

Perl Testing

A Developer's Notebook™

Ian Langworth & chromatic

- Testing strategies
- Testing whole applications
- Testing Perl & non-Perl code
- Helpful testing libraries

O'REILLY®

Distributing Your Tests (and Code)

The goal of all testing is to improve the quality of code. Quality isn't just the absence of bugs and features behaving as intended. High-quality code and projects install well, behave well, have good and useful documentation, and demonstrate reliability and care outside of the code itself. If your users can run the tests too, that's a good sign.

It's not always easy to build quality into a system, but if you can test your project, you can improve its quality. Perl has several tools and techniques to distribute tests and test the non-code portions of your projects. The labs in this chapter demonstrate how to use them and what they can do for you.

Testing POD Files

The Plain Old Documentation format, or POD, is the standard for Perl documentation. Every Perl module distribution should contain some form of POD, whether in standalone *.pod* files or embedded in the modules and programs themselves.

As you edit documentation in a project, you run the risk of making errors. While typos and omissions can be annoying and distracting, formatting errors can render your documentation incorrectly or even make it unusable. Missing an `=cut` on inline POD may cause bizarre failures by turning working code into documentation. Fortunately, a test suite can check the syntax of all of the POD in your distribution.

How do I do that?

Consider a module distribution for a popular racing sport. The directory structure contains a *t/* directory for the tests and a *lib/* directory for the

modules and POD documents. To test all of the POD in a distribution, create an extra test file, *t/pod.t*, as follows:

```
use Test::More;

eval 'use Test::Pod 1.00';
plan( skip_all => 'Test::Pod 1.00 required for testing POD' ) if $@;

all_pod_files_ok();
```

Run the test file with *prove*:

```
$ prove -v t/pod.t
t/pod....1..3
ok 1 - lib/Sports/NASCAR/Car.pm
ok 2 - lib/Sports/NASCAR/Driver.pm
ok 3 - lib/Sports/NASCAR/Team.pm
ok
All tests successful.
Files=1, Tests=3, 0 wallclock secs ( 0.45 cusr + 0.03 csys = 0.48 CPU)
```

People who build modules likely need to run the tests. People who install prebuilt packages may not.

What just happened?

Because `Test::Pod` is a prerequisite only for testing, it's an optional prerequisite for the distribution. The second and third lines of *t/pod.t* check to see whether the user has `Test::Pod` installed. If not, the test file skips the POD-checking tests.

One of the test functions exported by `Test::Pod` is `all_pod_files_ok()`. If given no arguments, it finds all Perl-related files in a *blib/* or *lib/* directory within the current directory. It declares a plan, planning one test per file found. The previous example finds three files, all of which have valid POD.

If `Test::Pod` finds a file that doesn't contain any POD at all, the test for that file will be a success.

What about...

Q: *How can I test a specific list of files?*

A: Pass `all_pod_files_ok()` an array of filenames of all the files to check. For example, to test the three files that `Test::Pod` found previously, change *t/pod.t* to:

```
use Test::More;

eval 'use Test::Pod 1.00';
plan( skip_all => 'Test::Pod 1.00 required for testing POD' ) if $@;
```

```

all_pod_files_ok(
    'lib/Sports/NASCAR/Car.pm',
    'lib/Sports/NASCAR/Driver.pm',
    'lib/Sports/NASCAR/Team.pm'
);

```

Q: *Should I ship POD-checking tests with my distribution?*

A: There's no strong consensus in the Perl QA community one way or the other, except that it's valuable for developers to run these tests before releasing a new version of the project. If the POD won't change as part of the build process, asking users to run the tests may have little practical value besides demonstrating that you consider the validity of your documentation to be important.

For projects released to the CPAN, the CPAN Testing Service (<http://cpants.dev.zsi.at/>) currently considers the presence of POD-checking tests as a mark of "kwalitee" (see "Validating Kwalitee," later in this chapter). Not everyone agrees with this metric.

Testing Documentation Coverage

When defining an API, every function or method should have some documentation explaining its purpose. That's a good goal—one worth capturing in tests. Without requiring you to hardcode the name of every documented function, `Test::Pod::Coverage` can help you to ensure that all the subroutines you expect other people to use have proper POD documentation.

How do I do that?

