

# **Test-Driven Development, Build Automation, Continuous Integration**

with Java, Eclipse and friends

Lorenzo Bettini

# **Test-Driven Development, Build Automation, Continuous Integration**

with Java, Eclipse and friends

Lorenzo Bettini

This book is for sale at <http://leanpub.com/tdd-buildautomation-ci>

This version was published on 2023-02-23



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2018 - 2023 Lorenzo Bettini

*to Silvia & Pietro*

# Contents

<b>About the author</b> . . . . .	<b>i</b>
<b>Introduction</b> . . . . .	<b>ii</b>
Conventions . . . . .	iv
Code of examples . . . . .	iv
Errata . . . . .	v
Install the JDK . . . . .	v
<b>Book structure</b> . . . . .	<b>vii</b>
<b>1. Testing</b> . . . . .	<b>1</b>
1.1 Automated tests . . . . .	1
1.2 Advantages of automated tests . . . . .	4
1.3 Test-Driven Development (TDD) . . . . .	5
1.3.1 Behavioral-Driven Development (BDD) . . . . .	6
<b>2. Eclipse</b> . . . . .	<b>8</b>
2.1 Eclipse workspace . . . . .	8
2.2 Perspectives and views . . . . .	9
2.3 Projects . . . . .	9
2.3.1 Importing an existing project . . . . .	10
2.3.2 Creating a new Java project . . . . .	10
2.3.3 Open projects and closed projects . . . . .	12
2.3.4 Automatic building . . . . .	12
2.4 IDE tools . . . . .	14
2.4.1 Content Assist . . . . .	14
2.4.2 Quick Fix . . . . .	16
2.4.3 Quick Assist . . . . .	17
2.4.4 Editable proposals . . . . .	17
2.4.5 The Source menu . . . . .	18
2.5 Eclipse Preferences . . . . .	18
2.6 Find Actions . . . . .	19
2.7 Eclipse Shortcuts . . . . .	20
2.7.1 Quick Search . . . . .	20
2.8 Eclipse Run configurations . . . . .	20

## CONTENTS

2.9	Browsing code . . . . .	21
2.9.1	Outline . . . . .	21
2.9.2	Navigate . . . . .	22
2.9.3	Local History . . . . .	23
2.10	Code Mining . . . . .	23
3.	<b>JUnit</b> . . . . .	26
3.1	Structure of a test . . . . .	26
3.2	A first example . . . . .	29
3.2.1	Exception testing . . . . .	35
3.2.2	Other tests . . . . .	39
3.2.3	Revising . . . . .	42
3.3	External collaborators (dependencies) . . . . .	44
3.3.1	Alternative implementations . . . . .	51
3.4	Testing private methods? . . . . .	52
3.5	Keep your tests clean and readable . . . . .	53
3.5.1	Beware of code duplication removal in tests . . . . .	55
3.6	Other testing libraries . . . . .	56
3.6.1	Using Hamcrest matchers . . . . .	56
3.6.2	Using AssertJ . . . . .	57
3.7	Keeping test code separate from main code . . . . .	61
3.7.1	Export a runnable JAR . . . . .	67
3.8	JUnit 5 . . . . .	67
4.	<b>TDD</b> . . . . .	75
4.1	Introduction to TDD . . . . .	75
4.1.1	The Red-Green-Refactor cycle . . . . .	75
4.1.2	The three laws of TDD . . . . .	77
4.1.3	Remove duplication . . . . .	78
4.2	The Transformation Priority Premise . . . . .	79
4.2.1	A first example of TDD . . . . .	80
4.3	Three strategies for the green state . . . . .	91
4.3.1	The factorial example with TDD . . . . .	91
4.4	Small or big steps? . . . . .	96
4.5	Tests and induction (part 1) . . . . .	98
4.6	TDD with JUnit 4 Parameterized Tests . . . . .	99
4.7	TDD with JUnit 5 Parameterized Tests . . . . .	103
4.8	Testing abstract classes? . . . . .	106
4.9	Writing a test that succeeds . . . . .	107
5.	<b>Code coverage</b> . . . . .	108
5.1	JaCoCo and EclEmma . . . . .	108
5.2	How code coverage can help . . . . .	111
5.3	Code coverage percentage . . . . .	114

## CONTENTS

<b>6. Mutation Testing . . . . .</b>	<b>116</b>
6.1 The first example with PIT . . . . .	117
6.2 Enabling additional mutators . . . . .	122
6.3 Further experiments with PIT . . . . .	125
6.3.1 Tests and induction (part 2) . . . . .	125
6.3.2 Mutations and iterative solutions . . . . .	128
6.3.3 Mutation testing on the MyStringUtils example . . . . .	130
6.3.4 Mutation testing on the Bank example . . . . .	130
6.4 Narrowing mutations . . . . .	133
<b>7. Maven . . . . .</b>	<b>135</b>
7.1 Introduction to Maven . . . . .	136
7.2 Maven installation . . . . .	137
7.3 Let's get started with a Maven archetype . . . . .	138
7.3.1 Create a Maven project from the command line . . . . .	138
7.3.2 Import a Maven project in Eclipse . . . . .	139
7.3.3 Create a Maven project from Eclipse using an archetype . . . . .	140
7.4 Structure of a Maven project . . . . .	141
7.5 Java settings . . . . .	142
7.6 Dependencies . . . . .	142
7.6.1 Maven coordinates (GAV) . . . . .	143
7.6.2 Dependencies in the POM . . . . .	143
7.6.3 SNAPSHOT . . . . .	144
7.6.4 Version ranges . . . . .	145
7.7 Eclipse m2e . . . . .	145
7.8 Properties . . . . .	146
7.9 Resources . . . . .	148
7.9.1 Layout of an Eclipse project . . . . .	149
7.9.2 Create a Maven project from Eclipse without an archetype . . . . .	150
7.10 The Eclipse m2e pom.xml multi-tab editor . . . . .	150
7.11 The bank example as a Maven project . . . . .	151
7.12 Build with Maven . . . . .	153
7.12.1 Maven lifecycles . . . . .	154
7.12.2 Run Maven from the command line . . . . .	157
7.12.3 Run Maven goals . . . . .	160
7.12.4 Run Maven from Eclipse . . . . .	162
7.12.5 Offline mode . . . . .	162
7.12.6 Updating dependencies . . . . .	163
7.13 Maven packaging . . . . .	163
7.13.1 Packaging “jar” . . . . .	163
7.13.2 Packaging “pom” . . . . .	164
7.14 Parent and modules . . . . .	164
7.14.1 Let's create a parent project and a module project . . . . .	166

## CONTENTS

7.14.2	Effective POM . . . . .	170
7.14.3	Project aggregation and project inheritance . . . . .	171
7.14.4	Build a single module . . . . .	176
7.14.5	Dependency management . . . . .	176
7.14.6	Bill of Material (BOM) . . . . .	177
7.14.7	How Eclipse resolves the projects in the workspace . . . . .	179
7.15	Configuring a Maven plugin . . . . .	180
7.15.1	Configuring the Maven compiler plugin . . . . .	183
7.15.2	Generate source and javadoc jars . . . . .	184
7.15.3	Configuring the generated jar . . . . .	186
7.15.4	Creating a FatJar . . . . .	188
7.15.5	Plugin management . . . . .	192
7.15.6	Configuring the PIT Maven plugin . . . . .	194
7.15.7	Configuring the JaCoCo Maven plugin . . . . .	196
7.16	Maven profiles . . . . .	202
7.16.1	Don't abuse profiles . . . . .	205
7.17	The bank example as a multi-module project . . . . .	208
7.17.1	The bank BOM . . . . .	208
7.17.2	The bank parent POM . . . . .	210
7.17.3	The bankaccount project . . . . .	211
7.17.4	The bank project . . . . .	212
7.17.5	The bank report project . . . . .	213
7.17.6	The bank app project . . . . .	214
7.17.7	The aggregator project . . . . .	216
7.18	JUnit 5 and Maven . . . . .	217
7.19	Maven deploy . . . . .	219
7.20	Maven wrapper . . . . .	219
7.21	Beyond Java 8 . . . . .	220
7.22	Reproducibility in the build . . . . .	223
8.	<b>Mocking . . . . .</b>	<b>224</b>
8.1	Testing state and testing interactions . . . . .	227
8.2	Mockito: an overview . . . . .	229
8.2.1	Creating a mock . . . . .	229
8.2.2	Method stubbing . . . . .	229
8.2.3	Interaction verification . . . . .	230
8.3	Mockito: a tutorial . . . . .	230
8.3.1	Spy . . . . .	242
8.3.2	Spying and stubbing . . . . .	244
8.3.3	Stubbing and exceptions . . . . .	246
8.3.4	Subsequent stubbing . . . . .	248
8.3.5	Argument matchers . . . . .	250
8.4	Alternative ways of initializing mocks and other elements . . . . .	250

## CONTENTS

8.5	Mockito BDD style . . . . .	255
8.6	Mockito and JUnit 5 . . . . .	257
8.7	Stubbing with answers . . . . .	258
8.7.1	Another example: Transactions . . . . .	261
8.8	Fakes . . . . .	266
8.9	What to mock . . . . .	268
<b>9.</b>	<b>Git . . . . .</b>	<b>270</b>
9.1	Let's start experimenting with Git . . . . .	271
9.1.1	Create a git repository . . . . .	271
9.1.2	Git configuration . . . . .	272
9.1.3	States of the working tree . . . . .	272
9.1.4	Perform a commit . . . . .	273
9.1.5	Commit history . . . . .	276
9.1.6	Ignore files and directories . . . . .	276
9.1.7	Branches . . . . .	277
9.1.8	Reset . . . . .	280
9.1.9	Cherry-pick . . . . .	280
9.1.10	Stash . . . . .	281
9.2	Remote repositories . . . . .	281
9.2.1	Bare repository . . . . .	281
9.2.2	Adding a remote repository . . . . .	281
9.2.3	Initialize an empty remote repository . . . . .	282
9.2.4	Cloning a remote repository . . . . .	283
9.2.5	Push and fetch . . . . .	284
9.2.6	Rebasing . . . . .	287
9.2.7	Pull requests . . . . .	288
9.3	EGit (Git in Eclipse) . . . . .	289
9.3.1	Add an Eclipse project to a new Git repository . . . . .	290
9.3.2	The Git Repositories view . . . . .	293
9.3.3	Importing an existing project . . . . .	294
9.3.4	Branch operations . . . . .	294
9.3.5	Cloning from Eclipse . . . . .	296
9.4	GitHub . . . . .	297
9.4.1	Create a repository on GitHub . . . . .	297
9.4.2	GitHub pull requests . . . . .	299
9.4.3	Contributing to other projects . . . . .	304
<b>10.</b>	<b>Continuous Integration . . . . .</b>	<b>309</b>
10.1	Our running example . . . . .	310
10.2	GitHub Actions . . . . .	312
10.2.1	Caching dependencies . . . . .	318
10.3	Build matrix . . . . .	320

## CONTENTS

10.3.1	Building using different JDKs . . . . .	321
10.3.2	Building using different operating systems . . . . .	323
10.3.3	Configuring the build matrix . . . . .	324
10.4	Building pull requests . . . . .	326
10.4.1	Multiple workflows . . . . .	326
10.5	Storing artifacts . . . . .	329
10.6	Code coverage in CI . . . . .	332
10.7	Code coverage with Coveralls . . . . .	335
10.7.1	Using Coveralls from our computer . . . . .	335
10.7.2	Using Coveralls from GitHub Actions . . . . .	339
10.7.3	Coverage threshold . . . . .	340
10.8	Badges . . . . .	344
10.9	Reproducibility in CI . . . . .	345
10.10	Let's revise our build process . . . . .	347
10.11	Further steps . . . . .	351
<b>11.</b>	<b>Docker . . . . .</b>	<b>353</b>
11.1	Let's get started with Docker . . . . .	355
11.2	Run a server in a container . . . . .	359
11.3	Dockerize a Java application . . . . .	361
11.3.1	The Java application to dockerize . . . . .	361
11.3.2	The Dockerfile . . . . .	362
11.4	Windows containers . . . . .	366
11.5	Build the Docker image from Maven . . . . .	368
11.6	Using Docker in GitHub Actions . . . . .	372
11.7	Docker networks . . . . .	376
11.8	Docker compose . . . . .	385
11.8.1	Control startup order of container . . . . .	388
11.9	Docker in Eclipse . . . . .	392
11.10	Push to DockerHub . . . . .	394
11.11	Reproducibility in Docker . . . . .	394
<b>12.</b>	<b>Integration tests . . . . .</b>	<b>395</b>
12.1	Our running example . . . . .	396
12.2	Unit tests with databases . . . . .	401
12.2.1	Mock the MongoClient? . . . . .	403
12.2.2	Use an in-memory database . . . . .	404
12.3	Integration tests . . . . .	411
12.3.1	Source folder for integration tests . . . . .	411
12.3.2	Integration tests with Docker and Testcontainers . . . . .	412
12.3.3	Running integration tests with Maven . . . . .	416
12.3.4	Integration tests with Docker and Maven . . . . .	418
12.3.5	Running integration tests in GitHub Actions . . . . .	424

## CONTENTS

12.4	Unit or integration tests?	425
<b>13.</b>	<b>UI tests</b>	<b>426</b>
13.1	The running example	426
13.2	UI unit tests	428
13.2.1	Testing the GUI controls	431
13.2.2	Implementing the StudentView interface	439
13.2.3	Unit tests for the UI frame's logic	441
13.3	Running UI tests in GitHub Actions	444
13.4	UI integration tests	445
13.5	Multithreading	451
13.5.1	Race conditions in the application	457
13.5.2	Race conditions in the database	460
13.6	Improvements	463
<b>14.</b>	<b>End-to-end tests</b>	<b>465</b>
14.1	The running example	465
14.2	E2e tests for our application	470
14.3	BDD with Cucumber	475
14.3.1	Data tables	487
14.3.2	Step decoupling	492
14.3.3	High-level specifications	493
14.4	Starting from the high-level specifications	497
14.5	Change some low-level details	501
<b>15.</b>	<b>Code Quality</b>	<b>504</b>
15.1	Using SonarQube locally	505
15.2	Analyze a project	507
15.2.1	False positives and rule exclusion	512
15.2.2	Analysis of test code	514
15.2.3	Security hotspots	515
15.2.4	Code coverage in SonarQube	515
15.2.5	JUnit reports	517
15.3	SonarCloud	518
15.4	SonarLint (IDE integration)	522
<b>16.</b>	<b>Learning tests</b>	<b>524</b>
16.1	Dependency Injection with Google Guice	524
16.1.1	Guice main concepts	526
16.1.2	Singleton	530
16.1.3	Field and method injection	531
16.1.4	Providers	533
16.1.5	Binding annotations	535
16.1.6	Overriding bindings	537

## CONTENTS

16.1.7	Factories and AssistedInject . . . . .	538
16.1.8	Cyclic dependencies . . . . .	540
16.1.9	Provides . . . . .	542
16.2	Apply our learnings . . . . .	542
	<b>Bibliography . . . . .</b>	<b>549</b>

# About the author

**Lorenzo Bettini** is an Associate Professor in Computer Science at the Dipartimento di Statistica, Informatica, Applicazioni “Giuseppe Parenti”, Università di Firenze, Italy. Previously, he was a researcher in Computer Science at Dipartimento di Informatica, Università di Torino, Italy. He has a Masters Degree summa cum laude in Computer Science (Università di Firenze) and a PhD in “Logics and Theoretical Computer Science” (Università di Siena).

His research interests cover design, theory, and the implementation of statically typed programming languages and Domain Specific Languages.

He is the project lead of the Eclipse projects EMF Parsley, <https://www.eclipse.org/emf-parsley/> and Xsemantics, <https://projects.eclipse.org/projects/modeling.xsemantics>. He is also a committer of the Eclipse projects Xtext, <https://www.eclipse.org/Xtext/>, and SWTBot, <https://www.eclipse.org/swtbot/>. He is the author of the two editions of the book “Implementing Domain-Specific Languages with Xtext and Xtend”, published by Packt Publishing (2013 and 2016).

He is also the author of about 100 research papers published in international conferences and international journals.

Home page <https://www.lorenzobettini.it>.

Twitter @lorenzo\_bettini

# Introduction

The main subject of this book is software testing. The central premise is that testing is a crucial part of software development, and you want to ensure that the software you write behaves correctly.

You can manually test your software. For a command-line application, this means running the entire application and interacting with it to see whether it provides the expected outputs, given some inputs. Inputs are fed manually by you, and the output's correctness is once again manually checked by you. For an application with a GUI (a window in a desktop application or the web interface in a web application), you manually interact with the GUI menus, buttons, text fields, etc., and see whether the GUI changes according to your expectations. Manual tests require lots of manual work, take a massive amount of time (even for small applications), and do not quickly scale when the complexity of the application increases. Manual testing is costly. When you change something in the application code, this manual testing process has to be executed once again. Finally, manually verifying the behavior of the software is error-prone.

On the contrary, this book focuses on automated tests. The developer must code these tests, but they run automatically: interactions (input and output) are carried out automatically by the tests, and verifications are automatic. Automated tests can be done at several levels: they can test a single Java class or the entire GUI of an application. The time you spend writing these automatic tests is an investment: when something changes in the application code, running automatic tests does not require further effort, it is much faster than verifying things manually, and it is not error-prone, as long as tests are written correctly. Thus, it is crucial to write the automatic tests correctly. This book will deal with techniques and tools for keeping the quality of your automated tests high.

In this book, we will see a few types of tests, from testing a single component in isolation to testing the entire application. We will also deal with tests in the presence of a database and with tests that verify the correct behavior of the graphical user interface. In particular, we will describe and apply the **Test-Driven Development** methodology, writing tests before the actual code.

Throughout this book, we will use **Java** as the primary programming language. We use **Eclipse** as the IDE. Using an IDE (Integrated Development Environment) is crucial for productivity during all the development phases. Java and Eclipse have a large ecosystem of “friends” like frameworks, tools, and plugins. Many of them are related to automated tests and perfectly fit the book’s goals. We will use **JUnit** throughout the book as the leading Java testing framework. We will also use additional testing frameworks and libraries that integrate seamlessly with JUnit.

In this book, we primarily use JUnit 4. Thus, if we only mention “JUnit,” we refer to JUnit 4. However, we also briefly introduce JUnit 5, and in some chapters of the book, we also sometimes show how the new features of JUnit 5 can be helpful for the subject of those chapters.

Of course, the presented techniques, methodologies, and tools could be reused for other IDEs and programming languages.

While using an IDE during development is crucial for productivity, it is also essential to completely automate the build process. Indeed, another relevant subject of the book is **Build Automation**. We will use one of the mainstream tools for build automation in the Java world: **Maven**.

We will use **Git** as the Version Control System. Git will allow us to keep our code's whole history and share our code on the Internet, for example, to collaborate with other developers. We will use **GitHub** as the hosting service for our Git repositories.

