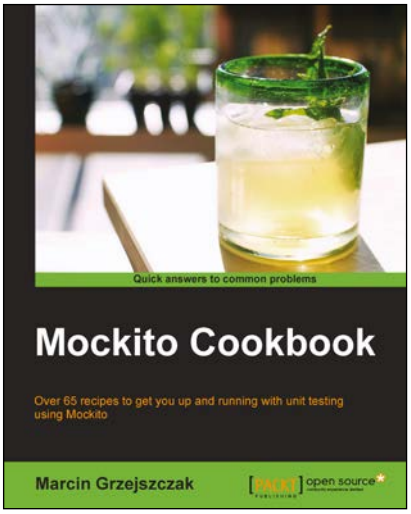
[PACKT] open source
community experience distilled
PUBLISHING

Mockito Cookbook

Marcin Grzejszczak



Chapter No. 1
"Getting Started with Mockito"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "Getting Started with Mockito"

A synopsis of the book's content

Information on where to buy this book

About the Author

Marcin Grzejszczak is an experienced Java programmer. He is enthusiastic about clean coding and good design. He has contributed to several open source projects (Drools, Moco, Mockito, Spock, and so on) and to Groovy core. He is the co-organizer of the Warsaw Groovy User Group. He is a member of the Most Valuable Blogger program at DZone and Java Code Geeks.

Marcin is the author of *Instant Mockito, Packt Publishing*, and *Drools Refcard* at DZone. You can visit his blog, `http://toomuchcoding.blogspot.com`, or his home page, `http://www.marcin.grzejszczak.pl`. Or, you can follow him on Twitter at `http://twitter.com/MGrzejszczak`.

I would like to thank my beloved Marta for showing extreme support, understanding, and encouragement during the creation of this book. I would also like to thank Tomasz Kaczanowski for the indispensable guidelines that allowed me to put the book on the right track. All the discussions with Brice Dutheil helped to deepen my understanding of the philosophy behind Mockito and testing as such. I would like to thank Jakub Nabrdalik, Maciej Zieliński, Kamil Trepczyński, and Michal Pasiński for the brainstorming sessions and reviews. Last but not least, I would like to express my gratitude to all of the official reviewers who helped to increase the quality of this book.

Mockito Cookbook

According to Google Trends, Mockito, compared to its main Java mocking framework competitors, EasyMock and jMock, has been the most widely used since 2011 and this trend has been upward ever since. Given its extremely simple and elegant API, Mockito gives you the possibility to test your application in a readable manner. Furthermore, it's syntax is so intuitive that you'll learn it in no time at all.

The very concept behind this book is to give the reader the possibility to use Mockito in order to write beautiful and comprehensive tests. The Mockito documentation as such is of very high quality, so you should always, regardless of the tool you are using, refer to it when in doubt. This book is an extension to this documentation since it covers its content but puts it in a real-life example. Where the Mockito documentation proves that the library, as such, is doing what it is supposed to do, you can come to a point where you don't actually know how to use it versus your production code. Worry not! *Mockito Cookbook* comes to the rescue. This book contains solutions to more than 60 problems that you may encounter throughout your Mockito testing endeavor. You will learn how to write tests that become the living documentation of your code. You will become A Mockito expert. (Since the book also explains some Mockito internals you might even be tempted to become its contributor!) And hopefully, your tests will become an example to be followed by your colleagues.

What This Book Covers

Chapter 1, *Getting Started with Mockito*, covers the Mockito configuration for JUnit and TestNG and some of its experimental features.

Chapter 2, *Creating Mocks*, presents numerous ways to create mocks.

Chapter 3, *Creating Spies and Partial Mocks*, covers the process of instantiating spy objects and partial mocks.

Chapter 4, *Stubbing Behavior of Mocks*, shows how to stub the method executions of mock objects.

Chapter 5, *Stubbing Behavior of Spies*, presents ways to stub the method executions of spies.

