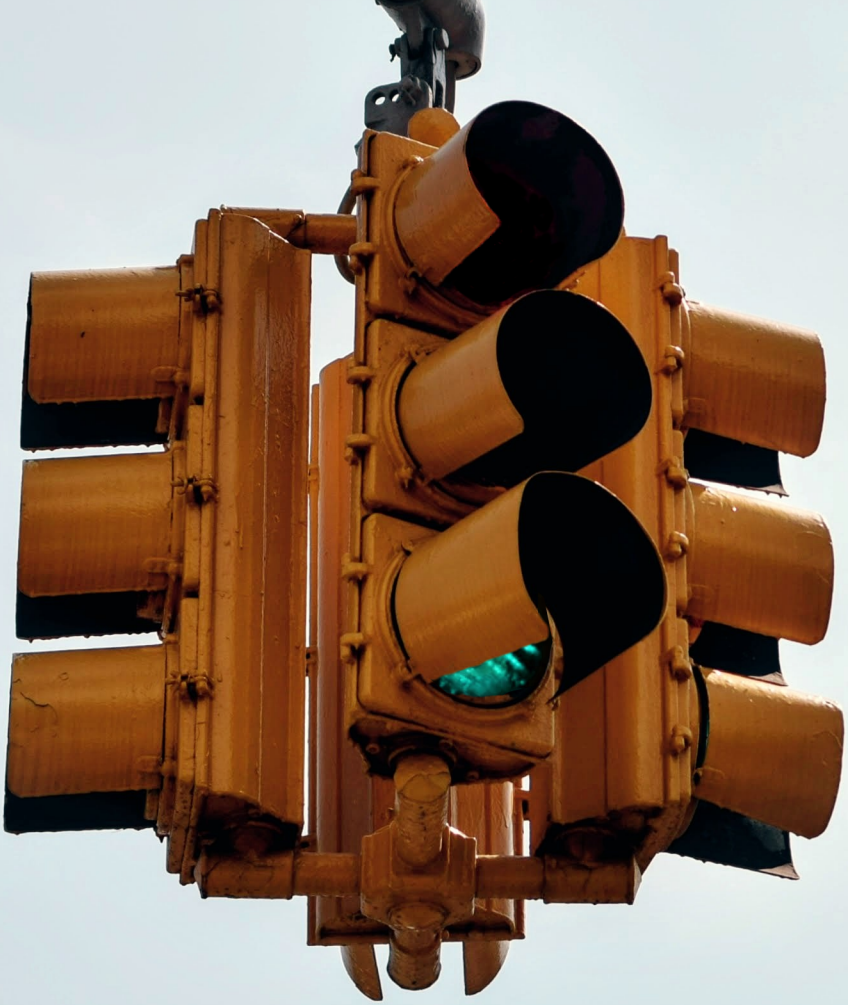

A practical guide to TESTING in Modern Perl

by Gregor Goldbach
and Viacheslav Tykhanovskyi
2018

A practical guide to testing in Modern Perl

Viacheslav Tykhanovskyi and Gregor Goldbach

This book is for sale at http://leanpub.com/perl-testing

This version was published on 2018-10-02

Contents

Introduction

Welcome to the fascinating world of testing!

Who should read this book

Perl developers who know that they should test their software, but just don't know how to do it.

What this book is about

This book teaches you how to test your Perl modules and applications in an automated fashion. It does this in a practical way with just about enough theory to get started. In addition to that, you'll learn *why* you should test software, but of course you know that already.

What this book is *not* about

We don't teach you all the testing theory there is. If you're preparing for an ISTQB certification, this book is not for you. We don't write about manual testing (which is of course needed aswell).

The purpose of testing

The prime purpose of testing is to find bugs.

To quote Edsger Wybe Dijkstra: “Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.” (Edsger Wybe Dijkstra, The Humble Programmer, ACM Turing Lecture 1972)

Each developer should be interested in testing, because by finding bugs we’re able to fix them. We thereby increase the internal quality of our code and have time for activities that are usually way more fun: implementing new features!

Having an automated suite of tests (and running it regularly of course) will help to prevent the bugs from reappearing in the future, which frustrates everybody. This makes changing code safer and in general makes developer happier.

Tests can serve also a documentation purpose. And comparing to normal documentation is always up-to-date and never lies.

When talking to business people (e.g. managers), we should change our vocabulary and speak the language they know and understand. By finding bugs we *reduce maintenance cost, increase both productivity and external quality*. And new features can be sold! Try to sell the time spent with bug hunting…

And even the user is experiencing the results of our testing. The product is now more often doing what it is supposed to do and is somehow *better*. Therefore the product makes the user’s life better since it actually solves problems instead of generating new ones.

Quick Start

Minimum prerequisites

Before starting to write the tests in Perl we have to have a perl interpreter. On most unix-derived systems it comes preinstalled. If not, check your OS package manager.

On Windows try installing Strawberry Perl[1].

It is recommended to install the latest perl available, something in the 20th versions. In addition to the perl the following modules have to be installed: Test2::Suite and Moo. The first one usually comes with modern perls, but some distributions package it separately (look for perl-modules or something like that in your package manager). It is recommended to use your OS package manager for installing the modules. If you cannot do that, take a look at cpanminus[2]. It can be installed locally with the following command:

```
curl -L https://cpanmin.us | perl - App::cpanminus
```

And then:

```
cpanm Test2::Suite Moo
```

Additionally you want to have a text editor for creating and changing the code and the tests. Any editor that you are familiar with would work, but vim[3] is the best one.