Assume that you have a module distribution for a popular auto-racing sport. The distribution's base directory contains a `t/` directory with tests and a `lib/` directory with modules. Create a test file, `t/pod-coverage.t`, that contains the following:

```

use Test::More;

eval 'use Test::Pod::Coverage 1.04';
plan(
    skip_all => 'Test::Pod::Coverage 1.04 required for testing POD coverage'
) if $@;

all_pod_coverage_ok();

```

Run the test file with `prove` to see output similar to:

```

$ prove -v t/pod-coverage.t
t/pod-coverage....1..3
not ok 1 - Pod coverage on Sports::NASCAR::Car

```

Module::Starter creates a pod-coverage.t test file if you use it to create the framework for your distribution.

```

# Failed test (/usr/local/share/perl/5.8.4/Test/Pod/Coverage.pm
# at line 112)
# Coverage for Sports::NASCAR::Car is 75.0%, with 1 naked subroutine:
#   restrictor_plate
ok 2 - Pod coverage on Sports::NASCAR::Driver
ok 3 - Pod coverage on Sports::NASCAR::Team
# Looks like you failed 1 tests of 3.
dubious
    Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
    Failed 1/3 tests, 66.67% okay
Failed Test      Stat Wstat Total Fail  Failed  List of Failed
-----
t/pod-coverage.t  1   256     3     1   33.33%  1
Failed 1/1 test scripts, 0.00% okay. 1/3 subtests failed, 66.67% okay.

```

What just happened?

The test file starts as normal, setting up paths to load the modules to test. The second and third lines of *t/pod-coverage.t* check to see whether the `Test::Pod::Coverage` module is available. If it isn't, the tests cannot continue and the test exits.

`Test::Pod::Coverage` exports the `all_pod_coverage_ok()` function, which finds all available modules and tests their POD coverage. It looks for a *lib/* or *blib/* directory in the current directory and plans one test for each module that it finds.

Unfortunately, the output of the *prove* command reveals that there's some work to do: the module `Sports::NASCAR::Car` is missing some documentation for a subroutine called `restrictor_plate()`. Further investigation of *lib/Sports/NASCAR/Car.pm* reveals that documentation is lacking indeed:

```

=head2 make

Returns the make of this car, e.g., "Dodge".

=cut

sub make
{
    ...
}

sub restrictor_plate
{
    ...
}

```

In the previous listing, `make()` has documentation, but `restrictor_plate()` has none.

`Pod::Coverage` considers a subroutine to have documentation if there exists an `=head` or `=item` that describes it somewhere in the module. The `restrictor_plate()` subroutine clearly lacks either of these. Add the following to satisfy that heuristic:

```
=head2 make

Returns the make of this car, e.g., "Dodge".

=cut

sub make
{
    ...
}

=head2 restrictor_plate

Returns whether this car has a restrictor plate installed.

=cut

sub restrictor_plate
{
    ...
}
```

Run the test again to see it pass:

```
$ prove -v t/pod-coverage.t
t/pod-coverage....1..3
ok 1 - Pod coverage on Sports::NASCAR::Car
ok 2 - Pod coverage on Sports::NASCAR::Driver
ok 3 - Pod coverage on Sports::NASCAR::Team
ok
All tests successful.
Files=1, Tests=3, 1 wallclock secs ( 0.51 cusr + 0.03 csys = 0.54 CPU)
```

What about...

- Q:** *I have private functions that I don't want to document, but `Test::Pod::Coverage` complains that they don't have documentation. How can I fix that?*
- A:** See the `Test::Pod::Coverage` documentation for the `also_private` and `trustme` parameters. These come from `Pod::Coverage`, which also has good documentation well worth reading. By default, `Test::Pod::Coverage` makes some smart assumptions that functions beginning with underscores and functions with names in all caps are private.

Distribution Signatures

Cryptographically signing a distribution is more of an integrity check than a security measure. As the documentation for `Test::Signature` explains, by the time the `make test` portion of the installation checks the signature of a module, you've already executed a *Makefile.PL* or *Build.PL*, giving potentially malicious code the chance to run. Still, a signed distribution assures you that every file in the distribution is exactly what the author originally uploaded.

Signing a module distribution creates a file called *SIGNATURE* in the top-level directory that contains checksums for every file in the distribution. The author then signs the *SIGNATURE* file with a PGP or equivalent key. If you sign your distribution, you should include a signature validity check as part of the test suite.

