NAME

Test::Fatal – incredibly simple helpers for testing code with exceptions

VERSION

version 0.014

SYNOPSIS

```
use Test::More;
use Test::Fatal;

use System::Under::Test qw(might_die);

is(
  exception { might_die; },
  undef,
  "the code lived",
);

like(
  exception { might_die; },
  qr/turns out it died/,
  "the code died as expected",
);

isa_ok(
  exception { might_die; },
  'Exception::Whatever',
  'the thrown exception',
);
```

DESCRIPTION

Test::Fatal is an alternative to the popular Test::Exception. It does much less, but should allow greater flexibility in testing exception-throwing code with about the same amount of typing.

It exports one routine by default: `exception`.

FUNCTIONS

exception

```
my $exception = exception { ... };
```

`exception` takes a bare block of code and returns the exception thrown by that block. If no exception was thrown, it returns undef.

Achtung! If the block results in a *false* exception, such as 0 or the empty string, Test::Fatal itself will die. Since either of these cases indicates a serious problem with the system under testing, this behavior is considered a *feature*. If you must test for these conditions, you should use Try::Tiny's try/catch mechanism. (Try::Tiny is the underlying exception handling system of Test::Fatal.)

Note that there is no TAP assert being performed. In other words, no "ok" or "not ok" line is emitted. It's up to you to use the rest of `exception` in an existing test like `ok`, `isa_ok`, `is`, et cetera. Or you may wish to use the `dies_ok` and `lives_ok` wrappers, which do provide TAP output.

`exception` does *not* alter the stack presented to the called block, meaning that if the exception returned has a stack trace, it will include some frames between the code calling `exception` and the thing throwing the exception. This is considered a *feature* because it avoids the occasionally twitchy `Sub::Uplevel` mechanism.

Achtung! This is not a great idea:

```
sub exception_like(&$;$) {
    my ($code, $pattern, $name) = @_;
    like( &exception($code), $pattern, $name );
}

exception_like(sub { }, qr/foo/, 'foo appears in the exception');
```

If the code in the `...` is going to throw a stack trace with the arguments to each subroutine in its call stack (for example via `Carp::confess`, the test name, "foo appears in the exception" will itself be matched by the regex. Instead, write this:

```
like( exception { ... }, qr/foo/, 'foo appears in the exception' );
```

Achtung: One final bad idea:

```
isnt( exception { ... }, undef, "my code died!");
```

It's true that this tests that your code died, but you should really test that it died *for the right reason*. For example, if you make an unrelated mistake in the block, like using the wrong dereference, your test will pass even though the code to be tested isn't really run at all. If you're expecting an inspectable exception with an identifier or class, test that. If you're expecting a string exception, consider using `like`.

success

```
try {
  should_live;
} catch {
  fail("boo, we died");
} success {
  pass("hooray, we lived");
};
```

`success`, exported only by request, is a Try::Tiny helper with semantics identical to `finally`, but the body of the block will only be run if the `try` block ran without error.

Although almost any needed exception tests can be performed with `exception`, success blocks may sometimes help organize complex testing.

dies_ok

lives_ok

Exported only by request, these two functions run a given block of code, and provide TAP output indicating if it did, or did not throw an exception. These provide an easy upgrade path for replacing existing unit tests based on `Test::Exception`.

RJBS does not suggest using this except as a convenience while porting tests to use Test::Fatal's `exception` routine.

```
use Test::More tests => 2;
use Test::Fatal qw(dies_ok lives_ok);

dies_ok { die "I failed" } 'code that fails';

lives_ok { return "I'm still alive" } 'code that does not fail';
```

AUTHOR

Ricardo Signes <rjbs@cpan.org>

COPYRIGHT AND LICENSE