An Introduction to Test Driven Development Using Perl

Grant McLean, Catalyst IT Limited <grant@catalyst.net.nz>
September 2008

This article describes the practise of Test Driven Development and illustrates it with a worked example. The sample tests, source code and tools are written in Perl however the concepts should be applicable to any programming language.

Introducing Test Driven Development

Few people would dispute that testing is a critical part of any software development project. It is unfortunate however that testing is often left until the end. This may be due to a mistaken belief that tests are useless until you have something to test. Test Driven Development (TDD) positions testing in a critical role at the centre of development activities.

Development models based around TDD favour a cyclic flow with a series of iterations building one upon another to deliver functionality in stages. Each iteration follows a flow similar to that illustrated in figure 1.

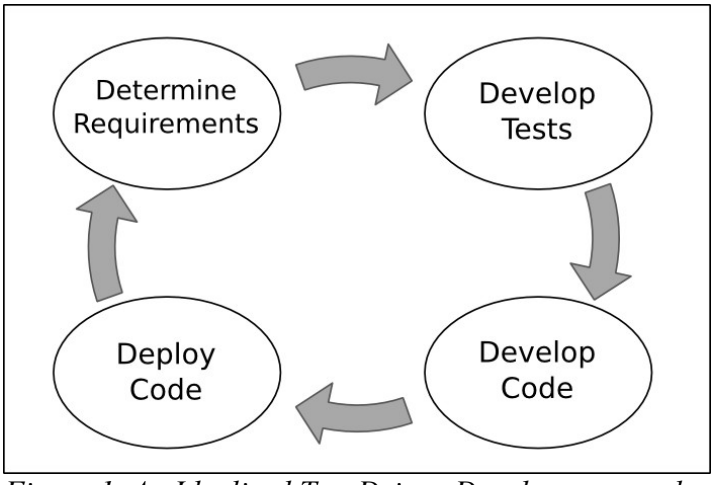


Figure 1. An Idealised Test Driven Development cycle

The process begins with requirements gathering – deciding what will be delivered and defining success criteria (how we'll know when we've delivered it). The requirements definition feeds into the development of a test plan. Development of the program code can proceed once tests have been written to specify correct behaviour for the code. When tests have been written to describe all the requirements and code has been written to pass all the tests, the development is complete and the system can be deployed. The process then begins again with the definition of the requirements for the next set of deliverables.

Of course the outline above paints a grossly oversimplified picture of the processes. The reality is more likely to resemble that shown in figure 2.

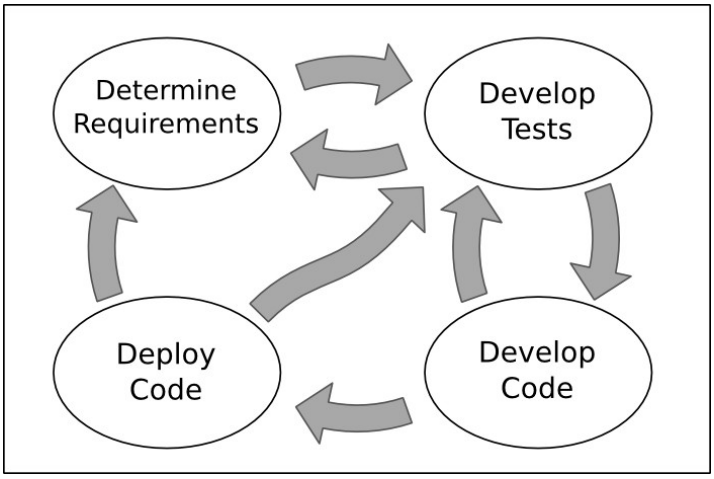


Figure 2. A more realistic TDD 'cycle'

The first problem is that it's impossible to accurately define the requirements if the “customer” doesn't know what they want. And, as any seasoned developer will tell you, the customer **never** knows what they want. When the developers have produced something that the customer can see and interact with, they will inevitably conclude that it's **not** what they want. However they'll usually be able to describe how the prototype differs from what they want and how it might be improved to more closely meet their actual requirements.

