NAME

Test::Builder - Backend for building test libraries

SYNOPSIS

```
package My::Test::Module;
use Test::Builder;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(ok);

my $Test = Test::Builder->new;
$Test->output('my_logfile');

sub import {
    my($self) = shift;
    my $pack = caller;

    $Test->exported_to($pack);
    $Test->plan(@_);

    $self->export_to_level(1, $self, 'ok');
}

sub ok {
    my($test, $name) = @_;

    $Test->ok($test, $name);
}
```

DESCRIPTION

Test::Simple and Test::More have proven to be popular testing modules, but they're not always flexible enough. Test::Builder provides the a building block upon which to write your own test libraries *which can work together*.

Construction

new

```
my $Test = Test::Builder->new;
```

Returns a Test::Builder object representing the current state of the test.

Since you only run one test per program `new` always returns the same Test::Builder object. No matter how many times you call new(), you're getting the same object. This is called a singleton. This is done so that multiple modules share such global information as the test counter and where test output is going.

If you want a completely new Test::Builder object different from the singleton, use `create`.

create

```
my $Test = Test::Builder->create;
```

Ok, so there can be more than one Test::Builder object and this is how you get it. You might use this instead of `new()` if you're testing a Test::Builder based module, but otherwise you probably want `new`.

NOTE: the implementation is not complete. `level`, for example, is still shared amongst **all**

Test::Builder objects, even ones created using this method. Also, the method name may change in the future.

reset

```
$Test->reset;
```

Reinitializes the Test::Builder singleton to its original state. Mostly useful for tests run in persistent environments where the same test might be run multiple times in the same process.

Setting up tests

These methods are for setting up tests and declaring how many there are. You usually only want to call one of these methods.

exported_to

```
my $pack = $Test->exported_to;
$Test->exported_to($pack);
```

Tells Test::Builder what package you exported your functions to. This is important for getting TODO tests right.

plan

```
$Test->plan('no_plan');
$Test->plan( skip_all => $reason );
$Test->plan( tests => $num_tests );
```

A convenient way to set up your tests. Call this and Test::Builder will print the appropriate headers and take the appropriate actions.

If you call plan(), don't call any of the other methods below.

expected_tests

```
my $max = $Test->expected_tests;
$Test->expected_tests($max);
```

Gets/sets the # of tests we expect this test to run and prints out the appropriate headers.

no_plan

```
$Test->no_plan;
```

Declares that this test will run an indeterminate # of tests.

has_plan

```
$plan = $Test->has_plan
```

Find out whether a plan has been defined. $plan is either `undef` (no plan has been set), `no_plan` (indeterminate # of tests) or an integer (the number of expected tests).

skip_all

```
$Test->skip_all;
$Test->skip_all($reason);
```

Skips all the tests, using the given $reason. Exits immediately with 0.

Running tests

These actually run the tests, analogous to the functions in Test::More.

$name is always optional.

ok

```
$Test->ok($test, $name);
```

Your basic test. Pass if $test is true, fail if $test is false. Just like Test::Simple's ok().

is_eq

```
$Test->is_eq($got, $expected, $name);
```

Like Test::More's is(). Checks if $got eq $expected. This is the string version.

is_num

```
$Test->is_num($got, $expected, $name);
```

Like Test::More's is(). Checks if $got == $expected. This is the numeric version.

isnt_eq

```
$Test->isnt_eq($got, $dont_expect, $name);
```

Like Test::More's isnt(). Checks if $got ne $dont_expect. This is the string version.

isnt_num

```
$Test->isnt_num($got, $dont_expect, $name);
```

Like Test::More's isnt(). Checks if $got ne $dont_expect. This is the numeric version.

like

```
$Test->like($this, qr/$regex/, $name);
$Test->like($this, '/$regex/', $name);
```

Like Test::More's like(). Checks if $this matches the given $regex.

You'll want to avoid qr// if you want your tests to work before 5.005.

unlike

```
$Test->unlike($this, qr/$regex/, $name);
$Test->unlike($this, '/$regex/', $name);
```

Like Test::More's unlike(). Checks if $this **does not match** the given $regex.

maybe_regex

```
$Test->maybe_regex(qr/$regex/);
$Test->maybe_regex('/$regex/');
```

Convenience method for building testing functions that take regular expressions as arguments, but need to work before perl 5.005.