Testing a logging module

For those who haven't used testing and haven't had any issues it's hard to recognize the best practices. In this chapter we will try to test classes by using practical examples. This is a quick start for impatient. If you do not understand anything from this chapter, do not worry, we will explain in detail all the techniques later.

As an example we will take a logging modules, which can log to stderr and to a file. The module should support the following log levels: error, warn and debug, and use this output format: <date> <level> <mesage>.

Here is the initial module structure:

[1] http://strawberryperl.com/
[2] https://cpanmin.us/
[3] https://www.vim.org/

```
lib/
    Logger.pm
t/
    logger.t
```

Where `lib` is a module directory, and `t` is a test directory.

The minimum Perl class looks like this (we use Moo and friends here just for pure simplicity, but you can use any OOP framework of course):

```
1  package Logger;
2  use Moo;
3
4  1;
```

And here is our test:

```
1  use Test2::V0
2  use Logger;
3
4  subtest 'creates correct object' => sub {
5      isa_ok(Logger->new, 'Logger');
6  };
7
8  done_testing;
```

Let's run the tests using `prove`:

```
$ prove -l t
t/logger.t .. ok
All tests successful.
```

At this point all the tests pass. But of course in reality we don't test anything.

Let's implement the log level setting feature first. For example, we want the default log level to be `error`. Let's write the test:

```
subtest 'has default log level' => sub {
    my $logger = Logger->new;

    is $logger->level, 'error';
};
```

And now let's run it.

```
$ prove t
t/logger.t .. 1/? Can't locate object method "level" via package "Logger"
```

As we can see our logging class doesn't have such method. Let's add it:

```
package Logger;
use Moo;

sub level {
    my $self = shift;

    return 'error';
}

1;
```

Of course we don't want the method `level` to always return the same value, we want to save it somehow. We are going to write a test that will assure that the set value is saved.

```
subtest 'sets log level' => sub {
    my $logger = Logger->new;

    $logger->set_level('debug');

    is $logger->level, 'debug';
};
```

Run the tests:

```
$ prove t
t/logger.t .. 1/? Can't locate object method "set_level" via package "Logger"
```

Again, not such method. Let's add it:

```
sub set_level {
        my $self = shift;
}
```

Now the method exists, but the tests still fail:

```
t/logger.t .. 1/?
    #   Failed test at t/logger.t line 22.
    #          got: 'error'
    #     expected: 'debug'
    # Looks like you failed 1 test of 1.
```

Now the method `level` returns the wrong value. It's time to implement the saving feature.

```
package Logger;
use Moo;

sub set_level {
    my $self = shift;
    my ($new_level) = @_;

    $self->{level} = $new_level;
}

sub level {
    my $self = shift;

    return $self->{level};
}

1;
```

Run the tests:

```
$ prove
t/logger.t .. 1/?
    #   Failed test at t/logger.t line 14.
    #          got: undef
    #     expected: 'error'
    # Looks like you failed 1 test of 1.
```

Hm, looks like the new test started to work, but the old one broke. Here is an example how the tests can detect future issues, that would arrise when we modify the code. Of course in this case we forgot that the default level is `error`. Let's re-add this functionality.

```
sub level {
    my $self = shift;

    return $self->{level} || 'error';
}
```

Now all the tests pass.

The next thing that we're going to is to make sure that only the allowed log levels can be set. If we set an unknown log level we should get an exception. The new test will look like this:

```
subtest 'throws exception when invalid log level' => sub {
    my $log = Logger->new;

    ok dies { $log->set_level('unknown') };
};
```

Also we have to check that all correct log levels do not throw exceptions. In order to not to write a lot of similar test cases, we will check the value in a loop:

```
subtest 'not throws when known log level' => sub {
    my $log = Logger->new;

    for my $level (qw/error warn debug/) {
        ok lives { $log->set_level($level) };
    }
};
```

If we run the tests, they will fail as expected:

```
$ prove t
t/logger.t .. 1/?
    #   Failed test at t/logger.t line 29.
    # Looks like you failed 1 test of 1.
```

But the second test passes without any code. In this case we know why this happens, but in real life everything can be different. We write the test, we write the code. Then we run test and it passes. So a programmer thinks that everything is working fine. But in reality the implementation is wrong. It is always very important to run the test first to make sure it is failing and only then add new code.

Let's implement the log level list check:

```
package Logger;
use Moo;

use Carp qw(croak);
use List::Util qw(first);

sub set_level {
    my $self = shift;
    my ($new_level) = @_;

    croak('Unknown log level')
      unless first { $new_level eq $_ } qw/error warn debug/;

    $self->{level} = $new_level;
}

sub level {
    my $self = shift;

    return $self->{level} || 'error';
}

1;
```

For exceptions we use `croak` from `Carp` modules, because it gives a better error message and its source. For list search we're using `first` from `List::Util` module since compared to default `grep` it stops after the first match. For our task that's irrelevant, but still a good practice in general.