Chapter 6, *Verifying Test Doubles*, covers the process of behavior verification of test doubles.

Chapter 7, *Verifying Behavior with Object Matchers*, shows how to confirm that your application works as it should using Hamcrest or AssertJ.

Chapter 8, Refactoring with Mockito, covers the process of easily refactoring your production and test code, thanks to Mockito.

Chapter 9, Integration Testing with Mockito and DI Frameworks, presents ways to inject mocks into your Spring– or Guice–based applications.

Chapter 10, Mocking Libraries Comparison, shows the differences and similarities between several mocking libraries and Mockito.

1
Getting Started with Mockito

In this chapter, we will cover the following recipes:

- Adding Mockito to a project's classpath
- Getting started with Mockito for JUnit
- Getting started with Mockito for TestNG
- Mockito best practices - test behavior, not implementation
- Adding Mockito hints to exception messages in JUnit (Experimental)
- Adding additional Mockito warnings to your tests in JUnit (Experimental)

Introduction

For those who don't know Mockito at all, I'd like to write a really short introduction about it.

Mockito is an open source framework for Java that allows you to easily create test doubles (mocks). What makes Mockito so special is that it eliminates the common expect-run-verify pattern (which was present, for example, in EasyMock—please refer to `http://monkeyisland.pl/2008/02/24/can-i-test-what-i-want-please` for more details) that in effect leads to a lower coupling of the test code to the production code as such. In other words, one does not have to define the expectations of how the mock should behave in order to verify its behavior. That way, the code is clearer and more readable for the user.

On one hand, Mockito has a very active group of contributors and is actively maintained; on the other hand, unfortunately, by the time this book is written, the last Mockito release (Version 1.9.5) have been in October 2012.

You may ask yourself the question, "Why should I even bother to use Mockito in the first place?" Out of many choices, Mockito offers the following key features:

- There is no expectation phase for Mockito—you can either stub or verify the mock's behavior
- You are able to mock both interfaces and classes
- You can produce little boilerplate code while working with Mockito by means of annotations
- You can easily verify or stub with intuitive argument matchers

Before diving into Mockito as such, one has to understand the concept behind **System Under Test** (**SUT**) and test doubles. We will base our work on what Gerard Meszaros has defined in the xUnit Patterns (`http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html`).

SUT (`http://xunitpatterns.com/SUT.html`) describes the system that we are testing. It doesn't have to necessarily signify a class or any part of the application that we are testing or even the whole application as such.

As for test doubles (`http://www.martinfowler.com/bliki/TestDouble.html`), it's an object that is used only for testing purposes, instead of a real object. Let's take a look at different types of test doubles:

- **Dummy**: This is an object that is used only for the code to compile—it doesn't have any business logic (for example, an object passed as a parameter to a method)
- **Fake**: This is an object that has an implementation but it's not production ready (for example, using an in-memory database instead of communicating with a standalone one)
- **Stub**: This is an object that has predefined answers to method executions made during the test
- **Mock**: This is an object that has predefined answers to method executions made during the test and has recorded expectations of these executions
- **Spy**: These are objects that are similar to stubs, but they additionally record how they were executed (for example, a service that holds a record of the number of sent messages)

An additional remark is also related to testing the output of our application. Throughout the book, you will see that the tests (in general, all of them apart from the chapter related to verification) are based on the assertion of behavior instead of the checking of implementation. The more decoupled your test code is from your production code, the better, since you will have to spend less time (or even none) on modifying your tests after you change the implementation of the code.

Coming back to the chapter's content—this chapter is all about getting started with Mockito. We will begin with how to add Mockito to your classpath. Then, we'll see a simple setup of tests for both JUnit and TestNG test frameworks. Next, we will check why it is crucial to assert the behavior of the system under test instead of verifying its implementation details. Finally, we will check out some of Mockito's experimental features, adding hints and warnings to the exception messages. The very idea of the following recipes is to prepare your test classes to work with Mockito and to show you how to do this with as little boilerplate code as possible.