It's important to note that the realities of requirements gathering described above are not intended as a criticism of the customer. In many cases the customer can't be expected to describe what they want a computer system to do simply because they don't know what's possible. Furthermore when a number of alternative approaches are possible, making the 'wrong' choice can have effects on development time frames and maintenance costs that are not immediately apparent. When these downstream effects do become apparent it is often necessary to reassess the requirements in light of the new information and business priorities.

The difficulty of requirements definition and the inevitability of changes in priorities are key drivers for selecting a development model with iterations of short duration focussed on delivering something quickly and improving on it incrementally.

The idealised view of requirements being translated into a test suite as a simple one-way flow is not realistic. A general description of a requirement must be translated into a detailed description of the expected behaviours under a variety of conditions. The process of developing the tests for these behaviours will inevitably uncover ambiguities and contradictions in the requirements and the feedback loop between defining the requirements and defining the tests will improve the accuracy of both.

It is also completely unrealistic to expect that a comprehensive test suite will emerge fully formed and ripe for implementation. Instead a general outline of the tests will emerge first and specifics will be fleshed out over time – development of the detailed tests and of the code will proceed in tandem.

The implementation process is also likely to uncover flawed thinking or ambiguity that goes right back to the requirements definition. In many cases the requirements will need to be revised to recognise factors which had been overlooked. This should not be seen as a failure of the requirements gathering process. Rather it is a feature of the development model that problems can be identified, turned around and resolved before time is wasted committing flawed designs to code.

Finally, it is not uncommon for unforeseen issues to arise during the deployment phase. Smaller issues will be fed back into the test suite to drive bug fixes and to ensure that the same problem does not reoccur in future deployments. Larger problems will feed into the requirements definition for the next iteration.

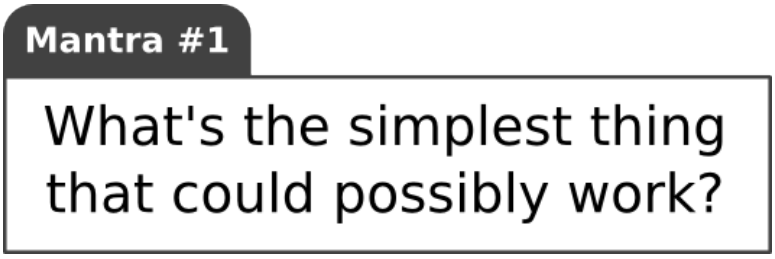
To summarise the “more realistic TDD cycle”:

- a number of feedback loops exist within the main cycle
- review and refinement of requirements can be triggered by the test development, the code development and the deployment phases
- development of detailed tests is driven at a high level by the initial requirements and at a more detailed level by the demands of the code development phase; feedback from the deployment phase is also possible
- nothing feeds into the code development phase except the tests; no code gets written until a test exists to confirm the behaviour of the code

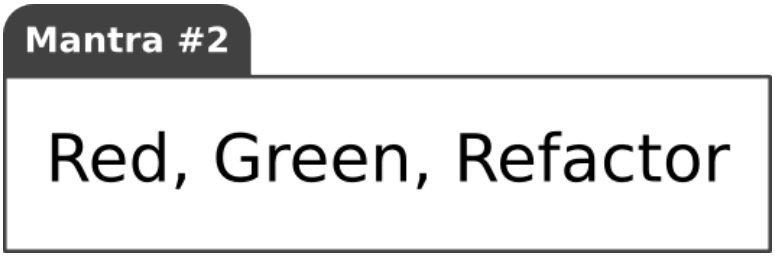
One common misconception with Test Driven Development is that there is no design. Designing a solution is the process of translating the 'what' into the 'how'. The requirements determine what a system must do, a design determines how it will do it. The design process will typically explore data flows, interfaces, object classes and the interactions between classes. Tests are used to support this exploration and to document in a concrete way how the different parts of the system will behave and how they will interact. This is the skeleton of the test suite which will be fleshed out by the detailed unit tests which follow. Because the design is embodied in the tests it is impossible for the two to get out of sync.