Now we write the test for the `log` method. It should print to `stderr`. Since the testing is automatic we have to catch the output somehow. For this we can use `Capture::Tiny` module which perfectly does the needed work.

```
use Capture::Tiny qw(capture_stderr);

subtest 'prints to stderr' => sub {
    my $log = Logger->new;

    my $stderr = capture_stderr {
        $log->log('error', 'message');
    };

    ok $stderr;
};
```

We run the test and see that it doesn't pass, so we implement the new functionality:

```
sub log {
    my $self = shift;
    my ($level, $message) = @_;

    print STDERR $message;
}
```

Now the tests pass, but this is of course not what we want, we want to have a formatted output, so for this we write another test:

```
subtest 'prints formatted line' => sub {
    my $log = Logger->new;

    my $stderr = capture_stderr {
        $log->log('error', 'message');
    };

    like $stderr, qr/\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d \[error\] message/;
};
```

Now the tests fail:

```
$ prove t
t/logger.t .. 1/?
    #   Failed test at t/logger.t line 58.
    #                   'message'
    #     doesn't match '(?^:\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d \[error\] message)'
```

For date formatting we will use `Time::Piece`. The code will look like this:

```
use Time::Piece;

sub log {
    my $self = shift;
    my ($level, $message) = @_;

    my $time = Time::Piece->new->strftime('%Y-%m-%d %T');
    print STDERR $time, " [$level] ", $message;
}
```

Now the tests pass.

Here we would have to implement a check that the correct `level` value is passed, but we can move this task to the compiler by adding a separate method for every level. Finally we will have to following tests:

```
subtest 'prints to stderr' => sub {
    my $log = Logger->new;

    my $stderr = capture_stderr {
        $log->error('message');
    };

    ok $stderr;
};

subtest 'prints formatted line' => sub {
    my $log = Logger->new;

    my $stderr = capture_stderr {
        $log->error('message');
    };

    like $stderr, qr/\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d \[error\] message/;
};
```

We should not forget that we have to check all combinations, so let's modify the tests again:

```
subtest 'prints to stderr' => sub {
    my $log = Logger->new;

    for my $level (qw/error warn debug/) {
        my $stderr = capture_stderr {
            $log->$level('message');
        };

        ok $stderr;
    }
};

subtest 'prints formatted line' => sub {
    my $log = Logger->new;

```

```
    for my $level (qw/error warn debug/) {
        my $stderr = capture_stderr {
            $log->$level('message');
        };

        like $stderr, qr/\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d \[$level\] message/;
    }
};
```

Let's implement the needed feature, hiding the `log` method:

```
sub error { shift->_log('error', @_) }
sub warn  { shift->_log('warn',  @_) }
sub debug { shift->_log('debug', @_) }

sub _log {
    my $self = shift;
    my ($level, $message) = @_;

    my $time = Time::Piece->new->strftime('%Y-%m-%d %T');
    print STDERR $time, " [$level] ", $message;
}
```

Looking at the code I found an error, we do not have a newline. Let's check if I am right but writing the following test:

```
subtest 'prints to stderr with \n' => sub {
    my $log = Logger->new;

    for my $level (qw/error warn debug/) {
        my $stderr = capture_stderr {
            $log->$level('message');
        };

        like $stderr, qr/\n$/;
    }
};
```

Ok, now we know that there is no newline and that we can reproduce the error. Now let's fix it:

```
print STDERR $time, " [$level] ", $message, "\n";
```

If we group the future bugs detected in our good to new test cases, in the future we can easily detect that we do not have any regressions, the bugs do not come to live, when we are fixing other bugs or adding new code. This is our protection.

So why did we implement log levels? Of course to generate output only when the appropriate level is set, which is higher than the current one. In other words in `debug` level we should get `error`, `warn` and `debug` messages. At level `warn` we should get `error` and `warn`, and finally at level `error` only `error`. In order to simplify the tests we will create a table with expected output values when using specific input cases.

For example, the test that checks that we get logging on appropriate level will look like this:

```
subtest 'logs when level is higher' => sub {
    my $log = Logger->new;

    my $levels = {
        error => [qw/error/],
        warn  => [qw/error warn/],
        debug => [qw/error warn debug/],
    };

    for my $level (keys %$levels) {
        $log->set_level($level);
        for my $test_level (@{$levels->{$level}}) {
            ok capture_stderr {
                $log->$test_level('message');
            };
        }
    }
};
```

The test that checks that the messages do not appear when the appropriate log level is set will look very similar:

```
subtest 'not logs when level is lower' => sub {
    my $log = Logger->new;

    my $levels = {
        error => [qw/warn debug/],
        warn  => [qw/debug/],
    };

    for my $level (keys %$levels) {
        $log->set_level($level);
        for my $test_level (@{$levels->{$level}}) {
            ok !capture_stderr {
                $log->$test_level('message');
            };
        }
    }
};
```

Here we do not have the `debug` level check, since in this mode everything is logged. Now we make sure that our tests fail and write the needed code. First of all, to every log level we assign its own value, the second, before logging we check if the level is matched.

When running tests you may find it hard to see what is failing, thus we can add a diagnostic message:

```
ok !capture_stderr {
    $log->$test_level('message');
}, "not log '$test_level' when '$level'";
```

Now the errors are much clearer:

```
    #   Failed test 'not log 'warn' when 'error''
    #   at t/logger.t line 109.
```

After adding the new code it looks like many of our old tests start to fail, for example this one:

```
subtest 'prints to stderr' => sub {
    my $log = Logger->new;

    for my $level (qw/error warn debug/) {
        my $stderr = capture_stderr {
            $log->$level('message');
        };

        ok $stderr;
    }
};
```

It's clear that we have to set the highest `debug` log level in other tests too.

