Devel::Cover - An Introduction

Paul Johnson
paul@pjcj.net

11.1 Introduction

Testing is an important part of the software development process. The more important the software, the more important the testing of it. Some people will even tell you that that you shouldn't write your software until you have written the tests for it.

That's all well and good, and you may even believe in or at least tolerate such philosophies, but how do you know whether your tests are useful? How do you know whether you are actually testing your software, or more specifically how do you know which parts of your software you are and are not testing?

This is where *code coverage* comes in. Code coverage will let you know which parts of your code your tests are exercising. If testing benefits your software development process then you'll likely be interested in what code is being tested and you may well want to write tests to exercise the currently untested code.

Devel::Cover is the Perl code coverage tool. The rest of this paper will provide an introduction to its use together will a basic introduction to code coverage concepts.

11.2 Preparation

In order to use Devel::Cover you will need to download and install it. Devel::Cover is available on CPAN and can be downloaded and installed using your favourite CPAN tool. Devel::Cover contains XS code and so you will need your C compiler available for the build process. Devel::Cover is also available in a standard packaged form for a number of common platforms including ActivePerl.

11.3 Using Devel::Cover

Devel::Cover can be used to measure the coverage of almost any Perl code but it is mostly used (I imagine) in the development of Perl modules, and that is the usage upon which I will focus here. I'll also assume that you are using a standard module layout such as that produced by h2xs or Module::Starter, that you have some tests, and that running them with `make test` does what you expect it to do.

If that is the case, as it is for the vast majority of CPAN modules for example, then getting code coverage data for your code can be as simple as running `cover -test`.

11.4 Getting some coverage

In order to demonstrate the use of Devel::Cover I am going to take a working CPAN module as my example. The module is Shell::Source and, since I am the author, I won't feel bad about telling you all the problems it has.

The primary job of Shell::Source is is to take a shell script and allow it to be sourced into a Perl script in the same way as it could be sourced into a shell script. This can be useful for automating interaction with software which puts important information in environment variables and expects you to source the files with those variables into your interactive shell before running the software. But for the purposes of this demonstration the purpose of the module is not too important.

The module being tested does not have to be installed. In fact, it might even be better if it is not installed, but `make test` should show that the module passes all its tests. To check this, I first run `make test`:

```
$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0,
'blib/lib', 'blib/arch')" t/*.t
t/bash....ok
t/csh.....ok
t/ksh.....ok
t/sh......ok
t/tcsh....ok
t/zsh.....ok
All tests successful.
Files=6, Tests=24,  1 wallclock secs ( 0.49 usr +  0.09 csys =  0.58 CPU)
```

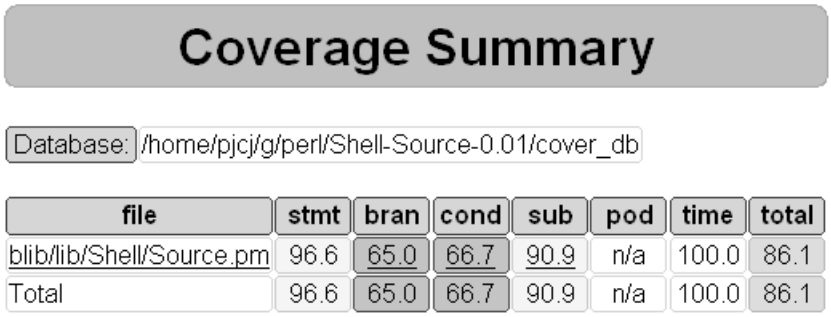
Then I run `cover -test`:

```
$ cover -test
Deleting database /home/pjcj/g/perl/Shell-Source-0.01/cover_db
PERL_DL_NONLAZY=1 /usr/local/pkg/cover/c1/perl-5.8.7/bin/perl "-MExtUtils::Command::MM"
"-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/bash....ok
t/csh.....ok
t/ksh.....ok
t/sh......ok
t/tcsh....ok
t/zsh.....ok
All tests successful.
Files=6, Tests=24, 17 wallclock secs (16.16 usr +  1.01 csys = 17.17 CPU)
Reading database from /home/pjcj/g/perl/Shell-Source-0.01/cover_db


---------------------------- ------ ------ ------ ------ ------ ------ ------
File                           stmt   bran   cond    sub    pod   time  total
---------------------------- ------ ------ ------ ------ ------ ------ ------
blib/lib/Shell/Source.pm       96.6   65.0   66.7   90.9    n/a  100.0   86.1
Total                          96.6   65.0   66.7   90.9    n/a  100.0   86.1
---------------------------- ------ ------ ------ ------ ------ ------ ------


Writing HTML output to /home/pjcj/g/perl/Shell-Source-0.01/cover_db/coverage.html ...
done.
```

