6
Mocking – Removing External Dependencies

"Talk is cheap. Show me the code."

– Linus Torvalds

TDD is about speed. We want to quickly demonstrate whether an idea, concept, or implementation is valid or not. Further on, we want to run all tests quickly. A major bottleneck to this speed is external dependencies. Setting up the DB data required by tests can be time-consuming. The execution of tests that verify code that uses third-party APIs can be slow. Most importantly, writing tests that satisfy all external dependencies can become too complicated to be worthwhile. Mocking both external and internal dependencies helps us solve these problems.

We'll build on what we did in `Chapter 3`, *Red-Green-Refactor – From Failure Through Success until Perfection*. We'll extend Tic-Tac-Toe to use MongoDB as data storage. None of our unit tests will actually use MongoDB since all communications will be mocked. At the end, we'll create an integration test that will verify that our code and MongoDB are indeed integrated.

The following topics will be covered in this chapter:

- Mocking
- Mockito
- Tic-Tac-Toe v2 requirements
- Developing Tic-Tac-Toe v2
- Integration tests

Mocking

Everyone who has done any of the applications more complicated than *Hello World* knows that Java code is full of dependencies. There can be classes and methods written by other members of the team, third-party libraries, or external systems that we communicate with. Even libraries found inside JDK are dependencies. We might have a business layer that communicates with the data access layer which, in turn, uses database drivers to fetch data. When working with unit tests, we take dependencies even further and often consider all public and protected methods (even those inside the class we are working on) as dependencies that should be isolated.

When doing TDD on the unit tests level, creating specifications that contemplate all those dependencies can be so complex that the tests themselves would become bottlenecks. Their development time can increase so much that the benefits gained with TDD quickly become overshadowed by the ever-increasing cost. More importantly, those same dependencies tend to create such complex tests that they contain more bugs than the implementation itself.

The idea of unit testing (especially when tied to TDD) is to write specifications that validate whether the code of a single unit works regardless of dependencies. When dependencies are internal, they are already tested, and we know that they do what we expect them to do. On the other hand, external dependencies require trust. We must believe that they work correctly. Even if we don't, the task of performing deep testing of, let's say, the JDK `java.nio` classes is too big for most of us. Besides, those potential problems will surface when we run functional and integration tests.

While focused on units, we must try to remove all dependencies that a unit may use. Removal of those dependencies is accomplished through a combination of design and mocking.

The benefits of using mocks include reduced code dependency and faster text execution.

Mocks are prerequisites for the fast execution of tests and the ability to concentrate on a single unit of functionality. By mocking dependencies external to the method that is being tested, the developer is able to focus on the task at hand without spending time setting them up. In a case of bigger or multiple teams working together, those dependencies may not even be developed. Also, the execution of tests without mocks tends to be slow. Good candidates for mocks are databases, other products, services, and so on.

Before we go deeper into mocks, let us go through reasons why one would employ them in the first place.

Why mocks?

The following list represents some of the reasons why we employ mock objects:

- The object generates nondeterministic results. For example, `java.util.Date()` provides a different result every time we instantiate it. We cannot test that its result is as expected:

```
java.util.Date date = new java.util.Date();
date.getTime(); // What is the result this method returns?
```

- The object does not yet exist. For example, we might create an interface and test against it. The object that implements that interface might not have been written at the time we test code that uses that interface.
- The object is slow and requires time to process. The most common example would be databases. We might have a code that retrieves all records and generates a report. This operation can last minutes, hours, or, in some cases, even days.

The preceding reasons in the support of mock objects apply to any type of testing. However, in the case of unit tests and, especially, in the context of TDD, there is one more reason, perhaps more important than others. Mocking allows us to isolate all dependencies used by the method we are currently working on. This empowers us to concentrate on a single unit and ignore the inner workings of the code that the unit invokes.

Terminology

Terminology can be a bit confusing, especially since different people use different names for the same thing. To make things even more complicated, mocking frameworks tend not to be consistent when naming their methods.

Before we proceed, let us briefly go through terminology.

Test doubles is a generic name for all of the following types:

- Dummy object's purpose is to act as a substitute for a real method argument
- Test stub can be used to replace a real object with a test-specific object that feeds the desired indirect inputs into the system under test
- **Test Spy** captures the indirect output calls made to another component by the **System Under Test** (**SUT**) for later verification by the test
- Mock object replaces an object the SUT depends on, with a test-specific object that verifies that it is being used correctly by the SUT
- Fake object replaces a component that the SUT depends on with a much lighter-weight implementation

If you are confused, it may help you to know that you are not the only one. Things are even more complicated than this, since there is no clear agreement, nor a naming standard, between frameworks or authors. Terminology is confusing and inconsistent, and the terms mentioned earlier are by no means accepted by everyone.

To simplify things, throughout this book we'll use the same naming used by Mockito (our framework of choice). This way, methods that you'll be using will correspond with the terminology that you'll be reading further on. We'll continue using mocking as a general term for what others might call **test doubles**. Furthermore, we'll use a mock or spy term to refer to `Mockito` methods.

Mock objects

Mock objects simulate the behavior of real (often complex) objects. They allow us to create an object that will replace the real one used in the implementation code. A mocked object will expect a defined method with defined arguments to return the expected result. It knows in advance what is supposed to happen and how we expect it to react.

