Test Driven Development With Perl

A Stonehenge Consulting Course

Joshua McAdams
joshua.mcadams@gmail.com

Image from SideLong on flickr (http://flickr.com/photos/sidelong/246816211/)

About This Class

- Concepts of testing
  - What is testing?
  - Types of tests
  - Making good tests
- Basic Testing In Perl
  - Test::More
  - Prove
  - TAP
  - Test::Simple
- Testing OO-Style
  - Test::Class
- Testing Utilities
  - Test::Differences
  - Test::Deep
  - Test::MockObject
  - Test::MockObject::Extends
  - Devel::Cover
- Case Study: Testing Code That Accesses Databases

Concepts of Testing

- What does it mean to test your software?
- What types of testing exist?
- What can and can’t tests do?
- Why should you even bother with testing?
- What makes a good test?
- What does it mean to “test first”?
- What are automated unit tests?
- Concepts of Test Driven Development

What does it mean to test your software?

Concepts of Testing

- Testing seems like a very simple concept.

- Testing is the process of doing something, collecting the results of what you just did, and then verifying that the results are what you expected.

- Even simpler, software testing can be reduced to saying that given a known input, an expected output is produced.

- Even with a simple definition, testing is a very complex subject that is continually argued about.

What types of testing exist?

Concepts of Testing

- Before you start testing, you need to answer a few questions like what exactly should you test, who should do the testing, and what should you test for?

- Another question is what class of tests you are performing? Unit tests? Integration tests? Functional tests? System tests? System integration tests? Acceptance tests? Black box tests? Grey box tests? White box tests?

- Does it really matter?

What can and can’t tests do?

Concepts of Testing

- What can’t they do?
  - Tests can not prove with full confidence that a program is correct.
  - Tests can not prove the non-existence of bugs.
  - A program can never be completely tested.

- What can they do?
  - Tests can route out many bugs in your program.
  - Tests can improve the quality of your software.
  - Tests can allow you to confidently change your code.
  - Tests can serve as documentation for your program.
  - Tests can help you code faster.

Why should you even both with testing?

- Software is no longer just the job of the “QA Person”.
- After you master the art of testing, you will code faster and with more confidence.
- Employers are looking for the ability to test software when they look at your resume.

What makes a good test?

Concepts of Testing

- Should you test every single aspect of your system?
- Should you test only the edge cases?
- How much setup is acceptable for a test?
- Can tests have side effects that impact the system they are running on?

What does it mean to “test first”?

Concepts of Testing

- When you test first, you are typically creating unit tests, though some integration testing and even system integration testing is occasionally performed.

- Often, these unit tests are tied into an automated system so that they can be easily executed.

- The tests can even be put into a continual integration system so that they can be repeated executed.

- Literal “test first” development demands that you write a failing test and then write the minimal amount of code necessary to fix it.

What are automated unit tests?

Concepts of Testing

- Unit tests are tests, so they take the result of some action and validate that result against what was expected.
- Unit tests check small components of your system to make sure that they are functioning correctly.
- An automated test is a test that doesn't necessarily require any human intervention in order to run.
- In Perl, you get a nice automated unit testing system for free with prove, Test::Harness, Test::More, ExtUtils::MakeMaker, Module::Build, and friends.

Concepts of Test Driven Development

Concepts of Testing

- Think along the lines of “Lather, Rinse, Repeat” only it’s “Fail, Fix, Pass”.
- You write a test before you write any ‘real’ code and then don’t actually write code until the test fails.
- After the test fails, you minimally modify your code in order to make the test pass.
- Once you think that you’ve got your code ready to make the test pass, you verify that the test does indeed pass and then repeat the process.
- The entire time you are coding, you are looking for places where you can both refactor your code and refactor your tests.
- Failing tests before code!

Setting up our example

- Throughout the class, we'll be creating a Perl module that will calculate percentage of ownership for an owner in a condo.

- The percentages are used for important things like association elections and assessments, so they have to be correct or lawyers will get involved.

Details of the calculation

Setting up our example

- The formula works by determining the percentage of the building that an individual owns in proportion to the total livable space. Common areas are not included.

- The trick is that the actual square footage of any unit is weighted using multiplier. Each floor up adds an extra hundredth to the weight. For instance, a 600 square foot unit on the 11th floor would be weighted at 600 * 1.11 = 666.

Details of the calculation

Setting up our example

- Don’t worry about units that span multiple floors.
- Ninety-nine floors are the most that any building can have.
- Don’t worry about matching up owners who have multiple units.
- Underground floors all get a weight of zero, no matter how many levels down they actually are.

Basic Testing in Perl

- Creating tests with Test::More
- Running tests with prove
- A deeper look at Test::More
- Taking a look at the TAP
- Taking a look back at Test::Simple

Creating tests with Test::More

- We have a rough set of requirements.
- What do we do first?
    - Create a technical design?
    - Write out an object interface specification?
    - Write some code?

Creating tests with Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
1 use warnings;
2 use strict;
3 use Test::More tests => 1;
4
5 BEGIN { use_ok('PercentOwnership'); }
```

Running tests with prove

- So we have a nice test written that tries to load our yet-to-be-created module, PercentOwnership. Now what do we do?
- Since the test is just a Perl script, we could run it using the ‘perl’ executable on our system.
- ‘perl’ alone isn’t the best candidate for the job though.
- When you installed Perl, it not only installed ‘perl’ and ‘perldoc’, but it also installed a handy program for running your tests, ‘prove’.

Running tests with prove

Basic Testing in Perl

```
--(0)> prove t/percent_ownership.t
t/percent_ownership....
#   Failed test 'use PercentOwnership;'
#   at t/percent_ownership.t line 5.
#     Tried to use 'PercentOwnership'.
#     Error:  Can't locate PercentOwnership.pm in @INC (@INC contains: ...) at
(eval 3) line 2.
# BEGIN failed--compilation aborted at t/percent_ownership.t line 5.
# Looks like you failed 1 test of 1.
t/percent_ownership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test           Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/percent_ownership.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
```

Running tests with prove

- Now that we've created a test and the test fails, it's time to write some 'real' code so that our test can pass.

- It is a common test driven development practice to write a test, watch it fail, and then fix the code that you are testing by doing only the **minimal** amount of work necessary to make the test work.

- The word minimal is stressed because that is one of the most difficult aspects of test driven development.

- It is very tempting to add a feature that you 'know' will be needed while you are in the code working to make a test pass. Don't!

- No code should be written until a test is written to verify that code.

Running tests with prove

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
1 package PercentOwnership;
2
3 use warnings;
4 use strict;
5
6 1;
```

Running tests with prove

Basic Testing in Perl

```
--(0)> prove t/percent_ownership.t
t/percent_ownership....
#   Failed test 'use PercentOwnership;'
#   at t/percent_ownership.t line 5.
#     Tried to use 'PercentOwnership'.
#     Error:  Can't locate PercentOwnership.pm in @INC (@INC contains: ...) at
(eval 3) line 2.
# BEGIN failed--compilation aborted at t/percent_ownership.t line 5.
# Looks like you failed 1 test of 1.
t/percent_ownership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test           Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/percent_ownership.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
```

Running tests with prove


```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.04 cusr +  0.02 csys =  0.06 CPU)
```

A deeper look at Test::More

Basic Testing in Perl

- Test::More is probably the most popular Perl testing module that you'll see in use today.
- It provides many utility subroutines beyond the 'use_ok' that we've already seen.
- It uses the Test::Builder framework underneath, so it can work in conjunction with any other testing utility module that uses Test::Builder.
- Test::More outputs data in the Test Anything Protocol (TAP), so it fits in well with TAP-friendly systems like human beings, Test::Harness, and prove.
- All of the test methods that we'll look at return a true or false value depending on wether or not the given test passed or failed. This can be helpful in adding some conditional logic to your test scripts.
- Most of the test methods also take a test name as their final argument. This test name is reported when the test is executed and also when the test fails.

A deeper look at Test::More

Basic Testing in Perl

```
BEGIN { use_ok( $module ) }
BEGIN { use_ok( $module, @imports ) }
```

- Checks to make sure that the given module was loaded.
- If the module can/should accept arguments to an import method, those can be passed into use_ok.
- Wrapping the method in a BEGIN block allows for some compile time function exports, as well as, proper prototype handling.
- If code in the BEGIN block depends on the module being loaded, you'll probably want to put the dependent code in a separate BEGIN block.
- There is also a 'require_ok' if you need to test the inclusion of a file or module at runtime.

A deeper look at Test::More

Basic Testing in Perl

```
ok( $test_expression, $test_name )
```

- ‘ok’ is the most basic, but also the most general purpose of the Test::More utilities.

- The first argument can be any expression that evaluates to true or false.

- If the expression is true, the test passes. If the expression is false, the test fails.

- Failures report only the test name, not the value of the expression.

A deeper look at Test::More

Basic Testing in Perl

```
is( $got, $expected, $test_name )
isnt( $got, $expected, $test_name )
```

- ‘is’ and ‘isnt’ perform a string equal ‘eq’ and not equal ‘ne’ to the got and expected arguments, respectively.

- Using these methods gives you more detailed diagnostics on test failure than you would get from ‘ok’. The “got” and “expected” strings are displayed.

- Since numbers in Perl can be compared as strings in most cases, you can typically test two numbers using ‘is’ and ‘isnt’.

A deeper look at Test::More

Basic Testing in Perl

```
like( $got, qr/$expected/, $test_name )
unlike( $got, qr/$expected/, $test_name )
```

- ‘like’ and ‘unlike’ check to see if the “got” data matches and doesn’t match the given regular expressions, respectively.

- Using these methods gives you more detailed diagnostics on test failure than you would get from ‘ok’. The “got” scalar and “expected” regular expression are displayed.

A deeper look at Test::More

Basic Testing in Perl

```
cmp_ok( $got, ‘eq’, $expected, $test_name )
```

- ‘cmp_ok’ allows for any Perl binary operator to be used to compare the “got” and “expected” data.

- Using this methods gives you more detailed diagnostics on test failure than you would get from ‘ok’. The “got” scalar and “expected” scalar are displayed.

- Though the diagnostics would be nice in many cases, you don’t see this function used too often in practice.

A deeper look at Test::More


```
can_ok( $module, @methods )
can_ok( $object, @methods )
```

- ‘can_ok’ checks to see if a given module name or object has implemented all of the methods specified.

- The test fails if any method is not found.

A deeper look at Test::More

Basic Testing in Perl

```
isa_ok( $object, $class, $object_name )
isa_ok( $reference, $type, $reference_name )
```

- When passed an object ‘isa_ok’ checks to see if the object is of the type of the provided class.

- When passed a regular unblessed reference ‘isa_ok’ checks to see if the reference is of the type (‘ARRAY’, ‘HASH’, etc.) specified.

- The final object name and reference name arguments are similar to the test name found in other Test::More functions. They are used to make the output for test diagnostics more meaningful.