Takes a quoted regular expression produced by qr//, or a string representing a regular expression.

Returns a Perl value which may be used instead of the corresponding regular expression, or undef if it's argument is not recognised.

For example, a version of like(), sans the useful diagnostic messages, could be written as:

```
sub laconic_like {
    my ($self, $this, $regex, $name) = @_;
    my $usable_regex = $self->maybe_regex($regex);
    die "expecting regex, found '$regex'\n"
        unless $usable_regex;
```

```
    $self->ok($this =~ m/$usable_regex/, $name);
}
```

cmp_ok

```
$Test->cmp_ok($this, $type, $that, $name);
```

Works just like Test::More's cmp_ok().

```
$Test->cmp_ok($big_num, '!=', $other_big_num);
```

BAIL_OUT

```
$Test->BAIL_OUT($reason);
```

Indicates to the Test::Harness that things are going so badly all testing should terminate. This includes running any additional test scripts.

It will exit with 255.

skip

```
$Test->skip;
$Test->skip($why);
```

Skips the current test, reporting $why.

todo_skip

```
$Test->todo_skip;
$Test->todo_skip($why);
```

Like skip(), only it will declare the test as failing and TODO. Similar to

```
print "not ok $tnum # TODO $why\n";
```

skip_rest

```
$Test->skip_rest;
$Test->skip_rest($reason);
```

Like skip(), only it skips all the rest of the tests you plan to run and terminates the test.

If you're running under no_plan, it skips once and terminates the test.

Test style

level

```
$Test->level($how_high);
```

How far up the call stack should $Test look when reporting where the test failed.

Defaults to 1.

Setting $Test::Builder::Level overrides. This is typically useful localized:

```
{
    local $Test::Builder::Level = 2;
    $Test->ok($test);
}
```

use_numbers

```
$Test->use_numbers($on_or_off);
```

Whether or not the test should output numbers. That is, this if true:

```
ok 1
ok 2
ok 3
```

or this if false

```
ok
ok
ok
```

Most useful when you can't depend on the test output order, such as when threads or forking is involved.

Defaults to on.

no_diag

```
$Test->no_diag($no_diag);
```

If set true no diagnostics will be printed. This includes calls to diag().

no_ending

```
$Test->no_ending($no_ending);
```

Normally, Test::Builder does some extra diagnostics when the test ends. It also changes the exit code as described below.

If this is true, none of that will be done.

no_header

```
$Test->no_header($no_header);
```

If set to true, no "1..N" header will be printed.

Output

Controlling where the test output goes.

It's ok for your test to change where STDOUT and STDERR point to, Test::Builder's default output settings will not be affected.

diag

```
$Test->diag(@msgs);
```

Prints out the given @msgs. Like `print`, arguments are simply appended together.

Normally, it uses the failure_output() handle, but if this is for a TODO test, the todo_output() handle is used.

Output will be indented and marked with a # so as not to interfere with test output. A newline will be put on the end if there isn't one already.

We encourage using this rather than calling print directly.

Returns false. Why? Because diag() is often used in conjunction with a failing test (`ok() || diag()`) it "passes through" the failure.

```
return ok(...) || diag(...);
```

_print

```
$Test->_print(@msgs);
```

Prints to the output() filehandle.

_print_diag

```
$Test->_print_diag(@msg);
```

Like _print, but prints to the current diagnostic filehandle.

output

```
$Test->output($fh);
$Test->output($file);
```

Where normal "ok/not ok" test output should go.

Defaults to STDOUT.

failure_output

```
$Test->failure_output($fh);
$Test->failure_output($file);
```

Where diagnostic output on test failures and diag() should go.

Defaults to STDERR.

todo_output

```
$Test->todo_output($fh);
$Test->todo_output($file);
```

Where diagnostics about todo test failures and diag() should go.

Defaults to STDOUT.

carp

```
$tb->carp(@message);
```

Warns with `@message` but the message will appear to come from the point where the original test function was called (`$tb->caller`).

croak

```
$tb->croak(@message);
```

Dies with `@message` but the message will appear to come from the point where the original test function was called (`$tb->caller`).

Test Status and Info

current_test

```
my $curr_test = $Test->current_test;
$Test->current_test($num);
```

Gets/sets the current test number we're on. You usually shouldn't have to set this.