How do I do that?

To sign a module, first install GnuPG and set up a private key that you'll use to do the signing with. For more information on how to use GnuPG, see the Documentation section on the GnuPG web site at <http://www.gnupg.org/>.

Next, install `Module::Signature`. `Module::Signature` provides the `cpansign` utility to create and verify *SIGNATURE* files. Describing module signatures, how to use `cpansign`, and considerations when bundling up modules is a bigger topic than this lab allows, so please see the `Module::Signature` documentation for information on how to sign your modules.

Once you've signed your distribution, you should see a *SIGNATURE* file in the distribution's directory containing something like:

```
This file contains message digests of all files listed in MANIFEST,
signed via the Module::Signature module, version 0.44.
```

```
...
```

```
-----BEGIN PGP SIGNED MESSAGE-----
```

```
Hash: SHA1
```

```
SHA1 e72320c0cd1a851238273f7d1jd7d46t395mrjbs Changes
```

```
SHA1 fm8b86bb3d93345751371f67chd01efe8tdua9f3 MANIFEST
```

```
SHA1 67i17fa0ff0ea897b0a2e43ddac01m6e5r8n132s META.yml
```

```
SHA1 cc010c8abd8a9941b1y0ad61fr808i7hfbcc32a1 Makefile.PL
```

```
SHA1 1fa0y76d5dac6c64d151b17f0td221sfmau2cci README
```

```
SHA1 fd94a423d3e42462fec2if7997a19y8b6abs3f7m lib/FAQ/Sciuridae.pm
```

```
SHA1 b7504edf3808b62742e3bm00dc464d3i9lf2b39m lib/FAQ/Sciuridae/Chipmunk.pm
```

```
SHA1 edde6f2c4608bfeee6acf9effff9644jbc815d6e lib/FAQ/Sciuridae/Marmot.pm
```

```
...
```

To verify the contents of *SIGNATURE* when the test suite is run, create a test file *00-signature.t*:

```
use Test::More;

eval 'use Test::Signature';

plan( skip_all => 'Test::Signature required for signature verification' )
    if $@;
plan( tests => 1 );
signature_ok();
```

Run the test file with *prove*:

```
$ prove -v t/00-signature.t
t/00-signature....1..1
ok 1 - Valid signature
ok
All tests successful.
Files=1, Tests=1, 1 wallclock secs ( 0.57 cusr + 0.05 csys = 0.62 CPU)
```

Because a broken signature is a showstopper when installing modules, it is common practice to prefix the file name with zeroes so that it runs early in the test suite.

What just happened?

Validating signatures is only a suggested step in installing modules, not a required one. Thus, *00-signature.t* checks to see whether the user has `Module::Signature` installed. It skips signature verification if not.

By default, `Test::Signature` exports a single function, `signature_ok()`, which reports a single test that indicates the validity of the *SIGNATURE* file.

To verify a *SIGNATURE* file, the test first checks the integrity of the PGP signature contained within. Next, it creates a list of checksums for the files listed in *MANIFEST*, comparing that list to the checksums supplied in *SIGNATURE*. If all of these steps succeed, the test produced by `signature_ok()` succeeds.

Internally, `Test::Signature`'s `signature_ok()` function and running `cpansign -v` use the same `verify()` function found in `Module::Signature`. If one of the steps to test the integrity of *SIGNATURE* fails, `signature_ok()` will produce the same or similar output to that of `cpansign -v`. For example, if one or more of the checksums is incorrect, the output will display a comparison of the list of checksums in the style of the *diff* utility.

Testing Entire Distributions

A proper Perl distribution contains a handful of files and lists any prerequisite modules that it needs to function properly. Each package should have a version number and have valid POD syntax. If you've signed your distribution cryptographically, the signature should validate. These are all important features, so why not test them?

The `Test::Distribution` module can do just that with one simple test script.

How do I do that?

