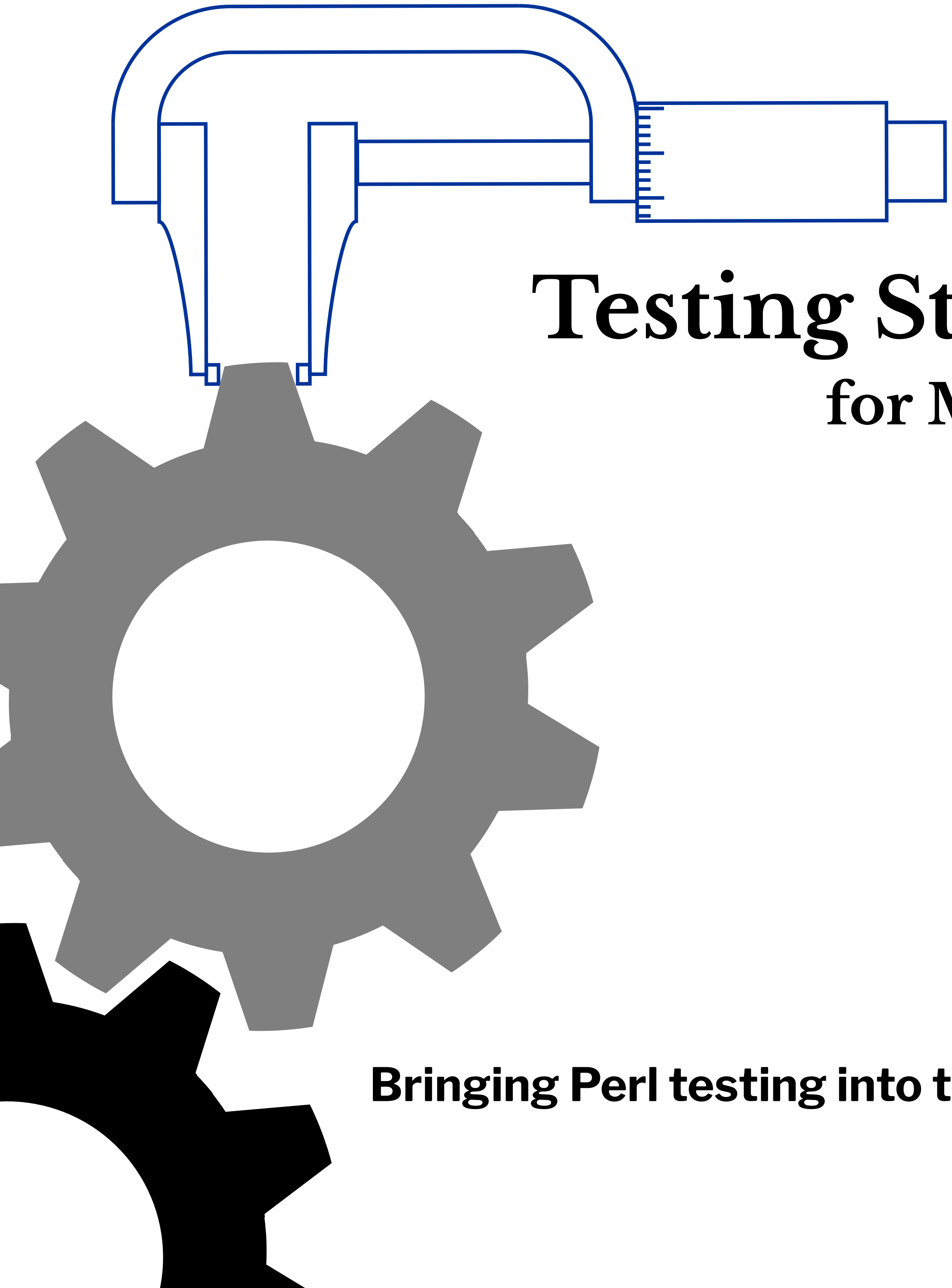
J. Timothy King

the Perl SHOP



Testing Strategies for Modern Perl

Bringing Perl testing into the 21st century

Testing Strategies for Modern Perl

Bringing Perl testing into the 21st century

Testing Strategies for Modern Perl: Bringing Perl testing into the 21st century

Author: J. Timothy King
Reviewers: Bill Ricker, John Tsangaris, Tom Metro

Published by The Perl Shop, LLC.

See more at: http://www.ThePerlShop.com/

Version 0.03α2, 2018-07-09

Table of Contents

chapter 1

Introduction

Modern Perl represents a set of idioms and best practices for writing reliable, maintainable Perl code. These include using newer Perl features like `say` and lexical `sub` to express logic cleanly, use `strict` and `warnings` to help assure correctness, and `Moose` (or a similar object system) for object-oriented code. It also includes adopting software development best practices, such as test-first; organizing code in highly cohesive, loosely coupled modules, with well documented APIs; adhering to a readable coding style; and refactoring frequently to keep it readable, maintainable, and self-consistent.

Part of developing Modern Perl is adhering to a testing regimen that incorporates best testing practices. Unfortunately, numerous Perl projects adopt best practices in their application code—for example, by using modular architecture, well designed frameworks, and documented APIs—but then they skimp on testing. They treat testing as an afterthought, and adopt an ad hoc testing philosophy. This results in incomplete tests, tests that don't run, and barely-maintainable test scripts that read like procedural code reminiscent of 1990's-era Perl CGI hacks.

Proper automated tests are essential to a professional software process.

- They help prove that the code actually does what we think it does.
- They catch bugs early, when they're cheap to fix.
- They demonstrate concrete progress, even if there's no user interface yet.
- They help developers code, by forcing developers to plan before they code and reducing coder's block, and can make the process of coding more enjoyable.
- They facilitate refactoring and Agile architectural improvements.
- They serve as documentation for what the code is supposed to do and how to use its APIs.

In *Testing Strategies for Modern Perl*, we will present a testing strategy that puts Perl testing on the same footing as other Modern Perl frameworks.

Modernizing Perl testing

Perl tests are usually implemented in **`.t`** files, checked into source-code control, typically in the **`t/`** directory under the project root. Each **`.t`** file is an independent executable test script, usually using a testing library such as `Test::More`, which is built on `Test::Builder` and generates TAP (Test Anything Protocol) output. The **`prove`** command sets up the test environment, finds the **`.t`** files, executes each one, and then collates the results.

This is all standard testing convention in the Perl ecosystem. You can find good introductions to Perl testing in Chapter 9 of *Modern Perl*, by chromatic, and in *Perl Testing: A Developer's Notebook*, by Ian Langworth and chromatic.

We extend these conventions in order to:

- distinguish between fast-running unit tests and slower-running subsystem tests
- quickly associate a test file with the code or feature that it tests
- reduce false "action at a distance" failures
- more easily maintain tests
- more easily run tests during development and facilitate Continuous Integration
- help assure that tests are kept up to date and that most code is tested
- encourage writing tests that help serve as documentation for how to use the code under test

Here's the kicker: even though this book is called "Testing Strategies for Modern Perl," these practices are equally suited to testing legacy Perl. In fact, in a later chapter we will specifically address testing and modifying legacy code, demonstrating how this makes it possible to modernize it.

Many of the techniques and concerns this book covers are also discussed in *xUnit Testing Patterns: Refactoring Test Code*, by Gerard Meszaros, which I heartily recommend.