We will then connect our code hosted on **GitHub** with its cloud platform for **Continuous Integration (CI)**: **GitHub Actions**. With the Continuous Integration process, we will implement a workflow where each time we commit a change in our Git repository, the CI server will automatically run the automated build process: compiling all the code, running all the tests, and possibly creating additional reports concerning the quality of our code and our tests. The code quality of tests can be measured in terms of a few metrics using code coverage and mutation testing. Other metrics are based on static analysis mechanisms, inspecting the code in search of bugs, code smells, and vulnerabilities. We will use **SonarQube** and its free cloud version **SonarCloud** for such a static analysis.

When we need our application to connect to a service like a database, we will use **Docker**, a virtualization program based on containers that is much more lightweight than standard virtual machines. Docker will allow us to configure the needed services in advance, once and for all, so that the services running in the containers will take part in the reproducibility of the whole build infrastructure. We will not need to install the services on our development machine. We will bring them up when we need them and remove them when we do not need them anymore. The configuration of the services will be used in our development environment, during build automation, and in the CI server.

The chapters of the book are meant to be read in order. Each chapter refers to concepts, and topics explained in the previous chapters.

Most of the chapters have a “tutorial” nature. Besides a few general explanations of the main concepts, the chapters will show lots of code. Following the chapters and writing the code to reproduce the examples should be straightforward.

The book’s primary goal is to give the basic concepts of the techniques and tools for testing, build automation, and continuous integration. Of course, the descriptions of these concepts in this book are far from exhaustive. However, you should get enough information to get familiar with all the presented techniques and tools.

The Java code presented in the book is simple and the examples presented are trivial toy applications. However, this does not mean that the presented techniques and tools cannot be applied to real-life applications. Conversely, all the presented techniques and tools are meant for complex and real-life applications. However, when we have to learn something new, it is better to focus on the new things we have to learn and make the surrounding context as simple as possible. The Java code in this book is just the context to learn these techniques and tools, which are the book’s main topics. Thus, the Java code will be kept simple. Moreover, Java is given for granted: the book will not provide any information about the Java programming language.

## Conventions

Commands to be inserted from the terminal are preceded by \$. If a command is too long, it is split into several lines, separated by \.

We will use several kinds of boxes in the book:



This is a warning box.



This is an error box.



This is an information box.



This is a tip box.



This is a discussion box.



This is an exercise box.

## Code of examples

The source code of the examples of this book is freely available on GitHub. Each example lives in its git repository.

The main repository is

<https://github.com/LorenzoBettini/tdd-buildautomation-ci-book-examples>

The repository contains all the links to all the repositories with book examples, classified by chapters.

If in a chapter, we start from an example implemented in a previous chapter, then there will be two or more git repositories, one for each chapter.

You should do all the examples from scratch by yourself, following the book's chapters. The final implemented code can then be compared with the code of our git repositories.



All the examples have been developed in a Linux operating system. All the code and tools should also work in Windows and macOS. However, Docker might require a few additional configurations when used in an OS different from Linux.

## Errata

If you find an error in the book, please open an issue on the following repository:

<https://github.com/LorenzoBettini/tdd-buildautomation-ci-book-examples>

I will fix it as soon as possible and release a new book version.

If you find an error in one of the examples of the book, please open an issue on the repository of the example. As said in the previous section, the above central GitHub repository contains all the links to all the repositories with book examples, classified by chapters.

## Install the JDK

You can use the standard mechanisms for installing the JDK, downloading the binaries from <https://jdk.java.net/> or <https://adoptium.net/>. Alternatively, you can install the JDK with the package manager of your operating system. If you use a Linux distribution, you should install the JDK (or several versions of the JDK) through the distribution's package manager. Typically, a Linux distribution also provides mechanisms to switch to a specific Java version from the command line in case you installed several JDK versions.

You may want to take a look at SdkMan <https://sdkman.io/>. Once you install SdkMan, you can use this command-line tool to install the JDK and other related tools, like Maven. You can install several versions and specify the default one. Then, in a terminal window, you can easily switch the default version and use another version only for that terminal session.



To run Eclipse, you only need a Java JRE (i.e., the virtual machine only, without the compiler: Eclipse has its own Java compiler). However, for other tools like Maven, you need the Java compiler included in the Java JDK (Java SE, Standard Edition). Recent versions of Eclipse distributions (like the one for Java developers) started to package a JRE so that you could run Eclipse even if you have no Java installed. In any case, you should install a JDK on your computer to follow this book.



To write the code following the book, you must use Java 8. While more recent versions of Java are already released, using Java 8 allows us to demonstrate a few things in the book better incrementally and switch to more recent versions of Java during the book. At some point, we will also need to use Java 11.

# Book structure

The book is structured as follows:

- Chapter *Testing* introduces a classification and provides a general description of a few types of tests, which will then be used throughout the book.
- Chapter *Eclipse* gives a brief introduction to the main features of Eclipse so that you can get familiar with its main tools, which will be used throughout the book.
- Chapter *JUnit* provides a brief introduction to the main testing framework we will use in the book, JUnit. We will also start to see a few best practices in writing tests and a few additional testing libraries that can be used with JUnit, like AssertJ.
- Chapter *TDD* describes the methodology of Test-Driven Development with a few examples. We will use this methodology in most parts of the book.
- Chapter *Code coverage* introduces the mechanisms of code coverage (showing how many parts of the code are covered by tests) as a possible metric to have some confidence in the quality of our tests.
- Chapter *Mutation Testing* introduces a more robust metric for the quality of our tests, using a testing framework, PIT, which automatically mutates a few parts of the code and checks whether our tests detect such changes.
- Chapter *Maven* describes the main features of Maven, a system for dependency management and build automation, which we will then use in the rest of the book.
- Chapter *Mocking* describes the technique of replacing dependencies of the class under test with “mocked” implementations when writing unit tests. In particular, it provides a few examples using the testing framework Mockito.
- Chapter *Git* introduces the main features of the Git Version Control System, which we will then use in the rest of the book, and GitHub, the hosting service for our Git repositories.
- Chapter *Continuous Integration* describes the process of Continuous Integration using the free online service GitHub Actions, showing a few examples. The examples in the rest of the book will use this process and this online service.
- Chapter *Docker* introduces the main features of the virtualization program Docker, which we will use in the rest of the book for automatically starting and stopping services that are needed by our applications, in particular, during tests.
- Chapter *Integration tests* shows how to write integration tests, that is, tests that verify the correct behavior of an application component when integrated with other components and external services. We will use Docker to start the external services in our tests using the testing framework Testcontainers. We will also see how to start and stop Docker containers during the Maven build.

- Chapter *UI tests* shows how to write both unit and integration tests for verifying the correct behavior of the GUI of an application. In particular, we will use AssertJ Swing for testing a GUI written using the standard Java window toolkit Swing. We will also sketch how to write tests in the presence of multithreading.
- Chapter *End-to-end tests* shows how to write special integration tests, which test the whole application through its user interface. The chapter also shows the use of Cucumber, a BDD testing framework, for writing and implementing tests with high-level specifications, which can be easily understood even by users and clients who are not developers.
- Chapter *Code Quality* shows how to use the program SonarQube to detect bugs, vulnerabilities, code smells, and, in general, bad practices in our code. We will also show how to use such a program on the cloud, using the free service SonarCloud, and how to use it directly in Eclipse using the SonarLint plugin.
- Chapter *Learning tests* demonstrates how tests can be used as a learning tool to get familiar with a third-party library or a framework. In particular, we will learn and use the Dependency Injection framework Google Guice.

# 1. Testing

In this chapter, we provide a classification of automated tests. There is no absolute universal consensus in the literature about definitions of several kinds of tests. The classifications and definitions we give in this chapter will be used throughout the book. However, in other books and articles, these definitions may vary, sometimes even substantially ([App15](#), [FG15](#), [Fra16](#), [Gar17](#), [Voc18](#)).

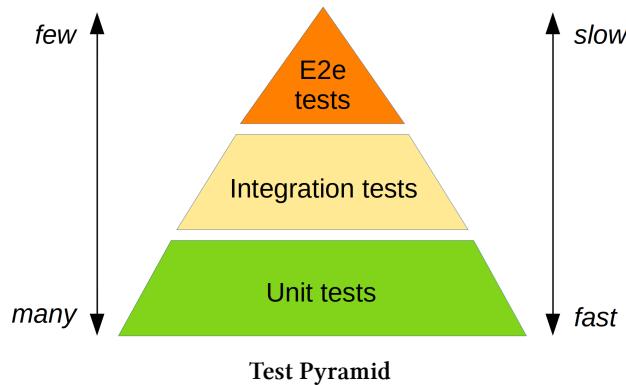
In this chapter, the primary testing concepts are introduced. In the subsequent chapters, these concepts are explained in more detail and with several running examples.

## 1.1 Automated tests

A **test** can be defined as a repeatable process that verifies the correct behavior of a tested component. This process is done in a specific situation, verifying that the output is as expected with a given input.

In this definition, “input” and “output” do not necessarily correspond to data. The “input” can represent the invocation of a Java method with some arguments. The “output” can represent the returned value of a called method or a side effect of such a call, e.g., the interaction with other components. The crucial thing is that the overall process must be deterministic. This requirement can be relaxed in applications where non-determinism is implicit, e.g., multithreaded applications. Even in the presence of concurrency, the determinism of tests should be enforced. At least, non-determinism should be kept to the minimum, e.g., using timeouts. In this book, we only show a minimal example using multithreading (Chapter [UI tests](#), Section [Multithreading](#)).

We can first classify tests by **levels** or **layers**. A great visual metaphor for describing the different layers of testing is the **Test Pyramid** ([Voc18](#)). You can see many versions of the pyramid in the literature. In this book, we use the one shown in the following figure.



The pyramid describes three primary levels of tests:

## Unit tests

In unit tests, single program units are tested. In Java, these program units are typically classes and, in particular, methods. An essential requirement of unit tests is that they should test a single unit in *isolation*, that is, independently from any other components it interacts with. We will see how to achieve this isolation. We anticipate that in unit tests, the external components will be replaced by *fake* implementations (Chapter [Mocking](#)).

## Integration tests

In integration tests, single units are combined to verify whether they still work as expected, typically assuming they have already been tested in isolation with unit tests. Usually, integration tests should test the collaboration of *at least two components*.

## End-to-end (e2e) tests

In e2e tests (often called also **system tests**), the whole application is tested. These tests verify that all the components interact correctly to accomplish the main goals of the application. To some extent, e2e tests are special integration tests where all the system components are integrated.

The tested component is often called **SUT, System Under Test**. The SUT depends on the type of test. Typically, the SUT in unit tests is a single Java class (we exercise method invocations on an instance of the class). In e2e tests, the SUT is the entire application (we interact with the user interface). In integration tests, you can still have a single SUT that interacts with other components or have several SUTs (i.e., instances of several classes).

The shape of the test pyramid highlights several facts. You should have lots of small and fast unit tests. Then you should have a few more coarse-grained integration tests, which are allowed and expected to be slower than unit tests. Finally, you should have very few e2e tests, which, by their nature, are very high-level and very slow.

Besides being fast in their execution (unit test execution time should be measured in milliseconds), unit tests are also meant to be easy to write. Integration tests require more work to connect a few components. E2e tests might be the hardest ones to write since you are meant to interact only with the external interface of the application, ignoring internal details. With that respect, unit tests are usually **white-box tests**: they are based on the deep and detailed knowledge of the internal logic of a component's code. On the contrary, e2e tests are usually **black-box tests**: they should treat the application as a black box, ignoring the low-level details. These tests rely on the specifications of the application in some form. The behavior of the black box should be verified only by checking its inputs and the related outputs. Integration tests are usually in the middle: they are not entirely black-box tests since they can rely on low-level internal details, but not as much as unit tests; thus, they are not entirely white-box tests.

Finally, unit tests are the base of the pyramid. Indeed, it might not make sense to write integration tests for components that have not been tested in isolation with unit tests. The same holds for tests for the whole application. In general, integration and e2e tests should only verify the behavior of already tested components in isolation; thus, they are expected to succeed from the beginning. Unit tests, when employing the TDD methodology (see Section [Test-Driven Development \(TDD\)](#)), are

instead expected to fail at the beginning since they are written before implementing the tested functionality. However, it might make sense to have e2e and integration tests from the beginning. They are expected to fail until all the single components are implemented and tested.

In the literature, you may find versions of the test pyramid with **UI tests (User Interface tests)** at the top, assimilated to e2e tests. UI tests are meant to verify the user interface of an application. The application could be a command-line application, a **GUI (Graphical User Interface)** application, or a web application (where the UI is the web page accessed through a browser). For example, in these tests, we verify that when the user provides some input in some text field and presses a button, something happens in the user interface: the state of the UI should change, some data should be presented to the user, etc. However, in this book, we consider UI tests and e2e tests as orthogonal concepts. Of course, e2e tests are usually UI tests since testing the whole application, as a black box, usually means testing it through its UI. However, the inverse is not necessarily true. Indeed, the UI can be tested with unit tests in isolation from its dependencies. As we will see in Chapter [UI tests](#), UI tests will be unit and integration tests. In Chapter [End-to-end tests](#), UI tests will be e2e tests for the same application.

UI tests are not as easy to write as unit tests since interacting with a UI requires additional effort. In particular, GUIs are inherently event-driven; thus, UI tests should “join” these events. Several testing frameworks make UI tests less painful to write, for example, providing an API that simulates user actions and gives access to GUI controls programmatically, hiding the internal details of GUI events. Moreover, UI unit tests might be slower than standard unit tests since the UI (e.g., a graphical window with widgets) adds some overhead during the tests.<sup>1</sup>

If something in the UI code changes concerning the layout of the components (text fields, buttons, etc.), the UI tests might still succeed, even if you broke the beautiful layout. In the end, the beauty of the layout of the UI must be verified manually. Still, the correct behavior of the UI can be tested automatically.

Since unit tests are meant to be fast, they are run as often as possible after each modification or refactoring to the codebase. Integration and e2e tests are usually run now and then by the developers. They are typically executed in a dedicated *Continuous Integration* server (Chapter [Continuous Integration](#)).

We saw that unit tests are meant to be very low-level, relying on the internal details of the SUT. They usually consist of testing individual methods of the classes. Indeed, unit tests are meant to be as exhaustive as possible. Of course, it is impossible to test a method for all possible inputs, for example. However, unit tests should test at least all possible paths of a method’s logic. For example, suppose a method is expected to throw an exception under certain conditions. In that case, a unit test should recreate such a condition and verify that the exception is effectively thrown. Suppose a method contains an “if..then..else”; both branches of the selection statements must be tested. Finally, comparison conditions, such as, e.g.,  $>$  or  $\geq$ , should also be tested for boundary cases. For example, a condition of the shape  $e > 0$  should be tested when  $e$  is less than 0, greater than 0, and when it is 0 (which is usually forgotten).

---

<sup>1</sup>Note that, generally, UI testing frameworks will not allow you to verify whether the UI “looks good”.

On the contrary, integration tests (including e2e tests) are not meant to verify all the possible paths of every single component but only the interesting cases (this, of course, depends on the application). Writing integration tests for exceptional and error cases might also be useless: Unit tests have already verified these. Moreover, while it is easy to recreate exceptional contexts and boundary situations in unit tests, it might not be easy to recreate such situations when writing integration tests.

Despite the more significant effort to write and maintain integration and e2e tests, if something changes in the implementation of single components, these tests are unlikely to fail unless the external interface changes significantly. On the contrary, changing a small detail in a single component, e.g., even changing a condition from `>` to `>=`, is expected to make at least one unit test fail. In Chapter [Mutation Testing](#), we will see that if there is no such a failure, then something is wrong with the design of our unit tests: they do not test the code correctly and comprehensively.

Tests should be straightforward to read and understand. As a form of executable documentation, at any level, it is crucial to make extremely clear what a test verifies in which specific context and scenario. It should be straightforward to know what a method does in a specific scenario by looking at its tests. For this reason, having lots of abstractions and code reuse in tests could quickly become an anti-pattern. Code duplication is admissible in tests if it makes them easy to read. On the contrary, tiny tests that rely on reused abstractions and utility methods, requiring a lot of code navigation, might be harmful.

In general, independently from the type of test, it is crucial to write tests that effectively recreate and test a specific behavior. False positives are the worst thing in automated tests, even worse than false negatives since they provide false confidence in the correctness of the code. We will use automatic tools that help in analyzing the effectiveness of automated tests (e.g., Chapter [Code coverage](#) and Chapter [Mutation Testing](#)). However, in the end, it boils down to acquiring skills in writing good tests.

Tests, especially at the unit level, should only be written for verifying code (e.g., methods) with some logic. Thus, writing unit tests for getter and setter methods is entirely useless unless getters and setters have some logic (e.g., some conditions and selection statements).

Of course, there are many other types of testing ([acceptance testing](#), [performance testing](#), etc.), which we will not cover in this book. In this book, we concentrate on the classification provided in this chapter.

## 1.2 Advantages of automated tests

When written correctly and effectively, tests, at any level, are a form of *executable documentation* that never gets stale. Keeping the traditional software documentation (manuals and Javadoc) in sync with the actual implementation is challenging. If some parts of the code change, the documentation will be out of date, if not maintained accordingly, losing its usefulness and increasing its harm. Readable tests are documentation: they provide executable examples of the application (at any level). If the application changes, some tests will fail, and you will be forced to fix them, implicitly keeping the executable documentation up-to-date.

**Refactoring** (Fow18) is the process of restructuring the code without changing its behavior. This restructuring is usually done to improve the code, e.g., to make it more readable, more maintainable, and, in general, *cleaner* (Mar08). Automated tests are crucial to refactoring with confidence. On the contrary, refactoring is risky without automated tests to confirm that the behavior did not change. In most cases, unit tests alone provide the required confidence in refactoring.

Automated tests, particularly unit tests, also make bug finding easy. The bug is likely to be in the parts of the application that are not tested. Even in the presence of excellent and complete code coverage, the automated testing methodology, and, in particular, TDD, leads to a systematic process for finding and fixing bugs: first, reproduce the bug in a failing test, then work on the code involved in that test until the test passes. Debugging is usually unnecessary when tests are written before the code and the code coverage is high.

## 1.3 Test-Driven Development (TDD)

As hinted above, it is crucial to have confidence in the effectiveness of tests. Tests verify the correct behavior of the code, but who tests the tests? How can we be sure that the tests are correct?

This is one of the several goals of **Test-Driven Development** (Bec02). This methodology, described in more detail in Chapter [TDD](#) and used throughout the book, is based on fast development cycles. First, we write a test for a feature not yet implemented and ensure the test fails. Since a test is written before the implementation, it is supposed to fail; otherwise, the test is a false positive. Then, the corresponding code must be implemented, but only to make the test pass. This process gives enough confidence in the effectiveness of tests.

This does not hold if tests are written after the code. Of course, it is better to write automated tests after implementing the code than to have no automated tests. However, TDD tells you that writing tests after the code is far from optimal.