Given a module distribution `Text::Hogwash`, create a test file `t/distribution.t` containing:

```
use Test::More;

eval 'require Test::Distribution';
plan( skip_all => 'Test::Distribution not installed' ) if $@;
Test::Distribution->import();
```

The `-l` option tells `prove` that modules for the distribution are in the `lib/` directory. Run `t/distribution.t` using `prove`:

```
$ prove -v -l t/distribution.t
t/distribution...1..14
ok 1 - Checking MANIFEST integrity
ok 2 - use Text::Hogwash::Tomfoolery;
ok 3 - use Text::Hogwash::Silliness;
ok 4 - Text::Hogwash::Tomfoolery defines a version
ok 5 - Text::Hogwash::Silliness defines a version
ok 6 - All non-core use()d modules listed in PREREQ_PM
ok 7 - POD test for lib/Text/Hogwash/Tomfoolery.pm
ok 8 - POD test for lib/Text/Hogwash/Silliness.pm
ok 9 - MANIFEST exists
ok 10 - README exists
ok 11 - Changes or ChangeLog exists
ok 12 - Build.PL or Makefile.PL exists
ok 13 - Pod Coverage ok
ok 14 - Pod Coverage ok
ok
All tests successful.
Files=1, Tests=14, 0 wallclock secs ( 0.19 cusr + 0.01 csys = 0.20 CPU)
```

What just happened?

`Test::Distribution` calculates how many tests it will run and declares the plan during its `import()` call. Some of these tests use modules

covered earlier, such as `Test::Pod` (“Testing POD Files”), `Test::Pod::Coverage` (“Testing Documentation Coverage”), and `Module::Signature` (“Distribution Signatures”). Others are simple checks that would be tedious to perform manually, such as ensuring that the *MANIFEST* and *README* files exist.

What about...

Q: *Is it possible to test a subset of distribution properties, such as the module prerequisites or package versions?*

A: The `Test::Distribution` documentation includes a list of the types of tests it performs, such as `prereq` and `versions`. Specify the types of tests you want to run by using `only` or `not` after the `import` statement:

```
Test::Distribution->import( only => [ qw( prereq versions ) ] );
```

The previous listing passes two additional arguments to `import()`: the string `only` and a reference to an array of the strings that represent the only types of tests that `Test::Distribution` should perform. When running the modified test file, the test output is much shorter because `Test::Distribution` runs only the named tests:

```
$ prove -v t/distribution.t
t/distribution....1.5
ok 1 - use Text::Hogwash::Tomfoolery;
ok 2 - use Text::Hogwash::Silliness;
ok 3 - Text::Hogwash::Tomfoolery defines a version
ok 4 - Text::Hogwash::Silliness defines a version
ok 5 - All non-core use()d modules listed in PREREQ_PM
ok
All tests successful.
Files=1, Tests=5, 1 wallclock secs ( 0.62 cusr + 0.04 csys = 0.66
CPU)
```

You can also use the `not` argument instead of `only` to prohibit `Test::Distribution` from running specified tests. It will run everything else.

Letting the User Decide

Installing a Perl module distribution is not always as simple as running the build file and testing and installing it. Some modules present the user with configuration options, such as whether to include extra features or to install related utilities. The example tests shown previously have simply skipped certain tests when prerequisite modules are not present. In

other cases, it is appropriate to ask the user to decide to run or to skip tests that require network connectivity or tests that may take an exorbitant amount of time to finish.

For example, consider the hypothetical module `MD5::Solve`, which reverses one-way MD5 checksums at the cost of an incredible amount of time, not to mention computing power and practicality. Performing this sort of task for even a small amount of data is costly, and the test suite for this module must take even more time to run. When installing the module, the user should have the option of skipping the expensive tests.

How do I do that?

`ExtUtils::MakeMaker` and `Module::Build` provide `prompt()` functions that prompt and receive input from the user who is installing the module. The functions take one or two arguments: a message to display to the user and a default value. These functions check the environment to make sure a human is indeed sitting at the terminal and, if so, display the message and wait for the user to enter a string. If there is no user present—in the case of an automated install, for example—they return the default value.