Changed code:

```
my $LEVELS = {
    error => 1,
    warn  => 2,
    debug => 3
};

...

sub set_level {
    my $self = shift;
    my ($new_level) = @_;

    croak('Unknown log level')
      unless first { $new_level eq $_ } keys %$LEVELS;

    $self->{level} = $new_level;
}

sub _log {
    my $self = shift;
    my ($level, $message) = @_;

    return unless $LEVELS->{$level} <= $LEVELS->{$self->level};

    my $time = Time::Piece->new->strftime('%Y-%m-%d %T');
    print STDERR $time, " [$level] ", $message, "\n";
}
```

Since log creation and initialization in every test looks almost identical, let's extract it to a separate method, thus lowering the duplicated code.

```
sub _build_logger {
    my $logger = Logger->new;
    $logger->set_level('debug');
    return $logger;
}
```

Now the logger is fully tested. To make sure, let's run `Devel::Cover` to see the test coverage:

```
$ PERL5OPT=-MDevel::Cover prove -l t
---------------------------------- ------ ------ ------ ------ ------ ------
File                                stmt   bran   cond    sub   time  total
---------------------------------- ------ ------ ------ ------ ------ ------
lib/Logger.pm                     100.0  100.0  100.0  100.0    3.4  100.0
```

Now we have to implement the logger which logs into the file. In order to reduce the duplicated code, we can use template methods, but of course it always depends on the task, there are lots of ways to do that.

As a start we rename our logger to `LoggerStderr`. Then we create `LoggerFile`, which will be `LoggerStderr` copy for now, also we will copy the test. Our directory structure looks like this:

```
lib/
    LoggerFile.pm
    LoggerStderr.pm
t/
    logger_file.t
    logger_stderr.t
```

In `logger_file.t` we change the tests that check that the log message was written into stderr, so they check that the message was written into a file. Insted of `Capture::Tiny`, let's write our own function which will read from the file:

```
sub _slurp {
    my $file = shift;
    return do { local $/; open my $fh, '<', $file or die $!; <$fh> };
}
```

For testing that the file was written we're going to create temporary files with the help of `File::Temp` and our tests will look like this:

```
subtest 'prints to file' => sub {
    my $file = File::Temp->new;
    my $log = _build_logger(file => $file->filename);

    for my $level (qw/error warn debug/) {
        $log->$level('message');

        my $content = _slurp($file->filename);

        ok $content;
    }
};
```

As can be seen we pass the file name into constructor and our `_build_logger` now looks like this:

```
sub _build_logger {
    my $logger = LoggerFile->new(@_);
    $logger->set_level('debug');
    return $logger;
}
```

We run the test and validate that they fail. Now we implement the file writing.

```
sub _log {
    my $self = shift;
    my ($level, $message) = @_;

    return unless $LEVELS->{$level} <= $LEVELS->{$self->level};

    my $time = Time::Piece->new->strftime('%Y-%m-%d %T');

    open my $fh, '>>', $self->{file} or die $!;
    print $fh $time, " [$level] ", $message, "\n";
    close $fh;
}
```

Now the tests and implementation both have a lot of duplicated code. First, let's get rid of it in implementation, extracting the base class with a template method `_print`, which will be implemented in `LoggerFile` and `LoggerStderr`. During the refactoring we should run the tests as often as possible, to make sure that nothing has been broken.

The base class:

```
package LoggerBase;
use Moo;

use Carp qw(croak);
use List::Util qw(first);
use Time::Piece;

my $LEVELS = {
    error => 1,
    warn  => 2,
    debug => 3
};

sub set_level {
    my $self = shift;
    my ($new_level) = @_;

    croak('Unknown log level')
      unless first { $new_level eq $_ } keys %$LEVELS;

    $self->{level} = $new_level;
}

sub level {
    my $self = shift;

    return $self->{level} || 'error';
}

sub error { shift->_log('error', @_) }
sub warn  { shift->_log('warn',  @_) }
sub debug { shift->_log('debug', @_) }

sub _log {
    my $self = shift;
    my ($level, $message) = @_;

    return unless $LEVELS->{$level} <= $LEVELS->{$self->level};

    my $time = Time::Piece->new->strftime('%Y-%m-%d %T');

    my $text = join '', $time, " [$level] ", $message, "\n";

```

```
    $self->_print($text);
}

sub _print { ... }

1;
```

LoggerStderr:

```
package LoggerStderr;
use strict;
use warnings;

use base 'LoggerBase';

sub _print {
    my $self = shift;
    my ($message) = @_;

    print STDERR $message;
}

1;
```

LoggerFile:

```
package LoggerFile;
use Moo;
BEGIN { extends 'LoggerBase' }

sub BUILD {
    my $self = shift;
    my (%params) = @_;

    $self->{file} = $params{file};

    return $self;
}

sub _print {
    my $self = shift;
    my ($message) = @_;
```

```
17
18      open my $fh, '>>', $self->{file} or die $!;
19      print $fh $message;
20      close $fh;
21  }
22
23  1;
```

Since now we have created a new class, we have to test it also. How do we test base classes? A common way to do that is to create a fake test class that will inherit from the testable one. And then we remove from the test everything that is duplicated. Some of the tests will remain duplicated, but let's do it step by step.