A deeper look at Test::More


```
is_deeply( $got, $expected, $test_name )
```

- ‘is_deeply’ expects to receive two references and will compare the data structures and fail if there are any differences to the makeup of structures.

- If the references are blessed as different objects, but are otherwise the same, ‘is_deeply’ will not think that the structures are different.

- ‘is_deeply’ is very useful; however, it does have its limitations and pain-points. Later in the class we’ll take a look at some modules that take the idea behind ‘is_deeply’ and expand on it.

A deeper look at Test::More


```
pass( $test_name )
fail( $test_name )
```

- ‘pass’ and ‘fail’ can be used when you just can wedge your tests into any of the methods provided by Test::More or any of the many other testing utility modules found on CPAN.

- These functions simply report a pass or failure for a test without actually doing any validation.

A deeper look at Test::More


```
diag( $message )
```

- Occasionally you'll need to report some diagnostic information while your tests are running. This could be general information or even additional debugging details that are printed when a test fails.
    - ok(1 == 2, 'numeric equivalents') or diag('numeric theory forbids 1 == 2')

A deeper look at Test::More

Basic Testing in Perl

```
use Test::More tests => 2
use Test::More qw(no_plan)
```

- Typically you'll tell Test::More how many tests you intend to run as soon as you load the module.

- If you really have no clue how many tests will be executed or possibly you're just in the development stage and don't want to keep updating your test count, then you can tell Test::More that you have no idea how many tests are going to run by using 'no_plan'.

- If you don't give Test::More a plan, it will create one for you after all of your tests are complete.

A deeper look at Test::More

Basic Testing in Perl

```
use Test::More;
plan tests => $test_count;
```

- If you will know how many tests you'll be executing a run time, but not at compile time, you can use a delayed plan.

- To delay your plan, just load Test::More with no import arguments and then call the 'plan' method later in your testing script.

- Pass 'plan' the same arguments that you'd have passed Test::More's import.

- Be sure to specify your plan before you run any tests.

A deeper look at Test::More

Basic Testing in Perl

```
SKIP: {
  skip $reason, $skip_count unless $conditions_are_right;
  ok( 1 == 1, ‘one is one’ );
}
```

- Sometimes you have a set of tests that can only run (and pass) if some set of conditions are just right. For instance, you might have to have an internet connection or be running on a specific operating system.
- Not running these tests doesn’t necessarily mean that the test suite should pass.
- Skipping tests is the answer. The skip function will fake that the tests passed and move to the end of the SKIP block.

A deeper look at Test::More

Basic Testing in Perl

```
TODO: {
  local $TODO = $reason;
  ok( 1 == 2, ‘one is two’ );
}
```

- It is possible to write a lot of tests before you write any code. You expect these tests to fail, but don’t want these tests to kill your test suite as you are developing.
- Marking the tests as “TODO” will cause Perl’s testing framework to expect the tests to pass and allow that to happen without killing the entire test suite.
- This isn’t always useful for TDD, but can be if you have a lot of good tests on your mind, but don’t have time to implement the code that makes them pass.

A deeper look at Test::More


```
BAIL_OUT( $reason )
```

- When everything is going wrong and you just need to stop running tests, ‘BAIL_OUT’ can save the day. It tells the Perl testing framework to stop running tests now.

- Typically you don’t want to do this because you want to get an assessment of what tests are actually failing. Still, in times of catastrophic failure, these could be useful.

A deeper look at Test::More

Basic Testing in Perl

- Now that we have a better understanding of Test::More, let's continue to develop our PercentOwnership module.

- We still need to write tests and functions to create a PercentOwnership object, add information about condos, and finally determine the percentage of ownership for each owner.

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
1 use warnings;
2 use strict;
3 use Test::More tests => 2;
4
5 BEGIN { use_ok('PercentOwnership') }
6
7 can_ok( 'PercentOwnership', 'new' );
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....
#   Failed test 'PercentOwnership->can('new')'
#   at t/percent_ownership.t line 7.
#     PercentOwnership->can('new') failed
# Looks like you failed 1 test of 2.
t/percent_ownership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 2
        Failed 1/2 tests, 50.00% okay
Failed Test           Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/percent_ownership.t    1   256     2    1  50.00%  2
Failed 1/1 test scripts, 0.00% okay. 1/2 subtests failed, 50.00% okay.
```

A deeper look at Test::More

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
1 package PercentOwnership;
2
3 use warnings;
4 use strict;
5
6 sub new {}
7
8 1;
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs ( 0.04 cusr +  0.02 csys =  0.06 CPU)
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
3 use Test::More tests => 3;
4
5 BEGIN { use_ok('PercentOwnership') }
6
7 can_ok( 'PercentOwnership', 'new' );
8 my $po = PercentOwnership->new();
9 isa_ok( $po, 'PercentOwnership' );
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....
#   Failed test 'The object isa PercentOwnership'
#   at t/percent_ownership.t line 9.
#     The object isn't defined
# Looks like you failed 1 test of 3.
t/percent_ownership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 3
        Failed 1/3 tests, 66.67% okay
Failed Test           Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------
t/percent_ownership.t    1   256     3    1  33.33%  3
Failed 1/1 test scripts, 0.00% okay. 1/3 subtests failed, 66.67% okay.
```

A deeper look at Test::More

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
 1 package PercentOwnership;
 2
 3 use warnings;
 4 use strict;
 5
 6 sub new {
 7     my ($class) = @_;
 8     my $self = bless {}, $class;
 9     return $self;
10 }
11
12 1;
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.04 cusr +  0.02 csys =  0.06 CPU)
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
 3 use Test::More tests => 4;
 4
 5 BEGIN { use_ok('PercentOwnership') };
 6
 7 can_ok( 'PercentOwnership', 'new' );
 8 my $po = PercentOwnership->new();
 9 isa_ok($po, 'PercentOwnership');
10
11 can_ok( $po, 'add_unit' );
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok 1/4
t/percent_ownership....NOK 4#   Failed test 'PercentOwnership->can('add_unit')'
#   at t/percent_ownership.t line 11.
#     PercentOwnership->can('add_unit') failed
# Looks like you failed 1 test of 4.
t/percent_ownership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 4
        Failed 1/4 tests, 75.00% okay
Failed Test           Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/percent_ownership.t    1   256     4    1  25.00%  4
Failed 1/1 test scripts, 0.00% okay. 1/4 subtests failed, 75.00% okay.
```

A deeper look at Test::More

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
 6 sub new {
 7     my ($class) = @_;
 8     my $self = bless {}, $class;
 9     return $self;
10 }
11
12 sub add_unit {}
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok
All tests successful.
Files=1, Tests=4,  1 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
 3 use Test::More tests => 4;
 4
 5 BEGIN { use_ok('PercentOwnership'); }
 6
 7 can_ok( 'PercentOwnership', 'new' );
 8 my $po = PercentOwnership->new();
 9 isa_ok( $po, 'PercentOwnership' );
10
11 can_ok( $po, qw(add_unit percent_ownership) );
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok 1/4
#   Failed test 'PercentOwnership->can(...)'
#   at t/percent_ownership.t line 11.
#     PercentOwnership->can('percent_ownership') failed
# Looks like you failed 1 test of 4.
t/percent_ownership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 4
        Failed 1/4 tests, 75.00% okay
Failed Test           Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/percent_ownership.t    1   256     4    1  25.00%  4
Failed 1/1 test scripts, 0.00% okay. 1/4 subtests failed, 75.00% okay.
```

A deeper look at Test::More

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
6  sub new {
7      my ($class) = @_;
8      my $self = bless {}, $class;
9      return $self;
10 }
11
12 sub add_unit {}
13
14 sub percent_ownership {}
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok
All tests successful.
Files=1, Tests=4,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
11 can_ok( $po, qw(add_unit percent_ownership) );
12
13 ok($po->add_unit(
14      unit_number    => 101,
15      square_footage => 450,
16      floor          => 1,
17 ), 'unit added successfully');
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok 1/5
#   Failed test 'unit added successfully'
#   at t/percent_ownership.t line 13.
t/percent_ownership....NOK 5# Looks like you failed 1 test of 5.
t/percent_ownership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 5
        Failed 1/5 tests, 80.00% okay
Failed Test           Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/percent_ownership.t    1   256     5    1  20.00%  5
Failed 1/1 test scripts, 0.00% okay. 1/5 subtests failed, 80.00% okay.
```

A deeper look at Test::More

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
11
12 sub add_unit { return 1; };
13
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok
All tests successful.
Files=1, Tests=5,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
19 is(
20     $po->percent_ownership( unit_number => 101 ),
21     100,
22     'single unit condo'
23 );
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....
#   Failed test 'single unit condo'
#   at t/percent_ownership.t line 19.
#          got: undef
#     expected: '100'
# Looks like you failed 1 test of 6.
t/percent_ownership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 6
        Failed 1/6 tests, 83.33% okay
Failed Test           Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/percent_ownership.t    1   256     6    1  16.67%  6
Failed 1/1 test scripts, 0.00% okay. 1/6 subtests failed, 83.33% okay.
```

A deeper look at Test::More

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
 1 package PercentOwnership;
 2
 3 use warnings;
 4 use strict;
 5
 6 sub new {
 7     my ($class) = @_;
 8     my $self = bless {}, $class;
 9     return $self;
10 }
11
12 sub add_unit { return 1; }
13
14 sub percent_ownership { return 100; }
15
16 1;
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok
All tests successful.
Files=1, Tests=6,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
 9 SINGLE_UNIT: {
10     my $po = PercentOwnership->new();
11     isa_ok( $po, 'PercentOwnership' );
12
13     can_ok( $po, qw(add_unit percent_ownership) );
14
15     ok($po->add_unit(
16         unit_number    => 101,
17         square_footage => 450,
18         floor          => 1,
19     ), 'unit added successfully');
20
21     is(
22         $po->percent_ownership( unit_number => 101 ),
23         100,
24         'single unit condo'
25     );
26 }
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
28 TWO_UNITS: {
29      my $po = PercentOwnership->new();
30      isa_ok( $po, 'PercentOwnership' );
31
32      can_ok( $po, qw(add_unit percent_ownership) );
33

...

57 }
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
...

34      ok($po->add_unit(
35          unit_number    => 101,
36          square_footage => 450,
37          floor          => 1,
38      ), 'first unit added successfully');
39
40      ok($po->add_unit(
41          unit_number    => 102,
42          square_footage => 450,
43          floor          => 1,
44      ), 'second unit added successfully');
45

...
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
...