Let's take a look at one simple example:

```
TicTacToeCollection collection = mock(TicTacToeCollection.class);
assertThat(collection.drop()).isFalse();
doReturn(true).when(collection).drop();

assertThat(collection.drop()).isTrue();
```

First, we defined `collection` to be a `mock` of `TicTacToeCollection`. At this moment, all methods from this mocked object are fake and, in the case of Mockito, return default values. This is confirmed in the second line, where we `assert` that the `drop` method returns `false`. Further on, we specify that our mocked object collection should return `true` when the `drop` method is invoked. Finally, we `assert` that the `drop` method returns `true`.

We created a mock object that returns default values and, for one of its methods, defined what should be the return value. At no point was a real object used.

Later on, we'll work with spies that have this logic inverted; an object uses real methods unless specified otherwise. We'll see and learn more about mocking soon when we start extending our Tic-Tac-Toe application. Right now, we'll take a look at one of the Java mocking frameworks called Mockito.

Mockito

Mockito is a mocking framework with a clean and simple API. Tests produced with Mockito are readable, easy to write, and intuitive. It contains three major static methods:

- `mock()`: This is used to create mocks. Optionally, we can specify how those mocks behave with `when()` and `given()`.
- `spy()`: This can be used for partial mocking. Spied objects invoke real methods unless we specify otherwise. As with `mock()`, behavior can be set for every public or protected method (excluding static). The major difference is that `mock()` creates a fake of the whole object, while `spy()` uses the real object.
- `verify()`: This is used to check whether methods were called with given arguments. It is a form of assert.

We'll go deeper into Mockito once we start coding our Tic-Tac-Toe v2 application. First, however, let us quickly go through a new set of requirements.

Tic-Tac-Toe v2 requirements

The requirements of our Tic-Tac-Toe v2 application are simple. We should add a persistent storage so that players can continue playing the game at some later time. We'll use MongoDB for this purpose.

Add MongoDB persistent storage to the application.

Developing Tic-Tac-Toe v2

We'll continue where we left off with Tic-Tac-Toe in `Chapter 3`, *Red-Green-Refactor – From Failure Through Success until Perfection*. The complete source code of the application developed so far can be found at `https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo.git`. Use the **VCS|Checkout from Version Control|Git** option from the **IntelliJ IDEA** to clone the code. As with any other project, the first thing we need to do is add the dependencies to `build.gradle`:

```
dependencies {
    compile 'org.jongo:jongo:1.1'
    compile 'org.mongodb:mongo-java-driver:2.+'
    testCompile 'junit:junit:4.12'
    testCompile 'org.mockito:mockito-all:1.+'
}
```

Importing the MongoDB driver should be self-explanatory. Jongo is a very helpful set of utility methods that make working with Java code much more similar to the Mongo query language. For the testing part, we'll continue using JUnit with the addition of Mockito mocks, spies, and validations.

You'll notice that we won't install MongoDB until the very end. With Mockito, we will not need it, since all our Mongo dependencies will be mocked.

Once dependencies are specified, remember to refresh them in the IDEA **Gradle Projects** dialogue.

The source code can be found in the `00-prerequisites` branch of the `tdd-java-ch06-tic-tac-toe-mongo` Git repository (`https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo/branch/00-prerequisites`).

Now that we have prerequisites set, let's start working on the first requirement.

Requirement 1 – store moves

We should be able to save each move to the DB. Since we already have all the game logic implemented, this should be trivial to do. Nonetheless, this will be a very good example of mock usage.

Implement an option to save a single move with the turn number, the *x* and *y* axis positions, and the player (`X` or `O`).

We should start by defining the Java bean that will represent our data storage schema. There's nothing special about it, so we'll skip this part with only one note.

Do not spend too much time defining specifications for Java boilerplate code. Our implementation of the bean contains overwritten `equals` and `hashCode`. Both are generated automatically by IDEA and do not provide a real value, except to satisfy the need to compare two objects of the same type (we'll use that comparison later on in specifications). TDD is supposed to help us design better and write better code. Writing 15-20 specifications to define boilerplate code that could be written automatically by IDE (as is the case with the `equals` method) does not help us meet these objectives. Mastering TDD means not only learning how to write specifications, but also knowing when it's not worth it.

That being said, consult the source code to see the bean specification and implementation in it's entirety.

The source code can be found in the `01-bean` branch of the `tdd-java-ch06-tic-tac-toe-mongo` Git repository (`https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo/branch/01-bean`). The particular classes are `TicTacToeBeanSpec` and `TicTacToeBean`.

Now, let's go to a more interesting part (but still without mocks, spies, and validations). Let's write specifications related to saving data to MongoDB.

For this requirement, we'll create two new classes inside the `com.packtpublishing.tddjava.ch03tictactoe.mongo` package:

- `TicTacToeCollectionSpec` (inside `src/test/java`)
- `TicTacToeCollection` (inside `src/main/java`)

Specification – DB name

We should specify what the name of the DB that we'll use will be:

```
@Test
public void whenInstantiatedThenMongoHasDbNameTicTacToe() {
  TicTacToeCollection collection = new TicTacToeCollection();
  assertEquals(
     "tic-tac-toe",
collection.getMongoCollection().getDBCollection().getDB().getName()
);
}
```

We are instantiating a new `TicTacToeCollection` class and verifying that the DB name is what we expect.