The Developer Workflow

Two mantras underpin a test-driven developer's workflow



This mantra stresses the importance of keeping the code under control and not letting it get ahead of the tests. Code should only be written to 'fix' a failing test. The code should do only what is required to make the test pass and nothing more. This is important because as long as there are no tests for extra functionality, the correctness of any extra code cannot be assured.



The second mantra describes the cycle through which code evolves. First a failing test is written (the tests go 'Red'). Next code is written to make the tests pass (the tests go 'Green'). Now the developer can take the opportunity to refactor the code (tidy it up, remove duplication, improve the

readability etc.) safe in the knowledge that if a bug is inadvertently introduced while making these changes, the tests will go 'Red' to highlight the problem. The constant reassessment of the code and the safety net provided by the tests encourage the developer to take more ownership of the code quality and to embrace change rather than resisting it.

A Worked Example

In this example I will follow Test Driven Development practices to develop a module to form up paragraphs from ragged lines of text. It will take plain text input in a form like this:

```
Lorem ipsum dolor sit amet,
consectetuer adipiscing elit. Vestibulum
leo
lorem, accumsan id, gravida nec.

Curabitur turpis
lacus, dignissim vitae, facilisis vel, blandit nec, mi. Donec mi.
Nunc facilisis sem
vitae justo. Vivamus a leo in velit mattis.
```

And produce output in a form like this:

```
Lorem ipsum dolor sit amet, consectetuer adipiscing
elit. Vestibulum leo lorem, accumsan id, gravida nec.

Curabitur turpis lacus, dignissim vitae, facilisis
vel, blandit nec, mi. Donec mi. Nunc facilisis sem
vitae justo. Vivamus a leo in velit mattis.
```

To keep the scope simple for this example:

- output line length will be specified as a number of (fixed-width) characters
- input will be a single text string, output will be another single text string
- blank lines in the input string will be used to delimit paragraphs
- output lines will be wrapped at word boundaries where possible
- naive hyphenation will be used to split words too long to be wrapped

A Design Outline

I have decided to implement the required functionality as a Perl class called Text::Para. To format text a programmer would create an instance of the Text::Para class, specify format parameters (initially just characters-per-line) and then call the object's format method – passing it a string of text and getting back a string of formatted text.

The First Test

In Test Driven Development before writing any code I must write a test. By convention, test scripts for Perl modules live in a directory called 't' and the files are named in such a way that the first tests to run check basic functionality and later tests check more advanced functions. I'll start by creating a file called `t/01_basics.t` with this content:

```
use strict;
use warnings;

use Test::More 'no_plan';

use_ok('Text::Para');
```

A Perl test script is just a series of assertions. In English it might be something like “perform action X, assert that the result was value Y”. In Perl a variety of assertion functions are provided by the `Test::More` module and a host of other modules on CPAN. In this example, `use_ok()` is an assertion function that takes a module name, loads that module and asserts that no errors occurred.

Apart from the standard Perl strict & warnings pragmata, the only part of the test script still needing explanation is the test plan. In this case I've specified `'no_plan'` meaning the test script contains an unspecified number of assertions. If a test script makes say 20 assertions, it would be possible for an error in the script to cause it to exit after the first 10 assertions had run. Without a test plan, the Perl test harness would see 10 successful assertions and no failed assertions, so the script would be marked as having passed. You can guard against such false positives by declaring at the outset how many tests you plan to run. Typically you would start with `'no_plan'` and go back and change it to something like `tests => 20` when you've written your last assertion.