46      is(
47          $po->percent_ownership( unit_number => 101 ),
48          50,
49          '50/50 ownership split for unit 101'
50      );
51
52      is(
53          $po->percent_ownership( unit_number => 102 ),
54          50,
55          '50/50 ownership split for unit 102'
56      );
57  }
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok 1/12
#   Failed test '50/50 ownership split for unit 101'
#   at t/percent_ownership.t line 46.
#          got: '100'
#     expected: '50'
t/percent_ownership....NOK 11
#   Failed test '50/50 ownership split for unit 102'
#   at t/percent_ownership.t line 52.
#          got: '100'
#     expected: '50'
# Looks like you failed 2 tests of 12.
t/percent_ownership....dubious
        Test returned status 2 (wstat 512, 0x200)
DIED. FAILED tests 11-12
        Failed 2/12 tests, 83.33% okay
Failed Test           Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/percent_ownership.t    2   512    12    2  16.67%  11-12
Failed 1/1 test scripts, 0.00% okay. 2/12 subtests failed, 83.33% okay.
```

A deeper look at Test::More

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
 5 use List::Util qw(sum);
...
13 sub add_unit {
14     my ( $self, %unit_info ) = @_;
15
16     $self->{unit_info}->{ $unit_info{unit_number} } = \%unit_info;
17 }
18
19 sub percent_ownership {
20     my ( $self, %args ) = @_;
21
22     my $building_size = sum map {
           $self->{unit_info}->{$_}->{square_footage} }
23       keys %{ $self->{unit_info} };
24
25     my $unit_size =
26       $self->{unit_info}->{ $args{unit_number} }->{square_footage};
27
28     return sprintf( "%0.4f", $unit_size / $building_size ) * 100;
29 }
```

A deeper look at Test::More

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok
All tests successful.
Files=1, Tests=12,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

Exercise One

- Using Test::More and prove, write a module that converts temperatures from Celsius to Fahrenheit.
- After the conversion starts working with Celsius to Fahrenheit, add a method to convert from Fahrenheit to Celsius.
- Now, add some testing to see how the methods work with potentially damaging cases such as fractional numbers, negative temperatures, and non-numeric input.

Taking a look at the TAP

- What is ‘prove’ doing when it runs our tests?
  - It is running our testing scripts using ‘perl’.
  - Then intercepting the output.
  - And finally formatting the output in a nice summarized form.

Taking a look at the TAP

Basic Testing in Perl

```
--(0)> perl -Ilib t/percent_ownership.t
1..12
ok 1 - use PercentOwnership;
ok 2 - PercentOwnership->can('new')
ok 3 - The object isa PercentOwnership
ok 4 - PercentOwnership->can(...)
ok 5 - unit added successfully
ok 6 - single unit condo
ok 7 - The object isa PercentOwnership
ok 8 - PercentOwnership->can(...)
ok 9 - first unit added successfully
ok 10 - second unit added successfully
ok 11 - 50/50 ownership split for unit 101
ok 12 - 50/50 ownership split for unit 102
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
1 use warnings;
2 use strict;
3 use Test::More qw(no_plan);
4
```

Taking a look at the TAP


```
--(0)> perl -Ilib t/percent_ownership.t
ok 1 - use PercentOwnership;
ok 2 - PercentOwnership->can('new')
ok 3 - The object isa PercentOwnership
ok 4 - PercentOwnership->can(...)
ok 5 - unit added successfully
ok 6 - single unit condo
ok 7 - The object isa PercentOwnership
ok 8 - PercentOwnership->can(...)
ok 9 - first unit added successfully
ok 10 - second unit added successfully
ok 11 - 50/50 ownership split for unit 101
ok 12 - 50/50 ownership split for unit 102
1..12
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
52      is(
53          $po->percent_ownership( unit_number => 102 ),
54          51,
55          '50/50 ownership split for unit 102'
56      );
```

Taking a look at the TAP

Basic Testing in Perl

```
--(0)> perl -Ilib t/percent_ownership.t
1..12
ok 1 - use PercentOwnership;
ok 2 - PercentOwnership->can('new')
ok 3 - The object isa PercentOwnership
ok 4 - PercentOwnership->can(...)
ok 5 - unit added successfully
ok 6 - single unit condo
ok 7 - The object isa PercentOwnership
ok 8 - PercentOwnership->can(...)
ok 9 - first unit added successfully
ok 10 - second unit added successfully
ok 11 - 50/50 ownership split for unit 101
not ok 12 - 50/50 ownership split for unit 102
#   Failed test '50/50 ownership split for unit 102'
#   at t/percent_ownership.t line 52.
#          got: '50'
#     expected: '51'
# Looks like you failed 1 test of 12.
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
57 SKIP: {
58     skip "The conditions are not right to run this test", 1;
59     ok(
60         $po->something_that_requires_network_access,
61         'did a network operation'
62     );
63 }
```

Taking a look at the TAP

Basic Testing in Perl

```
--(0)> perl -Ilib t/percent_ownership.t
1..13
ok 1 - use PercentOwnership;
ok 2 - PercentOwnership->can('new')
ok 3 - The object isa PercentOwnership
ok 4 - PercentOwnership->can(...)
ok 5 - unit added successfully
ok 6 - single unit condo
ok 7 - The object isa PercentOwnership
ok 8 - PercentOwnership->can(...)
ok 9 - first unit added successfully
ok 10 - second unit added successfully
ok 11 - 50/50 ownership split for unit 101
ok 12 - 50/50 ownership split for unit 102
ok 13 # skip The conditions are not right to run this test
```

A deeper look at Test::More

Basic Testing in Perl

"t/percent_ownership.t"

```
58 TODO: {
59     local $TODO =
60         "Need to return an undefined for non-existant units";
61     is(
62         $po->percent_ownership( unit_number => 103 ),
63         undef,
64         'no ownership'
65     );
66 }
```

Taking a look at the TAP

Basic Testing in Perl

```
--(0)> perl -Ilib t/percent_ownership.t
1..13
ok 1 - use PercentOwnership;
ok 2 - PercentOwnership->can('new')
ok 3 - The object isa PercentOwnership
ok 4 - PercentOwnership->can(...)
ok 5 - unit added successfully
ok 6 - single unit condo
ok 7 - The object isa PercentOwnership
ok 8 - PercentOwnership->can(...)
ok 9 - first unit added successfully
ok 10 - second unit added successfully
ok 11 - 50/50 ownership split for unit 101
ok 12 - 50/50 ownership split for unit 102
not ok 13 - no ownership # TODO Need to return an undefined for non-existant units
#   Failed (TODO) test 'no ownership'
#   at t/percent_ownership.t line 61.
#          got: '0'
#     expected: undef
```

Taking a look at the TAP

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok
All tests successful.
Files=1, Tests=13,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

A deeper look at Test::More

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
19 sub percent_ownership {
20     my ($self, %args) = @_;
21
22     return unless exists $self->{unit_info}->{ $args{unit_number} };
23
24     my $building_size = sum map
           { $self->{unit_info}->{$_}->{square_footage} }
25         keys %{ $self->{unit_info} };
26
27     my $unit_size =
28         $self->{unit_info}->{ $args{unit_number} }->{square_footage};
29
30     return sprintf( "%0.4f", $unit_size / $building_size ) * 100;
31 }
```

Taking a look at the TAP

Basic Testing in Perl

```
--(0)> perl -Ilib t/percent_ownership.t
1..13
ok 1 - use PercentOwnership;
ok 2 - PercentOwnership->can('new')
ok 3 - The object isa PercentOwnership
ok 4 - PercentOwnership->can(...)
ok 5 - unit added successfully
ok 6 - single unit condo
ok 7 - The object isa PercentOwnership
ok 8 - PercentOwnership->can(...)
ok 9 - first unit added successfully
ok 10 - second unit added successfully
ok 11 - 50/50 ownership split for unit 101
ok 12 - 50/50 ownership split for unit 102
ok 13 - no ownership # TODO Need to return an undefined for non-existant units
```

Taking a look at the TAP

Basic Testing in Perl

```
--(0)> prove -Ilib t/percent_ownership.t
t/percent_ownership....ok
        1/13 unexpectedly succeeded
All tests successful (1 subtest UNEXPECTEDLY SUCCEEDED).
Files=1, Tests=13,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

Taking a look back at Test::Simple

- Sometimes you might encounter code that uses Test::Simple.
- Test::Simple pre-dates test more.
- There is only an ‘ok’ function.
- The plan must be specified on the ‘use’ line.

Object Oriented Test Frameworks

- Understanding Objected Oriented Testing
- Porting our tests to use Test::Class
- Adding more tests with Test::Class