Implementation

The implementation is very straightforward, as follows:

```
private MongoCollection mongoCollection;
protected MongoCollection getMongoCollection() {
  return mongoCollection;
}
public TicTacToeCollection() throws UnknownHostException {
  DB db = new MongoClient().getDB("tic-tac-toe");
  mongoCollection = new Jongo(db).getCollection("bla");
}
```

When instantiating the `TicTacToeCollection` class, we're creating a new `MongoCollection` with the specified DB name (`tic-tac-toe`) and assigning it to the local variable.

Bear with us. There's only one more specification left until we get to the interesting part where we'll use mocks and spies.

Specification – a name for the Mongo collection

In the previous implementation, we used `bla` as the name of the collection because `Jongo` forced us to put some string. Let's create a specification that will define the name of the Mongo collection that we'll use:

```
@Test
public void whenInstantiatedThenMongoCollectionHasNameGame() {
  TicTacToeCollection collection = new TicTacToeCollection();
  assertEquals(
            "game",
            collection.getMongoCollection().getName());
}
```

This specification is almost identical to the previous one and probably self explanatory.

Implementation

All we have to do to implement this specification is change the string we used to set the collection name:

```
public TicTacToeCollection() throws UnknownHostException {
  DB db = new MongoClient().getDB("tic-tac-toe");
  mongoCollection = new Jongo(db).getCollection("game");
}
```

Refactoring

You might have got the impression that refactoring is reserved only for the implementation code. However, when we look the objectives behind refactoring (more readable, optimal, and faster code), they apply as much to specifications as to the implementation code.

The last two specifications have the instantiation of the `TicTacToeCollection` class repeated. We can move it to a method annotated with `@Before`. The effect will be the same (the class will be instantiated before each method annotated with `@Test` is run) and we'll remove the duplicated code. Since the same instantiation will be needed in further specs, removing duplication now will provide even more benefits later on. At the same time, we'll save ourselves from throwing `UnknownHostException` over and over again:

```
TicTacToeCollection collection;

@Before
public void before() throws UnknownHostException {
```

```
  collection = new TicTacToeCollection();
}
@Test
public void whenInstantiatedThenMongoHasDbNameTicTacToe() {
//        throws UnknownHostException {
//  TicTacToeCollection collection = new TicTacToeCollection();
  assertEquals(
    "tic-tac-toe",
collection.getMongoCollection().getDBCollection().getDB().getName()
);
}

@Test
public void whenInstantiatedThenMongoHasNameGame() {
//        throws UnknownHostException {
//  TicTacToeCollection collection = new TicTacToeCollection();
  assertEquals(
    "game",
    collection.getMongoCollection().getName());
}
```

Use setup and teardown methods. The benefits of these allow preparation or setup and disposal or teardown code to be executed before and after the class or each test method.

In many cases, some code needs to be executed before the test class or each method in a class. For this purpose, JUnit has the `@BeforeClass` and `@Before` annotations that should be used in the setup phase. The `@BeforeClass` executes the associated method before the class is loaded (before the first test method is run). `@Before` executes the associated method before each test is run. Both should be used when there are certain preconditions required by tests. The most common example is setting up test data in the (hopefully in-memory) database. On the opposite end are the `@After` and `@AfterClass` annotations, which should be used as the teardown phase. Their main purpose is to destroy the data or state created during the setup phase or by tests themselves. Each test should be independent from others. Moreover, no test should be affected by the others. The teardown phase helps maintain the system as if no test were previously executed.

Now let's do some mocking, spying, and verifying!

Specification – adding items to the Mongo collection

We should create a method that saves data to MongoDB. After studying Jongo documentation, we discovered that there is the `MongoCollection.save` method, which does exactly that. It accepts any object as a method argument and transforms it (using Jackson) into JSON, which is natively used in MongoDB. The point is that after playing around with Jongo, we decided to use and, more importantly, trust this library.

We can write Mongo specifications in two ways. One more traditional and appropriate for **End2End** (**E2E**) or integration tests would be to bring up a MongoDB instance, invoke the Jongo's save method, query the database, and confirm that data has indeed been saved. It does not end here, as we would need to clean up the database before each test to always guarantee that the same state is unpolluted by the execution of previous tests. Finally, once all tests are finished executing, we might want to stop the MongoDB instance and free server resources for some other tasks.

As you might have guessed, there is quite a lot of work involved for a single test written in this way. Also, it's not only about work that needs to be invested into writing such tests. The execution time would be increased quite a lot. Running one test that communicates with a DB does not take long. Running ten tests is usually still fast. Running hundreds or thousands can take quite a lot of time. What happens when it takes a lot of time to run all unit tests? People lose patience and start dividing them into groups or give up on TDD all together. Dividing tests into groups means that we lose confidence in the fact that nothing got broken, since we are continuously testing only parts of it. Giving up on TDD... Well, that's not the objective we're trying to accomplish. However, if it takes a lot of time to run tests, it's reasonable to expect developers to not want to wait until they are finished running before they move to the next specification, and that is the point when we stop doing TDD. What is a reasonable amount of time to allow our unit tests to run? There is no one-fits-all rule that defines this; however, as a rule of thumb, if the time is longer than 10-15 seconds, we should start worrying, and dedicate time to optimizing them.

Tests should run quickly. The benefits are that the tests are used often.

If it takes a lot of time to run tests, developers will stop using them or run only a small subset related to the changes they are making. One benefit of fast tests, besides fostering their usage, is fast feedback. The sooner the problem is detected, the easier it is to fix it. Knowledge about the code that produced the problem is still fresh. If a developer has already started working on the next feature while waiting for the completion of the execution of tests, they might decide to postpone fixing the problem until that new feature is developed. On the other hand, if they drops their current work to fix the bug, time is lost in context switching.

