NAME

Test::Builder - Backend for building test libraries

SYNOPSIS

```
package My::Test::Module;
use base 'Test::Builder::Module';

my $CLASS = __PACKAGE__;

sub ok {
    my($test, $name) = @_;
    my $tb = $CLASS->builder;

    $tb->ok($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 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.

child

```
my $child = $builder->child($name_of_child);
$child->plan( tests => 4 );
$child->ok(some_code());
...
$child->finalize;
```

Returns a new instance of `Test::Builder`. Any output from this child will be indented four spaces more than the parent's indentation. When done, the `finalize` method *must* be called explicitly.

Trying to create a new child with a previous child still active (i.e., `finalize` not called) will `croak`.

Trying to run a test when you have an open child will also `croak` and cause the test suite to fail.

subtest

```
$builder->subtest($name, \&subtests);
```

See documentation of `subtest` in Test::More.

_plan_handled

```
if ( $Test->_plan_handled ) { ... }
```

Returns true if the developer has explicitly handled the plan via:

* Explicitly setting the number of tests

* Setting 'no_plan'

* Set 'skip_all'.

This is currently used in subtests when we implicitly call `$Test->done_testing` if the developer has not set a plan.

finalize

```
my $ok = $child->finalize;
```

When your child is done running tests, you must call `finalize` to clean up and tell the parent your pass/fail status.

Calling finalize on a child with open children will `croak`.

If the child falls out of scope before `finalize` is called, a failure diagnostic will be issued and the child is considered to have failed.

No attempt to call methods on a child after `finalize` is called is guaranteed to succeed.

Calling this on the root builder is a no-op.

parent

```
if ( my $parent = $builder->parent ) {
    ...
}
```

Returns the parent `Test::Builder` instance, if any. Only used with child builders for nested TAP.

name

```
diag $builder->name;
```

Returns the name of the current builder. Top level builders default to `$0` (the name of the executable). Child builders are named via the `child` method. If no name is supplied, will be named "Child of $parent->name".

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.

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.

If a child calls "skip_all" in the plan, a `Test::Builder::Exception` is thrown. Trap this error, call `finalize()` and don't run any more tests on the child.

```
 my $child = $Test->child('some child');
 eval { $child->plan( $condition ? ( skip_all => $reason ) : ( tests
=> 3 )  ) };
 if ( eval { $@->isa('Test::Builder::Exception') } ) {
    $child->finalize;
    return;
 }
 # run your tests
```

expected_tests

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

Gets/sets the number 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 number of tests.

_output_plan

```
  $tb->_output_plan($max);
  $tb->_output_plan($max, $directive);
  $tb->_output_plan($max, $directive => $reason);
```

Handles displaying the test plan.

If a `$directive` and/or `$reason` are given they will be output with the plan. So here's what skipping all tests looks like:

```
    $tb->_output_plan(0, "SKIP", "Because I said so");
```

It sets `$tb->{Have_Output_Plan}` and will croak if the plan was already output.

done_testing

```
  $Test->done_testing();
  $Test->done_testing($num_tests);
```

Declares that you are done testing, no more tests will be run after this point.

If a plan has not yet been output, it will do so.

$num_tests is the number of tests you planned to run. If a numbered plan was already declared, and if this contradicts, a failing test will be run to reflect the planning mistake. If `no_plan` was declared, this will override.

If `done_testing()` is called twice, the second call will issue a failing test.

If `$num_tests` is omitted, the number of tests run will be used, like no_plan.

`done_testing()` is, in effect, used when you'd want to use `no_plan`, but safer. You'd use it like so:

```
$Test->ok($a == $b);
$Test->done_testing();
```

Or to plan a variable number of tests:

```
for my $test (@tests) {
    $Test->ok($test);
}
$Test->done_testing(@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.

exported_to

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

Tells Test::Builder what package you exported your functions to.

This method isn't terribly useful since modules which share the same Test::Builder object might get exported to different packages and only the last one will be honored.

Running tests

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

They all return true if the test passed, false if the test failed.

`$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.

`undef` only ever matches another `undef`.

is_num

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

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

`undef` only ever matches another `undef`.

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`.

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`.

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);
```

Other Testing Methods

These are methods which are used in the course of writing a test but are not themselves tests.

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 building utility methods

These methods are useful when writing your own test methods.

maybe_regex

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

This method used to be useful back when Test::Builder worked on Perls before 5.6 which didn't have qr//. Now its pretty useless.

Convenience method for building testing functions that take regular expressions as arguments.

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 its 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);
}
```

_try

```
my $return_from_code          = $Test->try(sub { code });
my($return_from_code, $error) = $Test->try(sub { code });
```

Works like eval BLOCK except it ensures it has no effect on the rest of the test (ie. $@ is not set) nor is effected by outside interference (ie. `$SIG{__DIE__}`) and works around some quirks in older Perls.

