Chapter 1

TEST-DRIVEN DEVELOPMENT

To vouch this, is no proof,
Without more wider and more overt test

- Othello, Act 1 Scene 3
William Shakespeare

From programmers to users, everyone involved in software development agrees: testing is good. So why are so many systems so badly tested? There are several problems with the traditional approach to testing:

- If testing is not comprehensive enough, errors can make it into production and cause potentially devastating problems.
- Testing is often done after all the code is written, or at least after the programmer who wrote the code has moved on to other things. When you are no longer living and breathing a particular program, it takes some time and effort to back *into* the code enough to deal with the problems.
- Often tests are written by programmers other than those who wrote the code. Since they may not understand all of the details of the code, it is possible that they may miss important tests.
- If the test writers base their tests on documentation or other artifacts *other than the code*, any extent to which those artifacts are out of date will cause problems.
- If tests are not automated, they most likely will not be performed frequently, regularly, or in exactly the same way each time.
- Finally, it is quite possible with traditional approaches to fix a problem in a way that creates problems elsewhere. The existing test infrastructure may or may not find these new problems.

Test-Driven Development solves all of these problems, and others.

- The programmer does the testing, working with tests while the code is freshly in mind. In fact, the code is based on the tests, which guarantees testability, helps ensure exhaustive test coverage, and keeps the code and tests in sync. All tests are automated. They are run quite frequently and identically each time.

- Exhaustive test coverage means that if a bug is introduced during debugging, the test scaffolding finds it immediately and pinpoints its location. And the test-debug cycle is kept quite short: there are no lengthy delays between the discovery of a bug and its repair.

- Finally, when the system is delivered, the exhaustive test scaffolding is delivered with it, making future changes and extensions to it easier.

So from a pure testing standpoint, before we begin to discuss non-testing benefits, TDD is superior to traditional testing approaches. You get more thoroughly tested code, period.

But you do indeed get much more than that. You get simpler designs. You get systems that reveal intent (describe themselves) clearly. The tests themselves help describe the system. You get extremely low-defect systems that start out robust, are robust at the end, and stay robust all the time. At the end of every day, the latest build is robust.

These are benefits for all project stakeholders. But perhaps the most immediate and most tangible benefit is exclusively the programmer's: more fun. TDD gives you, the programmer, small, regular, frequent doses of positive feedback while you work. You have tangible evidence that you are making progress, and that your code works.

There is a potential problem with all this, of course. It is more addictive than caffeine. Once you're hooked, you'll want to program this way, and only this way, from then on. And I for one certainly hope you do.

This chapter will give you an overview of Test-Driven Development, including a short example of a programming session.

WHAT IS TEST-DRIVEN DEVELOPMENT?

Test-Driven Development (TDD) is a style of development where:

- you maintain an exhaustive suite of Programmer Tests,

- no code goes into production unless it has associated tests,

- you write the tests first,

- the tests determine what code you need to write.

Let's look at each of these in turn.

Maintain an exhaustive suite of Programmer Tests

You have Programmer Tests to test that your classes exhibit the proper behavior. Programmer Tests are written by the developer who writes the code being tested. They're called Programmer Tests because although they are similar to unit tests, they are written for a different reason. Unit tests are written to test that the code you've written works. Programmer Tests are written to define what it means for the code to work. Finally, they're called *Programmer* Tests to differentiate them from the tests that the *Customer* writes (called, logically enough, *Customer Tests*) to test that the system behaves as required form the point of view of a user.

Using Test-Driven Development implies, in theory, that you have an exhaustive test suite. This is because there is no code unless there is a test that requires it in order to pass. You write the test, then (and not until then) write the code that is tested by the test. There should be no code in the system which was not written in response to a test. Hence, the test suite is, by definition, exhaustive.

No code goes into production unless it has associated tests

One of eXtreme Programming's tenets is that a feature does not exist until there is a suite of tests to go with it. The reason for this is that everything in the system has to be testable as part of the safety net that gives you confidence and courage. Confidence that all the code tests clean gives you the courage (not to mention the simple ability) to refactor and integrate. How can you possibly make changes to the code without some way to confidently tell whether you have broken the previous behavior? How can you integrate if you don't have a suite of tests that will immediately (or at least in a short time) tell you if you have inadvertently broken some other part of the code?

Write the tests first

Now we're getting eXtreme. What do I mean by *write the tests first*? I mean that when you have a task to do (i.e., some bit of functionality to implement) you write code that will test that the functionality works as required before you implement the functionality itself.

Furthermore, you write a little bit of test, followed by just enough code to make that test pass, then a bit more test, and a bit more code, test, code, test, code, etc.

