

NAME

Test::Harness::Straps - detailed analysis of test results

SYNOPSIS

```
use Test::Harness::Straps;

my $strap = Test::Harness::Straps->new;

# Various ways to interpret a test
my %results = $strap->analyze($name, \@test_output);
my %results = $strap->analyze_fh($name, $test_filehandle);
my %results = $strap->analyze_file($test_file);

# UNIMPLEMENTED
my %total = $strap->total_results;

# Altering the behavior of the strap UNIMPLEMENTED
my $verbose_output = $strap->dump_verbose();
$strap->dump_verbose_fh($output_filehandle);
```

DESCRIPTION

THIS IS ALPHA SOFTWARE in that the interface is subject to change in incompatible ways. It is otherwise stable.

Test::Harness is limited to printing out its results. This makes analysis of the test results difficult for anything but a human. To make it easier for programs to work with test results, we provide Test::Harness::Straps. Instead of printing the results, straps provide them as raw data. You can also configure how the tests are to be run.

The interface is currently incomplete. *Please* contact the author if you'd like a feature added or something change or just have comments.

CONSTRUCTION

new()

```
my $strap = Test::Harness::Straps->new;
```

Initialize a new strap.

\$strap->_init

```
$strap->_init;
```

Initialize the internal state of a strap to make it ready for parsing.

ANALYSIS

\$strap->analyze(\$name, \@output_lines)

```
my %results = $strap->analyze($name, \@test_output);
```

Analyzes the output of a single test, assigning it the given \$name for use in the total report. Returns the %results of the test. See *Results*.

@test_output should be the raw output from the test, including newlines.

\$strap->analyze_fh(\$name, \$test_filehandle)

```
my %results = $strap->analyze_fh($name, $test_filehandle);
```

Like `analyze`, but it reads from the given filehandle.

\$strap->analyze_file(\$test_file)

```
my %results = $strap->analyze_file($test_file);
```

Like `analyze`, but it runs the given `$test_file` and parses its results. It will also use that name for the total report.

\$strap->_command_line(\$file)

Returns the full command line that will be run to test *\$file*.

\$strap->_command()

Returns the command that runs the test. Combine this with `_switches()` to build a command line.

Typically this is `$^X`, but you can set `$ENV{HARNESS_PERL}` to use a different Perl than what you're running the harness under. This might be to run a threaded Perl, for example.

You can also overload this method if you've built your own strap subclass, such as a PHP interpreter for a PHP-based strap.

\$strap->_switches(\$file)

Formats and returns the switches necessary to run the test.

\$strap->_cleaned_switches(@switches_from_user)

Returns only defined, non-blank, trimmed switches from the parms passed.

\$strap->_INC2PERL5LIB

```
local $ENV{PERL5LIB} = $self->_INC2PERL5LIB;
```

Takes the current value of `@INC` and turns it into something suitable for putting onto `PERL5LIB`.

\$strap->_filtered_INC()

```
my @filtered_inc = $self->_filtered_INC;
```

Shortens `@INC` by removing redundant and unnecessary entries. Necessary for OSes with limited command line lengths, like VMS.

\$strap->_restore_PERL5LIB()

```
$self->_restore_PERL5LIB;
```

This restores the original value of the `PERL5LIB` environment variable. Necessary on VMS, otherwise a no-op.

Parsing

Methods for identifying what sort of line you're looking at.

_is_diagnostic

```
my $is_diagnostic = $strap->_is_diagnostic($line, \%comment);
```

Checks if the given line is a comment. If so, it will place it into `$comment` (sans #).

_is_header

```
my $is_header = $strap->_is_header($line);
```

Checks if the given line is a header (1..M) line. If so, it places how many tests there will be in `$strap->{max}`, a list of which tests are todo in `$strap->{todo}` and if the whole test was skipped `$strap->{skip_all}` contains the reason.

_is_bail_out

```
my $is_bail_out = $strap->_is_bail_out($line, \$reason);
```

Checks if the line is a "Bail out!". Places the reason for bailing (if any) in `$reason`.

_reset_file_state

```
$strap->_reset_file_state;
```

Resets things like `$strap->{max}`, `$strap->{skip_all}`, etc. so it's ready to parse the next file.

Results

The `%results` returned from `analyze()` contain the following information:

| | |
|----------|--|
| passing | true if the whole test is considered a pass (or skipped), false if its a failure |
| exit | the exit code of the test run, if from a file |
| wait | the wait code of the test run, if from a file |
| max | total tests which should have been run |
| seen | total tests actually seen |
| skip_all | if the whole test was skipped, this will contain the reason. |
| ok | number of tests which passed (including todo and skips) |
| todo | number of todo tests seen |
| bonus | number of todo tests which unexpectedly passed |
| skip | number of tests skipped |

So a successful test should have `max == seen == ok`.

There is one final item, the details.

| | |
|---------|---|
| details | an array ref reporting the result of each test looks like this: |
|---------|---|

```
$results{details}[$test_num - 1] =
  { ok          => is the test considered ok?
    actual_ok   => did it literally say 'ok'?
    name        => name of the test (if any)
    diagnostics => test diagnostics (if any)
    type        => 'skip' or 'todo' (if any)
```

```
        reason      => reason for the above (if any)
    };
```

Element 0 of the details is test #1. I tried it with element 1 being #1 and 0 being empty, this is less awkward.

EXAMPLES

See *examples/mini_harness.plx* for an example of use.

AUTHOR

Michael G Schwern <schwern@pobox.com>, currently maintained by Andy Lester
<andy@petdance.com>.

SEE ALSO

Test::Harness