If set forward, the details of the missing tests are filled in as 'unknown'. if set backward, the details of the intervening tests are deleted. You can erase history if you really want to.

summary

```
my @tests = $Test->summary;
```

A simple summary of the tests so far. True for pass, false for fail. This is a logical pass/fail, so todos are passes.

Of course, test #1 is $tests[0], etc...

details

```
my @tests = $Test->details;
```

Like summary(), but with a lot more detail.

```
$tests[$test_num - 1] =
        { 'ok'       => is the test considered a pass?
          actual_ok  => did it literally say 'ok'?
          name       => name of the test (if any)
          type       => type of test (if any, see below).
          reason     => reason for the above (if any)
        };
```

'ok' is true if Test::Harness will consider the test to be a pass.

'actual_ok' is a reflection of whether or not the test literally printed 'ok' or 'not ok'. This is for examining the result of 'todo' tests.

'name' is the name of the test.

'type' indicates if it was a special test. Normal tests have a type of ''. Type can be one of the following:

```
skip        see skip()
todo        see todo()
todo_skip   see todo_skip()
unknown     see below
```

Sometimes the Test::Builder test counter is incremented without it printing any test output, for example, when current_test() is changed. In these cases, Test::Builder doesn't know the result of the test, so it's type is 'unkown'. These details for these tests are filled in. They are considered ok, but the name and actual_ok is left undef.

For example "not ok 23 - hole count # TODO insufficient donuts" would result in this structure:

```
$tests[22] =    # 23 - 1, since arrays start from 0.
  { ok        => 1,   # logically, the test passed since it's
todo
    actual_ok => 0,   # in absolute terms, it failed
    name      => 'hole count',
    type      => 'todo',
    reason    => 'insufficient donuts'
  };
```

todo

```
my $todo_reason = $Test->todo;
my $todo_reason = $Test->todo($pack);
```

todo() looks for a $TODO variable in your tests. If set, all tests will be considered 'todo' (see Test::More and Test::Harness for details). Returns the reason (ie. the value of $TODO) if running as todo tests, false otherwise.

todo() is about finding the right package to look for $TODO in. It uses the exported_to() package to find it. If that's not set, it's pretty good at guessing the right package to look at based on $Level.

Sometimes there is some confusion about where todo() should be looking for the $TODO variable. If you want to be sure, tell it explicitly what $pack to use.

caller

```
    my $package = $Test->caller;
    my($pack, $file, $line) = $Test->caller;
    my($pack, $file, $line) = $Test->caller($height);
```

Like the normal caller(), except it reports according to your level().

_sanity_check

```
    $self->_sanity_check();
```

Runs a bunch of end of test sanity checks to make sure reality came through ok. If anything is wrong it will die with a fairly friendly error message.

_whoa

```
    $self->_whoa($check, $description);
```

A sanity check, similar to assert(). If the $check is true, something has gone horribly wrong. It will die with the given $description and a note to contact the author.

_my_exit

```
    _my_exit($exit_num);
```

Perl seems to have some trouble with exiting inside an END block. 5.005_03 and 5.6.1 both seem to do odd things. Instead, this function edits $? directly. It should ONLY be called from inside an END block. It doesn't actually exit, that's your job.

EXIT CODES

If all your tests passed, Test::Builder will exit with zero (which is normal). If anything failed it will exit with how many failed. If you run less (or more) tests than you planned, the missing (or extras) will be considered failures. If no tests were ever run Test::Builder will throw a warning and exit with 255. If the test died, even after having successfully completed all its tests, it will still be considered a failure and will exit with 255.

So the exit codes are...

```
0                   all tests successful
255                 test died or all passed but wrong # of tests run
any other number    how many failed (including missing or extras)
```

If you fail more than 254 tests, it will be reported as 254.

THREADS

In perl 5.8.1 and later, Test::Builder is thread-safe. The test number is shared amongst all threads. This means if one thread sets the test number using current_test() they will all be effected.

While versions earlier than 5.8.1 had threads they contain too many bugs to support.

Test::Builder is only thread-aware if threads.pm is loaded *before* Test::Builder.

EXAMPLES

CPAN can provide the best examples. Test::Simple, Test::More, Test::Exception and Test::Differences all use Test::Builder.

SEE ALSO

Test::Simple, Test::More, Test::Harness

AUTHORS

Original code by chromatic, maintained by Michael G Schwern <schwern@pobox.com>

COPYRIGHT