Best testing practices

In our approach to testing there are a number of practices we try to follow consistently, which we will elaborate on throughout this book.

- Write tests first, at all levels from system through unit.
- Write test methods to be readable, with setup, testing, and assertions in separate code paragraphs.
- Use literal test inputs and expected outputs, rather than re-implementing the logic under test.
- Use table-driven tests for batches of related inputs and expected outputs.
- Group related test methods in the same module.
- Store ancillary modules and other files close to the corresponding test file.
- Use `Devel::Cover` as a quality check when testing legacy code.

Organization of this book

This book is organized into three parts. In Part One, we'll cover modern testing techniques in Perl including:

- Test-driven development with `Test::Class` and `Test::Most`;
- Table-driven testing, and other test-reuse strategies;
- Mock objects and mocking modules, the process of creating a simplified simulation of a dependency, such as a mock method that substitutes for reading data from a file but instead always returns a constant record of data;
- Subsystem and integration testing, testing modules working together and testing the system as a whole;
- Testing user-interfaces, both command-line and web UI;
- Testing with databases and application configuration files;
- Refactoring strategies, both for production code and for refactoring tests.

Part Two will cover tooling:

- Testing frameworks, specifically our preferred `Test::Class`;
- A survey of Perl mocking libraries from CPAN;
- Using Continuous Integration (CI) tools with your Perl projects to automatically run test suites and report when a code change breaks a test.

In Part Three, we will explore further applications of modern testing techniques in Perl: testing legacy Perl, modernizing it as described in Michael Feathers' *Working Effectively with Legacy Code*; and testing Perl 6.

We're excited that you've embarked on this journey with us to discovering the benefits and joys of Perl testing.

Part One

Techniques

chapter 2

Test-driven Development

Kent Beck “rediscovered” test-driven development in 1999. In fact, test-first is actually a very old and well-established programming practice.

He tells the oft-repeated story:

> The original description of TDD was in an ancient book about programming. It said you take the input tape, manually type in the output tape you expect, then program until the actual output tape matches the expected output… When describing TDD to older programmers, I often hear, “Of course. How else could you program?”

I don’t know what book he was referring to. However, a few minutes on Google will uncover allusions to this programming practice from the 1970’s, 1960’s, and 1950’s.

The process is itself very straightforward:

1. **Write a test that fails.** That is, write the test, run it, and *see* it fail. You can’t write any production code until you’ve first written a failing test. You also shouldn’t write any more of a test than you need to fail. And the module or method not existing is a form of test failure.

2. **Write production code that makes the test pass.** You can’t write any more production code than is needed to make the currently failing test pass. This is often just a single line of code. If you accidentally wrote more, then comment out or delete the excess, until *after* you’ve written tests for it, too.

3. **Refactor production code.** That is, refactor it, rerun all the tests, and confirm that they all still pass. The biggest obstacle developers cite that stops them from beautifying their code is that they’re afraid they’ll introduce new bugs. But since we have a whole test suite testing every line of code, we can safely refactor it.

We follow this same process when modifying existing code. We change the test first to reflect the new functionality. Then we change just enough code to make the tests pass. Then clean up the code if necessary.

Benefits of test-first

Aside from making it safe to refactor the code under test, tests provide many other benefits.

- By starting with writing a test, you are forced to think about the exact inputs your code consumes, the expected outputs it produces, and importantly the dependencies and side effects it has. This steers you towards a design that has small, coherent, and loosely coupled methods. If you find your code is hard to test, that’s probably because the design is poor.

- Seeing a test go from red to green often produces a rush of happy brain chemicals. This can be enjoyable and motivating for the developer, and might help improve performance on a project.

- A test proves that your code actually works. Bugs are exponentially more expensive to fix the later you find them. So what if you could find them even before you write the code?

- They are an executable specification for the code under test. And an automated regression suite.

- They also serve as documentation for the code's APIs. This makes the code under test easier to maintain.

- We at the Perl Shop have shown off written-and-passing tests to a client to demonstrate progress, even in the early stages of a project before there's much of a user interface.

What makes a good test

Since we're going to be writing a lot of tests, let's quickly review what makes a good test. Good tests should be easy to write, easy to run, and easy to read and maintain.

- **Easy to write:** A test should be simple. Each test should exercise a single piece of functionality. The developer's focus should be on the code under test, not on the test logic itself.

- **Easy to run**: A test should test its part of the system in isolation. A unit test, in particular, should test one and only one module without involving any other part of the system under test. Each test should run independently and should not depend on other tests or on outside system state. And it should give the same result every time it's run.

- **Easy to read and maintain:** It should be easy to see how a test works by simply looking at it. The test is being written for humans to read, not simply to produce output for `prove`. It should be clear how the test sets up the conditions for the test, how it calls the code under test, and how it verifies the test results. We also want each piece of code to affect a single test, or as few tests as possible, so that a desired change to the code will not require modifying a large number of tests.

We're going to be demonstrating all of these as we work our way through an example project.

The project

We're building a Tic-Tac-Toe application. This is a simple, easy-to-understand problem with fairly clear requirements, but complex enough to demonstrate real-world architecture and development techniques. Our version of the game uses `Catalyst`, and is designed to support a variety of user interfaces and data-storage formats.

The project uses `Moose`, but I'll try to keep the Moose deep-magic to a minimum, and explain any parts of it that I do use. The same testing techniques apply regardless of which object system a particular project uses.

Our current task is to create a business-logic class for the main gameplay. This class will manage the game board, has methods to place X's and O's on the game board, and can tell whose turn it is and whether a player has won the game.

All the project files are available in a companion **`TestingGuideFiles`** repository on GitHub.

Intro to testing with `Test::Class`

Before we create the first test file, a brief word about test frameworks.

We use `Test::Class` for all our tests. It's built using `Test::Builder`, and works with other `Test::Builder` modules (like `Test::More`). In fact, the code inside each test method is `Test::More` code, and compatible with all `Test::More` features (including `SKIP:{}` blocks and `subtest` blocks, if you want). And it has a number of appealing features:

- It's object-oriented, like xUnit. "xUnit" is the collective name for several unit testing frameworks—including Java's jUnit—that derive their structure and functionality from Smalltalk's SUnit. `Test::Class` uses a similar style to xUnit, though it isn't identical.
- It splits up testing into small, bite-sized pieces, which are easier to understand, easier to maintain, and easier to modify than pure `Test::More` scripts common in the Perl world.
- It supports special methods to setup the test environment and cleanup after running tests.
- It facilitates reusing test code by inheritance as well as by import. This is especially useful for reusing test setup and cleanup code.
- Each test method is run independently and does not depend on tests that have run before.
- You can also run an individual test method from a test module, which is often used during development to speed up the code-test cycle by running just the test method that you're working on.

Some people see `Test::Class` as testing object-oriented code, but it can be used to test pure procedural code as well. We even use it for system-level integration tests. A common practice in the Perl community is to create a hierarchy of test classes that mirrors the hierarchy of modules under test, but this is an anti-pattern (to be avoided). `Test::Class` is simply a framework with which to organize test code, whether object-oriented or procedural, and provides some useful features to make that process smoother.

We'll go into more detail regarding `Test::Class` in Part Two.

The first test: create the class

True to test-first, we begin development with a new test script. We create and run the first test even before generating the code module that we're developing. The test script is based on a **`test.t`** template. We'll go into the template in more detail in a later chapter, but it is included in the companion **`TestingGuideFiles`**, in the `code_templates` directory.