Let's implement the base class test:

```
1   use Test2::V0
2
3   subtest 'creates correct object' => sub {
4       isa_ok(LoggerTest->new, 'LoggerTest');
5   };
6
7   subtest 'has default log level' => sub {
8       my $logger = LoggerTest->new;
9
10      is $logger->level, 'error';
11  };
12
13  subtest 'sets log level' => sub {
14      my $logger = LoggerTest->new;
15
16      $logger->set_level('debug');
17
18      is $logger->level, 'debug';
19  };
20
21  subtest 'not throws when known log level' => sub {
22      my $log = LoggerTest->new;
23
24      for my $level (qw/error warn debug/) {
25          ok lives { $log->set_level($level) };
26      }
27  };
28
```

```
29  subtest 'throws exception when invalid log level' => sub {
30      my $log = LoggerTest->new;
31
32      ok dies { $log->set_level('unknown') };
33  };
34
35  sub _build_logger {
36      my $logger = LoggerTest->new(@_);
37      $logger->set_level('debug');
38      return $logger;
39  }
40
41  done_testing;
42
43  package LoggerTest;
44  use base 'LoggerBase';
45
46  sub _print { }
```

Now we remove the base class tests from other files.

The files `logger_stderr.t` and `logger_file.t` have tests that are almost identical, but they are closely coupled with the implementation, for example in this test:

```
1   subtest 'not logs when level is lower' => sub {
2       my $log = _build_logger();
3
4       my $levels = {
5           error => [qw/warn debug/],
6           warn  => [qw/debug/],
7       };
8
9       for my $level (keys %$levels) {
10          $log->set_level($level);
11          for my $test_level (@{$levels->{$level}}) {
12              ok !capture_stderr {
13                  $log->$test_level('message');
14              }, "not log '$test_level' when '$level'";
15          }
16      }
17  };
```

actually here we have to check that `_print` method is not called, and not that there is a message in stderr. We can move this test into the base one, but before we have to change the base class a little:

```
package LoggerTest;
use Moo;
BEGIN { extends 'LoggerBase' }

sub BUILD {
    my $self = shift;
    my (%params) = @_;

    $self->{output} = $params{output};

    return $self;
}

sub _print {
    my $self = shift;

    push @{$self->{output}}, @_;
}
```

This way by passing `$output` we can later on in the test check that something was written there. Now the test that check the correctness of formatting can look like this:

```
subtest 'prints formatted line' => sub {
    my $output = [];
    my $log = _build_logger(output => $output);

    for my $level (qw/error warn debug/) {
        $log->$level('message');

        like $output->[-1], qr/\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d \[$level\] message/;
    }
};
```

and we can remove that from the child classes. The same thing we perform with the tests that do not check the writing implementation, but check the internal behaviour, implemented in the base class. Final tests look like this:

`logger_base.t`:

```
use Test2::V0

subtest 'creates correct object' => sub {
    isa_ok(LoggerTest->new, 'LoggerTest');
};

subtest 'has default log level' => sub {
    my $logger = LoggerTest->new;

    is $logger->level, 'error';
};

subtest 'sets log level' => sub {
    my $logger = LoggerTest->new;

    $logger->set_level('debug');

    is $logger->level, 'debug';
};

subtest 'not throws when known log level' => sub {
    my $log = LoggerTest->new;

    for my $level (qw/error warn debug/) {
        ok lives { $log->set_level($level) };
    }
};

subtest 'throws exception when invalid log level' => sub {
    my $log = LoggerTest->new;

    ok dies { $log->set_level('unknown') };
};

subtest 'prints formatted line' => sub {
    my $output = [];
    my $log = _build_logger(output => $output);

    for my $level (qw/error warn debug/) {
        $log->$level('message');

        like $output->[-1],
          qr/\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d \[$level\] message/;
```

```
    }
};

subtest 'logs when level is higher' => sub {
    my $output = [];
    my $log = _build_logger(output => $output);

    my $levels = {
        error => [qw/error/],
        warn  => [qw/error warn/],
        debug => [qw/error warn debug/],
    };

    for my $level (keys %$levels) {
        $log->set_level($level);
        for my $test_level (@{$levels->{$level}}) {
            $log->$test_level('message');

            ok $output->[-1];
        }
    }
};

subtest 'not logs when level is lower' => sub {
    my $output = [];
    my $log = _build_logger(output => $output);

    my $levels = {
        error => [qw/warn debug/],
        warn  => [qw/debug/],
    };

    for my $level (keys %$levels) {
        $log->set_level($level);
        for my $test_level (@{$levels->{$level}}) {
            $log->$test_level('message');

            ok !$output->[-1], "not log '$test_level' when '$level'";
        }
    }
};

sub _build_logger {
```

```
    my $logger = LoggerTest->new(@_);
    $logger->set_level('debug');
    return $logger;
}

done_testing;

package LoggerTest;
use Moo;
BEGIN { extends 'LoggerBase' }

sub BUILD {
    my $self = shift;
    my (%params) = @_;

    $self->{output} = $params{output};

    return $self;
}

sub _print {
    my $self = shift;

    push @{$self->{output}}, @_;
}
```