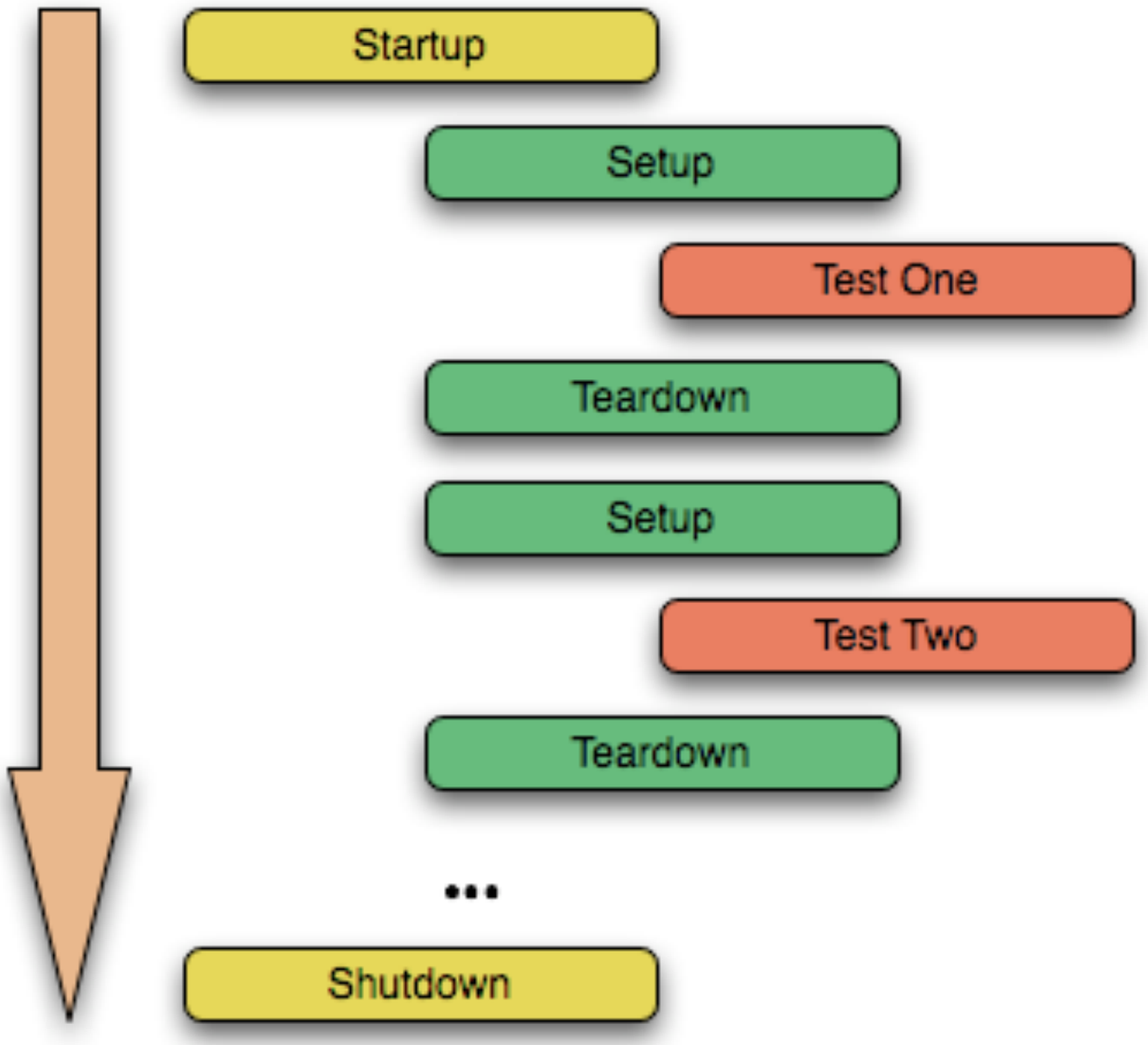
Understanding Object Oriented Testing

Object Oriented Test Frameworks

- Object oriented testing is not necessarily about testing objects, but is instead about using objects to control your tests.
- JUnit and TestNG are examples of popular OO test frameworks.
- Many OO frameworks use the xUnit design.
- As always, Perl has more than one way to do it. There are many frameworks such a Test::Unit (a port of JUnit) and Test::Class.

Understanding Object Oriented Testing

Object Oriented Test Frameworks



Porting our tests to use Test::Class

Object Oriented Test Frameworks

"t/PercentOwnership.pm"

```
1 package TestPercentOwnership;
2
3 use warnings;
4 use strict;
5 use base qw(Test::Class);
6 use Test::More;
7
8 __PACKAGE__->runtests unless caller;
9
10 sub my_test : Test {
11     pass;
12 }
13
14 1;
```

Porting our tests to use Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.10 cusr +  0.03 csys =  0.13 CPU)
```

Porting our tests to use Test::Class

"t/PercentOwnership.pm"

```
 1 package TestPercentOwnership;
 2
 3 use warnings;
 4 use strict;
 5 use base qw(Test::Class);
 6 use Test::More;
 7
 8 __PACKAGE__->runtests unless caller;
 9
10 sub startup_test : Test( startup => 2 ) {
11     use_ok( 'PercentOwnership' );
12     can_ok( 'PercentOwnership', 'new' );
13 }
14
15 1;
```

Porting our tests to use Test::Class

Basic Testing in Perl

```
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=2,  1 wallclock secs ( 0.11 cusr +  0.03 csys =  0.14 CPU)
```

Porting our tests to use Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
15 sub single_unit : Test(4) {
16     my ($self) = @_;
17     my $po = PercentOwnership->new();
18     isa_ok( $po, 'PercentOwnership' );
19     can_ok( 'PercentOwnership', qw( add_unit percent_ownership ) );
20
21     ok(
22         $po->add_unit(
23             unit_number    => 101,
24             square_footage => 450,
25             floor          => 1,
26         ), ‘added single unit’
27     );
28
29     is( $po->percent_ownership( unit_number => 101 ), 100,
30             'single unit condo' );
31 }
```

Porting our tests to use Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=6,  1 wallclock secs ( 0.04 cusr +  0.02 csys =  0.06 CPU)
```

Porting our tests to use Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
33 sub two_units : Test(6) {
34     my ($self) = @_;
35     my $po = PercentOwnership->new();
36     isa_ok( $po, 'PercentOwnership' );
37     can_ok( 'PercentOwnership', qw( add_unit percent_ownership ) );
38
```

Porting our tests to use Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
39      ok(
40          $po->add_unit(
41              unit_number    => 101,
42              square_footage => 450,
43              floor          => 1,
44          ), ‘added first unit’
45      );
46
47      ok(
48          $po->add_unit(
49              unit_number    => 102,
50              square_footage => 450,
51              floor          => 1,
52          ), ‘added second unit’
53      );
```

Porting our tests to use Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
54
55      is( $po->percent_ownership( unit_number => 101 ), 50,
56              'unit 101 got the correct percentage' );
57
58      is( $po->percent_ownership( unit_number => 102 ), 50,
59              'unit 102 got the correct percentage' );
60  }
```

Porting our tests to use Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=12,  0 wallclock secs ( 0.04 cusr +  0.02 csys =  0.06 CPU)
```

Porting our tests to use Test::Class

"t/PercentOwnership.pm"

```
15 sub setup_test : Test( setup => 2 ) {
16     my ($self) = @_;
17     $self->{po} = PercentOwnership->new();
18     isa_ok( $self->{po}, 'PercentOwnership' );
19     can_ok( $self->{po}, qw(add_unit percent_ownership) );
20 }
```

Porting our tests to use Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
22 sub single_unit : Test(2) {
23     my ($self) = @_;
24     my $po = $self->{po};
25
26     ok(
...
38 sub two_units : Test(4) {
39     my ($self) = @_;
40     my $po = $self->{po};
41
42     ok(
```

Porting our tests to use Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=12,  0 wallclock secs ( 0.04 cusr +  0.02 csys =  0.06 CPU)
```

Porting our tests to use Test::Class


```
--(0)> perl -Ilib t/TestPercentOwnership.pm
1..12
ok 1 - use PercentOwnership;
ok 2 - PercentOwnership->can('new')
ok 3 - The object isa PercentOwnership
ok 4 - PercentOwnership->can(...)
ok 5 - added single unit
ok 6 - single unit condo
ok 7 - The object isa PercentOwnership
ok 8 - PercentOwnership->can(...)
ok 9 - added first unit
ok 10 - added second unit
ok 11 - unit 101 got the correct percentage
ok 12 - unit 102 got the correct percentage
```

Porting our tests to use Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
 7 use List::MoreUtils qw(pairwise);
 8
 9 our ( $a, $b );
10
11 __PACKAGE__->runtests unless caller;
12
13 sub unit_adder {
14     my ( $self, @args ) = @_;
15     my @arg_names = qw(unit_number square_footage floor);
16     $self->{po}->add_unit(
           pairwise { ( $a, $b ) } @arg_names, @args );
17 }
```

Porting our tests to use Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
31 sub single_unit : Test(2) {
32     my ($self) = @_;
33     my $po = $self->{po};
34
35     ok( $self->unit_adder( 101, 450, 1 ), 'added the only unit' );
36
37     is( $po->percent_ownership( unit_number => 101 ), 100,
38             'single unit condo' );
39 }
```

Porting our tests to use Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
41 sub two_units : Test(4) {
42     my ($self) = @_;
43     my $po = $self->{po};
44
45     ok( $self->unit_adder( 101, 450, 1 ), 'added the first unit' );
46     ok( $self->unit_adder( 102, 450, 1 ), 'added the second unit' );
47
48     is( $po->percent_ownership( unit_number => 101 ), 50,
49             'unit 101 got the correct percentage' );
50
51     is( $po->percent_ownership( unit_number => 102 ), 50,
52             'unit 102 got the correct percentage' );
53 }
```

Porting our tests to use Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=12,  1 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Adding more tests with Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
55 sub multipe_floors : Test(2) {
56     my ($self) = @_;
57
58     $self->unit_adder( @{$_} )
59         for ( [ 101, 500, 1 ], [ 201, 500, 2 ] );
60
61     is(
62         $self->{po}->percent_ownership( unit_number => 101 ),
63         49.75,
64         'multiple floors - 101'
65     );
66
67     is(
68         $self->{po}->percent_ownership( unit_number => 201 ),
69         50.25,
70         'multiple floors - 201'
71     );
72 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok 1/16
t/TestPercentOwnership....NOK 5/16#   Failed test 'multiple floors - 101'
#   at t/TestPercentOwnership.pm line 61.
#   (in TestPercentOwnership->multipe_floors)
#          got: '50'
#     expected: '49.75'

#   Failed test 'multiple floors - 201'
#   at t/TestPercentOwnership.pm line 67.
#   (in TestPercentOwnership->multipe_floors)
#          got: '50'
#     expected: '50.25'
t/TestPercentOwnership....NOK 6/16# Looks like you failed 2 tests of 16.
t/TestPercentOwnership....dubious
        Test returned status 2 (wstat 512, 0x200)
DIED. FAILED tests 5-6
        Failed 2/16 tests, 87.50% okay
Failed Test                  Stat Wstat Total Fail  List of Failed
-------------------------------------------------------------------------------
t/TestPercentOwnership.pm    2   512    16    2  5-6
Failed 1/1 test scripts. 2/16 subtests failed.
Files=1, Tests=16,  1 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Adding more tests with Test::Class

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
18
19 sub percent_ownership {
20     my ($self, %args) = @_;
21
22     my $adjusted_building_size = sum map {
23         $self->{unit_info}->{$_}->{square_footage} *
24             sprintf( "1.%02d", $self->{unit_info}->{$_}->{floor} )
25         } keys %{ $self->{unit_info} };
26
27     my $adjusted_unit_size =
28         $self->{unit_info}->{ $args{unit_number} }->{square_footage} *
29         sprintf( "1.%02d",
           $self->{unit_info}->{ $args{unit_number} }->{floor} );
30
31     return sprintf( "%0.4f",
           $adjusted_unit_size / $adjusted_building_size ) * 100;
32 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=16,  1 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Adding more tests with Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
74 sub basement_units : Test(2) {
75     my ($self) = @_;
76     $self->unit_adder( @{$_} )
77         for( [ '001', 500, 0 ], [ 101, 500, 1 ], );
78     is(
79         $self->{po}->percent_ownership( unit_number => '001' ),
80         49.75,
81         'basement units - 001'
82     );
83     is(
84         $self->{po}->percent_ownership( unit_number => '101' ),
85         50.25,
86         'basement units - 101'
87     );
88 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=20,  1 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Adding more tests with Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
 90 sub subbasement_units : Test(2) {
 91     my ($self) = @_;
 92     $self->unit_adder( @{$_} )
 93         for( [ '001', 500, -1 ], [ 101, 500, 1 ], );
 94     is(
 95         $self->{po}->percent_ownership( unit_number => '001' ),
 96         49.75,
 97         'subbasement units - 001'
 98     );
 99     is(
100         $self->{po}->percent_ownership( unit_number => '101' ),
101         50.25,
102         'subbasement units - 101'
103     );
104 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok 1/24Argument "1.-1" isn't numeric in multiplication (*) at
lib/PercentOwnership.pm line 22.
Argument "1.-1" isn't numeric in multiplication (*) at lib/PercentOwnership.pm line 27.
Argument "1.-1" isn't numeric in multiplication (*) at lib/PercentOwnership.pm line 22.
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=24,  1 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Adding more tests with Test::Class

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
13 sub add_unit {
14     my ($self, %unit_info) = @_;
15
16     $unit_info{floor} = 0 if( $unit_info{floor} < 0 );
17
18     $self->{unit_info}->{ $unit_info{unit_number} } = \%unit_info;
19 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=24,  0 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Exercise Two

- Port your temperature converter to Test::Class.
- Add some edge case tests to make sure that your converter can handle strange values in a predictable (and non-fatal) way.