Now since I'm doing TDD, I'll run the test and watch it fail spectacularly. Since the tests script is just a Perl script, I could run it with: `perl t/01_basics.t` however a better strategy is to use the prove utility which uses the test harness to capture the script's output, evaluate the results against the plan and produce a summary output:

```
$ prove t/01_basics.t
t/01_basics....
#   Failed test 'use Text::Para;'
#   in t/01_basics.t at line 6.
#     Tried to use 'Text::Para'.
#     Error:  Can't locate Text/Para.pm in @INC (@INC contains:
/etc/perl /usr/local/lib/perl/5.8.8 /usr/local/share/perl/5.8.8
/usr/lib/perl5 /usr/share/perl5 /usr/lib/perl/5.8 /usr/share/perl/5.8
/usr/local/lib/site_perl .) at (eval 3) line 2.
# BEGIN failed--compilation aborted at t/01_basics.t line 6.
# Looks like you failed 1 test of 1.
t/01_basics....dubious
      Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
      Failed 1/1 tests, 0.00% okay
Failed Test   Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------
t/01_basics.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
```

It's fairly clear from that mess that the test failed. The status is now 'Red'.

Some Code at Last

To make the test pass, I must write some code. Keeping in mind the first mantra of TDD I ask myself “what's the simplest thing that could possibly work”. I create the file `Text/Para.pm`:

```
package Text::Para;

use strict;
use warnings;

1;
```

Now at last I can run the test and watch it pass:

```
$ prove t/01_basics.t
t/01_basics....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs (0.02 cusr + 0.00 csys = 0.02 CPU)
```

Having first seen the test fail, then written the code and finally watched the test pass, I can be sure that the test is passing because of the code I have written. Although it seems trivial, this is an important ritual as it helps ensure my tests are doing what I think they're doing.

The status is now 'Green'. At this stage my code hasn't had a chance to gather too much cruft so I'll skip the refactoring and move on to the next test.

Adding a Constructor

Now that my module is loading successfully I'll add a couple of lines to my test script to test the constructor:

```
use strict;
use warnings;

use Test::More 'no_plan';# tests => 1;

use_ok('Text::Para');

my $formatter = Text::Para->new;
isa_ok($formatter, 'Text::Para');
```

and watch it fail:

```
t$ prove t/01_basics.t

t/01_basics....Can't locate object method "new" via package "Text::Para"
at t/01_basics.t line 8.
```

I could write my own constructor, but these days the smart Perl programmers are using the Moose object framework. Simply using the Moose module gives me a default constructor:

```
package Text::Para;

use strict;
use warnings;

use Moose;

1;
```

which is enough to make the tests pass:

```
$ prove t/01_basics.t
t/01_basics....ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs (0.57 cusr + 0.00 csys =  0.57 CPU)
```

It turns out that using Moose also turns on both strict and warnings so I can refactor them away:

```
package Text::Para;

use Moose;

1;
```

and running the tests again confirms I didn't break anything.

Implement the 'columns' Attribute

I want my formatter object to have a configurable attribute (or 'property') for the number of columns in the formatted text output. This attribute will be exposed via an accessor method, so I write a test for the accessor:

```
can_ok($formatter, 'columns');
```

once I've seen the test fail, I use the Moose syntax to declare an attribute called columns:

```
package Text::Para;

use Moose;

has 'columns' => (is => 'rw', isa => 'Int');

1;
```

I want a default width of 72 columns so I add my test:

```
is($formatter->columns, 72, 'default column width is 72');
```

and then my implementation:

```
has 'columns' => (is => 'rw', isa => 'Int', default => 72);
```

This 'is' assertion function used in this test compares two values – the value being tested and the value we expect. The optional third argument is a description for the test which will be printed out in the event of a failure, this is particularly useful for identifying regressions.