The development cycle in TDD then basically consists of 3 states:

**red** write a test for a feature yet to implement and make sure it fails;

**green**

    write just the code to make the test pass, and make sure that all tests still pass;

**refactor**

    if needed, clean up the code and ensure all tests still pass.

TDD treats tests as executable specifications, and specifications are written before the code that implements them, not the other way around. A specification written after the code is done would only formalize the current behavior, losing its intrinsic value. The same holds for tests.

Moreover, TDD is not only about testing: it is a methodology to approach the design of the code. Writing tests first will force us to express the requirements of our code before implementing it. It is also one of the best ways of defining the structure and interface of our code since we write the test

using our code before its interface has been defined. What can be better for designing the interface of a class than to first write code (i.e., the test) that will use it?

TDD will implicitly force you to write code that is easy to test, modular, and loosely coupled. On the contrary, writing tests after the code might be more complicated since the code probably has not been written with testability in mind.

TDD is not necessarily only about unit testing. TDD can be used in all kinds of tests, including integration and e2e tests. However, when using TDD with unit tests, the development cycle is expected to quickly fix the failing test by implementing only the code to make the failing test succeed. This should take only a few minutes. When applied to integration or e2e tests, that is, before implementing the code used by such tests, those tests will fail for some time, even days or months. In fact, the single components have to be implemented first. In particular, e2e tests written in advance will likely fail until the application is implemented.

Thus, TDD with unit tests forces you to start a new cycle only after fixing the single new failing test. This requirement, of course, does not hold when writing integration and e2e tests in advance.

### 1.3.1 Behavioral-Driven Development (BDD)

All the tests we write in this book will be written in Java, relying on JUnit. Indeed, the “Unit” in “JUnit” does not mean it must be used only for unit tests. We will rely on additional testing frameworks, but in the end, our tests as specifications will be written and read in Java.

When writing high-level specifications, it might be better to use a high-level language that could be easily read and understood by non-programmers, e.g., the clients and the final users. Moreover, documentation implemented through unit tests is often not enough: unit tests are usually low-level, with too many insights into details, making it too easy to miss the whole big picture.

**Behavioral-Driven Development (BDD)** is a methodology designed to keep the focus on the final user throughout the full project development. Indeed, it is a form of TDD: specifications are defined in advance, and then the components are implemented according to those specifications. Such specifications are high-level, and before satisfying them all, many small components (implemented with TDD and unit tests) will have to be implemented first. Integration tests will then be written to verify the interaction of the single components. Finally, e2e tests will be written to show that the high-level specifications are satisfied.

As mentioned above, the development cycle duration radically changes if we start with high-level specifications written in advance, like in BDD. We move from red to green in minutes with TDD and unit tests. On the contrary, the high-level approach implicit in BDD might require even days to conclude a development cycle. Since the audience of BDD is mainly the final user, the specifications should be written in a language close to the natural language. Several frameworks for BDD allow you to write high-level specifications using a simple Domain-Specific Language (DSL) using English-like sentences to express the behavior and the expected outcomes. Such specifications describe scenarios and user stories with the following typical structure:

- *Given* [some initial context],
- *When* [some event occurs],
- *Then* [something should happen].

Then, Java code must be written to implement all the “steps” of the user stories. The BDD framework will then run the Java code and connect the results with the initial specifications. Of course, BDD could also be employed at the unit test level, but typically it fits better when describing higher-level scenarios. The idea is that a BDD scenario and user story can be written by anyone without coding skills, abstracting from internal implementation details.

Thus, with BDD

- You start defining the high-level specifications with scenarios (or user stories)
- You start working on a scenario
  - implement single components with TDD and unit tests for that scenario
  - verify the integration of components with integration tests
  - validate the scenario, possibly with an e2e test
- You go on with the next scenario

In this book, we will not follow the BDD development cycle since we aim at giving the initial notions of unit, integration, and e2e tests (including UI tests), relying on TDD incrementally. However, near the end of the book, when dealing with e2e tests, Chapter [End-to-end tests](#), Section [BDD with Cucumber](#), we will use a framework for writing specifications in a BDD style, and we use it for e2e tests. We also show an example of the BDD cycle: we first specify a new high-level scenario for our application and gradually implement it following the steps sketched above.

# 2. Eclipse

This chapter describes the main concepts and mechanisms for using Eclipse proficiently for the book’s aims. This chapter is not meant to be an extensive tutorial on how to use Eclipse. For example, we will not describe how to debug a Java program since that is not a goal of the book. On the contrary, refactoring and quick assistant mechanisms are crucial for efficiently writing tests, especially following TDD, and cleaning up the code.

Download an Eclipse distribution from <https://www.eclipse.org/downloads/packages/>. You can now use the distribution “Eclipse IDE for Java Developers”. We will provide further instructions when we need additional plugins throughout the book.

Unzip the archive (zip or .tar.gz) in any directory of your computer.

Alternatively, you can use the Eclipse Installer (see the blog post <https://www.lorenzobettini.it/2015/05/using-the-new-eclipse-installer/>).

To run Eclipse, just run the executable in the main unzipped folder.



In this book, we are using the version of Eclipse that was released when the book was published. If you are using an older version, the views, commands, and preferences shown in this chapter might differ, and some features might not be available. Similarly, you might experience differences if you use a more recent version.

## 2.1 Eclipse workspace

When you run Eclipse, you’ll be asked to select a “workspace”.

An Eclipse **workspace** is a file system directory containing all the settings, preferences, and, most of all, your open projects. The selected directory for the workspace will be automatically created if not already present. Note that while the workspace metadata are physically stored in the workspace directory, open projects might not be physically stored in the workspace directory, as we will see later.

It is common to have several workspaces, each with somehow related projects. Using a single workspace with all your projects might make Eclipse less efficient.

The first time you use a new workspace, the “Welcome” view will be shown (with a few links to documentation and tutorials); to access the actual development workspace, you need to select “Workbench”.

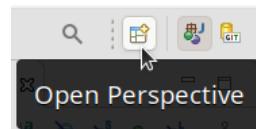


## Resource in Eclipse

In Eclipse, any element in a workspace is called **resource**. From our point of view, resources will be projects, files, and folders. This concept must not be confused with the concept of resources in Java.

## 2.2 Perspectives and views

A **perspective** in Eclipse is a set of **views** arranged in a specific way to make a specific task easy (e.g., for the actual Java programming, for Java debugging, etc.). There are already a few predefined perspectives, e.g., “Resource”, “Java”, and “Debugging”.



Toolbar button for switching or opening a perspective

You can change perspective with the sub-menu in “Window” or by clicking the toolbar “Open Perspective” in the top-right corner (a dialog will pop up to select the perspective). The available perspectives depend on the installed plugins in your Eclipse distribution. We will use the “Java” perspective most of the time. In the top-right corner, you can quickly switch between the perspectives you have already opened in the past.

Most Eclipse views have a “View Menu” represented by vertical dots (or a down arrow in older versions of Eclipse). By clicking the menu, we can modify specific configurations of the views. View configurations are an example of meta-data stored in the workspace.



The “View Menu”

## 2.3 Projects

An Eclipse **project** could represent an application, a library, etc., and it can contain source code written in one or several languages. Suppose you have installed a plugin for a given language or a particular kind of project. In that case, Eclipse will automatically handle the compilation of your sources and provide tools to run the project.

For the moment, we will use Java projects.

An Eclipse project consists of a folder in your file system with all your sources and the file `.project`. The file `.project` contains the metadata about your project (e.g., its name) and the kind of project (e.g., in our case, a Java project).

An Eclipse Java project will contain other files, e.g., `.classpath`, with all the information about the folders with Java sources (source folders), the folders where the compiler will create the binary `.class` files, and possible libraries (e.g., jar files) used by that project.



Files starting with `.` are hidden by default in an operating system's typical file system explorer. You should enable the visualization of those files in your OS to see them with your file explorer. Similarly, by default, they are hidden in Eclipse; you might want to enable their visualization. For example, for the “Package Explorer” view, select the “View Menu” and “Filters...” and then unselect the “`*` resources” checkbox.

Each Eclipse project can have several **natures**, specifying the kind of project and, in particular, the kind of source code in the project. Natures are specified in the `.project` file and typically added/removed using context menus or relying on a project wizard, which automatically adds the appropriate natures.

For example, the “Java Project” wizard, which we will use in Section [Creating a new Java project](#), automatically adds the “Java nature” to the project. Project natures are strictly connected to project builders, which we will see in Section [Automatic building](#).

### 2.3.1 Importing an existing project

You can import an existing project in the workspace with  
File → Import... → General → Existing Projects into Workspace.



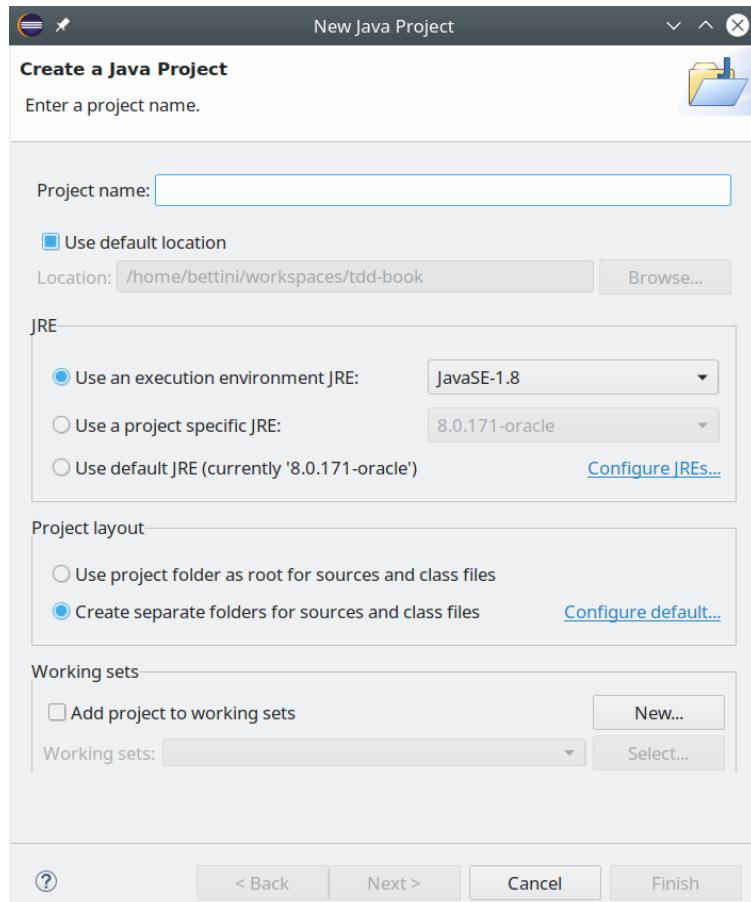
This import is a “virtual import”: the original project folder will NOT be physically copied into the workspace folder, and the project folder will keep living in its original folder. The checkbox “Copy projects into workspace” can be selected if needed, but, in general, it is considered best practice to keep the physical locations of workspaces and projects separate.

### 2.3.2 Creating a new Java project

Select File → New → Project... → Java → Java Project.<sup>1</sup>

---

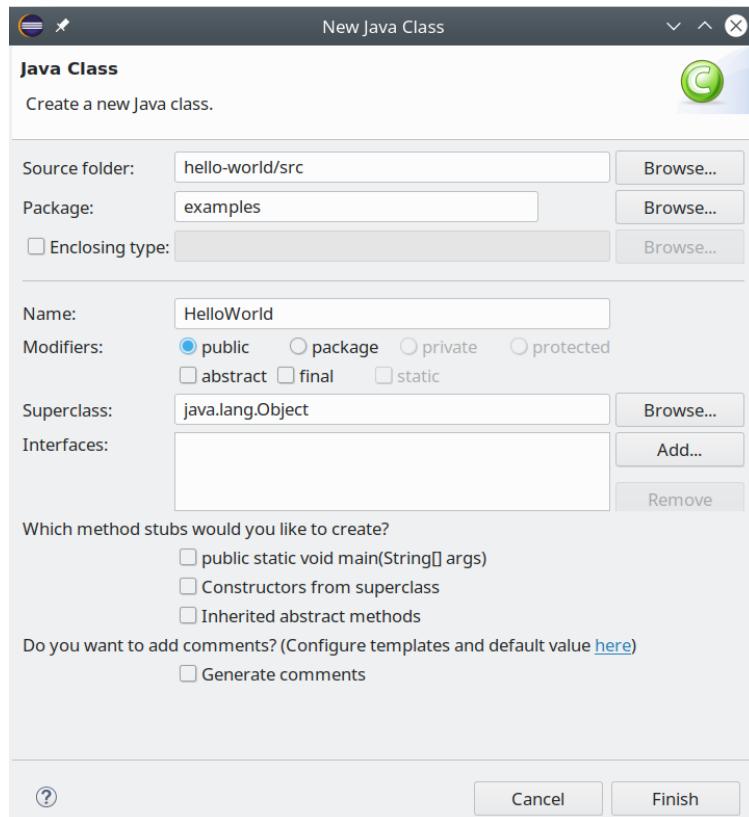
<sup>1</sup>Depending on the perspective, the menu path could be shorter; for example, in the Java perspective, it is enough to select File → New → Java Project.



Specify the name of the project, which is the required information. By default, the project will be physically created inside the workspace directory. As hinted above, this is discouraged. It is much better to specify a directory for the project which is physically unrelated to the workspace directory. Just unselect the checkbox “Use default location” and specify another directory, where Eclipse will physically create the project’s folder structure. The project name and the final directory can be different, even though you typically use the same name for both. It is instead best practice to keep the directory for sources and the directory for class files separate (i.e., make sure you select “Create separate folders for sources and class files”).

For example, specify `hello-world` as the project’s name and a directory unrelated to the workspace directory. Then press “Finish”. We skip the next page of the wizard for the moment since we do not need to customize the project further.

Let’s create a new Java class: right-click on the source folder `src` and select `New → Class` (or use the corresponding toolbar button). In the dialog, specify the properties of the new class, e.g., package name and class name (the available configurations in this dialog should be straightforward to understand):



### 2.3.3 Open projects and closed projects

A project in a workspace can be **closed** or **open** using the corresponding contextual menu: right-click on the project and then “Close project” in case the project is open, or “Open project” in case the project is closed. A closed project can be told by its simple folder icon and the fact that you cannot expand it.

When a project is closed, its Java types are not visible in the workspace (see the following sections).

You can “Delete” a project from the workspace. You will not physically delete the project’s directory unless you select the corresponding checkbox in the dialog that pop-ups when you delete a project.

### 2.3.4 Automatic building

Project natures are strictly connected to **project builders**. For example, a Java project is configured with the “Java nature” and the “Java builder”. A project can also have several associated builders.

When a resource changes inside a project, the builders are notified and can perform tasks such as validating and generating other resources.

For example, the Java builder will validate the project’s Java sources according to the changed Java files and will, in case, regenerate the corresponding .class files (by default, in the output directory

`bin`). The same happens when a Java file is physically deleted: the Java builder will remove the corresponding `.class` files.

Eclipse plugins provide natures and builders. Thus, a project with associated natures and builders only benefits from the Eclipse building mechanisms if the corresponding plugins are installed.



In Chapter [Maven](#), we will create Java projects that also have the “Maven nature” and the corresponding “Maven builder”. The Maven builder will validate its files and possibly download dependencies (these concepts will be clear after reading that chapter).

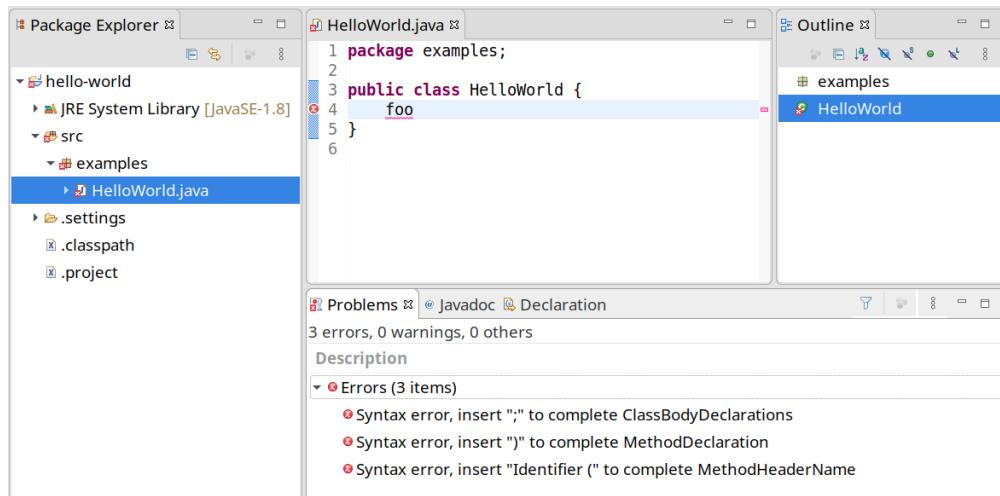
Builders are meant to be incremental. For example, the Java builder keeps track of dependent files and recompiles them only when needed.

The building mechanism is, by default, active in Eclipse. The corresponding checkbox menu is **Project → Build Automatically**. You can manually trigger a complete rebuild of a set of projects with **Project → Clean...** (you can select the projects or clean all the opened projects in the workspace).

Besides validating the project and generating files upon resource changes, that is, saving or removing, Eclipse plugins, like the Java plugin, automatically validate resources when the user is typing in the editor.

Before saving an edited file, possible validation errors are shown in Eclipse as error markers on the sidebars of the editor (**editor ruler**) and in the editor itself, in correspondence with the invalid parts of the file. Moving the mouse pointer and stopping over a part highlighted in red (this operation is called **hovering**), a pop-up window will give you further details about the error. The same happens when hovering on a marker in the editor ruler.

After saving or removing resources, possible errors are also shown in the “Problems” view. In such a case, error markers are also placed in the project tree, starting from the invalid resource to the project itself. Error markers are also shown in other views of the workspace.



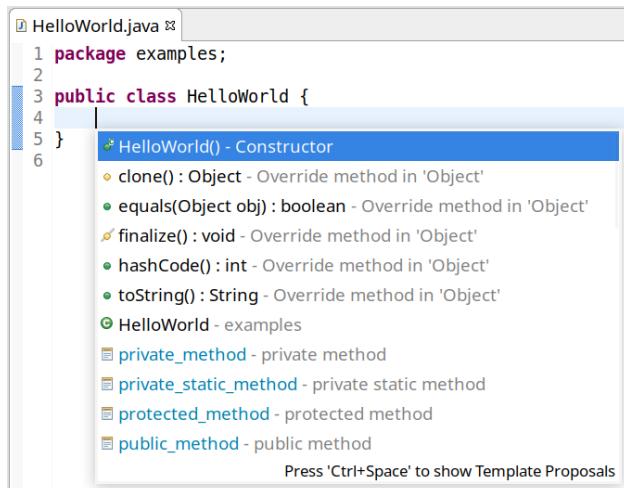
Error markers shown in several parts

## 2.4 IDE tools

Besides the automatic building mechanisms and editor features like syntax highlighting, the main power of an IDE consists of the tools it provides to the programmers so that they can easily write code as quickly as possible, avoiding repetitive and manual tasks. In this section, we review some of these tools provided by Eclipse, focusing on the ones we will use throughout the book.