Due to my fondness for the behavior driven development (`http://dannorth.net/introducing-bdd/` first introduced by Dan North), I'm using Mockito's `BDDMockito` and AssertJ's `BDDAssertions` static methods to make the code even more readable and intuitive in all the test cases. Also, please read Szczepan Faber's blog (author of Mockito) about the given, when, then separation in your test methods—`http://monkeyisland.pl/2009/12/07/given-when-then-forever/`—since these are omnipresent throughout the book.

Even though some of the previous methods might sound not too clear to you or the test code looks complicated—don't worry, it will all be explained throughout the book. I don't want the book to become a duplication of the Mockito documentation, which is of high quality—I would like you to take a look at good tests and get acquainted with Mockito syntax from the beginning. What's more, I've used static imports in the code to make it even more readable, so if you get confused with any of the pieces of code, it would be best to consult the repository and the code as such.

Adding Mockito to a project's classpath

Adding Mockito to a project's classpath is as simple as adding one of the two jars to your project's classpath:

- `mockito-all`: This is a single jar with all dependencies (with the `hamcrest` and `objenesis` libraries—as of June 2011).
- `mockito-core`: This is only Mockito core (without `hamcrest` or `objenesis`). Use this if you want to control which version of `hamcrest` or `objenesis` is used.

How to do it...

If you are using a dependency manager that connects to the Maven Central Repository, then you can get your dependencies as follows (examples of how to add `mockito-all` to your classpath for Maven and Gradle):

For Maven, use the following code:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
    <scope>test</scope>
</dependency>
```

For Gradle, use the following code:

```
testCompile "org.mockito:mockito-all:1.9.5"
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

If you are not using any of the dependency managers, you have to either download `mockito-all.jar` or `mockito-core.jar` and add it to your classpath manually (you can download the jars from `https://code.google.com/p/mockito/downloads/list`). To see more examples of adding Mockito to your classpath, please check the book, *Instant Mockito*, *Marcin Grzejszczak*, *Packt Publishing*, for more examples of adding Mockito to your classpath (it includes Ant, Buildr, Sbt, Ivy, Gradle, and Maven).

See also

- Refer to *Instant Mockito*, *Marcin Grzejszczak*, *Packt Publishing* for an introduction to Mockito together with examples of Mockito configuration in several build tools at `http://www.packtpub.com/how-to-create-stubs-mocks-spies-using-mockito/book`

Getting started with Mockito for JUnit

Before going into details regarding Mockito and JUnit integration, it is worth mentioning a few words about JUnit.

JUnit is a testing framework (an implementation of the xUnit framework) that allows you to create repeatable tests in a very readable manner. In fact, JUnit is a port of Smalltalk's SUnit (both the frameworks were originally implemented by Kent Beck). What is important in terms of JUnit and Mockito integration is that under the hood, JUnit uses a test runner to run its tests (from xUnit—test runner is a program that executes the test logic and reports the test results).

Mockito has its own test runner implementation that allows you to reduce boilerplate in order to create test doubles (mocks and spies) and to inject them (either via constructors, setters, or reflection) into the defined object. What's more, you can easily create argument captors. All of this is feasible by means of proper annotations as follows:

- `@Mock`: This is used for mock creation
- `@Spy`: This is used to create a spy instance
- `@InjectMocks`: This is used to instantiate the `@InjectMock` annotated field and inject all the `@Mock` or `@Spy` annotated fields into it (if applicable)
- `@Captor`: This is used to create an argument captor

By default, you should profit from Mockito's annotations to make your code look neat and to reduce the boilerplate code in your application.