Now I need to test that I can set a new value for 'columns':

```
$formatter = Text::Para->new(columns => 10);
is($formatter->columns, 10, 'initialised column width to 10');
```

This time when I run the tests it doesn't fail – the Moose accessor method already implements this functionality. Just to be quite sure, I'll test changing the value a second time and confirm that the new value overwrites the old one:

```
$formatter->columns(8);
is($formatter->columns, 8, 'successfully set column width to 8');
```

Sure enough, the tests still pass:

```
$ prove t/01_basics.t
t/01_basics....ok
All tests successful.
Files=1, Tests=6,  0 wallclock secs (0.57 cusr + 0.00 csys =  0.57 CPU)
```

The 'format' Method

I now have a formatter class and an object to hold the formatting parameters. The next step is to add a `format` method to do the real work. This method will take a string of ragged lines as input and return a string of formatted paragraphs. At the most basic level, if I pass the method an empty string, I expect it to return an empty string:

```
is(
  $formatter->format(""),
  "",
  'empty string formatted correctly'
);
```

and of course it fails:

```
$ prove t/01_basics.t
t/01_basics....ok 1/0Can't locate object method "format" via package
"Text::Para" at t/01_basics.t line 20.
```

Going back to our first mantra, the simplest thing that could possibly work is a method that simply returns its input as output:

```
package Text::Para;

use Moose;

has 'columns' => (is => 'rw', isa => 'Int', default => 72);

sub format {
    my($self, $text) = @_;
    return $text;
}
```

and now the tests pass.

As you may recall from the earlier tests, the paragraph width is currently set to 8 columns. So if I pass in a line less than 8 columns I should get the same thing back:

```
is(
  $formatter->format('one two'),
  'one two',
  'short line formatted correctly'
);
```

and our stub implementation passes this test too.

Handling Whitespace

The next thing the formatter needs to do is discard leading and trailing whitespace. So I add tests for both:

```
is(
  $formatter->format(" one"),
  "one",
  'leading space was stripped'
);

is(
  $formatter->format(" one "),
  "one",
  'trailing space was stripped too'
);
```

and then an implementation:

```
sub format {
    my($self, $text) = @_;

    $text =~ s/\A\s+//;
    $text =~ s/\s+\Z//;
    return $text;
}
```

I also want to remove extra spaces between words – one is enough:

```
is(
  $formatter->format("one  two"),
  "one two",
  'extra internal whitespace handled correctly'
);
```

and then a simple implementation:

```
sub format {
    my($self, $text) = @_;

    $text =~ s/\A\s+//;
    $text =~ s/\s+\Z//;
    my @words = split /\s+/, $text;
    return join ' ', @words;
}
```

Now I have the tests passing again, it's time to do some refactoring. It's pretty clear that if I ignore the whitespace and just extract a list of non-whitespace 'words', I can do it with less code:

```
sub format {
    my($self, $text) = @_;

    my @words = $text =~ m/(\S+)/g;
    return join ' ', @words;
}
```

I re-run the tests and confirm they still pass.

Word Wrap

The next step is to handle long lines by inserting newlines between words. Remembering that the paragraph width is still set to 8 columns, I add a test for a longer line:

```
is(
  $formatter->format('one two three'),
  "one two\nthree",
  'third word was wrapped correctly'
);
```

which of course fails:

```
#   Failed test 'third word was wrapped correctly'
#   in t/01_basics.t at line 50.
#          got: 'one two three'
#     expected: 'one two
# three'
```

My formatter code now needs to get quite a bit smarter. I need to throw away the simple call to join and replace it with a loop that builds up lines a word at a time:

```
sub format {
    my($self, $text) = @_;

    my @words     = $text =~ m/(\S+)/g;
    my $para      = '';
    my $cols_left = $self->columns;

    foreach my $word (@words) {
        my $word_length = length $word;
        if($cols_left > $word_length) {
            $para .= ' ' if $cols_left < $self->columns;
            $para .= $word;
            $cols_left -= $word_length;
        }
    }

    return $para;
}
```