### 2.4.1 Content Assist

With the key combination **Ctrl+Space** inside a Java editor, a list of possible completions is shown that make sense in that specific program context.

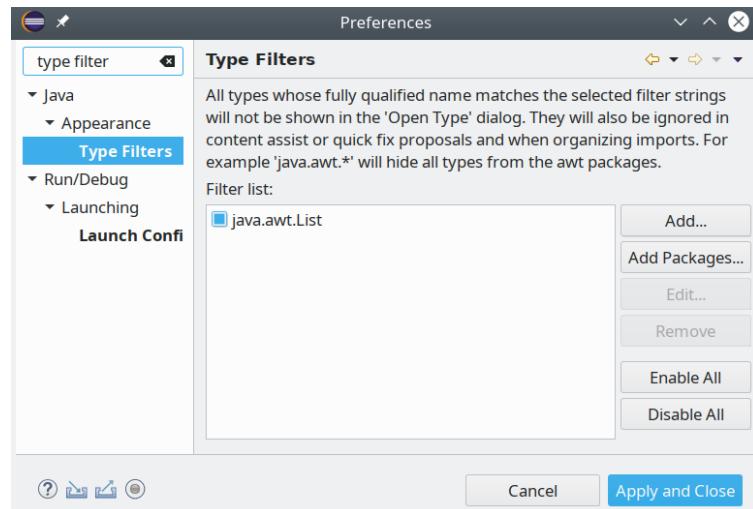


Pressing **Ctrl+Space** again, while the pop-up is already active, toggles among several kinds of proposals. Pressing **ENTER** (or selecting with the mouse) over a selected completion will make Eclipse insert the corresponding text in the file.

When hovering over a specific proposal representing a Java type or a member, another pop-up will open with the corresponding Javadoc.

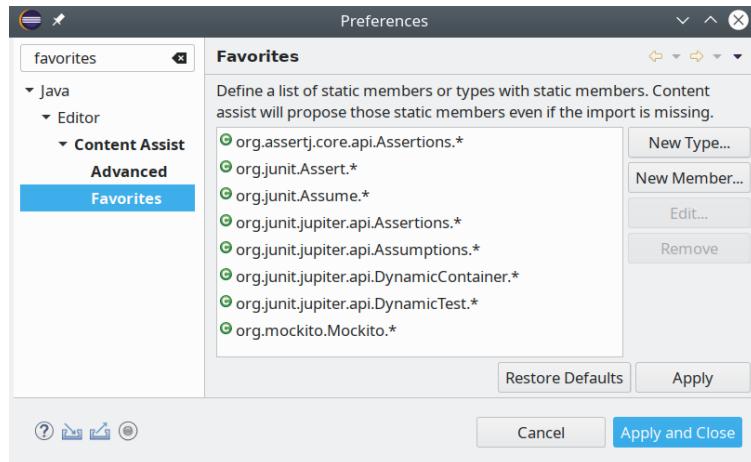
When a content assist pop-up is active, you can start writing to filter the suggestions.

It might be helpful to configure a few Java “Type Filters” using the corresponding preference. You can list the fully qualified name (possibly with wildcards) of the types you want to be ignored by the content assist. The typical example is `java.awt.List` which by default comes before `java.util.List`. Usually, you want to use the latter, and probably, you will never need to use the former (since AWT is obsolete). We will still need a few AWT classes in Chapter *UI tests*, so instead of ignoring the whole types from AWT with `java.awt.*` we only ignore `java.awt.List` as shown in the following screenshot:



In recent versions of Eclipse, some predefined type filters are already configured, including the one for `java.awt.List`.

Using the “Favorites” preference of the Java content assist is also helpful. You list the types with static members, and the content assist will propose such members even if the import is missing (and the corresponding static import will be automatically inserted):



A few types are already present in the default configuration, including AssertJ assertions (which we will first show in Chapter [JUnit](#), Section [Using Assert](#)) and Mockito (which we will first show in Chapter [Mocking](#)). We will use the static members of these types extensively throughout the book.

Besides proposals related to Java syntax (e.g., a keyword, a type, a method override, or a method invocation), Eclipse provides some **Template Proposals** to avoid manually writing typical blocks of code. In the above screenshot of the content assist, `private_method` is an example of a template proposal. When hovering over a specific template proposal, another pop-up will show a preview of what will be inserted in the file in case that proposal is selected.

For example, from within our `HelloWorld` Java class, press `Ctrl+space`, type `m`, and choose the template proposal `main` method that will automatically insert the typical Java `main` method and place the cursor inside the method block. Now press `Ctrl+space` and write `syso`, and you will obtain another template proposal for the typical `System.out.println` statement. The cursor is placed inside the `()`. Type "Hello World!". This procedure is a quick way to write a Java Hello World program.

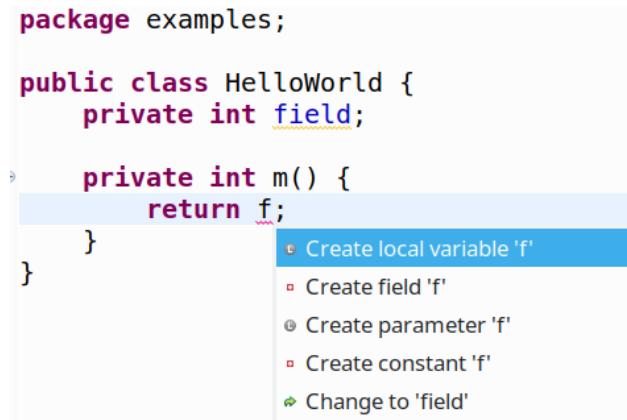


The Java editor will automatically insert the closing character when inserting a character like `", (, [,`, etc. Inserting a character like `{` and pressing `ENTER`, the Java editor will automatically insert the closing `}` and place the cursor inside the block after inserting the indentation character(s).

If a proposal refers to Java type, Eclipse will automatically insert the corresponding import statement (unless that type is already imported).

## 2.4.2 Quick Fix

When there's an error in your source files, Eclipse can suggest a few quick fixes for that specific error. For example, refer to a non-existent symbol. Eclipse can create for you a field, a parameter, or a local variable with that name, or it can suggest you change the reference to an existing declared element (guessing that you misspelled the reference).



You can have the list of quick fixes by clicking on the editor rule in correspondence with the error marker, by hovering on the editor section containing the error, or from a marker's context menu in the Problems view. Without using the mouse, if you navigate to the editor section containing the error, you can access the quick fixes with the shortcut **Ctrl+1**.

The ability to have Eclipse create missing classes, fields, and methods automatically will be crucial for TDD, as we will see in Chapter [TDD](#).

### 2.4.3 Quick Assist

Quick Fix (and its shortcut **Ctrl+1**) is also available when no error is present in the current file. In such a case, it is called **Quick Assist**.

For example, after creating a field in a class, press **Ctrl+1** on the field name, and Eclipse provides a menu for creating getter and setter methods.

After writing an expression, Quick Assist allows you to assign that expression to a new local variable or a new field. Of course, the corresponding type of the declared element is automatically inferred from the type of the expression.

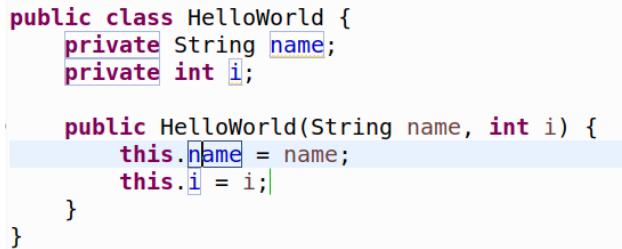
Another example is when you have an empty class and create a constructor taking some parameters, intending to assign those parameters to corresponding fields (which have not been declared yet). When positioned in the constructor's parameters region, invoke Quick Assist and select "Assign all parameters to new fields". The new fields will be created, the parameters will be assigned to the new fields, and some regions will be editable and quickly navigable, as explained in the next section.

### 2.4.4 Editable proposals

When blocks of code are inserted by the tools of the Java editor, e.g., after selecting a Template Proposal or a Quick Fix, some regions of the inserted code might be editable after the code has been inserted. Such regions are also navigable using the TAB key. This happens when the inserted code has parts that are meant to be further customized by the programmer.

For example, this does not happen for the `main` method Template Proposal since the inserted code is always the same. On the contrary, the template for inserting a method declaration, like `private` method, allows us to easily change the return type and the method's name by navigating to the corresponding regions.

Another example is the “Assign all parameters to new fields” Quick Assist described in the previous section. The editable regions are the accessibility level of the fields and their names. Note that the editing regions of a field’s name are connected: changing the field’s name in the declaration will also change it in the field assignment and vice-versa.



```
public class HelloWorld {  
    private String name;  
    private int i;  
  
    public HelloWorld(String name, int i) {  
        this.name = name;  
        this.i = i;  
    }  
}
```

Editable regions after proposal insertion

Such regions disappear on file save, by pressing ESC or focusing on another window. Thus, they are meant to be used right after the proposal insertion.

## 2.4.5 The Source menu

The menu **Source** (available both as a menu and as a context menu of the editor) provides you with many mechanisms to automatically create many parts of a Java source file, e.g., constructors, getters/setters, etc. Many of the submenus also have a keyboard shortcut. To increase your productivity, you should learn by heart the most used shortcuts.

**Format**, **Ctrl+Shift+F**, automatically formats your Java source according to some standard templates. These can be changed (see Section [Eclipse Preferences](#)). This way, your source files will always be neatly formatted, which is crucial for code quality and readability. When a part of the file is selected, only that part will be formatted.

**Organize Imports**, **Ctrl+Shift+O**, organizes the imports in the current file by sorting them, removing unused imports, and adding imports when needed (i.e., when a Java type is used but without the corresponding import). This menu can also act at the package and project level when the corresponding element is selected, e.g., in the “Package Explorer”.

**Clean Up...** opens a dialog where some clean code criteria are used to make the current Java file cleaner (e.g., removing unused imports, adding missing `@Override` annotations, etc.).

## 2.5 Eclipse Preferences

With **Window → Preferences** you access a dialog with all the Eclipse preferences (use the text field to filter by typing the name of a preference).

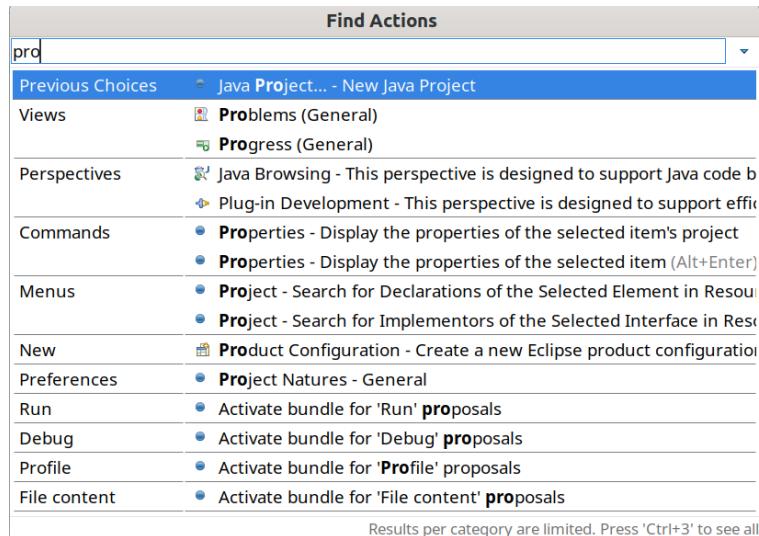
The list of available preferences depends on the installed plugins. Some preferences, e.g., those of Java projects, can also be modified for a single project, and they will be saved in the project's folder in the (by default hidden) folder `.settings`.

You can also enable “preference recording”: each changed preference will be recorded in your home folder and automatically applied to any new workspace created. You can also store such preferences on the Eclipse remote server if you have a free Eclipse account.

## 2.6 Find Actions

One of the most useful (and probably one of the most unknown) features in Eclipse is the **Find Actions** menu on the top right corner (represented by a “magnifying glass”). You activate that either by clicking on it or with the combination **Ctrl+3**. (In older versions of Eclipse, the menu was known as “Quick Access”; in the top right corner, it was represented by a text field.)

This will open a dialog. After activation, the most recent choices are shown. Start typing, and you can quickly access any Eclipse view, command, preference, etc. As noted in the dialog, “Results per category are limited”, and you can press **Ctrl+3** again to see them all.



For example, use Find Actions to create a new Java project and a new Java class (start typing “New Java Project” and select the appropriate entry). The Find Actions is also a valuable mechanism for quickly opening a view in Eclipse, instead of looking for the view to open in **Window → Show View → Other....** You can use it for everything in Eclipse, including all the refactorings we will apply in the book. Instead of navigating to a context menu, use the Find Actions, start typing a word (or parts of words) that you remember is part of a menu, and choose the appropriate one from the proposals.

## 2.7 Eclipse Shortcuts

To increase your productivity, you should learn keyboard shortcuts for the commands you use most. Shortcuts are sometimes shown in menus when available. In any case, you can have a list of all the available shortcuts with **Ctrl+Shift+L**: start typing to quickly jump to the command of which you need to know the shortcut.

Pressing **Ctrl+Shift+L** again will open up the preference page for the shortcuts, where you can specify a few customizations. In particular, you can define a new shortcut for commands that do not have one by default.

Alternatively, you can use the Find Actions described in the previous section to learn the shortcuts of commands.

### 2.7.1 Quick Search

Besides a few shortcuts we have seen so far, a useful shortcut is **Ctrl+Alt+Shift+L**, which corresponds to **Quick Search**. This is a quick way to perform a textual search across all the contents of your workspace. (The corresponding preference page allows you to exclude files from such a search according to a few criteria). A dialog appears, and as you start typing a few letters, the dialog will show incrementally the matching items, that is, the files in the workspace containing the words you typed. From the results, you can quickly navigate to the corresponding file.

## 2.8 Eclipse Run configurations

Depending on the selected file, Eclipse provides **Run Configurations**. For example, in the “Package Explorer”, right-click on a Java file containing the “main” method and see the menu **Run As → Java Application**. The same is available in the standard menu **Run**. The output of the run program will be shown in the “Console view”. If the Java program blocks waiting for keyboard input, you can use the Console view to type something. A button in the main toolbar also keeps track of all the previous run configurations to relaunch anything quickly.

The dialog appearing with the menu **Run → Run Configurations...** allows you to modify a Run Configuration (or to create a new one), e.g., by adding command-line arguments for the Java application in the tab “Arguments”, “Program arguments” or arguments for the JVM. Then, press “Apply” to store the modifications or relaunch with “Run”. You can have many different Run Configurations for the same Java application. A Run Configuration is saved, by default, in the workspace settings. Alternatively, you can save it in the project using the tab “Common” and specify the path by selecting “Shared file”. Launch configurations are XML files with the extension `.launch`. If you save a Run Configuration in a project, you can run that by right-clicking and by selecting **Run As**. A run configuration saved in a project can be reused even when switching to another workspace.

Alternatively, with **Debug As**, you'll run the application in debug mode. A dialog will ask you to switch to the debug perspective. A Run Configuration previously saved can be reused also for debugging.

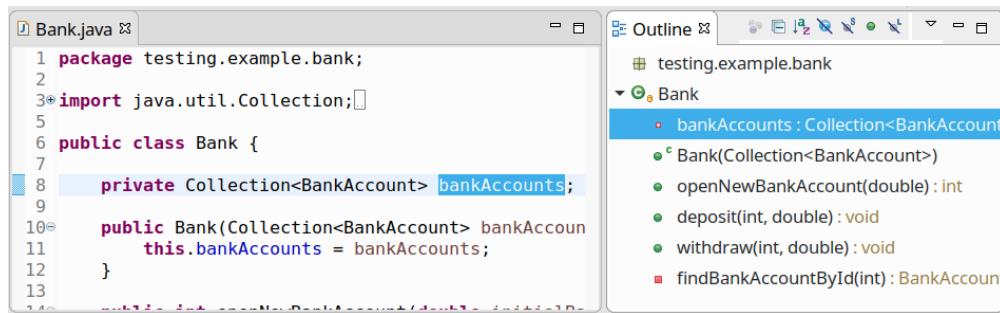
## 2.9 Browsing code

Another important feature that an IDE should provide is related to mechanisms for easily and quickly browsing code.

### 2.9.1 Outline

The “Outline” view in Eclipse, typically on the right side of the workspace, gives you a compact representation of the currently open file in the shape of a tree. Eclipse plugins implement such a representation for their source contents. For example, the Java plugin provides a compact representation of a Java source file.

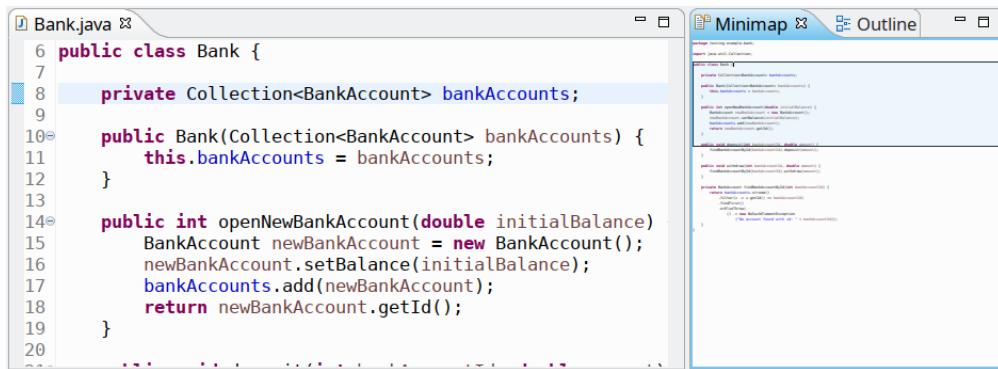
The tree in this view is synchronized with the cursor's position in the editor. Vice-versa, by clicking on a node in the tree, the editor will position the cursor in correspondence with the section represented by the node.



Toolbar buttons in the Outline allow you to change the representation of the tree's nodes. For example, the Java outline allows you to sort the members alphabetically (by default, elements are shown according to their definition order), to hide fields, non-public members, etc.

With the menu **Navigate → Quick Outline** (**Ctrl+O**), you access the “Quick Outline”, a pop-up window with the outline of the current file. Start typing something and you'll filter the view contents. In a big source file, this will allow you to navigate to a field or a method quickly. Press **Ctrl+O** again when the Quick Outline is shown, and you can also see (and filter) inherited members.

The “Minimap” view, recently added to Eclipse, provides a high-level overview of the contents of the active text editor and a better navigation experience.



## 2.9.2 Navigate

The menu “Navigate” provides many mechanisms for navigating through the Java source files (including sources of libraries if available) and, in general, through any resource opened in the workspace. We have already seen the Outline capabilities in the previous section.