11.5 Looking at the results

So now we have gathered our code coverage information. We'll look at what the above output means in a little while, but for now, lets jump straight in and look at the results. The second to

last line above tells us where our HTML output has been sent, so just open up that file in your browser of choice. The output should look something like:



Coverage Summary

Database: /home/pjcj/g/perl/Shell-Source-0.01/cover_db

file	stmt	bran	cond	sub	pod	time	total
blib/lib/Shell/Source.pm	96.6	65.0	66.7	90.9	n/a	100.0	86.1
Total	96.6	65.0	66.7	90.9	n/a	100.0	86.1

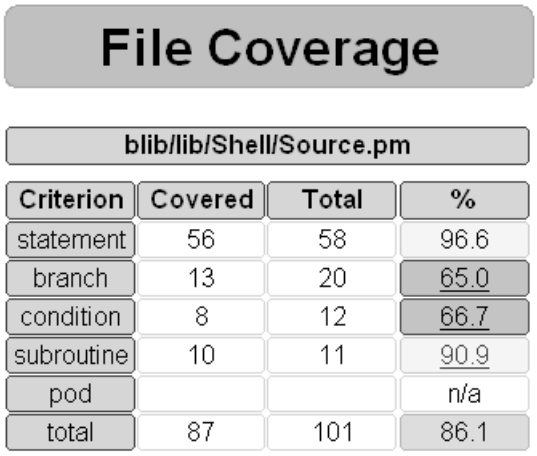
Let's investigate what this is telling us. The first line is simply reminding us of which coverage database we used to generate this report. The interesting part is the next table. Here we see that we have one file for which we have collected coverage information and we see some percentages indicating the amount of code coverage we achieved.

That there is only one file is correct - this is a fairly small module we are testing and all the code is contained in one file. This means that the total figures are obviously just the same as for the one file.

The percentages for the coverage are what we are really interested in though. The headers tell us that there are six types of coverage reported: statement, branch, condition, subroutine, pod and time. The figures given are the percentages we have achieved for those criteria.

If you have one of those new fangled graphical browsers, hovering the mouse over the percentage may well bring up a little popup letting you know how the percentage was calculated. For example, hovering the mouse over the 90.9% of subroutine coverage brings up a little popup containing the text `10 / 11` letting us know that we have covered ten of the eleven subroutines in our code. The more mathematically inclined will be able to confirm that this is indeed 90.9%, or at least that it is pretty close to that.

So let's investigate our coverage further. The name of the file is a hyperlink, taking us to the *File Coverage* page. That page starts something like:



File Coverage

blib/lib/Shell/Source.pm

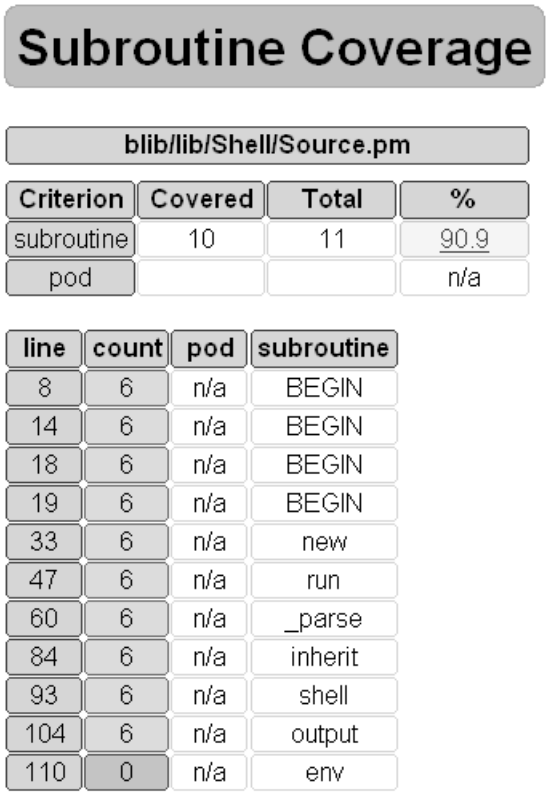
Criterion	Covered	Total	%
statement	56	58	96.6
branch	13	20	65.0
condition	8	12	66.7
subroutine	10	11	90.9
pod			n/a
total	87	101	86.1

Here we see the same information we had on the summary page plus a bit more. We can see the numbers which give us the percentages even if we are using a dodgy old browser. When

looking at coverage for the first time it is often useful to start with subroutine coverage, it being the most coarse criterion.

Subroutine Coverage

We can see that we have covered ten of the eleven subroutines in our module. Obviously, calling a subroutine is a prerequisite to testing it, so it seems that we are doing fairly well here. The question naturally arises as to which of the eleven subroutines we are calling and, more importantly which we aren't calling. This information is available in another page of the report which we can get to by following the link from the subroutine coverage percentage. Doing so takes us to the following page:



Subroutine Coverage

blib/lib/Shell/Source.pm

Criterion	Covered	Total	%
subroutine	10	11	90.9
pod			n/a

line	count	pod	subroutine
8	6	n/a	BEGIN
14	6	n/a	BEGIN
18	6	n/a	BEGIN
19	6	n/a	BEGIN
33	6	n/a	new
47	6	n/a	run
60	6	n/a	_parse
84	6	n/a	inherit
93	6	n/a	shell
104	6	n/a	output
110	0	n/a	env

Paying attention only to the subroutine coverage on that page, we can see again that there are eleven subroutines in our code and that we have covered ten of them, giving us our 90.9% coverage. Looking further we can see that each subroutine has been called exactly six times with the exception of one subroutine, which has not been called at all. We can also see that the offending subroutine is `env` which is found on line 110 of our module. We've already got enough information to start improving our test suite.

You might wonder what all those `BEGIN`s are at the start of the list. Recall that `use Module LIST` is exactly equivalent to `BEGIN { require Module; import Module LIST }`. Recall also that although BEGIN blocks are not normal subroutines and although it is not considered good style, they can be prefixed with `sub` to give the appearance that they are subroutines. So the BEGIN subroutines here are actually our `use` statements.

You might also wonder what all this pod n/a business is about. That's basically telling us that I haven't documented any of the subroutines in the module, or at least not in a way which

Pod::Coverage understands. Oops. Bad programmer! So let's gloss over that for a bit.

Statement Coverage

Now we know that there is one subroutine which is uncovered, lets take a look at the statements in our subroutines. A statement is covered if it is executed at least once, and uncovered if it is not. Uncovered statements may be hiding bugs, so it would be nice if we could execute them all as part of our test suite.

Returning to the previous page we note that we have covered 56 of 58 of the statements in our module, making 96.6%. That sounds pretty good. Looking at the next table on the page we can see that we have a listing of our source code with figures for the coverage of each criterion on each line. Scrolling down, and keeping an eye on the stmt column, we can see that each statement has been covered with the exception of the two statements in the `env` subroutine, which we already knew was uncovered. So writing a test which calls `env` should not only get us 100% subroutine coverage but 100% statement coverage too. A moment's thought tells us that 100% statement coverage will automatically bring us 100% subroutine coverage.