If using live DB to run unit tests is not a good option, then what is the alternative? Mocking and spying! In our example, we know which method of a third-party library should be invoked. We also invested enough time to trust this library (besides integration tests that will be performed later on). Once we know how to use the library, we can limit our job to verifying that correct invocations of that library have been made.

Let us give it a try.

First, we should modify our existing code and convert our instantiation of the `TicTacToeCollection` into a `spy`:

```
import static org.mockito.Mockito.*;
...
@Before
public void before() throws UnknownHostException {
  collection = spy(new TicTacToeCollection());
}
```

Spying on a class is called **partial** mocking. When applied, the class will behave exactly the same as it would if it was instantiated normally. The major difference is that we can apply partial mocking and substitute one or more methods with mocks. As a general rule, we tend to use spies mostly on classes that we're working on. We want to retain all the functionality of a class that we're writing specifications for, but with an additional option to, when needed, mock a part of it.

Now let us write the specification itself. It could be the following:

```
@Test
public void whenSaveMoveThenInvokeMongoCollectionSave() {
  TicTacToeBean bean = new TicTacToeBean(3, 2, 1, 'Y');
  MongoCollection mongoCollection = mock(MongoCollection.class);
  doReturn(mongoCollection).when(collection).getMongoCollection();
  collection.saveMove(bean);

  verify(mongoCollection, times(1)).save(bean);
}
```

Static methods, such as `mock`, `doReturn`, and `verify`, are all from the `org.mockito.Mockito` class.

First, we're creating a new `TicTacToeBean`. There's nothing special there. Next, we are creating a `mock` object out of the `MongoCollection`. Since we already established that, when working on a unit level, we want to avoid direct communication with the DB, mocking this dependency will provide this for us. It will convert a real class into a mocked one. For the class using `mongoCollection`, it'll look like a real one; however, behind the scenes, all its methods are shallow and do not actually do anything. It's like overwriting that class and replacing all the methods with empty ones:

```
MongoCollection mongoCollection = mock(MongoCollection.class);
```

Next, we're telling that a mocked `mongoCollection` should be returned whenever we call the `getMongoCollection` method of the collection spied class. In other words, we're telling our class to use a fake collection instead of the real one:

```
doReturn(mongoCollection).when(collection).getMongoCollection();
```

Then, we're calling the method that we are working on:

```
collection.saveMove(bean);
```

Finally, we should verify that the correct invocation of the `Jongo` library is performed once:

```
verify(mongoCollection, times(1)).save(bean);
```

Let's try to implement this specification.

Implementation

To better understand the specification we just wrote, let us do only a partial implementation. We'll create an empty method, `saveMove`. This will allow our code to compile without implementing the specification yet:

```
public void saveMove(TicTacToeBean bean) {
}
```

When we run our specifications (`gradle test`), the result is the following:

```
Wanted but not invoked:
mongoCollection.save(Turn: 3; X: 2; Y: 1; Player: Y);
```

Mockito tells us that, according to our specification, we expect the `mongoCollection.save` method to be invoked, and that the expectation was not fulfilled. Since the test is still failing, we need to go back and finish the implementation. One of the biggest sins in TDD is to have a failing test and move onto something else.

> All tests should pass before a new test is written. The benefits of this are that the focus is maintained on a small unit of work, and implementation code is (almost) always in a working condition.

It is sometimes tempting to write multiple tests before the actual implementation. In other cases, developers ignore problems detected by the existing tests and move towards new features. This should be avoided whenever possible. In most cases, breaking this rule will only introduce technical debt that will need to be paid with interest. One of the goals of TDD is ensuring that the implementation code is (almost) always working as expected. Some projects, due to pressures to reach the delivery date or maintain the budget, break this rule and dedicate time to new features, leaving the fixing of the code associated with failed tests for later. Those projects usually end up postponing the inevitable.

Let's modify the implementation too, for example, the following:

```
public void saveMove(TicTacToeBean bean) {
  getMongoCollection().save(null);
}
```

If we run our specifications again, the result is the following:

```
Argument(s) are different! Wanted:
mongoCollection.save(Turn: 3; X: 2; Y: 1; Player: Y);
```

This time we are invoking the expected method, but the arguments we are passing to it are not what we hoped for. In the specification, we set the expectation to a bean (new `TicTacToeBean(3, 2, 1, 'Y')`) and in the implementation, we passed null. Not only that, Mockito verifications can tell us whether a correct method was invoked, and also whether the arguments passed to that method are correct.

The correct implementation of the specification is the following:

```
public void saveMove(TicTacToeBean bean) {
  getMongoCollection().save(bean);
}
```

This time all specifications should pass, and we can, happily, proceed to the next one.

