CONTENTS

1 ABOUT AUTHOR: 6

2 INTRODUCTION TO TEST DRIVEN DEVELOPMENT 7

3 PART 1 8
- 3.1 Cycle: Red-Green-Refactor 8
- 3.2 Black-box testing and White-box testing: 9
  - 3.2.1 White-box Testing: 9
  - 3.2.2 Black-box Testing: 9
  - 3.2.3 The grey box : 10
- 3.3 JUnit: 10
- 3.4 JUnit Test Class: 11
- 3.5 Test driven development using with Junit: 12
- 3.6 Refactoring Unit Tests: 23
- 3.7 Code Coverage: 23

4 PART 2 27
- 4.1 Junit: 27
  - 4.1.1 Structure 27
- 4.2 Example 30
  - 4.2.1 Exception testing 36
- 4.3 MUTATION TEST WITH PIT 41
- 4.4 Mocking 44
  - 4.4.1 Mockito 44
- 4.5 Mockito - Adding Behavior 53
- 4.6 Mockito - Verifying Behavior 57
- 4.7 Mockito - Expecting Calls 62
- 4.8 Mockito - Varying Calls 66
- 4.9 Mockito - Exception Handling 70
- 4.10 Mockito - Create Mock 74
- 4.11 Mockito - Ordered Verification 76
- 4.12 Mockito - Callbacks 80

5 PART 3 86
- 5.1 Maven: 86
  - 5.1.1 Maven: Installation 88
  - 5.1.2 Import a Maven project in Eclipse 89
  - 5.1.3 Create a Maven project from Eclipse using an archetype 89
  - 5.1.4 Java settings 91

5.1.5 Dependencies: POM 92
5.1.6 SNAPSHOT 92
5.1.7 m2e 93
5.1.8 Properties 95
5.1.9 Resources 96
5.1.10 Example of a Maven project 97
5.1.11 Build with Maven 98
5.1.12 Maven lifecycles 99
5.1.13 Run Maven using command line 101
5.1.14 Run Maven goals 104
5.1.15 Run Maven from Eclipse 107
5.2 Maven packaging 107
5.2.1 Packaging "jar" 107
5.2.2 Packaging "pom" 107
5.2.3 Parent and modules 108
5.2.4 Create a parent project and a module project 109
5.3 Configuring a Maven plugin 115
5.3.1 Configuring the Maven compiler plugin 117
5.3.2 Create source and javadoc jars 119
5.3.3 Configuring the generated jar 121
5.3.4 Plugin management 123
5.3.5 PIT Maven plugin 124
5.3.6 Configuring the JaCoCo Maven plugin 125
5.4 Maven profiles 126
5.5 Maven deploy 129
5.6 Maven wrapper 130
6 CONTINUOUS INTEGRATION 132
6.1 Example 134
6.2 Travis CI 135
6.2.1 Configure the Travis build 136
6.3 Code coverage with Coveralls 137
6.3.1 Badges 141
7 DOCKER 144
7.0.1 Container: virtualization with minimal overhead 144
7.0.2 Scalability, high availability and portability 146
7.0.3 Docker: structure and functions 146
7.0.4 Docker Images 146
7.0.5 Docker Hub 147
7.1 Docker- Start 147
7.2 Docker and Java application 151

7.2.1 The Dockerfile 153
8 DOCKER CONTINUE 159
8.0.1 Build the Docker image from Maven 159
9 INTEGRATION TESTS 164
9.0.1 Example 165
9.1 Unit tests with databases 169
9.1.1 Integration tests 172
9.1.2 Source folder for integration tests 172
9.1.3 Docker and Testcontainers 173
9.1.4 Tests with Maven 175
9.1.5 tests with Docker and Maven 177
9.1.6 integration tests in Travis CI 182
10 GRAPHICAL USER INTERFACE TESTING 183
10.1 What do you Check-in GUI Testing? 183
10.1.1 Example GUI Testing Test Cases 184
10.1.2 Challenges in GUI Testing 184
10.1.3 GUI Testing Tools 185
10.2 Consider using a BDD framework 185
11 END-TO-END TESTING 195
11.1 Why End to End Testing 195
11.1.1 End to End Testing Process: 195
11.1.2 How to create End-to-End Test Cases? 196
12 PART 4 198
12.1 Code Quality-SonarQube -: 198
12.1.1 Key Aspects to Measure 198
12.1.2 How to Improve Code Quality 199
12.2 SonarQube 201
12.3 Key concepts in Static Code Analysis 202
12.3.1 Write Clean Code 203
13 JAVA PROJECT USING TEST DRIVEN DEVLOPMENT 207
14 CONCLUSION 208

LIST OF FIGURES

Figure 1 Annotation 11
Figure 2 The JUnit test lifecycle 30
Figure 3 Create Maven project 90
Figure 4 Java Setting 91
Figure 5 Dependencies: POM 92
Figure 6 Getting the code for the Travis badge 142
Figure 7 Preview of the README file 143
Figure 8 Getting the code for the Coveralls badge 143
Figure 9 The README file with both badges 143

1

ABOUT AUTHOR:

Joseph Thachil George is a Technical Consultant for International Game Technology (IGT), Rome, Italy. Additionally, Joseph is pursuing doctorate (PhD) in Computer Science and Engineering at the University of Lisbon, Portugal. He has completed M. S in Cyber Security from the Università degli Studi di Firenze, Italy. In addition, he is also part of the research group (DISIA) of the University of Florence, Italy, and the research group (INESC-ID Lisbon) of the University of Lisbon, Portugal.

His research interests cover Dynamic Malware Analysis, Blockchain technology- Hyperledger fabric, and cyber security. He published five books *Cybercrime and Social Media Relationships*, *Designing Distributed Systems* , *Social Network Analysis* ,*Advanced Distributed Systems Design* and ,*Network Security Management* respectively. In IGT he is been a part of various project related to game configuration and integration in various platform. Specialized in Java and spring boot-based projects.

He has also worked in various companies in India, Angola, Portugal, and UK. In total he has seven years of experience in various IT companies.

2

INTRODUCTION TO TEST DRIVEN DEVELOPMENT

In computer science, software development, test-driven development (in short TDD is a software development model, which provides that the drafting of the automatic testing occurs before the software that must be tested, and that the development of a software application is exclusively oriented to the goal of passing the automated tests you have previously prepared.

In recent years, tests have become even more integral to the software development phase. Practices such as Test-Driven Development and Continuous Integration are widely used by industry. But writing the tests is not enough, you need to write quality tests. TDD provides for the repetition of a short three-phase development cycle, called the "TDD cycle". In the first phase (called the "Red phase"), the programmer writes an automatic test for the new function to be developed, which must fail because the function has not yet been realized. In the second phase (called the "green phase"), the programmer develops the minimum amount of code needed to pass the test. In the third phase (called "gray phase" or refactoring), the programmer refactors the code to adapt it to certain quality standards:

3

PART 1

3.1 CYCLE: RED-GREEN-REFACTOR

Red Phase:

In TDD, the development of a new feature always begins with writing an automatic test to validate that feature, i.e. to check if the software exposes it. Since the implementation does not yet exist, writing the test is a creative task, since the programmer must determine in what form the functionality will be exposed by the software, and understand and define its details. For the test to be complete, it must be executable, and when it is performed, produce a negative result.

Green phase:

In this stage, the programmer must write the minimum amount of code needed to pass the test that fails. Written code is not required to be of good quality, elegant, or general; the only explicit goal is for it to work, that is, pass the test. In fact, it is explicitly forbidden by the practice of TDD to develop parts of code not strictly aimed at passing the test that fails. When the code is ready, the programmer again launches all available tests on the modified software (not just the one that previously failed). This way, the programmer can immediately realize if the new implementation caused pre-existing test failures, i.e. caused regression in the code. The green phase ends when all tests are " green "(i.e. they are passed successfully).

Refactor (gray phase):

When the software passes all the tests, the programmer devotes a certain amount of time to refactoring it, that is to improve its structure through a procedure based on small controlled modifications aimed at eliminating or reducing objectively recognizable defects in the internal structure of the code. Typical examples of refactoring actions include

choosing more expressive identifiers, eliminating duplicate code, simplifying, and streamlining the source architecture.

In general, we could summarize:

Red:

Write a test and
Make sure it fails

Green:

Write only enough code to make the test succeed

Refactor:

In case, refactor the code to make it clean.
Re-execute all the tests to make sure they still succeed.

3.2 BLACK-BOX TESTING AND WHITE-BOX TESTING:

Within the automated testing world there are two predominate testing methodologies: black-box and white-box.

3.2.1 *White-box Testing:*

The white box approach is based on the premise of testing a product knowing how it is done and having access to the code. This type of approach is generally pursued by developers because they know the code. Such tests would be better if they were performed by industry experts or at least not by the same person who developed the code. This is because those who test what he has produced are less likely to find problems regarding the code that he himself has created. The advantage of this approach is that you can do tests (unit tests) much more targeted than the other approaches and therefore could lead to detect problems that could hardly be found otherwise. Examples of this approach are techniques based on covering paths, decisions, conditions, etc.

3.2.2 *Black-box Testing:*

The black box approach is what is generally used by testers. It is based on the assumption of not knowing how the functionality is realized, only interested in what it does, so the point of view is (almost) the same as

the end user focusing on what the system does and the requirements provided by the specifications. The disadvantage of this approach is that it often leads to a coverage of all scenarios rather poor so that bugs could be left latent that could, then, be detected because of later developments. Examples of this approach are only partitioning techniques, limit value analysis, cause and effect tables, the exploration test, the error guessing test, the state diagram, etc.

3.2.3 *The grey box :*

approach is an intermediate approach, in practice you know broadly how the functionality was implemented and the data structures used. In this way you can perform more targeted tests than the black box approach, obviously not knowing the whole code the tests will be less effective than the white box technique.

3.3 JUNIT:

In Computer Science JUnit is a unit testing framework for the Java programming language. The experience with JUnit has been important in the growth of the idea of Test-Driven Development and is one of a family of unit testing Frameworks known collectively as xUnit.

The framework is currently at version 5, which is organized into 3 subprojects / modules and needs java version 8 or newer. Version 4 brought structural changes compared to Version 3, with which it is incompatible. The classes that make up the framework belong to different packages for versions 3 and 4; junit.framework up to 3.8, org. junit from 4.

Each test is represented by a public method annotated with @Test.

Each test method should verify a single scenario – Using a part of the SUT (e.g., calling a method on the SUT class).

Verification consists in checking that the output (or outcome) is as expected.

4 stages:

Setup: prepare the context for testing the SUT:

Creates an instance of the SUT (this instance is usually called “fixture”).

Prepare the context for the test.

Exercise: invoke the method we want to test on the fixture.

Verify: verify that the outcome is as expected.

IMPORTANT: a test should verify a single behaviour

If a method has 5 different possible behaviours.

Then write 5 test methods

Teardown: clean-up to bring the execution environment as it was before (e.g., clean-up a database, remove some files).

This way, subsequent tests are not influenced.

In our example this phase is not required.

3.4 JUNIT TEST CLASS:

The execution process is then driven by the annotations that we see below:

Annotation	Usage
@Test	to annotate test methods
@Before	to annotate a method to run before a test case runs (for example, opening a connection to a database, or a stream)
@After	to annotate a method to run after running a test case (closing previously open resources)
@BeforeClass	like @before but only at the beginning of the test run
@AfterClass	as @after but only at the end of test execution
@Ignore	used to avoid running a test (avoiding commenting)

Figure 1: Annotation

Methods for assertions:

Static methods defined in org.junit.Assert

- assertEquals(expected, actual) – use equals
- assertSame(expected, actual) – use ==
- assertTrue(expression), assertFalse(expression)
- assertNull(expression), assertNotNull(expression)
- Etc.

If an assertion fails, the test terminates with failure.

If an exception is thrown during the test, the test terminates with error.

Otherwise it succeeds.

We need to remember that every single test method must be executable in isolation and Never ever write a test method relying on the side effects of other test methods, even because JUnit does not specify the order of execution.

The order could be specified.

but do not do that just to make tests work

From JUnit Runner's javadoc: says" – "The default runner implementation guarantees that the instances of the test case class will be constructed immediately before running the test and that the runner will retain no reference to the test case instances, generally making them available for garbage collection."

You cannot be sure that JUnit reuses the same instance to execute test methods. For example, consider below program:

One of the two tests should fail, should not it, No they both succeed.

3.5 TEST DRIVEN DEVELOPMENT USING WITH JUNIT:

The unit tests specify and validate the functionality of small pieces of code. Producing more code will depend on as much code will enable the unit test to pass. Then the refactor will simplify the production code and the test code.

Create a new Java project, e.g., junit.example.

Inside the project, create a new source folder, tests, where we will write all our JUnit tests. (Tests should be kept separate from the actual

```java
package junit.example;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

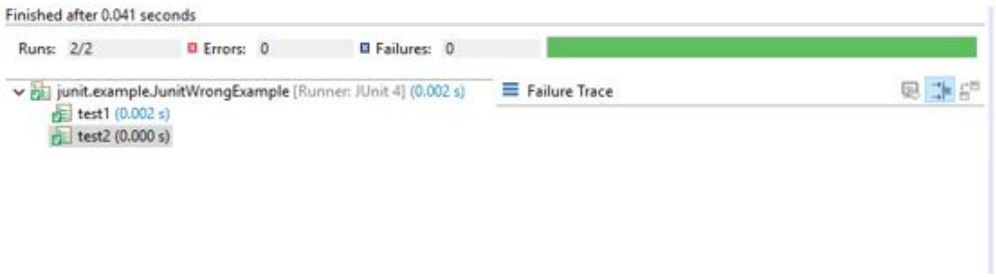
public class JunitWrongExample {

 private int count = 1;

 @Test
 public void test1() {
 count++;
 assertEquals(2, count);
 }

 @Test
 public void test2() {
 count++;
 assertEquals(2, count);
 }

}
```



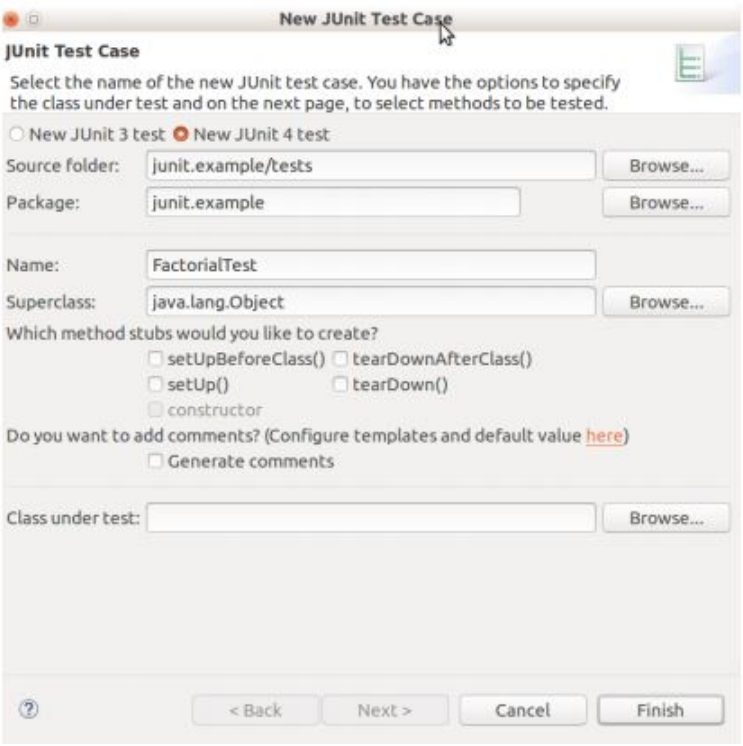
production code, which will be written in the source folder src): right-click to project, New → Source Folder.

Since we will use TDD from scratch, we will write tests in the source folder tests and we will create the class and the method to implement in the source folder src, using the tools of Eclipse.

Create the test case FactorialTest: Right-click on the tests folder, and select New →Junit Test Case, in the dialog use these settings:

Press Finish and you will be asked to add JUnit to your build path (accept the request):

Remember that each test method in a test case should test a specific scenario of the SUT. Since we are implementing a recursive function, we must start dealing with the base case of the recursion. For the factorial, the base case is input 0.



Inside the test case class FactorialTest we write our first test method:

```
@Test
 public void testBaseCase() {
 Factorial factorial = new Factorial();
```

The test fails since it does not compile: the class Factorial does not exist yet. Use Eclipse quickfix to create the class (you can hover on the error, click on the marker in the editor's rules or, even better, use the key shortcut Ctrl + 1 which you should learn so that you quickly access any quickfix):

The Factorial class must be created in the source folder src:

Now the test compiles. Inside the test we now call the method compute on the Factorial object, passing an integer and saving the result in an integer variable:

Again, the test will not compile since there's no such method in the class Factorial. We will use the quick fix to create it:

```
@Test
 public void testBaseCase() {
 Factorial factorial = new Factorial();
 int result = factorial.compute(0);
```

Note that since we pass to the method an integer and we store the result in an integer variable, Eclipse can infer the signature of the method to create:

```
    public int compute(int i) {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

Save the file and the test will compile.

Eclipse implemented the method by returning the default value for the return type. Let us complete the test with the "verify" phase. We write an assertion checking that the result is 1 (the factorial of 0 should be 1):

```
@Test
 public void testBaseCase() {
 Factorial factorial = new Factorial();
 int result = factorial.compute(0);
 assertEquals(1, result);
 }
```

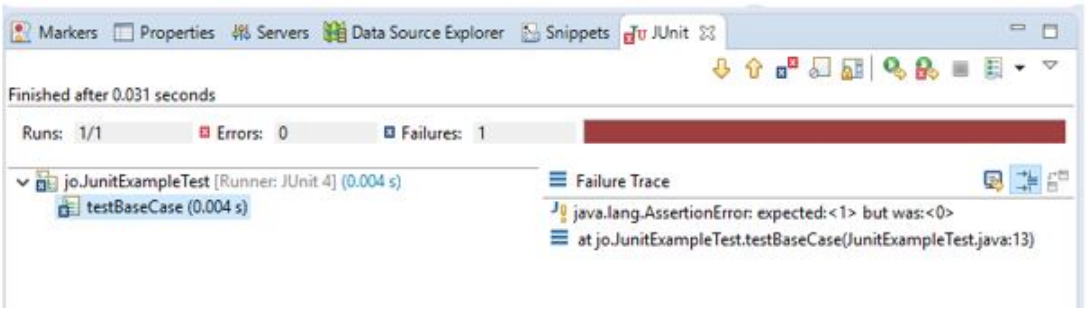
We expect the test to fail and we verify that by running the test: right-click on FactorialTest.java and select Run As →. JUnit Test.

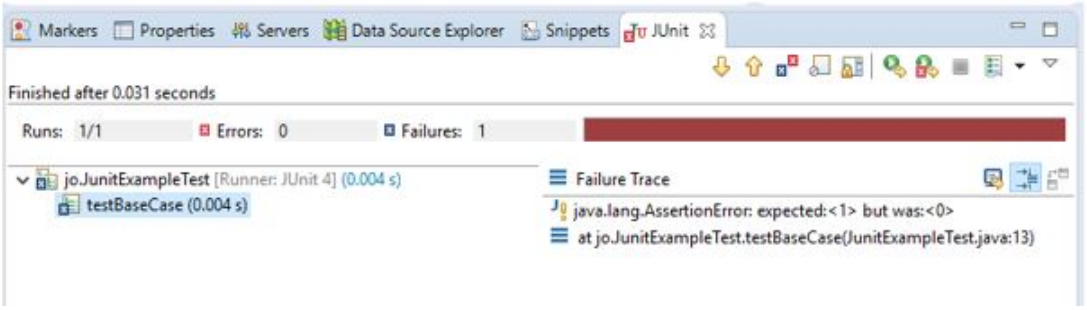
TIP: When a package contains several JUnit test cases, you can execute them all with a right click on the package and selecting Run As → JUnit Test. Similarly, if a project contains several packages with test cases, you can execute them all with the same contextual menu on the project.

JUnit will execute all the test method of the class and report the results in the view "JUnit":

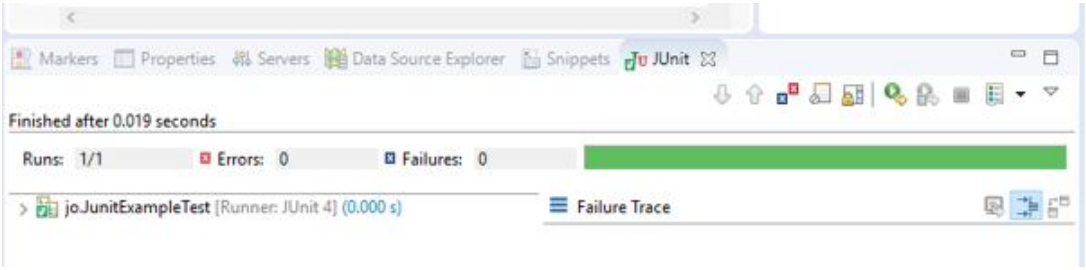
The test fails for the reason we knew: we expect 1 but the result is 0 (note that the expected and actual values are reported by the failure). Recall that it is crucial to make the test fail first, otherwise we would not test the behaviour of the SUT correctly.

Now we must write in the Factorial class ONLY the code to make this very test succeed:





Let us run the test again and this time the test succeeds, as we expected. The JUnit view will show the "green bar": all tests succeeded (in our case, there is currently one test):



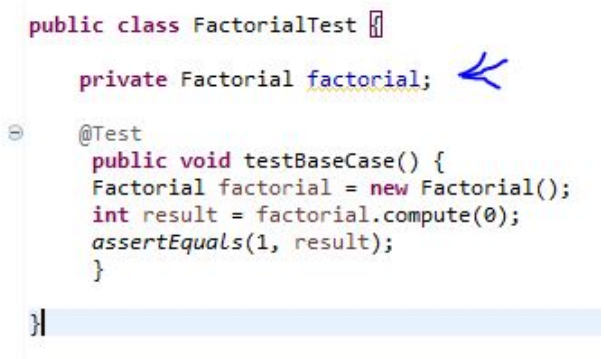
Now we must consider the non-base case scenario, that is, the recursive case. So, we must write another test method.

Do not be tempted to copy and paste the previous tests: tests must be clean code as well, so we must avoid copy-and-paste. What do the two tests have in common? Given an input to the compute method they expect a specific output. So, let us refactor the previous test to have some reusable code for the second test.

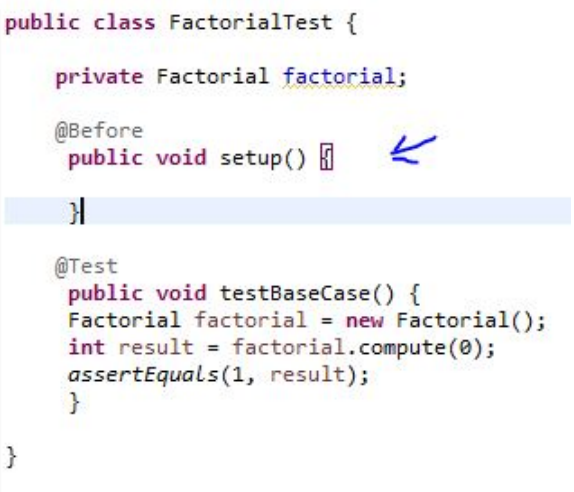
First of all, the SUT object should be a field of the test class, which is created freshly before every test (remember that each test must be executable independently from each other; in this very example, the Factorial class is stateless so reusing the same instance would not be a problem, but in the future we could make Factorial stateful – for instance, we could cache the results). Such field should be initialized in a method annotated with @Before

We will perform refactoring using the tools of Eclipse.

Add a private variable in FactorialTest class .



We create a method annotated with @Before (Use the content assist for the annotation, so that the corresponding import will be automatically inserted); the name of the method is not important, but we use a meaningful name:



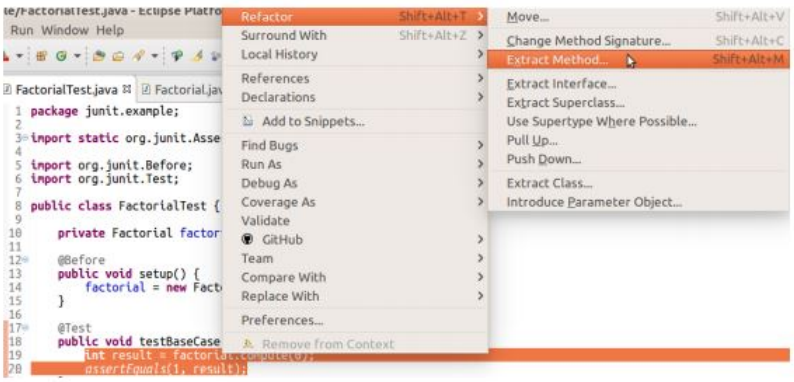
Move the initialization of the field from the test method to the setup method.

```
import org.junit.Before;
import org.junit.Test;

public class FactorialTest {

    private Factorial factorial;
    @Before
    public void setup() {
    factorial = new Factorial();
    }
    @Test
    public void testBaseCase() {
    int result = factorial.compute(0);
    assertEquals(1, result);
    }
   }
```

Now we use the "Extract method": select the body of the method testBaseCase, rightclick, Refactor Extract Method... → (or the shortcut Shift+Alt+M):



In the dialog we choose a name for the new method, e.g., assertFactorial (we make it private since we use it only in this very test class):

Now we use the refactoring for turning an expression in a method into a new parameter of the method (this will add a new method and in each call site it will add as argument the original expression).

We want to parameterize the new method with a parameter for the input (to be passed to compute) and a parameter for the expected output (to be passed to assertEquals). The expression that must be turned into

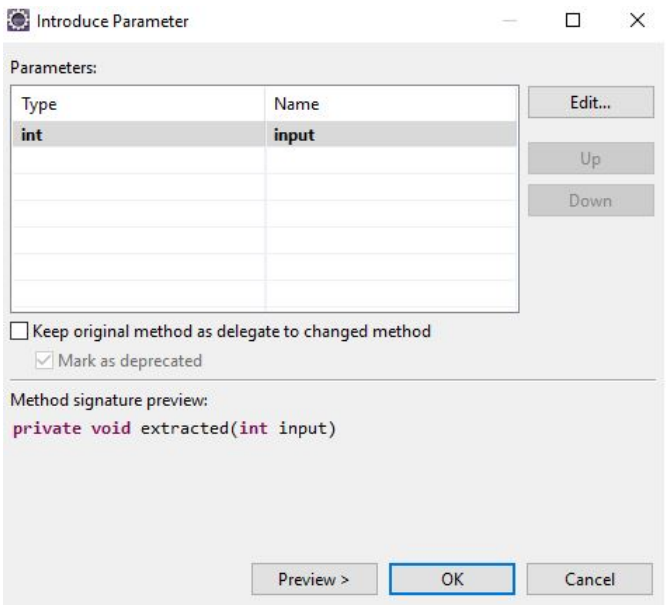
```
import org.junit.Before;
import org.junit.Test;

public class FactorialTest {

    private Factorial factorial;
    @Before
    public void setup() {
    factorial = new Factorial();
    }
    @Test
    public void testBaseCase() {
    extracted();
    }
    private void extracted() {
        int result = factorial.compute(0);
         assertEquals(1, result);
    }
    }
```

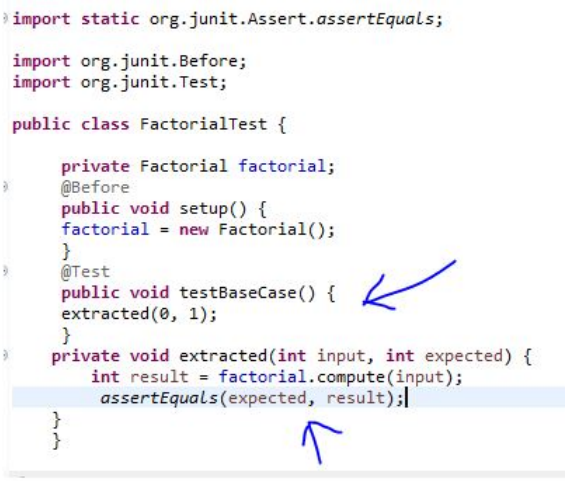
the input parameter is 0 in this example, and the other expression to be turned into the other parameter is 1 in this example.

Select "0" in the method assertFactorial, right-click and Refactor → Introduce Parameter..., and specify a name for the new paramenter, e.g., input:



Do the same for "1" specifying the name expected. The final result after this refactoring should look like this:

Of course, run the test again just to make sure we didn't break anything with the refactoring. Now we can easily write the new test for the non



base case. For example, with input 5 we expect 120. Note that the new test is just one line of code and is readable:

```
@Test
public void testNonBaseCase() {
    assertFactorial(5, 120);
}
```

Let's run all the tests. We expect the new test to fail, since our method always return 1.

By following strictly TDD, we should modify the method compute so that the new test passes (see below for additional explanations):

```
public int compute(int i) {
    return 120;
}
```

Let us run all the tests again: the new test succeeds but the previous test will fail. You see that it is crucial to run ALL the tests after any modification.

NOTE: since unit tests must be fast, running all the tests does not require much time, so it does not impact your productivity.

We should now modify the method compute so that all tests succeed. A trivial way to do that is this one:

Now all tests succeed.

What is wrong in the process we just performed

```
public int compute(int i) {
    if (i == 0)
        return 1;
    return 120;
}
```

The implementation does not make sense (and it does not implement the factorial), though we followed TDD

```
@Test
public void testNonBaseCase() {
    assertFactorial(5, 5 * factorial.compute(4));
}
```

This test will fail with our stupid implementation of the non-base case. Moreover, it specifies the behaviour of the factorial correctly.

Let us fix the implementation of compute to make all tests succeed:

```
public int compute(int i) {
    if (i == 0)
        return 1;
    return i * compute(i-1);
}
```

Note that these tests specify the behavior of factorial at least for 0 and 5; due to the recursive nature of the test, which reflects the recursive nature of factorial, it is much harder to write an implementation that works only for these two input values and that does not work for other inputs, thus we're much safer now.

NOTE: all this process for implementing the factorial might seem odd: couldn't we simply write the factorial all together? If you want to follow TDD the answer is no. Even because in a more complex scenario all these steps would make the implementation much easier. Moreover, since you have to learn TDD, you must practice it all the way. Finally, this allowed us to write the tests in a clean and clear way.

Are we done? Did we miss something?

What happens if the input is negative?

The factorial is not defined on negative numbers.

Let us write a test, specifying a fake attended result:

```
@Test
public void testNegativeInput() {
    assertFactorial(-1, 0);
}
```

TDD tells you that you must be aware of the reason why the test will fail. In this example, we expect a StackOverflowException since our recursion does not terminate.

Now we must write a test that specifies that given a negative (invalid) input we expect an exception; for example, an IllegalArgumentException. (Of course, since we're writing tests as specification of software that we still need to implement, it is now time to design and decide what happens with invalid inputs; you could also return a value such as -1, but that would not make sense for the factorial: it is much better to throw exceptions when input are invalid).

We can write the test like that:

```
@Test
public void testNegativeInput() {
    try {
        assertFactorial(-1, 0);
        fail("should not get here");
    } catch (IllegalArgumentException e) {
        // OK
    }
}
```

Verify that the test still fails, and modify compute so that it succeeds:

```
public int compute(int i) {
    if (i < 0)
        throw new IllegalArgumentException("negative input");
    if (i == 0)
        return 1;
    return i * compute(i-1);
}
```

Now all tests succeed.

A better way to write tests with expected exceptions is to use JUnit specific mechanisms (see also JUnit documentation, https://github.com/junit-team/junit4/wiki/exceptiontesting):

