NAME

Test::Builder::Tester - test testsuites that have been built with Test::Builder

SYNOPSIS

```
use Test::Builder::Tester tests => 1;
use Test::More;

test_out("not ok 1 - foo");
test_fail(+1);
fail("foo");
test_test("fail works");
```

DESCRIPTION

A module that helps you test testing modules that are built with *Test::Builder*.

The testing system is designed to be used by performing a three step process for each test you wish to test. This process starts with using `test_out` and `test_err` in advance to declare what the testsuite you are testing will output with *Test::Builder* to stdout and stderr.

You then can run the test(s) from your test suite that call *Test::Builder*. At this point the output of *Test::Builder* is safely captured by *Test::Builder::Tester* rather than being interpreted as real test output.

The final stage is to call `test_test` that will simply compare what you predeclared to what *Test::Builder* actually outputted, and report the results back with a "ok" or "not ok" (with debugging) to the normal output.

Functions

These are the six methods that are exported as default.

test_out

test_err

Procedures for predeclaring the output that your test suite is expected to produce until `test_test` is called. These procedures automatically assume that each line terminates with "\n". So

```
test_out("ok 1","ok 2");
```

is the same as

```
test_out("ok 1\nok 2");
```

which is even the same as

```
test_out("ok 1");
test_out("ok 2");
```

Once `test_out` or `test_err` (or `test_fail` or `test_diag`) have been called, all further output from *Test::Builder* will be captured by *Test::Builder::Tester*. This means that you will not be able perform further tests to the normal output in the normal way until you call `test_test` (well, unless you manually meddle with the output filehandles)

test_fail

Because the standard failure message that *Test::Builder* produces whenever a test fails will be a common occurrence in your test error output, and because it has changed between Test::Builder versions, rather than forcing you to call `test_err` with the string all the time like so

```
test_err("# Failed test ($0 at line ".line_num(+1).")");
```

`test_fail` exists as a convenience function that can be called instead. It takes one argument, the offset from the current line that the line that causes the fail is on.

```
test_fail(+1);
```

This means that the example in the synopsis could be rewritten more simply as:

```
test_out("not ok 1 - foo");
test_fail(+1);
fail("foo");
test_test("fail works");
```

test_diag

As most of the remaining expected output to the error stream will be created by *Test::Builder*'s `diag` function, *Test::Builder::Tester* provides a convenience function `test_diag` that you can use instead of `test_err`.

The `test_diag` function prepends comment hashes and spacing to the start and newlines to the end of the expected output passed to it and adds it to the list of expected error output. So, instead of writing

```
test_err("# Couldn't open file");
```

you can write

```
test_diag("Couldn't open file");
```

Remember that *Test::Builder*'s diag function will not add newlines to the end of output and test_diag will. So to check

```
Test::Builder->new->diag("foo\n","bar\n");
```

You would do

```
test_diag("foo","bar")
```

without the newlines.

test_test

Actually performs the output check testing the tests, comparing the data (with eq) that we have captured from *Test::Builder* against what was declared with `test_out` and `test_err`.

This takes name/value pairs that effect how the test is run.

title (synonym 'name', 'label')

The name of the test that will be displayed after the `ok` or `not ok`.

skip_out

Setting this to a true value will cause the test to ignore if the output sent by the test to the output stream does not match that declared with `test_out`.

skip_err

Setting this to a true value will cause the test to ignore if the output sent by the test to the error stream does not match that declared with `test_err`.

As a convenience, if only one argument is passed then this argument is assumed to be the name of the test (as in the above examples.)

Once `test_test` has been run test output will be redirected back to the original filehandles that *Test::Builder* was connected to (probably STDOUT and STDERR,) meaning any further

tests you run will function normally and cause success/errors for *Test::Harness*.

line_num

A utility function that returns the line number that the function was called on. You can pass it an offset which will be added to the result. This is very useful for working out the correct text of diagnostic functions that contain line numbers.

Essentially this is the same as the `__LINE__` macro, but the `line_num(+3)` idiom is arguably nicer.

In addition to the six exported functions there exists one function that can only be accessed with a fully qualified function call.

color

When `test_test` is called and the output that your tests generate does not match that which you declared, `test_test` will print out debug information showing the two conflicting versions. As this output itself is debug information it can be confusing which part of the output is from `test_test` and which was the original output from your original tests. Also, it may be hard to spot things like extraneous whitespace at the end of lines that may cause your test to fail even though the output looks similar.

To assist you `test_test` can colour the background of the debug information to disambiguate the different types of output. The debug output will have its background coloured green and red. The green part represents the text which is the same between the executed and actual output, the red shows which part differs.

The `color` function determines if colouring should occur or not. Passing it a true or false value will enable or disable colouring respectively, and the function called with no argument will return the current setting.

To enable colouring from the command line, you can use the *Text::Builder::Tester::Color* module like so:

```
perl -Mlib=Text::Builder::Tester::Color test.t
```

Or by including the *Test::Builder::Tester::Color* module directly in the PERL5LIB.

BUGS

Test::Builder::Tester does not handle plans well. It has never done anything special with plans. This means that plans from outside Test::Builder::Tester will effect Test::Builder::Tester, worse plans when using Test::Builder::Tester will effect overall testing. At this point there are no plans to fix this bug as people have come to depend on it, and Test::Builder::Tester is now discouraged in favor of `Test2::API::intercept()`. See *https://github.com/Test-More/test-more/issues/667*

Calls `Test::Builder->no_ending` turning off the ending tests. This is needed as otherwise it will trip out because we've run more tests than we strictly should have and it'll register any failures we had that we were testing for as real failures.

The color function doesn't work unless *Term::ANSIColor* is compatible with your terminal. Additionally, *Win32::Console::ANSI* must be installed on windows platforms for color output.

Bugs (and requests for new features) can be reported to the author though GitHub: *https://github.com/Test-More/test-more/issues*

AUTHOR

MAINTAINERS

Chad Granum <exodist@cpan.org>

NOTES

Thanks to Richard Clamp <richardc@unixbeard.net> for letting me use his testing system to try this module out on.

SEE ALSO

Test::Builder, *Test::Builder::Tester::Color*, *Test::More*.