`$error` is what would normally be in $@.

It is suggested you use this in place of eval BLOCK.

is_fh

```
my $is_fh = $Test->is_fh($thing);
```

Determines if the given `$thing` can be used as a filehandle.

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:

```
sub my_ok {
    my $test = shift;

    local $Test::Builder::Level = $Test::Builder::Level + 1;
    $TB->ok($test);
}
```

To be polite to other functions wrapping your own you usually want to increment `$Level` rather than set it to a constant.

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(...);
```

note

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

Like `diag()`, but it prints to the `output()` handle so it will not normally be seen by the user except in verbose mode.

explain

```
my @dump = $Test->explain(@msgs);
```

Will dump the contents of any references in a human readable format. Handy for things like...

```
is_deeply($have, $want) || diag explain $have;
```

or

```
is_deeply($have, $want) || note explain $have;
```

_print

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

Prints to the `output()` filehandle.

output

failure_output

todo_output

```
my $filehandle = $Test->output;
$Test->output($filehandle);
$Test->output($filename);
$Test->output(\$scalar);
```

These methods control where Test::Builder will print its output. They take either an open `$filehandle`, a `$filename` to open and write to or a `$scalar` reference to append to. It will always return a `$filehandle`.

output is where normal "ok/not ok" test output goes.

Defaults to STDOUT.

failure_output is where diagnostic output on test failures and `diag()` goes. It is normally not read by Test::Harness and instead is displayed to the user.

Defaults to STDERR.

`todo_output` is used instead of `failure_output()` for the diagnostics of a failing TODO

test. These will not be seen by the user.

Defaults to STDOUT.

reset_outputs

```
$tb->reset_outputs;
```

Resets all the output filehandles back to their defaults.

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.

is_passing

```
my $ok = $builder->is_passing;
```

Indicates if the test suite is currently passing.

More formally, it will be false if anything has happened which makes it impossible for the test suite to pass. True otherwise.

For example, if no tests have run `is_passing()` will be true because even though a suite with no tests is a failure you can add a passing test to it and start passing.

Don't think about it too much.

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 its type is 'unknown'. 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 its 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);
```

If the current tests are considered "TODO" it will return the reason, if any. This reason can come from a `$TODO` variable or the last call to `todo_start()`.

Since a TODO test does not need a reason, this function can return an empty string even when inside a TODO block. Use `$Test->in_todo` to determine if you are currently inside a TODO block.

`todo()` is about finding the right package to look for `$TODO` in. It's pretty good at guessing the right package to look at. It first looks for the caller based on `$Level + 1`, since `todo()` is usually called inside a test function. As a last resort it will use `exported_to()`.

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.

find_TODO

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

Like `todo()` but only returns the value of `$TODO` ignoring `todo_start()`.

Can also be used to set `$TODO` to a new value while returning the old value:

```
my $old_reason = $Test->find_TODO($pack, 1, $new_reason);
```

in_todo

```
my $in_todo = $Test->in_todo;
```

Returns true if the test is currently inside a TODO block.

todo_start

```
$Test->todo_start();
$Test->todo_start($message);
```

This method allows you declare all subsequent tests as TODO tests, up until the `todo_end` method has been called.

The `TODO:` and `$TODO` syntax is generally pretty good about figuring out whether or not we're in a TODO test. However, often we find that this is not possible to determine (such as when we want to use `$TODO` but the tests are being executed in other packages which can't be inferred beforehand).

Note that you can use this to nest "todo" tests

```
$Test->todo_start('working on this');
# lots of code
$Test->todo_start('working on that');
# more code
$Test->todo_end;
$Test->todo_end;
```

This is generally not recommended, but large testing systems often have weird internal needs.

We've tried to make this also work with the TODO: syntax, but it's not guaranteed and its use is also discouraged:

```
TODO: {
    local $TODO = 'We have work to do!';
    $Test->todo_start('working on this');
    # lots of code
    $Test->todo_start('working on that');
    # more code
    $Test->todo_end;
    $Test->todo_end;
}
```

Pick one style or another of "TODO" to be on the safe side.

`todo_end`

```
$Test->todo_end;
```

Stops running tests as "TODO" tests. This method is fatal if called without a preceding `todo_start` method call.

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()`.

`$height` will be added to the `level()`.

If `caller()` winds up off the top of the stack it report the highest context.

_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. 5.6.1 does some 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.

MEMORY

An informative hash, accessible via `<details()>`, is stored for each test you perform. So memory usage will scale linearly with each test run. Although this is not a problem for most test suites, it can become an issue if you do large (hundred thousands to million) combinatorics tests in the same run.

In such cases, you are advised to either split the test file into smaller ones, or use a reverse approach, doing "normal" (code) compares and triggering fail() should anything go unexpected.

Future versions of Test::Builder will have a way to turn history off.

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