```
@Test(expected=IllegalArgumentException.class)
public void testNegativeInput() {
    assertFactorial(-1, 0);
}
```

Such a test will fail if an exception of the specified type is not thrown during the execution.

3.6 REFACTORING UNIT TESTS:

One of the great things about unit tests is that they allow you to refactor and restructure your code in safety, because you get immediate feedback when you break it. Because you can (and do) constantly refactor your code, it is simpler, more modular, and easier to extend and maintain. But what about the tests themselves? Over time, you will develop a testing codebase that may be larger than the mainline code.

After refactoring, your tests will better convey exactly what you are trying to test. There is a hidden prize, however: in many cases you will discover or create methods that can be reused. For example, an assertion that uses XPath to verify the contents of a generated XML document. When you find such code, pull it out into a "testing library," available for your next project.

3.7 CODE COVERAGE:

In computer science, code coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. A program with high code coverage, measured as a percentage, has had more of its source code executed during testing which suggests it has a lower chance of containing undetected software bugs compared to a program with low code coverage.

There are a few tools for computing code coverage. In Java the most used is JaCoCo (Java Code Coverage), http://www.eclemma.org/jacoco/.

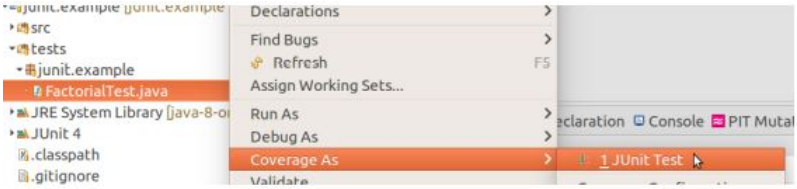
The tool executes a Java application and provides a report with all the lines of code executed during the application. If there are branch instructions, like an if then else, it also says if all branches are executed during the application. If the application we run is a test suite we get the parts of our application that are executed during the tests. If the tests are written correctly, this will tell you which parts of the applications are tested.

The JaCoCo plug-in for Eclipse is EclEmma: http://www.eclemma.org/.

Use the update site http://update.eclemma.org and install it "EclEmma Java Code Coverage". Since version 3, EclEmma is an official Eclipse project so it is available also from the main Eclipse update site.

NOTE: If you installed the package "Eclipse IDE for Java Developers" then EclEmma is already installed in your Eclipse.

Once installed, you can run tests using this additional context menu "Coverage As → JUnit Test":



Tests will execute as before, but this time, JaCoCo will keep track of instructions that are executed and will show a report in the view "Coverage". Java editors will also be colored accordingly:

"green" means covered

"red" means uncovered (not executed at all)

"yellow" means that some branches are not executed (e.g., in an if then else)

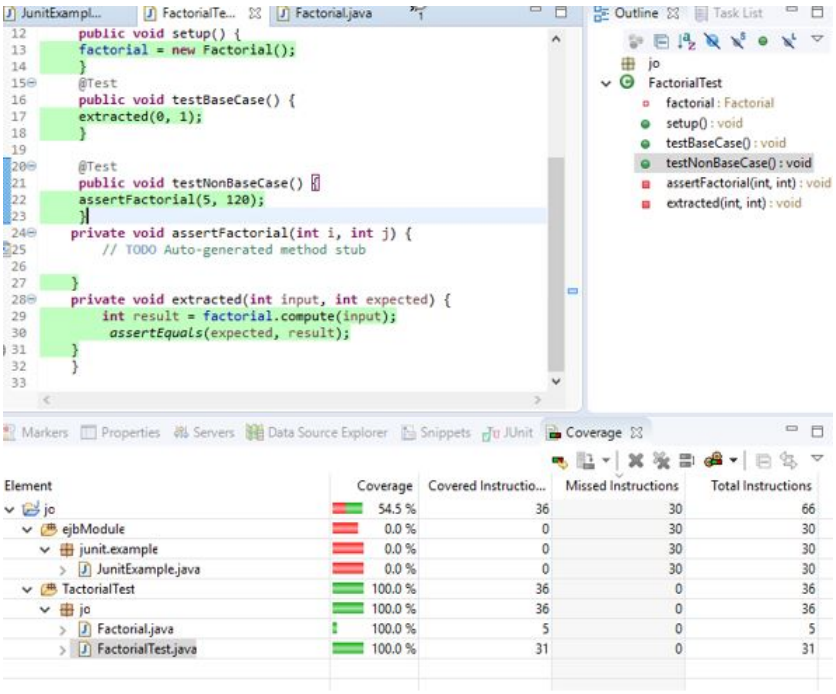
In our example, the Factorial class will all be green:

Note that the test class will have a red color for the testNegativeInput, as if that test was never executed:

This is a limitation of JaCoCo that does not consider statements that throw exceptions as executed. This is not a problem, since test classes should be excluded from code coverage: we must concentrate on SUT classes. In EclEmma such exclusion can be configured in the Run Configuration of the test: in the tab "Coverage" you can refine the scope of the analysis by excluding tests directories:

In EclEmma such exclusion can be configured in the Run Configuration of the test: in the tab "Coverage" you can refine the scope of the analysis by excluding tests directories:

Run the coverage again and you will see that tests will not be considered in the report. Once configured correctly, you may want to save the

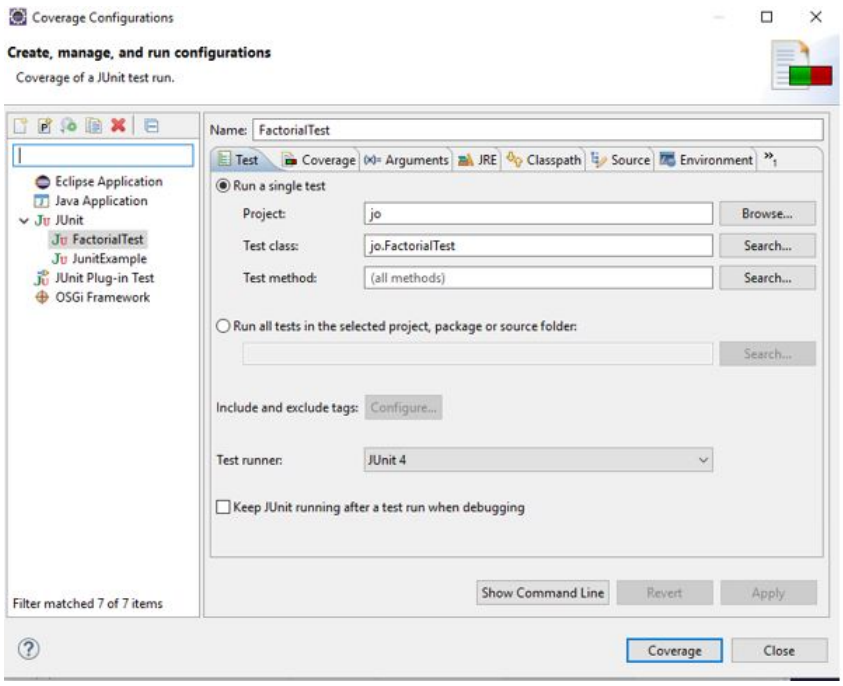


launch configuration of the coverage (and tests) in the Eclipse project (recall that otherwise launch configurations are saved by default in the workspace): in the tab "Common", specify "Shared file" and specify a directory inside the project (e.g., the root of the project folder):

A file will be saved FactorialTest.launch (an XML file). From now on, you can use that launch to launch tests with code coverage (with the contextual menu on that launch file). If you import that project in another workspace, you will still have your custom launch. Let's experiment with code coverage let's uncomment the test for the exception and run the code coverage again. You will see that one line will not be covered and the if will have an uncovered branch:

In big project, it is best practice to have a high code coverage percentage. For such projects, reaching 100 Percentage is not that important. In our context, you should always reach 100 Percentage code coverage (excluding code coverage of tests) We will see in future lessons, that even a 100 Percentage code coverage does not guarantee that your code is correct.

In particular, you should always ask yourself "do my tests effectively test all the behaviour of my code?".



Moreover, remember that code coverage only checks that code is executed, but if your tests do not perform assertions then code coverage is completely meaningless.

4

PART 2

4.1 JUNIT:

In this chapter we get familiar with the main testing framework that we will use throughout the book, JUnit, which is the most popular testing framework for Java developers. As mentioned in Chapter Testing, The "Unit" in "JUnit" is definitely misleading: JUnit is not only for unit tests. In this book, we use JUnit 4, https://junit.org/junit4/, in particular, its latest version, 4.13. Although, JUnit 5 has been released for some time now and it's pretty stable, we prefer to stick with JUnit 4 since not all the testing frameworks and tools we are going to use are working completely fine with JUnit 5.

In this chapter, we will not apply TDD. Thus, we will write the tests after writing the code to test. In this chapter we give an introduction to JUnit. Moreover, we start to see how to structure projects, separating tests from production code. We also see a few additional testing frameworks for enhancing the use of JUnit, especially the assertion API.

Moreover, we also show a few best practices in writing tests and some strategies that instead should be avoided.

4.1.1 *Structure*

The typical structure of a test consists of the following 3 main phases.

1. **Setup** → In this phase we create the environment for the test. In unit tests, this usually corresponds to creating the SUT (Software Under Test) and in Java, this usually corresponds to creating an instance of the class to test. This phase establishes the state of the SUT prior to any test activities. The state created in this phase,

consisting of the SUT instance and of possible other related objects needed during the tests, is usually called test fixture. A test fixture is all the things that must be in place in order to test the SUT. It is a fixed state of a set of objects used as a baseline for running tests (https://github.com/junit-team/junit4/wiki/test-fixtures). The test fixture must ensure that there is a well-known and fixed environment in which tests are run. Thus, the test fixture, when setup correctly, ensures that test results are repeatable.

2. **Exercise** → In this phase we interact with the SUT (e.g., by calling an instance method), possibly getting a result from it or generating some side effect in other instances.

3. **Verify** → In this phase we verify that the outcome actually matches the expected behavior, typically using some assertions. This could consist in checking that the returned value of the SUT invoked method is as expected. Alternatively, we could verify that the side effects of the invoked method are as expected, by performing assertions on the changed state of the fixture. These 3 phases are also known in the literature as Arrange, Act, Assert (Beco2), or, typically using the BDD style, Given, When, Then. There's also a fourth phase, which is usually not strictly required, depending on the testing context:

4. **Teardown** → In this phase we cleanup the environment, bringing it back to the condition it used to be before the execution of the test. For example, if we created a few files, we should remove them in this phase.

In JUnit, a test case is a standard Java class, which does not have to inherit from any specific base class. Each test is represented by a public void method in this class, annotated with @Test (org.junit.Test).

When a test case is run, using the JUnit runner, JUnit will execute all methods annotated with @Test. We will run JUnit tests from Eclipse and, later, from Maven, Chapter Maven.

Assertions are performed using static methods of the class org.junit.Assert. Here are some examples:

Assertion methods also have an additional overloaded version taking as the first parameter a string with the custom message to be shown in case of failure. Assertions based on equality usually do not need an

- `assertEquals(expected, actual)`: there are several overloaded versions of this method for different types. This compares the two expressions relying on the method `equals`
- `assertSame(expected, actual)`: as above, but it compares object references with `==`
- `assertTrue(expression)`, `assertFalse(expression)`, `assertNull(expression)`, `assertNotNull(expression)` with the straightforward meaning

additional error message since, in case of failure, JUnit automatically reports the expected and the actual value. On the contrary, if you directly assert a boolean expression, e.g., with assertTrue, it might be useful to provide a better failure message than the default error message, which only states that the assertion failed.

If an assertion fails, the test terminates with a failure. If an exception is thrown during the test, the test terminates with an error. Otherwise it succeeds.

In general, each unit test should test only a single behavior of the SUT. This often means that a test method should contain only a single assertion. Recall that if you have several assertions in a test method, if one fails, the test method terminates immediately, and subsequent assertions are not executed. However, in some cases in order to verify a single behavior several assertions must be written in the same test method. Summarizing, a single test method should exercise and verify a single scenario and behavior, but to do that, it is allowed to write several assertions.

JUnit follows the all or nothing philosophy: if a single assertion fails in a test method, the method fails, the whole test case fails, and, if the test case is part of a test suite (that is, we run several test cases in the same run), then the whole test suite is marked with a failure.

JUnit provides additional method annotations for the setup and teardown phases, which are useful when these phases have a behavior common to all tests.

The following annotations must be used with public static void methods:

@BeforeClass The annotated method is executed only once before any of the test methods in the class is run

@AfterClass The annotated method is executed only once after all the tests in the class have been run

The following annotations must be used with public void methods:

@Before The annotated method is executed before each individual test method

@After The annotated method is executed after each individual test method, even in case of failure or error

The lifecycle of JUnit tests can be represented as in the following figure.

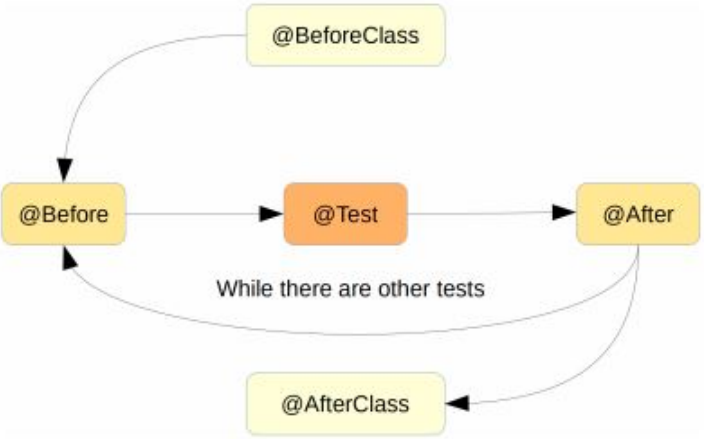


Figure 2: The JUnit test lifecycle

JUnit only takes annotations into consideration during the test lifecycle. However, it is best practice to give annotated methods a very meaningful way. This is especially crucial for test methods: their names should describe what the test is trying to verify.

Each test method must execute independently from other test methods. This means that a test method must never rely on possible side effects of other test methods. In fact, the setup and teardown phases are meant to make each test independent from each other. Moreover, JUnit does not guarantee that test methods are executed in a predefined order. Finally, JUnit does not necessarily reuse the same instance of the test case class for running all its methods. Thus, the state of the test case class, that is, its fields, should never be treated as an object state in standard programs.

4.2 EXAMPLE

In the source folder src we create this class (you first need to create the package testing.example.bank):

```
1  package testing.example.bank;
2  public class BankAccount {
3  private int id;
4   private double balance = 0;
5  private static int lastId = 0;
6
7  public BankAccount() {
8  this.id = ++lastId;
9   }
10
11  public int getId() {
12 return id;
13 }
14
15 public double getBalance() {
16 return balance;
17 }
18
19 public void deposit(double amount) {
20 (amount < 0) {
21 throw new IllegalArgumentException("Negative amount: " + amount);
22 }
23 balance += amount;
24 }
25
26 public void withdraw(double amount) {
27 if (amount < 0) {
28 throw new IllegalArgumentException("Negative amount: " + amount);
29 }
30 if (balance - amount < 0) {
31 throw new IllegalArgumentException
32 ("Cannot withdraw " + amount + " from " + balance);
33 }
34 balance -= amount;
35 }
36 }
```

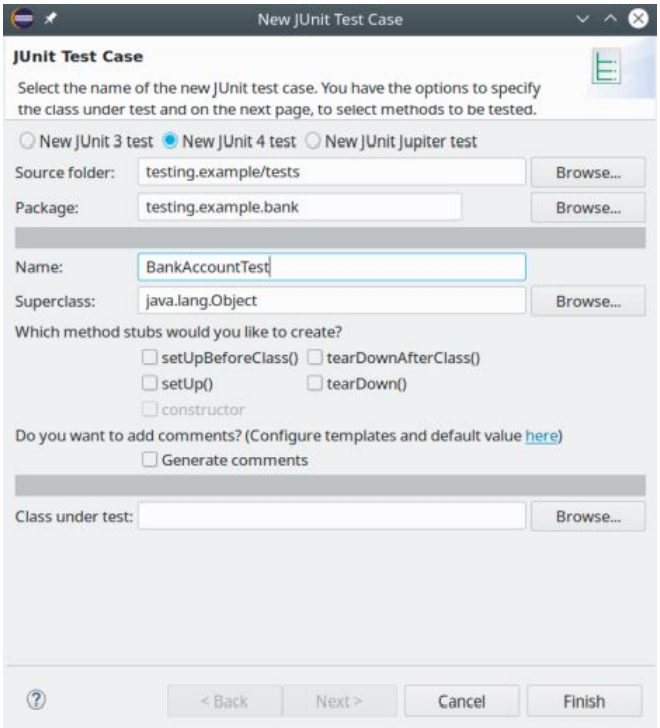
Listing 4.1: BankAccount.c

If you have already read Chapter Eclipse, you should be able to quickly write such a class, using Eclipse tools.

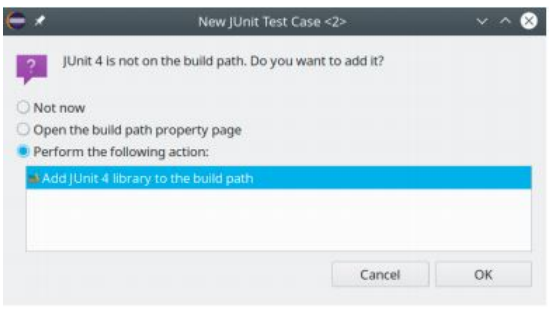
Let's write a few JUnit tests for testing the behavior of this class.

Inside the project, create a new source folder, named tests, where we'll write all our JUnit tests. To create a new source folder, right-click on the project, New → Source Folder.

Create the test case BankAccountTest: Right-click on the tests folder, and select New → JUnit Test Case, in the dialog use these settings:



Press Finish and you'll be asked to add JUnit to your build path (accept the request):



The created test case has the following shape

```
1 package testing.example.bank;
```

```
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Test;
6
7  public class BankAccountTest {
8
9  @Test
10 public void test() {
11 fail("Not yet implemented");
12 }
13
14 }
```

Listing 4.2: BankAccountTest.c

First of all, note that we created the test case in the same package as the one of the SUT testing.example.bank; The SUT and the test case are in the same package but in different source folders. The fact that they are both on the same package has some benefits as we will see later at the end of the chapter.

The import static org.junit.Assert.* will allow us to use all the static methods of JUnit for assertions without using the class prefix Assert. The fail method is one of such methods; which explicitly makes the test fail.

Let's run this test case, just to see how it works. You can do that in several ways:

Right-click on the BankAccountTest.java in the Project Explorer and select Run As → JUnit Test

Right-click on the editor with the BankAccountTest.java file and select Run As → JUnit Test Use the keyboard shortcut (this depends on the operating system, in Linux it can be either F11 or Shift+Alt+X T).

Use the links in the file, if JUnit code minings have been enables.

The JUnit view should appear with the run test, which, as expected, is marked with failure.

Let's remove the current test method and write our first JUnit test method in BankAccountTest. For example, we want to verify that id is automatically assigned and it is positive:

```
1 @Test
```

```
2 public void testIdIsAutomaticallyAssignedAsPositiveNumber() {
3 // setup
4 BankAccount bankAccount = new BankAccount();
5 // verify
6 assertTrue("Id should be positive", bankAccount.getId() > 0);
7 }
```

Listing 4.3: PositiveNumber.c

In this very first test, we do not have any exercise phase, which can be seen as implicitly performed during the setup phase.

Let's run the test case as above, and now the JUnit view should show the green bar, stating that our test succeeded.

Remember that the first argument passed to assertTrue is a descriptive error message to be shown in case of failure of the assertion.

Let's verify that the automatically assigned ids are incremental:

```
1 @Test
2 public void testIdsAreIncremental() {
3 assertTrue("Ids should be incremental",
4 new BankAccount().getId() < new BankAccount().getId());
5 }
```

Listing 4.4: testIdsAreIncremental.c

The test passes.

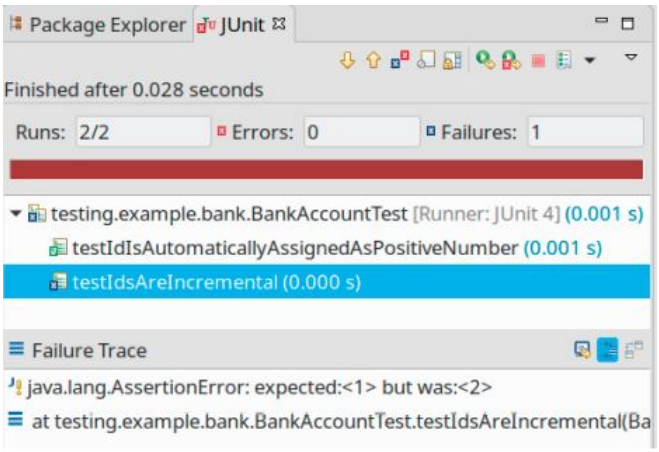
You might be tempted to write the above test in another form, asserting directly the values of the ids:

```
1 // WRONG VERSION!
2 // Works only if this is the first executed test
3 @Test
4 public void testIdsAreIncremental() {
5 assertEquals(1, new BankAccount().getId());
6 assertEquals(2, new BankAccount().getId());
7 }
```

Listing 4.5: testIds.c

This is the wrong way of writing such a test: it will succeed only if it is the first executed test. Remember that BankAccount uses a private static

counter to generate incremental ids, and such a counter is not reset while executing tests. If this test is not executed as the first test it will fail.




```
@Test
public void testDepositWhenAmountIsCorrectShouldIncreaseBalance() {
// setup
BankAccount bankAccount = new BankAccount();
// exercise
bankAccount.deposit(10);
// verify
assertEquals(10, bankAccount.getBalance(), 0);
}
```

Listing 4.6: testDeposit.c

In this example, it is not necessary to test the actual values of the counter, and it is enough to verify that ids are incremental when objects are created. If you really need to assert the actual values of the ids, then you will have to add to BankAccount a mechanism to reset the static counter. Note that this would require to change the interface of the class BankAccount for testing purposes. We'll go back to this matter later. For the moment let's go on testing this class.

Let's write a test for the "happy" case of deposit, that is, when the passed argument is valid: in this case, we have the exercise phase, where we call the method under test and then the verify phase will consist in checking that the balance has been updated correctly:

The test succeeds.

Note that we chose a long name for the test method that should explicitly express what this test verifies. The name of the test follows the convention where the name of the method under test is mentioned, then a specific condition for the test, and the expected result. Thus the name of the method can be read as

Test the method deposit: when the amount is correct then it should increase the balance. Moreover, when comparing two doubles, we need to use the version of assertEquals for double values, which requires a third argument:

delta: the maximum delta between expected and actual for which both numbers are still considered equal.

Since in this test we are not using decimals and we do not perform any advanced mathematical operation on the values, we specify 0 as the delta.

4.2.1 *Exception testing*

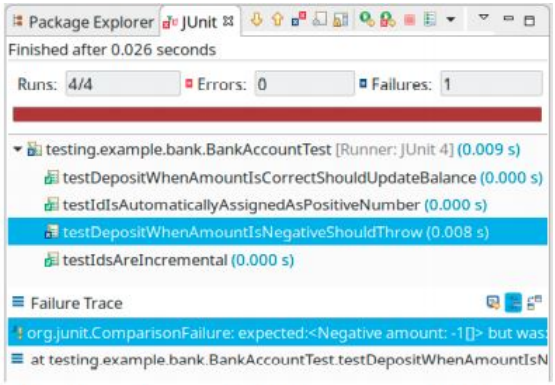
Now we write a test for the case when the passed amount is invalid. The idea is to perform the method call with a negative argument inside a try-catch block; the caught exception is an IllegalArgumentException since that's the one that our SUT should throw in such a situation. If after the method call we don't reach the catch block then our test must fail, since the expected exception is not thrown. In the catch block we verify that the message of the exception is the correct one (this makes sure that our BankAccount threw such an exception). It is also important to verify that the balance has not been changed.

```
1  @Test
2  public void testDepositWhenAmountIsNegativeShouldThrow() {
3  // setup
4  BankAccount bankAccount = new BankAccount();
5  try {
6  // exercise
7  bankAccount.deposit(-1);
8  fail("Expected an IllegalArgumentException to be thrown");
9  } catch (IllegalArgumentException e) {
10 // verify
11 assertEquals("Negative amount: -1", e.getMessage());
12 assertEquals(0, bankAccount.getBalance(), 0);
```

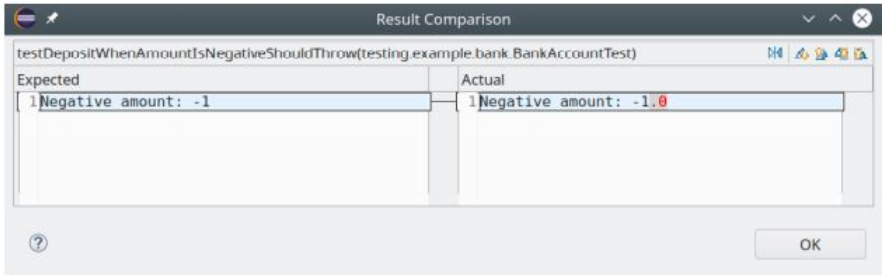
```
13 }
14 }
```

Listing 4.7: ShouldThrow.c

The message will tell you that the two strings passed to the first assertEquals are different. If you double click on the top line in the "Failure Trace" a popup dialog will appear that will highlight the differences (this dialog is available only when comparing strings, in case of failure):



Select the top element of the "Failure Trace", double-click..



The dialog with the highlighted differences will appear

The cause of the failure should now be clear: the amount is a double value and when it is converted to a string (when passing the string message to the exception constructor) the decimal separator is added.

Fix the test so that it succeeds:

```
1 assertEquals("Negative amount: -1.0", e.getMessage());
```

Listing 4.8: assertEquals.c

As we saw in previous example Code Coverage is crucial to discover parts of the code that are not tested – A low percentage should make you

worried – But even a 100 Percentage code coverage does not guarantee that the code is correct and well tested: Mutation testing is a mechanism that helps you evaluate the quality of your tests.

The mutation test is a structural test method aimed at evaluating / improving the adequacy of the test suites and the estimate of the number of errors present in systems under test. The mutation test introduces small mutations within our source code. There are several types of mutations:

1. **Mutation by value:** – the value of a constant or parameter used in your test unit is changed (e.g. the ends of a loop).
2. Decision change: the conditions in your test unit are changed (if a < B raise error)
3. Mutation of statements: some statements are deleted, or their execution order is altered.

Performance:

1. Using such a tool increment the time to run the tests
2. You can limit mutation testing to only a reduced set of classes
3. You can execute mutation testing only sometimes

Or during nightly builds. (this has to do with continuous integration, that we will see in future lessons)

Mutation test is a structural testing method aimed at evaluating / improving the adequacy of the test suites and the estimate of the number of errors present in systems under test. The mutation test introduces small mutations within our source code. There are several types of mutations:

After introducing the mutation, all tests are launched and the result is observed. Tests on a mutation must fail (i.e. at least 1 test in our suite must fail). If the test does not fail, the mutant is said to have survived, otherwise the mutant has been killed. The higher the percentage of mutants killed, the more effective the tests. The result we want is obviously to kill all the mutants.

DON'T LOOK ON THE COVERAGE CODE, KILL THE MUTANTS

DON'T STARE AT CODE COVERAGE, KILL MUTANTS:

- Suppose we have in our SUT this line

  if (a && b) {...}

- A "mutant" consists of the following mutated line

  if (a || b) {...}

- If at least one test fails, then we say that **"the mutant was killed"**

  Otherwise, "the mutant survived"

  – And that is bad!

- If our tests effectively test the overall behaviour of the SUT then all mutants must be killed!

Automatic tools:

- **They create mutants on the fly**

  – For each mutant they run all the tests

  – And report survived mutants

- The creation of mutants is based on a set of mutation operators, e.g.,
  – Change > to <=
  – Change >= to >
  – Change == to !=
  – Set an initialized field to null
- These mutants aim at spotting typical programming errors.

To convince you of the power of mutation tests I will show you a simple example in java with unit tests written using the junit framework. Suppose we need to implement a very simple class that validates a password entered by the user. The specification is very simple:

null or empty string is not accepted as possible password .

the password must be at least 3 characters long.

Below is a simple java implementation of the described algorithm.

We also propose the PasswordValidator test class.

After introducing the mutation, all tests are launched, and the result is observed. Tests on a mutation must fail (i.e. at least 1 Test in our suite must fail). If the test does not fail, it says the mutant survived or the mutant was killed. The higher the percentage of mutants killed, the more

```java
public class PasswordValidator {

    public PasswordValidator() {
    }

    public boolean isValidPassword(String s) {
        if (s == null) {
            return false;
        }
        if (s.length() == 0) {
            return false;
        }
        if (s.length() >= 3) {
            return true;
        } else {
            return false;
        }
    }
}
```

effective the tests are. the result we want is obviously to kill all mutants. If you are also a Marvel fan, then you will understand the reason for choosing the cover of this article.

note: for those unfamiliar with Junit: Junit is a java framework for the automatic drafting and execution of tests. Work on assertions: trivially compare the result of the method you are testing and the expected value, if they are different the test fails otherwise it passes. As a good rule, a test method is made for each test scenario; therefore a maximum of one assertion per method. So let's now try to run the PasswordValidatorTest tests; the result is the following:.

Green: all tests passed successfully.

100 percentage code coverage: that is, the tests solicited 100 percentage of the PasswordValidator instructions.

Many could naively conclude that 100 percentage of the code coverage guarantees a high quality of the tests in this example, but this is not the case at all: the tests in this example are not robust and I will show you with an example. Suppose that during the implementation of PasswordValidator the developer makes this mistake:

That is to say that the greater operator is mistakenly used instead of the greater equal as a guard. Would our tests have found this bug? Then try to modify the isValidPassword method going to use the> and run the tests with the code coverage. The result is the same.

```
1.  public class PasswordValidatorTest {
2.      private PasswordValidator passwordValidator;
3.
4.      @Before
5.      public void init() {
6.          this.passwordValidator = new PasswordValidator();
7.      }
8.
9.      @Test
10.     public void shouldNotBeNull() {
11.         assertFalse(this.passwordValidator.isValidPassword(null));
12.     }
13.
14.     @Test
15.     public void shouldNotBeEmpty() {
16.         assertFalse(this.passwordValidator.isValidPassword(""));
17.
18.     }
19.
20.     @Test
21.     public void shouldNotBeSmallerThanThree() {
22.         assertFalse(this.passwordValidator.isValidPassword("ab"));
23.     }
24.
25.
26.     @Test
27.     public void shouldBeGreaterOrEqualThanThree() {
28.         assertTrue(this.passwordValidator.isValidPassword("abcd"));
29.     }
```

Green: all tests passed successfully.

100 percentage code coverage.

Assuming that a test is good if it catches a bug, we can conclude that our test is not robust even if it has 100 percentage code coverage. So how can we know that our test suite is not robust? How can we understand how to strengthen it? The answer is one: let's run the mutation test and kill the mutants.

4.3 MUTATION TEST WITH PIT

Manually introducing changes in the code is not a viable practice; there are tools in each language that autonomously create mutants and for each of these they run the test suite to identify how many mutants survive. In java the de facto standard is PIT: http://pitest.org/. PIT applies mutations to the bytecode generated by the compilation and executes the test suite for each mutation. To install it is simple (below I

```
1.    public boolean isValidPassword(String s) {
2.  ...
3.        if (s.length() > 3) {
4.            return true;
5.        } else {
6.            ...
7.        }
8.    }
```

show how to integrate it into your maven project but on the PIT website you will find the documentation to integrate it in Gradle too).

```
1.  <build>
2.    <plugins>
3.      <plugin>
4.        <groupId>org.pitest</groupId>
5.        <artifactId>pitest-maven</artifactId>
6.        <version>1.1.10</version>
7.      </plugin>
8.    </plugins>
9.  </build>
```

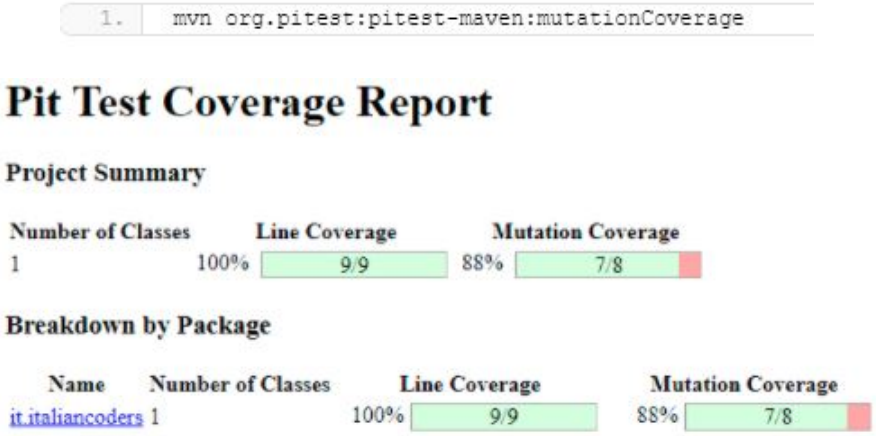
By default, PIT introduces mutations throughout your code. Since the execution of the mutation test is an expensive operation, we can also configure PIT to insert mutations only in a specific package or class.

```
1.  <plugin>
2.      <groupId>org.pitest</groupId>
3.      <artifactId>pitest-maven</artifactId>
4.      <version>LATEST</version>
5.      <configuration>
6.          <targetClasses>
7.              <param>com.your.package.root.want.to.mutate*</param>
8.          </targetClasses>
9.          <targetTests>
10.             <param>com.your.package.root*</param>
11.         </targetTests>
12.     </configuration>
13. </plugin>
```

Let's now go and launch PIT within our example. Using maven just launch the PIT goal within the project folder:

PIT will perform the mutation tests and will create html reports on the result of the latter in target / pit-reports. These reports will be very useful to get a measure of how robust our tests are and any advice on how to strengthen them. Here is the result of our mutation tests.