Using `ExtUtils::MakeMaker`, the build script for the module *Makefile.PL*, appears as follows:

```
use strict;
use warnings;
use ExtUtils::MakeMaker qw( WriteMakefile prompt );

my %config = (
    NAME          => 'MD5::Solve',
    AUTHOR        => 'Emily Anne Perlmonger <emmils@example.com>',
    VERSION_FROM  => 'lib/MD5/Solve.pm',
    ABSTRACT_FROM => 'lib/MD5/Solve.pm',
    PREREQ_PM     => { 'Test::More' => 0, },
    dist          => { COMPRESS => 'gzip -9f', SUFFIX => 'gz', },
    clean         => { FILES => 'MD5-Solve-*' },
);

my @patterns = qw( t/*.t );

print "=> Running the extended test suite may take weeks or years! <=\\n";
my $answer = prompt( 'Do you want to run the extended test suite?', 'no' );

if ( $answer =~ m/^y/i )
{
    print "I'm going to run the extended tests.\\n";
    push @patterns, 't/long/*.t';
}
```

```

else
{
    print "Skipping extended tests.\n";
}

$config{test} = { TESTS => join ' ', map { glob } @patterns };

WriteMakefile(%config);

```

Running the build script generates a *Makefile* and displays the following output, prompting the user to make a decision:

```

$ perl Makefile.PL
==> Running the extended test suite may take weeks or years! <==
Do you want to run the extended test suite? [no] no
Skipping extended tests.
Checking if your kit is complete...
Looks good
Writing Makefile for MD5::Solve

```

What just happened?

Many *Makefile.PL* files consist of a single `WriteMakefile()` statement. The previous *Makefile.PL* has an additional bit of logic to determine which sets of test scripts to run. The test files in *t/* always run, but those in *t/long/* run only if the user consents.

This file stores all of the options that a typical *Makefile.PL* provides to `WriteMakefile()` in a hash instead. By default, the program expands the pattern `t/*.t` into filenames that use `glob` by using the techniques described in “Bundling Tests with Modules,” later in this chapter. The program then adds these filenames to `%config`.

Before modifying `%config`, however, the file uses the `prompt()` function to ask the user to decide whether to run the lengthy tests. If the user’s answer begins with the letter *y*, the code adds the `glob` string `t/long/*.t` to the list of patterns of test files to run as part of the test suite during `make test`:

```

$ make test
cp lib/MD5/Solve.pm blib/lib/MD5/Solve.pm
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command:MM" "-e"
"test_harness(0, 'blib/lib', 'blib/arch')" t/00.load.t t/pod-coverage.t
t/pod.t t/long/alphanumeric.t t/long/digits.t t/long/long-string.t
t/long/longer-string.t t/long/punctuation.t t/long/random.t
t/long/short.t t/long/simple.t
t/00.load.....ok
t/long/alphanumeric....ok
t/long/digits.....ok
t/long/long-string.....ok
t/long/longer-string....ok

```

```
t/long/punctuation.....ok
t/long/random.....ok
...
```

However, if `ExtUtils::MakeMaker` decides not to ask for user input or the user hits the Enter key to accept the default value, the return value of `prompt()` will be `no`. In the previous example, the user entered `no` explicitly, so the tests in `t/long/` will not run:

```
$ make test
cp lib/MD5/Solve.pm blib/lib/MD5/Solve.pm
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e"
  "test_harness(0, 'blib/lib', 'blib/arch')" t/00.load.t t/pod-coverage.t
  t/pod.t
t/00.load.....ok
t/pod-coverage....ok
t/pod.....ok
All tests successful.
Files=3, Tests=3, 1 wallclock secs ( 1.09 cusr + 0.09 csys = 1.18 CPU)
```

Letting the User Decide (Continued)

`Module::Build` provides a `prompt()` method that takes the same arguments as the `prompt()` function exported by `ExtUtils::MakeMaker`. However, this `prompt()` is a method, so either call it on the `Module::Build` class or a `Module::Build` or subclass object.

`Module::Build` also provides a `y_n()` method that returns either `true` or `false`, to simplify asking boolean questions. The `y_n()` method takes the same arguments as `prompt()`, except that the default answer, if supplied, must be either `y` or `n`.

How do I do that?

The *Build.PL* file for the `MD5::Solve` module is:

```
use strict;
use warnings;
use Module::Build;

print "=> Running the extended test suite may take weeks or years! <=&\n";
my $answer = Module::Build->y_n(
    'Do you want to run the extended test suite?', 'n'
);

my $patterns = 't/*.t';

if ($answer)
{
```

```

    print "I'm going to run the extended tests.\n";
    $patterns .= ' t/long/*.t';
}
else
{
    print "Skipping extended tests.\n";
}

my $builder = Module::Build->new(
    module_name    => 'MD5::Solve',
    license        => 'perl',
    dist_author    => 'Emily Anne Perlmonger <emmils@example.com>',
    dist_version_from => 'lib/MD5/Solve.pm',
    build_requires => { 'Test::More' => 0, },
    add_to_cleanup  => ['MD5-Solve-*'],
    test_files     => $patterns,
);

$builder->create_build_script();

```

Module::Build automatically expands the pattern(s) of files given to test_files.