Specification – adding operation feedback

Let us change the return type of our `saveMove` method to `boolean`:

```
@Test
public void whenSaveMoveThenReturnTrue() {
  TicTacToeBean bean = new TicTacToeBean(3, 2, 1, 'Y');
  MongoCollection mongoCollection = mock(MongoCollection.class);
  doReturn(mongoCollection).when(collection).getMongoCollection();
  assertTrue(collection.saveMove(bean));
}
```

Implementation

This implementation is very straightforward. We should change the method return type. Remember that one of the rules of TDD is to use the simplest possible solution. The simplest solution is to return `true` as in the following example:

```
public boolean saveMove(TicTacToeBean bean) {
  getMongoCollection().save(bean);
  return true;
}
```

Refactoring

You have probably noticed that the last two specifications have the first two lines duplicated. We can refactor the specifications code by moving them to the method annotated with `@Before`:

```
TicTacToeCollection collection;
TicTacToeBean bean;
MongoCollection mongoCollection;

@Before
public void before() throws UnknownHostException {
  collection = spy(new TicTacToeCollection());
  bean = new TicTacToeBean(3, 2, 1, 'Y');
  mongoCollection = mock(MongoCollection.class);
}
...
@Test
public void whenSaveMoveThenInvokeMongoCollectionSave() {
// TicTacToeBean bean = new TicTacToeBean(3, 2, 1, 'Y');
// MongoCollection mongoCollection = mock(MongoCollection.class);
```

```
  doReturn(mongoCollection).when(collection).getMongoCollection();
  collection.saveMove(bean);
  verify(mongoCollection, times(1)).save(bean);
}

@Test
public void whenSaveMoveThenReturnTrue() {
// TicTacToeBean bean = new TicTacToeBean(3, 2, 1, 'Y');
// MongoCollection mongoCollection = mock(MongoCollection.class);
   doReturn(mongoCollection).when(collection).getMongoCollection();
   assertTrue(collection.saveMove(bean));
}
```

Specification – error handling

Now let us contemplate the option that something might go wrong when using MongoDB. When, for example, an exception is thrown, we might want to return `false` from our `saveMove` method:

```
@Test
public void givenExceptionWhenSaveMoveThenReturnFalse() {
  doThrow(new MongoException("Bla"))
    .when(mongoCollection).save(any(TicTacToeBean.class));
  doReturn(mongoCollection).when(collection).getMongoCollection();
  assertFalse(collection.saveMove(bean));
}
```

Here, we introduce to another Mockito method: `doThrow`. It acts in a similar way to `doReturn` and throws an `Exception` when set conditions are fulfilled. The specification will throw the `MongoException` when the save method inside the `mongoCollection` class is invoked. This allows us to `assert` that our `saveMove` method returns `false` when an exception is thrown.

Implementation

The implementation can be as simple as adding a `try/catch` block:

```
public boolean saveMove(TicTacToeBean bean) {
  try {
    getMongoCollection().save(bean);
    return true;
  } catch (Exception e) {
    return false;
  }
}
```

Specification – clear state between games

This is a very simple application that, at least at this moment, can store only one game session. Whenever a new instance is created, we should start over and remove all data stored in the database. The easiest way to do this is to simply drop the MongoDB collection. Jongo has the `MongoCollection.drop()` method that can be used for that. We'll create a new method, `drop`, that will act in a similar way to `saveMove`.

If you haven't worked with Mockito, MongoDB, and/or Jongo, the chances are you were not able to do the exercises from this chapter by yourself, and just decided to follow the solutions we provided. If that's the case, this is the moment when you may want to switch gears and try to write the specifications and implementation by yourself.

We should verify that `MongoCollection.drop()` is invoked from our own method `drop()` inside the `TicTacToeCollection` class. Try it by yourself before looking at the following code. It should be almost the same as what we did with the `save` method:

```
@Test
public void whenDropThenInvokeMongoCollectionDrop() {
  doReturn(mongoCollection).when(collection).getMongoCollection();
  collection.drop();
  verify(mongoCollection).drop();
}
```

Implementation

Since this is a wrapper method, implementing this specification should be fairly easy:

```
public void drop() {
  getMongoCollection().drop();
}
```

Specification – drop operation feedback

We're almost done with this class. There are only two specifications left.

Let us make sure that, in normal circumstances, we return `true`:

```
@Test
public void whenDropThenReturnTrue() {
  doReturn(mongoCollection).when(collection).getMongoCollection();
  assertTrue(collection.drop());
}
```

Implementation

If things look too easy with TDD, then that is on purpose. We are splitting tasks into such small entities that, in most cases, implementing a specification is a piece of cake. This one is no exception:

```
public boolean drop() {
  getMongoCollection().drop();
  return true;
}
```

Specification – error handling

Finally, let us make sure that the `drop` method returns `false` in case of an `Exception`:

```
@Test
public void givenExceptionWhenDropThenReturnFalse() {
  doThrow(new MongoException("Bla")).when(mongoCollection).drop();
  doReturn(mongoCollection).when(collection).getMongoCollection();
  assertFalse(collection.drop());
}
```

Implementation

Let us just add a `try/catch` block:

```
public boolean drop() {
  try {
    getMongoCollection().drop();
    return true;
  } catch (Exception e) {
    return false;
  }
}
```

With this implementation, we are finished with the `TicTacToeCollection` class that acts as a layer between our `main` class and MongoDB.

The source code can be found in the `02-save-move` branch of the `tdd-java-ch06-tic-tac-toe-mongo` Git repository (`https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo/branch/02-save-move`). The classes in particular are `TicTacToeCollectionSpec` and `TicTacToeCollection`.

Requirement 2 – store every turn

Let us employ the `TicTacToeCollection` methods inside our main class `TicTacToe`. Whenever a player plays a turn successfully, we should save it to the DB. Also, we should drop the collection whenever a new class is instantiated so that a new game does not overlap the old one. We could make it much more elaborate than this; however, for the purpose of this chapter and learning how to use mocking, this requirement should do for now.