```
1.  mvn org.pitest:pitest-maven:mutationCoverage
```



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	9/9	88%	7/8

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
it.italiancoders	1	100%	9/9	88%	7/8

As you can see, mutation coverage is telling us that 8 mutants have been created and tests have been performed on them. 7 out of 8 mutants were killed, or at least one test failed. A mutant survived and therefore our test suite is not robust. Let's go into more detail in the report in order to learn more about the surviving mutant:

The detail of the analysis performed by PIT during the mutation test is fantastic:

shows us the types of mutations made.

shows us the tests performed on the mutants.

shows in red the line changed during the mutation test and that did not fail the tests. It also highlights the type of mutation of the surviving mutant in red. In our example, the surviving mutant was born from a mutation of the condition that controls the length of the password. By introducing a decision mutation on the password length check (replacing the equal greater with the narrow greater) our tests do not fail and therefore the mutant survives.

From these reports it can be deduced that the tests are not robust and a possible bug on this condition may not be detected by our unit tests (we already understood this at the beginning of the article when we tried to insert the bug on this condition). How can we fix this gap in our tests? Let's go and strengthen our suite. As is known in the theoretical studies of tests, when you go to write unit tests on a property that is valid within two extremes, the following test scenarios must be provided:

test case in which the property is outside the permitted extremes.

in the event that you have the property within the permitted limits.

PasswordValidator.java

```
1   package it.italiancoders;
2
3   public class PasswordValidator {
4
5       public PasswordValidator() {
6       }
7
8       public boolean isValidPassword(String s) {
9 1         if (s == null) {
10 1            return false;
11          }
12 1        if (s.length() == 0) {
13 1            return false;
14          }
15 2        if (s.length() >= 3) {
16 1            return true;
17          } else {
18 1            return false;
19          }
20      }
21  }
```

Mutations

```
9   1. negated conditional → KILLED
10  1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
12  1. negated conditional → KILLED
13  1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
15  1. changed conditional boundary → SURVIVED
    2. negated conditional → KILLED
16  1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
18  1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
```

case in which you have the property within the permitted extremes, in particular in the minimum and maximum permitted extremes (limit values).

4.4 MOCKING

Mocking is a way to test the functionality of a class in isolation. Mocking does not require a database connection or properties file read or file server read to test a functionality. Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.

4.4.1 *Mockito*

Mockito facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations.

Consider a case of Stock Service which returns the price details of a stock. During development, the actual stock service cannot be used to get real-time data. So we need a dummy implementation of the stock service. Mockito can do the same very easily, as its name suggests.

Benefits of Mockito

1. **No Handwriting** - No need to write mock objects on your own.
2. **Refactoring Safe** - Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
3. **Return value support** - Supports return values.
4. **Exception support** - Supports exceptions.
5. **Order check support** - Supports check on order of method calls.
6. **Annotation support** - Supports creating mocks using annotation.

Lets consider the following Java code.

```
1  package com.tutorialspoint.mock;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import static org.mockito.Mockito.*;
7
8  public class PortfolioTester {
9     public static void main(String args){
10
11       //Create a portfolio object which is to be tested
12       Portfolio portfolio = new Portfolio();
13
14       //Creates a list of stocks to be added to the portfolio
15       List<Stock> stocks = new ArrayList<Stock>();
16       Stock googleStock = new Stock("1","Google", 10);
17       Stock microsoftStock = new Stock("2","Microsoft",100);
18
19       stocks.add(googleStock);
20       stocks.add(microsoftStock);
21
22       //Create the mock object of stock service
```

```
23        StockService stockServiceMock = mock(StockService.class);
24
25        // mock the behavior of stock service to return the value of
      various stocks
26
      when(stockServiceMock.getPrice(googleStock)).thenReturn(50.00);
27
      when(stockServiceMock.getPrice(microsoftStock)).thenReturn(1000.00);
28
29        //add stocks to the portfolio
30        portfolio.setStocks(stocks);
31
32        //set the stockService to the portfolio
33        portfolio.setStockService(stockServiceMock);
34
35        double marketValue = portfolio.getMarketValue();
36
37        //verify the market value to be
38        //10*50.00 + 100* 1000.00 = 500.00 + 100000.00 = 100500
39        System.out.println("Market value of the portfolio: "+
      marketValue);
40     }
41  }
```

Listing 4.9: PortfolioTester.c

Let's understand the important concepts of the above program.

Portfolio -An object to carry a list of stocks and to get the market value computed using stock prices and stock quantity.

Stock -An object to carry the details of a stock such as its id, name, quantity, etc.

StockService -A stock service returns the current price of a stock.

mock(...) -Mockito created a mock of stock service.

when(...).thenReturn(...) -Mock implementation of getPrice method of stockService interface. For googleStock, return 50.00 as price.

portfolio.setStocks(...) -The portfolio now contains a list of two stocks.

portfolio.setStockService(...) -Assigns the stockService Mock object to the portfolio.

portfolio.getMarketValue() -The portfolio returns the market value based on its stocks using the mock stock service.

Before going into the details of the Mockito Framework, let's see an application in action. In this example, we've created a mock of Stock Service to get the dummy price of some stocks and unit tested a java class named Portfolio.

The process is discussed below in a step-by-step manner.

Step 1 -Create a JAVA class to represent the Stock.

File: Stock.java

```
public class Stock {
   private String stockId;
   private String name;
   private int quantity;

   public Stock(String stockId, String name, int quantity){
      this.stockId = stockId;
      this.name = name;
      this.quantity = quantity;
   }

   public String getStockId() {
      return stockId;
   }

   public void setStockId(String stockId) {
      this.stockId = stockId;
   }

   public int getQuantity() {
      return quantity;
   }

   public String getTicker() {
      return name;
   }
}
```

Listing 4.10: Stock.c

Step 2 -Create an interface StockService to get the price of a stock.

File: StockService.java

```
public interface StockService {
   public double getPrice(Stock stock);
}
```

Listing 4.11: StockService.c

Step 3 -Create a class Portfolio to represent the portfolio of any client.

File: Portfolio.java

```
import java.util.List;

public class Portfolio {
   private StockService stockService;
   private List<Stock> stocks;

   public StockService getStockService() {
      return stockService;
   }

   public void setStockService(StockService stockService) {
      this.stockService = stockService;
   }

   public List<Stock> getStocks() {
      return stocks;
   }

   public void setStocks(List<Stock> stocks) {
      this.stocks = stocks;
   }

   public double getMarketValue(){
      double marketValue = 0.0;

      for(Stock stock:stocks){
```

```
27          marketValue += stockService.getPrice(stock) *
    stock.getQuantity();
28       }
29       return marketValue;
30    }
31 }
```

Listing 4.12: Portfolio.c

Step 4 -Test the Portfolio class

Let's test the Portfolio class, by injecting in it a mock of stockservice. Mock will be created by Mockito.

File: PortfolioTester.java

```
1  package com.tutorialspoint.mock;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import static org.mockito.Mockito.*;
7
8  public class PortfolioTester {
9
10    Portfolio portfolio;
11    StockService stockService;
12
13
14    public static void main(String args){
15       PortfolioTester tester = new PortfolioTester();
16       tester.setUp();
17       System.out.println(tester.testMarketValue()?"pass":"fail");
18    }
19
20    public void setUp(){
21       //Create a portfolio object which is to be tested
22       portfolio = new Portfolio();
23
24       //Create the mock object of stock service
25       stockService = mock(StockService.class);
26
27       //set the stockService to the portfolio
28       portfolio.setStockService(stockService);
```

```
29     }
30
31     public boolean testMarketValue(){
32
33        //Creates a list of stocks to be added to the portfolio
34        List<Stock> stocks = new ArrayList<Stock>();
35        Stock googleStock = new Stock("1","Google", 10);
36        Stock microsoftStock = new Stock("2","Microsoft",100);
37
38        stocks.add(googleStock);
39        stocks.add(microsoftStock);
40
41        //add stocks to the portfolio
42        portfolio.setStocks(stocks);
43
44        //mock the behavior of stock service to return the value of
      various stocks
45        when(stockService.getPrice(googleStock)).thenReturn(50.00);
46
      when(stockService.getPrice(microsoftStock)).thenReturn(1000.00);

47
48        double marketValue = portfolio.getMarketValue();
49        return marketValue == 100500.0;
50     }
51  }
```

Listing 4.13: Portfolio-Tester.c

Step 5 -Verify the result
Compile the classes using javac compiler as follows

```
C:\Mockito_WORKSPACE>javac Stock.java StockService.java Portfolio.java PortfolioTester.java
```

Now run the PortfolioTester to see the result.

```
C:\Mockito_WORKSPACE>java PortfolioTester
```

Verify the Output

Pass

Here we will create a Math Application which uses CalculatorService to perform basic mathematical operations such as addition, subtraction, multiply, and division.

We'll use Mockito to mock the dummy implementation of CalculatorService. In addition, we've made extensive use of annotations to showcase their compatibility with both JUnit and Mockito.

The process is discussed below in a step-by-step manner.

Step 1 Create an interface called CalculatorService to provide mathematical functions.

File: CalculatorService.java.

```
1  public interface CalculatorService {
2     public double add(double input1, double input2);
3     public double subtract(double input1, double input2);
4     public double multiply(double input1, double input2);
5     public double divide(double input1, double input2);
6  }
```

Listing 4.14: CalculatorService.c

Step 2 Create a JAVA class to represent MathApplication.

File: MathApplication.java

```
1  public class MathApplication {
2     private CalculatorService calcService;
3
4     public void setCalculatorService(CalculatorService calcService){
5        this.calcService = calcService;
6     }
7
8     public double add(double input1, double input2){
9        return calcService.add(input1, input2);
10    }
11
12    public double subtract(double input1, double input2){
13       return calcService.subtract(input1, input2);
14    }
15
16    public double multiply(double input1, double input2){
17       return calcService.multiply(input1, input2);
```

```
18     }
19
20     public double divide(double input1, double input2){
21        return calcService.divide(input1, input2);
22     }
23  }
```

Listing 4.15: MathApplication.c

Step 3 -Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

File: MathApplicationTester.java

```
1  public class MathApplication {
2     private CalculatorService calcService;
3
4     public void setCalculatorService(CalculatorService calcService){
5        this.calcService = calcService;
6     }
7
8     public double add(double input1, double input2){
9        return calcService.add(input1, input2);
10    }
11
12    public double subtract(double input1, double input2){
13       return calcService.subtract(input1, input2);
14    }
15
16    public double multiply(double input1, double input2){
17       return calcService.multiply(input1, input2);
18    }
19
20    public double divide(double input1, double input2){
21       return calcService.divide(input1, input2);
22    }
23 }
```

Listing 4.16: MathApplication.c

Step 4 -Create a class to execute to test cases

Create a java class file named TestRunner in C> Mockito WORKSPACE to execute Test case(s).

File: TestRunner.java

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String args) {
      Result result =
    JUnitCore.runClasses(MathApplicationTester.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

Listing 4.17: TestRunner.c

Step 5 -Verify the Result

Compile the classes using javac compiler as follows.

```
C:\Mockito_WORKSPACE>javac CalculatorService.java MathApplication.
   java MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result.

```
C:\Mockito_WORKSPACE>java TestRunner
```

Verify the output.

True .

4.5 MOCKITO - ADDING BEHAVIOR

Mockito adds a functionality to a mock object using the methods when(). Take a look at the following code snippet.

```
//add the behavior of calc service to add two numbers
when(calcService.add(10.0,20.0)).thenReturn(30.00);
```

Here we've instructed Mockito to give a behavior of adding 10 and 20 to the add method of calcService and as a result, to return the value of 30.00.

At this point of time, Mock recorded the behavior and is a working mock object.

```
//add the behavior of calc service to add two numbers
when(calcService.add(10.0,20.0)).thenReturn(30.00);
```

Example

Step 1 Create an interface called CalculatorService to provide mathematical functions

File: CalculatorService.java

```
1  public interface CalculatorService {
2     public double add(double input1, double input2);
3     public double subtract(double input1, double input2);
4     public double multiply(double input1, double input2);
5     public double divide(double input1, double input2);
6  }
```

Listing 4.18: CalculatorService1.c

Step 2 Create a JAVA class to represent MathApplication File: MathApplication.java

```
1  public class MathApplication {
2     private CalculatorService calcService;
3
4     public void setCalculatorService(CalculatorService calcService){
5        this.calcService = calcService;
6     }
7
8     public double add(double input1, double input2){
9        return calcService.add(input1, input2);
10    }
11
12    public double subtract(double input1, double input2){
```

```
13       return calcService.subtract(input1, input2);
14    }
15
16    public double multiply(double input1, double input2){
17       return calcService.multiply(input1, input2);
18    }
19
20    public double divide(double input1, double input2){
21       return calcService.divide(input1, input2);
22    }
23 }
```

Listing 4.19: MathApplication.c

Step 3 Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

File: MathApplicationTester.java

```
1  import static org.mockito.Mockito.when;
2
3  import org.junit.Assert;
4  import org.junit.Test;
5  import org.junit.runner.RunWith;
6  import org.mockito.InjectMocks;
7  import org.mockito.Mock;
8  import org.mockito.runners.MockitoJUnitRunner;
9
10 // @RunWith attaches a runner with the test class to initialize the
       test data
11 @RunWith(MockitoJUnitRunner.class)
12 public class MathApplicationTester {
13
14    //@InjectMocks annotation is used to create and inject the mock
       object
15    @InjectMocks
16    MathApplication mathApplication = new MathApplication();
17
18    //@Mock annotation is used to create the mock object to be
       injected
19    @Mock
20    CalculatorService calcService;
```

```
21
22    @Test
23    public void testAdd(){
24       //add the behavior of calc service to add two numbers
25       when(calcService.add(10.0,20.0)).thenReturn(30.00);
26
27       //test the add functionality
28       Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
29    }
30 }
```

Listing 4.20: MathApplicationTester1.c

Step 4 Execute test cases

Create a java class file named TestRunner in C: Mockito WORKSPACE to execute the test case(s).

File: TestRunner.java

```
1  import org.junit.runner.JUnitCore;
2  import org.junit.runner.Result;
3  import org.junit.runner.notification.Failure;
4
5  public class TestRunner {
6     public static void main(String args) {
7        Result result =
       JUnitCore.runClasses(MathApplicationTester.class);
8
9        for (Failure failure : result.getFailures()) {
10          System.out.println(failure.toString());
11       }
12
13       System.out.println(result.wasSuccessful());
14    }
15 }
```

Listing 4.21: TestRunner1.c

```
C:\Mockito_WORKSPACE>javac CalculatorService.java MathApplication.
   java MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result.

```
C:\Mockito_WORKSPACE>java TestRunner

Verify the output.

true
```

4.6 MOCKITO - VERIFYING BEHAVIOR

Mockito can ensure whether a mock method is being called with reequired arguments or not. It is done using the verify() method. Take a look at the following code snippet.

```
//test the add functionality
Assert.assertEquals(calcService.add(10.0, 20.0),30.0,0);

//verify call to calcService is made or not with same arguments.
verify(calcService).add(10.0, 20.0);
```

Example - verify() with same arguments.

Step 1 Create an interface called CalculatorService to provide mathematical functions.

File: CalculatorService.java

```
1 public interface CalculatorService {
2    public double add(double input1, double input2);
3    public double subtract(double input1, double input2);
4    public double multiply(double input1, double input2);
5    public double divide(double input1, double input2);
6 }
```

Listing 4.22: CalculatorService.c

Step 2 Create a JAVA class to represent MathApplication

File: MathApplication.java

```
1 public class MathApplication {
2    private CalculatorService calcService;
3 
4    public void setCalculatorService(CalculatorService calcService){
5       this.calcService = calcService;
6    }
7 
8    public double add(double input1, double input2){
```

```
9        return calcService.add(input1, input2);
10    }
11
12    public double subtract(double input1, double input2){
13       return calcService.subtract(input1, input2);
14    }
15
16    public double multiply(double input1, double input2){
17       return calcService.multiply(input1, input2);
18    }
19
20    public double divide(double input1, double input2){
21       return calcService.divide(input1, input2);
22    }
23 }
```

Listing 4.23: MathApplication.c

Step 3 Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

File: MathApplicationTester.java

```
1  import static org.mockito.Mockito.when;
2
3  import org.junit.Assert;
4  import org.junit.Test;
5  import org.junit.runner.RunWith;
6  import org.mockito.InjectMocks;
7  import org.mockito.Mock;
8  import org.mockito.runners.MockitoJUnitRunner;
9
10 // @RunWith attaches a runner with the test class to initialize the
       test data
11 @RunWith(MockitoJUnitRunner.class)
12 public class MathApplicationTester {
13
14    //@InjectMocks annotation is used to create and inject the mock
       object
15    @InjectMocks
16    MathApplication mathApplication = new MathApplication();
17
```

```
18    //@Mock annotation is used to create the mock object to be
      injected
19    @Mock
20    CalculatorService calcService;
21
22    @Test
23    public void testAdd(){
24       //add the behavior of calc service to add two numbers
25       when(calcService.add(10.0,20.0)).thenReturn(30.00);
26
27       //test the add functionality
28       Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
29    }
30 }
```

Listing 4.24: MathApplicationTester.c

Step 4 Execute test cases

Create a java class file named TestRunner in C: Mockito WORKSPACE to execute Test case(s).

File: TestRunner.java

```
1  import org.junit.runner.JUnitCore;
2  import org.junit.runner.Result;
3  import org.junit.runner.notification.Failure;
4
5  public class TestRunner {
6     public static void main(String args) {
7        Result result =
       JUnitCore.runClasses(MathApplicationTester.class);
8
9        for (Failure failure : result.getFailures()) {
10          System.out.println(failure.toString());
11       }
12
13       System.out.println(result.wasSuccessful());
14    }
15 }
```

Listing 4.25: TestRunner.c

Step 5 Verify the Result Compile the classes using javac compiler as

follows

```
C:\Mockito_WORKSPACE>javac CalculatorService.java MathApplication.
   java MathApplicationTester.java TestRunner.java

Now run the Test Runner to see the result

C:\Mockito_WORKSPACE>java TestRunner
Verify the output.

true
```

Example verify() with different arguments

Step 1 -Create an interface CalculatorService to provide mathematical functions

File: CalculatorService.java

```
1  public interface CalculatorService {
2     public double add(double input1, double input2);
3     public double subtract(double input1, double input2);
4     public double multiply(double input1, double input2);
5     public double divide(double input1, double input2);
6  }
```

Listing 4.26: CalculatorService.c

Step 2 Create a JAVA class to represent MathApplication

File: MathApplication.java

```
1  public class MathApplication {
2     private CalculatorService calcService;
3
4     public void setCalculatorService(CalculatorService calcService){
5        this.calcService = calcService;
6     }
7
8     public double add(double input1, double input2){
9        return calcService.add(input1, input2);
10    }
11
12    public double subtract(double input1, double input2){
13       return calcService.subtract(input1, input2);
14    }
15
```

```
16    public double multiply(double input1, double input2){
17       return calcService.multiply(input1, input2);
18    }
19
20    public double divide(double input1, double input2){
21       return calcService.divide(input1, input2);
22    }
23 }
```

Listing 4.27: MathApplication.c

Step 3 Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

File: MathApplicationTester.java

```
1  import static org.mockito.Mockito.when;
2
3  import org.junit.Assert;
4  import org.junit.Test;
5  import org.junit.runner.RunWith;
6  import org.mockito.InjectMocks;
7  import org.mockito.Mock;
8  import org.mockito.runners.MockitoJUnitRunner;
9
10 // @RunWith attaches a runner with the test class to initialize the
       test data
11 @RunWith(MockitoJUnitRunner.class)
12 public class MathApplicationTester {
13
14    //@InjectMocks annotation is used to create and inject the mock
       object
15    @InjectMocks
16    MathApplication mathApplication = new MathApplication();
17
18    //@Mock annotation is used to create the mock object to be
       injected
19    @Mock
20    CalculatorService calcService;
21
22    @Test
23    public void testAdd(){
```

```
24        //add the behavior of calc service to add two numbers
25        when(calcService.add(10.0,20.0)).thenReturn(30.00);
26
27        //test the add functionality
28        Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
29     }
30  }
```

Listing 4.28: MathApplicationTester.c

Step 4 Execute test cases

Create a java class file named TestRunner in C: Mockito WORKSPACE to execute Test case(s).

File: TestRunner.java

```
1  import org.junit.runner.JUnitCore;
2  import org.junit.runner.Result;
3  import org.junit.runner.notification.Failure;
4
5  public class TestRunner {
6     public static void main(String args) {
7        Result result =
       JUnitCore.runClasses(MathApplicationTester.class);
8
9        for (Failure failure : result.getFailures()) {
10          System.out.println(failure.toString());
11       }
12
13       System.out.println(result.wasSuccessful());
14    }
15 }
```

Listing 4.29: TestRunner.c

4.7 MOCKITO - EXPECTING CALLS

Mockito provides a special check on the number of calls that can be made on a particular method. Suppose MathApplication should call the CalculatorService.serviceUsed() method only once, then it should not be able to call CalculatorService.serviceUsed() more than once.

Create CalculatorService interface as follows.

```
Step 5 - Verify the Result

Compile the classes using javac compiler as follows -

C:\Mockito_WORKSPACE>javac CalculatorService.java MathApplication.
   java MathApplicationTester.java TestRunner.java

Now run the Test Runner to see the result -

C:\Mockito_WORKSPACE>java TestRunner
Verify the output.

testAdd(MathApplicationTester):
Argument(s) are different! Wanted:
calcService.add(20.0, 30.0);
-> at MathApplicationTester.testAdd(MathApplicationTester.java:32)
Actual invocation has different arguments:
calcService.add(10.0, 20.0);
-> at MathApplication.add(MathApplication.java:10)

false

//add the behavior of calc service to add two numbers
when(calcService.add(10.0,20.0)).thenReturn(30.00);

//limit the method call to 1, no less and no more calls are allowed
verify(calcService, times(1)).add(10.0, 20.0);
```

File: CalculatorService2.java

```
1 public interface CalculatorService {
2    public double add(double input1, double input2);
3    public double subtract(double input1, double input2);
4    public double multiply(double input1, double input2);
5    public double divide(double input1, double input2);
6 }
```

Listing 4.30: CalculatorService2.c

Example

Step 1 Create an interface called CalculatorService to provide mathematical functions

File: CalculatorService2.java

```
1 public interface CalculatorService {
2    public double add(double input1, double input2);
3    public double subtract(double input1, double input2);
4    public double multiply(double input1, double input2);
```

```
5    public double divide(double input1, double input2);
6 }
```

Listing 4.31: CalculatorService2.c

Step 2 Create a JAVA class to represent MathApplication

File: MathApplication2.java

```
1  public class MathApplication {
2     private CalculatorService calcService;
3
4     public void setCalculatorService(CalculatorService calcService){
5        this.calcService = calcService;
6     }
7
8     public double add(double input1, double input2){
9        return calcService.add(input1, input2);
10    }
11
12    public double subtract(double input1, double input2){
13       return calcService.subtract(input1, input2);
14    }
15
16    public double multiply(double input1, double input2){
17       return calcService.multiply(input1, input2);
18    }
19
20    public double divide(double input1, double input2){
21       return calcService.divide(input1, input2);
22    }
23 }
```

Listing 4.32: MathApplication2.c

Step 3 Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

File: MathApplicationTester2.java

```
1  import static org.mockito.Mockito.verify;
2  import static org.mockito.Mockito.when;
3  import static org.mockito.Mockito.times;
```

```
4  import static org.mockito.Mockito.never;
5
6  import org.junit.Assert;
7  import org.junit.Test;
8  import org.junit.runner.RunWith;
9  import org.mockito.InjectMocks;
10 import org.mockito.Mock;
11 import org.mockito.runners.MockitoJUnitRunner;
12
13 // @RunWith attaches a runner with the test class to initialize the
       test data
14 @RunWith(MockitoJUnitRunner.class)
15 public class MathApplicationTester {
16
17    //@InjectMocks annotation is used to create and inject the mock
       object
18    @InjectMocks
19    MathApplication mathApplication = new MathApplication();
20
21    //@Mock annotation is used to create the mock object to be
       injected
22    @Mock
23    CalculatorService calcService;
24
25    @Test
26    public void testAdd(){
27       //add the behavior of calc service to add two numbers
28       when(calcService.add(10.0,20.0)).thenReturn(30.00);
29
30       //add the behavior of calc service to subtract two numbers
31       when(calcService.subtract(20.0,10.0)).thenReturn(10.00);
32
33       //test the add functionality
34       Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
35       Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
36       Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
37
38       //test the subtract functionality
39       Assert.assertEquals(mathApplication.subtract(20.0,
      10.0),10.0,0.0);
40
41       //default call count is 1
```

```
42       verify(calcService).subtract(20.0, 10.0);
43
44       //check if add function is called three times
45       verify(calcService, times(3)).add(10.0, 20.0);
46
47       //verify that method was never called on a mock
48       verify(calcService, never()).multiply(10.0,20.0);
49    }
50 }
```

Listing 4.33: MathApplicationTester2.c

Step 4 Execute test cases

Create a java class file named TestRunner in C Mockito WORKSPACE to execute Test case(s).

File: TestRunner.java

```
1  import org.junit.runner.JUnitCore;
2  import org.junit.runner.Result;
3  import org.junit.runner.notification.Failure;
4
5  public class TestRunner {
6     public static void main(String args) {
7        Result result =
      JUnitCore.runClasses(MathApplicationTester.class);
8
9        for (Failure failure : result.getFailures()) {
10          System.out.println(failure.toString());
11       }
12       System.out.println(result.wasSuccessful());
13    }
14 }
```

Listing 4.34: TestRunner2.c

4.8 MOCKITO - VARYING CALLS

Mockito provides the following additional methods to vary the expected call counts.

atLeast (int min) expects min calls.

```
Step 5 - Verify the Result

Compile the classes using javac compiler as follows -

C:\Mockito_WORKSPACE>javac CalculatorService.java MathApplication.
   java MathApplicationTester.java TestRunner.java

Now run the Test Runner to see the result -

C:\Mockito_WORKSPACE>java TestRunner
Verify the output.

true
```

atLeastOnce () expects at least one call.

atMost (int max) expects max calls.

Example

Step 1 Create an interface CalculatorService to provide mathematical functions

File: CalculatorService4.java

```
1  public interface CalculatorService4 {
2     public double add(double input1, double input2);
3     public double subtract(double input1, double input2);
4     public double multiply(double input1, double input2);
5     public double divide(double input1, double input2);
6  }
```

Listing 4.35: CalculatorService4.c

Step 2 Create a JAVA class to represent MathApplication

File: MathApplication4.java

```
1  public class MathApplication4 {
2     private CalculatorService calcService;
3
4     public void setCalculatorService(CalculatorService calcService){
5        this.calcService = calcService;
6     }
7
8     public double add(double input1, double input2){
9        return calcService.add(input1, input2);
10    }
```

```
11
12    public double subtract(double input1, double input2){
13       return calcService.subtract(input1, input2);
14    }
15
16    public double multiply(double input1, double input2){
17       return calcService.multiply(input1, input2);
18    }
19
20    public double divide(double input1, double input2){
21       return calcService.divide(input1, input2);
22    }
23 }
```

Listing 4.36: MathApplication4.c

Step 3 Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

File: MathApplicationTester4.java

```
1  import static org.mockito.Mockito.verify;
2  import static org.mockito.Mockito.when;
3  import static org.mockito.Mockito.atLeastOnce;
4  import static org.mockito.Mockito.atLeast;
5  import static org.mockito.Mockito.atMost;
6
7  import org.junit.Assert;
8  import org.junit.Test;
9  import org.junit.runner.RunWith;
10 import org.mockito.InjectMocks;
11 import org.mockito.Mock;
12 import org.mockito.runners.MockitoJUnitRunner;
13
14 // @RunWith attaches a runner with the test class to initialize the
       test data
15 @RunWith(MockitoJUnitRunner.class)
16 public class MathApplicationTester4 {
17
18    //@InjectMocks annotation is used to create and inject the mock
       object
19    @InjectMocks
```

```
20    MathApplication mathApplication = new MathApplication();
21
22    //@Mock annotation is used to create the mock object to be
       injected
23    @Mock
24    CalculatorService calcService;
25
26    @Test
27    public void testAdd(){
28       //add the behavior of calc service to add two numbers
29       when(calcService.add(10.0,20.0)).thenReturn(30.00);
30
31       //add the behavior of calc service to subtract two numbers
32       when(calcService.subtract(20.0,10.0)).thenReturn(10.00);
33
34       //test the add functionality
35       Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
36       Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
37       Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
38
39       //test the subtract functionality
40       Assert.assertEquals(mathApplication.subtract(20.0,
       10.0),10.0,0.0);
41
42       //check a minimum 1 call count
43       verify(calcService, atLeastOnce()).subtract(20.0, 10.0);
44
45       //check if add function is called minimum 2 times
46       verify(calcService, atLeast(2)).add(10.0, 20.0);
47
48       //check if add function is called maximum 3 times
49       verify(calcService, atMost(3)).add(10.0,20.0);
50    }
51 }
```

Listing 4.37: MathApplicationTester4.c

Step 4 Execute test cases

Create a java class file named TestRunner in C: Mockito WORKSPACE to execute Test case(s)

File: TestRunner4.java

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner4 {
   public static void main(String args) {
      Result result =
    JUnitCore.runClasses(MathApplicationTester.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

Listing 4.38: TestRunner4.c

```
Step 5 - Verify the Result

Compile the classes using javac compiler as follows -

C:\Mockito_WORKSPACE>javac CalculatorService4.java MathApplication4.
   java MathApplicationTester4.java TestRunner4.java

Now run the Test Runner to see the result -

C:\Mockito_WORKSPACE>java TestRunner4
Verify the output.

true
```

4.9 MOCKITO - EXCEPTION HANDLING

Mockito provides the capability to a mock to throw exceptions, so exception handling can be tested. Take a look at the following code snippet.

```
//add the behavior to throw exception
doThrow(new Runtime Exception("divide operation not implemented"))
   .when(calcService).add(10.0,20.0);
```

Here we've added an exception clause to a mock object. MathApplication makes use of calcService using its add method and the mock throws a RuntimeException whenever calcService.add() method is invoked.

Example

Step 1 Create an interface called CalculatorService to provide mathematical functions

File: CalculatorService5.java

```
public interface CalculatorService5 {
   public double add(double input1, double input2);
   public double subtract(double input1, double input2);
   public double multiply(double input1, double input2);
   public double divide(double input1, double input2);
}
```

Listing 4.39: CalculatorService5.c

Step 2 Create a JAVA class to represent MathApplication

File: MathApplication5.java

```
public class MathApplication5 {
   private CalculatorService calcService;

   public void setCalculatorService(CalculatorService calcService){
      this.calcService = calcService;
   }

   public double add(double input1, double input2){
      return calcService.add(input1, input2);
   }

   public double subtract(double input1, double input2){
      return calcService.subtract(input1, input2);
   }

   public double multiply(double input1, double input2){
      return calcService.multiply(input1, input2);
   }

   public double divide(double input1, double input2){
      return calcService.divide(input1, input2);
```

```
22    }
23 }
```

Listing 4.40: MathApplication5.c

Step 3 Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

File: MathApplicationTester5.java

```
1  import static org.mockito.Mockito.doThrow;
2
3  import org.junit.Assert;
4  import org.junit.Test;
5  import org.junit.runner.RunWith;
6  import org.mockito.InjectMocks;
7  import org.mockito.Mock;
8  import org.mockito.runners.MockitoJUnitRunner;
9
10 // @RunWith attaches a runner with the test class to initialize the
       test data
11 @RunWith(MockitoRunner.class)
12 public class MathApplicationTester5 {
13
14    // @TestSubject annotation is used to identify class
15       which is going to use the mock object
16    @TestSubject
17    MathApplication mathApplication = new MathApplication();
18
19    //@Mock annotation is used to create the mock object to be
       injected
20    @Mock
21    CalculatorService calcService;
22
23    @Test(expected = RuntimeException.class)
24    public void testAdd(){
25       //add the behavior to throw exception
26       doThrow(new RuntimeException("Add operation not implemented"))
27          .when(calcService).add(10.0,20.0);
28
29       //test the add functionality
30       Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
```

```
31    }
32 }
```

Listing 4.41: MathApplicationTester5.c

Step 4 Execute test cases

Create a java class file named TestRunner in C: Mockito WORKSPACE to execute Test case(s).

File: TestRunner5.java

```
1  import org.junit.runner.JUnitCore;
2  import org.junit.runner.Result;
3  import org.junit.runner.notification.Failure;
4
5  public class TestRunner5 {
6     public static void main(String args) {
7        Result result =
       JUnitCore.runClasses(MathApplicationTester.class);
8
9        for (Failure failure : result.getFailures()) {
10          System.out.println(failure.toString());
11       }
12
13       System.out.println(result.wasSuccessful());
14    }
15 }
```

Listing 4.42: TestRunner5.c

```
Step 5 - Verify the Result

Compile the classes using javac compiler as follows -

C:\Mockito_WORKSPACE>javac CalculatorService5.java MathApplication5.
   java MathApplicationTester5.java TestRunner5.java

Now run the Test Runner to see the result -

C:\Mockito_WORKSPACE>java TestRunner5
Verify the output.

testAdd(MathApplicationTester5): Add operation not implemented

false
```

4.10 MOCKITO - CREATE MOCK

So far, we've used annotations to create mocks. Mockito provides various methods to create mock objects. mock() creates mocks without bothering about the order of method calls that the mock is going to make in due course of its action.

Syntax

```
calcService = mock(CalculatorService.class);
```

Example

Step 1 Create an interface called CalculatorService to provide mathematical functions

File: CalculatorService6.java

```
1  public interface CalculatorService6 {
2     public double add(double input1, double input2);
3     public double subtract(double input1, double input2);
4     public double multiply(double input1, double input2);
5     public double divide(double input1, double input2);
6  }
```

Listing 4.43: CalculatorService6.c

Step 2 Create a JAVA class to represent MathApplication

File: MathApplication6.java

```
1  public class MathApplication6 {
2     private CalculatorService calcService;
3
4     public void setCalculatorService(CalculatorService calcService){
5        this.calcService = calcService;
6     }
7
8     public double add(double input1, double input2){
9        return calcService.add(input1, input2);
10    }
11
12    public double subtract(double input1, double input2){
13       return calcService.subtract(input1, input2);
14    }
```

```
15
16    public double multiply(double input1, double input2){
17       return calcService.multiply(input1, input2);
18    }
19
20    public double divide(double input1, double input2){
21       return calcService.divide(input1, input2);
22    }
23 }
```

Listing 4.44: MathApplication6 .c

Step 3 Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

Here we've added two mock method calls, add() and subtract(), to the mock object via when(). However during testing, we've called subtract() before calling add(). When we create a mock object using create(), the order of execution of the method does not matter.

File: MathApplicationTester6.java