Getting ready

In order to add JUnit to your classpath, if you are using a dependency manager that connects to the Maven Central Repository, then you can get your dependencies as follows (examples for Maven and Gradle):

To add JUnit in Maven, use the following code:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
```

To add JUnit in Gradle, use the following code:

```
testCompile('junit:junit:4.11')
```

If you are not using any of the dependency managers, you have to download the following jars:

- `junit.jar`
- `hamcrest-core.jar`

Add the downloaded files to your classpath manually (you can download the jars from `https://github.com/junit-team/junit/wiki/Download-and-Install`).

For this recipe, our system under test will be a `MeanTaxFactorCalculator` class that will call an external service, `TaxService`, to get the current tax factor for the current user. It's a tax factor and not tax as such, since for simplicity, we will not be using `BigDecimals` but `doubles`, and I'd never suggest using `doubles` to anything related to money, as follows:

```
public class MeanTaxFactorCalculator {

    private final TaxService taxService;

    public MeanTaxFactorCalculator(TaxService taxService) {
        this.taxService = taxService;
    }

    public double calculateMeanTaxFactorFor(Person person) {
        double currentTaxFactor = taxService.
getCurrentTaxFactorFor(person);
        double anotherTaxFactor = taxService.
getCurrentTaxFactorFor(person);
        return (currentTaxFactor + anotherTaxFactor) / 2;
    }

}
```

How to do it...

To use Mockito's annotations, you have to perform the following steps:

1. Annotate your test with the `@RunWith(MockitoJUnitRunner.class)`.
2. Annotate the test fields with the `@Mock` or `@Spy` annotation to have either a mock or spy object instantiated.

3. Annotate the test fields with the `@InjectMocks` annotation to first instantiate the `@InjectMock` annotated field and then inject all the `@Mock` or `@Spy` annotated fields into it (if applicable).
4. Annotate the test fields with the `@Captor` annotation to make Mockito instantiate an argument captor (refer to *Chapter 6*, *Verifying Test Doubles*, for more details).

The following snippet shows the JUnit and Mockito integration in a test class that verifies the SUT's behavior (remember that I'm using `BDDMockito.given(...)` and AssertJ's `BDDAssertions.then(...)` static methods; refer to *Chapter 7*, *Verifying Behavior with Object Matchers*, for how to work with AssertJ or how to do the same with Hamcrest's `assertThat(...)` method):

```
@RunWith(MockitoJUnitRunner.class)
public class MeanTaxFactorCalculatorTest {

    static final double TAX_FACTOR = 10;

    @Mock TaxService taxService;

    @InjectMocks MeanTaxFactorCalculator systemUnderTest;

    @Test
    public void should_calculate_mean_tax_factor() {
        // given
        given(taxService.getCurrentTaxFactorFor(any(Person.class)))
.willReturn(TAX_FACTOR);

        // when
        double meanTaxFactor = systemUnderTest.
calculateMeanTaxFactorFor(new Person());

        // then
        then(meanTaxFactor).isEqualTo(TAX_FACTOR);
    }

}
```

To profit from Mockito's annotations using JUnit, you just have to annotate your test class with `@RunWith(MockitoJUnitRunner.class)`.

How it works...

The Mockito test runner will adapt its strategy depending on the version of JUnit. If there exists a `org.junit.runners.BlockJUnit4ClassRunner` class, it means that the codebase is using at least JUnit in Version 4.5.What eventually happens is that the `MockitoAnnotations.initMocks(...)` method is executed for the given test, which initializes all the Mockito annotations (for more information, check the subsequent *There's more...* section).

There's more...

You may have a situation where your test class has already been annotated with a `@RunWith` annotation and, seemingly, you may not profit from Mockito's annotations. In order to achieve this, you have to call the `MockitoAnnotations.initMocks` method manually in the `@Before` annotated method of your test, as shown in the following code:

```
public class MeanTaxFactorCalculatorTest {

    static final double TAX_FACTOR = 10;

    @Mock TaxService taxService;

    @InjectMocks MeanTaxFactorCalculator systemUnderTest;

