Compares deeply nested structures
Test::Pod	Validates POD documentation
Test::Pod::Coverage	Checks if all functions are documented
Test::NoWarnings	Alerts on warnings
Test::Exception	Checks if exceptions are thrown
Test::Warn	Checks if warnings are correctly output
Test::Differences	Graphical display of deviations
Test::LongString	Checks long strings
Test::Output	Catches output to STDERR/STDOUT
Test::Output::Tie	Catches output to file handles
Test::DatabaseRow	Checks database query results
Test::MockModule	Simulates additional modules
Test::MockObject	Simulates additional objects

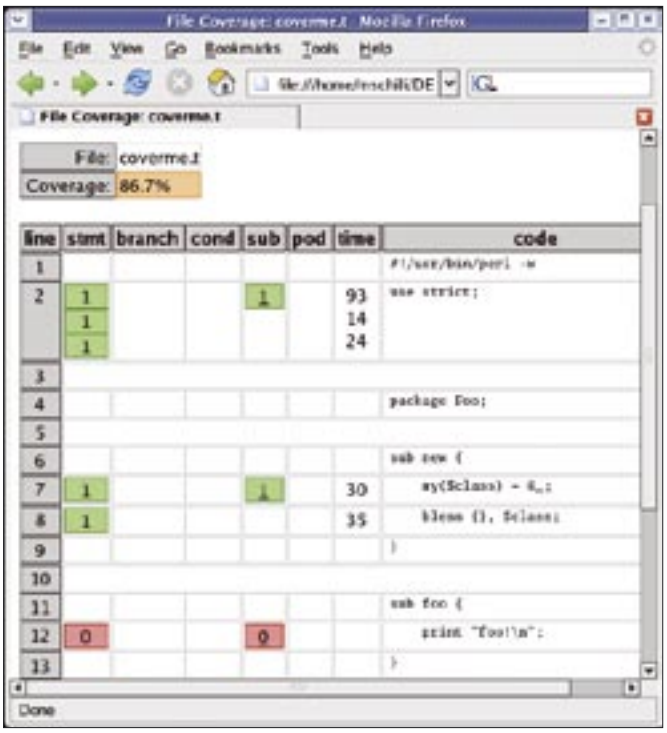

Figure 2: Test coverage: not all methods have been called.

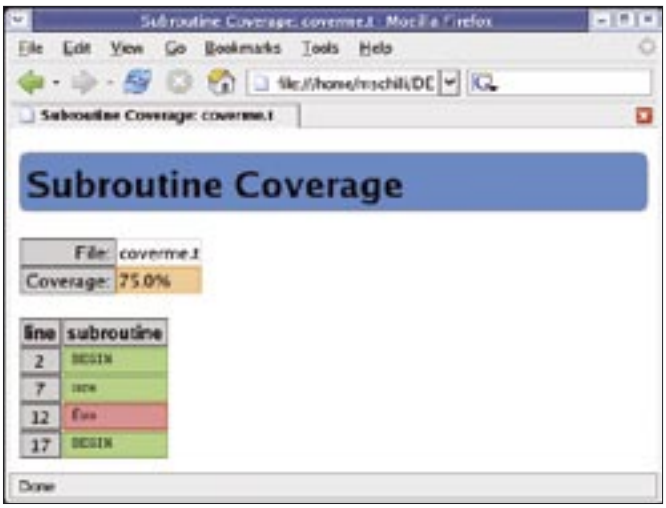

Figure 3: Functions/Methods covered in the coverme.t test script.

something goes wrong, the test script not only displays the test comment and the line number, but also the difference between the expected and actual values. Line 19 in Listing 1 causes an error, just for demonstration purposes. To achieve meaningful output with *is()*, you need to pass the returned value as the first parameter, and the value you expected as the second parameter.

The *like()* function used in the fourth test case, expects a regular expression as the second argument that the first parameter must match instead of a comparative value. If the Regex doesn't match, a detailed error message is output in a similar fashion to *is()*. Finally, *Test::More* outputs a polite "Looks like you failed 1 tests of 4." Being polite is important: nobody wants to be scolded by a stickler from QA.

The *prove* script, which is part of the Test::Harness CPAN module, runs one or more test scripts. The version of Test::Harness that comes with perl-5.8 does not include the script, so make sure to download the latest version from CPAN. Running *prove* against a test script produces the following if all tests succeed:

```
$ prove ./simple.t
./simple....ok
All tests successful.
Files=1, Tests=4, 0 ➥
wallclock secs
(0.08 cusr +  0.01 csys =  ➥
0.09 CPU)
```

If you are interested in a breakdown of the results, run *prove* with the *-v* (for verbose) option to display the individual *ok* and *not ok* lines along with the comments. If you are running a test file from a module distribution without installing the module, the *-b* parameter is useful, as it uses the module files that *make* drops in the *blib* directory.

CPAN modules use *make test* prior to installation to do what *prove* does at the command line. The *ExtUtils::MakeMaker* module, which provides this functionality, messes with Perl's library include path, *@INC*, to allow the test suite to run modules that have not been installed so far. Newer modules come with the *Module::Build* module, which provides similar but advanced functionality.

In Depth

Besides *Test::More*, CPAN has innumerable utility modules that facilitate the process of creating test code without needing to retype the same lines again and again. *Test::Deep*, which compares deeply nested structures, is an example.