```
1  package com.tutorialspoint.mock;
2
3  import static org.mockito.Mockito.mock;
4  import static org.mockito.Mockito.verify;
5  import static org.mockito.Mockito.when;
6
7  import org.junit.Assert;
8  import org.junit.Before;
9  import org.junit.Test;
10 import org.junit.runner.RunWith;
11 import org.mockito.runners.MockitoJUnitRunner;
12
13 // @RunWith attaches a runner with the test class to initialize the
       test data
14 @RunWith(MockitoJUnitRunner.class)
15 public class MathApplicationTester6 {
16
17    private MathApplication mathApplication;
18    private CalculatorService calcService;
19
20    @Before
```

```
21    public void setUp(){
22       mathApplication = new MathApplication();
23       calcService = mock(CalculatorService.class);
24       mathApplication.setCalculatorService(calcService);
25    }
26
27    @Test
28    public void testAddAndSubtract(){
29
30       //add the behavior to add numbers
31       when(calcService.add(20.0,10.0)).thenReturn(30.0);
32
33       //subtract the behavior to subtract numbers
34       when(calcService.subtract(20.0,10.0)).thenReturn(10.0);
35
36       //test the subtract functionality
37       Assert.assertEquals(mathApplication.subtract(20.0,
     10.0),10.0,0);
38
39       //test the add functionality
40       Assert.assertEquals(mathApplication.add(20.0, 10.0),30.0,0);
41
42       //verify call to calcService is made or not
43       verify(calcService).add(20.0,10.0);
44       verify(calcService).subtract(20.0,10.0);
45    }
46 }
```

Listing 4.45: MathApplicationTester6.c

Step 4 Execute test cases

Create a java class file named TestRunner in C: Mockito WORKSPACE to execute Test case(s).

File: TestRunner6.java

4.11 MOCKITO - ORDERED VERIFICATION

Mockito provides Inorder class which takes care of the order of method calls that the mock is going to make in due course of its action.

Example

```
Step 5 - Verify the Result

Compile the classes using javac compiler as follows -

C:\Mockito_WORKSPACE>javac CalculatorService6.java MathApplication6.
   java MathApplicationTester6.java TestRunner6.java

Now run the Test Runner to see the result -

C:\Mockito_WORKSPACE>java TestRunner6
Verify the output.

true
```

```
Syntax

//create an inOrder verifier for a single mock
InOrder inOrder = inOrder(calcService);

//following will make sure that add is first called then subtract is called.
inOrder.verify(calcService).add(20.0,10.0);
inOrder.verify(calcService).subtract(20.0,10.0);
```

Step 1 Create an interface called CalculatorService to provide mathematical functions

File: CalculatorService7.java

```
1 public interface CalculatorService7 {
2    public double add(double input1, double input2);
3    public double subtract(double input1, double input2);
4    public double multiply(double input1, double input2);
5    public double divide(double input1, double input2);
6 }
```

Listing 4.46: CalculatorService7.c

Step 2 Create a JAVA class to represent MathApplication

File: MathApplication7.java

```
1 public class MathApplication7 {
2    private CalculatorService calcService;
3
4    public void setCalculatorService(CalculatorService calcService){
5       this.calcService = calcService;
```

```
6     }
7
8     public double add(double input1, double input2){
9        return calcService.add(input1, input2);
10    }
11
12    public double subtract(double input1, double input2){
13       return calcService.subtract(input1, input2);
14    }
15
16    public double multiply(double input1, double input2){
17       return calcService.multiply(input1, input2);
18    }
19
20    public double divide(double input1, double input2){
21       return calcService.divide(input1, input2);
22    }
23 }
```

Listing 4.47: MathApplication7.c

Step 3 Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

Here we've added two mock method calls, add() and subtract(), to the mock object via when(). However during testing, we've called subtract() before calling add(). When we create a mock object using Mockito, the order of execution of the method does not matter. Using InOrder class, we can ensure call order.

File: MathApplicationTester7.java

```
1  import static org.mockito.Mockito.mock;
2  import static org.mockito.Mockito.verify;
3  import static org.mockito.Mockito.when;
4  import static org.mockito.Mockito.inOrder;
5
6  import org.junit.Assert;
7  import org.junit.Before;
8  import org.junit.Test;
9  import org.junit.runner.RunWith;
10 import org.mockito.InOrder;
```

```
11 import org.mockito.runners.MockitoJUnitRunner;
12
13 // @RunWith attaches a runner with the test class to initialize the
       test data
14 @RunWith(MockitoJUnitRunner.class)
15 public class MathApplicationTester7 {
16
17    private MathApplication mathApplication;
18    private CalculatorService calcService;
19
20    @Before
21    public void setUp(){
22       mathApplication = new MathApplication();
23       calcService = mock(CalculatorService.class);
24       mathApplication.setCalculatorService(calcService);
25    }
26
27    @Test
28    public void testAddAndSubtract(){
29
30       //add the behavior to add numbers
31       when(calcService.add(20.0,10.0)).thenReturn(30.0);
32
33       //subtract the behavior to subtract numbers
34       when(calcService.subtract(20.0,10.0)).thenReturn(10.0);
35
36       //test the add functionality
37       Assert.assertEquals(mathApplication.add(20.0, 10.0),30.0,0);
38
39       //test the subtract functionality
40       Assert.assertEquals(mathApplication.subtract(20.0,
       10.0),10.0,0);
41
42       //create an inOrder verifier for a single mock
43       InOrder inOrder = inOrder(calcService);
44
45       //following will make sure that add is first called then
       subtract is called.
46       inOrder.verify(calcService).subtract(20.0,10.0);
47       inOrder.verify(calcService).add(20.0,10.0);
48
49    }
```

```
50 }
```

Listing 4.48: MathApplicationTester7.c

Step 4 Execute test cases

Create a java class file named TestRunner in C: Mockito WORKSPACE to execute Test case(s).

File: TestRunner7.java

```
1  import org.junit.runner.JUnitCore;
2  import org.junit.runner.Result;
3  import org.junit.runner.notification.Failure;
4
5  public class TestRunner7 {
6     public static void main(String args) {
7        Result result =
       JUnitCore.runClasses(MathApplicationTester.class);
8
9        for (Failure failure : result.getFailures()) {
10          System.out.println(failure.toString());
11       }
12
13       System.out.println(result.wasSuccessful());
14    }
15 }
```

Listing 4.49: TestRunner7.c

4.12 MOCKITO - CALLBACKS

Mockito provides a Answer interface which allows stubbing with generic interface.

Example

Step 1 Create an interface called CalculatorService to provide mathematical functions

File: CalculatorService8.java

```
1  public interface CalculatorService8 {
2     public double add(double input1, double input2);
3     public double subtract(double input1, double input2);
```

```
Step 5 - Verify the Result

Compile the classes using javac compiler as follows -

C:\Mockito_WORKSPACE>javac CalculatorService7.java MathApplication7.
   java MathApplicationTester7.java TestRunner7.java

Now run the Test Runner to see the result -

C:\Mockito_WORKSPACE>java TestRunner7
Verify the output.

testAddAndSubtract(MathApplicationTester7):
Verification in order failure
Wanted but not invoked:
calculatorService.add(20.0, 10.0);
-> at MathApplicationTester.testAddAndSubtract(MathApplicationTester7.java:48)
Wanted anywhere AFTER following interaction:
calculatorService.subtract(20.0, 10.0);
-> at MathApplication.subtract(MathApplication7.java:13)

false
```

```
4     public double multiply(double input1, double input2);
5     public double divide(double input1, double input2);
6  }
```

Listing 4.50: CalculatorService8.c

Step 2 Create a JAVA class to represent MathApplication

File: MathApplication8.java

```
1  public class MathApplication8 {
2     private CalculatorService calcService;
3
4     public void setCalculatorService(CalculatorService calcService){
5        this.calcService = calcService;
6     }
7
8     public double add(double input1, double input2){
9        return calcService.add(input1, input2);
10    }
11
12    public double subtract(double input1, double input2){
```

```
Syntax

//add the behavior to add numbers
when(calcService.add(20.0,10.0)).thenAnswer(new Answer<Double>() {
   @Override
   public Double answer(InvocationOnMock invocation) throws Throwable {
      //get the arguments passed to mock
      Object[] args = invocation.getArguments();
      //get the mock
      Object mock = invocation.getMock();
      //return the result
      return 30.0;
   }
});
```

```
13       return calcService.subtract(input1, input2);
14    }
15
16    public double multiply(double input1, double input2){
17       return calcService.multiply(input1, input2);
18    }
19
20    public double divide(double input1, double input2){
21       return calcService.divide(input1, input2);
22    }
23 }
```

Listing 4.51: MathApplication8.c

Step 3 Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

Here we've added one mock method calls, add() to the mock object via when(). However during testing, we've called subtract() before calling add(). When we create a mock object using Mockito.createStrictMock(), the order of execution of the method does matter.

File: MathApplicationTester8.java

```
1 import static org.mockito.Mockito.mock;
2 import static org.mockito.Mockito.verify;
3 import static org.mockito.Mockito.when;
4 import static org.mockito.Mockito.inOrder;
```

```
5
6  import org.junit.Assert;
7  import org.junit.Before;
8  import org.junit.Test;
9  import org.junit.runner.RunWith;
10 import org.mockito.InOrder;
11 import org.mockito.runners.MockitoJUnitRunner;
12
13 // @RunWith attaches a runner with the test class to initialize the
       test data
14 @RunWith(MockitoJUnitRunner.class)
15 public class MathApplicationTester8 {
16
17    private MathApplication mathApplication;
18    private CalculatorService calcService;
19
20    @Before
21    public void setUp(){
22       mathApplication = new MathApplication();
23       calcService = mock(CalculatorService.class);
24       mathApplication.setCalculatorService(calcService);
25    }
26
27    @Test
28    public void testAdd(){
29
30       //add the behavior to add numbers
31       when(calcService.add(20.0,10.0)).thenAnswer(new
       Answer<Double>() {
32
33          @Override
34          public Double answer(InvocationOnMock invocation) throws
       Throwable {
35             //get the arguments passed to mock
36             Object args = invocation.getArguments();
37
38             //get the mock
39             Object mock = invocation.getMock();
40
41             //return the result
42             return 30.0;
43          }
```

```
44        });
45
46        //test the add functionality
47        Assert.assertEquals(mathApplication.add(20.0, 10.0),30.0,0);
48     }
49  }
```

Listing 4.52: MathApplicationTester8.c

Step 4 Execute test cases

Create a java class file named TestRunner in C: Mockito WORKSPACE to execute Test case(s).

File: TestRunner8.java

```
1  import static org.mockito.Mockito.mock;
2  import static org.mockito.Mockito.verify;
3  import static org.mockito.Mockito.when;
4  import static org.mockito.Mockito.inOrder;
5
6  import org.junit.Assert;
7  import org.junit.Before;
8  import org.junit.Test;
9  import org.junit.runner.RunWith;
10 import org.mockito.InOrder;
11 import org.mockito.runners.MockitoJUnitRunner;
12 import org.junit.runner.JUnitCore;
13 import org.junit.runner.Result;
14 import org.junit.runner.notification.Failure;
15
16 public class TestRunner8 {
17    public static void main(String args) {
18       Result result =
     JUnitCore.runClasses(MathApplicationTester.class);
19
20       for (Failure failure : result.getFailures()) {
21          System.out.println(failure.toString());
22       }
23
24       System.out.println(result.wasSuccessful());
25    }
26 }
```

Listing 4.53: TestRunner8.c

Step 5 - Verify the Result

Compile the classes using javac compiler as follows -

```
C:\Mockito_WORKSPACE>javac CalculatorService8.java MathApplication8.
   java MathApplicationTester8.java TestRunner8.java
```

Now run the Test Runner to see the result -

```
C:\Mockito_WORKSPACE>java TestRunner8
```

Verify the output.

```
true
```

5

PART 3

5.1 MAVEN:

So far we have always used Eclipse to compile and run tests. We also used Eclipse for manage the dependencies of the project by modifying the project classpath.

When we needed additional libraries (i.e. JARs) we turned to Eclipse. If the library is part of Eclipse itself, like JUnit, is quite simple. Otherwise we had to manually download the JARs from the file Internet and add them to our projects in Eclipse by setting the classpath as well. Adding JAR to the file classpath is easy with Eclipse anyway. The most difficult problem with the JAR download process deals with transitive dependencies, that is, addictions of dependencies (recursively).

Dealing with transitive addictions could become a nightmare, as they need to be fixed manually. First you need to know the external JARs needed by the libraries you want to use. Then, you have to download them too, until all dependencies are resolved and available in your file project.

So far we have been lucky as external JARs had no further dependencies (or if they did, these were part of the JAR). In any case, this manual approach requires some effort. Remember that for Log4j we had to download a zip file, unzip the file and then take only the JARs we actually have necessary.

We have chosen to add the JARs directly into the project itself. When working on a team, the fact that external libraries are part of the project which relieves other developers from doing all of the above manual steps repeatedly each time they want to set up the development environment. However, Having external libraries as part of your code base is also not ideal.

In the JUnit Chapter, Section Keeping Test Code Separate From Main Code, we showed the importance to keep the main code separate from the test code and that this separation should also cover the corresponding dependencies. In the section Export an executable JAR we also saw how to export a JAR with all the addictions. After separating the main code from the test code and the corresponding dependencies, the jar will contain only the external libraries needed by the main code, excluding dependencies needed only for testing.

Packing dependencies in our JARs makes sense if we want to provide an executable JAR, which it is autonomous. However, if we want to distribute a library, including dependencies in the JAR it could be overwhelming. For libraries, we should therefore document the necessary dependencies, since the users of our JAR will need to add our library dependencies to the classpath as well. Again, this it is cumbersome for our library users.

Either way, dealing with addictions manually is effort and error prone Not to mention which does not fit complex programs with many dependencies.

We will soon deal with “Continuous Integration servers”, Chapter Continuous Integration, which automatically.

- Compile the entire program.
- Run all tests.
- Track possible problems with reports on:

- Compilation errors.

- Failed tests.

- Code coverage.

- Quality of the code.

Continuous integration relies on a build automation tool, which is a capable "command line" tool to automatically perform all the above tasks. In particular, such a tool should mimic what it usually does do in Eclipse but automatically:

- Download all project dependencies
- Compile the entire application
- Run all tests

- Generate reports

Build automation and Continuous Integration is a requirement even for medium-sized projects. when there are many tests and some of them are integration tests, which take much longer to run compared to unit tests, during a TDD cycle, a developer only runs unit tests that affect the current SUT. A single developer can also run all application unit tests, as they are fast, while all other long-running tests are performed by the Continuous Integration server.

Maven comes to help us in all the above issues, with the aim of making all the above processes automatic possible, starting from a configuration file that defines both the dependencies of the project and the file build automation infrastructures.

5.1.1 *Maven: Installation*

Maven is a command line Java program. It is available for any platform that is able to execute the JVM.

It can be downloaded from https://maven.apache.org/download.cgi. It is just a matter of extracting the archive somewhere in your system and of setting the system PATH accordingly. Alternatively, you may want to use SdkMan also for installing Maven.

Maven relies on the Java JDK, which must be already installed in your system.

Eclipse comes with very nice support for Maven, in the shape of the M2Eclipse project (m2e for short), https://www.eclipse.org/m2e/. This plugin is already part of most Eclipse distributions, including the one for Java developers. Otherwise it can be installed from the update site that can be found in the above project's web page.

M2Eclipse comes with Maven binaries, so you do not need to manually install Maven in your system if you plan to run Maven builds only from Eclipse and not manually from the command line. The main features of M2Eclipse project are summarized in its main web page:

- Starting Maven builds from within Eclipse.
- Dependency management for pom.xml based Eclipse build path.
- Resolve Maven dependencies from the Eclipse workspace without installing locally Maven Repository.

• Automatic download of required dependencies from remote Maven repositories.

• Wizards for creating new Maven projects, pom.xml and for enabling Maven support on plain Java project.

5.1.2 *Import a Maven project in Eclipse*

A Maven project is configured in the pom.xml file. Obviously there is no trace of Eclipse project metadata, therefore, Eclipse cannot directly open a Maven project of this type. However, thanks to the M2E plugin, Eclipse is able to import the Maven project and automatically derive e create your own project metadata. ccc

Select File → Impor → Maven → Existing Maven Projects and specify the root path of the file project created through the above archetype. The specified path can also contain several Mavens projects and Eclipse can import them all. Once the root path is specified, the detected projects are displayed for selection. Make sure the one we created is selected and hit "Finish".

Once the import terminates, Eclipse will have created the project metadata .project and .classpath, together with the directory .settings with some settings for the Java compiler and the encoding of the resources. All these files are derived from the pom.xml file.

Since this is probably the first time you are using Maven, you will note also that some JARs are downloaded from the Internet after the project has been imported. We'll get back to that later. Note that the name of the Eclipse project, by default, will be the same as the artifactId specified when creating the project through the archetype..

5.1.3 *Create a Maven project from Eclipse using an archetype*

You can also create a Maven project starting from an archetype directly from Eclipse. Obviously, Eclipse will also automatically import the project into the workspace and create its metadata, just like happened in the previous section.

Select File → New Project ... → Maven → Maven Project and choose a different can location for the created project. Make sure the checkbox

to skip archetype selection is unchecked and press "Next". The first time, it must be the huge list of archetypes available downloaded from the Internet. Use the "Filter:" box and type maven-archetype-quickstart. Make make sure to uncheck "Show only the latest version of Archetype" and choose the one with "Group ID" org.apache.maven.archetypes and "Version" 1.1 (the same one we used when we created the project from the command line). Then provide groupId and artifactId. If you imported into Eclipse the project created from the command line must provide a different artifactId. In fact, like seen in the previous section, by default, M2E creates an Eclipse project with the same name as the file artifactId.

In general, you can specify a different schema for the Eclipse project's name by selecting a different "Name template". For example, in the following screenshot we chose com.mycompany.app for the groupId and myapp as the artifactId, but we selected the template "[groupId].[artifactId]" so that the Eclipse project's name will be com.mycompany.app.myapp, instead of the default myapp. Note that you can also specify the Java package name for the created sample Java files.

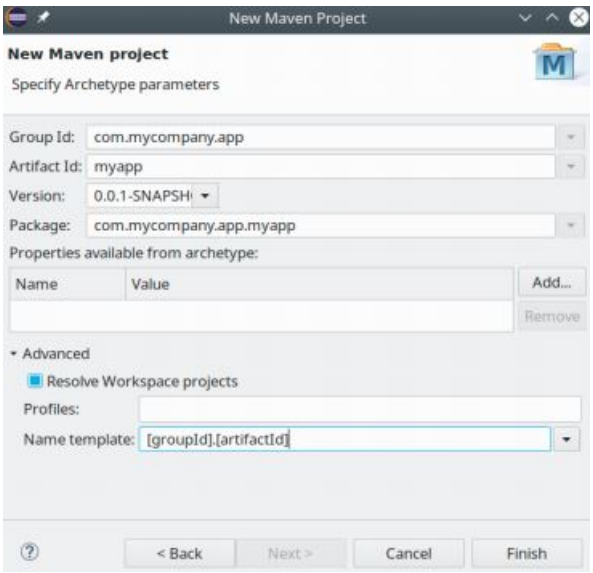


Figure 3: Create Maven project

Press Finish and verify that the project has been created with the expected name and imported in Eclipse.

5.1.4 *Java settings*

At the time of writing, by default, a Maven project is automatically configured to be compatible with Java 5. This default is automatically used again by Eclipse to configure project metadata. In fact, both in the imported project and in the one created by Eclipse, you can see the Java setting J2SE-1.5. This is also reflected in the Java compiler level of the project: with the current settings, you can only use the features present in Java 5.

Once the Eclipse project metadata is created, Eclipse does not automatically preserve the project metadata and settings in the pom.xml in sync.

Therefore, you can change the Java compilation setting for the Eclipse project through its properties (right click on the project and select Properties → Java Build Path and in the "Libraries" tab edit the "JRE System Library"), but then, when we later want to build the Java project with Maven, the default Java 5 build settings will be used.

The solution is to do the opposite. So, first let's change pom.xml to enable Java 8 compiler (remember we're already using JDK 8) by adding the following two properties in the pom.xml file (we will see the Maven properties in more details in the Properties section):

```
1 <properties>
2   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3   <!-- Change the Java compilation level to 8 -->
4   <maven.compiler.source>1.8</maven.compiler.source>
5   <maven.compiler.target>1.8</maven.compiler.target>
6 </properties>
```

Figure 4: Java Setting

And then we force Eclipse to update the project metadata from the modified pom.xml: right click on the project and select Maven → Update project ..., in the dialog box make sure the project you want to update is selected and the "Update project configuration from pom.xml" checkbox is selected. You will see that Eclipse has updated the project metadata by setting up the Java compilation level a 8, JavaSE-1.8.

5.1.5 *Dependencies: POM*

Project dependencies are specified in the pom.xml in the root level dependencies section, each in a separate dependency element. Each of these elements must provide the coordinates and the scope of addiction.

Scope has an impact on the classpath where the dependency will be available and on transitivity of dependence.

By default, the scope is set to compile - the dependency will be available in all classpaths of a project and will be propagated to dependent projects. This means that the addiction will be transitive dependence on your project. The test scope indicates a dependency that is not required for the application itself, but only for testing: it will only be available for building the test and stages of execution. Furthermore, this realm is not transitive.

For example, in the project pom.xml we created using the archetype we have

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Figure 5: Dependencies: POM

Therefore, this project uses JUnit 3.8.1 for testing. Recall that in the JUnit chapter, Section Keeping the test code separate from the main code, we configured the classpath of the Eclipse project so that some dependencies were only available for testing. In Maven this is managed through the above concept of scope. The two areas mentioned above are the only ones we will need for now. For further and complete details on the available scopes, refer to http://maven.apache.org/guides/introduction/Introduction-to-dependency-Mechanism.html.

5.1.6 *SNAPSHOT*

If we are using dependencies with the version ending with -SNAPSHOT, then we are using the versions still below development: that version was

not officially released. Its implementation is not to be considered final, nor necessarily stable. Developers can release several different snapshots in a single day. Expert of has a period of time after which it checks for new snapshots. You can force this check.

- From the command line: -U
- From Eclipse in the launch configuration by selecting the checkbox "Update Snapshots" (see Section Run Maven from Eclipse).

Note that the version of our Maven projects ends with -SNAPSHOT, as we are still working on them implementation. We may already be deploying the snapshot version to a remote repository (including the central repository of Maven).

When we are ready to release the official final implementation of that version, we need to remove it -SNAPSHOT from version before deploying it effectively.

5.1.7 *m2e*

The integration of Maven into Eclipse provided by m2e is a bridge between the dependency of Maven management and the Eclipse compiler.

This means that when you develop a Maven project in Eclipse you are still using Eclipse compiler and its automatic construction mechanisms. The project is still compiled by Eclipse, not by Expert of. However, the classpaths of the Eclipse project are automatically calculated by Eclipse using the Dependencies of the Maven project.

In the JUnit chapter, when we created the first JUnit test case, Eclipse automatically suggested adding the JUnit library to the project's classpath, because Eclipse contains JUnit. For other external JARs, we had to download them ourselves and update the project classpath accordingly. When they were developing a maven project, m2e automatically downloads the dependencies specified in the pom.xml (and caches them as usual in the .m2 folder) and the Eclipse classpath is automatically configured to use those dependencies.

For example, with the project pom.xml we created, we can see the "Maven Dependencies" node in the project. Expand the node and you can see all the JARs that are part of the classpath. They refer to the locally cached version in the .m2 folder. Note that since JUnit is specified in the

pom.xml as a "test" dependency, the JAR is colored as an Eclipse test dependency.

When a dependency is added in the pom.xml or the version of an existing dependency is changed, just saving the pom.xml file in Eclipse automatically activates the updating of the classpath, afterwards download the new JARs, if they are not already cached. As an Eclipse plugin, m2e provides an multi-tab editor for pom.xml file.

For the moment we use the "pom.xml" tab of the editor, which allows us to edit the file directly XML file. Let's change the version of JUnit to 4.13:

```
1 <dependencies>
2   <dependency>
3     <groupId>junit</groupId>
4     <artifactId>junit</artifactId>
5     <version>4.13</version>
6     <scope>test</scope>
7   </dependency>
8 </dependencies>
```

Save the file and the JARs in the "Maven Dependencies" node will be updated (after the JARs have been downloaded and cached locally). Besides junit-4.13.jar we also see hamcrest-core-1.3.jar, which, as we learned in the JUnit Chapter, Using Hamcrest Pairing Section, is a transitive dependency of JUnit 4 (was not the case with the old JUnit 3). As a transitive dependency of a test addiction, it is also colored as test addiction.

We can now use JUnit 4 in the project without further configuration.

In the m2e preferences (Preferences → Maven) you can specify whether the JAR sources should be downloaded automatically, when a dependency is downloaded. This allows you to have Java dependency sources available for inspection by expanding the "Maven Dependencies" node or browsing in a Java type of a dependency. If this preference is not set, the first time you try this inspect the dependency type, the sources will still be downloaded automatically (so you will there is a delay the first time a Java type of a dependency is opened).

5.1.8 *Properties*

We have already used the properties section when we switched the Java compilation level to version 8 Maven properties are key/value pairs to define values and can be used in other parts of the POM. In the <properties> section properties are defined with the syntax.

```
<key>value</key>
```

Defining a property also implicitly overrides the possibly pre-existing property with the same key. Many Maven plugins pick properties defined in the POM or default to some values for such properties. If you know a plugin picks a property with a given key, then specifying a value for that property in the <properties> section results in configuring the plugin behavior.

As we did in Section Java settings, knowing in advance that the Maven Java compiler plugin honors the properties <maven.compiler.source> and <maven.compiler.target> we can configure the Java compilation level by providing values for those properties. Typically, a Maven plugin can be configured by specifying values for some of its parameters. If the value for an optional parameter is not specified, then the default value is assumed for that parameter. However, if for a parameter a user property is taken into consideration, that user property has the precedence. For example, for the Maven compiler plugin, <maven.compiler.source> is the user property corresponding to the parameter <source>

For example, we can define a new property for the JUnit version and use it as the version of corresponding dependence:

```
<properties>
  ...
  <junit.version>4.13</junit.version>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

5.1.9 *Resources*

We have already seen the concept of a Java resource in Chapter JUnit. For example, the configuration file log4j2.xml, placed in the root of the src source folder, is a resource of our application, that is, some data that is needed by our application to run and that can be accessed by the application. Resources must be available at runtime in the classpath.

In that chapter, we directly put the resource file log4j2.xml in the src source folder. Eclipse automatically copies any data file (that is, files that are not Java sources) in the output folder of the source folder. We also had another resource file log4j2.xml in the tests source folder, used by tests. The test resource has the precedence over the homonymous file in the src directory.

Maven fosters the separation of source Java files from resource files, both for the main code and for the test code. By default main resources should be placed in the directory src/main/resources and test resources should be placed in the directory src/test/resources. A Maven Java project's POM is implicitly configured with such directories; if they are not present they are ignored.

For example, in the project we created with the archetype, create the directory resources both in src/main and in src/test. Then create any text file in both directories. They will be automatically copied in the corresponding output folders, that is, target/classes and target/test-classes, respectively.

```
target
├── classes
│   ├── afile.txt
│   └── com
│       └── mycompany
│           └── app
│               └── myapp
│                   └── App.class
└── test-classes
    ├── afile.txt
    └── com
        └── mycompany
            └── app
                └── myapp
                    └── AppTest.class
```

Note that while Eclipse is automatically copying such resource files, the corresponding source folders are not marked as source directories. It is enough to update the Eclipse project with Maven → Update Project. . . .

This also has the benefit of updating the Eclipse project setting concerning the encoding of the files in the resources folders (again, taking the setting from the pom.xml). Once again, the src/test/resources is colored as a source folder for tests.

5.1.10 *Example of a Maven project*

In the POM we specify the encoding for the resources and the Java compilation settings as properties, as we have already seen:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>bank-example</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
</project>
```

Save the file, wait for Eclipse to download these dependencies (if they are not already cached) and the project will compile. Let's run the Main application. We don't see any output from Log4j since we did not specify any configuration file. Such a file is a good candidate for being a resource file. Let's copy the log4j2.xml from the src folder of the original example into the src/main/resources of this project.

Now the application runs and information is correctly logged on the console.

Now let's copy the test code from the original project into the src/test/-java folder of this Maven project. The code does not compile since neither JUnit nor AssertJ are available to this project. We find the Maven coordinates for AssertJ from its home page https://assertj.github.io/doc/assertjcore-quick-start and we already know how to add JUnit:

Save the POM and once again, after the Maven dependencies are downloaded, the test code compiles and tests can be run.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <version>3.15.0</version>
  <scope>test</scope>
</dependency>
```

5.1.11 *Build with Maven*

Up to now, by relying on Eclipse m2e, we created Maven projects and develop them in Eclipse. Thanks to m2e, the settings specified in the POM allowed us to update the Eclipse project metadata accordingly. Most of all, we relied on the Maven dependency management to have dependencies automatically downloaded from the Internet and to have the Eclipse project's classpath updated accordingly. Moreover, we had for free the separation between main and test code and their corresponding dependencies. However, the compilation of Java sources (and the copy of resources to the output folders) is still performed by Eclipse. Similarly, we run JUnit tests with Eclipse as well.

In this section we will see how to perform a build of the project with Maven. During the build the following tasks will be performed:

Dependencies are downloaded (if not already cached)

- The main code, that is, by default, what's in src/main/java will be compiled, using only the dependencies with scope compile
- The test code, that is, by default, what's in src/test/java will be compiled, using also the dependencies with scope test and, of course, the main compiled code.
- Tests will be run and their results recorded
- The JAR with only the main code will be created

The complete set of executed tasks is much more complex, and it will be detailed later. Of course, if anything goes wrong in any of the above tasks, the whole build will be interrupted and marked with failure (including the case when unit tests fail). As already sketched before, all generated artifacts will be created in the target directory (the so called

build directory). You can always safely remove this directory and it will be recreated on the next build.

5.1.12 *Maven lifecycles*

In the end, to understand how a Maven build works, and thus, to be able to configure a Maven project, it is enough to understand the concept of Maven lifecycle, which can be summarized as follows:

- A Maven **lifecycle** is an ordered list of
  - **Phases**, which have some associated
    - **Goals**, provided by Maven **plugins**
      - Maven plugins have some goals that are bound to some phases; some plugin goals are bound by default to specific phases; otherwise, plugin goals can be explicitly bound to specific phases in the configuration of the plugin in the POM.

Maven provides 3 built-in lifecycles:

clean
removes all the generated artifacts from previous builds
default
generates all the artifacts of the build
site generates a web site with documentation and reports (e.g., unit test results and code coverage)

We will not deal with the site lifecycle so we won't further discuss that.

For example, the **clean** lifecycle defines these 3 phases:

- pre-clean
- clean
- post-clean

When you request Maven to execute a phase then that phase is executed only after executing all the previous phases in the corresponding lifecycle. Executing a phase corresponds to executing all the goals of plugins that are bound to that phase. Such goals are executed according to the order with which plugins have bound their goals. For example, if we execute

```
mvn clean
```

First the phase pre-clean is executed (that is, all the goals bound to that phase) and then the phase clean is executed.

Note that clean passed on the command line is the phase clean of the lifecycle clean, not the name of the lifecycle clean.

The most interesting lifecycle is of course the default lifecycle, where most of the work is done (compilation, testing, packaging). This lifecycle

has more than 20 phases. We now see the main phases and summarize their intents:

validate
validates the `pom.xml` file and checks that all the necessary information is available.
initialize
initializes the build, including creating and setting up the directories and setting properties.
generate-sources
generates any required source code that is meant to be part of the compilation. (This is useful if you use additional tools that generate Java code.)
generate-resources
generates resources that will be part of the final artifact (and used in tests).
process-resources
copies the resources to their build directory.
compile
compiles the main source code.
process-classes
processes the byte-code generated during the compilation phase, e.g., for code enhancements.
generate-test-sources,..., test-compile, process-test-classes
as above, but for test resources and test code.
test run tests using a unit testing framework.

prepare-package
prepares the artifacts to be packaged.
package
packages the artifacts into a distributable format, depending on the packaging type of the project, e.g., `jar`, `war`, etc. (we will see packaging types in Section *Maven packaging*).
pre-integration-test, integration-test, post-integration-test
These are related to *integration tests* that we will see in details in Chapter *Integration tests*.
verify
verifies the validity of the package and in general of the project (typically including verifying the results of integration tests).
install
installs the packaged artifacts into the local repository (where also downloaded artifacts are cached, e.g., in the home directory in the subdirectory `.m2`), for use as dependencies in other projects locally.
deploy
deploys the packaged artifacts to a remote repository.

Some standard Maven plugins have goals that are by default bound to specific phases. For the **default** lifecycle, the bindings depend on the packaging type.[5]

The bindings can be inspected by running this Maven command

```
mvn help:describe -Dcmd=<phase name>
```

For example,

```
mvn help:describe -Dcmd=clean

[INFO] 'clean' is a phase within the 'clean' lifecycle, which has the following phas\
es:
* pre-clean: Not defined
* clean: org.apache.maven.plugins:maven-clean-plugin:2.5:clean
* post-clean: Not defined
```

Showing that there are no plugin goals bound to the phase `pre-clean` and that the goal `clean` of the plugin `maven-clean-plugin` is bound to the phase `clean`.

As you can imagine, the goal `clean` of the plugin `maven-clean-plugin`, clears the build directory, i.e., it deletes the `target` directory.