Code for this first step can be found in the **TestingGuideFiles** repository, on the **TicTacToe/task/create_game_class** branch. Normally, we wouldn't split up such small tasks into their own branches, but for the purpose of this tutorial we've placed each step in the process into its own branch, no matter how trivial.

The **test.t** template provides all the `Test::Class` boilerplate, including a space to load the module under test:

```
# load code to be tested
use TicTacToe::BusinessLogic::Game;
```

Note that we haven't actually added any test methods yet. The above line is the last executable statement in the test script (aside from the final `1;` terminator). (See Figure 1, and note the code above highlighted in blue.)

This file we place in a subdirectory corresponding to the module under test: **t/lib/TicTacToe/BusinessLogic/Game/**. The **t/** directory lives in the project root, and all tests are contained thereunder. The **lib/** directory contains all unit tests, organized by the module being tested, in this case **TicTacToe::BusinessLogic::Game**. The test script we call **new.t**, because it is going to be testing the **new()** method.

Before we do anything else, let's run the test, from the project root directory:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/new.t
t/lib/TicTacToe/BusinessLogic/Game/new.t ..
Can't locate TicTacToe/BusinessLogic/Game.pm in @INC (you may need to install the
TicTacToe::BusinessLogic::Game module)
```

The **-l** switch to **prove** indicates that it should look in the **lib/** directory under the current directory for modules. The **-v** switch indicates that it should display output verbosely, including details that would otherwise be hidden.

We've just confirmed that our test module is attempting to load the code under test. At this point, we can create a skeleton module, from the **MooseClass.pm** template. This module contains no functionality whatsoever, just summary POD describing the API that we will be implementing. (See Figure 2.) This file is **lib/TicTacToe/BusinessLogic/Game.pm**.

Now when we run the test:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/new.t
t/lib/TicTacToe/BusinessLogic/Game/new.t .. No subtests run
```

Excellent! That's exactly what we expected. The code under test loads without errors, but no tests were run (because we haven't yet added any test methods). So let's write our first test method:

```
sub test_basic_construction : Test(1) {
    my $test = shift;

    ok( TicTacToe::BusinessLogic::Game->new() );
}
```

As per the **`test.t`** template, this goes inside the `BEGIN { }` block, before the final `1;`. Also note that we're adding POD for the test method at this time, but it's excluded from the above snippet for brevity.

The `:Test(1)` subroutine attribute tells `Test::Class` that this is a test method, and that it contains a single assertion. `Test::Class` will define the test plan automatically based on this information.

This test method does one simple thing. It instantiates a game object, and passes if `new()` returns a true value.

The `new()` method is implemented by Moose, and our class inherits it automatically from `Moose::Object`. So we should not be surprised when we see the following:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/new.t
t/lib/TicTacToe/BusinessLogic/Game/new.t ..
#
# t::lib::TicTacToe::BusinessLogic::Game::new->test_basic_construction
1..1
ok 1 - test basic construction
ok
All tests successful.
```

This is actually a problem. We haven't confirmed that our test actually tests what it claims to test. What we need to do is to cause the code under test to fail, to actually *break* the module under test so that we can confirm that the test script detects the failure. This doesn't need to be a permanent failure. We only need to break the code module long enough to run **`prove`** again.

So temporarily replace the "`make_immutable`" line of **`Game.pm`**, with the following:

```
sub new { return undef }
# __PACKAGE__->meta->make_immutable;
```

We've written a `new()` method that simply fails. It's completely useless, except for testing our test.

Commenting out `make_immutable()` is a Moose thing. The `make_immutable()` method does some cool fancy stuff for performance, and it complains if you define your own `new()` method. There are better ways of handling this in production code, but as this change is temporary, let's just

```
package TicTacToe::BusinessLogic::Game;
use 5.026;
use Moose;
use namespace::autoclean;


=head1 NAME

TicTacToe::BusinessLogic::Game - A class that encompasses the gameplay
of a Tic-Tac-Toe game.

=head1 SYNOPSIS

    use TicTacToe::BusinessLogic::Game;

    my $game = TicTacToe::BusinessLogic::Game->new();

    $game->move('X', 0); # puts an 'X' in slot 0 of the board

    my $board = $game->board;
    @$board; # a list of 9 slots, each ' ', 'X', or 'O'

    my $winner = $game->winner; # 'X', 'O', or undef

=cut


__PACKAGE__->meta->make_immutable;

1;
```

Figure 2: Initial Code Module

comment out the call to `make_immutable()`.

Now when we run the test:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/new.t
t/lib/TicTacToe/BusinessLogic/Game/new.t ..
#
# t::lib::TicTacToe::BusinessLogic::Game::new->test_basic_construction
1..1
not ok 1 - test basic construction

#   Failed test 'test basic construction'
#   at t/lib/TicTacToe/BusinessLogic/Game/new.t line 55.
#   (in t::lib::TicTacToe::BusinessLogic::Game::new->test_basic_construction)
# Looks like you failed 1 test of 1.
Dubious, test returned 1 (wstat 256, 0x100)
Failed 1/1 subtests
```

Seeing this may bring an unexpected rush of accomplishment. We usually associate that rush with green, but in this case, the red error message confirms that our test is working as designed.

All that's left is to undo the breakage in **`Game.pm`**, rerun the test, and see that it shows green. Now we're ready to start adding features.

A note on running the test suite before committing changes

It's good practice to run the entire test suite before committing changes:

```
$ prove -lr t/
t/01app.t .................................. ok
t/02pod.t .................................. skipped: set TEST_POD to enable this test
t/03podcoverage.t .......................... skipped: set TEST_POD to enable this test
t/lib/TicTacToe/BusinessLogic/Game/new.t .... ok
All tests successful.
```

The **`-r`** switch tells **`prove`** to recurse into subdirectories to find **`.t`** files to run.

This is a regression test on the system, to assure that the changes we're making have not broken something somewhere else in the system. You can see above that there are several stock tests that were added by the Catalyst bootstrapping procedure, and our new test is the only other one included.

However, as more tests get added on a large project, running the entire test suite may become unwieldy. This is why we organize the tests in subdirectories. You might want to run only the unit tests, for example, which all reside under **`t/lib/`**:

```
$ prove -lr t/lib/
```

This would exclude broader application or integration tests, which, as we will see later, can take much longer to run if they need to set up test databases or other application configuration.

At the very least, you can run the unit tests just for the module you modified:

```
$ prove -lr t/lib/TicTacToe/BusinessLogic/Game/
```

By the time a project is complex enough that developers are skipping running tests manually, you really should have set up continuous integration (CI), which will run the test suite automatically whenever changes are committed. We'll talk about CI in Part Two.

Incrementally add features: initialize the board

The Tic-Tac-Toe game object needs to keep track of the game board. When we first start a new game, we expect the game board to be blank. So that's the first feature we'll implement. We're going to add a `board` attribute to the `Game` class, and assure that it's initialized properly, to an array of 9 elements, each containing a space.

The following changes are committed in the **`TestingGuideFiles`** repository, on the **`TicTacToe/task/initialize_game_board`** branch.