    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void should_calculate_mean_tax_factor() {
        // given
        given(taxService.getCurrentTaxFactorFor
(Mockito.any(Person.class))).willReturn(TAX_FACTOR);

        // when
        double meanTaxFactor = systemUnderTest.
calculateMeanTaxFactorFor(new Person());

        // then
        then(meanTaxFactor).isEqualTo(TAX_FACTOR);
    }

}
```

To use Mockito's annotations without a JUnit test runner, you have to call the `MockitoAnnotations.initMocks` method and pass the test class as its parameter.

Mockito checks whether the user has overridden the global configuration of `AnnotationEngine` and, if this is not the case, the `InjectingAnnotationEngine` implementation is used to process annotations in tests. What is done internally is that the test class fields are scanned for annotations and proper test doubles are initialized and injected into the `@InjectMocks` annotated object (either by a constructor, property setter, or field injection, in that precise order).

You have to remember several factors related to the automatic injection of test doubles as follows:

- If Mockito is not able to inject test doubles into the `@InjectMocks` annotated fields through either of the strategies, it won't report failure—the test will continue as if nothing happened (and most likely, you will get `NullPointerException`).
- For constructor injection, if arguments cannot be found, then null is passed
- For constructor injection, if nonmockable types are required in the constructor, then the constructor injection won't take place.
- For other injection strategies, if you have properties with the same type (or same erasure) and if Mockito matches mock names with a field/property name, it will inject that mock properly. Otherwise, the injection won't take place.
- For other injection strategies, if the `@InjectMocks` annotated object wasn't previously initialized, then Mockito will instantiate the aforementioned object using a no-arg constructor if applicable.

See also

- JUnit documentation at `https://github.com/junit-team/junit/wiki`
- Martin Fowler's article on xUnit at `http://www.martinfowler.com/bliki/Xunit.html`
- Gerard Meszaros's *xUnit Test Patterns* at `http://xunitpatterns.com/`
- `@InjectMocks` Mockito documentation (with description of injection strategies) at `http://docs.mockito.googlecode.com/hg/1.9.5/org/mockito/InjectMocks.html`

Getting started with Mockito for TestNG

Before going into details regarding Mockito and TestNG integration, it is worth mentioning a few words about TestNG.

TestNG is a unit testing framework for Java that was created, as the author defines it on the tool's website (refer to the *See also* section for the link), out of frustration for some JUnit deficiencies. TestNG was inspired by both JUnit and TestNG and aims at covering the whole scope of testing—from unit, through functional, integration, end-to-end tests, and so on. However, the JUnit library was initially created for unit testing only.

The main differences between JUnit and TestNG are as follows:

- The TestNG author disliked JUnit's approach of having to define some methods as static to be executed before the test class logic gets executed (for example, the `@BeforeClass` annotated methods)—that's why in TestNG you don't have to define these methods as static
- TestNG has more annotations related to method execution before single tests, suites, and test groups
- TestNG annotations are more descriptive in terms of what they do, for example, the JUnit's `@Before` versus TestNG's `@BeforeMethod`

Mockito in Version 1.9.5 doesn't provide any out-of-the-box solution to integrate with TestNG in a simple way, but there is a special Mockito subproject for TestNG (refer to the *See also* section for the URL) that should be part one of the subsequent Mockito releases. In the following recipe, we will take a look at how to profit from that code and that very elegant solution.

Getting ready

When you take a look at Mockito's TestNG subproject on the Mockito GitHub repository, you will find that there are three classes in the `org.mockito.testng` package, as follows:

- `MockitoAfterTestNGMethod`
- `MockitoBeforeTestNGMethod`
- `MockitoTestNGListener`

Unfortunately, until this project eventually gets released, you have to just copy and paste those classes to your codebase.

How to do it...

To integrate TestNG and Mockito, perform the following steps:

1. Copy the `MockitoAfterTestNGMethod`, `MockitoBeforeTestNGMethod`, and `MockitoTestNGListener` classes to your codebase from Mockito's TestNG subproject.
2. Annotate your test class with `@Listeners(MockitoTestNGListener.class)`.
3. Annotate the test fields with the `@Mock` or `@Spy` annotation to have either a mock or spy object instantiated.
4. Annotate the test fields with the `@InjectMocks` annotation to first instantiate the `@InjectMock` annotated field and inject all the `@Mock` or `@Spy` annotated fields into it (if applicable).
5. Annotate the test fields with the `@Captor` annotation to make Mockito instantiate an argument captor (check *Chapter 6*, *Verifying Test Doubles*, for more details).

Now let's take a look at this snippet that, using TestNG, checks whether the mean tax factor value has been calculated properly (remember that I'm using the `BDDMockito.given(...)` and AssertJ's `BDDAssertions.then(...)` static methods—refer to *Chapter 7*, *Verifying Behavior with Object Matchers*, on how to work with Hamcrest `assertThat(...)` method):