logger_stderr.t:

```
use Test2::V0
use Capture::Tiny qw(capture_stderr);
use LoggerStderr;

subtest 'creates correct object' => sub {
    isa_ok(LoggerStderr->new, 'LoggerStderr');
};

subtest 'prints to stderr' => sub {
    my $log = _build_logger();

    for my $level (qw/error warn debug/) {
        my $stderr = capture_stderr {
            $log->$level('message');
        };
```

```

        ok $stderr;
    }
};

subtest 'prints to stderr with \n' => sub {
    my $log = _build_logger();

    for my $level (qw/error warn debug/) {
        my $stderr = capture_stderr {
            $log->$level('message');
        };

        like $stderr, qr/\n$/;
    }
};

sub _build_logger {
    my $logger = LoggerStderr->new;
    $logger->set_level('debug');
    return $logger;
}

done_testing;
```

logger_file.t:

```
use Test2::V0
use File::Temp;
use LoggerFile;

subtest 'creates correct object' => sub {
    isa_ok(LoggerFile->new, 'LoggerFile');
};

subtest 'prints to file' => sub {
    my $file = File::Temp->new;
    my $log = _build_logger(file => $file->filename);

    for my $level (qw/error warn debug/) {
        $log->$level('message');

        my $content = _slurp($file);
```

```
17
18          ok $content;
19      }
20  };
21
22  subtest 'prints to stderr with \n' => sub {
23      my $file = File::Temp->new;
24      my $log = _build_logger(file => $file);
25
26      for my $level (qw/error warn debug/) {
27          $log->$level('message');
28
29          my $content = _slurp($file);
30
31          like $content, qr/\n$/;
32      }
33  };
34
35  sub _slurp {
36      my $file = shift;
37      my $content = do { local $/; open my $fh, '<', $file->filename or die $!; <$fh> \
38  };
39      return $content;
40  }
41
42  sub _build_logger {
43      my $logger = LoggerFile->new(@_);
44      $logger->set_level('debug');
45      return $logger;
46  }
47
48  done_testing;
```

As a final touch let's implement a factory that will create a needed logger. We have to test that the correct object is returned and that if we pass an unknown type we get an exception. The test can look like this:

```
use Test2::V0
use Logger;

subtest 'creates stderr logger' => sub {
    my $logger = Logger->build('stderr');

    isa_ok $logger, 'LoggerStderr';
};

subtest 'creates file logger' => sub {
    my $logger = Logger->build('file');

    isa_ok $logger, 'LoggerFile';
};

subtest 'throws when unknown logger' => sub {
    ok dies { Logger->build('unknown') };
};

done_testing;
```

And the factory itself:

```
package Logger;

use strict;
use warnings;

use Carp qw(croak);
use LoggerStderr;
use LoggerFile;

sub build {
    my $class = shift;
    my ($type, @args) = @_;

    if ($type eq 'stderr') {
        return LoggerStderr->new(@args);
    } elsif ($type eq 'file') {
        return LoggerFile->new(@args);
    }

    croak('Unknown type');
```

```
21 }
22
23 1;
```

We have implemented a simple logger using TDD methodology.

The Testing Infrastructure

This section explains the testing infrastructure of Perl we used in the quick start. We explain how to write basic tests and how to run them.

What are tests anyway?

Tests are part of a test program which itself is a normal Perl program. It can be run like a normal program. The real magic happens when you call it with `prove` as we will see in a few seconds.

A test compares a result computed by the code under test with an expected result. In Perl we have many modules with special comparisons, let's start with an example using the the most basic one: `Test::More`.

```
1  use strict;
2  use warnings;
3
4  use Test::More;
5
6  subtest 'basic comparison of constants' => sub {
7      my $expected = 'success';
8      my $got      = 'error';
9
10     is $got, $expected;
11 };
12
13 done_testing;
```

The test program uses the function `is` which is imported by the module `Test::More` to compare an expected result with a computed one. Traditionally in Perl those two values are put in two variables named `$expected` and `$got`. It's best practice to follow that tradition.

As you can see, both values differ, so our test should fail. We use the function `subtest` (which is also imported by `Test::More`) to give our test a descriptive name of what it does.

`done_testing` tells the testing infrastructure that we're done with our tests. We'll explain that in a few paragraphs.

Proving that we are wrong

So let's run the test program with `prove` and look at it's output.

```
$ prove code/testing-infrastructure/1-basic-comparison-of-constants.t
code/testing-infrastructure/1-basic-comparison-of-constants.t ..
    #   Failed test at code/testing-infrastructure/1-basic-comparison-of-constants.t\
 line 7.
    #          got: 'bug'
    #     expected: 'error'
    # Looks like you failed 1 test of 1.
code/testing-infrastructure/1-basic-comparison-of-constants.t .. 1/?
#   Failed test 'basic comparison of constants'
#   at code/testing-infrastructure/1-basic-comparison-of-constants.t line 8.
# Looks like you failed 1 test of 1.
code/testing-infrastructure/1-basic-comparison-of-constants.t .. Dubious, test retur\
ned 1 (wstat 256, 0x100)
Failed 1/1 subtests

Test Summary Report
-------------------
code/testing-infrastructure/1-basic-comparison-of-constants.t (Wstat: 256 Tests: 1 F\
ailed: 1)
  Failed test:  1
  Non-zero exit status: 1
Files=1, Tests=1,  0 wallclock secs ( 0.02 usr  0.00 sys +  0.05 cusr  0.00 csys =  \
0.07 CPU)
Result: FAIL
```

Whoa! That's a lot of output for such a simple test program. Let's look at it bit by bit.