Tests determine what code you need to write

By writing only the code required to pass the latest test, you are putting a limit on the code you will write. You write only enough to pass the test, *no more*. That means that you do *the simplest thing that could possibly work*. I think an example is in order. Let's say you are working on a list class. The logical place to start is with the behavior of an empty list (it makes sense to start with the basis, or simplest, case). So you write the test:

```
public void testEmptyList() {
    MovieList emptyList = new MovieList();
    assertEquals("Empty list should have size of 0", 0, emptyList.size());
}
```

To pass this test we need a MovieList class that has a size() method.

When you are working this way, you want to work in small increments. . . sometimes increments that seem ridiculously small. When you grasp the significance of this, you will be on your way to mastering TDD. Later we'll explore the important and sometimes unexpected benefits and side effects of testing and coding in tiny increments.

LET THE COMPUTER TELL YOU

Write your tests (and your code for that matter) without worrying about what classes or methods you will need to add. Don't even bother keeping a To Do list. Well, at least in terms of what classes, methods, etc., you need to create. You will likely want a To Do list to keep track of tests you want to write and other higher level items. Just write your test and compile.

If you need to add a class or method the compiler will tell you. It provides a better To Do list than you could, and faster. In the previous example when I compile[1] after writing the test (with nothing else written) I get the error:

```
MovieList cannot be resolved or is not a type.
```

This immediately tells me that I need to create a new MovieList class, so I do:

```
public class MovieList {
}
```

I compile again and get another error:

```
The method size() is undefined for the type MovieList
```

In response to this I add a stub size() method:

```
public int size() {
  return 0;
}
```

Now it will compile. Run the test, and it works. Due to Java requiring a return statement when a return type is defined, we need to combine the steps of creating the method and adding the simplest return statement. I have made a

[1] Modern Java programming environments will alert me to these missing items even before I compile. Furthermore, they will offer solutions and do the work of creating the stubs for me.

habit of always stubbing methods to return the simplest value possible (i.e., 0, false, or null).

What!?! Just return 0? That can't be right. Ah... but it is right. It is the simplest thing that could possibly work to pass the test we just wrote. As we write more tests we will likely need to revisit the size() method, generalizing and refactoring, but for now return 0 is all that is required.

A QUICK EXAMPLE

Let's take a peek into the development of the project from later in the book. We have a Movie class which now needs to accept multiple ratings (e.g., 3 as in "3 stars out of 5") and give access to the average.

As we go through the example, we will be alluding to a metaphor for the TDD flow originated by William Wake: The TDD Traffic Light[URL 9][URL 61].

We start by writing a test, and we start the test by making an assertion that we want to be true:

```
public void testRating() {
  assertEquals("Bad average rating.", 4, starWars.getAverageRating());
}
```

Now we need to set the stage for that assertion to be true. To do that we'll add some rating to the Movie:

```
public void testRating() {
  starWars.addRating(3);
  starWars.addRating(5);
  assertEquals("Bad average rating.", 4, starWars.getAverageRating());
}
```

Finally, we need to create the Movie instance we are working with:

```
public void testRating() {
  Movie starWars = new Movie("Star Wars");
  starWars.addRating(3);
  starWars.addRating(5);
  assertEquals("Bad average rating.", 4, starWars.getAverageRating());
}
```

When we compile this, the compiler complains that addRating(int) and getAverageRating() are undefined. This is our yellow light. Now we make it compile by adding the following code to Movie:

```
public void addRating(int newRating) {
}
```

```
public int getAverageRating() {
  return 0;
}
```

Note that since we are using Java, we must provide a return value for getAverageRating() since we've said it returns an int.

Now it compiles, but the test fails. This is the red light (aka red bar). This term is derived by the JUnit interfaces that present a progress bar that advances as tests are run. As long as all tests pass, the bar is green. As soon as a test fails, the bar turns red and remains red. The message we get is:

```
Bad average rating. expected:<4> but was:<0>
```

Now we have to make the test pass. We add code to getAverageRating() to make the test pass:

```
public int getAverageRating() {
  return 4;
}
```

Recompile and rerun the test. Green light! Now we refactor to remove the duplication and other smells that we introduced when we made the test pass.

You're probably thinking "Duplication... what duplication?" It's not always obvious at first. We'll start by looking for constants that we used in making the test work. Sure enough, look at getAverageRating(). It returns a constant. Remember that we set the test up to get the desired result. How did we do that? In this case we gave the movie two ratings: 3 and 5. The average result is the 4 that we are returning. So, that 4 is duplicated. We provide the information required to compute it, as well as returning it as a constant. Returning a constant when we can compute its value is a form of duplication. Let's get rid of it.