Adding more tests with Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
106 sub non_existant_unit : Test(1) {
107     my ($self) = @_;
108     $self->unit_adder( 100, 500, 1 );
109     is(
110         $self->{po}->percent_ownership( unit_number => 0 ),
111         undef,
112         'return undef for non-existant units'
113     );
114 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok 1/27Use of uninitialized value in sprintf at lib/
PercentOwnership.pm line 29.
Use of uninitialized value in multiplication (*) at lib/PercentOwnership.pm line 29.

#   Failed test 'return undef for non-existant units'
#   at t/TestPercentOwnership.pm line 109.
#   (in TestPercentOwnership->non_existant_unit)
#          got: '0'
#     expected: undef
t/TestPercentOwnership....NOK 13/27# Looks like you failed 1 test of 27.
t/TestPercentOwnership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 13
        Failed 1/27 tests, 96.30% okay
Failed Test                Stat Wstat Total Fail  List of Failed
-------------------------------------------------------------------------------
t/TestPercentOwnership.pm    1   256    27    1  13
Failed 1/1 test scripts. 1/27 subtests failed.
Files=1, Tests=27,  0 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
Failed 1/1 test programs. 1/27 subtests failed.
```

Adding more tests with Test::Class

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
21 sub percent_ownership {
22     my ($self, %args) = @_;
23
24     return unless exists $self->{unit_info}->{$args{unit_number}};
25
26     my $adjusted_building_size = sum map {
27         $self->{unit_info}->{$_}->{square_footage} *
28             sprintf( "1.%02d", $self->{unit_info}->{$_}->{floor} )
29         } keys %{ $self->{unit_info} };
30
31     my $adjusted_unit_size =
32         $self->{unit_info}->{ $args{unit_number} }->{square_footage} *
33         sprintf( "1.%02d", $self->{unit_info}->{ $args{unit_number} }->{floor} );
34
35     return sprintf( "%0.4f", $adjusted_unit_size / $adjusted_building_size ) * 100;
36 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=27,  1 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Adding more tests with Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
116 sub add_unit_with_no_arguments : Test(1) {
117     my ($self) = @_;
118     is(
119         $self->{po}->add_unit,
120         undef,
121         'no arguments provided to add_unit'
122     );
123 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok 1/30Use of uninitialized value in numeric lt (<) at lib/
PercentOwnership.pm line 16.
Use of uninitialized value in hash element at lib/PercentOwnership.pm line 18.

#   Failed test 'no arguments provided to add_unit'
#   at t/TestPercentOwnership.pm line 118.
#   (in TestPercentOwnership->add_unit_with_no_arguments)
#          got: 'HASH(0x86fde0)'
#     expected: undef
t/TestPercentOwnership....NOK 5/30# Looks like you failed 1 test of 30.
t/TestPercentOwnership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 5
        Failed 1/30 tests, 96.67% okay
Failed Test                Stat Wstat Total Fail  List of Failed
-------------------------------------------------------------------------------
t/TestPercentOwnership.pm    1   256    30    1  5
Failed 1/1 test scripts. 1/30 subtests failed.
Files=1, Tests=30,  0 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
Failed 1/1 test programs. 1/30 subtests failed.
```

Adding more tests with Test::Class

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
13 sub add_unit {
14     my ($self, %unit_info) = @_;
15
16     return unless
17         (grep { exists $unit_info{$_} }
18             qw(unit_number square_footage floor)) == 3;
19
20     $unit_info{floor} = 0 if( $unit_info{floor} < 0 );
21
22     $self->{unit_info}->{ $unit_info{unit_number} } = \%unit_info;
23 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=30,  0 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Adding more tests with Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
125 sub percent_ownership_with_no_arguments : Test(1) {
126     my ($self) = @_;
127     $self->unit_adder( 100, 500, 1 );
128     is(
129         $self->{po}->percent_ownership,
130         undef,
131         'no arguments provided to percent_ownership'
132     );
133 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok 1/33Use of uninitialized value in exists at lib/
PercentOwnership.pm line 28.
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=33,  0 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Adding more tests with Test::Class

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
25 sub percent_ownership {
26     my ($self, %args) = @_;
27
28     return unless exists $args{unit_number};
29     return unless exists $self->{unit_info}->{$args{unit_number}};
30
31     my $adjusted_building_size = sum map {
32         $self->{unit_info}->{$_}->{square_footage} *
33             sprintf( "1.%02d", $self->{unit_info}->{$_}->{floor} )
34         } keys %{ $self->{unit_info} };
35
36     my $adjusted_unit_size =
37         $self->{unit_info}->{ $args{unit_number} }->{square_footage} *
38         sprintf( "1.%02d", $self->{unit_info}->{ $args{unit_number} }->{floor} );
39
40     return sprintf( "%0.4f", $adjusted_unit_size / $adjusted_building_size ) * 100;
41 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=33,  1 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Adding more tests with Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
135 sub add_unit_with_invalid_square_footage : Test(1) {
136     my ($self) = @_;
137     is(
138         $self->unit_adder( 100, -500, 1 ),
139         undef,
140         'negative square footage'
141     );
142 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok 1/36
#   Failed test 'negative square footage'
#   at t/TestPercentOwnership.pm line 137.
#   (in TestPercentOwnership->add_unit_with_invalid_square_footage)
#          got: 'HASH(0x871290)'
#     expected: undef
t/TestPercentOwnership....NOK 5/36# Looks like you failed 1 test of 36.
t/TestPercentOwnership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 5
        Failed 1/36 tests, 97.22% okay
Failed Test                Stat Wstat Total Fail  List of Failed
-------------------------------------------------------------------------------
t/TestPercentOwnership.pm    1   256    36    1  5
Failed 1/1 test scripts. 1/36 subtests failed.
Files=1, Tests=36,  1 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
Failed 1/1 test programs. 1/36 subtests failed.
```

Adding more tests with Test::Class

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
13 sub add_unit {
14     my ($self, %unit_info) = @_;
15
16     return unless
17         (grep { exists $unit_info{$_} }
18             qw(unit_number square_footage floor)) == 3;
19
20     return if $unit_info{square_footage} < 0;
21
22     $unit_info{floor} = 0 if( $unit_info{floor} < 0 );
23
24     $self->{unit_info}->{ $unit_info{unit_number} } = \%unit_info;
25 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=36,  0 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Adding more tests with Test::Class

Basic Testing in Perl

"t/PercentOwnership.pm"

```
144 sub add_unit_with_string_square_footage : Test(1) {
145     my ($self) = @_;
146     is(
147         $self->unit_adder( 100, 'oops', 1 ),
148         undef,
149         'string square footage'
150     );
151 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok 1/39Argument "oops" isn't numeric in numeric lt (<) at lib/
PercentOwnership.pm line 20.

#   Failed test 'string square footage'
#   at t/TestPercentOwnership.pm line 146.
#   (in TestPercentOwnership->add_unit_with_string_square_footage)
#          got: 'HASH(0x8723c8)'
#     expected: undef
t/TestPercentOwnership....NOK 11/39# Looks like you failed 1 test of 39.
t/TestPercentOwnership....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 11
        Failed 1/39 tests, 97.44% okay
Failed Test                Stat Wstat Total Fail  List of Failed
-------------------------------------------------------------------------------
t/TestPercentOwnership.pm    1   256    39    1  11
Failed 1/1 test scripts. 1/39 subtests failed.
Files=1, Tests=39,  0 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
Failed 1/1 test programs. 1/39 subtests failed.
```

Adding more tests with Test::Class

Basic Testing in Perl

"lib/PercentOwnership.pm"

```
13 sub add_unit {
14     my ($self, %unit_info) = @_;
15
16     return unless
17         (grep { exists $unit_info{$_} }
18             qw(unit_number square_footage floor)) == 3;
19
20     return if $unit_info{square_footage} =~ /\D/;;
21     return if $unit_info{square_footage} < 0;
22
23     $unit_info{floor} = 0 if( $unit_info{floor} < 0 );
24
25     $self->{unit_info}->{ $unit_info{unit_number} } = \%unit_info;
26 }
```

Adding more tests with Test::Class

Basic Testing in Perl

```
--(0)> prove -Ilib t/TestPercentOwnership.pm
t/TestPercentOwnership....ok
All tests successful.
Files=1, Tests=39,  1 wallclock secs ( 0.05 cusr +  0.02 csys =  0.07 CPU)
```

Testing Utilities

- Testing and reporting on complex data.
- Setting up your tests with mock objects.
- Determining test coverage.

Testing and reporting on complex data

Testing Utilities

- Not all tests can easily be performed with a simple binary expression.
- Sometimes the data being tested is large enough the presenting differences between what was received and what was expected is challenging.
- Test::Differences and Test::Deep can help simplify testing of large and/or complex data structures.

Testing and reporting on complex data

Testing Utilities

```
use Test::Differences;
eq_or_diff( $got, $expected, $test_name )
```

- Test::Differences picks up where Test::More's 'is_deeply' leaves off by showing side-by-side differences in multiline strings or complex data structures.

- This module is only really reliable on Perl's > 5.8

Testing and reporting on complex data

Testing Utilities

"scalar_is_deeply.t"

```
 3 use Test::More qw(no_plan);
 4
 5 my $expected = 'This is a multiline
 6 string of text
 7 that is not the easiest thing to
 8 display.';
 9
10 my $got = $expected;
11 substr($got, 12, 1) = 'i';
12
13 is $got, $expected, 'The are who we thought they were';
```

Testing and reporting on complex data

Testing Utilities

```
--(0)> prove scalar_is_deeply.t
is_deeply....
#   Failed test 'The are who we thought they were'
#   at is_deeply.t line 13.
#          got: 'This is a muitiline
# string of text
# that is not the easiest thing to
# display.'
#     expected: 'This is a multiline
# string of text
# that is not the easiest thing to
# display.'
# Looks like you failed 1 test of 1.
is_deeply....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
is_deeply.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
```

Testing and reporting on complex data

Testing Utilities

"scalar_test_differences.t"

```
 3 use Test::More qw(no_plan);
 4 use Test::Differences;
 5
 6 my $expected = 'This is a multiline
 7 string of text
 8 that is not the easiest thing to
 9 display.';
10
11 my $got = $expected;
12 substr($got, 12, 1) = 'i';
13
14 eq_or_diff $got, $expected,
       'They are who we thought they were';
```

Testing and reporting on complex data

Testing Utilities