The output can be divided in two sections: the first section contains the test output. That are the lines starting with the pound symbol. The second and shorter section is headed by it's name: Test Summary Report. We will discuss it when we have more tests, you can ignore it for now except the last line: FAIL. This meets our expectations.

The test output tells us quite clearly what's wrong: two values differ, and 1 of 1 tests fail. The output even tells us what line to look at in which file, which is quite handy if our code base gets bigger. And there's the red coloured eye catcher pointing us at the failed test program.

So nows let's correct our error to see what prove tells us when all tests pass.

```
use strict;
use warnings;

use Test::More;

subtest 'basic comparison of constants' => sub {
    my $expected = 'success';
    my $got      = 'success';

    is $got, $expected;
};

done_testing;
```

Again, run it with `prove`:

```
$ prove code/testing-infrastructure/2-basic-comparison-of-constants.t
code/testing-infrastructure/2-basic-comparison-of-constants.t .. ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.03 usr  0.00 sys +  0.05 cusr  0.00 csys =  \
0.08 CPU)
Result: PASS
```

That's much better now! And since there are no errors to correct we have much lesser output. And we have a nice green looking line telling us that all things went well.

This simple example is a display of the basic red-green-cycle: First we have a failing test indicated by a symbolic red light, and after we have corrected our code the green light is lit.

The "Test Anything" Protocol

The Perl folks invented a protocol for software testing along the way: The "Test Anything" Protocol, or TAP in short. Basically it's a way of collecting the information of the success or failure of tests. We can see it's full beauty when we run `prove` with the `-v` switch (for verbose output).

```
$ prove -v testing-infrastructure/2-basic-comparison-of-constants.t
testing-infrastructure/2-basic-comparison-of-constants.t ..
# Subtest: basic comparison of constants
    ok 1
    1..1
ok 1 - basic comparison of constants
1..1
ok
All tests successful.
Files=1, Tests=1,  1 wallclock secs ( 0.03 usr  0.01 sys +  0.08 cusr  0.02 csys =  \
0.14 CPU)
Result: PASS
```

The output starts with the name of the test program followed by two dots. Then we see the test name after a pound sign, and the tests of that subtest indented. The `1..1` and the following lines tell us that `prove` ran 1 of 1 tests and everything is "ok".

The line before the final line holds statistical information.

Usually you'll look at this verbose information only if you're curious about the strange output you're seeing in the console. It can be quite long, so normally you wouldn't call `prove` with the `-v` switch.

TAP was thought to become a standard and several implementation for different languages exist, but in practice it is rarely used outside of the Perl community. There is a dedicated website testanything.org[4] where you can find the specification, history and other useful information.

Testing a subroutine

Let's get away from this tiny example and work on a real function that reads the contents of a file. How hard can that be?

We'll let our development driven by tests, so let's write the test first. That way we design the API before we have written a single line of code. Our function reads a file, so we'll just name it that way: `read-file`. It gets the file to read as it's single argument called `$filename`. It's in a module called `FileRead`.

But what file should it read? It's surely not a good idea to hard-wire the name of a file that's in our home directory since this won't be there if some other user calls the test program. It's best practice to either generate the test data that's needed or to include it along with the tests. We'll choose the latter option and put the file containing some random text in `t/data/read-file-happy-path.txt`.

So here's the test:

[4] http://testanything.org/

```
use strict;
use warnings;

use Test::More;
use FileRead;

subtest 'reading an existing file (happy path)' => sub {
    my $filename = 't/data/read-file-happy-path.t';

    my $expected = 'random content';
    my $got      = read_file($filename);

    is $got, $expected;
};

done_testing;
```

Let's run it with `prove`:

```
$ prove 3-read-file-happy-path.t
3-read-file-happy-path.t .. Can't locate FileRead.pm in @INC [..] at 3-read-file-hap\
py-path.t line 3.
BEGIN failed--compilation aborted at 3-read-file-happy-path.t line 3.
3-read-file-happy-path.t .. Dubious, test returned 2 (wstat 512, 0x200)
No subtests run

Test Summary Report
-------------------
3-read-file-happy-path.t (Wstat: 512 Tests: 0 Failed: 0)
  Non-zero exit status: 2
  Parse errors: No plan found in TAP output
Files=1, Tests=0,  1 wallclock secs ( 0.04 usr  0.01 sys +  0.06 cusr  0.01 csys =  \
0.12 CPU)
Result: FAIL
```

It fails. Quite loudly. This is no surprise since we haven't written the code yet, so let's add the subroutine's definition in the module `ReadFile`:

```
1  package FileRead;
2  use strict;
3  use warnings;
4  use base 'Exporter';
5
6  our @EXPORT_OK = qw(read_file);
7
8  use Path::Tiny;
9
10 use v5.20;
11 use feature qw(signatures);
12 no warnings qw(experimental::signatures);
13
14 sub read_file ($filename) {
15     my $file = path($filename);
16     return $file->slurp;
17 }
18
19 1;
```