```
@Listeners(MockitoTestNGListener.class)
public class MeanTaxFactorCalculatorTestNgTest {

    static final double TAX_FACTOR = 10;

    @Mock TaxService taxService;

    @InjectMocks MeanTaxFactorCalculator systemUnderTest;

    @Test
    public void should_calculate_mean_tax_factor() {
        // given
        given(taxService.getCurrentTaxFactorFor(any(Person.class)))
.willReturn(TAX_FACTOR);

        // when
        double meanTaxFactor = systemUnderTest.
calculateMeanTaxFactorFor(new Person());

        // then
        then(meanTaxFactor).isEqualTo(TAX_FACTOR);
    }

}
```

How it works...

TestNG allows you to register custom listeners (your listener class has to implement the `IInvokedMethodListener` interface). Once you do this, the logic inside the implemented methods will be executed before and after every configuration, and test methods get called. Mockito provides you with a listener whose responsibilities are as follows:

- Initialize mocks annotated with the `@Mock` annotation (it is done only once)
- Validate the usage of Mockito after each test method

Remember that with TestNG all mocks are reset (or initialized if it hasn't already been done) before any TestNG method!

See also

- The TestNG homepage at `http://testng.org/doc/index.html`
- The Mockito TestNG subproject at `https://github.com/mockito/mockito/tree/master/subprojects/testng`
- The *Getting started with Mockito for JUnit* recipe on the `@InjectMocks` analysis

Mockito best practices – test behavior not implementation

Once you start testing with Mockito you might be tempted to start mocking everything that gets in your way. What is more, you may have heard that you have to mock all of the collaborators of the class and then verify whether those test doubles executed the desired methods. Of course, you can code like that, but since it is best to be a pragmatic programmer, you should ask yourself the question whether you would be interested in changing the test code each time someone changes the production code, even though the application does the same things.

It's worth going back to distinguishing stubs from mocks. Remember that, if you create a mock, it's for the sake of the verification of its method execution. If you are only interested in the behavior of your test double—if it behaves as you tell it to—then you have a stub. In the vast majority of cases, you shouldn't be interested in whether your test double has executed a particular method; you should be more interested in whether your application does what it is supposed to do. Also, remember that there are cases where it makes no sense to create a stub of an external dependency—it all depends on how you define the system under test.

It might sound a little confusing but, hopefully, the following recipe will clear things up. We will take a look at the simple example of a tax factor summing class that changes in time (whereas its tests should not change much).

Getting ready

Let's assume that we have the following tax factor calculator that calculates a sum of two tax factors:

```
public class TaxFactorCalculator {

    public double calculateSum
(double taxFactorOne, double taxFactorTwo) {
        return taxFactorOne + taxFactorTwo;
    }