5.1.13 *Run Maven using command line*

In order to run Maven from the command line, we must use its binary, called mvn. If we run this command from the root directory of our Maven bank example project we can see this output:

Several phases can be specified when invoking Maven. For example, with the following command we first execute the phase clean (after executing the previous phases of the lifecycle clean) and then the phase compile (after executing the previous phases of the lifecycle default):

We can see the execution of a few goals:

• the goal clean of the plugin maven-clean-plugin bound to the phase clean (that we have already seen before),

• the goal resources of the maven-resources-plugin (bound to the phase process-resources of the default lifecycle)

• the goal compile of the maven-compiler-plugin (bound to the phase compile of the default lifecycle).

From the output we can verify that the goal resources of the maven-resources-plugin copies our log4j2.xml from the src/main/resources to the output folder target/classes and the goal compile of the maven-compiler-plugin compiles the Java files in src/main/java into the output folder target/classes.

If we now invoke the phase test, all the previous phases of the default lifecycles will be executed (including the ones seen in the previous run), in particular test resources are copied to the output folder target/test-classes, Java files in src/test/java are compiled into the output folder target/test-classes and finally the unit tests are run (a report of the executed tests will also be generated into the output directory target/surefire-reports).

We can see that, since no clean phase has been invoked the previously generated Java classes of the main sources are still there, thus, they are not compiled again. The maven-resources-plugin and maven-compiler-plugin have specific goals for copying test resources and compile test code, bound to the corresponding phases of the default lifecycle. The goal test of the maven-surefire-plugin, bound to the phase test of the default lifecycle, is the one responsible for executing JUnit tests.

Note: When Java sources are compiled, Maven uses the default Java compiler taken from the currently installed JDK. Of course, the version of the JDK must be at least equal to the compilation level specified in the POM. Instead, recall that Eclipse compiles Java sources using its own Java compiler. In fact, Eclipse does not need a JDK, the JRE is enough. The Java compiler used by Maven can be changed: refer

to http://maven.apache.org/plugins/mavencompiler-plugin/non-javac-compilers.html.

```
mvn clean

[INFO] Scanning for projects...
[INFO]
[INFO] ---< com.example:bank-example >---
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ bank-example ---
[INFO] Deleting <full path>/bank-example/target
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

```
mvn clean compile

[INFO] Scanning for projects...
[INFO]
[INFO] ---< com.example:bank-example >---
[INFO] Building bank-example 0.0.1-SNAPSHOT
[INFO] ---[ jar ]---
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ bank-example ---
[INFO] Deleting <full path>/bank-example/target
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ bank-example -\
--
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ bank-example ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 3 source files to <full path>/bank-maven-example/target/classes
```

The output directory target/surefire-reports will contain the reports of executed tests both in textual and XML format. The XML format is suitable to be opened directly in the Eclipse JUnit view, e.g., by double-clicking on the XML in Eclipse, navigating to that folder).

Note that, besides specifying dependencies in our POM, we did not have to configure anything else, thanks to the fact that many standard plugins are already bound to the corresponding phases, with defaults based on conventions. Besides the conventions on source and resource folder, we also rely on the default conventions of the maven-surefire-plugin, which automatically executes all the tests that match these file

```
mvn test

[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ bank-example -\
--
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ bank-example ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ bank-e\
xample ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ bank-exampl\
e ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to <full path>/bank-example/target/test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ bank-example ---
[INFO] Surefire report directory: <full path>/bank-example/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running testing.example.bank.BankTest
2018-10-12 14:31:22,970 [main] DEBUG testing.example.bank.Bank - Success: withdraw(1\
, 5.00)
2018-10-12 14:31:23,003 [main] INFO  testing.example.bank.Bank - New account opened \
with id: 2
2018-10-12 14:31:23,017 [main] DEBUG testing.example.bank.Bank - Success: deposit(3,\
 5.00)
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.378 sec
Running testing.example.bank.BankAccountTest
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec

Results :

Tests run: 12, Failures: 0, Errors: 0, Skipped: 0

[INFO] BUILD SUCCESS
```

patterns and that are not abstract classes: **/Test*.java, **/*Test.java, **/*Tests.java, **/*TestCase.java.

5.1.14 *Run Maven goals*

When running the Maven command, besides phases, you can execute single goals. You can run a single goal with the syntax.

<groupId>:<artifactId>:<version>:<goal>

The version can be omitted. In that case, the version will be taken from the configuration of that plugin in the POM (taking into consideration also the parent POM, see Section Parent and modules). If no configuration is found, the latest version will be used.

<groupId>:<artifactId>:<goal>

For the standard Maven plugins, the group can be omitted:

<artifactId>:<goal>

If a plugin has several goals with the same name attached to the same phase (that is, executions, which we'll see later in Section Configuring a Maven plugin), you can specify the id of a single execution of that goal.

<groupId>:<artifactId>:<goal>@<id> Moreover, if an artifactId has

occurrences of "maven" and "plugin", surrounded by hyphens ("-"), a shorter form can be used, by specifying only the prefix. For example, instead of maven-compiler-plugin we can simply specify compiler.

This is based on a convention for naming Maven plugins:

maven-${prefix}-plugin
for official plugins maintained by the Apache Maven team itself
${prefix}-maven-plugin
for plugins developed by other teams

Specifying a goal at command line means executing only that goal independently from any phase. For example, the following command only compiles the Java sources (compare this with the execution of mvn compile):

mvn compiler:compile

That is, it executes only the compile goal of the maven-compiler-plugin.

Of course, phases and goals can be mixed on the command line. This is especially useful when you need to explicitly run a goal of a plugin,

which by default is not bound to any phase. Note that some goals of plugins can automatically invoke complete phases. An example is the goal report of the maven-surefire-report-plugin, which automatically executes the phase test of the default lifecycle.

Such a plugin also provides the goal report-only, which does not execute the phase test, thus it assumes that tests have already been run. The web site will be generated by default in the target/site directory, since such web contents are meant to be included in the web site generated during the site lifecycle. In fact, the web contents generated by this plugin rely on images and CSS files that are generated by the goal site of the maven-site-plugin (the one whose goals are bound by default to phases of the site lifecycle). If we want a nice rendering of the test web page, we need images and CSS files as well. If we don't want to execute the whole site lifecycle, we can invoke only the goal site of the maven-site-plugin, by specifying that we don't want the whole site generation. This can be achieved by passing the value false for the property generateReports. As an example, we run the phase test, the goal for generating the web page with test results and the goal for generating the images and CSS files:

```
1 mvn test surefire-report:report-only \
2   site:site \
3   -DgenerateReports=false
```

Open the file target/site/surefire-report.html to see the generated web report with the run tests.

In this case, we use the default versions of maven-surefire-report-plugin and maven-site-plugin; these defaults are part of the current Maven installation. We could use the complete syntax for invoking the goal by specifying an explicit version of a plugin (in the following example, we specify version 3.7.1 of the maven-site-plugin):

```
1 mvn test surefire-report:report-only \
2   org.apache.maven.plugins:maven-site-plugin:3.7.1:site \
3   -DgenerateReports=false
```

5.1.15 *Run Maven from Eclipse*

The Eclipse plugin m2e also provides tools for running a Maven build directly from Eclipse. By default, a version of Maven embedded in the m2e plugin is used, thus you can run Maven builds from Eclipse without installing Maven in your system. This can be configured with the property Maven → Installations, in case you want to use a Maven distribution installed in your system. The contextual menu Run As of a Maven project has many submenus for invoking Maven with some predefined goals. The submenu Maven build... allows you to specify the phases or goals to execute, the profiles to activate and other configurations. Remember that this is an Eclipse run configuration, and, as such, once created it can be stored in the project .

5.2 MAVEN PACKAGING

Each Maven project specifies the packaging in the pom.xml, which represents the final produced artifact from the project. This is also strictly related to the Maven lifecycles and also to the Maven plugins that are enabled by default (with the default settings) in the Maven project. We will now describe the main packaging types we will use in the book.

5.2.1 *Packaging "jar"*

The default, if not specified, is jar. This is explicit in the projects created starting from the archetype "simple Java project". In fact, most of the time, from a Java project you want to create a JAR, e.g., with a library or with a complete application. Other examples of packaging types, which we will not use, are war or ear, intended for Java web applications.

5.2.2 *Packaging "pom"*

Besides "jar", the only other packaging type we will use in this book is pom. This kind of packaging is not meant to produce any (binary) artifact, but it is crucial for dealing with Maven modules, which we'll describe in Section Parent and modules.

In particular, a project of type pom can be used

• for aggregating configurations, dependencies and plugins that are common to several projects;

• for building together several sub-projects (Maven modules).

Note that usually a project with type pom is used for both the above goals, but it is not strictly required: you can have two separate pom projects for the two goals.

5.2.3 *Parent and modules*

Each pom.xml can specify a Parent POM: all the configurations of the parent POM will be inherited, similarly to what happens with Java class inheritance.

The parent POM is specified as follows:

```
<parent>
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <relativePath>...</relativePath> <!-- optional -->
</parent>
```

As usual the GAV coordinates must be specified. By default, the parent POM is looked up in the physical parent directory of the current project. In such a case, the <relativePath> needs not be specified. Otherwise, the full relative path to the parent's pom.xml file must be specified, e.g.,

```
<parent>
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <relativePath>../my.parent/pom.xml</relativePath>
</parent>
```

If the current project's groupId and version are the same as the one of the parent, then they can be omitted. For example, instead of One can simply write:

```
<parent>
  <groupId>com.mycompany.app.</groupId>
  <artifactId>my-app-parent</artifactId>
  <version>1.0.0-SNAPSHOT</version>
</parent>
<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<version>1.0.0-SNAPSHOT</version>
```

```
<parent>
  <groupId>com.mycompany.app.</groupId>
  <artifactId>my-app-parent</artifactId>
  <version>1.0.0-SNAPSHOT</version>
</parent>
<artifactId>my-app</artifactId>
```

Even if no parent pom is specified, all POMs implicitly inherit from the Super POM, which is part of the current Maven installation (similarly to the class Object in Java).

Code modularity is known to be an important characteristic aiming at easy maintainability and high code reuse. It is also important to have a modular structure of the files and folders of an application as well. If we can split the code of an application in separate projects with a few dependency relations among them we can maintain them easily and in other applications we can reuse, by adding a dependency, only the projects that we actually need, instead of the whole application. However, we would like to avoid repeating common configurations in all the POMs of the projects, by reusing common configurations just like we do, for example, with class inheritance in Java.

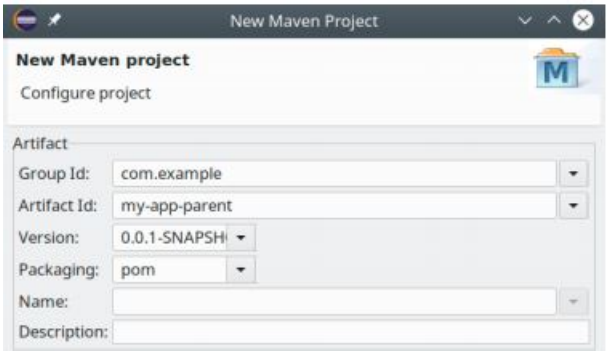
It is common practice to have several Maven modules, each one representing a Maven project, that have in common the configuration of a Parent pom.

5.2.4 *Create a parent project and a module project*

We will now create a parent project and two module projects. For simplicity, the module projects will be in two subfolders of the parent project folder.

Select File → New Project... → Maven → Maven Project and choose a possible different location for the created project. Make sure that the checkbox for skipping archetype selection is checked and press "Next".

Provide the groupId and artifactId, e.g., com.example and my-app-parent; then, make sure you select pom as the "Packaging", and press "Finish".



The resulting pom.xml will be minimal:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>my-app-parent</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
</project>
```

The parent project also contains the empty folder src/site; since we won't create a site for our Maven projects, you can delete that folder.

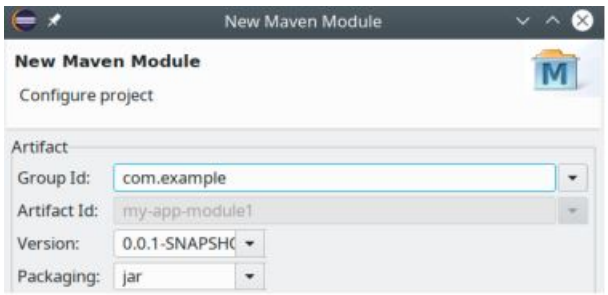
Now we create the first module project. As usual, Eclipse can infer default settings when actions are run as context menus when a project or a folder is selected. Since we want to create a module of a parent project, we select the parent project we have just created and we use the context menu New Project... → Maven → Maven Module (note "Module", not "Project", this time).

We specify the "Module Name", e.g., my-app-module1. Note that the "Parent Project" is already filled. Alternatively, another parent project can

be selected, choosing among the Maven projects currently opened in the workspace. We skip the archetype selection:



On the next wizard page, the groupId is automatically inferred from the specified parent project. The artifactId corresponds to the "Module Name" previously specified and cannot be changed.



Press "Finish" and the Maven module is created.

During this process, the parent project's pom.xml has been updated with the newly created module:

```
1  <project>
2    <modelVersion>4.0.0</modelVersion>
3    <groupId>com.example</groupId>
4    <artifactId>my-app-parent</artifactId>
5    <version>0.0.1-SNAPSHOT</version>
6    <packaging>pom</packaging>
7    <modules>
8      <module>my-app-module1</module>
9    </modules>
10 </project>
```

The pom.xml of the module project refers to the parent project. Note that since the module lives in a subdirectory of the parent project, no

relative path information has to be specified. groupId and version are inherited from the parent POM and are not repeated:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.example</groupId>
    <artifactId>my-app-parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>my-app-module1</artifactId>
</project>
```

We create the <properties> and the <dependencies> sections in the parent POM. The POM of the parent is now as follows:

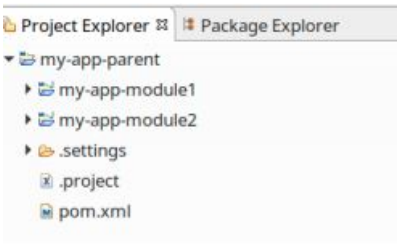
```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>my-app-parent</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <modules>
    <module>my-app-module1</module>
  </modules>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <!-- Change the Java compilation level to 8 -->
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

We update the Maven projects. We can do that starting from the parent project: right-click on the parent project and select Maven → Update Project..., in the dialog make sure that both the parent and the module projects are selected and that the checkbox "Update project configuration from pom.xml" is selected.

Although the Java compiler setting is specified in the parent project, the module project inherits this setting as well.

Let's add a Java class in the src/main/java and a test case Java class in src/test/java in the module project (e.g., the ones created by the Maven archetype). Although the module POM does not declare JUnit as a test dependency, the test still compiles: the module inherits the JUnit dependency from the parent POM.

We can now create another module project, my-app-module2, with the above procedure. Also this submodule will inherit all the configurations from the parent POM.



Hierarchical presentation of nested projects

Note that in Eclipse, you could access the sources of the module projects both from the module project itself and from the module project's subfolder from the parent project.

The "Project Explorer" view allows you to have a single a "Hierarchical" presentation of nested projects, so that you avoid the above duplicate folders: from the view menu select Projects Presentation → Hierarchical.

In this second module, we add a dependency on the first module. Recall that the full Maven coordinates must be provided. Since both modules are part of the same multi-module project, they share the same groupId and the same version. Thus, when specifying the dependency, in order to avoid repeating such information and to force consistency, we use the corresponding Maven properties.

Once we save this POM, in Eclipse, let's make sure we can effectively refer to a Java class of the first module from the second module:

It is useful to be able to see the "effective POM" that Maven uses, that is, the POM after applying POM inheritance (parent and super POM).

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.example</groupId>
    <artifactId>my-app-parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>my-app-module2</artifactId>
  <dependencies>

    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>my-app-module1</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>
</project>
```

From the command line the effective POM can be obtained as follows:

mvn help:effective-pom

In Eclipse, the m2e editor provides a dedicated tab (of course, read-only),

For example, let's examine the effective POM of the my-app-module2 that we created in the previous section:

- The groupId and version, which we omitted, are there (and they are the same as the ones of the parent POM);
- Properties are inherited as well as the JUnit dependency;
- The properties in the module1 dependency are resolved;
- Many configurations are present;
- All paths are made absolute (source and output directories).
- The <modules> section is NOT inherited from the parent POM.

```java
package org.my.app.module2;

public class App {
   public static void main(String[] args) {
      org.my.app.module1.App.main(args);
   }
}
```

5.3 CONFIGURING A MAVEN PLUGIN

In the previous sections we saw that some Maven plugins are already configured so that their goals are bound to the lifecycles of Maven. Such configurations are inherited from the super POM and the bindings to the lifecycle phases are also based on the packaging type of the current project. We also saw that, independently from whether they are configured or not, goals of plugins can be executed directly.

If we want to configure a Maven plugin in the project POM (or in its parent POM, in case of a multi-module project) we must do that in the following section.

```xml
<build>
  <plugins>
  ...
  </plugins>
</build>
```

Each plugin is configured in an element <plugin>.

The configuration of a plugin typically consists in specifying

- its Maven coordinates.

- an optional configuration, that is, passing some arguments to the plugin, in a <configuration> element; these have the element shape <key>value</key>;

- the binding of its goals to the lifecycle phases; each binding is specified in an <execution> element, inside the <executions> element.

Each <execution> can have a different configuration if needed (otherwise the main <configuration> is used for all the <executions>). Thus,

the same goal can be configured to run multiple times with different configurations in different phases. Each execution can also be assigned an <id> (see Section Run Maven goals concerning the specification of an id when running a Maven goal). When multiple executions are bound to the same phase, they will be executed in the order specified in the POM; executions that are inherited from the parent POM will be executed first.

Thus, a plugin configuration typically has the following shape

```
<plugin>
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <executions>
    <execution>
      <id>...</id>
      <phase>phase to bind the goal to</phase>
      <goals>
        <goal>plugin goal</goal>
      </goals>
      <configuration>
        ... configuration for this execution only
      </configuration>
    </execution>
    <execution>
      ... as above
    </execution>
  </executions>
  <configuration>
    ... configuration common to all executions
  </configuration>
</plugin>
```

Note that a plugin goal might have a default phase; in that case, if no phase is specified, the goal mentioned in the <execution> element will run in its default phase. When a plugin is configured, possible configurations of the same plugin that are inherited from the parent POM are merged in the current project. Before going on, let's have a look at the default configuration of the maven-compiler-plugin in one of our Maven projects. We can examine this by looking at the "Effective POM"

Note that the groupId is omitted since this is a standard Maven plugin. There is no explicit <configuration>, thus the default configuration values

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <executions>
    <execution>
      <id>default-compile</id>
      <phase>compile</phase>
      <goals>
        <goal>compile</goal>
      </goals>
    </execution>
    <execution>
      <id>default-testCompile</id>
      <phase>test-compile</phase>
      <goals>
        <goal>testCompile</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

are used (recall that this plugin also takes into consideration the user properties we used for specifying the Java compilation level). Then, the goal compile is bound to the phase compile and the goal testCompile is bound to the phase test-compile.

In the rest of this section we will see a few examples of Maven plugin configurations.

5.3.1 *Configuring the Maven compiler plugin*

An alternative way of configuring the Java compilation setting is to configure the maven-compiler-plugin passing the arguments corresponding to the user properties we used before

In one of the Maven projects that we had already configured for Java 8, let's remove the properties we had previously set

In Eclipse, when changing a plugin configuration like the maven-compiler-plugin in the POM, you get an error like this

So we also must update the project. The Java compilation level will still be 1.8, though we used a different method to configure it.

```
<maven.compiler.source>1.8</maven.compiler.source>
<maven.compiler.target>1.8</maven.compiler.target>
```

and let's configure the `maven-compiler-plugin` passing `source` and `target`:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

```
Project configuration is not up-to-date with pom.xml. Select: Maven->Update Project.\
.. from the project context menu or use Quick Fix.
```

Let's have a look at the "Effective POM" and we can verify that our configuration has been merged with the one inherited from the super POM:

If we want to use a plugin in all projects of a multi-module project, we configure the plugin in the parent POM: all sub-modules will inherit such a configuration.

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <executions>
    <execution>
      <id>default-compile</id>
      <phase>compile</phase>
      <goals>
        <goal>compile</goal>
      </goals>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </execution>
    <execution>
      <id>default-testCompile</id>
      <phase>test-compile</phase>
      <goals>
        <goal>testCompile</goal>
      </goals>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </execution>
  </executions>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
  </configuration>
</plugin>
```

5.3.2 *Create source and javadoc jars*

For open source projects that are released on a remote repository, it is best practice to publish also a JAR with sources and a JAR with Javadoc, besides the standard binary JAR artifact. This is achieved using two standard Maven plugins: maven-source-plugin and maven-javadoc-plugin, respectively.

Note that these two plugins, besides generating the JARs with sources and javadoc, will also "attach" the two additional JARs as artifacts of the current project. This way, these two additional JARs will also be installed and deployed in the corresponding lifecycle phases.

In case of a multi-module project we can configure these two plugins in the parent POM, so that all sub-modules will automatically have such plugins enabled and configured.

The idea is to configure these two plugins by binding the goals jar (the name of the goal is the same in the two plugins) to the phase package, that is, the phase where the standard binary JAR is also created.

For the sources we configure:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-source-plugin</artifactId>
  <version>3.0.1</version>
  <executions>
    <execution>
      <id>attach-sources</id>
      <!-- the phase would not be required in this case: goal jar of
        this plugin already binds by default to the "package" phase -->
      <phase>package</phase>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Note that we could skip the specification of the phase: by default, the goal jar is already bound to the phase package.

For the Javadoc:

Now, if we run the Maven build specifying the phase package (or some later phase), the two additional JARs will be generated.

For example, if we configured the above plugins in the my-app-parent of the example we created in the previous sections, by running mvn package on the parent project the two sub-modules will contain the additional JARs:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-javadoc-plugin</artifactId>
  <version>3.0.1</version>
  <executions>
    <execution>
      <id>attach-javadocs</id>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
```

```
my-app-module1-0.0.1-SNAPSHOT.jar
my-app-module1-0.0.1-SNAPSHOT-javadoc.jar
my-app-module1-0.0.1-SNAPSHOT-sources.jar
```

The Javadoc html files are generated in the target/apidocs directory, before they are packaged into the jar. You can browse them with a browser.

5.3.3 *Configuring the generated jar*

Let's create a new Java project using the maven-archetype-quickstart archetype, as already done before in this chapter. We use com.mycompany.app for the groupId, my-app for the artifactId and 1.0-SNAPSHOT for the version. The created project has a Java class with the main method, com.mycompany.app.App:

```
package com.mycompany.app;

public class App {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Let's create the jar for this project, with mvn package. Now we try to run from the command line the Java application by running the generated jar, which can be found in target/my-app-1.0-SNAPSHOT.jar:

Indeed, the generated META-INF/MANIFEST.MF inside the generated JAR, by default, has no information about the Java class with the main method. In such a case, in order to run a Java application packaged in a

```
java -jar <fullpath>/target/my-app-1.0-SNAPSHOT.jar

no main manifest attribute, in <fullpath>/target/my-app-1.0-SNAPSHOT.jar
```

JAR, we need to be more explicit: we add the JAR to the classpath and we specify the fully qualified name of the Java class with the main method:

```
java -cp <fullpath>/target/my-app-1.0-SNAPSHOT.jar com.mycompany.app.App

Hello World!
```

This time the main method is correctly executed.

However, when a jar is created with a main class, it is better to add this information to the manifest in the generated JAR, so that we can directly run the JAR, as we tried to do previously.

It is just a matter of configuring the maven-jar-plugin. Such a plugin is already configured with its goal jar bound to the phase package (have a look at the "Effective POM"). We just need to configure it so that we add the Main-Class section attribute in the generated JAR; this is done by specifying the fully qualified name of the main class with the element <mainClass> of the element <manifest>:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.4</version>
      <configuration>
        <archive>
          <manifest>
            <mainClass>com.mycompany.app.App</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Once again, our configuration will be merged with the inherited one, so that the jar goal is still bound to the phase package (have a look at the "Effective POM").

Let's run mvn package once again and this time we can simply run the JAR:

```
java -jar <fullpath>/target/my-app-1.0-SNAPSHOT.jar

Hello World!
```

5.3.4 *Plugin management*

The <pluginManagement> section in the <build> section has the same aim as the <dependencyManagement> section (see Section Dependency management) but for plugins.

This section is even more useful for plugins since Maven will still merge the configurations inherited from a parent POM (including the super POM) and the ones of the current POM, taking into consideration both the ones in <pluginManagement> and the ones in <plugins>.

In Section Configuring the Maven compiler plugin we saw how to configure the Java compilation level directly in the maven-compiler-plugin. We can do that in <pluginManagement>:

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

Note that in this case we don't even need to specify the version of the plugin, since that is taken automatically after merging the inherited configuration of the plugin.

5.3.5 *PIT Maven plugin*

We saw that it might take some time to run such a tool, thus, in a Maven build, we might want to run it only from time to time, by specifying the plugin goal explicitly. However, we configure the plugin once and for all in the <pluginManagement>, by specifying the classes to mutate (targetClasses), the tests to run with the mutators (targetTests) and many additional parameters, like, the mutators to apply. See http://pitest.org/quickstart/maven/ for more information about the plugin configuration. As we anticipated in Chapter Mutation Testing, Section Narrowing mutations, the PIT Maven plugin allows us to exploit the advanced configuration features of PIT itself for narrowing the mutation testing procedure during a Maven build.

For the Maven bank-example we could configure the plugin as follows:

```
<build>
  <pluginManagement>
    <plugins>
      ...
      <plugin>
        <groupId>org.pitest</groupId>
        <artifactId>pitest-maven</artifactId>
        <version>1.4.10</version>
        <configuration>
          <targetClasses>
            <!-- excludes testing.example.bank.app.Main -->
            <param>testing.example.bank.Bank*</param>
          </targetClasses>
          <targetTests>
            <param>testing.example.bank.*</param>
          </targetTests>
          <mutators>
            <mutator>DEFAULTS</mutator>
          </mutators>
          <mutationThreshold>80</mutationThreshold>
        </configuration>
      </plugin>
    ...
```

We specify the pattern for the classes to mutate (excluding the Main class, which has no test) and the pattern for the tests to execute. Moreover, we enable only the default mutators. Finally, by using <mutationThreshold> we specify the threshold for killed mutations. This represents the percentage of mutants that must be killed in order to get a successful build. If the percentage is below the threshold, the Maven build will fail.

We can now run PIT from Maven with the following command:

```
mvn test org.pitest:pitest-maven:mutationCoverage
```

The web report from PIT can be found in the directory target/pit-reports; each report is created in a subdirectory with the date and time.

5.3.6 *Configuring the JaCoCo Maven plugin*

In Chapter Code coverage we used JaCoCo for code coverage, in particular, its integration in Eclipse, EclEmma. In this section we use JaCoCo and its Maven plugin, org.jacoco:jacoco-maven-plugin, to keep track of code coverage and create a code coverage report.

This also allows us to further experiment with Maven plugins executions and configurations. In that chapter, we saw that JaCoCo implements code coverage by using a custom Java agent. EclEmma provides a special run configuration to run the JUnit tests through this Java agent and then it shows the report in an Eclipse view. In order to use JaCoCo from Maven, instead, we need to use its plugin goals appropriately so that the tests will be executed through the JaCoCo agent.

The jacoco-maven-plugin provides the goal prepare-agent that prepares a Maven property pointing to the JaCoCo Java agent. The idea is to pass this property as a VM argument to the maven-surefire-plugin, so that tests are run through the agent and code coverage information is collected in a file. By default, the property set by this goal is argLine. Since that is the property used by maven-surefire-plugin when running the tests, this goal alone, executed before running tests, is enough to create the file with code coverage information. As usual, this goal can be configured (see https://www.eclemma.org/jacoco/trunk/doc/prepare-agent-mojo.html) by specifying another property to set with the path of the agent and by configuring the output folders for JaCoCo. In this section, we will use the defaults. If you change the defaults, you will have to configure other plugins accordingly.

Let's run this Maven build from one of our projects, e.g., the bank example, invoking the prepare-agent goal explicitly:

```
mvn clean org.jacoco:jacoco-maven-plugin:prepare-agent test
```

Note that the goal prepare-agent must be specified before the phase test, otherwise the JaCoCo agent won't be used when running tests. The jacoco.exec file will be automatically generated in the target folder.

This file is in binary format and only contains information about the covered lines, but it does not represent the report.

5.4 MAVEN PROFILES

Maven profiles are a mechanism that allows you to activate a subset of configurations (including properties, dependencies, plugins and even modules) according to some criteria.

Profiles are defined in the section <profiles>, each one in a <profile> element. Each profile is given an <id> and, as hinted above, can define properties, dependencies, plugins and even modules. The parts defined in the profile are active during the build (and in dependency resolution) only when the profile is active.

When profiles are activated their contents are merged into the "Effective POM", following the definition order. Otherwise, the content of an inactive profile is simply ignored. Any profile can be explicitly activated on the command line with the option -P, specifying its id. In Eclipse, in the Maven build run configuration, there is a text field, "Profiles", where the ids of profiles to be activated can be listed.

Moreover, a profile can be activated automatically based on its element <activation>. For the complete reference of the <activation> element, we refer to https://maven.apache.org/guides/ introduction/introduction-to-profiles.html.

A few activation policies are useful when the project is meant to be built in different execution environments.

This activation rule is based on the current Java version, e.g., for JDK 9 or newer

```
<activation>
  <jdk>[1.9,)</jdk>
</activation>
```

This activation rule is based on the current operating systems, e.g., when running on a Mac

```
<activation>
  <os>
    <family>mac</family>
  </os>
</activation>
```

We can use profiles to activate specific plugins only on demand. Let's take the Maven bank example developed up to now. We can have a profile for activating the JaCoCo plugin (already configured in the <pluginManagement> section) and another one where the PIT Maven plugin has the goal mutationCoverage bound to the phase verify (remember that the other configurations of the plugin are already written in the <pluginManagement> section). This is useful since both code coverage and mutation testing typically increase the build time, thus we might want to activate them only on-demand, not for every build.

For JaCoCo, we remove the plugin enabling from the main <build> section and we move it to the <build> section of a new profile. For PIT, we bind its goal mutationCoverage to the phase verify:

Now in a standard Maven build, neither JaCoCo nor PIT will be used; they can be enabled on the command line with -P and their ids. For example, in this build they will both be enabled:

mvn clean verify -Pjacoco,mutation-testing

The simplest activation policy is the following one:

This activates a profile by default. Note however that all profiles that are active by default are automatically deactivated when another profile is activated explicitly on the command line or through its activation rule. Thus, you cannot assume that the content of a profile activeByDefault is always considered.