Save each turn to the database and make sure that a new session cleans the old data.

Let's do some setup first.

Specification – creating new collection

Since all our methods that will be used to communicate with MongoDB are in the `TicTacToeCollection` class, we should make sure that it is instantiated. The specification could be the following:

```
@Test
public void whenInstantiatedThenSetCollection() {
  assertNotNull(ticTacToe.getTicTacToeCollection());
}
```

The instantiation of `TicTacToe` is already done in the method annotated with `@Before`. With this specification, we're making sure that the collection is instantiated as well.

Implementation

There is nothing special about this implementation. We should simply overwrite the default constructor and assign a new instance to the `ticTacToeCollection` variable.

To begin with, we should add a local variable and a getter for `TicTacToeCollection`:

```
private TicTacToeCollection ticTacToeCollection;

protected TicTacToeCollection getTicTacToeCollection() {
  return ticTacToeCollection;
}
```

Now all that's left is to instantiate a new `collection` and assign it to the variable when the `main` class is instantiated:

```
public TicTacToe() throws UnknownHostException {
  this(new TicTacToeCollection());
}
protected TicTacToe(TicTacToeCollection collection) {
  ticTacToeCollection = collection;
}
```

We also created another way to instantiate the class by passing `TicTacToeCollection` as an argument. This will come in handy inside specifications as an easy way to pass a mocked collection.

Now let us go back to the specifications class and make use of this new constructor.

Specification refactoring

To utilize a newly created `TicTacToe` constructor, we can do something such as the following:

```
private TicTacToeCollection collection;

@Before
public final void before() throws UnknownHostException {
  collection = mock(TicTacToeCollection.class);
// ticTacToe = new TicTacToe();
  ticTacToe = new TicTacToe(collection);
}
```

Now all our specifications will use a mocked version of the `TicTacToeCollection`. There are other ways to inject mocked dependencies (for example, with Spring); however, when possible, we feel that simplicity trumps complicated frameworks.

Specification – storing current move

Whenever we play a turn, it should be saved to the DB. The specification can be the following:

```
@Test
public void whenPlayThenSaveMoveIsInvoked() {
  TicTacToeBean move = new TicTacToeBean(1, 1, 3, 'X');
  ticTacToe.play(move.getX(), move.getY());
  verify(collection).saveMove(move);
}
```

By now, you should be familiar with Mockito, but let us go through the code as a refresher:

1. First, we are instantiating a `TicTacToeBean` since it contains the data that our collections expect:

   ```
   TicTacToeBean move = new TicTacToeBean(1, 1, 3, 'X');
   ```

2. Next, it is time to play an actual turn:

   ```
   ticTacToe.play(move.getX(), move.getY());
   ```

3. Finally, we need to verify that the `saveMove` method is really invoked:

   ```
   verify(collection, times(1)).saveMove(move);
   ```

As we have done throughout this chapter, we isolated all external invocations and focused only on the unit (`play`) that we're working on. Keep in mind that this isolation is limited only to the public and protected methods. When it comes to the actual implementation, we might choose to add the `saveMove` invocation to the `play` public method or one of the private methods that we wrote as a result of the refactoring we did earlier.

Implementation

This specification poses a couple of challenges. First, where should we place the invocation of the `saveMove` method? The `setBox` private method looks like a good place. That's where we are doing validations of if the turn is valid, and if it is, we can call the `saveMove` method. However, that method expects a `bean` instead of the variables `x`, `y`, and `lastPlayer` that are being used right now, so we might want to change the signature of the `setBox` method.

This is how the method looks now:

```
private void setBox(int x, int y, char lastPlayer) {
  if (board[x - 1][y - 1] != '\0') {
    throw new RuntimeException("Box is occupied");
  } else {
    board[x - 1][y - 1] = lastPlayer;
  }
}
```

This is how it looks after the necessary changes are applied:

```
private void setBox(TicTacToeBean bean) {
  if (board[bean.getX() - 1][bean.getY() - 1] != '\0') {
    throw new RuntimeException("Box is occupied");
  } else {
    board[bean.getX() - 1][bean.getY() - 1] = lastPlayer;
    getTicTacToeCollection().saveMove(bean);
  }
}
```

The change of the `setBox` signature triggers a few other changes. Since it is invoked from the `play` method, we'll need to instantiate the `bean` there:

```
public String play(int x, int y) {
  checkAxis(x);
  checkAxis(y);
  lastPlayer = nextPlayer();
// setBox(x, y, lastPlayer);
```

```
    setBox(new TicTacToeBean(1, x, y, lastPlayer));
    if (isWin(x, y)) {
      return lastPlayer + " is the winner";
    } else if (isDraw()) {
      return RESULT_DRAW;
    } else {
      return NO_WINNER;
    }
  }
```

You might have noticed that we used a constant value `1` as a turn. There is still no specification that says otherwise, so we took a shortcut. We'll deal with it later.

All those changes were still very simple, and it took a reasonably short period of time to implement them. If the changes were bigger, we might have chosen a different path; and made a simpler change to get to the final solution through refactoring. Remember that speed is the key. You don't want to get stuck with an implementation that does not pass tests for a long time.

Specification – error handling

What happens if a move could not be saved? Our helper method `saveMove` returns `true` or `false` depending on the MongoDB operation outcome. We might want to throw an exception when it returns `false`.