Branch Coverage

The meaning of both subroutine and statement coverage is fairly obvious. But what about branch coverage? What is it measuring? Notice that we have not done so well on branch coverage, covering only thirteen out of twenty branches in the module, giving us 65% branch coverage.

There is a branch in our code wherever we have a conditional statement. That could be an `if` statement, the trinary `?:` operator, or something which reduces to an `if` statement such as `unless` or `or`. In order to fully test our code we may want to exercise both parts of the conditional statement, the part where the condition is true and the part where the condition is false.

Looking at the branch coverage percentages we see that of the ten branches, seven have 50% coverage and three have 100% coverage. Now we just need a bit more information about the branches and whether we need to write tests for the true or false parts of the branch. Fortunately there's just such a page available by following the hyperlink from any of the branch coverage percentages. It will look something like:

Branch Coverage

blib/lib/Shell/Source.pm

Criterion	Covered	Total	%
branch	13	20	65.0

line	true	false	branch
35	0	6	`unless $$self{'shell'}`
37	0	6	`unless $$self{'run'}`
41	6	0	`if length $$self{'file'}`
49	0	6	`unless length $$self{'file'}`
51	0	6	`unless my $fh = $$self{'fh'} = 'FileHandle'->new($run)`
54	0	6	`unless $fh->close`
65	338	12	`if ($line =~ /^(\w+)=(.*)$/) { }`
68	26	312	`if (!defined($ENV{$1}) || $ENV{$1} ne $2 and not grep {$1 eq $_;} @{$$self{'ignore'};})`
76	6	6	`unless $env`
105	6	0	`if defined $$self{'output'}`

So we can see which parts of which branches have not been exercised. Returning to the *File*

Coverage page we can see that all the branches which have not been exercised relate to errors and boundary conditions. I've done fair-weather testing. Testing of errors and boundary conditions can be quite an effort, but you may well be glad you did it should that code ever need to be executed for real. Should you manage to achieve 100% branch coverage you will also have 100% statement coverage.

Condition Coverage

The final code coverage criterion to look at is condition coverage. When we have an `and`, `or` or `xor` condition in our code we want to be sure that that condition evaluaties to both `true` and `false` with every possible combination of inputs. We have covered eight out of twelve conditions giving 66.7% coverage.

Following the link from any of the condition coverage percentages takes us to the *Condition Coverage* page.

Hitting 100% condition coverage can be quite tricky. You might decide that your efforts would be better targetted towards other areas. It is also possible to write quite valid Perl code for which 100% condition coverage is impossible to achieve. If Devel::Cover doesn't have a solution to this problem by the time you read this is will soon, for some value of soon.

Pod Coverage

Pod coverage isn't code coverage at all, it is documentation coverage. In other words, have you documented your code? Devel::Cover uses Pod::Coverage to do the heavy lifting here and just reports its results, so look at the Pod::Coverage docs for options and caveats. In this case my documentation doesn't conform to Pod::Coverage's expectations so Pod::Coverage doesn't think I wrote any documentation at all.

Time Coverage

Time coverage isn't code coverage either. It's a little like profiling at the statement level but it is not particularly accurate. You can use it to get a feel for where the time is being spent in your code, but I wouldn't use it for much more than making rough comparisons.

11.6 Conclusion

So there you have it - a brief introduction to using Devel::Cover and what you can expect from it. If you're one of those people who reads documentation you might notice that I'm still calling this alpha code. That's basically because I can, having no marketing department pushing me to make a stable release. The code itself is fairly stable, though it is not without bugs. I am resisting making a stable release primarily because I may make interface changes.

If you use Devel::Cover you might consider joining the perl-qa list, details of which are available at `http://lists.perl.org/showlist.cgi?name=perl-qa`. And if all this is new to you, take your time, get to understand what Devel::Cover is trying to tell you and how using it might fit into your development methodology. You might find that using Devel::Cover could even change the way you develop Perl code. Stranger things have happened, or so I have heard.