Again, following the test-first process, we'll create a test script to unit-test the `board` attribute. This script will be called **`board.t`**. Then we'll add a test method to retrieve the board on a new object and assure that it's blank:

```
sub test_board_initialized_blank : Test(1) {
    my $test = shift;

    my $game = TicTacToe::BusinessLogic::Game->new();

    my $board = $game->board;

    cmp_deeply(
        $board,
        [ ' ', ' ', ' ',
          ' ', ' ', ' ',
          ' ', ' ', ' ' ],
        'blank game board',
    );
}
```

This is slightly more complex than the earlier test we wrote. You can see the phases of testing laid out in the code above:

1. setting up the test – instantiating the game object,
2. calling the code under test – retrieving the value of the `board` attribute,
3. verifying that the test passed – checking the board data,
4. tearing down after the test – destroying the game object, done automagically by Perl.

We tested the board data using `cmp_deeply`, from `Test::Deep`. This allows us to test the entire data structure in a single statement.

(Our test scripts use `Test::Most`, which pulls in a number of other testing modules, including `Test::More` and `Test::Deep`. That's how we can use `ok()` and `cmp_deeply()` without specifically importing them.)

Additionally, we wrote out the full expected board state, rather than computing it programmatically. In tests, we generally spell out the inputs we give to and outputs we expect from the code under test, as this makes it clear exactly what behavior we're testing. We reserve algorithms and other logic abstractions for code under test, which may implement a suite of behaviors.

When we run this test, it fails, as expected, because the `board` attribute doesn't exist yet:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/board.t
t/lib/TicTacToe/BusinessLogic/Game/board.t ..
#
# t::lib::TicTacToe::BusinessLogic::Game::board->test_board_initialized_blank
1..1
not ok 1 - test_board_initialized_blank died (Can't locate object method "board" via
package "TicTacToe::BusinessLogic::Game" at t/lib/TicTacToe/BusinessLogic/Game/board.t
line 57.
```

So let's add a `board` attribute, using standard Moose syntax:

```
has board => (
    is => 'rw',
);
```

This is the bare minimum to keep the test from dying. However, it still doesn't make it pass:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/board.t
t/lib/TicTacToe/BusinessLogic/Game/board.t ..
#
# t::lib::TicTacToe::BusinessLogic::Game::board->test_board_initialized_blank
1..1
not ok 1 - blank game board

#   Failed test 'blank game board'
#   at t/lib/TicTacToe/BusinessLogic/Game/board.t line 59.
#   (in t::lib::TicTacToe::BusinessLogic::Game::board->test_board_initialized_blank)
# Compared reftype($data)
#    got : undef
# expect : 'ARRAY'
# Looks like you failed 1 test of 1.
Dubious, test returned 1 (wstat 256, 0x100)
Failed 1/1 subtests
```

Now all that's needed is to initialize the board upon instantiation, by specifying a default in the attribute declaration:

```
has board => (
    is => 'rw',
    default => sub { [ (' ') x 9 ] },
);
```

And now rerunning the test:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/board.t
t/lib/TicTacToe/BusinessLogic/Game/board.t ..
#
# t::lib::TicTacToe::BusinessLogic::Game::board->test_board_initialized_blank
1..1
ok 1 - blank game board
ok
All tests successful.
```

Adding a method: putting a piece on the board

One more example. Let's add a method that allows a user to place their piece, 'X' or 'O', at a given location on the board. We're using the same test-first process, which is hopefully feeling a little more familiar by now.

These changes are in the `TestingGuideFiles` repository on the `TicTacToe/task/move_game_piece` branch.

We create a unit test `move.t`, and a test method to test moving player 'X' to location `0` as the first move of a new game:

```
sub test_move_X_to_0 : Test(1) {
    my $test = shift;

    my $game = TicTacToe::BusinessLogic::Game->new();

    $game->move('X', 0);

    my $board = $game->board;
    cmp_deeply(
        $board,
        [ 'X', ' ', ' ',
          ' ', ' ', ' ',
          ' ', ' ', ' ' ],
        'game board with X at location 0',
    );
}
```

Running this test without any additional development gives the error:

```
not ok 1 - test_move_X_to_0 died (Can't locate object method "move" via package
"TicTacToe::BusinessLogic::Game" at t/lib/TicTacToe/BusinessLogic/Game/move.t line 57.
```

So we add a stub for method `move()`. Now we get a more meaningful error:

```
not ok 1 - game board with X at location 0

#   Failed test 'game board with X at location 0'
#   at t/lib/TicTacToe/BusinessLogic/Game/move.t line 60.
#   (in t::lib::TicTacToe::BusinessLogic::Game::board->test_move_X_to_0)
# Compared $data->[0]
#    got : ' '
# expect : 'X'
```

Implementing the full method body:

```
sub move {
    my $self = shift;
    my ($piece, $location) = @_;

    $self->board->[$location] = $piece;
}
```

You may be noticing that the number of lines of test code is far exceeding the number of lines of production code. This won't always be true, although it is for these introductory features.

A lot of this test code has also been duplicated between tests, so it didn't take as much effort as if it were written each time from scratch. Normally, copy-paste coding is frowned upon, as it indicates that you're duplicating knowledge that should be abstracted into a reusable unit. But with tests, we allow much more duplication, especially when it makes the final test method easier to understand.

Later on, we'll be discussing ways of combining similar tests into a single method using table-driven tests. And we'll be looking at other ways to abstract parts of the test setup and assertions, without compromising test clarity.

For now:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/move.t
t/lib/TicTacToe/BusinessLogic/Game/move.t ..
#
# t::lib::TicTacToe::BusinessLogic::Game::board->test_move_X_to_0
1..1
ok 1 - game board with X at location 0
ok
All tests successful.
```

Testing error cases: catching invalid moves

As you may have guessed, the `move()` method has a giant hole in it. What if someone tries to move a piece to an invalid location? Or a location that's already occupied? Or move a piece that isn't 'X' or 'O'? Or tries to move out of turn? That's what we'll be addressing next.

These changes are in the **TestingGuideFiles** repository on the **TicTacToe/task/prevent_simple_invalid_moves** branch.

In the case of an invalid move, we want the `move()` method to die with a message identifying why the attempted move was illegal. We also want to make sure that the board state does not change because of an attempted illegal move. So with this, we can write a test for an error case:

```
sub test_move_invalid_location_9 : Test(2) {
    my $test = shift;

    my $game = TicTacToe::BusinessLogic::Game->new();

    throws_ok {
        $game->move('X', 9);
    } qr/invalid location: 9/, 'invalid location thrown';

    my $board = $game->board;
    cmp_deeply(
        $board,
        [ ' ', ' ', ' ',
          ' ', ' ', ' ',
          ' ', ' ', ' ' ],
        'game board unchanged',
    );
}
```

This test method has 2 test assertions, as you can see in the `:Test(2)` declaration.

We also use `throws_ok` to verify that the code under test dies with an appropriate message. See Test::Exception for more on `throws_ok`.