Run *Build.PL* to see:

```

$ perl Build.PL
==> Running the extended test suite may take weeks or years! <==
Do you want to run the extended test suite? [n] n
Skipping extended tests.
Checking whether your kit is complete...
Looks good
Creating new 'Build' script for 'MD5-Solve' version '0.01'

```

What just happened?

Similar to the `Makefile.PL` example earlier, the build script prompts the user whether to run the extended tests. If the user responds positively, `$answer` will be true, and the code will append `t/long/*.t` to the list of patterns of files to run in the test suite. Otherwise, only test files matching `t/*.t` will run during `make test`.

Bundling Tests with Modules

When releasing modules, you should always include a test suite so that the people installing your code can have confidence that it works on their systems. Tools such as the CPAN shell will refuse to install a distribution if any of its tests fail, unless the user forces a manual installation. If you upload the module to the CPAN, a group of dedicated individuals will report the results of running your test suite on myriad platforms. The CPAN Testers site at <http://testers.cpan.org/> reports their results.

This lab explains how to set up a basic distribution, including the directory structure and minimal test suite.

How do I do that?

Module distributions are archives that, when extracted, produce a standard directory tree. Every distribution should contain at least a *lib/* directory for the reusable module files, a *Build.PL* or *Makefile.PL* to aid in testing and installing the code, and a *t/* directory that contains the tests for the module and any additional data needed for testing.

If you haven't already created a directory structure for the distribution, the simplest way to start is by using the `module-starter` command from the `Module::Starter` distribution. `module-starter` creates the directories you need and even includes sample tests for your module.

Go ahead and install `Module::Starter`. Once installed, you should also have the `module-starter` program in your path. Create a fictitious distribution for calculating taxes that includes two modules, `Taxes::Autocomplete` and `Taxes::Loophole`:

Perl's documentation suggests using h2xs to create new modules. Module::Starter is just a modern alternative.

```
$ module-starter --mb --distro=Taxes \  
  --module=Taxes::Autocomplete,Taxes::Loophole \  
  --author='John Q. Taxpayer' \  
  --email='john@bigpockets.com' --verbose  
Created Taxes  
Created Taxes/lib/Taxes  
Created Taxes/lib/Taxes/Autocomplete.pm  
Created Taxes/lib/Taxes/Loophole.pm  
Created Taxes/t  
Created Taxes/t/pod-coverage.t  
Created Taxes/t/pod.t  
Created Taxes/t/00-load.t  
Created Taxes/Build.PL  
Created Taxes/MANIFEST  
Created starter directories and files
```

`module-starter` creates a complete distribution in the directory `Taxes/`. Further inspection of the `Taxes/t/` directory reveals three test files:

```
$ ls -1 Taxes/t/  
00-load.t  
pod-coverage.t  
pod.t
```

Any test files you add to `Taxes/t/` will run during the testing part of the module installation.

What just happened?

The `module-starter` command creates a skeleton directory structure for new modules. This structure includes the three test files in the previous output. These files perform basic tests to make sure your module

maintains a certain level of quality (or “kwalitee”—see “Validating Kwalitee,” later in this chapter).

t/pod-coverage.t and *t/pod.t* test POD documentation validity and coverage, respectively. *t/00-load.t* contains the “basic usage” test, which may be the most common type of test within different Perl module distributions. This test simply checks whether the modules in the distribution load properly. Note that *module-starter* has lovingly filled in all of the module names for you:

```
use Test::More tests => 2;

BEGIN
{
    use_ok( 'Taxes::Autocomplete' );
    use_ok( 'Taxes::Loophole' );
}

diag( "Testing Taxes::Autocomplete $Taxes::Autocomplete::VERSION,
      Perl 5.008004, /usr/bin/perl" );
```

You might see the same sort of tests in a test file with a different name, such as *OO_basic.t* or just *load.t*, or it may be one of several tests in another file.

What about?