From a reference to a member, e.g., a field, using the menu **Navigate → Open Declaration**, you navigate to the corresponding definition. Alternatively, you can press F3. Typically, you use this functionality with the mouse: **Ctrl + (left) click** on a reference. Using the mouse like that, you can also navigate to the original version of a method if you’re overriding the method in a derived class. If you’re on a method declaration in an interface, you can quickly navigate to any implementation of that method.

If you need to navigate to a method definition, possibly in another file, to look at its code, there’s even a quicker way, which does not require you to open the corresponding source file. Indeed, hovering on a member reference shows its Javadoc, and hovering with Shift pressed shows its implementation:

The screenshot shows a tooltip window appearing over some Java code. The tooltip contains the following implementation of the `withdraw` method:

```

public void withdraw(int bankAccountId, double amount) {
    findBankAccountById(bankAccountId)
        .withdraw(amount);
}
public void withdraw(double amount) {
    if (amount < 0) {
        throw new IllegalArgumentException("Negative amount: " +
    }
    if (balance - amount < 0) {
        throw new IllegalArgumentException(
            "Cannot withdraw " + amount + " from " + balance);
    }
    balance -= amount;
}

```

Hovering (with Shift pressed) shows the code of the invoked method in a pop-up

You can use the same mechanism to look at the implementation of a Java type quickly. In particular, focusing on the pop-up windows enables the scrolling of the contents.

Other useful sub-menus are the ones related to showing the type hierarchy of the selected member. The super-type hierarchy can also be shown. A variant, similar to Quick Outline, is the menu **Quick Type Hierarchy** (Ctrl + T), showing the type hierarchy of the currently opened Java file.

With the menu **Navigate → Open Type...** (**Ctrl+Shift+T**), you open a dialog that allows you to open any Java type available in the workspace, that is, defined in any of the open projects or any dependency of the open projects, including JAR libraries. Start typing in the text field, and you'll be presented with the Java types matching what you typed. You can use regular expressions, and you can use the Camel-Case convention.

## Camel Case

It is the practice of writing composed sentences by joining many words, with no spaces, but leaving each word with an initial uppercase. For example, `NullPointerException`. This notation lets you quickly filter Java types in many parts of Eclipse. To open `NullPointerException`, type `NPoE` in the “Open Type” dialog.

Camel Case can be used anywhere, e.g., for content assist, if you want to insert a Java type (for a variable declaration, parameter, etc.).

With **Navigate → Open Resource...** (**Ctrl+Shift+R**), you can quickly open any file (not necessarily a Java source file) available in the workspace.

Select a Java method and use the menu **Navigate → Open Call Hierarchy**, (**Ctrl+Alt+H**), to open the view “Call Hierarchy”, which shows the call hierarchy for that specific method in any class in the workspace. You can do the same for fields and Java types (showing the instantiation call hierarchy).

## 2.9.3 Local History

Eclipse keeps a history, inside the workspace, of the modifications made to files in open projects. You can access this history with the context menu **Show in → History**. The view “History” will show the date and time of all the snapshots saved for each file. Double-click on one of these items, and the editor will show you the contents of that specific snapshot. By using the context menu **Compare With → Local History...**, you'll get a similar view, but by double-clicking on an item, you'll see a split editor comparing the two versions. You can use the context menu **Replace With** to restore a specific version.

When we start using a version control system later in the book, we can use many more features and mechanisms concerning the history of files (Chapter [Git](#)).

## 2.10 Code Mining

A **code mining** represents some content, e.g., a label or an icon, shown along with source text. Examples of code minings are the number of references to a specific declaration, the names of the parameters in a method invocation, and a way to run tests (with run/debug icons). The main goal of code mining is to help developers better understand the code with additional information and

menus on the code itself. Note that the content of code mining is NOT part of the text being edited in an editor. Indeed, such additional information is not even copyable.

Such a mechanism has been recently added to Eclipse, and code mining support for languages is still considered experimental. Besides the standard Eclipse code minings, there are additional code minings for Java and JUnit that are not part of the official Eclipse distribution and update sites. Such support is implemented here <https://github.com/angelozerr/jdt-codemining>. You can also find the update site to install the corresponding plugins at this URL. Once installed, code minings for Java and JUnit have to be enabled using the corresponding preferences. In the preferences, you will have a section for standard “Code Minings” and for the experimental ones, “Code Minings (Experimental)”<sup>2</sup>.

In this case, we enable the method parameter names in method invocations. The Eclipse Java editor will show the names of the parameters of the invoked method, as shown in the following screenshot:

```
private BankAccount findBankAccountById(int bankAccountId) {
    return bankAccounts.stream()
        .filter( predicate: a -> a.getId() == bankAccountId)
        .findFirst()
        .orElseThrow(
            exceptionSupplier: () -> new NoSuchElementException
                ( s: "No account found with id: " + bankAccountId));
}
```

Moreover, we enable labels, links, and icons showing the status of JUnit tests and for directly running tests clicking on the links in the editor (JUnit will be used throughout the book, starting from Chapter [JUnit](#)):

---

<sup>2</sup>Since code mining for Java and JUnit is still experimental and under development, many features, including its preference page, are likely to change after the publishing of this book.

```
1 package testing.example.bank;
2
3+import static org.junit.Assert.*;□
10
11 public class BankAccountTest {
12
13@ Rule
14     public ExpectedException thrown = ExpectedException.none();
15     private static final int AMOUNT = 3;
16     private static final int INITIAL_BALANCE = 10;
17
18@ Test
19     public void testIdIsAutomaticallyAssignedAsPositiveNumber() {
20         BankAccount bankAccount = new BankAccount();
21         assertTrue( message: "Id should be positive", condition: ban
22     }
23
24@ Test
25     public void testIdsAreIncremental() {
26         assertTrue( message: "Ids should be incremental", condition:
27     }
```

The icons showing the last state of the run tests and the “Run”/“Debug” links are implemented as code minings.

# 3. JUnit

In this chapter, we get familiar with the leading testing framework 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 misleading: JUnit is not only for unit tests.

This book primarily uses JUnit 4, <https://junit.org/junit4/>, particularly its latest version, **4.13**. Thus, if we only mention “JUnit,” we refer to JUnit 4. However, in Section [JUnit 5](#), we also briefly introduce JUnit 5, <https://junit.org/junit5/>. In the book’s other chapters, we also sometimes show how the new features of JUnit 5 can be helpful for the subject of those chapters.

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

## 3.1 Structure of a test

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

### Setup

In this phase, we create the environment for the test. Typically, we create the SUT (Software Under Test, System Under Test, or Subject Under Test) in unit tests. In Java, this usually corresponds to creating an instance of the class to test. This phase establishes the state of the SUT before any test activities. The state created in this phase, consisting of the SUT instance and possibly other related objects needed during the tests, is usually called the **test fixture**. A test fixture is all the things that must be in place to test the SUT ([Mes07](#)). It is a fixed state 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, when set up correctly, the test fixture ensures that test results are repeatable.

### 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.

### Verify

In this phase, we verify that the outcome matches the expected behavior, typically using some assertions. We could check 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** ([Bec02](#)), 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:

## Teardown

In this phase, we clean up the environment, bringing it back to the condition it used to be before the execution of the test. For example, if we create a few files, we should remove them in this phase.

## Test fixture

In the book, we will see several examples of test fixtures. Initially, the test fixture will be the SUT instance only. Later, when the SUT needs external collaborators, the test fixture will also include instances of collaborators. In particular, in unit tests, the collaborators of the fixture will be mocked as fake objects (Chapter [Mocking](#)). In integration tests, the collaborators will instead be real (Chapter [Integration tests](#)). Another example is loading a database, possibly with a specific, known set of data. If the SUT accesses files, the test fixture will include a specific known set of files.

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:

- `assertEquals(expected, actual)`: there are several overloaded versions of this method for different types. The method 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 a straightforward meaning.

Assertion methods also have another 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 additional error message since JUnit automatically reports the expected and the actual value in case of failure. On the contrary, if you directly assert a boolean expression, e.g., with `assertTrue`, it might be helpful 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, to verify a single behavior, we must write several assertions in the same test method. Thus, a single test method should exercise and verify a single scenario and behavior; to that aim, it can 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 (i.e., 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 helpful when these phases have a behavior common to all tests.

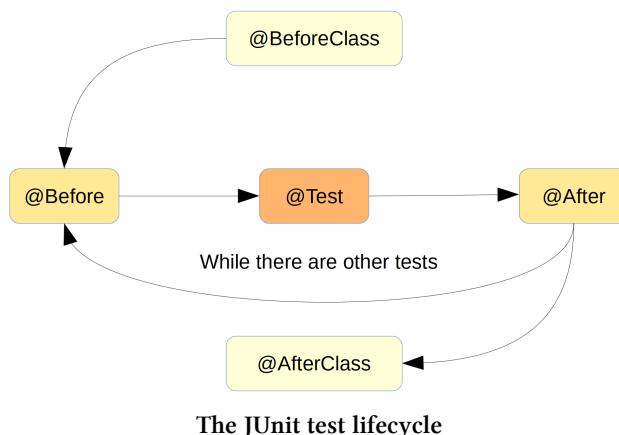
The following annotations must be used with `public static void` methods:

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

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



In the book's first chapters, we will not use the annotations `@BeforeClass` and `@AfterClass`. These are useful when we need to start a service, like a database, that we need to run our integration tests. We start using such annotations in Chapter [Integration tests](#).

JUnit only considers annotations during the test lifecycle. However, it is best practice to name annotated methods meaningfully. This is especially crucial for test methods: their names should describe what the test aims to verify.

Each test method must execute independently from other test methods. This means that a test method must never rely on the possible side effects of other test methods. Indeed, the setup and teardown phases are meant to make each test independent from the other. Moreover, JUnit does not guarantee 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.

## 3.2 A first example

First, let's create a new Java project in Eclipse, following the procedure we saw in Chapter [Eclipse](#). Let's call this project `testing.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
3 public class BankAccount {
4
5     private int id;
6     private double balance = 0;
7     private static int lastId = 0;
8
9     public BankAccount() {
10         this.id = ++lastId;
11     }
12
13     public int getId() {
14         return id;
15     }
16
17     public double getBalance() {
18         return balance;
19     }
20
21     public void deposit(double amount) {
```

```
22     if (amount < 0) {
23         throw new IllegalArgumentException("Negative amount: " + amount);
24     }
25     balance += amount;
26 }
27
28 public void withdraw(double amount) {
29     if (amount < 0) {
30         throw new IllegalArgumentException("Negative amount: " + amount);
31     }
32     if (balance - amount < 0) {
33         throw new IllegalArgumentException(
34             ("Cannot withdraw " + amount + " from " + balance));
35     }
36     balance -= amount;
37 }
38 }
```

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

Let's write a few JUnit tests to test this class' behavior.

Inside the project, let's 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**.

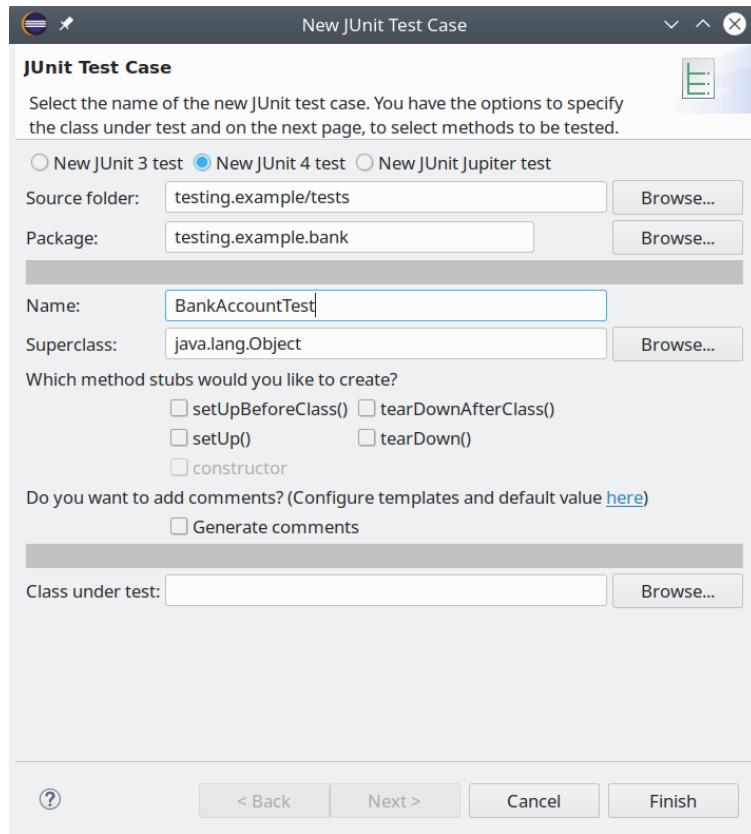


Tests should be kept separate from the production code, written in the source folder **src**. We will return to this in Section *Keeping test code separate from main code*.

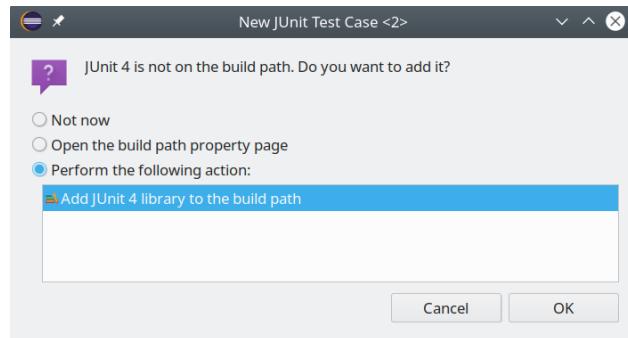
Create the test case **BankAccountTest**: Right-click on the tests folder, and select **New → JUnit Test Case**; in the dialog, use these settings:



Please make sure you select “New JUnit 4 test”: the default choice “New JUnit Jupiter test” would create a JUnit 5 test.



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 }
```



Another way of quickly creating a test case for an existing class is to navigate with the cursor to the class name and use the Quick Assist **Ctrl + 1** (see Chapter [Eclipse](#), Section [Quick Assist](#)). Then, select “Create new JUnit test case for...”. If the project has a source folder configured as a test folder (see the later Section [Keeping test code separate from main code](#)), then the test case will be created by default in the test source folder.

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 assertion methods of JUnit without using the class prefix `Assert` (alternatively, you can set the “Favorites” as shown in Chapter [Eclipse](#), Section [Content Assist](#)). The `fail` method is one such method: it explicitly makes the test fail.

Let’s run this test case 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 enabled (see Chapter [Eclipse](#), Section [Code Mining](#)).



This will run all the test methods of the current test case class. The order of the execution of the test methods is **not** predefined and can change across several executions.

The JUnit view should appear with the run test, which, as expected, is marked as a 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 }
```

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.



You can also run a single test method of a test case by clicking on the test method in the Java editor and using the context menu **Run As → JUnit Test**.

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

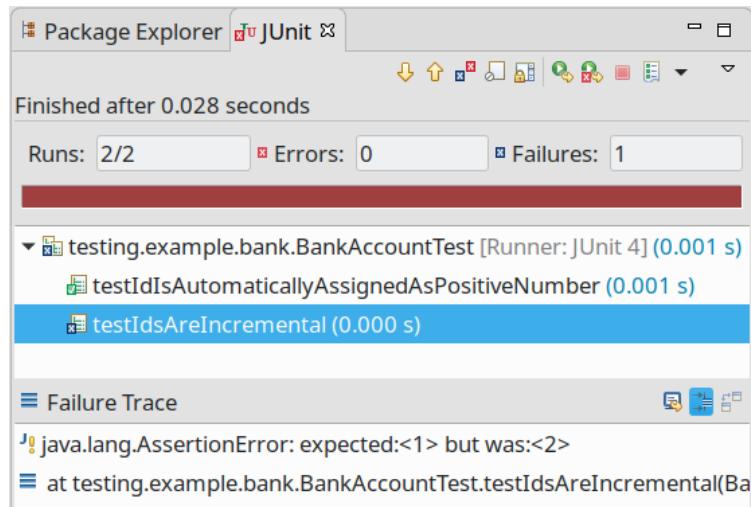
```
1 @Test
2 public void testIdsAreIncremental() {
3     BankAccount firstAccount = new BankAccount();
4     BankAccount secondAccount = new BankAccount();
5     assertTrue("Ids should be incremental",
6             firstAccount.getId() < secondAccount.getId());
7 }
```

The test passes.

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

```
1 // WRONG VERSION!
2 // Works only if this is the first executed test
3 @Test
4 public void testIdsAreIncremental() {
5     BankAccount firstAccount = new BankAccount();
6     BankAccount secondAccount = new BankAccount();
7     assertEquals(1, firstAccount.getId());
8     assertEquals(2, secondAccount.getId());
9 }
```

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.



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 changing 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:

```
1  @Test
2  public void testDepositWhenAmountIsCorrectShouldIncreaseBalance() {
3      // setup
4      BankAccount bankAccount = new BankAccount();
5      // exercise
6      bankAccount.deposit(10);
7      // verify
8      assertEquals(10, bankAccount.getBalance(), 0);
9  }
```

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, 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.

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



## Do you spot any problems in the above test?

We'll get back to that in a moment.

### 3.2.1 Exception testing

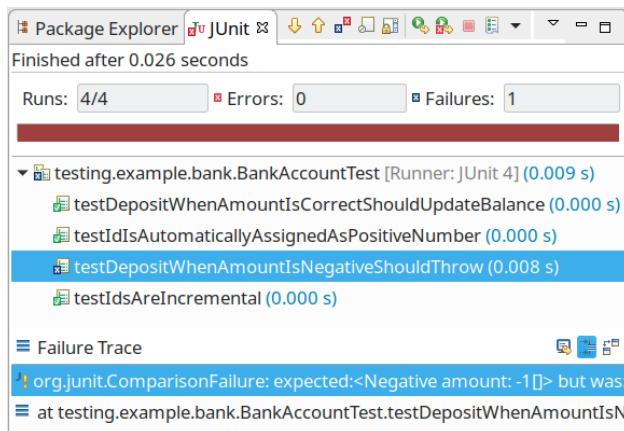
We write a test for the case when the passed `amount` is invalid. The idea is to perform the method call with a negative value inside a `try-catch` block; the caught exception is an `IllegalArgumentException` since that's the one that our SUT should throw in such a situation. Our test must fail if we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



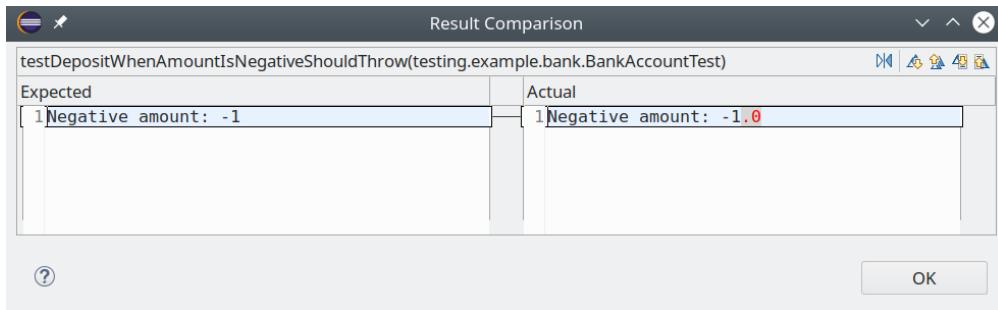
We said before that each test should test only one specific behavior of a method, but that does not necessarily mean that a test method should perform a single assertion. In this case, verifying the correct behavior of deposit in the presence of a negative input requires two assertions.