Run again with `prove`, this time using the switch `-l` which adds the `lib` directory to the module search path. This should work now:

```
$ prove -l testing-infrastructure/3-read-file-happy-path.t
testing-infrastructure/3-read-file-happy-path.t ..     # No tests run!
testing-infrastructure/3-read-file-happy-path.t .. 1/?
#   Failed test 'No tests run for subtest "reading an existing file (happy path)"'
#   at testing-infrastructure/3-read-file-happy-path.t line 12.
Error open (<) on 't/data/read-file-happy-path.txt': No such file or directory at [.\
..]/lib/FileRead.pm line 16.
# Tests were run but no plan was declared and done_testing() was not seen.
# Looks like your test exited with 255 just after 1.
testing-infrastructure/3-read-file-happy-path.t .. Dubious, test returned 255 (wstat\
 65280, 0xff00)
Failed 1/1 subtests

Test Summary Report
-------------------
testing-infrastructure/3-read-file-happy-path.t (Wstat: 65280 Tests: 1 Failed: 1)
  Failed test:  1
  Non-zero exit status: 255
  Parse errors: No plan found in TAP output
Files=1, Tests=1,  0 wallclock secs ( 0.03 usr  0.01 sys +  0.06 cusr  0.00 csys =  \
```

```
0.10 CPU)
Result: FAIL
```

Oh, we forgot the test file. Correct that by creating it and run again with `prove -l`:

```
$ prove -l t/testing-infrastructure/3-read-file-happy-path.t
t/testing-infrastructure/3-read-file-happy-path.t .. ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.02 usr  0.00 sys +  0.06 cusr  0.01 csys =  \
0.09 CPU)
Result: PASS
```

Now for some real world cases. The function is a tool that's used by others and may be misused. What if the argument is undefined? Or the file doesn't exist? Or can't be read by the user calling the function? Or something else goes wrong whilst reading the file? Or function should check for that.

Let's tickle that one by one. First, what if the subroutine is called with an undefined file name? We expect the function to return `undef` in that case, so we'll add a test that checks for that case and run it with `prove -l`:

```
use strict;
use warnings;

use Test::More;
use FileRead;

subtest 'reading an existing file (happy path)' => sub {
    my $filename = 't/data/read-file-happy-path.t';
    my $expected = 'random content';

    my $got = read_file($filename);

    is $got, $expected;
};

subtest 'trying to read a file with undefined name' => sub {
    my $filename = undef;
    my $expected = undef;

    my $got = read_file($filename);

    is $got, $expected;
};
```

```
24
25 done_testing;
```

Well, not quite. It dies and throws an expection. (We won't include the exact error message so you have the call the example by yourself.) We'll have to catch that one using `Try::Tiny` and return `undef` if it is catched:

```
1  package FileReadWithTry;
2  use strict;
3  use warnings;
4  use base 'Exporter';
5
6  our @EXPORT_OK = qw(read_file);
7
8  use Path::Tiny;
9  use Try::Tiny;
10
11 use v5.20;
12 use feature qw(signatures);
13 no warnings qw(experimental::signatures);
14
15 sub read_file ($filename) {
16     my $contents;
17     try {
18         my $file = path($filename);
19         $contents=$file->slurp;
20     };
21     return $contents;
22 }
23
24 1;
```

Run it with `prove -l`:

```
$ prove -l t/testing-infrastructure/4-read-file-undefined-filename.t
t/testing-infrastructure/4-read-file-undefined-filename.t .. ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs ( 0.02 usr  0.00 sys +  0.06 cusr  0.00 csys =  \
0.08 CPU)
Result: PASS
```

To reduce the verbosity of this text we'll assume that you know how to run a test program from this point on and won't include it anymore. Additionally, we hightlight only the interesting parts of `prove`'s output.

Arrange, Act and Assert

In this example we display the testing pattern called "AAA": arrange, act, assert. Again, this is a best practice pattern. First we arrange for the preconditions, than we act on the function to test, and then we assert that the expected result has occurred.

Arrange:

```
my $filename;
my $expected;
```

Act:

```
my $got = read_file($filename);
```

Assert:

```
is $got, $expected;
```

Now that we know the AAA concept, let's try to refactor our test program for the happy path and follow that principle. We hard-wired the file to read from and needed an extra file somewhere else. Our goal is to have no unneccessary hard-wired items in our code, and since test code is just normal code we apply our usual standards to test code.

So let's fix that: We don't expect the file to exist but we create it on the fly. We need the file just for the test, so it's a temporary file. Guess what, we have a module that does just that: `Path::Tiny` has a method for that.

The arrage phase of our happy path test creates the temporary file and writes contents to it, the act phase calls `read_file`, and the assert phase checks that we read the contents we've written to the file:

```
1  use strict;
2  use warnings;
3
4  use Test::More;
5  use File::Temp qw(tempfile);
6  use FileRead;
7
8  subtest 'reading an existing file (happy path)' => sub {
9      my ( $fh, $filename ) = tempfile( UNLINK => 1 );
10     my $content = 'random content';
11     print $fh $content;
```

```
12      close $fh;
13      my $expected = $content;
14
15      my $got = read_file($filename);
16
17      is $got, $expected;
18  };
19
20  done_testing;
```

Check that the test is still green by running `prove`.

Key Take Aways

- Tests are part of test programs that are called by the tool `prove`.
- The functions doing the actual testing produce output adhering to the "Test Anything" Protocol (TAP).
- If your test needs additional data, it's best practice to generate it in the test itself or bundle it with the test in a special directory under `t`.
- It's a good practice to write a test first, and the code fulfilling the test afterwards. This is the "red-green cycle".
- It's best practice to structure tests following the "Arrange, Act, Assert" pattern.
- It's good practice to treat your test code as normal code and apply your standard coding rules.