}
```

After some time, it turned out that we read about a library that allows you to hide the implementation details of summing and you decided to rewrite your calculator to use this library. Now your code looks as follows:

```
public class TaxFactorCalculator {

    private final Calculator calculator;

    public TaxFactorCalculator(Calculator calculator) {
        this.calculator = calculator;
    }

    public double calculateSum
(double taxFactorOne, double taxFactorTwo) {
        return calculator.add(taxFactorOne, taxFactorTwo);
    }

}
```

How to do it...

Since you want to test whether your system under test works fine, you should focus on the following points:

- Start by writing a test—not with an implementation. That way, you will constantly ask yourself the question of what you want to do and only then will you think about how to do it.
- Focus on asserting the result—what you want to verify in most cases is whether your system under test works as it is supposed to. You shouldn't care much how exactly it is done.

Let's take a look at a test of the first version of the class (I'm using the `BDDMockito.given(...)` and AssertJ's `BDDAssertions.then(...)` static methods—refer to *Chapter 7*, *Verifying Behavior with Object Matchers*, for how to work with AssertJ or how to do the same with Hamcrest's `assertThat(...)` method):

```
    @Test
    public void should_calculate_sum_of_factors() {
        // given
        TaxFactorCalculator systemUnderTest =
 new TaxFactorCalculator();
        double taxFactorOne = 1;
        double taxFactorTwo = 2;
        double expectedSum = 3;

        // when
        double sumOfFactors = systemUnderTest.
calculateSum(taxFactorOne, taxFactorTwo);

        // then
        then(sumOfFactors).isEqualTo(expectedSum);
    }
```

As you can see, we are testing a class that should add two numbers and produce a result. We are not interested in how it is done—we want to check that the result is satisfactory. Now, assuming that our implementation changed—having a good test would require only to comply to the new way that our system under test is being initialized and the rest of the code remains untouched. In other words, change `TaxFactorCalculator systemUnderTest = new TaxFactorCalculator()` to `TaxFactorCalculator systemUnderTest = new TaxFactorCalculator(new Calculator())`. Moreover, since we are checking behavior and not the implementation, we don't have to refactor the test code at all.

See also

- Martin Fowler on TDD at `http://martinfowler.com/bliki/TestDrivenDevelopment.html`
- Kent Beck's Test Driven Development on Amazon at `http://www.amazon.com/Test-Driven-Development-By-Example/dp/0321146530`
- Mockito's wiki page concerning how to write good tests at `https://github.com/mockito/mockito/wiki/How-to-write-good-tests`

Adding Mockito hints to exception messages (JUnit) (Experimental)

When a JUnit test fails, an exception is thrown and a message is presented. Sometimes, it is enough to find a reason for this mistake and to find the solution. Mockito, however, goes a step further and tries to help the developer by giving him additional hints regarding the state of the stubbed methods.

Remember that this feature is experimental and the API, name, or anything related to it may change in time. What is more, the whole feature may get deleted in time!

Getting ready

For this recipe, let's assume that our system is the `MeanTaxFactorCalculator` class that calculates tax through `TaxService`, which has two methods—`performAdditionalCalculation()` and `getCurrentTaxFactorFor(...)`. For the sake of this example, let's assume that only the latter is used to calculate the mean value:

```
public class MeanTaxFactorCalculator {

    private final TaxService taxService;

    public MeanTaxFactorCalculator(TaxService taxService) {
        this.taxService = taxService;
    }

    public double calculateMeanTaxFactorFor(Person person) {
        double currentTaxFactor = taxService.
getCurrentTaxFactorFor(person);
        double anotherTaxFactor = taxService.
getCurrentTaxFactorFor(person);
        return (currentTaxFactor + anotherTaxFactor) / 2;
    }

}
```

We wanted to check whether our system under test is calculating the proper result, so we wrote the following test but made a mistake and stubbed a wrong method (I'm using the `BDDMockito.given(...)` and AssertJ's `BDDAssertions.then(...)` static methods—refer *Chapter 7*, *Verifying Behavior with Object Matchers*, for how to work with AssertJ or how to do the same with Hamcrest's `assertThat(...)` method):