Our first step is to rewrite that constant into something related to the provided information:

```
public int getAverageRating() {
  return (3 + 5) / 2;
}
```

Compile and run the tests. We're OK. We have the courage to continue. The 3 and 5 are duplicate with the arguments to addRating() so let's capture them. Since we add the constants we can simply accumulate the arguments. First we add a variable to accumulate them:

```
private int totalRating = 0;
```

Then we add some code to addRating():

```
public void addRating(int newRating) {
  totalRating += newRating;
}
```

Now we use it in getAverageRating():

```
public int getAverageRating() {
  return totalRating / 2;
}
```

Compile, test, it works! We're not finished yet, though. While we were refactoring we introduced another constant: the 2 in getAverageRating(). The duplication here is a little subtler. The 2 is the number ratings we added, i.e., the number of times addRating() was called. We need to keep track of that in order to get rid of the 2.

Like before, start by defining a place for it:

```
private int numberOfRatings = 0;
```

Compile, run the tests, green. Now, increment it every time addRating() is called:

```
public void addRating(int newRating) {
  totalRating += newRating;
  numberOfRatings++;
}
```

Compile, run the tests, green. OK, finally we replace the constant 2 with numberOfRatings:

```
public int getAverageRating() {
  return totalRating / numberOfRatings;
}
```

Compile, run the tests, green. OK, we're done. If we want to reinforce our confidence in what we did, we can add more calls to addRating() and check against the appropriate expected average. For example:

```
public void testLotsOfRatings() {
  Movie godzilla = new Movie("Godzilla");
  godzilla.addRating(1);
  godzilla.addRating(5);
  godzilla.addRating(1);
  godzilla.addRating(2);
  assertEquals("Bad average rating.", 2, godzilla.getAverageRating());
}
```

I need to underline the fact that I recompiled and ran the tests after each little change above. This cannot be stressed enough. Running tests after each small change gives us confidence and reassurance. The result is that we have courage to continue, one little step at a time. If at any point a test failed we know exactly what change caused the failure: the last one. We back it out and rerun the tests. The tests should pass again. Now we can try again... with courage.

The above example shows one school of thought when it comes to cleaning up code. In it we worked to get rid of the duplication that was embodied in the constant. Another school of thought would leave the constant 4 in place and write another test that added different ratings, and a different number of them. This second test would be designed to require a different returned average. This would force us to refactor and generalize in order to get the test to pass.

Which approach should you take? It really depends on how comfortable you are with what you are attempting. Remember that you do have the test to safeguard you. As long as the test runs, you know that you haven't broken anything. In either case you will want to write that second test: either to drive the generalization, or to verify it.

SUMMARY

We've explored what Test-Driven Development is:

- an exhaustive suite of Programmer Tests,
- no code without tests,
- tests first,
- tests determine the code.

We've seen how to leverage feedback from the computer to keep track of what we should do next: if we need to create a class, method, variable, etc., the system will let us know.

We've even seen a quick example of TDD in action, step by step, building some code to maintain the average rating of a movie.

However, before we can jump in and start practicing it in earnest, we need to make sure we have a few basic techniques and skills that TDD builds on. The next few chapters will explore these.

Agile Modeling and TDD

A primary benefit of both modeling and TDD is that they promote a *think before you act* approach to development. Just because they offer the same type of benefit doesn't mean that they are incompatible with one another. Instead, experience shows that we can and should model on a project taking a TDD approach.

Consider XP projects that clearly take a TDD approach. Modeling is definitely an important part of XP. XP practitioners work with user stories, and user stories are clearly agile models. XP practitioners also create CRC cards whenever they need to, also agile models. In *eXtreme Programming Explained* [8], Kent Beck even includes sketches of class diagrams. Heresy? No. Just common sense. If creating a model can help our software development efforts then that's what we do. It's as simple as that.

Creating agile models can help our TDD efforts because they can reveal the need for some tests. As an agile modeler sketches a diagram they will always be thinking, "How can I test this?" in the back of their minds because they will be following the practice *consider testability*. This will lead to new test cases. Furthermore, we are likely to find that some of our project stakeholders or even other developers simply don't think in terms of tests; instead, they are visual thinkers. There's nothing wrong with this as long as we recognize it as an issue and act accordingly—use visual modeling techniques with visual thinkers and test-driven techniques with people that have a testing mindset.

TDD can also improve our agile modeling efforts. Following a test-first approach, agile developers quickly discover whether their ideas actually work or not—the tests will either validate their models or not—providing rapid feedback regarding the ideas captured within the models. This fits in perfectly with AM's practice of *Prove it With Code*.

Agile Modeling (AM) and Test-Driven Development (TDD) go hand in hand. The most effective developers have a wide range of techniques in their intellectual toolboxes, and AM and TDD should be among those techniques.

See Appendix B for more information on Agile Modeling.