As it stands, this code doesn't do what I want, it simply stops adding words when it reaches the end of the first line:

```
#   Failed test 'third word was wrapped correctly'
#   in t/01_basics.t at line 50.
#          got: 'one two'
#     expected: 'one two
# three'
```

but the important point is that I've had to change quite a few lines to get to this point and yet I'm able to confirm that I haven't broken any of the earlier tests. The only one that's failing is the last one which expects to see a line break, so I can now add in the '`else`' branch for that:

```
        if($cols_left > $word_length) {
            $para .= ' ' if $cols_left < $self->columns;
            $para .= $word;
            $cols_left -= $word_length;
        }
        else {
            $para .= "\n$word";
            $cols_left = $self->columns - $word_length;
        }
```

and now the tests pass.

Edge Cases

Coding errors commonly occur around the conditional sections of code. I'm already testing lines that go to one character before the column count so I'll add a test for lines that exactly match the column count and another for lines that are one character over:

```
is(
  $formatter->format('one two three go'),
  "one two\nthree go",
  'packing to exactly the end of the line worked'
);

is(
  $formatter->format('one two three go!'),
  "one two\nthree\ngo!",
  'packing to just past the end of the line worked'
);
```

Running the test script shows that these tests pass so the status is still 'Green':

```
$ prove t/01_basics.t
t/01_basics....ok
All tests successful.
Files=1, Tests=14,  0 wallclock secs (0.54 cusr + 0.00 csys = 0.54 CPU)
```

Breaking Long Words

The one remaining edge case is the situation where a single word is longer than the column count. The code as it stands will never add a line break within a word. A clever formatter would insert a hyphen between syllables but for this simple example I'm just going to add one as the last character on the line. So I add a test for this situation:

```
is(
  $formatter->format('one two three fourfivesix'),
  "one two\nthree\nfourfiv-\nesix",
  'long word was broken correctly'
);
```

I watch the test fail:

```
#   Failed test 'long word was broken correctly'
#   in t/01_basics.t at line 68.
#          got: 'one two
# three
# fourfivesix'
#     expected: 'one two
# three
# fourfiv-
# esix'
```

and then implement the fix:

```
    while(@words) {
        my $word = shift @words;
        my $word_length = length $word;
        if($cols_left > $word_length) {
            $para .= ' ' if $cols_left < $self->columns;
            $para .= $word;
            $cols_left -= $word_length;
        }
        elsif($word_length <= $self->columns) {
            $para .= "\n$word";
            $cols_left = $self->columns - $word_length;
        }
        else {
            my $part = substr $word, 0, $self->columns - 1, '';
            $para .= "\n$part-\n";
            $cols_left = $self->columns;
            unshift @words, $word if length $word;
        }
    }
```

Note I've changed the main loop from a `for` loop to a `while` loop so that I can take the part of the word that followed the hyphen and put it back into `@words` to be processed next time around.

Running the tests confirms that the status is back to 'Green' so I have an opportunity to refactor. One thing that could be improved is the else block I've just added. Factoring some of this code out into its own method would have two benefits. First, by introducing a method name (say `split_at`) the intent of the code in the `else` block will be made clearer. Second, putting the naive word splitting logic into its own method will allow a subclass to override it with a smarter algorithm:

```
        else {
            my($part, $rest) = $self->split_at($word, $self->columns);
            $para .= "\n$part\n";
            $cols_left = $self->columns;
            unshift @words, $rest if length $rest;
        }
# ...

sub split_at {
    my($self, $word, $chars) = @_;

    my $part = substr $word, 0, $chars - 1, '';
    return "$part-", $word;
}
```

Often moving a chunk of code out into its own method will make the original method shorter. In this case we don't get that advantage.