Listing 2 shows a short test case that calls the *MP3::Info* module's *get_mp3tag*

Table 2: Test Analysis Tools

Modul	Function
Test::Harness	Standard harness
Test::Builder	Base for new test utilities
Test::Builder::Tester	Tests for new test utilities
Test::Harness::Straps	Base for a newly developed test harness
Devel::Cover	Analysis and display of test coverage
Test::Distribution	Checks module distributions for completeness

Listing 2: mp3.t

```
01 #!/usr/bin/perl
02 use warnings;
03 use strict;
04
05 use Test::More tests => 1;
06 use Test::Deep;
07 use MP3::Info;
08
09 my $tag =
10   get_mp3tag(
11   "Gimme_Shelter.mp3");
12
13 cmp_deeply(
14  $tag,
15  superhashof(
16   {
17    YEAR   => re(qr(^\d+$)),
18    ARTIST =>
19      re(qr(^[\s\w]+$)),
20   }
21  )
22 );
```

function. If the MP3 file has a proper set of tags, the function returns a reference to a hash, which contains a number of keys, such as *ARTIST*, *ALBUM*, and so on. Instead of checking if the returned value really points to a hash, and then walking through a number of required hash keys, the *cmp_deeply* function does this at one fell swoop.

cmp_deeply expects array or hash references as its first two arguments, performs an in-depth check, and compares the underlying elements. The call to *cmp_deeply($ref1, $ref2)* thus returns true if *$ref1* and *$ref2* point to equivalent data structures.

But that's not all: this direct comparison can be manipulated using a number of additional functions. For example, you can check if an element in one data structure matches an element in its counterpart using a regular expression. The *re()* function handles this. Or, if an element in the structure contains a reference to a hash, *superhashof()* allows you to specify that the first hash needs only contain a subset of the keys in the second hash.

Leveraging this functionality, Listing 2 checks several things at the same time:

Listing 3: coverme.t

```
01 #!/usr/bin/perl -w
02 use strict;
03
04 package Foo;
05
06 sub new {
07  my ($class) = @_;
08  bless {}, $class;
09 }
10
11 sub foo {
12  print "foo!\n";
13 }
14
15 package main;
16
17 use Test::More tests => 1;
18
19 my $t = Foo->new();
20 isa_ok($t, "Foo",
21   "New Foo object");
```

whether *$tag* is a reference to a hash, for instance, and whether the hash it points to contains the *YEAR* and *ARTIST* keys; additionally, Listing 2 checks whether the values stored below the keys in the hash match the specified regular expression: text with blanks for the *ARTIST* tag and a number for *YEAR*. *Test::Deep* also has a number of practical helper functions that allow you to check simple subtrees of the data structures passed to *cmp_deeply* without resorting to *for* loops. For example, *array_each()* specifies that a node must contain a pointer to an array and runs a test passed to it (*re()* for example) against each element in the array.

Besides *superhashof()*, there is also *subhashof()* to handle cases where the reference hash contains optional elements. *set()* and *bag()* help you discover whether an array contains a series of elements in arbitrary order, with and without repetitions. The array counterparts to the hash functions for optional elements are *subbagof()*, *superbagof()*, *subsetof()* and *supersetof()*.

Completed Tests

Listing 3 shows the definition of a *Foo* class and a test script that then runs, calling the class constructor and using *isa_ok()* to check if an object of class *Foo* is actually returned.

But the test suite has a weakness: it never runs the *Foo*'s *foo()* method; a nasty runtime error might just be holed up somewhere in there, and the test suite would never notice.

In case of smaller projects, a developer would soon notice this lapse; for larger projects, the *Devel::Cover* CPAN module takes over the tedious work; it checks how many possible paths the suite actually covers. Calling the Perl script, we test as follows:

```
perl -MDevel::Cover coverme.t
```

This command creates coverage data in a newly created *cover_db* directory. A subsequent call to *cover* analyzes the data and provides graphics output. (*cover* is an executable script that is installed along with Devel::Cover.) If you then point your browser to *cover_db/coverage.html*, you get a neat overview of the coverage data, as shown in Figure 2. Figure 3 shows the coverage for the test script file *coverme.t*; the output is available in *cover_db/coverme-t.html*.

Devel::Cover not only checks all the functions and methods, but also the coverage of all branches of *if*, *else*, and other conditions. Even though it might be impossible to cover all branches in a large-scale project, it is useful to know where it might be worthwhile putting in some extra effort to improve coverage.

Mockup

One major requirement for a test suite is that it is quick and easy to run, without asking the developer to install too many extra bits and bobbles or to get involved in configuration marathons. But many applications reference complex databases or need a working Internet connection and a specific server. To work around requirements of this kind in the test phase, the Mock Utilities, *Test::MockModule* and *Test::MockObject*, allow you to spoof a very realistic Internet server or database.

Of course, analysis tools such as *Test::Harness* are designed for generic cases, and it is up to developers to decide whether to use the generic tools or design their own analysis tools for more specific requirements. But to remove the need to re-invent basic functionality such as parsing of TAP output time and time again, *Test::Harness::Straps* provides a base class, which developers can extend for private smoke testing.

If you need more information on Perl testing, I can recommend a really excellent new book [2] that has more detailed discussions of all the modules we have looked at here and lots more test tips. ■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/61/Perl*

[2] *Perl Testing*, Ian Langworth & Shane Warden, O'Reilly 2005.

THE AUTHOR

Michael Schilli works as a Software Developer at Yahoo!, Sunnyvale, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at *mschilli@perlmeister.com*. His homepage is at *http://perlmeister.com*.