First things first: we should change the implementation of the `before` method and make sure that, by default, `saveMove` returns `true`:

```
@Before
public final void before() throws UnknownHostException {
  collection = mock(TicTacToeCollection.class);
doReturn(true).when(collection).saveMove(any(TicTacToeBean.class));
  ticTacToe = new TicTacToe(collection);
}
```

Now that we have stubbed the mocked collection with what we think is the default behavior (return `true` when `saveMove` is invoked), we can proceed and write the specification:

```
@Test
public void whenPlayAndSaveReturnsFalseThenThrowException() {
doReturn(false).when(collection).saveMove(any(TicTacToeBean.class))
;
  TicTacToeBean move = new TicTacToeBean(1, 1, 3, 'X');
```

```
    exception.expect(RuntimeException.class);
    ticTacToe.play(move.getX(), move.getY());
  }
```

We're using Mockito to return `false` when `saveMove` is invoked. Since, in this case, we don't care about a specific invocation of `saveMove`, we used `any(TicTacToeBean.class)` as the method argument. This is another one of Mockito's static methods.

Once everything is set, we use a JUnit expectation in the same way as we did before throughout `Chapter 3`, *Red-Green-Refactor – From Failure Through Success until Perfection*.

Implementation

Let's do a simple `if` and throw a `RuntimeException` when the result is not expected:

```
private void setBox(TicTacToeBean bean) {
  if (board[bean.getX() - 1][bean.getY() - 1] != '\0') {
    throw new RuntimeException("Box is occupied");
  } else {
    board[bean.getX() - 1][bean.getY() - 1] = lastPlayer;
//  getTicTacToeCollection().saveMove(bean);
    if (!getTicTacToeCollection().saveMove(bean)) {
      throw new RuntimeException("Saving to DB failed");
    }
  }
}
```

Specification – alternate players

Do you remember the turn that we hard coded to be always `1`? Let's fix that behavior.

We can invoke the `play` method twice and verify that the turn changes from `1` to `2`:

```
@Test
public void whenPlayInvokedMultipleTimesThenTurnIncreases() {
  TicTacToeBean move1 = new TicTacToeBean(1, 1, 1, 'X');
  ticTacToe.play(move1.getX(), move1.getY());
  verify(collection, times(1)).saveMove(move1);
  TicTacToeBean move2 = new TicTacToeBean(2, 1, 2, 'O');
  ticTacToe.play(move2.getX(), move2.getY());
  verify(collection, times(1)).saveMove(move2);
}
```

Implementation

As with almost everything else done in the TDD fashion, implementation is fairly easy:

```
private int turn = 0;
...
public String play(int x, int y) {
  checkAxis(x);
  checkAxis(y);
  lastPlayer = nextPlayer();
  setBox(new TicTacToeBean(++turn, x, y, lastPlayer));
  if (isWin(x, y)) {
    return lastPlayer + " is the winner";
  } else if (isDraw()) {
    return RESULT_DRAW;
  } else {
    return NO_WINNER;
  }
}
```

Exercises

A few more specifications and their implementations are still missing. We should invoke the `drop()` method whenever our `TicTacToe` class is instantiated. We should also make sure that `RuntimeException` is thrown when `drop()` returns `false`. We'll leave those specifications and their implementations as an exercise for you.

The source code can be found in the `03-mongo` branch of the `tdd-java-ch06-tic-tac-toe-mongo` Git repository (`https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo/branch/03-mongo`). The classes in particular are `TicTacToeSpec` and `TicTacToe`.

Integration tests

We did a lot of unit tests. We relied a lot on trust. Unit after unit was specified and implemented. While working on specifications, we isolated everything but the units we were working on, and verified that one invoked the other correctly. However, the time has come to validate that all those units are truly able to communicate with MongoDB. We might have made a mistake or, more importantly, we might not have MongoDB up and running. It would be a disaster to discover that, for example, we deployed our application, but forgot to bring up the DB, or that the configuration (IP, port, and so on) is not set correctly.

The integration test's objective is to validate, as you might have guessed, the integration of separate components, applications, systems, and so on. If you remember the testing pyramid, it states that unit tests are the easiest to write and fastest to run, so we should keep other types of tests limited to things that UTs did not cover.

We should isolate our integration tests in a way that they can be run occasionally (before we push our code to repository, or as a part of our **continuous integration** (**CI**) process) and keep unit test as a continuous feedback loop.

Tests separation

If we follow some kind of convention, it is fairly easy to separate tests in Gradle. We can have our tests in different directories and distinct packages or, for example, with different file suffixes. In this case, we choose the latter. All our specification classes are named with the `Spec` suffix (that is, `TicTacToeSpec`). We can make a rule that all integration tests have the `Integ` suffix.

With that in mind, let us modify our `build.gradle` file.

First, we'll tell Gradle that only classes ending with `Spec` should be used by the `test` task:

```
test {
    include '**/*Spec.class'
}
```

Next, we can create a new task, `testInteg`:

```
task testInteg(type: Test) {
    include '**/*Integ.class'
}
```

With those two additions to `build.gradle`, we continue having the test tasks that we used heavily throughout the book; however, this time, they are limited only to specifications (unit tests). In addition, all integration tests can be run by clicking the `testInteg` task from the Gradle projects IDEA window or running the following command from command prompt:

```
gradle testInteg
```

Let us write a simple integration test.