The tests confirm that the refactoring didn't break anything, so I move on to remove the repeated calls to `$self->columns:`

```
    my $columns   = $self->columns;
    my $cols_left = $columns;

    while(@words) {
        my $word = shift @words;
        my $word_length = length $word;
        if($cols_left > $word_length) {
            $para .= ' ' if $cols_left < $columns;
            $para .= $word;
            $cols_left -= $word_length;
        }
        elsif($word_length <= $columns) {
            $para .= "\n$word";
            $cols_left = $columns - $word_length;
        }
        else {
            my($part, $rest) = $self->split_at($word, $columns);
            $para .= "\n$part\n";
            $cols_left = $columns;
            unshift @words, $rest if length $rest;
        }
    }
```

Once again I run the tests and confirm nothing was broken.

Formatting Paragraphs

The final missing feature is paragraph handling. The code currently treats all whitespace as equal and returns one formatted paragraph. What I need to do is treat a blank line as a paragraph separator. So I add a test for that:

```
is(
  $formatter->format("one two three\n\nfour five six"),
  "one two\nthree\n\nfour\nfive six",
  'paragraphs handled correctly'
);
```

The implementation is fairly simple. I just rename my existing `format` method to `format_para` and make a new `format` method that handles the blank lines:

```
sub format {
    my($self, $text) = @_;

    return join "\n\n",
           map { $self->format_para($_) }
           split /\n\n+/, $text;
}

sub format_para {
    my($self, $text) = @_;
```

It would be even better if the new format method ignored spaces on the blank lines between paragraphs, so I add a test for that:

```
is(
  $formatter->format("one two\n three\n  \nfour five six"),
  "one two\nthree\n\nfour\nfive six",
  'whitespace between paragraphs handled correctly'
);
```

and then an implementation to make the test pass:

```
sub format {
    my($self, $text) = @_;

    return join "\n\n",
           map { $self->format_para($_) }
           split /\n(?:[ \t]*\n)+/, $text;
}
```

And that's it. The code now meets the original specification and has a fairly exhaustive test suite to prove it.

Code Coverage

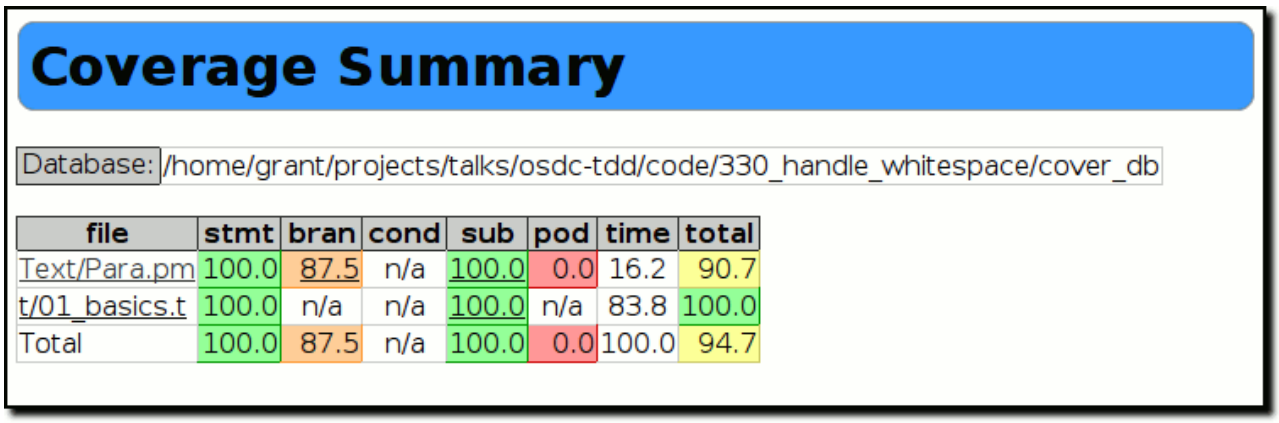
I described the test suite as 'fairly exhaustive' but it would be good to get an objective measure of how much of the code is being tested. The `Devel::Cover` module from CPAN can do exactly that. By running the test script via `Devel::Cover` metering is added to count how many times each line of code is executed and which conditional branches are executed:

```
$  perl -MDevel::Cover t/01_basics.t
```

and then the `cover` utility is run to generate HTML reports:

```
$ cover
```

The summary report shows that the test suite achieves 100% statement coverage of the `Text::Para` module, meaning that every line of code is executed at least once. The branch coverage is only 87.5% however. This means that not all conditional branches were executed.