And when we run it:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/move.t
t/lib/TicTacToe/BusinessLogic/Game/move.t ..
#
# t::lib::TicTacToe::BusinessLogic::Game::board->test_move_X_to_0
1..3
ok 1 - game board with X at location 0
#
# t::lib::TicTacToe::BusinessLogic::Game::board->test_move_invalid_location_9
not ok 2 - invalid location thrown

#   Failed test 'invalid location thrown'
#   at t/lib/TicTacToe/BusinessLogic/Game/move.t line 85.
#   (in t::lib::TicTacToe::BusinessLogic::Game::board->test_move_invalid_location_9)
# expecting: Regexp ((?^u:invalid location: 9))
# found: normal exit
```

```
not ok 3 - game board unchanged

#   Failed test 'game board unchanged'
#   at t/lib/TicTacToe/BusinessLogic/Game/move.t line 88.
#   (in t::lib::TicTacToe::BusinessLogic::Game::board->test_move_invalid_location_9)
# Compared array length of $data
#    got : array with 10 element(s)
# expect : array with 9 element(s)
# Looks like you failed 2 tests of 3.
Dubious, test returned 2 (wstat 512, 0x200)
Failed 2/3 subtests
```

As you can see, the initial test method still passes, but when we try to move to location 9, not only does the method exit normally, but also it expands the game board to fit the additional location.

So we add a simple guard at the beginning of `move()`:

```
       die "move to invalid location: $location\n"
           unless $location < 9;
```

Now all tests pass:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/move.t
t/lib/TicTacToe/BusinessLogic/Game/move.t ..
#
# t::lib::TicTacToe::BusinessLogic::Game::board->test_move_X_to_0
1..3
ok 1 - game board with X at location 0
#
# t::lib::TicTacToe::BusinessLogic::Game::board->test_move_invalid_location_9
ok 2 - invalid location thrown
ok 3 - game board unchanged
ok
All tests successful.
```

We can do similarly with locations that are less than 0, writing a test method called `test_move_invalid_location_negative_1` that is identical to `test_move_invalid_location_9`, except that it does `$game->move('X', -1)`.

Running **`move.t`** again fails with the expected two test assertions, but adding the following line fixes everything:

```
       die "move to invalid location: $location\n"
           unless $location >= 0;
```

Refactoring to clean up the code

Now that the test is passing, we can refactor the code to be cleaner. We favor generalized code and reusable abstractions in production code, so let’s combine the two `die` statements into one.

```
    die "move to invalid location: $location\n"
        unless $location >= 0 && $location < 9;
```

Rerun the tests to confirm that everything still works, and we’re done! (At least for the moment.)

In the next chapter, we’re going to expand this error checking further, and we’re going to want to stop copying and pasting the same test method umpteen times. As mentioned earlier, we’ll explore a way to reuse the same test code without muddying the clarity of the test methods.

Conclusion

We just developed tested production code, without a user interface of any sort. At each step during writing the code, we saw it work and had opportunity to discover and fix any bugs that cropped up. In the end, we’re left with not only with an automated regression test suite, but also real-life examples of how to use the code we just wrote.

- We always wrote a test and saw it fail, so we know all our tests work, each associated with a unique behavior in the code.
- This includes the default, trivial cases, e.g., instantiating a default object.
- Our test methods were easy to read, clearly showing (a) test setup, (b) invoking the code under test, and (c) verifying the expected result.
- We used literal inputs and outputs in tests, rather than generating them programmatically, but preferred more abstract logic in the production code being tested.
- We also tested error cases, as well as “happy-path” cases.

Now that we have the basics, in the next chapter let’s build on this code, exploring some patterns that will allow us to generate tests more quickly and more succinctly.

chapter 3

Reusing Test Code

Extending the `TicTacToe::BusinessLogic::Game` class

There are a number of basic features still to be implemented in the `TicTacToe::BusinessLogic::Game` class that we began in Chapter 2:

- Disallow moving an invalid piece. (Only `'X'` and `'O'` are permitted.)
- Disallow moving to an already-occupied space.
- Disallow moving out of turn. (`'X'` always goes first, then `'O'`, then `'X'` again, etc.)
- Detect when the game has been won.
- Disallow moving when the game is finished (i.e., when a player has already won).
- Disallow initializing the board to an invalid state. (This one is more complicated.)

As we implement these features, we're going to encounter a number of opportunities to refactor common test code in order to reduce test-code duplication and make our test methods easier to read and maintain.

Factoring out common data

Let's take another look at **`move.t`**. (The following code can be found in the **`TestingGuideFiles`** repository, on the **`TicTacToe/task/detect_invalid_moves`** branch.) We already have multiple blocks that look like this:

```
my $board = $game->board;
cmp_deeply(
    $board,
    [ 'X', ' ', ' ',
      ' ', ' ', ' ',
      ' ', ' ', ' ' ],
    'game board with X at location 0',
);
```

The only differences between the Instances of `cmp_deeply()` in this file are in (1) the expected board configuration and (2) the test description message. Additionally, board configurations can be used multiple times. If we take a minute to think about it, we can already see that as we implement the list of features above, we're going to use each board configuration numerous times. For some tests, we're also going to want to initialize a board to a known configuration, and it would really be handy to be able to just use a named board configuration in all these instances.

So let's factor the inline literals out as named symbols. Near the top of **move.t**—I put test-specific `use` statements in a separate paragraph just before loading the code to be tested:

```
use Readonly;
```

I'm using the `Readonly` module, and not `constant`, because it can mark complex data structures as read-only. Then if some of our code errantly tries to modify the data, our test will die with a clear error. (In fact, while developing this sample code, that actually happened, and I'll point out where a little later.)

Then in a section before the tests themselves:

```
## Game board configurations for testing.

# An empty board (initial state).
Readonly::Array my @BOARD_EMPTY =>
    ( ' ', ' ', ' ',
      ' ', ' ', ' ',
      ' ', ' ', ' ' );

# A board with X in location 0
Readonly::Array my @BOARD_X =>
    ( 'X', ' ', ' ',
      ' ', ' ', ' ',
      ' ', ' ', ' ' );
```

We can then replace the board literals like so:

```
    my $board = $game->board;
    cmp_deeply(
        $board,
        \@BOARD_X,
        'game board with X at location 0',
    );
```

And similarly in each location where a board literal occurs. This is just an application of "Don't repeat yourself," and also makes the code slightly more self-documenting. And as mentioned above, we're going to want to use these constants in other places in other tests as well.

Now we can run the modified tests:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/move.t
t/lib/TicTacToe/BusinessLogic/Game/move.t ..
#
# t::lib::TicTacToe::BusinessLogic::Game::move->test_move_X_to_0
1..5
ok 1 - game board with X at location 0
```

```
#
# t::lib::TicTacToe::BusinessLogic::Game::move->test_move_invalid_location_9
ok 2 - invalid location thrown
ok 3 - game board unchanged
#
# t::lib::TicTacToe::BusinessLogic::Game::move->test_move_invalid_location_negative_1
ok 4 - invalid location thrown
ok 5 - game board unchanged
ok
All tests successful.
Files=1, Tests=5,  0 wallclock secs ( 0.02 usr  0.01 sys +  0.18 cusr  0.02 csys =
0.23 CPU)
Result: PASS
```

Yay! But...

How do we know that we haven't broken any of the tests in our refactoring?

Making a modified test fail

Refactoring, by definition, is changing the structure of code without changing its behavior. But we're also only human, and we sometimes make mistakes. Whenever we refactor code—including test code—we open ourselves up to the possibility that we've accidentally introduced bugs. And with tests, it's conceptually possible to break the test such that it *always passes*.

When we first wrote each test, we saw that it failed. That gave us confidence that the test would complain when our code didn't do what the test wanted. Then we wrote enough code to make the test pass. That gave us confidence that our code actually does what we think it's doing.

But now we're modifying the test, and we're potentially introducing errors, and we're not running through the process again. So let's do a quick check to make sure that our tests are still detecting bugs in the code under test.

In **`Game.pm`**, there's a line to write the piece to a location on the board:

```
#     $self->board->[$location] = $piece;
```

We temporarily comment it out, by putting a # character at the beginning of the line. Don't worry: the unmodified, working module is still contained in the Git repository, and besides, my editor has an "undo" function.

Now when we run the test, we see the following failed test:

```
# t::lib::TicTacToe::BusinessLogic::Game::move->test_move_X_to_0
1..5
not ok 1 - game board with X at location 0

#   Failed test 'game board with X at location 0'
#   at t/lib/TicTacToe/BusinessLogic/Game/move.t line 77.
#   (in t::lib::TicTacToe::BusinessLogic::Game::move->test_move_X_to_0)
# Compared $data->[0]
#    got : ' '
# expect : 'X'
```

We've just tested the `cmp_deeply()` call that we modified above.

There are two other instances of `cmp_deeply()` that I changed but didn't talk about, in the two tests that attempt to move to invalid locations. So I restore **`Game.pm`** to its original state, and then I comment out the following lines:

```
#    die "move to invalid location: $location\n"
#        unless $location >= 0 && $location < 9;
```

Now it won't throw an exception when we try to move to an invalid location, but will happily poke invalid data into the array.

Rerunning the test:

```
not ok 3 - game board unchanged

#   Failed test 'game board unchanged'
#   at t/lib/TicTacToe/BusinessLogic/Game/move.t line 103.
#   (in t::lib::TicTacToe::BusinessLogic::Game::move->test_move_invalid_location_9)
# Compared array length of $data
#    got : array with 10 element(s)
# expect : array with 9 element(s)
```

And similarly for `test_move_invalid_location_negative_1`, for which the last element of the array (the one of index `-1`) errantly contains `'X'`.

The same rule applies if we accidentally write a buggy test in the first place. Suppose that we write a test, see it fail, and then set about writing the code to make the test pass. And suppose that in the process of writing the code, we discover that we had made a mistake in our test. So we happily fix our test to pass with the newly written (correct) code. But how do we know that the test is working as advertised, if we haven't seen it fail? We don't. So we comment out the new code we just wrote, run the test, see it fail, and then un-comment the code, run the test, see it pass.

Custom test assertions

We refactored our tests to use symbolic data, defined once, rather than to use repeated literals. But as Billy Mays said: But wait! There's more!

At this point, we have a number of test assertions, all of which fit the following pattern:

```
    my $board = $game->board;
    cmp_deeply(
        $board,
        $board_configuration,
        $description,
    );
```

...where `$board_configuration` and `$description` are replaced by test-specific values.

We can refactor these into a custom test assertion function:

```
## Private functions and methods

# Assert that a game's board matches the specified board configuration.
sub _cmp_board {
    my ($game, $board_configuration, $description) = @_;

    local $Test::Builder::Level = $Test::Builder::Level + 1;

    my $board = $game->board;
    cmp_deeply(
        $board,
        $board_configuration,
        $description,
    );
}
```

The line highlighted in blue tells the test framework that the stack level has increased. That way, it will report test failures at the line in the test method where `_cmp_board()` is called, not the line at which `cmp_deeply()` is called. That makes it easier to hunt down the code that failed. (This is documented in the `Test::Builder` documentation.)

Then we call this new function elsewhere in the test module, thusly:

```
sub test_move_X_to_0 : Test(1) {
    my $test = shift;

    my $game = TicTacToe::BusinessLogic::Game->new();

    $game->move('X', 0);

    _cmp_board($game, \@BOARD_X, 'game board with X at location 0');
}
```

We comment out lines from `TicTacToe::BusinessLogic::Game::move()` as before, to see the refactored tests fail.

This might not seem like a big deal. We only saved one line per test. But one line of code that's not copied-and-pasted from one method to the next and the next and *the next* etcetera, that's umpteen copies of that code that don't need to be read, understood, or maintained separately. And we will be testing a lot more of these invalid-move cases below.

Even so, it bears noting that the same method can be used—and we've often used it—to factor out complex, multi-step comparisons. Consider, for example, an application that works with custom, line-oriented data, with one data record per line. Each record begins with a serialized record-type, which identifies what data is included in that line. We'll want to write tests of the code that generates the output data, compare it to what we expect, and report any differences in an easy-to-read format. So we split the data by line; pull off the record types and compare them using `Algorithm::Diff`; then report which lines were added, removed, and changed, and for each pair of lines whose record types match, report any differences using `String::Diff`. The code looks something like the listing in Figure 3.

This is an extreme case, but not unheard of. Of particular interest in Figure 3 is the `subtest` block. The inner loop will pass or fail for each line of data, depending on whether that line matches in `$got` and `$expected`. By putting the guts of the function into a subtest, we can report test results for each line, and `linerecords_eq_or_diff()` itself only counts as one test.

Parameterized setup

Sometimes, we need to set up objects for our tests. For example, let's write a test method that moves an `'O'` to the board:

```
sub test_move_O_to_1 : Test(1) {
    my $test = shift;

    my $game = TicTacToe::BusinessLogic::Game->new(
        board => [@BOARD_X],
    );

    $game->move('O', 1);

    _cmp_board($game, \@BOARD_XO, 'game board with O added at location 1');
}
```

X always moves first; therefore, before moving `'O'` to location 1, we initialize the board with an `'X'` at location 0, using Moose's built-in attribute initialization. This is called the "test fixture," the `$game` object and all the data and helper modules it accesses.

Note that we make a copy of the data in `@BOARD_X` in an anonymous array rather than just taking a reference to it. (That is, we use `[@BOARD_X]` instead of `\@BOARD_X`.) The reason is that `move()` modifies the data stored in the `board` attribute directly. So if we had just taken a reference, then `move()` would have tried to modify the data in `@BOARD_X`, which would cause the test to die, because we declared `@BOARD_X` using `Readonly::Array`. (And that's why we used `Readonly::Array`

```perl
sub linerecords_eq_or_diff {
    my ($got, $expected, $description) = @_;

    local $Test::Builder::Level = $Test::Builder::Level + 1;

    subtest $description => sub {
        my @got_lines = split("\n", $got);
        my @expected_lines = split("\n", $expected);

        my $got_record_types = [ map { _record_type($_) } @got_lines ];
        my $expected_record_types = [ map { _record_type($_) } @expected_lines ];
        my $record_type_diffs =
            Algorithm::Diff::sdiff($expected_record_types, $got_record_types);
        return pass('empty data matches') if ! @$record_type_diffs;

        my ($idx_expected_line, $idx_got_line) = (0, 0);
        for my $record_type_diff (@$record_type_diffs) {
            my ($diff_type, $expected_record_type, $got_record_type) =
              @$record_type_diff;
            my $expected_line = $expected_lines[$idx_expected_line];
            my $got_line = $got_lines[$idx_got_line];

            if ($diff_type eq 'u') {
                _eq_or_diff_line($got_line, $expected_line,
                  "$expected_record_type line matches at index $idx_got_line");
                $idx_expected_line ++;
                $idx_got_line ++;
            }

            if ($diff_type eq '-' || $diff_type eq 'c') {
                fail("expected line missing: $expected_record_type"
                  . " at index $idx_got_line");
                note("expected line: '$expected_line'");
                $idx_expected_line ++;
            }

            if ($diff_type eq '+' || $diff_type eq 'c') {
                fail("unexpected line added: $got_record_type"
                  . " at index $idx_got_line");
                note("got line: '$got_line'");
                $idx_got_line ++;
            }
        }
    };
}
```

Figure 3: Sketch of a function to test line-oriented record data

in the first place, to detect such effects.)

This is a trivial example of parameterized setup, passing parameters to an initializer to setup the test fixture as appropriate for a specific test. This technique particularly comes in handy when you have multi-step setup of interrelated objects, which especially can occur with integration testing.

So let’s say we were testing an invoice class. To instantiate the invoice, you need to instantiate a customer object and an array of invoice items, and the customer refers to a customer address object, and each item is pulled from a database which stores the item price, and you need to populate the database for test as well (which we’ll get to in a later chapter)…