If you really want a profile to be always active unless it is explicitly disabled, then setting <activeByDefault> to true will not be enough. A

```
<profiles>
  <profile>
    <id>jacoco</id>
    <build>
      <plugins>
        <plugin>
          <!-- configured in pluginManagement -->
          <groupId>org.jacoco</groupId>
          <artifactId>jacoco-maven-plugin</artifactId>
        </plugin>
      </plugins>
    </build>
  </profile>
  <profile>
    <id>mutation-testing</id>
    <build>
      <plugins>
        <plugin>
          <!-- configured also in pluginManagement -->
          <groupId>org.pitest</groupId>
          <artifactId>pitest-maven</artifactId>
          <executions>
            <execution>
              <goals>
                <goal>mutationCoverage</goal>
              </goals>
              <phase>verify</phase>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

possible solution is to specify its <activation> based on the fact that a specific system property is NOT defined. For example, this makes the jacoco profile active when the system property skipCoverage is NOT defined at all. Thus, this profile is effectively active by default, unless the system property skipCoverage is defined:

Now, if you run verify the jacoco profile will be active. If you run verify -DskipCoverage or verify -DskipCoverage=<any value> the jacoco profile will be deactivated.

Alternatively, we could define the <activation> as follows:

```
<activation>
  <activeByDefault>true</activeByDefault>
</activation>
```

```
<profile>
  <id>jacoco</id>
  <activation>
    <!-- Activated when the system property "skipCoverage" is not defined at
      all, thus, implicitly active by default -->
    <property>
      <name>!skipCoverage</name>
    </property>
  </activation>
  ...
```

```
<profile>
  <id>jacoco</id>
  <activation>
    <!-- Activated when the system property "skipCoverage" is not defined at
      all or is defined with a value which is not "true", thus, implicitly active
      by default -->
    <property>
      <name>skipCoverage</name>
      <value>!true</value>
    </property>
  </activation>
  ...
```

The comment in the XML should be clear: if you run verify the jacoco profile will be active. It will be active also if you run verify -DskipCoverage=<any value different from true>. It will be deactivated if you run verify -DskipCoverage or verify -DskipCoverage=true.

5.5 MAVEN DEPLOY

The last phase of the default lifecycle, deploy, aims at copying your artifacts to a remote Maven repository, for sharing with other developers.

The repository to deploy to can be specified and deployment to a local filesystem is supported as well.

Deployment to Maven Central is the best solution for sharing your Java projects. This requires to follow a (long) procedure, which is described here https://central.sonatype.org/pages/ossrh-guide. html. The length of the procedure concerns only the initial setup: after the first release,

deploying is just a matter of running the deploy phase (with a good Internet connection). This procedure includes specifying several information in the POM and configuring the digital signing of artifacts through GPG.

Note that deploying snapshot versions is different from deploying final versions. Snapshot artifacts will have the SNAPSHOT replaced using a timestamp and a build number. Moreover, snapshot artifacts are meant to be removed from time to time, while actually released artifacts are meant to stay forever. The deployment phase is out of the scope of this book.

5.6 MAVEN WRAPPER

If you want to run a Maven build from the command line, you need Maven installed on your system. The same holds for every team member working on the same project and for the Continuous Integration server, which we'll see in Chapter Continuous Integration. The Maven Wrapper, https://github.com/takari/maven-wrapper, provides an easy way to make sure that every user of your Maven project has everything necessary to run the Maven build, enforcing a specific Maven version.

This wrapper consists of a shell script mvnw (to be used in a Unix-like system like Linux and Mac) and a batch script mvnw.cmd (for Windows). These scripts will first check whether the specific version of Maven is already installed on the current system; if not, they download it and install it by default in the home directory in the subdirectory .m2/wrapper/dists. Then, the scripts will run Maven from the above installation directory.

These scripts are created by running the following Maven command, typically from the root of your Maven project or multi-module project (of course, Maven is needed at least when creating the wrapper):

```
mvn -N io.takari:maven:wrapper
```

This command will create, in the current directory, the above mentioned scripts and a small JAR .mvn/wrapper/maven-wrapper.jar. The JAR is used to bootstrap the download and the invocation of Maven from the wrapper shell scripts.

Now, in order to build the project with Maven, instead of running the command mvn, you run the command ./mvnw, from the directory where the scripts are located. The first time, the Maven binary distribution will be downloaded and unpacked in your home directory in the subdirectory .m2/wrapper/dists. Distributing the wrapper (i.e., the scripts and the wrapper JAR) together with your project will allow anyone to build it by running the scripts, without an already installed Maven program.

The wrapper has also the benefit of enforcing a specific version of Maven, so that anyone using the wrapper will use that Maven version. Recall that each Maven version comes with default versions of the standard plugins, thus using a specific Maven version also implies using those versions of the standard plugins (of course, unless the POM explicitly specifies plugin versions, as we saw in Section Plugin management).

The specific version of Maven is hardcoded in the download URL in the file .mvn/wrapper/maven-wrapper.properties. To switch the wrapper to another version of Maven, simply change the version in the URL in that file. Otherwise, the Maven version can be specified when creating the wrapper or updated by running the Maven command above with the command line argument -Dmaven=<maven version>, e.g.,

```
mvn -N io.takari:maven:wrapper -Dmaven=3.6.0
```

6

CONTINUOUS INTEGRATION

Continuous Integration in the world of modern software development is an increasingly widespread and practiced practice, which reduces the possibility of bugs and favors the correct integration of the components developed by many programmers, especially in very large software projects.

The typical workflow of the CI process can be summarized as follows:

Continuous integration, abbreviated to CI, is a software development practice that consists to integrate all changes into the entire application or project continuously, multiple times a day. This integration is meant to be built automatically, verifying that the compilation is successful, and fully tested, running all application tests. If a bug is introduced in the application by a single component, this is detected at the very first integration of that component into the application. Obviously, CI is based on an autofill mechanism (like Maven, Chapter Maven) to perform all his duties. Also, a version control system is required (like Git, Chapter Git) store the entire application code base. Finally, the compilation process is delegated to a dedicated server, called Continuous Integration server (CI server).

1. A change is pushed to a remote VCS repository;

2. The push notifies the CI server, which fetches the pushed changes into its repository, builds the whole application and runs all the tests;

3. The CI server notifies the developer(s) about the result of the build and of the tests.

While a full integration build can take several minutes, developers can keep working while the server is building. Periodically, the developers check the results provided by the CI server and, in the event of a particular build failure, breaking changes of the corresponding commit it can be analyzed and corrected.

In addition to building and running tests, a CI build process typically performs additional checks code, such as creating a code coverage report and performing further code analysis with the code quality tools (see chapter Code quality). Again, these additional steps are generally skipped when developing, as they take time. These are delegated to the CI server and will be part of its reports. These reports are usually stored in a database and provided to developers via some websites pages. The history of these reports will then be kept in order to have some statistics on the project relationships.

In this book we implement the CI process using

GitHub
as the Git repository host (see Chapter Git, Section GitHub)

Travis CI
(https://travis-ci.com/) as the CI server, which will use Maven to build and test our projects. By default, Travis provides a Linux OS (details about the available Linux distributions and how to select one can be found on the Travis web site); as we will see in this chapter, we can also build the projects on Travis on a MacOS.

Coveralls
(https://coveralls.io/) to store code coverage history and statistics .

SonarCloud
(https://sonarcloud.io) to perform code quality analysis (this will be described in a separate chapter, Chapter Code Quality)

All of these tools are freely available for public open source projects. Plus, they're all set up to be interoperable. It's just a matter of creating free accounts on these services and then linking them to implement the CI process:

- We push to our Git repository on GitHub;
- This triggers a build on Travis CI;
- The project is completed;
- Tests run with code coverage;
- The result of the code coverage is sent to Tute;
- Code quality analysis is performed on SonarCloud;
- The results of all remote services are sent back to GitHub and will be used to annotate the file commit and possibly the PR with those results (including failures).

6.1 EXAMPLE

First of all, we need an example to experiment with the CI process detailed above. As usual, we will keep the example extremely simple, to focus on the tools and the CI process.

Create a Maven project in Eclipse with the archetype maven-archetype-quickstart as done before. You can create it in the workspace since later it will be automatically moved inside a git repository.

We'll keep the POM to the very minimum (we do not even change the Java compilation level, we only change the JUnit version as usual):

```
<project>
<modelVersion>4.0.0</modelVersion>

<groupId>com.examples</groupId>
<artifactId>myproject</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
 <version>4.13</version>
 <scope>test</scope>
</dependency>
 </dependencies>
</project>
```

Listing 6.1: depend.c

The SUT and the corresponding test case are really simple:

```
package com.examples.myproject;

public class App {
public String sayHello() {
return "Hello";
```

```
}
}
```

Listing 6.2: app.c

```
package com.examples.myproject;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class AppTest {
private App app;

 @Before
 public void setup() {
app = new App();
}

@Test
public void testSayHello() {
assertEquals("Hello", app.sayHello());
}
}
```

Listing 6.3: AppTest.c

Add an Eclipse project to a new Git repository, we add this project to a new Git repository. We call the Git repository github-travis-coveralls-example. As seen in Chapter Git, Section Create a repository on GitHub, we create a Git repository on GitHub (with the same name of the local Git repository) and we push our local repository on GitHub.

6.2 TRAVIS CI

First of all, log into Travis CI - go to https://travis-ci.com/ and log in with your GitHub account (the the first time you will be asked to authorize Travis CI to log into your GitHub account). Then, you will be redirected to a webpage on the Travis CI website where you need to activate yours GitHub repositories using the Travis CI GitHub app. By clicking on the button you will be redirected to your GitHub account to approve and install the Travis CI app on your GitHub account (you

you may want to select "All repositories", so all your current and future GitHub repositories will be activated automatically on Travis CI).

6.2.1 *Configure the Travis build*

Travis uses YAML, https://yaml.org/, which is a typical format for configuration files, for the specification the build configuration and expects to find the .travis.yml file in the git root directory repository. Let's create that file outside of Eclipse since it doesn't have to be in the project root directory, but in the root directory of the git repository. Then, edit that file from Eclipse (with File → Open File ...) or with the default system text editor.

The details of such a file are documented on the Travis website and change according to the language used by the project. These are the initial contents:

```
language: java

jdk: openjdk8

# skip installation step
install: true

script:
- mvn -f com.examples.myproject/pom.xml clean verify
```

Listing 6.4: TravisYml.c

For this example, the content of that file should be clear: we specify that our project uses Java, in in particular, we specify the version of the JDK to use. The important part is the script in which we specify the Maven command to run during build. The installation specification is used to simply skip the file installation phase in Travis, which has nothing to do with the installation phase of Maven. The installation phase in Travis is used to install the dependencies needed for compilation. Since we use Maven, that automatically takes care of addictions, let's skip that step in Travis.

Add this file (stage) to the git repository, commit and submit to GitHub. In a few seconds, Travis will start the build, cloning the GitHub repository and running the commands specified in the file .Travis.yml file (in this

case, the Maven build). We can follow the build using the in button upper right corner of the black section ("Follow log"). The build should finish without errors:





Information about the Linux build machine can be viewed by expanding "Build System Information" at the beginning of the compilation log.

From now on, whenever we submit this project's repository to GitHub, Travis will do it automatically build the project.

The interaction between GitHub and Travis is bidirectional: Travis tells GitHub whether the build for a commit succeeded or not, and Github will show this information on the commit.

6.3 CODE COVERAGE WITH COVERALLS

Coveralls is a web service to help you track your code coverage over time, and ensure that all your new code is fully covered.

There is but one prerequisite for Coveralls Cloud (Coveralls Enterprise can use a variety of repo-hosting options):

The Coveralls service is language-agnostic and CI-agnostic, but we haven't yet built easy solutions for all the possibilities as far as repo hosting. Creating an account is fast and easy, just click the "Sign in"

button for your git service (you can link accounts later) and authorize Coveralls to access your repos - no forms to fill out.

Coveralls, https://coveralls.io/, is a free service that tracks code coverage of public projects on GitHub. Code coverage reports are sent to Coveralls, for example, by a Travis build. Tracksuits it can also be used when compiling from our computer, using a user-available token key profile on the site Tracksuits. When communicating to Suits from a Travis build, the token is not necessary and should not be used.

First of all, we need to add our GitHub repository to Coveralls. Go to https://coveralls.io/ and sign in with the GitHub account (the very first time you will be asked to authorize Coveralls to access to your GitHub account).

On the left panel choose "Add Repos".



Use "Sync Repos" if the project is not yet available from the list. Once found, enable the project.

We will experiment with Coveralls in a brand new local branch, e.g., coveralls.

We use the Maven plugin for Coveralls (https://github.com/trautonen/coveralls-maven-plugin). This plugin relies on the XML report generated by JaCoCo, so we must configure JaCoCo in our POM accordingly. We have already seen the configuration of JaCoCo in Chapter Maven, Section Configuring the JaCoCo Maven plugin.

We configure the plugin in the <pluginManagement> section:

```
<build>
<pluginManagement>
 <plugins>
<plugin>
<groupId>org.jacoco</groupId>
<artifactId>jacoco-maven-plugin</artifactId>
<version>0.8.5</version>
<executions>
<execution>
```

```
<goals>
<!-- binds by default to the phase "initialize" -->
<goal>prepare-agent</goal>
<!-- binds by default to the phase "verify" -->
<goal>report</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</pluginManagement>
</build>
```

Listing 6.5: dep.c

Then we create a specific profile

```
<profiles>
<profile>
<id>jacoco</id>
<build>
<plugins>
<plugin>
<!-- configured in pluginManagement -->
<groupId>org.jacoco</groupId>
<artifactId>jacoco-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</profile>
</profiles>
```

Listing 6.6: dep1.c

And we first check whether this works as expected by running locally (either from the command line or from Eclipse):

mvn clean verify -Pjacoco

We verify that the XML report, jacoco.xml is generated in target/site/jacoco/jacoco.xml. Commit and push the new branch. On Travis we verify that our build still works, though we haven't enabled the jacoco profile in .travis.yml.

Let's now enable the profile in the Travis build, by modifying the script section:

script:
- mvn -f com.examples.myproject/pom.xml clean verify -Pjacoco

Commit and push. We verify that JaCoCo is used during the Travis build. We should see something like this at the end of the log

```
[INFO] --- jacoco-maven-plugin:0.8.5:report (default) @ myproject ---
[INFO] Loading execution data file ...com.examples.myproject/target/jacoco.exec
[INFO] Analyzed bundle 'myproject' with 1 classes
```

Now we configure the coveralls-maven-plugin. We configure it in the <pluginManagement> section as well, since we plan to run its goal, coveralls:report, directly from the Maven command line:

```
<build>
<pluginManagement>
<plugins>
...
<plugin>
<groupId>org.eluder.coveralls</groupId>
<artifactId>coveralls-maven-plugin</artifactId>
<version>4.3.0</version>
</plugin>
</plugins>
</pluginManagement>
</build>
```

Listing 6.7: Plug.c

The goal coveralls:report will scan the project in search of jacoco.xml files. By default, it searches for project.reporting.outputDirectory/jacoco/jacoco.xml for every Maven module. Since that is the default output path of the JaCoCo report, and since we did not customize such a path in our configuration, we should be OK. Otherwise, if jacoco.xml is generated in a different folder, the coveralls-maven-plugin must be configured using <jacocoReport> elements, e.g.,

```
<jacocoReports>
<jacocoReport>custom path to jacoco.xml</jacocoReport>
</jacocoReports>
```

Listing 6.8: Path.c

Finally, we call this plugin's goal in our Travis build configuration

script:

- mvn -f com.examples.myproject/pom.xml clean verify -Pjacoco coveralls:report

Commit and push.

Now, at the end of the Travis log, we should see something like

```
INFO Starting Coveralls job for travis-ci (...)
INFO Processing coverage report from
    ...github-travis-coveralls-example/com.exampl\
es.myproject/target/site/jacoco/jacoco.xml
INFO Successfully wrote Coveralls data in 170ms
INFO Gathered code coverage metrics for 1 source files with 7 lines
    of code:
INFO - 2 relevant lines
INFO - 2 covered lines
INFO - 0 missed lines
INFO Submitting Coveralls data to API
INFO https://coveralls.io/jobs/...
INFO *** It might take hours for Coveralls to update the actual
    coverage numbers f\
or a job
INFO If you see question marks in the report, please be patient
```

Listing 6.9: Result.c

We can visit our Coveralls web page, or simply use the URL found in the log. Coveralls show the coverage results for each job, keeping track in its history of the coverage trend.

6.3.1 *Badges*

Both Travis and Coveralls provide badges to show in GitHub, e.g., in the README.md of the root of the repository (written in Markdown, which is automatically rendered when one visits the GitHub project page). These badges show the state of the project concerning the build status and the code coverage, respectively.

On Travis, clicking on "build passing", a popup allows us to select the code to embed in any file to render the badge (select "Markdown" in this



Our first coverage report on Coveralls

By clicking on a file link we can see the colored source:



case, since we want to create a file with the Markdown syntax). Copy the Markdown code to the clipboard:



Figure 6: Getting the code for the Travis badge

Note that the build status of the badge is bound to a specific branch, in our case, we chose master. Then, create the README.md in the root of the Git repository. We can do directly from GitHub, which allows us to create that file from the web interface and edit that online (use the "Add a README" from the main page of the repository). The file is pre-filled with the description we specified when we created the GitHub repository. Here we paste the Markdown code we copied from Travis. We can use the "Preview" tab to see how the README.md will be rendered:

Note that the badge is also clickable and will bring to the corresponding Travis web page. We then use the dialog at the bottom to create a commit directly on the GitHub repository. Of course, we will then have to fetch such changes locally in our Git repository. If the Travis build fails for the master branch, the badge is automatically updated showing a failing status (in red instead of green).

Similarly, on Coveralls, select "Embed" to get the Markdown code to paste in the README.md:

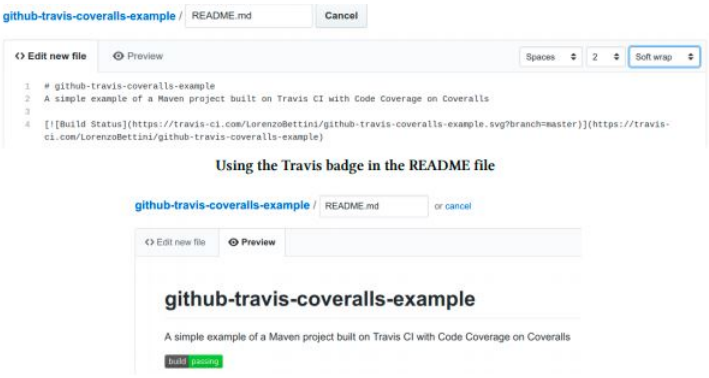


Figure 7: Preview of the README file

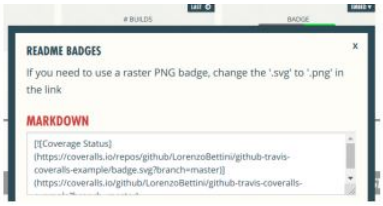


Figure 8: Getting the code for the Coveralls badge

Now that the README.md file has already been created, we can edit in GitHub using the "pencil" icon. The resulting README.md file should be rendered with both badges:



Figure 9: The README file with both badges

7

DOCKER

"Build, Ship and Run Any App, Anywhere", this is the motto with which the open source container platform promotes a flexible and not too costly alternative to the emulation of hardware components, based on virtual machines (VMs) . In our Docker tutorial for beginners we delve into the differences between the two virtualization technologies and introduce you to the open source Docker project with a clear step-by-step guide.

While classic hardware virtualization is based on booting several guest systems on a common host system, with Docker, thanks to its so-called containers, applications are launched as isolated processes on the same system. In this regard, there is talk of container-based virtualization and also of an operation system level virtualization.

7.0.1 *Container: virtualization with minimal overhead*

When applications are encapsulated within a classic hardware virtualization, this occurs by means of a so-called hypervisor, which acts as an abstraction layer between the host system and the various virtual guest systems. Each guest system is built as a complete machine with a separate kernel. The hardware resources of the host system (CPU, memory, hard disk space, peripheral units available) are proportionally allocated by the hypervisor.

With container-based virtualization, however, a complete guest system is not reproduced: the applications are launched inside the containers, which share the same kernel (that of the host system), however functioning as isolated processes within the user space.

The great advantage of container-based virtualization is that applications with different requirements can run in isolation from each other, without requiring the overhead of a separate guest system to be provided. To do this, the container technology uses two basic functions of the Linux Kernel: Control Groups (Cgroups) and the Kernel Namespaces:

Cgroups limit the access of processes to memory, CPU and I / O resources and thus prevent the resources needed for a process from compromising other processes in progress.

Namespaces (namespace) restrict a process and its child processes to a limited portion of the system on which they are based. In order to encapsulate processes, Docker uses Namespaces in five different areas:

System identificaion (UTS): in container-based virtualization UTS Namespaces are used in order to give each container its own host and domain name.

Process IDentifier (PID): Each Docker container uses its own namespace for identification processes. The processes performed outside a container are not visible from the inside. In this way, the processes encapsulated within containers on the same host system can have the same PIDs, without however creating a conflict.

Inter-Process Communication (IPC or also Inter-Process Communication): IPC namespaces isolate processes in a container so that communication with processes outside the container is prevented.

Network Neighborhood (NET): With network namespaces, you can assign separate network resources to each container, such as IP addresses and routing tables.

File System Mount Point (MNT): Thanks to the Mount namespaces, an isolated process never sees the entire file system of the host, but rather only a part of it, which usually corresponds to a specific image for this container.

Up to version 0.8.1, process isolation with Docker was based on Linux containers (LXC), but starting from version 0.9, users have access to the container format developed by Docker itself: Libcontainer. This allows

you to use Docker on various platforms and to run the same container on different host systems. In this way it has also become possible to offer a version of Docker for Windows and macOS.

7.0.2 *Scalability, high availability and portability*

Container technology is not just an alternative to classic hardware virtualization that is more attentive to saving resources: containers also allow applications to be used on different platforms and various infrastructures, without having to specifically adapt them to the hardware and software configurations of the respective host systems.

Docker uses so-called images as portable copies of containers. The images contain individual applications including all the libraries and all the binary and configuration files necessary for the duration of the encapsulated processes, thus minimizing the requests on the host system. In this way a container can be moved between Linux, Windows or macOS systems, without difficulty and without having to reconfigure. It is enough to install the Docker platform which will act as an abstraction layer. Docker is the ideal foundation for building scalable, high availability (HA) software architectures. Docker is in fact used in the production systems of companies such as Spotify, Google, eBay or Zalando.

7.0.3 *Docker: structure and functions*

Docker is the most popular software project among those that make container-based virtualization technology available to users. The open source platform is based on three fundamental components: to run a container, users only need the Docker Engine and the special Docker Images, which either refer to the Docker Hub or can be created independently.

7.0.4 *Docker Images*

Similar to virtual machines, Docker Containers are based on images. An Image is a non-editable template that contains all the information needed by the Docker Engine to create a container. As a portable copy of a container a Docker image is described as a text file, also referred to as a

Dockerfile. If a container is to be launched on a system, first a package with the corresponding image is loaded, as long as this is not already present locally. The uploaded image provides the data system necessary for the duration, together with all the parameters. A running container can be considered as an ongoing process of an Image.

7.0.5 *Docker Hub*

The Docker Hub is a cloud-based registry for repositories, or in easier-to-understand words a sort of Docker Image library. The online service is divided between public and private. The public section offers users the opportunity to upload images developed independently and share them with the community. A large number of official images from the Docker development team and established open source projects are available here. The images that are uploaded to the private section of the registry are not openly accessible but can be shared, for example, with people inside your company or in your circles of friends or acquaintances.

7.1 DOCKER- START

First of all, install Docker Community Edition (CE), following the instructions that can be found here https://docs.docker.com/install/. In this book we use Linux, and the installation depends on the distribution used.

When run from the command line, Docker commands have the shape

docker <command> <arguments>

For getting help, that is, the format of a specific command and the accepted command line argument, just type

docker <command> –help

In Linux, Docker commands must be run as super user, e.g., in Ubuntu, by using sudo, unless you add your user to the docker group, e.g., with

The first time you run this command, the latest version of the hello-world image is downloaded (and cached locally) and then a container running the image is started. The image is meant to only print a hello

```
sudo usermod -aG docker ${USER}
```

After that, you need to logout and login again. Then, you can run Docker with your own user.

Containers are executed with the command `run` and the name of the image. Let's run the `hello-world` with Docker

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
... feedback of downloading ...
Status: Downloaded newer image for hello-world:latest

Hello from Docker!

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

message and some initial documentation, before exiting. Note that it also summarizes what happened when you run this container with the docker run command.

The output also invites you to run an Ubuntu container with the command docker run -it ubuntu bash.

This shows two arguments for the run command:

```
    -i, --interactive                    Keep STDIN open even if not attached
    -t, --tty                            Allocate a pseudo-TTY
```

Finally, the last argument, bash, is the name of the executable that will be run in the container. Thus, after downloading the ubuntu image (if not already downloaded), you are provided with a new command prompt inside the container, since, with the -it command line arguments, we requested an interactive session with a terminal attached. The Bash

shell is executed in the container. Existing such a shell will also exit and terminate the container. In fact, containers run as long as their main process runs.

Let's try that (recall that the first time you will have to wait for the ubuntu image to be downloaded, that is, several hundreds of megabytes):

```
1 $ docker run -it ubuntu bash
2 root@...:/#
```

Note that the prompt has changed and the user name is now root; recall that from now on, all commands are executed in the container, until you exit the Bash shell.

Let's create a file inside the container and then we exit.

```
1 root@...:/# touch aFile.txt
2 root@...:/# ls aFile.txt
3 aFile.txt
4 root@...:/# exit
```

Now, we're back to the prompt of our OS.

Let's run exactly the same docker run command. This time the execution is almost immediate, since the image is not downloaded again. However, the previously created file is not in the container. That's because without additional command line arguments each time you use docker run, even with the image already used, a brand new container is started.

By default, each time a container is created, a random name is chosen by Docker. You can list the currently running container with the command docker ps and you can list also the containers that are not running with docker ps -a. The command shows the status of the container, the image, the executed command (bash for the previous example) and the name associated with the container (Docker creates random names with an adjective and the name of a notable person).

An explicit name can be given with the command line argument –name. For example,

```
$ docker run --name my-ubuntu-container -it ubuntu /bin/bash
root@...:/# touch aFile.txt
root@...:/# exit
```

Now, we could try to rerun the very same container specifying its name:

```
$ docker run --name my-ubuntu-container -it ubuntu /bin/bash
docker: Error response from daemon: Conflict.
The container name "/my-ubuntu-container" is already in use...
```

But that does not work, since `run` would try to create a new container with the same name of an existing one. If it is our intention to restart the previous container, we must do that explicitly with the command `docker restart` and then attach our terminal to the running container with `docker attach`. Let's run these commands and verify that the previously created file is still in the container:

```
$ docker restart my-ubuntu-container
$ docker attach my-ubuntu-container
root@...:/# ls aFile.txt
aFile.txt
root@...:/# exit
```

In general, if we are interested in persisting data generated by and used by Docker containers, the above strategy is not the recommended one. In fact, the solution is to use volumes, which are special directories used by one or several containers and are not subject to the Docker file system. Volumes are initialized when a container is created and they can be shared and reused by several containers They survive even when containers are deleted: Docker never deletes volumes neither it garbage collects them.

Containers are removed with the command docker rm, while images are removed with docker rmi. Since persisting data related to containers should be handled with volumes, containers should be considered ephemeral. Once terminated, containers still live in your file system. Thus, typically, containers (unless they are meant for volumes) should be started with the argument –rm, which automatically removes the container when it exits.

You can mount a local directory as a data volume with the syntax -v <host dir>:<container dir>. Now <host dir> is mounted in the container in the directory <container dir>. The host and the container will share the contents of that directory. For example

For example,

```
$ docker run --name myvolume -v /adirectory ubuntu
```

Creates a container with name `myvolume` with a volume in the directory `/adirectory` in the container. The command also returns after the creation since we do not specify any command to execute. Now the volume of this container can be used by other several containers:

```
$ docker run -it --volumes-from myvolume ubuntu /bin/bash
```

Executes the Ubuntu Bash in a container, using the volumes of the container `myvolume`. In this new container we will find the directory `/adirectory` shared with `myvolume`. Thus, what's written in the directory `/adirectory` will be persisted in the volume of the container `myvolume`. The directory `/adirectory` will never be removed even when no container is running. The directory will be removed only when the container `myvolume` is removed.

```
$ docker run --rm -it -v "$PWD"/mydir:/adirectory ubuntu /bin/bash
```

What's written in the container's `/adirectory` will end up in the directory `mydir` of the current directory in the local file system. (`$PWD` is the environment variable representing the current directory.)

7.2 DOCKER AND JAVA APPLICATION

In this section we see how to dockerize an application, that is, create a Docker image for an application. In our case, it is a Java application.

To keep things simple and focus on Docker, we will create a Maven project with the usual archetype and use the generated initial Java contents. That is, our Java application will be a plain "Hello World".

```
<project>
<modelVersion>4.0.0</modelVersion>

<groupId>com.examples</groupId>
<artifactId>hellodocker</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

```

```
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<maven.compiler.source>1.8</maven.compiler.source>
<maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
... JUnit as usual
</dependencies>
</project>
```

Listing 7.1: com.examples.txt

Let's run the Maven build so that the JAR is created: target/hellodocker-0.0.1-SNAPSHOT.jar. In Chapter Maven, Section Configuring the generated jar we saw how to configure the generated JAR to include information about the class with the main method; in Section Creating a FatJar we also saw how to include all dependencies in the JAR. For this very simple example, with no external dependencies, we rely on the default generated JAR. Thus, we run the generated JAR as follows:

```
java -cp target/hellodocker-0.0.1-SNAPSHOT.jar com.examples.hellodocker.App
```

7.2.1 *The Dockerfile*

A Dockerfile is a textual file with the instructions to create a Docker image using the docker build command. The syntax of this file is documented here https://docs.docker.com/engine/reference/ builder/. In this book we only use a minimal part of the syntax:

FROM

Defines the base image for our image (recall that images reuse other images' layers); the base image highly depends on the kind of application we want to dockerize, since the base image should provide enough software for the application. In our example, it makes sense to start from a Java base image.

COPY

Copies source files from the context into the file system of the container (we'll get back to the concept of context soon).

RUN

Executes a command; for example, this is typically used to install programs in the container or create configuration files. If the container is running in a Linux distribution based on apt, we can run apt commands to install Linux packages.

CMD

Default command to execute when executing the container. This can be overridden from the command line specifying the command to execute as the last argument (like we did for the ubuntu image, specifying the command bash)

The instructions of the Dockerfile can access only the directories of the context, when the image is built. The context directory is passed as argument to the build command. Thus, source directories of the COPY command are relative to the context directory (i.e., / is the root of the context). The context directory is processed recursively and all its contents

are sent to the Docker host for the build. A file .dockerignore can be used to exclude paths (similar to .gitignore), to increment build performance.

Let's now build a Docker image for our Java JAR.

A first possible attempt (which we'll see in a minute is not the optimal one) could be to start from the Ubuntu image (either the latest, or, by specifying the bionic tag, for the 18.04 release). We will then install the openjdk-8-jdk package from the official Ubuntu repositories. Finally, we copy the JAR file from the target directory into a directory of the container and specify the default command, i.e., we run the main class in the JAR.

For our Java application, the initial Dockerfile could be

```
1  FROM ubuntu:bionic
2
3  RUN apt-get update && \
4  apt-get install -y openjdk-8-jdk && \
5  apt-get clean
6
7  # Just to make sure Java can now be run in the container
8  RUN java -version
9
10  COPY /target/hellodocker-0.0.1-SNAPSHOT.jar /app/app.jar
11
12 CMD "java", "-cp", "/app/app.jar", "com.examples.hellodocker.App"
```

Listing 7.2: com.examples.hellodocker.txt

We create this file in the root directory of the project.

The RUN command executes the apt commands for installing the official Ubuntu package for OpenJDK 8 after updating the Ubuntu repositories. Recall that this command is meant to be executed in a Docker image running Ubuntu. We also then print the Java version just to make sure that Java can be executed when we run this image in a container. Then we copy the JAR generated by Maven into a directory of the image. Note that the source of COPY must be an absolute path, which will then be made relative to the context directory passed to the build command, as shown in the next paragraph. Finally, we specify that the default command when executing our image into a container is the execution

of our Java application (note how the command to execute is split in strings). Summarizing, all the above instructions of the Dockerfile will be executed during the build of the image, NOT when running the image in a container. The only exception is CMD, which will NOT be executed during the building of the image: it will be the default command that will be executed when running this image in a container.

We build our image with the docker build command, specifying the name of the image with the argument -t (followed by the name and optionally a tag in the name:tag format) and the path of the context as the last argument. We run the command from root directory of the project, where the Dockerfile was created and where the target file is also generated (recall that the source of the COPY command is relative to the context directory). Thus, the context is the current directory, ".". There's no need to specify the path of the Dockerfile, since, by default, is read from the directory where the docker build command is executed.

```
1  $ docker build -t java-hello-world .
2  Step 1/5 : FROM ubuntu:bionic
3  ... download the ubuntu:bionic image if not already downloaded...
4  Step 2/5 : RUN apt-get update && \
5  apt-get install -y openjdk-8-jdk && \
6  apt-get clean
7 ... run the apt commands, these will take a while:
8  ... the Ubuntu repositories are updated, all the packages are
     downloaded
9  ... and finally installed
10 Step 3/5 : RUN java -version
11  ---> Running in 39443451d56a
12  openjdk version "1.8.0_191"
13  OpenJDK Runtime Environment (build
     1.8.0_191-8u191-b12-0ubuntu0.16.04.1-b12)
14  OpenJDK 64-Bit Server VM (build 25.191-b12, mixed mode)
15  Step 4/5 : COPY /target/hellodocker-0.0.1-SNAPSHOT.jar /app/app.jar
16  ...
17  Step 5/5 : CMD "java", "-cp", "/app/app.jar",
     "com.examples.hellodocker.App"
18  ...
19  Successfully built ...
20  Successfully tagged java-hello-world:latest
```

Listing 7.3: com.examples.hellodocker.App.txt

We can manually test our image by running

```
$ docker run --rm java-hello-world
Hello World!
```

Listing 7.4: rm.txt

If we tried and build the image once again, we will see that Docker is smart enough to understand that no changes took place since the last build in the Dockerfile, thus it will rebuild the Docker image by reusing the layers: it will not even execute again the apt commands, nor it will print the Java version:

```
$ docker build -t java-hello-world .
...
Step 2/5 : RUN apt-get update && \
apt-get install -y openjdk-8-jdk && \
apt-get clean
---> Using cache
Step 3/5 : RUN java -version
---> Using cache
Step 4/5 : COPY /target/hellodocker-0.0.1-SNAPSHOT.jar /app/app.jar
---> Using cache
...
```

Listing 7.5: dockerbuild.txt

Note however that Docker also keeps track of possible changes to the sources of the COPY command: if we rebuild the Java JAR, then during the build of the image Docker will re-execute all the steps starting from COPY. This will ensure that the image we build will always contain the latest version of our JAR.

Besides that, even changing spaces in the RUN commands of the Dockerfile will invalidate these caches. For example, changing

```
...
RUN apt-get update && \
apt-get install -y openjdk-8-jdk && \
apt-get clean
...
```

```
6
7  into
8
9
10  ...
11  RUN apt-get update && apt-get install -y openjdk-8-jdk && \
12  apt-get clean
13  ...
```

Listing 7.6: apt-get.txt

will make the build re-execute the apt commands, and all the subsequent steps in the Dockerfile.

The approach we followed in the above Dockerfile, that is, starting from a generic Linux OS image and installing our required software, might not be optimal. Indeed, we should search right away if there is an already available Docker image that we can use as the base image for our image. Since we want to dockerize a Java application, we would need a Docker image where Java is already installed. In fact, we can start from the official Docker image for OpenJDK, openjdk. Since our Java application is meant for Java 8, we specify the tag 8

Our Dockerfile can then be simplified as follows:

```
1  FROM openjdk:8
2
3  COPY /target/hellodocker-0.0.1-SNAPSHOT.jar /app/app.jar
4
5  CMD "java", "-cp", "/app/app.jar", "com.examples.hellodocker.App"
```

Listing 7.7: com..examples.hellodocker.App.txt

Let's build the image again and verify that it still works.