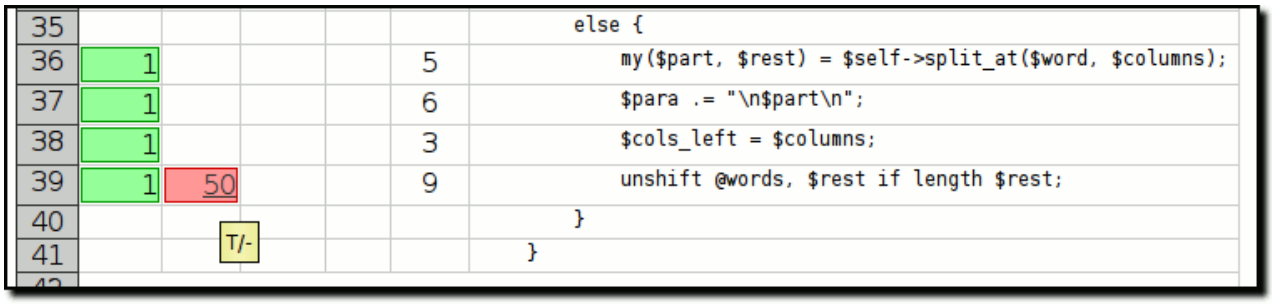


Coverage Summary

Database: /home/grant/projects/talks/osdc-tdd/code/330_handle_whitespace/cover_db

file	stmt	bran	cond	sub	pod	time	total
Text/Para.pm	100.0	87.5	n/a	100.0	0.0	16.2	90.7
t/01_basics.t	100.0	n/a	n/a	100.0	n/a	83.8	100.0
Total	100.0	87.5	n/a	100.0	0.0	100.0	94.7

By drilling down into the report:



... it is revealed that one conditional branch is only exercised in the mode where the condition is true. This can mean one of two things: either the test suite is inadequate and another test case is required; or the condition being checked cannot possibly occur. In this case, the conditional is checking whether there are any characters left after splitting a word to insert a hyphen. After a little analysis it becomes clear that this condition will always be true since if there were no characters left over it would not have been necessary to split the word. Simply removing the conditional boosts the branch coverage up to 100%.

The other useful metric on the summary page is the POD coverage – the proportion of public methods which have embedded POD documentation. Documenting the methods is an important job but outside the scope of this article.

A Final Review

So 100% of the tests are passing and 100% of the code is tested. Does this mean that the code is 100% correct? Sadly the answer is no, as illustrated by this extra test case:

```
$formatter->columns(8);
is(
  $formatter->format('abcdefgh'),
  "abcdefgh",
  'packing one full-length word worked'
);
```

which produces this result:

```
#   Failed test 'packing one full-length word worked'
#   in t/01_basics.t at line 87.
#          got: '
# abcdefgh'
#     expected: 'abcdefgh'
```

I'll leave it as an exercise for the reader to track down the cause of the problem. But suffice to say there is always more testing you *could* do, however economic and time constraints will usually be the limiting factors.

In this (admittedly trivial) example I've demonstrated some of the key features of a Test Driven Development process. The essential feature is that no code is written until a test exists to confirm the new functionality. Developers are encouraged to regularly review and refactor their code. The tests provide a safety net to catch any regressions the refactoring might introduce. The regular refactoring helps developers to become more bold about making changes to existing code and also to be less resistant to change dictated by new requirements.