```
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 }
```

If you run this test, it fails! Look at the JUnit view after selecting the failed method, particularly the “Failure Trace” part. 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());
```

Note that the test for the exception is not that clean, and it might require some time to be understood. A better way to write tests with expected exceptions is to use JUnit's specific mechanisms (see the JUnit documentation, <https://github.com/junit-team/junit4/wiki/exception-testing>).

We can use the `expected` argument of the `@Test` annotation to specify the type of the expected exception: if an exception of the specified type is not thrown during the test, then the test will fail; otherwise, it will succeed:

```
1 @Test(expected = IllegalArgumentException.class)
2 public void testDepositWhenAmountIsNegativeShouldThrowWithExpected() {
3     BankAccount bankAccount = new BankAccount();
4     bankAccount.deposit(-1);
5 }
```

All tests are still green. However, this form of testing exceptions has a few drawbacks.

- First of all, you cannot specify any expectations on the exception's message (in our previous manual form of the test, we also verified the exception's message).
- Secondly, you cannot perform any further assertion after invoking the method that should throw the expected exception. In fact, we cannot verify that the `balance` is not modified in this new form.
- Even worse, the above test will pass if any statement in the test method throws an `IllegalArgumentException`. In this example, it's hard to get into trouble because of this (our test methods are concise); however, this mechanism is discouraged in more involved and longer tests.

For example, this test will succeed (the example is intentionally fictitious and for demonstration purposes only) even though our method invocation does not throw:

```

1 // False positive!
2 @Test(expected = IllegalArgumentException.class)
3 public void testDepositWhenAmountIsNegativeShouldThrowWithExpected() {
4     BankAccount bankAccount = new BankAccount();
5     System.getProperty(""); // throws IllegalArgumentException
6     bankAccount.deposit(10); // this does not throw
7 }

```

As a better alternative, we can use the `ExpectedException` rule. This rule allows the tester to indicate the expected exception and the expected message:

```

1 package testing.example.bank;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Rule;
6 import org.junit.Test;
7 import org.junit.rules.ExpectedException;
8
9 public class BankAccountTest {
10
11     @Rule
12     public ExpectedException thrown = ExpectedException.none();
13     ...
14     @Test
15     public void testDepositWhenAmountIsNegativeShouldThrowWithExpectedException() {
16         BankAccount bankAccount = new BankAccount();
17         thrown.expect(IllegalArgumentException.class);
18         thrown.expectMessage("Negative amount: -1.0");
19         bankAccount.deposit(-1);
20         // but we can't perform further assertions...
21     }
22 }

```



JUnit 4 rules allow flexible addition or redefinition of the behavior of each test method in a test class, <https://github.com/junit-team/junit4/wiki/rules>.

This solution is much more readable than the manual try-catch solution since it is declarative.

Moreover, the exception is expected to be thrown only after the rule has been configured (with `expect`); thus, there's no risk of false positives due to exceptions thrown in other parts of the test method. For example, this test will fail because another statement throws the exception before the setup of the rule:

```

1 // This will fail
2 @Test
3 public void testDepositWhenAmountIsNegativeShouldThrowWithExpectedException() {
4     BankAccount bankAccount = new BankAccount();
5     System.getProperty(""); // throws IllegalArgumentException
6     thrown.expect(IllegalArgumentException.class);
7     thrown.expectMessage("Negative amount: -1.0");
8     bankAccount.deposit(-1);
9 }
```

However, even this solution does not allow us to perform further assertions after the method under test has thrown an exception.

In case you really need to perform further assertions, besides verifying that an exception is thrown, and you find the “manual” solution too cumbersome, you can use the method `assertThrows` introduced in version 4.13. With this method, you assert that a given code block (specified as a lambda expression or method reference) throws a particular type of exception. The expected exception type is specified as the first parameter, and the code that should throw the exception is the second parameter. Of course, the test will fail if no exception of the specified type is thrown when executing the code. The method `assertThrows` returns the exception that was thrown. This allows you to perform assertions on the thrown exception. Finally, you can assert further after the code throws an exception. Here’s an example:

```

1 @Test
2 public void testDepositWhenAmountIsNegativeShouldThrowWithAssertThrows() {
3     BankAccount bankAccount = new BankAccount();
4     IllegalArgumentException e = assertThrows(IllegalArgumentException.class,
5         () -> bankAccount.deposit(-1));
6     // perform assertions on the thrown exception
7     assertEquals("Negative amount: -1.0", e.getMessage());
8     // and we can perform further assertions...
9     assertEquals(0, bankAccount.getBalance(), 0);
10 }
```



The method `ExpectedException.none()` has been deprecated in version 4.13 in favor of `assertThrows`.

### 3.2.2 Other tests

Now let’s write tests for the `withdraw` method. We first write the tests for the exceptions; for this method, we need to test two exception situations, following the techniques we have already seen:

```

1  @Test
2  public void testWithdrawWhenAmountIsNegativeShouldThrow() {
3      BankAccount bankAccount = new BankAccount();
4      IllegalArgumentException e = assertThrows(IllegalArgumentException.class,
5          () -> bankAccount.withdraw(-1));
6      assertEquals("Negative amount: -1.0", e.getMessage());
7      assertEquals(0, bankAccount.getBalance(), 0);
8  }
9
10 @Test
11 public void testWithdrawWhenBalanceIsUnsufficientShouldThrow() {
12     BankAccount bankAccount = new BankAccount();
13     IllegalArgumentException e = assertThrows(IllegalArgumentException.class,
14         () -> bankAccount.withdraw(10));
15     assertEquals("Cannot withdraw 10.0 from 0.0", e.getMessage());
16     assertEquals(0, bankAccount.getBalance(), 0);
17 }
```

The two tests succeed. Now we need to test the “happy” case. How can we recreate a situation when `withdraw` succeeds? Every `BankAccount` object is initialized with an empty balance. We need to *setup* the SUT object so that its balance is sufficient for performing the withdrawal. Since we also have the `deposit` method, it would be tempting to write such a test like that:

```

1 // Not a very clean test! DON'T DO THAT!
2 @Test
3 public void testWithdrawWhenBalanceIsSufficientShouldDecreaseBalance() {
4     // setup
5     BankAccount bankAccount = new BankAccount();
6     // use another method with logic for the setup
7     bankAccount.deposit(10);
8     // exercise
9     bankAccount.withdraw(3); // the method we want to test
10    // verify
11    assertEquals(7, bankAccount.getBalance(), 0);
12 }
```

There’s a problem with this test: it uses another method, not under test, of the SUT for performing *setup*. Thus, this test verifies the behavior of the method `withdraw` in terms of another method of the SUT, `deposit`. Indeed, we have already tested the method `deposit`, but relying on the correctness of another method of the SUT might have harmful consequences when such a method does have logic. For example, if in the future `deposit` is modified by introducing a bug, some tests about the `deposit` method will fail, and this test for `withdraw` will also fail, which is a bad thing! Indeed, it will take some time to understand whether the problem is in the `withdraw` or in `deposit`. (Of course, in this

example, it won't be that hard to spot the bug, but it might be more complicated in a more complex scenario.)

For example, break the implementation of `deposit` like that:

```

1 public void deposit(double amount) {
2     if (amount < 0) {
3         throw new IllegalArgumentException("Negative amount: " + amount);
4     }
5     // BUG: balance is not updated
6     // balance += amount;
7 }
```

Run all the tests, and `testDepositWhenAmountIsCorrectShouldIncreaseBalance` will fail (as it should), but also `testWithdrawWhenBalanceIsSufficientShouldDecreaseBalance` will fail, despite `withdraw` is correct!



Note that there's nothing wrong with using getters and setters for *setup* and *verify* when they have no logic. They might also contain bugs, but you will agree that writing incorrect getters and setters is rare. By the way, getters and setters should be generated automatically using IDE tools, like the ones provided by Eclipse, as we saw in Chapter *Eclipse*, Section *The Source menu*.

We should be able to set up the SUT by not using other methods of the SUT itself, except for constructors, getters, and setters.

Now, this would imply changing the interface of the SUT, and exposing additional modifiers, just for testing purposes. However, you can always resort to **package-private** members.

In this example, we could add a `setBalance` as a package-private method: this will be available only to classes of the same package. The test case we are writing is in the same package (we anticipated that when we started writing it), so it can use such a setter. This way, you will not clutter the class `BankAccount` with additional setters exposed to the world:

```

1 /**
2  * Package-private, for internal use only, for example, for testing.
3  * @param balance
4 */
5 void setBalance(double balance) {
6     this.balance = balance;
7 }
```

Now we modified the test as follows:

```
1  @Test
2  public void testWithdrawWhenBalanceIsSufficientShouldDecreaseBalance() {
3      // setup
4      BankAccount bankAccount = new BankAccount();
5      bankAccount.setBalance(10);
6      // exercise
7      bankAccount.withdraw(3); // the method we want to test
8      // verify
9      assertEquals(7, bankAccount.getBalance(), 0);
10 }
```

Even in the presence of the bug intentionally introduced in `deposit`, this test will still succeed.



## Code injection for package-private

Having package-private classes and members might expose your code to code injection from hostile programs: any external class can be defined in the same package of `BankAccount` and then have access to package-private members. To avoid this problem, if you release jar files of your code, you can *seal packages within a jar file*, as shown here: <https://docs.oracle.com/javase/tutorial/deployment/jar/sealman.html>.

As an alternative to the setter, you could add a constructor in `BankAccount` that takes the initial balance. Again, if you are unhappy with exposing such a constructor as a public constructor, make it package-private.



You may have realized that `BankAccount` is based on the invariant that the `balance` is never negative. Initially, this invariant is ensured by the initialization to 0. Suppose you expose a public constructor with the initial balance. In that case, you need to check that the initial balance is not negative and, in case, throw an exception (just like we do in `deposit` and `withdraw`) and test such a situation as well.

### 3.2.3 Revising

In the previous box *Do you spot any problem in the above test?*, we anticipated that also the first tests we wrote for `deposit` contained a problem. If you go back to the test `testDepositWhenAmountIsCorrectShouldIncreaseBalance`, you note that we are not effectively testing that the balance is increased: we only verify that the balance is the same as the deposited amount!

For example, introduce this bug:

```
1 public void deposit(double amount) {  
2     if (amount < 0) {  
3         throw new IllegalArgumentException("Negative amount: " + amount);  
4     }  
5     // BUG: simply assigned, not increased!  
6     balance = amount;  
7 }
```

And run all the tests; they are still green! That's one of the worst things that could happen: the tests we wrote do not effectively test the SUT behavior!

On the other hand, if we introduced the same bug in the `withdraw` method, the test method `testWithdrawWhenBalanceIsSufficientShouldDecreaseBalance` would fail; indeed, in that test, we are effectively testing the balance is decreased.

Since now we have the package-private setter (or the alternative constructor with the initial balance if we chose that way), we can rewrite the bad test so that it effectively tests the increment and decrement:

```
1 @Test  
2 public void testDepositWhenAmountIsCorrectShouldIncreaseBalance() {  
3     // setup  
4     BankAccount bankAccount = new BankAccount();  
5     bankAccount.setBalance(5);  
6     // exercise  
7     bankAccount.deposit(10);  
8     // verify  
9     assertEquals(15, bankAccount.getBalance(), 0);  
10 }
```

With the manually introduced bug in `deposit`, this test fails, as it should. Remove the bug, and everything is all green.



What if we are not allowed to modify `BankAccount`? Unfortunately, we will not be able to write good and effective tests. That is another consequence of not using TDD.



## Have we now tested everything correctly?

Now, you might be pretty sure that we effectively tested all the paths in the `BankAccount` code. However, something still escaped us (Hint: have we tested the corner cases?). However, do not worry; there are automatic tools that can help us have better confidence in our tests, as we will see in the following chapters [Code Coverage](#) and [Mutation Testing](#).

### 3.3 External collaborators (dependencies)

The previous example was relatively easy since the class `BankAccount` was self-contained: it was not using any other Java class but only primitive values.

Let us now consider a class, `Bank`, that relies on Java collections:

```

1 package testing.example.bank;
2 ...
3 public class Bank {
4
5     private List<BankAccount> bankAccounts = new ArrayList<>();
6
7     public int openNewBankAccount(double initialBalance) {
8         BankAccount newBankAccount = new BankAccount();
9         newBankAccount.setBalance(initialBalance);
10        bankAccounts.add(newBankAccount);
11        return newBankAccount.getId();
12    }
13
14    public void deposit(int bankAccountId, double amount) {
15        findBankAccountById(bankAccountId).deposit(amount);
16    }
17
18    public void withdraw(int bankAccountId, double amount) {
19        findBankAccountById(bankAccountId).withdraw(amount);
20    }
21
22    private BankAccount findBankAccountById(int bankAccountId) {
23        return bankAccounts.stream()
24            .filter(a -> a.getId() == bankAccountId)
25            .findFirst()
26            .orElseThrow(
27                () -> new NoSuchElementException
28                    ("No account found with id: " + bankAccountId));
29    }
30 }
```

Internally, this class keeps track of opened bank accounts and provides a means to deposit some amount on an account given its id (or throw a `NoSuchElementException` if no account with the given id is found). Similarly, for `withdraw`.

In the test case for this class, `BankTest`, which can be created with the procedure we have already used for `BankAccountTest`, we have a field holding the SUT object. We initialize it in a `@Before`

method since for testing Bank, we only need one instance of Bank, always initialized in the same way.

In the given shape, there's not much we can do to test this class appropriately:

```
1 package testing.example.bank;
2 ...
3 public class BankTest {
4
5     @Rule
6     public ExpectedException thrown = ExpectedException.none();
7
8     private Bank bank;
9
10    @Before
11    public void setup() {
12        bank = new Bank();
13    }
14
15    @Test
16    public void testOpenNewAccountShouldReturnAPositiveId() {
17        int newAccountId = bank.openNewBankAccount(0);
18        assertTrue("Unexpected non positive id: " + newAccountId, newAccountId > 0);
19    }
20
21    @Test
22    public void testDepositWhenAccountIsNotFoundShouldThrow() {
23        thrown.expect(NoSuchElementException.class);
24        thrown.expectMessage("No account found with id: 1");
25        bank.deposit(1, 10);
26    }
27
28    @Test
29    public void testDepositWhenAccountIsFoundShouldNotThrow() {
30        int newAccountId = bank.openNewBankAccount(10);
31        bank.deposit(newAccountId, 5);
32    }
33
34    @Test
35    public void testWithdrawWhenAccountIsNotFoundShouldThrow() {
36        thrown.expect(NoSuchElementException.class);
37        thrown.expectMessage("No account found with id: 1");
38        bank.withdraw(1, 10);
```

```
39     }
40
41     @Test
42     public void testWithdrawWhenAccountIsFoundShouldNotThrow() {
43         int newAccountId = bank.openNewBankAccount(10);
44         bank.withdraw(newAccountId, 5);
45     }
46 }
```

There are many problems with this test case, mainly because we are not effectively testing the internal behavior of `Bank` but only the external interface. However, unit tests should test the internal behavior of the SUT:

1. the first test does not make sure that the account is effectively added to the internal list of accounts;
2. the second test is OK even in the current form;
3. the third test only verifies that no exception is thrown but
  1. it performs setup by using another method of the SUT that is not meant to be tested by this test (just like in the previous example before adding `setBalance`);
  2. it does not verify that the balance of the given account is effectively incremented);
4. the last two tests have just the same problems.



The fact that we are not testing the case when the `amount` to deposit is negative is OK. It is not the responsibility of this class to check that the `amount` must not be negative: the `deposit` operation is performed by `BankAccount` and is already tested by `BankAccountTest`. The same holds for `withdraw` when the `amount` is negative, and the `balance` is insufficient.

Once again, the main problem is using another method (with logic) of the SUT for the setup phase. To solve the problem, we should keep in mind that for the setup and the verification of a single method of the SUT, we need access to the internal list of accounts. The list of accounts is indeed an implementation detail of `Bank`, but it is also true that a unit test must test the implementation behavior, not just the external interface.

Once again, we can solve the problem by adding a package-private getter in `Bank` giving access to the list of accounts:

```
1 /**
2  * Package-private, for internal use only, for example, for testing.
3 */
4 List<BankAccount> getBankAccounts() {
5     return bankAccounts;
6 }
```

Let's update our test case (note that we also change the name of the test methods to reflect that we effectively test the behavior of the methods):

```
1 package testing.example.bank;
2
3 import static org.junit.Assert.*;
4
5 import java.util.List;
6 import java.util.NoSuchElementException;
7
8 import org.junit.Before;
9 import org.junit.Rule;
10 import org.junit.Test;
11 import org.junit.rules.ExpectedException;
12
13 public class BankTest {
14
15     @Rule
16     public ExpectedException thrown = ExpectedException.none();
17
18     private Bank bank;
19
20     // the collaborator of Bank that we manually instrument and inspect
21     private List<BankAccount> bankAccounts;
22
23     @Before
24     public void setup() {
25         bank = new Bank();
26         bankAccounts = bank.getBankAccounts();
27     }
28
29     @Test
30     public void testOpenNewAccountShouldReturnAPositiveIdAndStoreTheAccount() {
31         int newAccountId = bank.openNewBankAccount(0);
32         assertTrue("Unexpected non positive id: " + newAccountId, newAccountId > 0);
33         assertEquals(newAccountId, bankAccounts.get(0).getId());
```

```
34     }
35
36     @Test
37     public void testDepositWhenAccountIsNotFoundShouldThrow() {
38         thrown.expect(NoSuchElementException.class);
39         thrown.expectMessage("No account found with id: 1");
40         bank.deposit(1, 10);
41     }
42
43     @Test
44     public void testDepositWhenAccountIsFoundShouldIncrementBalance() {
45         // setup
46         BankAccount testAccount = createTestAccount(10);
47         bankAccounts.add(testAccount);
48         // exercise
49         bank.deposit(testAccount.getId(), 5);
50         // verify
51         assertEquals(15, testAccount.getBalance(), 0);
52     }
53
54     @Test
55     public void testWithdrawWhenAccountIsNotFoundShouldThrow() {
56         thrown.expect(NoSuchElementException.class);
57         thrown.expectMessage("No account found with id: 1");
58         bank.withdraw(1, 10);
59     }
60
61     @Test
62     public void testWithdrawWhenAccountIsFoundShouldDecrementBalance() {
63         // setup
64         BankAccount testAccount = createTestAccount(10);
65         bankAccounts.add(testAccount);
66         // exercise
67         bank.withdraw(testAccount.getId(), 5);
68         // verify
69         assertEquals(5, testAccount.getBalance(), 0);
70     }
71
72     /**
73      * Utility method for creating a BankAccount for testing.
74      */
75     private BankAccount createTestAccount(double initialBalance) {
76         BankAccount bankAccount = new BankAccount();
```

```
77     bankAccount.setBalance(initialBalance);  
78     return bankAccount;  
79 }  
80 }
```

Again, each method is tested in isolation, independently from the other methods of the SUT, with the benefits we have already described.