```
1  $ docker build -t java-hello-world .
2  Sending build context to Docker daemon 29.18kB
3  Step 1/3 : FROM openjdk:8
4  8: Pulling from library/openjdk
5  ... the first time, the openjdk image is downloaded ...
6  Status: Downloaded newer image for openjdk:8
7  Step 2/3 : COPY /target/hellodocker-0.0.1-SNAPSHOT.jar /app/app.jar
```

```
Step 3/3 : CMD "java", "-cp", "/app/app.jar",
    "com.examples.hellodocker.App"
 ...
Successfully built ...
Successfully tagged java-hello-world:latest


 $ docker run --rm java-hello-world
 Hello World!
```

Listing 7.8: docker-build.txt

Remember that the context directory will be recursively sent to the Docker host for the build. In this example, all the target directory contents and the sources will be sent. For this project, this is not a huge problem since the sent contents are still small (in the output of the docker build we can see that it's less than 30kB)

8

DOCKER CONTINUE

8.0.1 *Build the Docker image from Maven*

Let's now configure Maven to build the image during the build. We will use the plugin io.fabric8:docker-maven-plugin, https://github.com/fabric8io/docker-maven-plugin. Here we see only a small part of the features provided by this plugin; we refer to the above URL for the complete manual.

First of all, we will configure this plugin in a separate profile, e.g., docker. As you saw, building a Docker image takes time, and we might want to avoid to build the Docker image for our application every time we run the Maven build.

```
1 1 <profiles>
2 2 <profile>
3 3 <id>docker</id>
4 4 <build>
5 5 <plugins>
6 6 ...
```

Listing 8.1: docker.txt

We configure the plugin so that the build goal is executed in the package phase; this goal will be executed after the JAR has already been created by the default Maven plugin maven-jar-plugin. Then, we bind the run goal to the verify phase, so that our built image is executed in a container. It is crucial to also bind stop to the verify phase so that the container is also stopped and removed (otherwise, further Maven executions will fail due to a conflict with a container already existing):

```
1 <plugin>
2 <groupId>io.fabric8</groupId>
```

```
<artifactId>docker-maven-plugin</artifactId>
<version>0.28.0</version>
<configuration>
<showLogs>true</showLogs>
<images>
<image>
<name>java-hello-java</name>
<build>
<dockerFileDir>${project.basedir}</dockerFileDir>
</build>
<run>
<wait>
<log>Hello World!</log>
<exit>0</exit>
<time>10000</time>
</wait>
</run>
</image>
</images>
</configuration>
<executions>
<execution>
<id>docker:build</id>
<phase>package</phase>
<goals>
<goal>build</goal>
</goals>
</execution>
<execution>
<id>docker:start</id>
<phase>verify</phase>
<goals>
<goal>start</goal>
<goal>stop</goal>
</goals>
</execution>
</executions>
</plugin>
```

Listing 8.2: dockerPlugin.txt

In the general configuration section, using <showLogs>, we require to print out all standard output and standard error messages for all

containers started (each container's output will be prefixed with the container ID). The binding of goals to phases in the <executions> element has already been explained. Let's now see in details how the other parts of this plugin are configured. The <images> element lists all the Docker images involved in the build. In this example we only deal with our own image. We have to specify how to build it and how to run it. Of course, the <build> configuration will be used by the build goal and the <run> configuration will be used by the start and stop goals .

The <build> element can be used to directly specify the build steps in the configuration. However, we already created a Dockerfile and we aim at reusing this, thus, we simply specify the path where the Dockerfile is located (i.e., in the project's base directory).

The <run> element allows us to specify how containers should be created and run. In our case, we simply want to execute the container with the default CMD. We also verify that everything works by waiting for the string printed by our application to appear and for the application to exit successfully (exit code 0), within 10 seconds. This timeout should be enough for our simple Java app to start and print the message on the screen and exit. If that does not happen, the build will fail (during the verify phase).

Let's run the Maven build enabling this profile (either form the command line or from Eclipse):

```
$ mvn clean verify -Pdocker
[INFO] --- docker-maven-plugin:0.28.0:build (docker:build) @ hellodocker ---
[INFO] Building tar: ...target/docker/java-hello-java/tmp/docker-build.tar
[INFO] DOCKER> [java-hello-java:latest]: Created docker-build.tar in 34 milliseconds
[INFO] DOCKER> [java-hello-java:latest]: Built image sha256:...
[INFO] DOCKER> [java-hello-java:latest]: Removed old image sha256:...
[INFO]
[INFO] --- docker-maven-plugin:0.28.0:start (docker:start) @ hellodocker ---
[INFO] DOCKER> [java-hello-java:latest]: Start container ab20b605bef8
[INFO] DOCKER> Pattern 'Hello World!' matched for container ab20b605bef8
ab20b6> Hello World!
[INFO] DOCKER> [java-hello-java:latest]: Waited on log out 'Hello World!' and on exi\
t code 0 509 ms
[INFO]
[INFO] --- docker-maven-plugin:0.28.0:stop (docker:start) @ hellodocker ---
[INFO] DOCKER> [java-hello-java:latest]: Stop and removed container ... after 0 ms
```

We can see that the image is built and during the verify phase the container is executed correctly, printing on the screen the expected string and exiting successfully. The container ID is used, in an abbreviated form, to identify its printed output. Of course, the ID will be different across builds. When the plugin builds the Docker image, all the files and directories located in the dockerFileDir directory will be added to the build context. In fact, this plugin does not take the .dockerignore into consideration. You can have a look at the target subdirectory where the plugin stores the files for the build:

```
target/docker
├── build.timestamp
└── java-hello-java
    ├── build
    │   └── Dockerfile
    ├── tmp
    │   └── docker-build.tar
    └── work
```

You can verify that the docker-build.tar contains all the contents of our project base directory. The documentation of the plugin provides several solutions for refining the contents for the build context. In our example, it is enough to place in the directory dockerFileDir (that is, the project base directory), the file .maven-dockerinclude, listing only the files for the context build:

```
target/*.jar
```

Run the build again. It still succeeds, and now the .tar file only contains the JAR (and the Dockerfile).

Before concluding this section, let's go back to our Dockerfile, in particular, this command:

```
COPY /target/hellodocker-0.0.1-SNAPSHOT.jar /app/app.jar
```

The name of the JAR to copy from our system into the image is hardcoded, and, most of all, it includes the version of the artifact. If we changed the version in the POM, we should also remember to change the version in the Dockerfile or the build will fail.

In this very example, simply using a wildcard of the shape *.jar would be enough. However, let's see another feature of the Dockerfile that allows us to be more generic.

We use the ARG instruction, which declares a variable to be used in the rest of the Dockerfile. The value for this variable must be passed at build-time using the –build-arg <varname>=<value>:

```
FROM openjdk:8

ARG jarToCopy

COPY /target/$jarToCopy /app/app.jar

CMD ["java", "-cp", "/app/app.jar", "com.examples.hellodocker.App"]
```

Now, when we build the image we must pass a value for `jarToCopy`:

```
docker build --build-arg jarToCopy=hellodocker-0.0.1-SNAPSHOT.jar \
    -t java-hello-world .
```

While this is tedious from the command line, in Maven we can use the standard properties to refer to the produced artifact: project.artifactId-project.version.jar. We then use the <args> element of docker-maven-plugin when building from Maven:

```
1 ...
2 <build>
3 <dockerFileDir>${project.basedir}</dockerFileDir>
4 <args>
5
    <jarToCopy>${project.artifactId}-${project.version}.jar</jarToCopy>
6 </args>
7 </build>
8 ...
```

Listing 8.3: project.basedir.txt

If we changed the project version or even the artifact identifier our Dockerfile would still be valid.

9

INTEGRATION TESTS

To build this confidence, we must have a framework for automatic regression testing. For doing regression testing, there are many tests that should be carried out from an API-level point of view, but here we'll cover two major types of tests.

Unit testing
where any given test covers the smallest unit of a program (a function or procedure). It may or may not take some input parameters and may or may not return some values.

Integration testing

where individual units are tested together to check whether all the units interact with each other as expected.

It is also important to keep unit and integration tests separate, in particular, during compilation execution. Also, unit tests must be run after any code changes, during integration tests may run less often, especially if they take much longer than unit tests. Typically, integration tests are performed on the CI server.

Finally, it usually doesn't make sense to run integration tests if the unit tests fail - integration tests are it should also fail, since, if the individual components do not perform properly in isolation and in a mocked environment, they are likely to fail if integrated together and with external services.

As we will see in the examples in this chapter, integration tests do not necessarily have to be tested the whole application: as long as at least some real dependencies are used, we consider a test a integration test.

Note that we still use JUnit to write integration tests. The "Unit" in the name is not meant for relegate JUnit to unit tests only.

Integration tests, in addition to external components, should connect our components (e.g. objects of our classes). Integration tests must be written after the individual components involved in the file test has been fully tested, with unit tests. Therefore, the individual components are known to function correctly in insulation and are now being tested together. Integration tests are usually expected to be successful right away, as they should verify that the components already tested still work together.

9.0.1 *Example*

To get familiar with integration testing, we'll implement some simple interacting classes together in a Model-View-Controller (MVC) architecture.

In this chapter we do not implement the "view" part: we will focus on the controller, which acts as a bridge between the view and the data store of our model. We will implement the view in User interface test chapter.

As done in the Mocking Chapter, Mockito Section: a tutorial, we use the Repository pattern (Eva03) to access to the database. In that chapter, we didn't implement the repository - in unit tests we were mocking and stubbing his methods. In this chapter, we use mockery for the repository when writing the unit tests for the controller. However, we also develop a repository implementation that it will be used in integration tests.

In this example, we're using Mockito and AssertJ as our test dependencies.

The only domain model class is the Student class:

```
1  package com.examples.school.model;
2
3  public class Student {
4  private String id;
5  private String name;
6
7  public Student() {}
8
9  public Student(String id, String name) {
10 this.id = id;
11 this.name = name;
12 }
```

```
// getters, setters, equals, hashCode, toString...
}
```

Listing 9.1: gett.c

The repository interface is as follows:

```
package com.examples.school.repository;
...
public interface StudentRepository {
public List<Student> findAll();
public Student findById(String id);
public void save(Student student);
public void delete(String id);
}
```

Listing 9.2: StudentRepository.java

The view interface is as follows:

```
package com.examples.school.view;
..
public interface StudentView {
void showAllStudents(List<Student> students);
void showError(String message, Student student);
void studentAdded(Student student);
void studentRemoved(Student student);
}
```

Listing 9.3: StudentView.java

Finally, we have the following implementation of the controller:

```
package com.examples.school.controller;
...
public class SchoolController {
private StudentView studentView;
private StudentRepository studentRepository;

public SchoolController(StudentView studentView,
StudentRepository studentRepository) {
this.studentView = studentView;
this.studentRepository = studentRepository;
}

```

```
13 public void allStudents() {
14 studentView.showAllStudents(studentRepository.findAll());
15 }
16
17 public void newStudent(Student student) {
18 Student existingStudent =
       studentRepository.findById(student.getId());
19 if (existingStudent != null) {
20 studentView.showError("Already existing student with id " +
       student.getId(),
21 existingStudent);
22 return;
23 }
24
25 studentRepository.save(student);
26  studentView.studentAdded(student);
27 }
28
29 public void deleteStudent(Student student) {
30 if (studentRepository.findById(student.getId()) == null) {
31 studentView.showError("No existing student with id " +
       student.getId(),
32  student);
33 return;
34 }
35
36 studentRepository.delete(student.getId());
37 studentView.studentRemoved(student);
38 }
39
40 }
```

Listing 9.4: SchoolController.java

Thus, the controller processes user requests from the view by delegating database operations to the repository (both read and write operations) and presents the results to the user by delegating to the view.

```
1 package com.examples.school.controller;
2 // ... imports and static imports as usual
3 public class SchoolControllerTest {
4
5 @Mock
```

```
6  private StudentRepository studentRepository;
7
8   @Mock
9  private StudentView studentView;
10
11 @InjectMocks
12 private SchoolController schoolController;
13
14 @Before
15 public void setUp() throws Exception {
16 MockitoAnnotations.initMocks(this);
17  }
18
19 @Test
20  public void testAllStudents() {
21  List<Student> students = asList(new Student());
22  when(studentRepository.findAll())
23 thenReturn(students);
24  schoolController.allStudents();
25 verify(studentView).showAllStudents(students);
26 }
27
28  @Test
29 public void testNewStudentWhenStudentDoesNotAlreadyExist() {
30 Student student = new Student("1", "test");
31  when(studentRepository.findById("1")).
32 thenReturn(null);
33  schoolController.newStudent(student);
34 InOrder inOrder = inOrder(studentRepository, studentView);
35 inOrder.verify(studentRepository).save(student);
36 inOrder.verify(studentView).studentAdded(student);
37 }
38
39 @Test
40 public void testNewStudentWhenStudentAlreadyExists() {
41  Student studentToAdd = new Student("1", "test");
42  Student existingStudent = new Student("1", "name");
43  when(studentRepository.findById("1")).
44 thenReturn(existingStudent);
45 schoolController.newStudent(studentToAdd);
46 verify(studentView)
47 showError("Already existing student with id 1", existingStudent);
```

```
48 verifyNoMoreInteractions(ignoreStubs(studentRepository));
49 }
50
51  @Test
52  public void testDeleteStudentWhenStudentExists() {
53 Student studentToDelete = new Student("1", "test");
54  when(studentRepository.findById("1")).
55  thenReturn(studentToDelete);
56 schoolController.deleteStudent(studentToDelete);
57  InOrder inOrder = inOrder(studentRepository, studentView);
58 inOrder.verify(studentRepository).delete("1");
59 inOrder.verify(studentView).studentRemoved(studentToDelete);
60 }
61
62 @Test
63 public void testDeleteStudentWhenStudentDoesNotExist() {
64 Student student = new Student("1", "test");
65 when(studentRepository.findById("1")).
66 thenReturn(null);
67 schoolController.deleteStudent(student);
68 verify(studentView)
69  .showError("No existing student with id 1", student);
70 verifyNoMoreInteractions(ignoreStubs(studentRepository));
71 }
72 }
```

Listing 9.5: SchoolControllerTest.java

In the test testNewStudentWhenStudentAlreadyExists we use Mockito ignoreStubs when verifying that there are no more interactions with the repository. Without this API, the verification would fail because we actually interacted with the repository in the controller by calling the stubbed method findById. With ignoreStubs we instruct Mockito to ignore the call to the stubbed method when performing the verification.

9.1 UNIT TESTS WITH DATABASES

Now we want to write a StudentRepository implementation using MongoDB, StudentMongoRepository.

We have already partially used this database in the Docker Chapter, Docker Networks Section. MongoDB, https://www.mongodb.com/, is a NoSQL database. It is document oriented instead of relying tables

and relationships as relational databases. We use it in this chapter since it is simple setting up. For example, unlike MySQL or PostgreSQL, it requires no configuration. From the focus of this chapter is on integration testing, not database programming, using such a NoSQL database allows us to focus on the topic of the chapter, without spending too much time on the database level. As we anticipated in the previous section, let's skip further advanced information topics in database programming such as transactions and automatic primary key management. Indeed, implementing a database-based application requires several adjustments, which we will not see in this book.

Here we only use the basic functionality of MongoDB. (We already used MongoDB in the chapter Docker, to present a Docker use case.) To make the code understandable, let's also give a lot minimal introduction to Mongo's Java API.

First of all, we need to add this dependency in order to use Mongo's Java API:

```
<dependency>
<groupId>org.mongodb</groupId>
<artifactId>mongo-java-driver</artifactId>
<version>3.9.1</version>
</dependency>
<dependency>
<!-- required to see Mongo Java Driver logs -->
<groupId>ch.qos.logback</groupId>
<artifactId>logback-classic</artifactId>
 <version>1.2.3</version>
</dependency>
```

Listing 9.6: dependency.c

The logging dependency is required to see the logs of communications with the database. We must create a MongoClient object. MongoClient represents a client communicating with the MongoDB server. This class has several constructors, e.g., accepting the host of the MongoDB server, the port (by default 27017), etc.

// create the connection with the server

MongoClient mongoClient = new MongoClient(mongoHost);

Then, we must get access to a database, providing its name, and then to a collection, which basically corresponds to a table in a SQL database

(non-existing databases and collections are created on-thefly, without any schema):

MongoDatabase db = mongoClient.getDatabase("mydb");

MongoCollection<Document> collection = db.getCollection("examples");

```
// create the connection with the server
MongoClient mongoClient = new MongoClient(mongoHost);
MongoDatabase db = mongoClient.getDatabase("school");
MongoCollection<Document> collection = db.getCollection("student");
Document doc = new Document("id", "id")
.append("name", "A student");
 collection.insertOne(doc);
collection.find().first().get("id");
```

Listing 9.7: connection.c

Let's create StudentMongoRepository implementing StudentRepository with the Eclipse wizard having all interface methods declared in the class with empty implementations.

We want to pass a MongoClient to the constructor so that we can create the client from outside and inject it in StudentMongoRepository. This allows us to create and configure MongoClient as we see fit, e.g., for testing purposes. Then we retrieve the collection student from the database school and store it in a field. Our repository implementation will use directly this collection.

```
package com.examples.school.repository.mongo;

import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.client.MongoCollection;
...
public class StudentMongoRepository implements StudentRepository {

public static final String SCHOOL_DB_NAME = "school";
public static final String STUDENT_COLLECTION_NAME = "student";
private MongoCollection<Document> studentCollection;

public StudentMongoRepository(MongoClient client) {
studentCollection = client
.getDatabase(SCHOOL_DB_NAME)
.getCollection(STUDENT_COLLECTION_NAME);
```

```
17 }
```

Listing 9.8: StudentMongoRepository.java

9.1.1 *Integration tests*

Now we start writing some integration tests. We know that, by convention, unit test Java classes should mention the SUT class and end with Test. For integration tests, the convention is to use the suffix IT (for "Integration Test"), instead of Test. We will follow this convention.

9.1.2 *Source folder for integration tests*

While not strictly required, it might be good to keep integration tests into a separate source folder. This allows us to keep our tests well organized, and, from Eclipse, we can easily run all unit tests and all integration tests in different development stages.

Typically, the source folder for integration tests is src/it/java. So let's create this folder in our project.

Unfortunately, Maven does not automatically handles such a source folder. We must add it manually using the Build Helper Maven Plugin, https://www.mojohaus.org/build-helper-maven-plugin/. This is a very useful plugin since it provides many goals to assist with the Maven build lifecycle. We use the goal add-test-source to add another test source folder to our project (we bind this goal to the phase generate-test-sources, which is executed before compiling and running tests):

```
1  <plugin>
2  <groupId>org.codehaus.mojo</groupId>
3  <artifactId>build-helper-maven-plugin</artifactId>
4   <version>3.0.0</version>
5   <executions>
6  <execution>
7   <id>add-test-source</id>
8   <phase>generate-test-sources</phase>
9  <goals>
10 <goal>add-test-source</goal>
11  </goals>
12  <configuration>
```

```
 <sources>
<source>src/it/java</source>
 </sources>
 </configuration>
 </execution>
 </executions>
 </plugin>
```

Listing 9.9: build-helper-maven-plugin.txt

Then we update the Maven project from Eclipse; the new folder will be now handled as a test source folder.

9.1.3 *Docker and Testcontainers*

The library Testcontainers, https://www.testcontainers.org/, provides several mechanisms to start throwaway instances of Docker containers directly from JUnit tests. It also provides several additional modules for running containers with mainstream databases like MySQL and PostgreSQL. When there is no such specific module, like for our case, MongoDB, we can still run a generic container, using GenericContainer.

We first need to add this test dependency in our POM:

```
<dependency>
<groupId>org.testcontainers</groupId>
<artifactId>testcontainers</artifactId>
<version>1.10.5</version>
<scope>test</scope>
</dependency>
```

Listing 9.10: testcontainers.txt

For demonstration, let's start writing an integration test for our StudentMongoRepository, StudentMongoRepositoryTestcontainersIT, in the folder src/it/java, by using Testcontainers:

```
 package com.examples.school.repository.mongo;

 import org.junit.ClassRule;
 import org.testcontainers.containers.GenericContainer;
 ...
 public class StudentMongoRepositoryTestcontainersIT {
```

```
7
8  @SuppressWarnings("rawtypes")
9  @ClassRule
10 public static final GenericContainer mongo =
11 new GenericContainer("mongo:4.2.3")
12 .withExposedPorts(27017);
13
14 private MongoClient client;
15 private StudentMongoRepository studentRepository;
16 private MongoCollection<Document> studentCollection;
17
18 @Before
19 public void setup() {
20 client = new MongoClient(
21 new ServerAddress(
22 mongo.getContainerIpAddress(),
23 mongo.getMappedPort(27017)));
24 studentRepository = new StudentMongoRepository(client);
25 MongoDatabase database = client.getDatabase(SCHOOL_DB_NAME);
26 // make sure we always start with a clean database
27 database.drop();
28 studentCollection = database.getCollection(STUDENT_COLLECTION_NAME);
29 }
30
31 @After
32 public void tearDown() {
33 client.close();
34 }
35
36 @Test
37 public void test() {
38 // just to check that we can connect to the container
39 }
```

Listing 9.11: StudentMongoRepositoryTestcontainersIT.java

The GenericContainer JUnit rule will execute the specified image in a container, before executing all tests and it will automatically stop and remove such a container when all tests are run. We configure such a container, e.g., by exposing ports. We retrieve the IP of the container and the container's port mapped to a random port of the host with getContainerIpAddress and getMappedPort, respectively. We then use such information for creating a MongoClient that will connect to the real

MongoDB server running in the container. The initial fake test is just to verify that we can connect to the database server.

Now we can write tests that access to a real MongoDB server.

It does not make sense to duplicate all our previous unit tests. Unit tests are meant to test all paths of the SUT, e.g., for findById, both the case when the document is found and when it is not found. Integration tests instead could simply test the positive cases. Indeed, as prescribed by the test pyramid, integration tests should be much less than unit tests. In our example, we have 4 integration tests, one for each method testing only the positive path, and 6 unit tests:

```
1  @Test
2  public void testFindAll() {
3  addTestStudentToDatabase("1", "test1");
4  addTestStudentToDatabase("2", "test2");
5  assertThat(studentRepository.findAll())
6 .containsExactly(
7 new Student("1", "test1"),
8 new Student("2", "test2"));
9  }
10
11  @Test
12  public void testFindById() {
13 addTestStudentToDatabase("1", "test1");
14  addTestStudentToDatabase("2", "test2");
15  assertThat(studentRepository.findById("2"))
16  .isEqualTo(new Student("2", "test2"));
17 }
18  ...
19
20
21  // testSave and testDelete are similar
```

Listing 9.12: testFindAll.java

9.1.4 *Tests with Maven*

As we already know, the plugin for unit tests is surefire, whose goal test is automatically bound to the phase test. In this phase only unit tests should be executed.

Maven, in its default lifecycle, has 4 phases and a specific plugin, maven-failsafe-plugin, for integration tests. These are the 4 phases involved in integration tests:

pre-integration-test

for setting up the integration test environment, e.g., starting a server needed by integration tests.

integration-test

where integration tests are effectively run.

post-integration-test

for cleaning the integration test environment, e.g., stopping a server used by integration tests.

verify

where results of integration tests are checked.

As documented here these phases are executed after package, which, in turn, is executed after test. Thus, they are executed after unit tests have already been executed with success. The name failsafe is due to the fact that the build does not fail immediately if integration tests fail (differently from surefire). Thus, when it fails, it does so in a "safe" way. If the build stopped when an integration test fails, during the phase integration-test, then the phase post-integration-test would not be executed and the additional resources started in pre-integration-test would still be running. Instead, in case of integration test failures, the build would fail in the phase verify, that is, after post-integration-test, where we have the chance to stop and clean resources allocated during pre-integration-test.

Differently from surefire, failsafe is not enabled by default. If we want to run integration tests we must configure the plugin explicitly, as we will see in a minute. The plugin, once configured, executes all the tests that match these file patterns and that are not abstract classes: **/IT*.java, **/*IT.java, **/*ITCase.java. As shown in Chapter Maven, surefire will not run such tests, since its file patterns are **/Test*.java, **/*Test.java, **/*Tests.java, **/*TestCase.java. Thus, following the name conventions for test cases, as we also did in the previous section, allows failsafe to automatically run our integration tests, once configured.

The pre-integration-test and post-integration-test phases are typically used by other plugins to start and stop, respectively, the execution environment for integration tests. We use them in the next section, as you can

imagine, for running and stopping a Docker container with the docker-maven-plugin. For the moment, we will not use pre and post integration test phases. In fact, our integration tests rely on a Docker container started automatically by Testcontainers. Thus, to enable failsafe it is enough to enable its goals, integration-test and verify. These goals are automatically bound to the homonymous phases of the default lifecycle:

```
1  <plugin>
2  <groupId>org.apache.maven.plugins</groupId>
3  <artifactId>maven-failsafe-plugin</artifactId>
4  <version>2.22.1</version>
5  <executions>
6  <execution>
7  <goals>
8  <goal>integration-test</goal>
9 <goal>verify</goal>
10  </goals>
11  </execution>
12  </executions>
13  </plugin>
```

Listing 9.13: integration-test.txt

Now, running mvn test will only run unit tests while running mvn verify will also run integration tests, after unit tests have passed.

9.1.5 *tests with Docker and Maven*

Let's now write a few integration tests for the StudentController. In these tests, the controller is meant to interact with a real StudentRepository implementation, i.e., StudentMongoRepository. For the sake of demonstration, we do not use Testcontainers in these integration tests.

If we had an implementation of the interface StudentView we could make the controller interact with such an implementation as well. For the moment, we don't have such an implementation, thus we still mock the view. In any case, an integration test does not necessarily have to use real implementations for all the dependencies of the SUT. Thus, such an integration test for the controller still makes sense even when mocking the view.

The initial part of such an integration test case is as follows (recall that we place this class in src/it/java):

```
1  package com.examples.school.controller;
2  ...
3  public class SchoolControllerIT {
4
5  @Mock
6  private StudentView studentView;
7
8  private StudentRepository studentRepository;
9
10 private SchoolController schoolController;
11
12 @Before
13 public void setUp() {
14 MockitoAnnotations.initMocks(this);
15 studentRepository = new StudentMongoRepository(new
     MongoClient("localhost"));
16 // explicit empty the database through the repository
17 for (Student student : studentRepository.findAll()) {
18 studentRepository.delete(student.getId());
19 }
20 schoolController = new SchoolController(studentView,
     studentRepository);
21 }
22 ...
```

Listing 9.14: SchoolControllerIT.java

In the @Before method we create a StudentMongoRepository passing a MongoClient that will connect to a real MongoDB server (meant to be executing on the localhost on the default port). We use the repository to interact with the database, e.g., to make sure we always start with an empty collection. Note that, besides the creation of MongoClient, this test ignores further details of the database structure: our controller interact only with a StudentRepository implementation.

The repository is also used to prepare the context for our controller integration tests, e.g., for populating the database. Recall that, as done before, if we are testing a SUT method that reads from the database we must not insert test elements in the database through the SUT by calling a method that writes. The latter might not be implemented yet and, most of

all, we must test the logic of a method independently from other methods of the same SUT.

As done previously, we do not duplicate all our previous unit tests of the controller in the integration tests. Integration tests instead could simply test the positive cases. In our example, we will have 3 integration tests, one for each controller's method testing only the positive path, instead of the 5 unit tests:

```
1  @Test
2  public void testAllStudents() {
3 Student student = new Student("1", "test");
4  studentRepository.save(student);
5  schoolController.allStudents();
6  verify(studentView)
7  .showAllStudents(asList(student));
8  }
9
10 @Test
11 public void testNewStudent() {
12 Student student = new Student("1", "test");
13 schoolController.newStudent(student);
14 verify(studentView).studentAdded(student);
15 }
16
17 @Test
18 public void testDeleteStudent() {
19 Student studentToDelete = new Student("1", "test");
20 studentRepository.save(studentToDelete);
21 schoolController.deleteStudent(studentToDelete);
22 verify(studentView).studentRemoved(studentToDelete);
23 }
```

Listing 9.15: testAllStudents.java

Note that we still use Mockito for verifying the expected behavior of our controller, while interacting with a real database.

Now, these tests need to communicate with a MongoDB server. If we run them from Eclipse, it is our responsibility to first start manually a Docker container for MongoDB (this was not required when using Testcontainers), publishing the container's port to the local system. For example:

```
docker run -p 27017:27017 --rm mongo:4.2.3
```

Listing 9.16: docker-run.txt

Now we can run these new integration tests and verify that they succeed.

While when running these tests from Eclipse it is fine to manually start the Docker container, when running the Maven build, the container must be started directly during the build, before running integration tests (or at least, before this very integration test case). Similarly, the container must be stopped during the build after integration tests are run, independently from whether they succeeded or not.

To achieve that, all we need to do is to configure the docker-maven-plugin (introduced in Chapter Docker, Section Build the Docker image from Maven) binding its start and stop goals to pre-integration-test and post-integration-test phases, respectively. Moreover, we configure the image for MongoDB, as already done in Chapter Docker. This time, we do not build any Docker image.