The integration test

We'll create a `TicTacToeInteg` class inside the `com.packtpublishing.tddjava.ch03tictactoe` package in the `src/test/java` directory. Since we know that Jongo throws an exception if it cannot connect to the database, a test class can be as simple as the following:

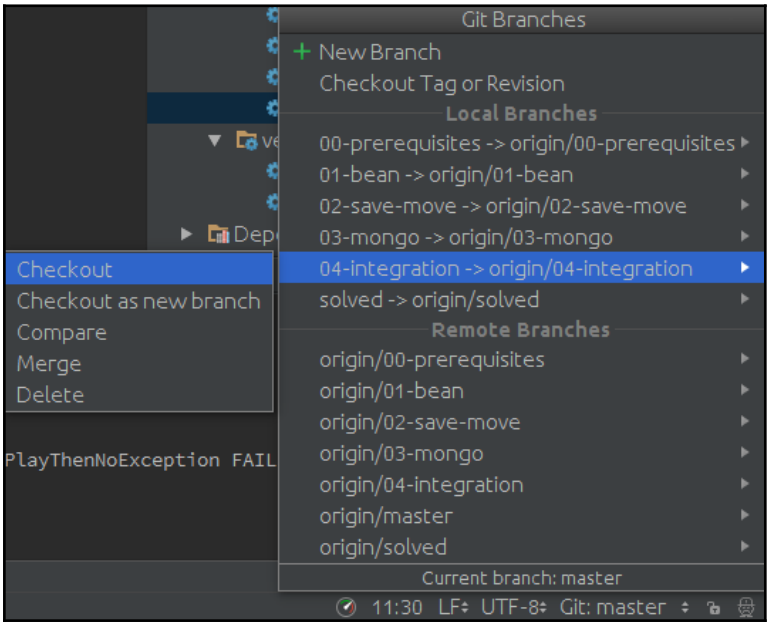
```
import org.junit.Test;
import java.net.UnknownHostException;
import static org.junit.Assert.*;

public class TicTacToeInteg {

  @Test
  public void givenMongoDbIsRunningWhenPlayThenNoException()
        throws UnknownHostException {
    TicTacToe ticTacToe = new TicTacToe();
    assertEquals(TicTacToe.NO_WINNER, ticTacToe.play(1, 1));
  }
}
```

The invocation of `assertEquals` is just as a precaution. The real objective of this test is to make sure that no `Exception` is thrown. Since we did not start MongoDB (unless you are very proactive and did it yourself, in which case you should stop it), `test` should fail:

```
vfarcic@viktor: ~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo

vfarcic@viktor:~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo$ gradle testInteg
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:testInteg

com.packtpublishing.tddjava.ch03tictactoe.TicTacToeInteg > givenMongoDbIsRunning
WhenPlayThenNoException                                                      F
AILED
    java.lang.RuntimeException at TicTacToeInteg.java:12

1 test completed, 1 failed
:testInteg FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':testInteg'.
> There were failing tests. See the report at: file:///home/vfarcic/IdeaProjects
/tdd-java-ch06-tic-tac-toe-mongo/build/reports/tests/index.html

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug
option to get more log output.

BUILD FAILED

Total time: 14.6 secs
vfarcic@viktor:~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo$
```

Now that we know that the integration test works, or in other words, that it indeed fails when MongoDB is not up and running, let us try it again with the DB started. To bring up MongoDB, we'll use Vagrant to create a virtual machine with Ubuntu OS. MongoDB will be run as a Docker.

Make sure that the **04-integration** branch is checked out:



From the command prompt, run the following command:

```
$ vagrant up
```

Be patient until VM is up and running (it might take a while when executed for the first time, especially on a slower bandwidth). Once finished, rerun integration tests:

```
vfarcic@viktor: ~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo
vfarcic@viktor:~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo$ gradle testInteg
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:testInteg

BUILD SUCCESSFUL

Total time: 4.46 secs
vfarcic@viktor:~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo$
```

It worked, and now we're confident that we are indeed integrated with MongoDB.

This was a very simplistic integration test, and in the real-world, we would do a bit more than this single test. We could, for example, query the DB and confirm that data was stored correctly. However, the purpose of this chapter was to learn both how to mock and that we should not depend only on unit tests. The next chapter will explore integration and functional tests in more depth.

The source code can be found in the `04-integration` branch of the `tdd-java-ch06-tic-tac-toe-mongo` Git repository (`https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo/branch/04-integration`).

Summary

Mocking and spying techniques are used to isolate different parts of code or third-party libraries. They are essential if we are to proceed with great speed, not only while coding, but also while running tests. Tests without mocks are often too complex to write and can be so slow that, with time, TDD tends to become close to impossible. Slow tests mean that we won't be able to run all of them every time we write a new specification. That in itself leads to deterioration in the confidence we have in the our tests, since only a part of them is run.

Mocking is not only useful as a way to isolate external dependencies, but also as a way to isolate our own code from a unit we're working on.

In this chapter, we presented Mockito as, in our opinion, the framework with the best balance between functionality and ease of use. We invite you to investigate its documentation in more detail (`http://mockito.org/`), as well as other Java frameworks dedicated to mocking. EasyMock (`http://easymock.org/`), JMock (`http://www.jmock.org/`), and PowerMock (`https://code.google.com/p/powermock/`) are a few of the most popular.

In the next chapter we are going to put some functional programming concepts as well as some TDD concepts applied to them. For that matter, part of the Java functional API is going to be presented.