```
sub test_total_cost : Test(1) {
    # 35 lines of additional setup populating the item database
    # and instantiating $customer_address, $customer, and @items
    # (just imagine it all here like 50 Shades of Grey read by
    # Fran Drescher and amplified to 120 dB SPL at 2:00 in the
    # morning) all leading up to...

    my $invoice = Invoice->new(
        customer => $customer,
        items => \@items,
    );

    my $total_cost = $invoice->total_cost;

    cmp_deeply($total_cost, num(9.00, 0.01), 'total cost as expected');
}
```

You end up with 40 lines of test-fixture setup, all for two lines of actual test. Anyone trying to read that test method is going to be inundated with setup code, and it’s going to be much more difficult for them to understand what the test actually does, which is actually the easy part.

It is much clearer to write:

```
sub test_total_cost : Test(1) {
    my $invoice = _new_invoice(
        customer => $CUSTOMER_JOE_SMITH,
        items => [
            $PART_NUMBER_1215,
            $PART_NUMBER_3370,
        ],
    );

    my $total_cost = $invoice->total_cost;
    cmp_deeply($total_cost, num(9.00, 0.01), 'total cost as expected');
}
```

…where `$CUSTOMER_JOE_SMITH` and `$PART_NUMBER_`*xxxx* are reusable data structures. You put all the guts of setting up the test fixture into `_new_invoice()`, and the resulting test method is so much easier to read.

Table-driven tests

We can finally talk about table-driven tests.

Table-driven tests take parameterization to a whole new level. If you find yourself basically writing the same test over and over again with tiny modifications to the data, then you might benefit from a table-driven test.

In our Tic-Tac-Toe code, we already have a couple tests that test invalid moves, `test_move_invalid_location_9` and `test_move_invalid_location_negative_1`. We wrote these in “Testing error cases: catching invalid moves” in Chapter 1. Quoting from that section above:

> We can do similarly with locations that are less than 0, writing a test method called `test_move_invalid_location_negative_1` that is ***identical*** to `test_move_invalid_location_9`, ***except*** that it does `$game->move('X', -1)`. [emphasis added]

And we’re ready to write more such tests:

- …if the piece is neither `'X'` nor `'O'`
- …if the location is already occupied by another piece
- …if X is trying to move but it’s O‘s turn
- …if O is trying to move but it’s X’s turn
- …if the game has already been won

A table-driven test will allow us to add these simply by adding elements to a list of test data. (See Figure 4.)

There’s a little bit more to a table-driven test than a simple test method, but it’s still fairly straightforward. The table-driven test basically has two sections. The first is a list of cases to test, which says *what* to test. Then there’s a loop, the body of which says *how* to test it. Each of these can be read using the four-phases of automated testing—setup, call, verify, and teardown—with uninteresting phases omitted from the case table.

Let’s step through `test_move_invalid`.

Firstly, the number of tests in the test plan is the same as the number of cases in the table. That is, the 2 in Test(2) matches the number of items in @cases. Unfortunately, it's not easy to automate that association, so we usually manage it manually. If you add new tests to @cases but forget to increment the number of tests in the method declaration, the test runner will remind you when you go to run the test.

The @cases array drives the test. The actual elements of the array may be arrayrefs, hashrefs, or anything else that makes sense. I used arrayrefs here, because each set of test data naturally describes the test: for $name, initialize the board to $initial_board (setup), move to $move_args (call), and verify that $error_regex is thrown (verify).

Often, it's clearer to use a hashref as the inner structure, even though that's also more verbose, for example:

```
my @cases = (
    {
        name => 'move to 9',
        initial_board => \@BOARD_EMPTY,
```

```
# One "test" per case in @cases.
sub test_move_invalid : Test(2) {
    my $test = shift;

    my @cases = (
        # [$name, $initial_board, [$piece, $location], $error_regex],
        ['move to 9', \@BOARD_EMPTY, ['X', 9], qr/invalid location: 9/],
        ['move to -1', \@BOARD_EMPTY, ['X', -1], qr/invalid location: -1/],
    );

    for my $case (@cases) {
        my ($name, $initial_board, $move_args, $error_regex) = @$case;

        subtest $name => sub {
            my $game = TicTacToe::BusinessLogic::Game->new(
                board => [@$initial_board],
            );

            throws_ok {
                $game->move(@$move_args);
            } $error_regex, "throws $error_regex";

            _cmp_board($game, $initial_board, 'game board unchanged');
        }
    }
}
```

Figure 4: A table-driven test to test invalid moves

```
            move_args => ['X', 9],
            error_regex => qr/invalid location: 9/,
        },
        {
            name => 'move to -1',
            initial_board => \@BOARD_EMPTY,
            move_args => ['X', -1],
            error_regex => qr/invalid location: -1/,
        },
    );
```

As I said, in this case I think the arrayref form is cleaner and easier to grok. But I've certainly worked on tests that used many parameters, and it would have been impossible to keep them all straight if they were simply laid out in order in an array form.

Then there's a `for` loop that iterates over all of the items in `@cases`, unpacking the data in each, and performing the test:

1. Setup a `$game` object, initializing the state of the board to `$initial_board` as specified.
2. Call `move()` with the specified `$move_args`.
3. Verify that `move()` threw an error that matches the specified `$error_regex`. (Also note that the test description in that line uses `$error_regex` to report to the user what it's checking.)
4. Also verify that the board remains in the state in which it was initialized, that is, that it still matches `$initial_board`.

We put the inner test into a `subtest` block, identified by the specified `$name`. This isn't required, but it can make the resulting test output easier to sift through:

```
$ prove -lv t/lib/TicTacToe/BusinessLogic/Game/move.t
t/lib/TicTacToe/BusinessLogic/Game/move.t ..
#
# t::lib::TicTacToe::BusinessLogic::Game::move->test_move_O_to_1
1..4
ok 1 - game board with O added at location 1
#
# t::lib::TicTacToe::BusinessLogic::Game::move->test_move_X_to_0
ok 2 - game board with X at location 0
#
# t::lib::TicTacToe::BusinessLogic::Game::move->test_move_invalid
# Subtest: move to 9
    ok 1 - throws (?^u:invalid location: 9)
    ok 2 - game board unchanged
    1..2
```

```
ok 3 - move to 9
# Subtest: move to -1
    ok 1 - throws (?^u:invalid location: -1)
    ok 2 - game board unchanged
    1..2
ok 4 - move to -1
ok
All tests successful.
Files=1, Tests=4,  0 wallclock secs ( 0.02 usr  0.01 sys +  0.17 cusr  0.01 csys =
0.21 CPU)
Result: PASS
```

(I also commented out the corresponding lines in **`Game.pm`**, as before, and reran the test and saw it fail.)

Now we're finally ready to add tests and code to handle additional invalid moves.