```
<plugin>
<groupId>io.fabric8</groupId>
<artifactId>docker-maven-plugin</artifactId>
<version>0.27.2</version>
<configuration>
<images>
<image>
<name>mongo:4.2.3</name>
<run>
<ports>
<port>27017:27017</port>
</ports>
</run>
</image>
</images>
</configuration>
<executions>
<execution>
<id>docker:start</id>
<phase>pre-integration-test</phase>
<goals>
<goal>start</goal>
</goals>
```

```
 </execution>
 <execution>
<id>docker:stop</id>
 <phase>post-integration-test</phase>
 <goals>
 <goal>stop</goal>
 </goals>
 </execution>
 </executions>
 </plugin>
```

Listing 9.17: pre-integration-test.txt

Note that we must explicitly publish the 27017 container's port to the 27017 local port, since our controller integration tests will connect to that port.

If we now run

mvn clean verify

All our integration tests will be executed, and the controller integration tests will communicate with the Docker container started during the build. Since Testcontainers publishes the port of the container to a local random free port, the two Docker containers will not interfere with each other. Similarly to what we did in Chapter Maven, Section Run Maven goals, we can generate both unit test reports and integration test reports by running

```
mvn clean verify \
surefire-report:report-only \
surefire-report:failsafe-report-only \
org.apache.maven.plugins:maven-site-plugin:3.7.1:site \
-DgenerateReports=false
```

Listing 9.18: surefire-report.txt

We find the reports in target/site/surefire-report.html and target/site/failsafe-report.html. This also gives us an idea of the time it takes to run several unit tests (probably not even a second) and to run a few integration tests (probably several seconds).

9.1.6 *integration tests in Travis CI*

Once the Maven build is setup, running this build in Travis is just a matter of creating the .travis.yml file and setup Travis adding the GitHub repository of this example. This procedure has already been illustrated in Chapter Continuous Integration. In Chapter Docker, Section Using Docker in Travis CI we saw how to enable Docker in Travis (currently only supported in Linux environments).

The Travis configuration file is then

```
language: java

jdk: openjdk8

services:
- docker

# skip installation step
install: true

cache:
directories:
- $HOME/.m2

script:
- mvn -f com.examples.school/pom.xml clean verify
```

Listing 9.19: travisYml.txt

10

GRAPHICAL USER INTERFACE TESTING

In software engineering, graphical user interface testing is the process of testing a product's graphical user interface to ensure it meets its specifications. This is normally done through the use of a variety of test cases.

10.1 WHAT DO YOU CHECK-IN GUI TESTING?

The following checklist will ensure detailed GUI Testing in Software Testing.

Check all the GUI elements for size, position, width, length, and acceptance of characters or numbers. For instance, you must be able to provide inputs to the input fields.

Check you can execute the intended functionality of the application using the GUI

Check Error Messages are displayed correctly

Check for Clear demarcation of different sections on screen

Check Font used in an application is readable

Check the alignment of the text is proper

Check the Color of the font and warning messages is aesthetically pleasing

Check that the images have good clarity

Check that the images are properly aligned

Check the positioning of GUI elements for different screen resolution.

10.1.1 *Example GUI Testing Test Cases*

GUI Testing basically involves

Testing the size, position, width, height of the elements.

Testing of the error messages that are getting displayed.

Testing the different sections of the screen.

Testing of the font whether it is readable or not.

Testing of the screen in different resolutions with the help of zooming in and zooming out like 640 x 480, 600x800, etc.

Testing the alignment of the texts and other elements like icons, buttons, etc. are in proper place or not.

Testing the colors of the fonts.

Testing the colors of the error messages, warning messages.

Testing whether the image has good clarity or not.

Testing the alignment of the images.

Testing of the spelling.

The user must not get frustrated while using the system interface.

Testing whether the interface is attractive or not.

Testing of the scrollbars according to the size of the page if any.

Testing of the disabled fields if any.

Testing of the size of the images.

Testing of the headings whether it is properly aligned or not.

Testing of the color of the hyperlink.

10.1.2 *Challenges in GUI Testing*

In Software Engineering, the most common problem while doing Regression Testing is that the application GUI changes frequently. It is very difficult to test and identify whether it is an issue or enhancement. The problem manifests when you don't have any documents regarding GUI changes.

10.1.3 *GUI Testing Tools*

Following is a list of popular GUI Testing Tools :
Ranorex

Selenium

QTP

Cucumber

SilkTest

TestComplete

Squish GUI Tester
Click here to learn Selenium, QTP and Cucumber.

10.2 CONSIDER USING A BDD FRAMEWORK

BDD is a software development methodology in which software is implemented in the way its behavior is described. BDD can be applied for any type of testing including unit tests, component, integration as well as for many other types. UI testing is one of the main areas where BDD can be applied with a great success. BDD is recommended for UI Automation, for many reasons.

First of all, BDD is a methodology that helps teams understand each other, creating strong outside and inside team collaboration. By writing your tests with BDD you also create specifications that can help your team understand tests and requirements much better. This means that along with writing your tests, you are creating a clear tests documentation. This ensures you don't waste other team members' time (who might work on your tests later), as well as your own time, because you don't need to explain and help with such tests if they are unclear.

Second, BDD helps the Business side (e.g.Test and Project managers) understand these tests. This brings additional value to testing because they can make recommendations based on business benefits.

Third, BDD usually forces you to follow a strict code organization pattern, which helps avoid code duplication. This is done by having separate components called steps or actions that will be the building blocks for your tests.

Now let's look at a simple example. Assume we want to write a very small test that verifies a basic workflow on the blazedemo.com website. Try to go over the test scenario and understand what it does:

```
HomePage homePage = new HomePage();

    homePage.open();
    homePage.waitUntilPageLoaded();
    Assert.assertEquals(homePage.getTitle(), "BlazeDemo");
    Assert.assertEquals(homePage.getTitle(), "Welcome to the Simple
     Travel Agency!");

    FlightsPage flightsPage = homePage.findFlights("Boston","New
     York");
    List<String> foundFlights = flightsPage.getFlights();

    Assert.assertTrue(foundFlights.size() > 0);
```

Listing 10.1: homePage.java

That was easy, right? But let's imagine that you are the manager - how easy would it be for you to read such tests? Or let's assume that you are a new team member and you have to go over existing tests to understand what they do. Would you like to go over such tests or maybe you would be happier to see the same tests, written like this:

The second example is the same test written in BDD style using Gherkin line-oriented language in one of the most famous BDD frameworks - Cucumber.

Even if you do not prefer to write tests in human-readable text files, there are plenty of options for applying the BDD model to your tests, no matter which programming language they are written in. For example, you can even put BDD style comments in your code yourself:

```
@Test
public void someTest() {
    // given
    Something something = getSomething();
=    // when
    something.doSomething();
    // then
    assertSomething();
}
```

Listing 10.2: someTest.java

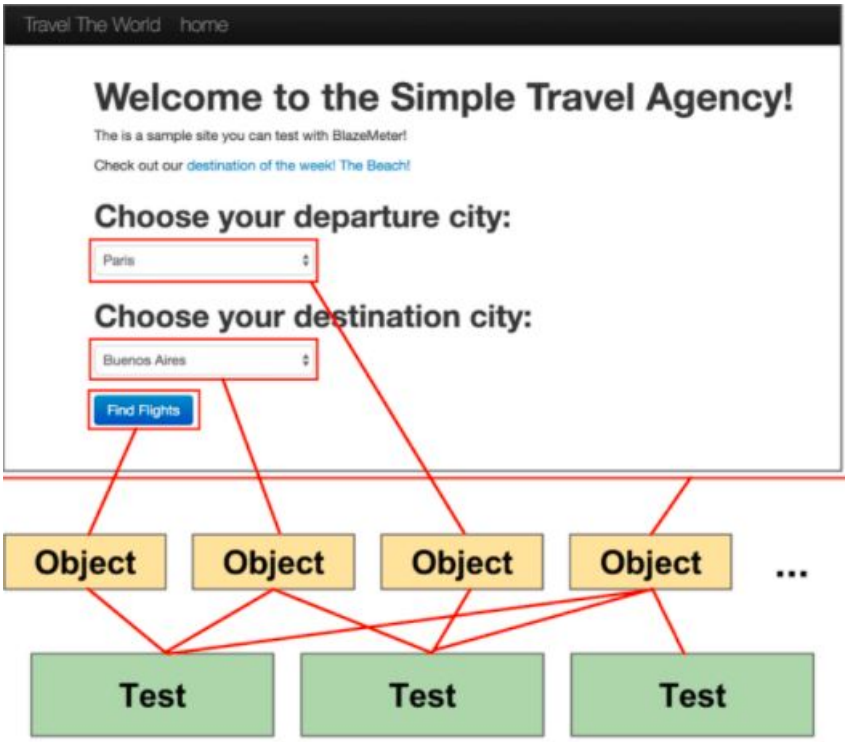
A design pattern is a reusable solution for a common problem in software design. We can say that each pattern is a particular example of a specific solution for a specific problem, regardless of the programming language or paradigm. Complementing design patterns, we have design principles. Design principles provide you with guidelines or rules that you need to follow for constructing well built and maintainable software. While patterns apply for specific issues, design principles apply regardless of context.

How is this relevant to UI Automation testing? As discussed before, UI testing is a hard and treacherous road full of different holes. But I have good news for you. You are not the first driver to travel this road. That's why there are almost no holes no one has ever passed through. You can think of design patterns and principles as powerful navigators. One tells you how to drive safely in general (design principle), while another gives clear instructions about how to handle each specific hole if it got in your way (design pattern).

In this section, I want to talk about two patterns: the Page Objects pattern, which has become the most popular web UI automation pattern among test automation engineers, and the Screenplay pattern, which organizes the Page Objects pattern and improves it.

The concept of the Page Objects pattern was introduced almost 10 years ago. Its purpose was to make UI automation tests consistent, to avoid code duplication, to improve readability and to organize code for web pages interaction. During web tests creation you always need to interact with web pages and web elements that are presented on these pages (buttons, input elements, images, etc.). The Page Objects pattern takes this requirement and applies object oriented programming principles on top of this, enforcing you to interact with all pages and elements as with objects.

For example, if you need to click on a button, you don't need to care about how to retrieve this button in the test, as it will already be handled in page objects. You should have the object of the page you are looking for and it should already contain the object of the button you are looking for inside it. All you need is to use a reference to this button object and apply the "click" action on it. You can think about all pages and web elements like this:



For each page and element you need to interact with, you should create a separate object that will be a reference to this web element in your test. This is an example of the test written without the Page Objects pattern:

```
WebDriver webDriver = hisDriver;

webDriver.navigate().to("blazemeter.com");

String heading =
    webDriver.findElement(By.cssSelector(HEADING_ELEMENT)).getText();
Assert.assertEquals(heading, "Welcome to the Simple Travel
    Agency!");
Assert.assertEquals(webDriver.getTitle(), "BlazeDemo");

Select flightsFromSelectElement = new
    Select(webDriver.findElement(By.cssSelector(FLIGHT_FROM_SELECT)));
Select flightsToSelectElement = new
    Select(webDriver.findElement(By.id(FLIGHT_TO_SELECT)));

flightsFromSelectElement.selectByValue("Boston");
flightsToSelectElement.selectByValue("New York");
webDriver.findElement(By.cssSelector(FIND_FLIGHTS)).click();
```

```
15
    Assert.assertTrue(webDriver.findElements(By.cssSelector(FLIGHTS_ROW_ELEMENTS)).si
    > 0);
```

Listing 10.3: webDriver.java

If we apply the Page Objects pattern and refactor the same test, we see an absolutely different picture:

```
1 homePage.open();
2 homePage.waitUntilPageLoaded();
3
4 Assert.assertEquals(homePage.getTitle(), "BlazeDemo");
5 Assert.assertEquals(homePage.getHeadingText(), "Welcome to the
    Simple Travel Agency!");
6
7 homePage.findFlights("Boston", "New York");
8 Assert.assertTrue(flightsPage.getFlights().size() > 0);
```

Listing 10.4: homePage1.java

You can find the working example of such a test in the test project under the "/src/test/java/ui/pageobject/FindFlightsSimplePageObjectExampleTest.java" location. Just go ahead and try it yourself.

To sum up, the Page Objects brings you these benefits: Makes your tests much clearer and easy to read by providing an interaction with pages and application page elements

Organises code structure

Helps to avoid duplications (we should never specify the same page locator twice)

Saves a lot of time on tests maintenance and makes the UI automation pipeline faster => reduces costs

If you want to you make this test even cleaner and more maintainable, you can introduce one more level of abstraction - steps or keywords. In different frameworks, you might see different names of these modules but the principles are the same. Steps (keywords) form modules of actions that you can reuse in any test. Once these steps (keywords) modules are written, all you need is to make a reference to the module in your test and you can use all the functionalities provided by these specific modules.

For example, if we create a module that aggregates all functionalities related to flights search on our website, then the test would look like this:

```
flightsSearchSteps.openHomePage();
flightsSearchSteps.verifyThatHomePageIsOpened();
flightsSearchSteps.verifyThatTitleAndHeadingTextIsCorrect();

flightsSearchSteps.searchFlightsBetween("Boston", "New York");
assertThat(flightsSearchSteps.foundFlights().size(),
    greaterThan(0));
```

Listing 10.5: openHomePage.java

We have a handy example for this case as well, which you can find by "/src/test/java/ui/pageobject/FindFlightsPageObjectWithStepsExampleTest.java" in the test project.

But even if you use the Page Objects pattern, you might sooner or later run into a contradiction that will get you stuck with a framework that is very heavy to support. Why? Because in addition to design patterns, you need to take care of design principles.

For example, it's a common case when Page Objects pattern becomes the reason of an anti-pattern called "Large Class". This happens when you have too many elements on some of the pages, so the number of functions on an object that represents the page might become really huge (this is called "Large Class" pattern). This goes against one of the most important design principles in object-oriented programming called SOLID:

Single Responsibility Principle

Open Closed Principle

Liskov Substitution Principle

Interface Substitution Principle

Dependency Inversion Principle

The main contradiction it goes against is the Single Responsibility Principle, which says that each class should have a single functionality. Robert C. Martin (one of the well-known software engineers and collaborators in SOLID) described this principle with these words: "A class should have only one reason to change.". That's why Page Objects might contradict

this principle, because page classes can contain hundreds of functions which do many different actions.

No worries, we are not going to cover the meaning of each principle in detail. You can go over many articles across the web to get an idea (e.g., you can start from wiki). But all you need to know for know that in order to follow SOLID principles with the Page Objects pattern, we should always care how we separate actions across pages and web elements and make additional code refactoring once in awhile to keep our framework maintainable.

Cue the Screenplay Pattern, which is designed to follow the SOLID principles from scratch. Briefly, Screenplay is the design pattern for acceptance tests (including UI tests) that allows you to easily follow SOLID principles. I recommend checking out this article, which will give you an outstanding explanation of the Screenplay pattern and it's usage in UI automation testing. But even without going into details, you can go over the same test we had before, but written with the usage of the Screenplay pattern, and feel the difference yourself:

```
1  givenThat(yuri).wasAbleTo(Start.withHomePage());
2
3  then(yuri).should(
4          seeThat(Application.information(),
5                  displays("title",equalTo("BlazeDemo")),
6                  displays("heading",equalTo("Welcome to the Simple
       Travel Agency!"))
7          )
8  );
9
10 when(yuri).attemptsTo(
11        SearchFlights.betweenCountries("Boston","New York")
12 );
13
14 then(yuri).should(
15         seeThat(
16                      TheFlights.displayed(),
17                      hasSize(greaterThan(0))
18         )
19 ;
```

Listing 10.6: withHomePage.java

You can find a fully working example in our test project, which has been implemented following these practices. It is located at "/src/test/-java/ui/screenplay/FindFlightsScreenPlayExampleTest.java".

The behavior of web applications depends on many factors like network speed, your machine capabilities or the current load on application servers. Because of all these factors, you cannot always predict how much time it will take to load a specific page or web element. That's why sometimes you might want to add a timeout and pause script check execution for at least a certain amount of time.

If you do not know how to deal with this in the right way, you will never see stability in your UI automation.

Let's assume that in our test we are going to open the home page and verify the heading of the main page. Extremely simple. You just need to implement two functions. One that opens the page and the second that verifies if the heading element is presented and has the right value. But what if we knew that our application can take up to 7-8 seconds to start?

Selenium, a web browser automation tool, works pretty fast and honestly saying, even faster than you. It can open the page in milliseconds and will try to get the text of heading element while the application itself is still starting. In this situation.... please, NEVER write a code like this:

```
yuri.openHomePage();
Thread.sleep(10000);
yuri.verifyThatTitleAndHeadingTextIsCorrect();
```

This is the strongest killer of UI automation tests stability. Why?

We lose time because you know that in 95 percentage of cases, the application should be up and running in 7-8 seconds. Therefore, each time we lose about 2-3 seconds of execution time.

What happens if application load takes 10.5 seconds one day? Just because of some network slowness. We will get failed tests. But when you try to run it locally on the next day, it works perfectly fine. This is one more example of the troubles you might get into by using this way of waiting in your tests.

This is how you can specify the implicit wait in Java by using Selenium:

```
1 WebDriver driver = new FirefoxDriver();
2 driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
```

Listing 10.7: FirefoxDriver.java

What is the explicit wait then? The explicit wait is designed for needs when a certain web element or action takes a much longer time to load than the rest. What if your application works in a way that it takes a long time to start (7-8 seconds) but after that it works very fast? There is no sense to specify the implicit wait as 10 or even 15 seconds only because you have a slow application loading. For this, you can use the explicit wait, which waits for a certain condition for a specified amount of time.

Here is how we can rewrite our previous example by using the idea of the explicit wait:

```
yuri.openHomePage();
withTimeoutOf(15, TimeUnit.SECONDS).waitFor(MAIN_HEADING_LABEL);
yuri.verifyThatTitleAndHadingTextIsCorrect();
```

In this case, we also do not waste any time and the script execution will continue immediately after the expected element will be found.

Looks clear, right? Not so clear as you might think... the official Selenium website displays this very important note:

"Do not mix implicit and explicit waits. Doing so can cause unpredictable wait times. For example setting an implicit wait of 10 seconds and an explicit wait of 15 seconds, could cause a timeout to occur after 20 seconds."

You can use this article as well as this one to learn why it is better and enough to use ONLY explicit waits.

We have seen many automation frameworks that require huge efforts to run them on another machine, unlike the one which was used for framework creation. This is a very bad practice because it doesn't allow new engineers or other team members to run tests, without struggling with setup troubles. And what if you need to run your tests on the CI server? It would be a very tricky task if your UI test automation framework is not portable. That's why we have a few tips that can help you avoid such issues.

The same principle goes for the web driver. For some reason, from project to project, I still see a lot of cases when web drivers are required to be installed in order to run tests. Moreover, sometimes you need a specific web driver version and if documentation is not so great you can

spend many hours trying to see the first green tests. How should this be dealt with?

There is an excellent helper tool called WebDriverManger. It takes care of the whole driver download and configuration workflows. All you need to do is to configure one more additional java dependency in your framework, and all the web drivers will be downloaded and configured automatically. It's a miracle that doesn't require human interaction once you've decided to run your tests on another machine that doesn't have any web drivers installed.

The installation of the WebDriverManager is very simple and is covered in a few easy steps on this page. However, I found it not so straightforward when I did it for the first time for the Serenity framework.

Serenity has its own workflow of web driver configuration. I couldn't find a proper solution across the web so this section might be useful if you also decided to use the Serenity framework. The main idea of the solution is that Serenity has the mechanism of a custom web driver. All you need is to create a separate class that extends the DriverSource class of the Serenity framework and create the required driver by using the WebDriverManager:

11

END-TO-END TESTING

End To End Testing is a software testing method that validates entire software from starting to the end along with its integration with external interfaces. The purpose of end-to-end testing is testing whole software for dependencies, data integrity and communication with other systems, interfaces and databases to exercise complete production like scenario.

Along with the software system, it also validates batch/data processing from other upstream/downstream systems. Hence, the name "End-to-End". End to End Testing is usually executed after functional and System Testing. It uses actual production like data and test environment to simulate real-time settings. End-to-End testing is also called Chain Testing.

11.1 WHY END TO END TESTING

End To End Testing verifies complete system flow and increases confidence by detecting issues and increasing Test Coverage of subsystems. Modern software systems are complex and interconnected with multiple subsystems that may differ from current systems. The whole system can collapse by failure of any subsystem that is major risk which can be avoided by End-to-End testing.

11.1.1 *End to End Testing Process:*

The following diagram gives an overview of the End to End testing process.

The chief activities involved in End to End Testing are -

Study of an end to end testing requirements

Test Environment setup and hardware/software requirements

Describe all the systems and its subsystems processes.

Description of roles and responsibilities for all the systems

Testing methodology and standards

End to end requirements tracking and designing of test cases

Input and output data for each system

11.1.2 *How to create End-to-End Test Cases?*

End to End Testing Design framework consists of three parts

Build user functions
Build Conditions
Build Test Cases
Let's look at them in detail: -
Build User Functions
Following activities should be done as a part of build user functions:

List down the features of the system and their interconnected components
List the input data, action and the output data for each feature or function
Identify the relationships between the functions
Determine whether the function can be reusable or independent

For example -Consider a scenario where you login into your bank account and transfer some money to another account from some other bank (3rdparty sub-system)

Login into the banking system
Check for the balance amount in the account
Transfer some amount from your account to some other bank account (3rdparty sub-system)
Check your latest account balance
Logout of the application

Build Conditions based on User Function Following activities are performed as a part of build conditions:

Building a set of conditions for each user function defined Conditions include sequence, timing and data conditions For example -Checking of more conditions like

Login Page

Invalid User Name and Password
Checking with valid username and password
Password strength checking
Checking of error messages
Balance Amount

Check the current balance after 24 hours. (If the transfer is sent to a different bank) Check for the error message if the transfer amount is greater than the current balance amount

Build a Test Scenario
Building the Test Scenario for the user function defined

In this case,
Login into the system
Check of bank balance amount
Transfer the bank balance amount
Build Multiple Test cases

Build one or more test cases for each scenario defined. Test cases may include each condition as a single test case.

Metrics for End to End testing:
Following are few of many metrics used for End to End Testing.
Test Case preparation status: It gives Test Case preparation progress against planned
Weekly Test Progress- Provides week-wise details of percentage test completion- Failed, not executed and executed against planned for execution tests.
Defects Status and Details- It gives Percentage of open and closed defects by the week. Also, week-wise defects distribution based on severity and priority
Environment Availability -Total number of hours "up" / Total number of hours scheduled per day for testing

12

PART 4

12.1 CODE QUALITY-SONARQUBE -:

Code quality is an extremely important parameter for software quality deliverables and affects the overall success of a software organization. It is critical to ensure utmost quality of delivered software as code quality defines how safe, secure and reliable the software is.

Ensuring high quality of software should definitely be the goal throughout any development process. It can surely be achieved if everyone in the team is determined to deliver quality and holds himself accountable for it.

Code quality depends on a number of different attributes and requirements, primarily determined and prioritized by the business needs.

12.1.1 *Key Aspects to Measure*

The typical structure of a test consists of the following 3 main phases.

The key traits to measure for higher quality are:

1. **Reliability** → Reliability measures the system's probability to run without failure over a specific period of operation. It relates to the number of defects and availability of the software.

2. **Maintainability** → Maintainability measures how easy is the system maintenance. It relates to the size, consistency, structure, and complexity of the codebase. Ensuring that source code is maintainable depends on a number of factors, such as testability and understandability.

3. **Testability** → Testability measures how well the software can be tested. It relies on the extent to which you can control, observe, isolate, and automate testing.

4. **Portability** → Portability measures how usable or portable the same software is across different environments. Platform independence is the key factor.

5. **Reusability** → Reusability measures whether existing code can be used again or not. Modular and loosely coupled code, is easy to reuse.

12.1.2 *How to Improve Code Quality*

Once, the code quality is measured, we can take steps to improve it.

Let us discuss some ways you can improve code quality:

Coding Standards

using a static code analyzer helps.

1. **Code Analysis before Code Reviews** → Quality should be a priority from the very start of developmenand that's why it's important to analyze code before code reviews begin. Infact, its best to analyze code as soon as it's written. The earlier the defects are detected, the easier and faster it is to resolve them.

2. **Code Review Best Practices** →Good code reviews help improve overall software quality. We will discuss Code reviews in detail shortly.

3. **Refactor Legacy Code** → One way to improve the quality of an existing codebase is through refactoring.

Code Review is a phase of the software development process in which the authors of the code use a system of review to examine the source code. The code is usually reviewed by another programmer who inspects the code for mistakes, irregular formatting, or inconsistencies with the system requirements that may create larger issues during the integration process.

Code review is the most commonly used procedure for validating the design and implementation of features.

Code Review is an instrumental component of an efficient software solution. It creates consistent code and enables better QA testing and seamless software integration. By diminishing mistakes and fixing errors earlier at the review stage, programmers become more efficient in designing customized solutions in line with the specifications to the client.

Code Reviews primarily check for four things:

1. Code issues

2. Consistency in formatting with the overall software design

3. Quality of documentation

4. Consistency with coding standards and project requirements

The major purpose of the review process is to create a collective responsibility of the software's quality through knowledge transfer by examining the source code collectively as a team.

Another advantage of maintaining consistent code standards is that it is easier for specialists and testers to understand the source code.

Source code that is constantly under review allows developers to learn more reliable techniques and best practices leading to robust software for seamless integration and functionality.

Code review can be performed in two levels:

Peer review

It is mainly focused on functionality, design, and the implementation and usefulness of proposed fixes for stated problems.

Peer reviewers look for the following in the code:

Feature Completion
Potential Side Effects
Readability and Maintenance
Consistency
Performance
Exception Handling
Simplicity
Reuse of Existing Code
Test Cases

External review

It is different than peer reviews as it addresses different kinds of issues. Specifically, external reviews focuses on how to enhance code quality, promote best practices, and remove "code smells."

External reviewers look for following in the code:

Readability and Maintenance
Coding Style
Code Smells

12.2 SONARQUBE

SonarQube is a web-based open source platform by SonarSource, used to measure and analyse the source code quality. Code quality analysis makes your code more reliable and more readable.

It is implemented in Java language and can analyze the code of about 20 different programming languages, including c/c++, PL/SQL, Cobol etc through plugins. More than 50 plugins are available which extend the functionality of SonarQube.

SonarQube can inspect and evaluate anything that affects code base, from minor styling details to critical design errors. Hence, it is a great aid to software application developers in identifying the issues and their impact.

SonarQube is in no way competing with any of the above static analysis tools, but rather it complements and works very well with these tools. In fact, it ceases to work if these static analysis tools (Checkstyle, PMD, and FindBugs) do not exist.

SonarQube's offerings span what its creators call the Seven Axes of Quality:

1. Architecture and Design
2. Unit tests
3. Duplicated code
4. Potential bugs
5. Complex code
6. Coding standards
7. Comments
8. We will explore this tool in detail in the further sections.

We will explore this tool in detail in the further sections.

12.3 KEY CONCEPTS IN STATIC CODE ANALYSIS

A code review typically starts with the static analysis of the source code using a static analysis tool. Various key statistics are analyzed on the basis of software metrics.

1. **Code Smell:** → It describes the estimated effort to fix to fix all the maintainability Issues / code smells. It is a programming concept that reflects the extra development work, which has to be done if long-term and best solutions are not sought, and things are implemented keeping the ease in mind, for short term.

2. **Technical Debt:** → It describes the estimated effort to fix to fix all the maintainability Issues / code smells. It is a programming concept that reflects the extra development work, which has to be done if long-term and best solutions are not sought, and things are implemented keeping the ease in mind, for short term.

3. **Code coverage:** → It is a measure which describes the degree of which the source code of the program has been tested. It is one form of white box testing which finds the areas of the program not exercised by a set of test cases.

4. **Code Bugs:** → A problem or an issue that shows that something is wrong in the code. It might not break the code right away, but surely will, and may be at the moment when its impact is huge. It has to be fixed.

5. **Vulnerabilities:** → It is a cyber-security term that refers to a flaw or loophole in the system that makes it vulnerable to attack. A vulnerability may also refer to any type of weakness in a set of procedures, or in anything that leaves information security exposed to a threat. Minimizing vulnerabilities leaves bare minimum chances of unwanted access to sensitive and secure data by malicious users.

SonarQube Integration is an open source static code analysis tool that has become quite popular in recent times. It can integrate with your existing workflow to enable continuous code inspection across your project branches and pull requests.

SonarQube provides reports for your projects after thorough analysis.

Everything right from minor styling details to critical design errors, is inspected and evaluated by SonarQube, thereby enabling developers to access and track code analysis data continuously, ranging from styling errors, potential bugs, and code defects to design inefficiencies, code duplication, lack of test coverage, and excess complexity.

The Sonar platform analyzes source code from different aspects and drills down to your code layer by layer, moving from the module level down to the class level, giving metric values and statistics, revealing issues in the source code at each level, that need to be addressed.

Your project home page tells you where you stand in terms of quality in a glimpse of an eye. It you an immediate sense of what you have achieved till now and the project status.

Features:

12.3.1 *Write Clean Code*

1. **Overall Health:** →Discovered issues can either be Unreachable source code, a Bug, Vulnerability, Code Smell, Coverage or Duplication. Each category has a corresponding number of issues. Dashboard page shows where you stand in terms of quality in a glimpse of an eye.

2. **Enforce Quality gate:** → To fully enforce a code quality practice across all teams, you need to set up a Quality Gate which is a set of standard conditions the project must meet before it can be released to production.

3. **Issue Resolution:** → There are five different severity levels of Issues like blocker, critical, major, minor and info. Also, the issues tab has different filter criteria like category, severity level, tag(s), and the calculated time required for the issue rectification.

Handle Multi-Language Projects

SonarQube automatically detects the project language and run corresponding code analyzer for each language.

Centralize Quality – All projects in one place

SonarQube enables organization to estimate and predict project risks as there is a centralized system of storing the code metrics. It simplifies deployment and helps monitor the project status.

Shared rulesets

SonarQube provides the facility to create your own quality profiles and define Sonar Rules shareable across different projects.

Quality Gates

Quality Gates are a powerful feature by SonarQube.

They are a combination of threshold measures set on your project.

A quality gate is the best way to enforce a quality policy in your company.

They are basically a set of conditions that the project must meet before it can be delivered to production.

You can define as many quality gates as you wish to, depending on the project need.

Each Quality Gate condition is typically a combination of:

measure

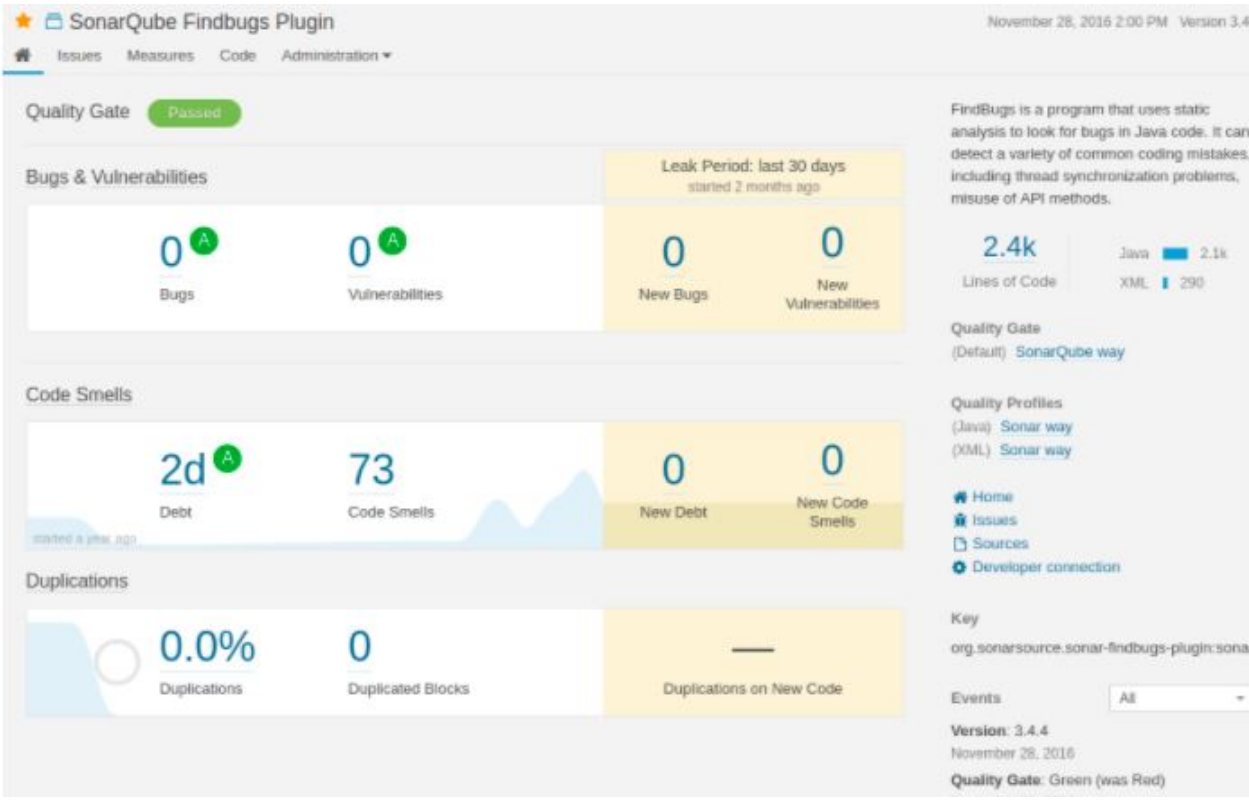
1. period: Value (to date) or New Code (differential value over the New Code period)
2. comparison operator
3. warning value (optional)
4. error value (optional)

For instance, a condition might be:

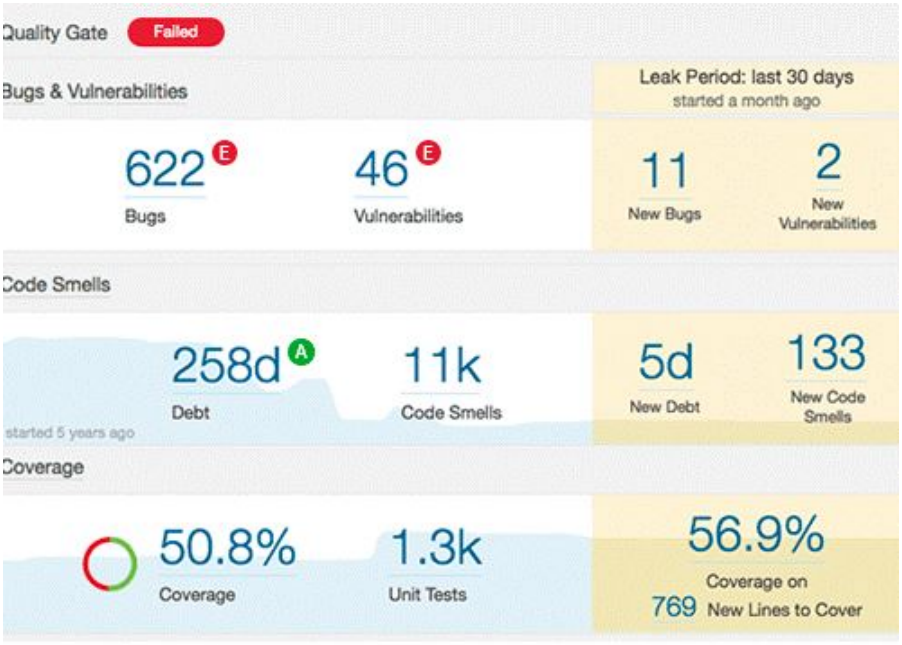
1. measure: Blocker issue
2. period: Value
3.comparison operator: >
4. error value: 0

All projects may not be verified against the same quality gate, as it's not practical that the threshold measures would be the same for all projects.

To manage Quality Gates, go to Quality Gates (top menu bar) in sonarqube. Quality Gates are defined and managed in the Quality Gates page found in the top menu. If your project passes the Quality Gate, this is how it will be displayed.



A failed Quality Gate would look like the one below, highlighting all the bugs and issues found in the project.


Quality Gate
Failed
Bugs & Vulnerabilities
Leak Period: last 30 days
started a month ago
622
E
Bugs
46
E
Vulnerabilities
11
New Bugs
2
New Vulnerabilities
Code Smells
258d
A
Debt
11k
Code Smells
started 5 years ago
5d
New Debt
133
New Code Smells
Coverage
50.8%
Coverage
1.3k
Unit Tests
56.9%
Coverage on
769
New Lines to Cover

13

JAVA PROJECT USING TEST DRIVEN DEVLOPMENT

The following report describes the various stages that make up the development of a small Java project and it has Login System using web application, which manages login system for a user. The application was developed through the use of TDD techniques (a development model of software that provides that the drafting of automatic tests takes place before that of software that must be tested), Here we focus on Build Automation. In general, techniques and the technologies illustrated in Test-Driven Development, Build Automation, Continuous Integration.

The main purpose of this project is to store login data of a user in database and retrieve those data. The project version is 1.0 is stored in In GitHub repository and able to access publicly. The language uses here is Java and Spring boot technology. Additionally, Docker tool used to design to make it easier to create, deploy, and run applications by using containers. Docker is integrated with travis-ci. As a database I use the document-based MongoDB for this application. This project can be useful for any company to login to their service. This project successfully tested and able to use in different platforms.

.

Project source code: https://github.com/JosephThachilGeorge/TDD

14

CONCLUSION

The main purpose of this book is to understand the concept of test-driven development using JAVA. The main concept JUNIT5 is discussed in this book. When implementing the test-driven development, the primary concern is to ensure confidentiality, integrity and secure products.