```
--(0)> prove scalar_test_differences.t
test_differences....NOK 1
#   Failed test 'They are who we thought they were'
#   at test_differences.t line 14.
# +---+----------------------------------+----------------------------------+
# | Ln|Got                               |Expected                          |
# +---+----------------------------------+----------------------------------+
# *  1|This is a muitiline               |This is a multiline               *
# |  2|string of text                    |string of text                    |
# |  3|that is not the easiest thing to  |that is not the easiest thing to  |
# |  4|display.                          |display.                          |
# +---+----------------------------------+----------------------------------+
# Looks like you failed 1 test of 1.
test_differences....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test        Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
test_differences.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
```

Testing and reporting on complex data

Testing Utilities

"ref_is_deeply.t"

```
1 use warnings;
2 use strict;
3 use Test::More qw(no_plan);
4
5 my $expected = { name => 'Josh',
                   pets => [qw( ella ginger )] };
6 my $got = bless {%$expected}, 'Person';
7 $got->{name} = 'Heather';
8
9 is $got, $expected, 'Structures are different';
```

Testing and reporting on complex data

Testing Utilities

```
--(0)> prove ref_is_deeply.t
ref_is_deeply....
#   Failed test 'Structures are different'
#   at ref_is_deeply.t line 9.
#          got: 'Person=HASH(0x183beb8)'
#     expected: 'HASH(0x183be40)'
# Looks like you failed 1 test of 1.
ref_is_deeply....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test     Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
ref_is_deeply.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
```

Testing and reporting on complex data

Testing Utilities

"ref_test_differences.t"

```
 1 use warnings;
 2 use strict;
 3 use Test::More qw(no_plan);
 4 use Test::Differences;
 5
 6 my $expected = { name => 'Josh',
                   pets => [qw( ella ginger )] };
 7 my $got = bless {%$expected}, 'Person';
 8 $got->{name} = 'Heather';
 9
10 eq_or_diff $got, $expected,
       'Structures are different';
```

Testing and reporting on complex data

Testing Utilities

```
--(0)> perl ref_test_differences.t
not ok 1 - Structures are different
#   Failed test 'Structures are different'
#   at ref_test_differences.t line 10.
# +----+----------------------+--------------------+
# | Elt|Got                   |Expected            |
# +----+----------------------+--------------------+
# *    0|bless( {              |{                   *
# *    1|  name => 'Heather',  |  name => 'Josh',   *
# |    2|  pets => [           |  pets => [         |
# |    3|    'ella',           |    'ella',         |
# |    4|    'ginger'          |    'ginger'        |
# |    5|  ]                   |  ]                 |
# *    6|}, 'Person' )         |}                   *
# +----+----------------------+--------------------+
1..1
# Looks like you failed 1 test of 1.
```

Testing and reporting on complex data

Testing Utilities

- Test::Deep is another module that can really help you test complex data structures.
- While utilities like ‘is_deeply’ and ‘eq_or_diff’ are nice for comparing entire data structures, they make you compare the ENTIRE data structure, as is.
- Sometimes you need more flexibility. Test::Deep gives you this.

Testing and reporting on complex data

Testing Utilities

```
use Test::Deep;
cmp_deeply( $got, $expected, $test_name )
```

- By default this looks like pretty much any other of the test methods that we've looked at.

- It doesn't do the nice side-by-side comparisons that Test::Differences does.

- The true power is in how you can manipulate your expected data structure to work around minor differences between it and the data structure that you got back from your test.

Testing and reporting on complex data

Testing Utilities

"ignore.t"

```
 3 use Test::More qw(no_plan);
 4 use Test::Deep;
 5
 6 ARRAY: {
 7     my $got = [qw(one two three)];
 8     my $expected = [ qw(one two), ignore() ];
 9     cmp_deeply( $got, $expected, 'deeply compared array' );
10 }
11
12 HASH: {
13     my $got = { one => 1, two => 2, three => 3 };
14     my $expected = {
                  one => 1, two => ignore(), three => 3 };
15     cmp_deeply( $got, $expected, 'deeply compared hash' );
16 }
```

Testing and reporting on complex data

Testing Utilities

```
--(0)> prove ignore.t
ignore....ok
All tests successful.
Files=1, Tests=2,  1 wallclock secs ( 0.10 cusr +  0.03 csys =  0.13 CPU)
```

Testing and reporting on complex data

Testing Utilities

"array.t"

```
3 use Test::More qw(no_plan);
4 use Test::Deep;
5
6 cmp_deeply( [ 1, 1, 2, 2 ], bag( 2, 1, 2, 1 )
    , 'bag of numbers' );
7 cmp_deeply( [ 1, 1, 2, 2, 3 ], superbagof( 2, 1, 2, 1 )
    , 'super-bag of numbers' );
8 cmp_deeply( [ 1, 1 ], subbagof( 2, 1, 2, 1 )
    , 'sub-bag of numbers' );
9
10 cmp_deeply( [ 1, 1, 2, 2 ], set( 1, 2 )
    , 'set of numbers' );
11 cmp_deeply( [ 1, 1, 2, 2, 3 ], supersetof( 1, 2 )
    , 'super-set of numbers' );
12 cmp_deeply( [ 1, 1, 2, 2 ], subsetof( 1, 2, 3 )
    , 'sub-set of numbers' );
```

Testing and reporting on complex data

Testing Utilities

"hash.t"

```
3 use Test::More qw(no_plan);
4 use Test::Deep;
5
6 cmp_deeply( { one => 1 },
      subhashof( { two => 2, one => 1 } ), 'sub-hash' );
7 cmp_deeply( { one => 1, two => 2 },
      superhashof( { one => 1 } ), 'super-hash' );
```

Testing and reporting on complex data

Testing Utilities

"objective.t"

```
 3 use Test::More qw(no_plan);
 4 use Test::Deep;
 5
 6 cmp_deeply( bless( { name => 'x' }, 'Thing' ),
 7     isa('Thing'), 'looking at the isa' );
 8 cmp_deeply(
 9     bless( { name => 'x' }, 'Thing' ),
10     noclass( { name => 'x' } ),
11     'classless compare'
12 );
13 cmp_deeply(
14     bless( { name => bless( {}, 'Name' ) }, 'Thing' ),
15     noclass( { name => useclass( bless( {}, 'Name' ) ) } ),
16     'classless compare'
17 );
```

Testing and reporting on complex data

Testing Utilities

"methods.t"

```
 3 use Test::More qw(no_plan);
 4 use Test::Deep;
 5 use Test::MockObject;
 6
 7 my $obj = Test::MockObject->new();
 8 $obj->mock( 'uc_it', sub { return uc $_[1] } );
 9 $obj->mock( 'do_it', sub { return 1 } );
10
11 cmp_deeply( $obj
       , methods( do_it => 1, [qw(uc_it one)] => 'ONE' ) );
```

Testing and reporting on complex data

Testing Utilities

"all_any.t"

```
3 use Test::More qw(no_plan);
4 use Test::Deep;
5
6 my $obj = bless { name => 'x' }, 'Thing';
7
8 cmp_deeply( $obj,
9     all( isa('Thing')
       , noclass(
         subhashof( { name => 'x', rank => 'y' } )
        ) ) );
10 cmp_deeply( $obj,
        any( isa('Thing'), isa('OtherThing') ) );
```

Testing and reporting on complex data

Testing Utilities

"basic.t"

```
3 use Test::More qw(no_plan);
4 use Test::Deep;
5
6 cmp_deeply( { one => '1one' },
              { one => num(1) },      'num' );
7 cmp_deeply( { one => '1one' },
              { one => str('1one') }, 'str' );
8 cmp_deeply( { one => '1one' },
              { one => bool(1) },     'bool' );
9 cmp_deeply( { one => '1one' },
              { one => code( sub { $_[0] } ) }, 'code' );
```

Testing and reporting on complex data

Testing Utilities

"array_each.t"

```
1 use warnings;
2 use strict;
3 use Test::More qw(no_plan);
4 use Test::Deep;
5
6 cmp_deeply( [ 1, 2, 3 ], array_each( bool(1) ) );
```

Exercise Three

- Taking the percent of ownership example that we've been using throughout the class, create method that prints out a scalar report of units ordered descending by their percent of ownership. Verify the report method works using Test::Differences.

- Do some "White Box" testing on your PercentOwnership class by using Test::Deep to inspect the object internals.

Setting up your tests with mock objects

Testing Utilities

- Many times you'll find yourself needing to instantiate many objects in order to actually run proper tests.
- The objects in your system might be difficult to create, have side effects, or only be partially needed for your tests.
- Sometimes it is possible to create a "mock" object as a stand-in for the real object you need. We can use Test::MockObject for this.
- Other times, it is necessary to overrider specific methods in an object so that your system doesn't suffer from testing side effects. We can use Test::MockObject::Extends for this.

Setting up your tests with mock objects

Testing Utilities

"mock.t"

```
 3 use Test::More qw(no_plan);
 4 use Test::MockObject;
 5
 6 my $mock = Test::MockObject->new();
 7 isa_ok( $mock, 'Test::MockObject' );
 8
 9 $mock->set_isa( 'DBI', 'DBIx::Class' );
10 isa_ok( $mock, 'DBIx::Class' );
11 isa_ok( $mock, 'DBI' );
```

Testing and reporting on complex data

Testing Utilities

```
--(0)> prove mock.t
mock....ok
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.03 cusr +  0.01 csys =  0.04 CPU)
```

Setting up your tests with mock objects

Testing Utilities

"mock.t"

```
13 $mock->set_true('this_is_true');
14 $mock->set_false(qw(f false untrue));
15 ok( $mock->this_is_true );
16 ok( !$mock->f );
17 ok( !$mock->false );
18 ok( !$mock->untrue );
```

Setting up your tests with mock objects

Testing Utilities

"mock.t"

```
20 $mock->set_always( 'rank', 'General' );
21 is( $mock->rank, 'General' );
22
23 $mock->mock( 'uc', sub { uc $_[1] } );
24 is( $mock->uc('small'), 'SMALL' );
25
26 my $var = 'A';
27 $mock->set_bound( 'next_letter', \$var );
28 is( $mock->next_letter, 'A' );
29 $var = 'B';
30 is( $mock->next_letter, 'B' );
```

Setting up your tests with mock objects

Testing Utilities

"mock.t"

```
32 $mock->set_list(qw(count one two three));
33 is_deeply( [ $mock->count ], [qw(one two three)] );
34
35 $mock->set_series(qw(next one two three));
36 is( $mock->next, 'one' );
37 is( $mock->next, 'two' );
38 is( $mock->next, 'three' );
39
40 $mock->set_series( 'fetchrow_array'
     , [ 1, 'Josh' ], [ 2, 'Heather' ] );
41 is_deeply( $mock->fetchrow_array, [ 1, 'Josh' ] );
42 is_deeply( $mock->fetchrow_array, [ 2, 'Heather' ] );
```

Setting up your tests with mock objects

Testing Utilities

"mock.t"