Q: *I have 8,000 test files in my t/ directory! Can I use subdirectories to organize them better?*

A: Sure thing. If you use `Module::Build`, specify a `test_files` key whose value is a space-delimited string containing just the patterns of test files. `Module::Build` automatically expands the patterns.

```
use Module::Build;

my $build = Module::Build->new(
    ...
    test_files => 't/*.t t/**/*.t',
    ...
);

$builder->create_build_script();
```

Alternatively, set the `recursive_test_files` flag to use every *.t* file found within the *t/* directory and all of its subdirectories:

```
use Module::Build;

my $build = Module::Build->new(
    ...
    recursive_test_files => 1,
```

```

    ...
);
$builder->create_build_script();

```

If you use `ExtUtils::MakeMaker` and *Makefile.PL* instead, do the equivalent by providing a test key to the hash given to `WriteMakefile()`:

```

use ExtUtils::MakeMaker;

WriteMakeFile(
    ...
    test => { TESTS => join ' ', map { glob } qw( t/*.t t/**/*.t ) },
    ...
);

```

The value of the test hash pair must be a hash reference with the key `TESTS`. The value is a space-delimited string of all test files to run. In the previous example, `join` and `glob` create such a string based on the two patterns `t/*.t` and `t/**/*.t`. This is necessary because `WriteMakeFile()` will not automatically expand the patterns when used with ActiveState Perl on Windows.

Collecting Test Results

Distributing your tests with your code is a good diagnostic practice that can help you to ensure that your code works when your users try to run it. At least it's good for diagnostics when you can convince your users to send you the appropriate test output. Rather than walk them through the steps of running the tests, redirecting their output to files, and sending you the results, consider automating the process of gathering failed test output and useful information.

As usual, the CPAN has the solution in the form of `Module::Build::TestReporter`.

How do I do that?

Consider a Chef module that can slice, dice, fricassee, and boil ingredients. Create a new directory for it, with *lib/* and *t/* subdirectories. Save the following code as *lib/Chef.pm*:

```

package Chef;

use base 'Exporter';

use strict;
use warnings;

```

```

our $VERSION = '1.0';
our @EXPORT = qw( slice dice fricassee );

sub slice
{
    my $ingredient = shift;
    print "Slicing $ingredient...\n";
}

sub dice
{
    my $ingredient = shift;
    print "Dicing $ingredient...\n";
}

sub fricassee
{
    my $ingredient = shift;
    print "Fricasseeing $ingredient...\n";
}

sub boil
{
    my $ingredient = shift;
    print "Boiling $ingredient...\n";
}

1;

```

*Yes, the missing
export of `boil()` is
intentional.*

Save a basic, “does it compile?” test file as *t/use.t*:

```

#!/perl

use strict;
use warnings;

use Test::More tests => 1;

my $module = 'Chef';
use_ok( $module ) or exit;

```

Save the following test of its exports as *t/chef_exports.t*:

```

#!/perl

use strict;
use warnings;

use Test::More tests => 5;

my $module = 'Chef';
use_ok( $module ) or exit;

for my $export (qw( slice dice fricassee boil ))
{
    can_ok( __PACKAGE__, $export );
}

```

Finally, save the following build file as *Build.PL*:

```
use Module::Build::TestReporter;

my $build = Module::Build::TestReporter->new(
    module_name    => 'Chef',
    license        => 'perl',
    report_file    => 'chef_failures.txt',
    report_address => 'chef-failures@example.com',
    dist_version_from => 'lib/Chef.pm',
);

$build->create_build_script();
```

Now build the module as normal and run the tests:

```
$ perl Build.PL
Creating new 'Build' script for 'Chef' version '1.0'
$ perl Build
lib/Chef.pm -> blib/lib/Chef.pm
$ perl Build test
t/use.t...ok
Tests failed!
Please e-mail 'chef_failures.txt' to chef-failures@example.com.
```

What just happened?

Hang on, that's a lot different from normal. What's *chef_failures.txt*? Open it with a text editor; it contains output from the failed tests as well as information about the currently running Perl binary:

```
Test failures in 't/chef.t' (1/5):
5: - main->can('boil')
    Failed test (t/chef.t at line 13)
    main->can('boil') failed

Summary of my perl5 (revision 5 version 8 subversion 6) configuration:
<...>
```

Module::Build::TestReporter diverts the output of the test run and reports any failures to the file specified in *Build.PL*'s *report_file* parameter. It also prints a message about the failures and gives the address to which to send the results.