```
    my @cases = (
        # [$name, $initial_board, [$piece, $location], $error_regex],
        ['move to 9', \@BOARD_EMPTY, ['X', 9], qr/invalid location: 9/],
        ['move to -1', \@BOARD_EMPTY, ['X', -1], qr/invalid location: -1/],
        ['piece is H', \@BOARD_EMPTY, ['H', 0], qr/invalid piece: H/],
        ['move to occupied location', \@BOARD_X, ['O', 0], qr/occupied by X/],
        ['O moves first', \@BOARD_EMPTY, ['O', 2], qr/it's X's turn/],
        ['X moves twice', \@BOARD_X, ['X', 1], qr/it's O's turn/],
        ['O moves twice', \@BOARD_XO, ['O', 2], qr/it's X's turn/],
        ['O moves after X won', \@BOARD_X_WINS, ['O', 5], qr/X already won/],
    );
```

We actually add these one at a time, run it, see it fail, and then implement the code that makes the test pass. As an alternative, you can write up all the test cases, then comment them all out, uncommenting them one at a time. Having only one failing test active at a time focuses development only on that bit of functionality.

In order to implement the "it's X's turn" and "X already won" cases, I also implemented `next_player()` and `winner()` methods, each using the same process, and each using a table-driven test. You can see all this code in the GitHub repository, on the **`TicTacToe/task/detect_invalid_moves`** branch. And you'll probably notice that this code is not exactly efficient and can most likely be optimized. That's a different book, but since this code is thoroughly unit-tested, we can start down that road with confidence.

Shared test modules

I've merely glossed over it above, but in implementing the `next_player()` and `winner()` methods, I created **`next_player.t`** and **`winner.t`**. Each of these contains a number of `@BOARD_`*foo* constants specifying board configurations for testing. And some of these board configurations are repeated in multiple **`.t`** files.

As we eliminated repetitions of the same array literal in multiple test methods by moving the literals into symbols, we can eliminate duplication between multiple test modules by moving the symbols into a shared module and let each test import the symbols that it needs.

It's common to put shared test code in the **lib/** hierarchy, for example, under `TicTacToe::Test`. We actually already have such a file in the project. I didn't draw attention to it in Chapter 2, but our test modules derive from `TicTacToe::Test::Class` (see Figure 1), which itself derives from `Test::Class`. So this custom test superclass contains code that is shared between all the test modules in the system.

However, we can put the helper module closer to the tests that use it, in the same directory as the **.t** files themselves. This follows a fundamental rule of good design: keep things that work together close to each other; keep things that work separately apart.

We create a file **t/lib/TicTacToe/BusinessLogic/Game/BoardConfigs.pm**, which starts with:

```
package t::lib::TicTacToe::BusinessLogic::Game::BoardConfigs;
use 5.026;
use strict;
use warnings;

use parent 'Exporter';

our @EXPORT = qw();

our @EXPORT_OK = qw(
    @BOARD_EMPTY @BOARD_X @BOARD_XO @BOARD_DRAW
    @BOARD_X_WINS_COL_0 @BOARD_O_WINS_COL_1 @BOARD_X_WINS_COL_2
    @BOARD_X_WINS_ROW_0 @BOARD_O_WINS_ROW_1 @BOARD_O_WINS_ROW_2
    @BOARD_X_WINS_BACKSLASH @BOARD_O_WINS_SLASH
);
```

In this module, we have blocks like:

```
Readonly::Array our @BOARD_EMPTY =>
    ( ' ', ' ', ' ',
      ' ', ' ', ' ',
      ' ', ' ', ' ' );
```

And then in each relevant **.t** file, a line like:

```
use t::lib::TicTacToe::BusinessLogic::Game::BoardConfigs qw(@BOARD_EMPTY);
```

(If you actually look at the code I wrote, you'll see it's more elaborate than this. I defined export tags for groups of `@BOARD_`*foo* variables that tend to be imported together. I also renamed one of the variables in order to be consistent between different test modules: the same board configuration had been called one thing in one place and another thing in another place.)

There's one other hitch. Perl needs to be able to find the `.pm` file in `@INC`. Perl used to search in the current directory by default, so that as long as you ran **`prove`** from project root, it would be able to find these modules. However, beginning with Perl 5.26, this feature was removed for security reasons. If an attacker were able to gain write access to the current directory (wherever that is), then they could potentially inject code.

There are several approaches that can work:

1. **Include `-I.` on the `prove` command line.** This will add the current directory to `@INC`. It's perfectly secure to include relative paths from the command line itself, just not automatically in the code. This has the advantage of being clean and simple and not requiring any changes to the project, but it does require a non-standard command-line switch.

2. **Use `FindBin` to add the top-level directory to `@INC` relative to the location of the `.t` file.** The test script can use `FindBin` to discover its path and then navigate up the appropriate number of parent levels to get to the directory that contains **`t/`**. This is messier than option #1, and the code needs to be included in every relevant **`.t`** file, but it does not require any change to the **`prove`** command line.

3. **Add the top-level directory to `@INC` relative to the location of the test superclass.** That is, `TicTacToe::Test::Class` can find its own path in `%INC`, navigate up to the project root, and add that directory to `@INC`. This requires no change to how we invoke **`prove`** and puts the search-path code in a single place rather than in every test script. And as long as this module remains the same number of levels below the project root, it will continue to work. The disadvantage is that this solution requires that `TicTacToe::Test::Class` know something about where it's installed in the filesystem relative to the **`t/`** directory, which couples it tightly to that aspect of the project directory structure.

4. **Add a symbolic link to `lib/`.** This is the option I chose to use in the **`TestingGuideFiles`** repository. In **`lib/`**, I added a symbolic link called **`t`**, which points to **`../t`**. This has the advantages of being implemented in a single place, working with the existing **`prove`** command line, being narrowly targeted (only adding **`t/`** to the library directory instead of, effectively, everything in the project root), and keeping knowledge about the project directory structure in the project directory tree. The potential disadvantage is portability: it may require patching if someone tries to check the project out on a non-POSIX-compliant filesystem. (Sorry, Windows users. You got the short end of the stick.)

So we refactor each of the **`.t`** files to use `BoardConfigs` and delete the `@BOARD_`*foo* variables in each test. Comment out the code under test to verify that the refactored tests fail, and un-comment to see that they still pass.

A tangent to fix a small bug

This refactoring actually revealed a bug. You'll find that's very common, that testing and refactoring code with reveal hidden bugs.

Putting the board configurations into a single module, I noticed that the same list of boards was used both for testing `move()` and for testing `next_player()`. That makes a lot of sense, as `next_player()` tells us some prospective information about the the next move, and in fact that relationship is expressed in the `move()` code:

```
    my $next_player = $self->next_player;
    die "$piece attempted to move, but it's ${next_player}'s turn\n"
        unless $piece eq $next_player;
```

Except there's something wrong: `next_player()` is never tested on `@BOARD_DRAW`. To be fair, `move()` isn't either, but that test isn't needed, because `move()` already disallows moves to an already-existing space. But `next_player()` should return `undef` if the board is completely full. I added a new test to exercise that case, then a line of code in `next_player()` to make it pass.

This bug fix is on the branch **`TicTacToe/task/next_player_draw`**.