```
44 $mock->set_true('t');
45 ok( $mock->t );
46 $mock->remove('t');
47 ok( !$mock->t );
```

Setting up your tests with mock objects

Testing Utilities

"mock.t"

```
49 $mock->fake_module('CGI');
50 $mock->fake_new('CGI');
51 my $mocked_cgi = CGI->new;
52 isa_ok( $mocked_cgi, 'Test::MockObject' );
```

Setting up your tests with mock objects

Testing Utilities

"extends.t"

```
 3 use Test::More qw(no_plan);
 4 use Test::MockObject::Extends;
 5
 6 use DateTime;
 7
 8 my $dt = DateTime->new(
                year => 2007, month => 12, day => 14 );
 9
10 $dt = Test::MockObject::Extends->new($dt);
11 $dt->set_always( year => 2006 );
12
13 is( $dt->year,  2006 );
14 is( $dt->month, 12 );
```

Exercise Four

- Mock the scenario of getting temperatures to convert out of a database.

- Extend so that no matter what time a file was really modified, it shows a time in the future.

Determining Test Coverage

Testing Utilities

- Often Test-Driven Development leads to only “Happy Path” tests, with edge cases missed or ignored.
- Perl proves an excellent coverage toolkit.
- Test coverage shows you what pieces of your code were exercised by tests and what pieces were missed.
- Coverage is shown deeper than just the statement level, with branch, conditional, statement, and POD coverage illustrated.
- Coverage can not only help you write more and better tests, but also help you find and remove unused code.

Determining Test Coverage

Testing Utilities

```
--(0)> perl -MDevel::Cover -Ilib t/TestPercentOwnership.pm
Devel::Cover 0.64: Collecting coverage data for branch, condition, pod, statement, subroutine and time.
Selecting packages matching:
Ignoring packages matching:
    /Devel/Cover[./]
Ignoring packages in:
    .
    /opt/local/lib/perl5/5.8.8
    /opt/local/lib/perl5/5.8.8/darwin-2level
    /opt/local/lib/perl5/site_perl
    /opt/local/lib/perl5/site_perl/5.8.8
    /opt/local/lib/perl5/site_perl/5.8.8/darwin-2level
    /opt/local/lib/perl5/vendor_perl
    /opt/local/lib/perl5/vendor_perl/5.8.8
    /opt/local/lib/perl5/vendor_perl/5.8.8/darwin-2level
ok 1 - use PercentOwnership;
...
ok 20 - two equal sized units
1..20
Devel::Cover: Writing coverage database to /Users/joshua/Documents/clients/stonehenge/Test Driven Development
Course/src/oo_testing/11/cover_db/runs/1208141988.5526.47211
---------------------------- ------ ------ ------ ------ ------ ------ ------
File                           stmt   bran   cond    sub    pod   time  total
---------------------------- ------ ------ ------ ------ ------ ------ ------
lib/PercentOwnership.pm       100.0  100.0    n/a  100.0    0.0    0.6   90.6
t/TestPercentOwnership.pm     100.0   50.0    n/a  100.0    n/a   99.4   99.0
Total                         100.0   75.0    n/a  100.0    0.0  100.0   97.0
---------------------------- ------ ------ ------ ------ ------ ------ ------
```

Determining Test Coverage

Testing Utilities

- Types of Coverage
  - Statement - which statements were executed?
  - Branch - were all cases that could be branched to achieved?
  - Conditional - were all logical combinations expressed?
  - Subroutine - were all subroutines executed?
  - POD - was there POD for all public subroutines?

Determining Test Coverage

Testing Utilities

Statement Coverage

- Which statements in the program were executed?
- A statement isn’t necessarily a single line of code.
- A single line of code isn’t limited to one statement.
- Statement coverage is one of the easiest metrics to achieve 100% coverage on.
- Some edge cases can be very difficult to create, especially those related to catastrophic system failure.

Determining Test Coverage

Testing Utilities

Branch Coverage

- Every logical branch that program execution can take is exercised.
- Branches include invisible branches, like missing ‘else’ cases on an ‘if’ statement.
- 100% branch coverage implies 100% statement coverage.

Determining Test Coverage

Conditional Coverage

- In each conditional, was every combination of boolean operators tested?

- Full conditional coverage requires more tests than branch coverage because conditional coverage ensures that every combination of boolean operators is used for each conditional while branch coverage just ensures that every possible piece of code is executed at least once around a branch.

- 100% conditional coverage implies 100% branch coverage.

Determining Test Coverage

Testing Utilities

Subroutine Coverage

- A higher-level of coverage that summarizes at the subroutine level.
- If statement coverage is at 100%, subroutine coverage will be 100%.
- Subroutine-level coverage helps you quickly find unused (or just untested) subroutines.

Determining Test Coverage

Testing Utilities

"extends.t"

POD Coverage

- Not really checking to see if any real code is covered with tests.
- Uses Pod::Coverage to make sure that your documentation is comprehensive.

Determining Test Coverage

Testing Utilities

```
---------------------------- ------ ------ ------ ------ ------ ------ ------
File                           stmt   bran   cond    sub    pod   time  total
---------------------------- ------ ------ ------ ------ ------ ------ ------
lib/PercentOwnership.pm       100.0  100.0    n/a  100.0    0.0    0.6   90.6
t/TestPercentOwnership.pm     100.0   50.0    n/a  100.0    n/a   99.4   99.0
Total                         100.0   75.0    n/a  100.0    0.0  100.0   97.0
---------------------------- ------ ------ ------ ------ ------ ------ ------
```

Determining Test Coverage

Testing Utilities

```
--(0)> cover
Reading database from /Users/joshua/Documents/clients/stonehenge/Test Driven
Development Course/src/oo_testing/11/cover_db


---------------------------- ------ ------ ------ ------ ------ ------ ------
File                           stmt   bran   cond    sub    pod   time  total
---------------------------- ------ ------ ------ ------ ------ ------ ------
lib/PercentOwnership.pm       100.0  100.0    n/a  100.0    0.0    0.9   90.6
t/TestPercentOwnership.pm     100.0   50.0    n/a  100.0    n/a   99.1   99.0
Total                         100.0   75.0    n/a  100.0    0.0  100.0   97.0
---------------------------- ------ ------ ------ ------ ------ ------ ------


Writing HTML output to /Users/joshua/Documents/clients/stonehenge/Test Driven
Development Course/src/oo_testing/11/cover_db/coverage.html ...
done.
```

Coverage Summary

Database: /Users/joshua/Documents/clients/stonehenge/Test Driven De

file	stmt	bran	cond	sub	pod	time	total
lib/PercentOwnership.pm	100.0	100.0	n/a	100.0	0.0	0.9	90.6
t/TestPercentOwnership.pm	100.0	50.0	n/a	100.0	n/a	99.1	99.0
Total	100.0	75.0	n/a	100.0	0.0	100.0	97.0

Determining Test Coverage

Testing Utilities

File Coverage

File:	t/TestPercentOwnership.pm
Coverage:	99.0%

line	stmt	bran	cond	sub	pod	time	code
1							`package TestPercentOwnership;`
2							
3	1 1 1			1		55 6 20	`use warnings;`
4	1 1 1			1		11 3 8	`use strict;`
5	1 1 1			1		8 3 10	`use base qw(Test::Class);`
6	1 1 1			1		3183 3 39	`use Test::More;`
7	1 1 1			1		3437 11 39	`use List::MoreUtils qw(pairwise);`
8							
9	1					10	`our ( $a, $b );`
10							
11	1	50				96	`__PACKAGE__->runtests unless caller;`
12							

Branch Coverage

File:	t/TestPercentOwnership.pm
Coverage:	50.0%

line	%	coverage		branch
11	50	T	F	`unless caller`

Determining Test Coverage

Testing Utilities

7							sub new {
8	5			5	0	19	my ($class) = @_;
9	5					42	my $self = bless {}, $class;
10	5					27	return $self;
11							}
12							
13							sub add_unit {
14	9			9	0	116	my ($self, %unit_info) = @_;
15							
16	9	100				65	$unit_info{floor} = 0 if($unit_info
17							
18	9					167	$self->{unit_info}->{ $unit_info{uni
19							}
20							
21							sub percent_ownership {
22	8			8	0	61	my ($self, %args) = @_;
23							

Determining Test Coverage

Testing Utilities

"lib/PercentOwnership.pm"

```
39
40 __END__
41
42 =head1 METHODS
43
44 =head2 new
45
46 Create a new PercentOwnership object.  No parameters required!
47
48 =head2 add_unit
49
50 Tell the PercentOwnership object about a unit in the building.  Provide three
51 named arguments: floor, unit_number, and square_footage.
52
53 =head2 percent_owership
54
55 Request the percent ownership for a given unit.  A single named parameter,
56 unit_number, is required.
```

Determining Test Coverage

Testing Utilities

```
--(0)> perl -MDevel::Cover -Ilib t/TestPercentOwnership.pm
...
---------------------------- ------ ------ ------ ------ ------ ------ ------
File                           stmt   bran   cond    sub    pod   time  total
---------------------------- ------ ------ ------ ------ ------ ------ ------
lib/PercentOwnership.pm       100.0  100.0    n/a  100.0  100.0    2.2  100.0
t/TestPercentOwnership.pm     100.0   50.0    n/a  100.0    n/a   97.8   99.0
Total                         100.0   75.0    n/a  100.0  100.0  100.0   99.2
---------------------------- ------ ------ ------ ------ ------ ------ ------
```

Determining Test Coverage

Testing Utilities

```
--(0)> cover -delete
Deleting database /Users/joshua/Documents/clients/stonehenge/Test Driven Development
Course/src/oo_testing/11/cover_db
```

Code That Accesses Databases

- You want to code to run fast, so directly accessing a database (especially over the network) can be a pain.
- It is not always practical to have a personal copy of the database for your tests.

Code That Accesses Databases

- Some options
  - Run the tests in a virtual machine
  - Mock the database interaction layer
  - Use a lightweight database such as SQLite

Mocking the Database Access Layer

Case Studies

"t/test_with_dbi_mock.t"

```
1 use warnings;
2 use strict;
3 use Test::More qw(no_plan);
4
5 BEGIN { use_ok( 'PrimeNumberIterator' ) };
```

Mocking the Database Access Layer

Case Studies

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
t/test_with_dbi_mock....
#   Failed test 'use PrimeNumberIterator;'
#   at t/test_with_dbi_mock.t line 5.
#     Tried to use 'PrimeNumberIterator'.
#     Error:  Can't locate PrimeNumberIterator.pm in @INC (@INC
contains:  ....) at (eval 3) line 2.
# BEGIN failed--compilation aborted at t/test_with_dbi_mock.t line 5.
# Looks like you failed 1 test of 1.
t/test_with_dbi_mock....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test             Stat Wstat Total Fail  List of Failed
-------------------------------------------------------------------------------
t/test_with_dbi_mock.t    1   256     1    1  1
Failed 1/1 test scripts. 1/1 subtests failed.
Files=1, Tests=1,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
Failed 1/1 test programs. 1/1 subtests failed.
```