```
@RunWith(MockitoJUnitRunner.class)
public class MeanTaxFactorCalculatorTest {

    static final double UNUSED_VALUE = 10;

    @Test
    public void should_calculate_mean_tax_factor() {
        // given
        TaxService taxService = given(Mockito.mock(TaxService.class).
performAdditionalCalculation()).willReturn(UNUSED_VALUE)
.getMock();
        MeanTaxFactorCalculator systemUnderTest =
 new MeanTaxFactorCalculator(taxService);

        // when
        double meanTaxFactor = systemUnderTest.
calculateMeanTaxFactorFor(new Person());

        // then
        then(meanTaxFactor).isEqualTo(UNUSED_VALUE);
    }

}
```

The test fails and what we can see is the standard JUnit comparison failure being thrown (presenting only the most important part of the stack trace) as follows:

```
org.junit.ComparisonFailure:
Expected :10.0
Actual   :0.0
```

Now let's take a look at how to use Mockito's experimental features to get more Mockito related information appended to the error message.

How to do it...

If you want to have more information presented in your error message, you have to perform the following steps:

1. Annotate your JUnit test with `@RunWith(VerboseMockitoJUnitRunner.class)`.
2. Define your mocks and perform stubbing inside the test method (unfortunately, you can't use annotations or initialize fields outside test methods).

What happens next is that additional exception messages can be seen in the exception that is thrown as follows:

```
org.mockito.internal.exceptions.
ExceptionIncludingMockitoWarnings:
 contains both: actual test failure *and* Mockito warnings.
This stubbing was never used   -> at ...MeanTaxFactorCalculatorTest.
should_calculate_mean_tax_factor
(MeanTaxFactorCalculatorTest.java:30)

 *** The actual failure is because of: ***

Expected :10.0
Actual   :0.0
```

How it works...

When the test is run, `VerboseMockitoJUnitRunner` appends a listener. When the test is started, this listener finds all the stubs through `WarningsCollector`, including the unused stubs for given mocks.

As the Mockito developers state it in the code, they are indeed using a very hacky way to append a message to the thrown exception after the test fails. The `JUnitFailureHacker` class is instantiated and, by means of the `Whitebox` class, the internal state of a private field of the JUnit's `Failure` object is modified with additional Mockito messages.

Adding additional Mockito warnings to your tests (JUnit) (Experimental)

If you would like Mockito to append some additional warning messages to the console, which would help you when your test fails, then this recipe is perfect for you. It's very much related to the previous one so, in order to understand the background, please refer to the introductory part of the previous recipe.

Remember that this feature is experimental and the API, name, or anything related to it may change in time. What's more, the whole feature may get deleted in time!

How to do it...

If you want to have more information presented in your error message, you have to perform the following steps:

1. Annotate your JUnit test with `@RunWith(ConsoleSpammingMockitoJUnitRunner.class)`.
2. Define your mocks and perform stubbing inside the test method (unfortunately, you can't use annotations or initialize fields outside test methods).

What happens then is that additional exception messages gets printed on the console after the exception that is thrown:

```
This stubbing was never used   -> at ....MeanTaxFactorCalculatorTest
.should_calculate_mean_tax_factor
(MeanTaxFactorCalculatorTest.java:25)
```

How it works...

When the test is run, `ConsoleSpammingMockitoJUnitRunner` appends a listener that finds all the stubs through `WarningsCollector`, including the unused stubs for given mocks. When all of the warnings get collected, the `ConsoleMockitoLogger` class prints them to the console after the test has failed.

Where to buy this book

You can buy Mockito Cookbook from the Packt Publishing website: `http://www.packtpub.com/mockito-cookbook/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/mockito-cookbook/book