The idea of this technique is simple:

- have a reference to the external collaborator(s) of the SUT in the test fixture
- use such a reference to
  - setup the specific test context
  - verify that the method under test modified the collaborator(s) as expected

We will use a similar technique in Chapter [Mocking](#).



We could have performed the addition to the `bankAccounts` list directly in the `createTestAccount` utility method. Still, for didactic purposes, we preferred explicitly showing the addition in the *setup* phase of single test methods.



The tests for `deposit` and `withdraw` when the account is expected to be found have a problem. Can you spot it? We will get back to that in Chapter [Code coverage](#), Section [How code coverage can help](#).

Since code quality is also one of this book's subjects, let's see whether we can improve the class `Bank` in that respect. We note that the type of the field `bankAccounts` is an interface, which is good since it abstracts from the actual implementation of the `List`. However, the field initialization is hard-coded in the class itself, which somehow vanishes the abstraction's benefits in the declaration type. This class has a hard-coded dependency on the concrete implementation type `ArrayList`. Of course, in this simple case, changing the implementation is just a matter of changing the initialization of the field. However, it is not possible to switch to another implementation of the list without modifying the source of the class. Indeed `Bank` is said to be **tightly-coupled** with `ArrayList`. Instead, classes should be **loosely-coupled** with other classes. In particular, a class should depend on other classes only through abstractions (interfaces or abstract classes) and avoid **direct** dependencies on implementation classes. This is known as the **Dependency Inversion Principle** ([Mar00](#)).

## Dependency Injection

Dependency Injection ([Fow04](#), [PP18](#)) is a design pattern that deals with the above issue. It is a pattern supporting the aforementioned “Dependency Inversion Principle”, making high-level components independent of the low-level component implementation details. It aims at removing hard-coded

dependencies. With this pattern, the implementations of dependencies are *injected* from outside so that the effective dependency resolution will take place at runtime, not compile-time. In a later chapter, Chapter [Learning tests](#), Section [Dependency Injection with Google Guice](#), we will see that there are frameworks specific for implementing dependency injection quickly. For the moment, we can implement this “manually”. A typical way of manually implementing “dependency injection” is to avoid initializing a dependency in the class itself and pass it from the outside at constructor time (or by calling a method, e.g., a setter).

Thus, we modify the class `Bank` by removing the initialization of the field `bankAccounts` and introducing a constructor that takes a value for that field. Since we are refactoring, we can also make the type of `bankAccounts` even more general: we are not using any specific from the class `List`, so we can turn it into a `Collection`:

```
1 import java.util.Collection;
2 import java.util.NoSuchElementException;
3
4 public class Bank {
5
6     private Collection<BankAccount> bankAccounts;
7
8     public Bank(Collection<BankAccount> bankAccounts) {
9         this.bankAccounts = bankAccounts;
10    }
11
12    // the package-private getBankAccounts can be removed
13
14    ... // as before
```

The class `Bank` is now completely independent from the implementation of `Collection` and can be seamlessly configured at run-time with any implementation of such an interface. We also get rid of the package-private `getBankAccounts` method, since, as we will see next, we do not need that anymore in the tests.

Of course, now the `BankTest` does not compile anymore, since we have to update the `setup` method: it is now the responsibility of the test to create an implementation for the dependency of `Bank` and pass it to the constructor. Note that in the test case, we still keep a reference to the created list so that we can still use it to set up the test and verify the behavior:

```
1 public class BankTest {  
2     ...  
3     @Before  
4     public void setup() {  
5         bankAccounts = new ArrayList<>();  
6         bank = new Bank(bankAccounts);  
7     }  
8     ...
```

Now the test case compiles again, and everything is still green.



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.

### 3.3.1 Alternative implementations

The aim of this chapter is to get started with JUnit (and, as we will see in the following sections, also with additional testing libraries); thus, we did not focus on optimizations of the design of the code under test.

For example, the class `BankAccount` might have to be used only by the class `Bank`, which could be the only class with which other external clients will interact. In such a situation, the class `BankAccount` itself could be made package-private (remember that `BankAccountTest` would still be able to access it since it is in the same package). After all, in the current implementation, `Bank` does not expose any further details about `BankAccount`: when opening a new account, you get only its identifier as a result. Of course, in this case, `Bank`'s constructor taking as argument a collection of `BankAccount` should be made package-private as well since external clients would have no access to `BankAccount` and would not be able to create such a collection; `Bank` should then provide another means for its initialization, or should be initialized somewhere else<sup>1</sup>.

Another alternative design implementation, especially if `BankAccount` is a package-private class, is to implement the responsibility of keeping the invariant of the non-negative balance directly in the `Bank` class (since it would be the only class that has direct access to `BankAccount`). In that case, `BankTest` should also verify such behavior. In the current implementation, `BankTest` does not have to verify such behavior, which is already tested in `BankAccountTest`.

In any case, you see that unit tests are tightly coupled with the internal implementation details of the SUT, as we had anticipated. Thus, changing the `Bank` and `BankAccount` classes as above would require changing their test cases accordingly.

---

<sup>1</sup>Dependency Injection frameworks also help in that respect.

On the contrary, if we had integration tests (e.g., UI tests or end-to-end tests) that verify the behavior of the bank application we are implementing, such tests usually would not be impacted by the above alternative implementations.

## 3.4 Testing private methods?

When starting writing unit tests, one typical question is “How can I test private methods?”. The answer is simple: “You don’t test private methods... directly”.

Private methods can only be tested indirectly via public methods (or protected or package-private methods). After all, if a method is private, it is meant not to be reachable from nothing but the defining class itself.

If you feel the need to test a private method directly because the private method has complex logic, which is not easily testable indirectly, then you have a design problem. Typically, it means that the class with the private method that is hard to test through the public interface violates the **Single responsibility principle** ([Mar00](#)). In this case, you may want to factor out the logic of the private method into a separate class, where the method is public and easily testable in isolation in a dedicated test case, and the previous class will delegate to the new class ([Voc18](#)). This will also improve the design of your code.

By the way, by using TDD, that is, writing the tests before the production code, you will never find yourself in a situation where you need to call a private method directly in your test. When using TDD, a private method appears in your SUT only on a refactoring phase by extracting code from other accessible methods; thus, your tests have already tested the code of the private method.

In any case, you should avoid “hacks” like calling private methods by reflection.

Even if we still haven’t applied TDD, in this example, we could extract a private method in `BankAccount` to avoid the duplicated code for the case of a negative amount:

```
1  public void deposit(double amount) {
2      if (amount < 0) {
3          throw new IllegalArgumentException("Negative amount: " + amount);
4      }
5      balance += amount;
6  }
7
8  public void withdraw(double amount) {
9      if (amount < 0) {
10          throw new IllegalArgumentException("Negative amount: " + amount);
11      }
12      if (balance - amount < 0) {
13          throw new IllegalArgumentException
14              ("Cannot withdraw " + amount + " from " + balance);
```

```
15     }
16     balance -= amount;
17 }
```

We can extract a new private method, say `handleNegativeAmount`: select the statements of the `if (amount < 0)...` and use the Eclipse refactoring **Extract Method...** (specifying to replace all the occurrences of the selected statements). The resulting code is:

```
1 public void deposit(double amount) {
2     handleNegativeAmount(amount);
3     balance += amount;
4 }
5
6 public void withdraw(double amount) {
7     handleNegativeAmount(amount);
8     if (balance - amount < 0) {
9         throw new IllegalArgumentException
10            ("Cannot withdraw " + amount + " from " + balance);
11    }
12    balance -= amount;
13 }
14
15 private void handleNegativeAmount(double amount) {
16     if (amount < 0) {
17         throw new IllegalArgumentException("Negative amount: " + amount);
18     }
19 }
```

The new private method is already tested by the tests we have written so far, indirectly through the public methods. Indeed, tests are still green after this refactoring.

In Chapter [TDD](#), Section [Testing abstract classes?](#), we will address an issue similar to this one related to testing abstract classes, which cannot be instantiated by their nature.

## 3.5 Keep your tests clean and readable

We have already said tests should be clean code as well, and in particular, they should be readable since they represent specifications and executable documentation.

For example, let's consider this test.

```

1  @Test
2  public void testWithdrawWhenAccountIsFoundShouldDecrementBalance() {
3      // setup
4      BankAccount testAccount = createTestAccount(10);
5      bankAccounts.add(testAccount);
6      // exercise
7      bank.withdraw(testAccount.getId(), 5);
8      // verify
9      assertEquals(5, testAccount.getBalance(), 0);
10 }

```

In this code, 10 and 5 are known as *Magic Numbers*<sup>2</sup>: using hardcoded constants may seem obvious when you're writing a piece of code, but they limit the future readability<sup>3</sup>.

That is why we should use constants with sensible names instead of magic numbers to make the code more readable. Moreover, in the above code, the 5 in the `assertEquals` might be misleading. It might not be immediately apparent whether we expect the resulting balance to be the same as the passed amount or to be the result of the subtraction of the passed amount from the initial balance (in this example, 10 and 5 have been intentionally used to create confusion; in general, constants in tests should be chosen with more care).

We use the Eclipse refactoring tools: select the 10 constant in the above test, and from the context menu, select **Refactor → Extract Constant**: in the appearing dialog, give a name to the constant, possibly following the convention that constant names should be all capitalized, e.g., `INITIAL_BALANCE` and select the checkbox to replace all occurrences of the selected expression with the constant (since these constants will be used only in this test case we can make them private). Do the same for 5, with the name `AMOUNT`. Finally, let's make the 5 in the `assertEquals` more explicit by replacing it with the expression `INITIAL_BALANCE-AMOUNT`:

```

1  private static final int AMOUNT = 5;
2  private static final int INITIAL_BALANCE = 10;
3  ...
4  @Test
5  public void testWithdrawWhenAccountIsFoundShouldDecrementBalance() {
6      // setup
7      BankAccount testAccount = createTestAccount(INITIAL_BALANCE);
8      bankAccounts.add(testAccount);
9      // exercise
10     bank.withdraw(testAccount.getId(), AMOUNT);
11     // verify
12     assertEquals(INITIAL_BALANCE-AMOUNT, testAccount.getBalance(), 0);
13 }

```

---

<sup>2</sup><https://refactoring.com/catalog/replaceMagicNumberWithSymbolicConstant.html>

<sup>3</sup>-1, 0, and 1 are usually not considered magic numbers.

This test is more readable and understandable.<sup>4</sup> You might also want to manually turn the `int` constants into `double` constants since we use them to perform comparisons with `double` values. The refactoring chose `int` since we did not write any decimal value in the original literals. If we had previously written `10.0`, `5.0`, and so on, the refactoring would have extracted them as `double` constants.



Apply the same technique to refactor all the tests we have seen so far by using constants and expressions to make the meaning of the tests more explicit.

### 3.5.1 Beware of code duplication removal in tests

We might be tempted to remove code duplication in tests, to keep them clean.

For example, we spot that these two test methods in `BankAccountTest` are rather similar:

```

1  @Test
2  public void testDepositWhenAmountIsNegativeShouldThrow() {
3      BankAccount bankAccount = new BankAccount();
4      IllegalArgumentException e = assertThrows(IllegalArgumentException.class,
5          () -> bankAccount.deposit(-1)); // difference
6      assertEquals("Negative amount: -1.0", e.getMessage());
7      assertEquals(0, bankAccount.getBalance(), 0);
8  }
9
10 @Test
11 public void testWithdrawWhenAmountIsNegativeShouldThrow() {
12     BankAccount bankAccount = new BankAccount();
13     IllegalArgumentException e = assertThrows(IllegalArgumentException.class,
14         () -> bankAccount.withdraw(-1)); // difference
15     assertEquals("Negative amount: -1.0", e.getMessage());
16     assertEquals(0, bankAccount.getBalance(), 0);
17 }
```

Indeed, the only difference is the code that is supposed to throw the exception, that is, the method invocation on the `bankAccount` object, `deposit` or `withdraw` (with the same argument `-1`). Thus, we could come up with a clever refactoring like the following, where the testing logic is in a reusable method, and the test methods only have to pass a method reference to the private method:

---

<sup>4</sup>In the literature, you might find alternative opinions saying this strategy is an anti-pattern. In my opinion, this version of the test is more readable than the previous one.

```
1  @Test
2  public void testDepositWhenAmountIsNegativeShouldThrow() {
3      assertActionWithNegativeAmount(BankAccount::deposit);
4  }
5
6  @Test
7  public void testWithdrawWhenAmountIsNegativeShouldThrow() {
8      assertActionWithNegativeAmount(BankAccount::withdraw);
9  }
10
11 private void assertActionWithNegativeAmount(ObjDoubleConsumer<BankAccount> code) {
12     BankAccount bankAccount = new BankAccount();
13     IllegalArgumentException e = assertThrows(IllegalArgumentException.class,
14         () -> code.accept(bankAccount, -1.0));
15     assertEquals("Negative amount: -1.0", e.getMessage());
16     assertEquals(0, bankAccount.getBalance(), 0);
17 }
```

Is this testing code more readable and more understandable than the original one? To understand what `testDepositWhenAmountIsNegativeShouldThrow` actually tests, we must look at the code of `assertActionWithNegativeAmount`, keeping in mind that the passed lambda (actually a method reference) is used inside another lambda in `assertThrows`. Moreover, the parameter `ObjDoubleConsumer<BankAccount> code` might not be immediate to grasp.

As we saw in Chapter [Testing](#), Section [Automated tests](#), Code duplication is admissible in tests if it makes them easy to read. On the contrary, tiny tests that rely on reused abstractions and utility methods, requiring a lot of code navigation, might be harmful. Tests should be straightforward to read and understand. For this reason, having lots of abstractions and code reuse in tests might quickly become an anti-pattern. Of course, this highly depends on the specific case: in some contexts, well-defined abstractions and reusable methods might make tests much more readable.

## 3.6 Other testing libraries

JUnit tests can also benefit from the API of other testing libraries, as we will see in the rest of this section.

### 3.6.1 Using Hamcrest matchers

JUnit 4 provides another form of assertion method, `assertThat`, which takes a value and a **matcher**. The matcher is defined by the Java Hamcrest library, <https://github.com/hamcrest/JavaHamcrest>, whose core part is shipped with JUnit (thus, it is already available in your project if you're using JUnit 4) <https://github.com/junit-team/junit4/wiki/matchers-and-assertthat>.

This form of assertion aims at being more readable and understandable, since you specify the actual value (*subject*), the comparison (*verb*), and then the expected value (*object*). On the contrary, with the standard JUnit assert methods, such as `assertEquals`, you specify the comparison, the expected value, and the actual value, which is counter-intuitive:

```
1 // verb, object, subject
2 assertEquals(expected, actual);
3 // subject, verb, object
4 assertThat(actual, equalTo(expected));
```

In the above snippet, `equalTo` is a static method of the Hamcrest library's `CoreMatchers` class that returns the matcher performing the comparison with `equals`. To avoid prefixing such static methods with the class name, just like we do for JUnit 4 `Assert` static methods, it is best to use such a static import in the test case:

```
1 import static org.hamcrest.CoreMatchers.*;
```

The static method `not` returns a matcher for negating a verb. For example, these two assertions are equivalent:

```
1 // JUnit style
2 assertEquals(expected, actual);
3 // with Hamcrest
4 assertThat(actual, not(equalTo(expected)));
```

Several matchers can also be composed with `allOf` and `anyOf`. Other compositions can be achieved with, for example, `either(...).or(...)`, whose meaning should be straightforward.

Failure messages are usually also clearer when using Hamcrest matchers. However, keep in mind that when using `assertThat(..., equalTo(...))` for comparing strings, in case of failure, the dialog highlighting the differences is unavailable from the JUnit view.

JUnit 4 only contains the core matchers from Hamcrest; other Hamcrest matchers, like those specific for collections, must be added explicitly to your project (they are available as separate jars).

While this style is a step forward concerning the readability of test assertions, we think that **AssertJ** does a better job in that respect, so we will not go into deeper details of Hamcrest.

### 3.6.2 Using AssertJ

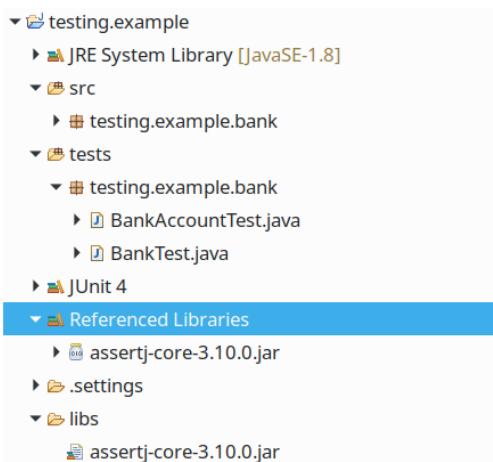
AssertJ is not part of the Eclipse distribution, and for the moment we need to download the jar of AssertJ manually, add it to our project, and configure the classpath accordingly.



The following procedure looks cumbersome and requires too many manual steps. Later in the book, in Chapter [Maven](#), we will learn how to add external libraries to our projects in a snap without even downloading anything manually.

1. Download the jar from this link:  
<https://search.maven.org/remotecontent?filepath=org/assertj/assertj-core/3.10.0/assertj-core-3.10.0.jar>
2. Create a folder in the project, for example, `libs`, to store the jars of external libraries: right-click on the project and **New → Folder** (note that it must NOT be a source folder);
3. Copy the downloaded jar in that folder (e.g., drag and drop from your file explorer into that folder in Eclipse);
4. Add the jar to the classpath: right-click on the jar in Eclipse and **Build Path → Add to Build Path**.

The layout of the Eclipse project should now look like this:



The layout of the project after adding the AssertJ core JAR to the build path

Note that the “Referenced Libraries” node can be expanded to inspect the jars’ packages in your classpath.

Now we can start using AssertJ assertions in our tests; once again, it is better to use a static import (unless you set the “Favorites” as shown in Chapter [Eclipse](#), Section [Content Assist](#)):

```
1 import static org.assertj.core.api.Assertions.*;
```

Also, AssertJ assertions are based on a method called `assertThat`, but AssertJ follows a fluent style ([Fow05](#)) based on **method chaining**: you provide a single argument (the subject to perform assertions on) and then call methods on the returned value. The object returned by `assertThat` provides methods for asserting properties on the passed subject, which depend on the static type of the subject. For example, if the subject is a collection, you can call assertions specific to collections.