Mocking the Database Access Layer

Case Studies

"lib/PrimeNumberIterator.pm"

```
1 package PrimeNumberIterator;
2
3 use warnings;
4 use strict;
5
6 1;
```

Mocking the Database Access Layer

Case Studies

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
t/test_with_dbi_mock....ok
All tests successful.
Files=1, Tests=1,  1 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

Mocking the Database Access Layer

Case Studies

"t/test_with_dbi_mock.t"

```
1 use warnings;
2 use strict;
3 use Test::More qw(no_plan);
4
5 BEGIN { use_ok( 'PrimeNumberIterator' ) };
6
7 can_ok( 'PrimeNumberIterator', qw(new) );
```

Mocking the Database Access Layer

Case Studies

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
t/test_with_dbi_mock....
#   Failed test 'PrimeNumberIterator->can('new')'
#   at t/test_with_dbi_mock.t line 7.
#     PrimeNumberIterator->can('new') failed
# Looks like you failed 1 test of 2.
t/test_with_dbi_mock....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 2
        Failed 1/2 tests, 50.00% okay
Failed Test             Stat Wstat Total Fail  List of Failed
-------------------------------------------------------------------------------
t/test_with_dbi_mock.t    1   256     2    1  2
Failed 1/1 test scripts. 1/2 subtests failed.
Files=1, Tests=2,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
Failed 1/1 test programs. 1/2 subtests failed.
```

Mocking the Database Access Layer

Case Studies

"lib/PrimeNumberIterator.pm"

```
1 package PrimeNumberIterator;
2
3 use warnings;
4 use strict;
5
6 sub new {}
7
8 1;
```

Mocking the Database Access Layer

Case Studies

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
t/test_with_dbi_mock....ok
All tests successful.
Files=1, Tests=2,  1 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

Mocking the Database Access Layer

Case Studies

"t/test_with_dbi_mock.t"

```
 5 BEGIN { use_ok( 'PrimeNumberIterator' ) };
 6
 7 can_ok( 'PrimeNumberIterator', qw(new) );
 8
 9 my $p = PrimeNumberIterator->new();
10 isa_ok( $p, 'PrimeNumberIterator' );
```

Mocking the Database Access Layer

Case Studies

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
t/test_with_dbi_mock....
t/test_with_dbi_mock....ok 1/0#   Failed test 'The object isa
PrimeNumberIterator'
#   at t/test_with_dbi_mock.t line 10.
t/test_with_dbi_mock....NOK 3/0#     The object isn't defined
# Looks like you failed 1 test of 3.
t/test_with_dbi_mock....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 3
        Failed 1/3 tests, 66.67% okay
Failed Test             Stat Wstat Total Fail  List of Failed
-------------------------------------------------------------------------------
t/test_with_dbi_mock.t    1   256     3    1  3
Failed 1/1 test scripts. 1/3 subtests failed.
Files=1, Tests=3,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
Failed 1/1 test programs. 1/3 subtests failed.
```

Mocking the Database Access Layer

Case Studies

"lib/PrimeNumberIterator.pm"

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
  1 package PrimeNumberIterator;
  2
  3 use warnings;
  4 use strict;
  5
  6 sub new {
  7     my $class = shift;
  8     my $self = bless {}, $class;
  9     return $self;
 10 }
```

Mocking the Database Access Layer

Case Studies

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
t/test_with_dbi_mock....ok
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

Mocking the Database Access Layer

Case Studies

"t/test_with_dbi_mock.t"

```
 9 my $p = PrimeNumberIterator->new();
10 isa_ok( $p, 'PrimeNumberIterator' );
11
12 is( $p->next, 2, '2 is the first prime' );
```

Mocking the Database Access Layer

Case Studies

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
t/test_with_dbi_mock....Can't locate object method "next" via package
"PrimeNumberIterator" at t/test_with_dbi_mock.t line 12.
t/test_with_dbi_mock....ok 1/0# Looks like your test died just after 3.
t/test_with_dbi_mock....dubious
        Test returned status 255 (wstat 65280, 0xff00)
        after all the subtests completed successfully
Failed Test             Stat Wstat Total Fail  List of Failed
-------------------------------------------------------------------------------
t/test_with_dbi_mock.t  255 65280     3    0  ??
Failed 1/1 test scripts. 0/3 subtests failed.
Files=1, Tests=3,  1 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
Failed 1/1 test programs. 0/3 subtests failed.
```

Mocking the Database Access Layer

Case Studies

"lib/PrimeNumberIterator.pm"

```
 6 sub new {
 7     my $class = shift;
 8     my $self = bless {}, $class;
 9     return $self;
10 }
11
12 sub next { 2 }
```

Mocking the Database Access Layer

Case Studies

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
t/test_with_dbi_mock....ok
All tests successful.
Files=1, Tests=4,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
```

Mocking the Database Access Layer

Case Studies

"t/test_with_dbi_mock.t"

```
 9 my $p = PrimeNumberIterator->new();
10 isa_ok( $p, 'PrimeNumberIterator' );
11
12 is( $p->next, 2, '2 is the first prime' );
13 is( $p->next, 3, '3 is the second prime' );
```

Mocking the Database Access Layer

Case Studies

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
t/test_with_dbi_mock....ok 1/0
t/test_with_dbi_mock....NOK 5/0#   Failed test '3 is the second prime'
#   at t/test_with_dbi_mock.t line 13.
#          got: '2'
#     expected: '3'
# Looks like you failed 1 test of 5.
t/test_with_dbi_mock....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 5
        Failed 1/5 tests, 80.00% okay
Failed Test             Stat Wstat Total Fail  List of Failed
-------------------------------------------------------------------------------
t/test_with_dbi_mock.t    1   256     5    1  5
Failed 1/1 test scripts. 1/5 subtests failed.
Files=1, Tests=5,  0 wallclock secs ( 0.02 cusr +  0.01 csys =  0.03 CPU)
Failed 1/1 test programs. 1/5 subtests failed.
```

Mocking the Database Access Layer

Case Studies

"t/test_with_dbi_mock.t"

```
 9 my $p = PrimeNumberIterator->new(
       dsn => 'dbi:Oracle:primes' );
10 isa_ok( $p, 'PrimeNumberIterator' );
11
12 is( $p->next, 2, '2 is the first prime' );
13 is( $p->next, 3, '3 is the second prime' );
```

Mocking the Database Access Layer

Case Studies

"t/test_with_dbi_mock.t"

```
 4 use Test::MockObject;
 5
 6 BEGIN { use_ok( 'PrimeNumberIterator' ) };
 7
 8 can_ok( 'PrimeNumberIterator', qw(new) );
 9
10 my $mock_dbi = Test::MockObject->new();
11 my $mock_sth = Test::MockObject->new();
12
13 $mock_dbi->fake_module( 'DBI', connect => sub { $mock_dbi } );
14 $mock_dbi->set_always( 'prepare', $mock_sth );
15 $mock_dbi->set_true( 'disconnect' );
16
17 $mock_sth->set_true( 'execute', 'finish' );
18 $mock_sth->set_series( 'fetchrow_array', qw(2 3) );
19
20 my $p = PrimeNumberIterator->new( dsn => 'dbi:Oracle:primes' );
21 isa_ok( $p, 'PrimeNumberIterator' );
```

Mocking the Database Access Layer

Case Studies

"lib/PrimeNumberIterator.pm"

```
1 package PrimeNumberIterator;
2
3 use warnings;
4 use strict;
5 use DBI;
```

Mocking the Database Access Layer

Case Studies

"lib/PrimeNumberIterator.pm"

```
 7 sub new {
 8     my ($class, %args) = @_;
 9     my $self = bless {%args}, $class;
10
11     $self->{dbh} = DBI->connect($args{dsn});
12     $self->{sth} = $self->{dbh}->prepare(q~
13         select number
14         from primes
15         order by number
16     ~);
17     $self->{sth}->execute();
18
19     return $self;
20 }
```

Mocking the Database Access Layer

Case Studies

"lib/PrimeNumberIterator.pm"

```
22 sub next {
23     my ($self) = @_;
24     return $self->{sth}->fetchrow_array();
25 }
```

Mocking the Database Access Layer

Case Studies

```
--(0)> prove -Ilib t/test_with_dbi_mock.t
t/test_with_dbi_mock....ok
All tests successful.
Files=1, Tests=5,  1 wallclock secs ( 0.04 cusr +  0.02 csys =  0.06 CPU)
```

Mocking the Database Access Layer

Case Studies

"t/test_with_dbi_mock.t"

```
17 my @primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
                 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
                 101, 103, 107, 109, 113, 127, 131, 137, 139);
18
19 $mock_sth->set_true( 'execute', 'finish' );
20 $mock_sth->set_series( 'fetchrow_array', @primes );
21
22 my $p = PrimeNumberIterator->new( dsn => 'dbi:Oracle:primes' );
23 isa_ok( $p, 'PrimeNumberIterator' );
24
25 for my $prime ( @primes ) {
26     is( $p->next, $prime, "got $prime" );
27 }
```

Mocking the Database Access Layer

Case Studies

"t/test_with_dbi_mock.t"

```
 3 use Test::More qw(no_plan);
 4 use DBI;
 5
 6 BEGIN { use_ok( 'PrimeNumberIterator' ) };
 7
 8 can_ok( 'PrimeNumberIterator', qw(new) );
 9
10 unlink 'test.db';
11
12 my $dsn = 'dbi:SQLite:test.db';
13 my $dbh = DBI->connect($dsn);
14 $dbh->do( 'create table primes (number)' );
15
16 my @primes = (2, 3, 5, 7, 11);
17
18 $dbh->do("insert into primes (number)  values ( $_ )")
19   for (@primes);
20
21 $dbh->disconnect;
```

Mocking the Database Access Layer

Case Studies

"t/test_with_dbi_mock.t"

```
23 my $p = PrimeNumberIterator->new( dsn => $dsn );
24 isa_ok( $p, 'PrimeNumberIterator' );
25
26 for my $prime ( @primes ) {
27     is( $p->next, $prime, "got $prime" );
28 }
```

Mocking the Database Access Layer

Case Studies

"t/test_with_sqlite.t"

```
--(0)> prove -Ilib t/test_with_sqlite.t
t/test_with_sqlite....ok
All tests successful.
Files=1, Tests=8,  1 wallclock secs ( 0.04 cusr +  0.02 csys =  0.06 CPU)
```