What happens if the tests all succeed? Open *lib/Chef.pm* and change the export line:

```
@EXPORT = qw( slice dice fricassee boil );
```

Then run the tests again:

```
$ perl Build test
All tests passed.
```

You're happy, the users are happy, and there's nothing left to do.

This lowers the barrier for users to report test failures. You don't have to walk them through running the tests in verbose mode, trying to capture the output. All they have to do is to email you the report file.

You can't guarantee that they will contact you, but you can make it easier.

What about...

Q: *What if I already have a `Module::Build` subclass?*

A: Make your subclass inherit from `Module::Build::TestReporter` instead. See the module's documentation for other ideas, too!

Q: *Can I have `Module::Build::TestReporter` email me directly? How about if it posted the results to a web page? That would make it even easier to retrieve failure reports from users.*

A: It would, but can you guarantee that everyone running your tests has a working Internet connection or an SMTP server configured for Perl to use? If so, feel free to subclass `Module::Build::TestReporter` to report directly to you.

Q: *My output looks different. Why?*

A: This lab covered an early version of the module. It may change its messages slightly. The basic functions will remain the same, though. As with all of the other testing modules, see the documentation for current information.

Validating Kwalitee

After all of the work coming up with the idea for your code, writing your code, and testing your code (or writing the tests and then writing the code), you may be ready to share your masterpiece with the world. You may feel understandably nervous; even though you know you have good tests, many other things could go wrong—things you won't recognize until they do go wrong.

Fortunately, the Perl QA group has put together loose guidelines of code kwalitee based on hard-won experience about what makes installing and using software easy and what makes it difficult. The CPAN Testing Service, or CPANTS, currently defines code kwalitee in 17 ways; see <http://cpants.dev.zsi.at/kwalitee.html> for more information.

Rather than walking through all 17 indicators by hand, why not automate the task?

Kwalitee isn't quite the same as quality, but it's pretty close and it's much easier to test.

How do I do that?

Download and install `Test::Kwalitee`. Then add the following code to your `t/` directory as `kwalitee.t`:

```
#!/perl

eval { require Test::Kwalitee };
exit if $@;
Test::Kwalitee->import();
```

Then run the code with *perl*:

```
$ perl t/kwalitee.t
1..8
ok 1 - checking permissions
ok 2 - looking for symlinks
ok 3 - needs a Build.PL or Makefile.PL
ok 4 - needs a MANIFEST
ok 5 - needs a META.yml
ok 6 - needs a README
ok 7 - POD should have no errors
ok 8 - code should declare all non-core prereqs
```

What just happened?

The test file is very simple. `Test::Kwalitee` does all of its work behind the scenes. The `eval` and `exit` lines exist to prevent the tests from attempting to run and failing for users who do not have the module installed.

`Test::Kwalitee` judges the kwalitee of a distribution on eight metrics:

- Are the permissions of the files sane? Read-only files cause some installers difficulty.
- Are there any symbolic links in the distribution? They do not work on all filesystems.
- Is there a file to run to configure, build, and test the distribution?
- Is there a *MANIFEST* file listing all of the distribution files?
- Is there a *META.yml* file containing the distribution's metadata?
- Is there a *README* file?
- Are there any errors in the included POD?
- Does the distribution use any modules it has not declared as prerequisites?

If all of those tests pass, the module has decent kwalitee. `Kwalitee` doesn't guarantee that your code works well, or even at all, but it is a sign that you've bundled it properly.

See the documentation for changes to these metrics.

What about...

Q: *Should I distribute this test with my other tests?*

A: Opinions vary. It's a useful test to run right before you release a new version of your distribution just to make sure that you haven't forgotten anything, but unless you're generating files that might change the code being tested on different platforms, this test won't reveal anything interesting when your users run it.

If you don't want to distribute the test and if you use `Module::Build` or `ExtUtils::MakeMaker` to bundle your distribution, add this test to your normal `t/` directory, but do not add it to your `MANIFEST` file. You can still run the test with `make test`, `perl Build test`, or `prove`, but `make tardist`, `make dist`, and `perl Build dist` will exclude it from the distribution file.

Q: *What if I disagree with a Kwalitee measurement and want to skip the test?*

A: See the documentation of `Test::Kwalitee` to learn how to disable certain tests.