Such methods also improve the user experience: since the value returned by `assertThat` depends on the static type of the passed subject, the content assist will be able to provide all completions that are valid for that specific subject of the assertions. Just type `assertThat(subject)`. and use the content assist after the `.` to quickly find a method that fits your assertions.

Note the differences among the assertion styles:

```

1 // JUnit 4
2 assertEquals(expected, actual);
3 // Hamcrest, more readable
4 assertThat(actual, equalTo(expected));
5 // AssertJ, even more readable: reads like a sentence
6 assertThat(actual).isEqualTo(expected);

```

The assertions will be even more readable than with Hamcrest matchers. For example, compare the two equivalent assertions (based on our previous tests):

```

1 int newAccountId = bank.openNewBankAccount(0);
2 // with JUnit
3 assertTrue("Unexpected non positive id: " + newAccountId, newAccountId > 0);
4 // with AssertJ
5 assertThat(newAccountId).isGreaterThan(0);
6 assertThat(newAccountId).isPositive(); // even better!

```

Note that, differently from `assertTrue`, which by default prints a generic and useless failure message and requires the tester to manually provide a meaningful failure message, AssertJ can print useful failure messages since the assertions are performed by using predefined methods for most common comparison operations.

When dealing with collections, it is easy to perform assertions with AssertJ, and the test is much more readable than with standard JUnit assertions. In particular, several assertions can be performed on the same subject, thanks to the fluent nature of AssertJ:

```

1 @Test
2 public void testOpenNewAccountShouldReturnAPositiveIdAndStoreTheAccount() {
3     int newAccountId = bank.openNewBankAccount(0);
4     // standard JUnit assertions
5     assertTrue("Unexpected non positive id: " + newAccountId, newAccountId > 0);
6     assertEquals(newAccountId, bankAccounts.get(0).getId());
7     // better and more readable assertions with AssertJ
8     assertThat(newAccountId).isGreaterThan(0);
9     assertThat(bankAccounts).// several assertions on the same subject
10    hasSize(1).

```

```

11     extracting(BankAccount::getId)
12         .contains(newAccountId);
13 }
```

In the source code of the example, you can find the project `testing.evolved.example` that contains the same `Bank` and `BankAccount` classes and all the tests have been ported to `AssertJ`. In the following, we highlight some other `AssertJ` excellent features for writing readable tests.

Besides comparing values with `isEqualTo` and `isGreaterThan`, we can directly use methods such as `isZero`, `isNotNegative`, `isPositive`, etc.

```

1 @Test
2 public void testIdIsAutomaticallyAssignedAsPositiveNumber() {
3     // setup
4     BankAccount bankAccount = new BankAccount();
5     // verify
6     assertThat(bankAccount.getId()).isPositive();
7 }
```

Indeed, `AssertJ` provides so many assertions methods specific to common types. In general, using dedicated ones will provide a better error message and make the tests more readable. For example, as shown above, when the argument of `assertThat` is an integer, `isPositive` should be preferred to `isGreaterThan(0)`.<sup>5</sup>

Also, exception testing is easier with `AssertJ`; compare this version with the ones we had already implemented with standard JUnit exception testing mechanisms (Section [Exception testing](#)):

```

1 @Test
2 public void testDepositWhenAmountIsNegativeShouldThrow() {
3     BankAccount bankAccount = new BankAccount();
4     assertThatThrownBy(() -> bankAccount.deposit(-1))
5         .isInstanceOf(IllegalArgumentException.class)
6         .hasMessage("Negative amount: -1.0");
7     // further assertions after the exception is thrown
8     assertThat(bankAccount.getBalance()).isZero();
9 }
```

Note that with `AssertJ` exception testing mechanisms, we have readability and can also perform further assertions after the exception has been thrown (and verified). The fluent style based on the method chaining of `AssertJ` does not even require us to save the thrown exception to perform assertions (as we did instead with JUnit `assertThrows` in Section [Exception testing](#))

---

<sup>5</sup>In Chapter [Code Quality](#), Section [Analysis of test code](#), we will see that the code quality tool that we will use can spot the non-optimal choice of an assertion method, such as using `isGreaterThan(0)` instead of `isPositive`.

Moreover, asserting the equality of double values does not require an explicit `delta` as with JUnit `assertEquals`, though it is still possible to perform an assertion specifying a delta (once again, in a readable way):

```
1  @Test
2  public void testDepositWhenAmountIsCorrectShouldIncreaseBalance() {
3      BankAccount bankAccount = new BankAccount();
4      bankAccount.setBalance(5);
5      bankAccount.deposit(10);
6      assertThat(bankAccount.getBalance()).isEqualTo(15);
7      // or with offset
8      assertThat(bankAccount.getBalance())
9          .isCloseTo(14.9, byLessThan(0.1));
10 }
```

## 3.7 Keeping test code separate from main code

From the beginning, we kept the test code separate from the main code by writing test code in a separate source folder, `tests`. We do not want to include tests when we package our code in a jar file and deploy it for others to consume it.

However, we should also keep the dependencies that are needed by our tests separate from those that are also needed by our main code. Test code needs the dependencies of the main code, but not the other way around.

In an Eclipse project, by default, the classpath is **flat**, thus all the libraries of the project are available in all the source folders of the project. This means that in the main code (e.g., `Bank`), you can access test dependencies like JUnit and AssertJ (since we added the AssertJ jar to our project's classpath in the previous section), which is a bad thing.

Moreover, if you export your code as a jar, besides excluding tests, you should also avoid packaging dependencies that are only needed by tests. In contrast, it makes sense to package the dependencies of the main code.

We will use the `testing.evolved.example` project from now on.

Starting from Eclipse *Photon*, in Java projects, source folders can have separate classpaths based on whether they contain main or test code. To activate this mechanism, you must specify a different output folder for source folders containing test code<sup>6</sup>. (Recall that by default, Eclipse Java projects have `bin` as the default output folder where Java class files are generated):

Open the **Java Build Path** property of the project, and in the **Source** tab, select the `tests` source folder:

---

<sup>6</sup>Specifying different output folders for source folders is also possible when creating a Java project with the Eclipse wizard.

- check the box “Allow output folders for source folders”;
- double-click the node “Output folder” of the `tests` source folder and in the dialog that appears:
  - select “Specific output folder”;
  - specify a new output folder, for example, `test-bin`;
  - close the dialog with “OK”;
- select the node “Contains test sources” of the `tests` source folder and press “Toggle” (so that it contains “Yes”);
- press “Apply and Close”.

Now the `tests` source folder will have a darker icon in the “Package Explorer”. This means that Eclipse now knows that the `tests` source folder contains only test code.

Now we make sure that JUnit and AssertJ are to be considered dependencies for test code only: Open the **Java Build Path** property of the project, and in the **Libraries** tab, toggle the node “Visible only for test sources” both of the “JUnit 4” node and the AssertJ JAR. Press “Apply and Close”. Now, also the “JUnit 4” node and the AssertJ JAR in the “Referenced Libraries” of the “Package Explorer” will have darker icons to indicate that they are dependencies for test code only.

The project still compiles, but if you now try to use a JUnit class or an AssertJ class in a Java file of the `src` source folder, you will get a compilation error: those libraries are not accessible from the main code.

Let’s add another dependency to our project, which will be used both by the main and test code: **Log4j**, one of the most used Java libraries for logging events.



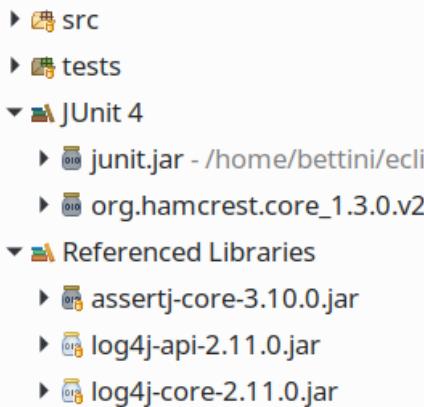
Once again, we do this procedure manually, but later, in Chapter [Maven](#), we will learn how to add external libraries to our projects in a snap, without even downloading anything manually.

1. Download the archive with binaries from this link  
<https://logging.apache.org/log4j/2.x/download.html>  
(choose either the `.zip` or the `.tar.gz` version as you prefer);
2. Extract the archive somewhere in your file system;
3. Copy the `log4j-api` and the `log4j-core` jar files (the files also contain the version number in the name) into the `libs` folder of the project that we have previously created;
4. Add the two jars from Eclipse to the “Build Path”, just like we did for the AssertJ jar file.<sup>7</sup>

This is a dependency of the main code so that it can be used both by the main code and the test code.

---

<sup>7</sup>Since we added a source folder specific for tests, the “Add to Build Path” context menu is available in a variant “Add to Build Path for Test Sources”.



Different colors for icons of main and test code and their dependencies.

Events will be logged to the console with the specified pattern (which will include the timestamp, the Java type, etc.). Log4j expects to find the file `log4j2.xml` in the classpath. Thus, we create such a file in the `src` folder.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <Configuration status="WARN">
3   <Appenders>
4     <Console name="ConsoleAppender" target="SYSTEM_OUT">
5       <PatternLayout
6         pattern="%d [%t] %-5level %logger{36} - %msg%n%throwable" />
7     </Console>
8   </Appenders>
9   <Loggers>
10    <Root level="INFO">
11      <AppenderRef ref="ConsoleAppender" />
12    </Root>
13  </Loggers>
14 </Configuration>
```

We can start using logging in our main code; typically, you get a logger from Log4j using the method `LogManager.getLogger(your class)`, and you store it into a private static constant in the class, for example:

Summarizing, the layout of the project in Eclipse should look like this (note the darker icons for tests, which is different from `src`, and for test dependencies, while Log4j jars have bright icons since they are main dependencies):

Log4j requires a configuration file stating what level of logging to perform and how to perform logging. A full description of Log4j is out of the scope of this book, and we refer to Log4j configuration documentation, <https://logging.apache.org/log4j/2.x/manual/configuration.html>. The following is a possible configuration for logging all log events with level INFO and the levels more specific (e.g., WARN and ERROR), thus excluding less specific levels (e.g., DEBUG and TRACE).

```
1 import org.apache.logging.log4j.LogManager;
2 import org.apache.logging.log4j.Logger;
3
4 public class Bank {
5
6     private static final Logger LOGGER = LogManager.getLogger(Bank.class);
7 }
```

Then you can log information using the logger's methods, according to the desired logging level, for example:

```
1 public int openNewBankAccount(double initialBalance) {
2 ...
3     LOGGER.info("New account opened with id: " + newBankAccount.getId());
4     return newBankAccount.getId();
5 }
6
7 public void deposit(int bankAccountId, double amount) {
8     findBankAccountById(bankAccountId).deposit(amount);
9     LOGGER.debug(String.format("Success: deposit(%d, %.2f)", bankAccountId, amount));
10 }
```

With the above configuration, the message logged with debug will not be appended to the console.

It is good practice to avoid creating strings for logging events when the level of logging is known to be disabled by default, like the debug case above, since creating strings increases memory consumption and has an overhead.

This could be a possible solution to create the string only when it is known that the event will be logged:

```
1 if (LOGGER.isDebugEnabled())
2     LOGGER.debug(String.format("Success: deposit(%d, %.2f)", bankAccountId, amount));
```

However, this is error-prone and could lead to uncovered lines (Chapter [Code Coverage](#)). A better solution is to use the logging methods that take a lambda acting as the supplier of the object to log instead of directly taking the object to log. This way, the string will be effectively created only when the supplier is used, and the check concerning whether the debug level is enabled will be performed directly by the debug method:

```
1 public void deposit(int bankAccountId, double amount) {
2     findBankAccountById(bankAccountId).deposit(amount);
3     LOGGER.debug(
4         () -> String.format("Success: deposit(%d, %.2f)", bankAccountId, amount));
5 }
6
7 public void withdraw(int bankAccountId, double amount) {
8     findBankAccountById(bankAccountId).withdraw(amount);
9     LOGGER.debug(
10        () -> String.format("Success: withdraw(%d, %.2f)", bankAccountId, amount));
11 }
```

If we run the `BankTest`, we should see in the console the logged event about the opening of a new bank account of the shape:

```
1 ...time... [main] INFO testing.example.bank.Bank - New account opened...
```

Recall that since Log4j is a dependency of the main code, it can also be used directly in the test code. After all, there are no good reasons for having dependencies available only to the main code and not to the test code. Not to mention that, in any case, the test code runs the main code; thus, the main code's dependencies must be available when running the tests.

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. Eclipse automatically copies any data file (that is, files that are not Java sources) into the output folder of the source folder. Thus, the file `log4j2.xml` will be copied into the `bin` folder and found at runtime by the classloader.

The classpath of the test code will include both `test-bin` (the output folder that we configured for the source folder `tests`) and `bin`. Thus, when running tests, Log4j will work as usual since the file `log4j2.xml` will be found. However, the output folder for test code, `test-bin`, will have precedence during class loading. This means we can have a slightly different `log4j2.xml` file in the root of the `tests` source folder. Eclipse will copy this into `test-bin` and be used when running tests. This is a standard technique in automated tests for having different resource files when running tests. Since configuration and property files are also resources, creating different or simplified environments for tests that do not require the final production environment is more manageable.

For example, we create in the `tests` folder this slightly different `log4j2.xml` file that sets the root level of logging to `DEBUG`:

```
1 ...
2 <Loggers>
3   <Root level="DEBUG">
4     <AppenderRef ref="ConsoleAppender" />
5   </Root>
6 </Loggers>
7 ...
```

If we run the `BankTest` again, we will also see the events logged with the `DEBUG` level.



Eclipse will issue a warning if a resource file with the same name is contained in different source folders with the same output folder; the output folder will contain only one resource file version. Thus, having a separate output folder for tests is crucial for dealing correctly with resources with the same name to be used in different runtime contexts.

Let's create a simple "main" class, in the `src` folder, like the following:

```
1 package testing.example.bank.app;
2
3 import java.util.ArrayList;
4
5 import org.apache.logging.log4j.LogManager;
6 import org.apache.logging.log4j.Logger;
7
8 import testing.example.bank.Bank;
9
10 public class Main {
11
12     private static final Logger LOGGER = LogManager.getLogger(Main.class);
13
14     public static void main(String[] args) {
15         LOGGER.info("App started");
16         Bank bank = new Bank(new ArrayList<>());
17         int bankAccountId = bank.openNewBankAccount(10);
18         bank.deposit(bankAccountId, 20);
19         LOGGER.info("App terminated");
20     }
21 }
```

Let's run it from Eclipse and verify that the `DEBUG` level is NOT enabled, since the `log4j` configuration of the main code is being used. Remember that when running the main code (contained in the `src` folder), only the `bin` directory is part of the classpath.



Besides the cumbersome procedures described above for adding dependencies to your project, they also have the problem that if we want to inspect sources of dependencies, we must manually download the jars with sources and configure them appropriately in Eclipse. In Chapter [Maven](#), we will see how to solve this problem automatically.

### 3.7.1 Export a runnable JAR

Eclipse provides a tool for creating an executable JAR file, based on a “main” class and containing all the dependencies needed by the application. These dependencies must NOT contain test dependencies, as we said above. Of course, resource files will be packaged in the JAR, excluding resource files used by tests.

Since we configured test folders and test dependencies correctly in the previous section, Eclipse will be able to avoid packaging test classes and test dependencies in the final jar.

1. Select File → Export... → Java → Runnable JAR file
2. Select the run configuration we created before when running our main file
3. Provide an “Export destination”, i.e., the full path of the output jar file, e.g.,  
`somepath/bank-app.jar`
4. Select “Extract required libraries into the generated JAR.”
5. Press “Finish” (and press “OK” in the dialog that warns you that your jar will contain external libraries whose licenses should be reviewed).

The JAR will be created; it should be around 2 Mb since it also contains the classes of Log4j. Open the JAR with an archive manager to verify that Log4j is part of the JAR, but neither JUnit nor AssertJ is present. You can also run the jar to make sure that the main application can execute correctly.



The creation of such a JAR with dependencies, called **FatJar**, will be automated with Maven, Chapter [Maven](#), Section [Creating a FatJar](#).

## 3.8 JUnit 5

JUnit 5 is the new version of JUnit, which has been around for a few years. It provides many improvements and powerful features with respect to JUnit 4. JUnit 4 is still supported and can still be used in existing and new projects, and, indeed, in this book we primarily use JUnit 4. In this section, we describe some of the main features of JUnit 5, and in the book’s other chapters, we also sometimes show how these features can be used for the subject of those chapters. In particular, we focus on some exciting features of JUnit 5 that might help to keep tests readable and clean.

Being a new major release, JUnit 5 breaks the API of JUnit 4. If you want to use the new features of JUnit 5, you need to use its new API: existing JUnit 4 test cases must be rewritten. However, the good news is that JUnit 5 can still run JUnit 4 tests. Thus, you can have both JUnit 4 and JUnit 5 test cases in the same project if you want. On the other hand, if you plan to migrate your existing JUnit 4 test cases to JUnit 5, you can do that gradually. Finally, you could keep the existing JUnit 4 tests and use JUnit 5 for writing the new test cases.

Of course, assertion libraries like AssertJ (Section [Using AssertJ](#)) can be seamlessly used in JUnit 5. JUnit 5 has been designed with modularity in mind to create extensions to JUnit easily. In particular, it consists of 3 main parts (or sub-projects):

## JUnit Platform

This project provides an abstract platform for launching and executing tests; in particular, it defines the API for doing that. It thus defines an abstraction to implement *test engines*.

## JUnit Jupiter

This project implements such a *test engine*, for running tests according to the new JUnit 5 programming model. That is why, when speaking about writing JUnit 5 tests, one typically uses the term “Jupiter tests”.

## JUnit Vintage

This project provides an implementation of a test engine for running JUnit 4 (and JUnit 3) tests on the new JUnit Platform.

The procedure to create a new JUnit 5 test in a brand-new project is similar to the one shown in Section [A first example](#). This time, in the wizard “New JUnit Test Case,” we must select “New JUnit Jupiter test”. Eclipse will propose adding the “JUnit 5 library” to the build path this time.

The JUnit 5 library shipped with Eclipse already contains all the three sub-projects described above, including the “Vintage” project, so we can still write JUnit 4 tests even if we have the JUnit 5 library in the build path.<sup>8</sup>

The wizard creates this test case:

```
1 package ...;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 import org.junit.jupiter.api.Test;
6
7 class ExampleTest {
8
9     @Test
10    void test() {
```

<sup>8</sup>When using Maven, Chapter [Maven](#), Section [JUnit 5 and Maven](#), we have to add the “Vintage” dependency explicitly, in case we want to use both JUnit 4 and JUnit 5 in the same project.

```

11     fail("Not yet implemented");
12 }
13
14 }
```

If we compare this test case with the one of Section [A first example](#) we spot a few interesting differences:

- Assert methods are now defined in the class `org.junit.jupiter.api.Assertions`;
- Method annotations, such as `@Test`, are defined in the package `org.junit.jupiter.api`;
- Test classes and methods do not have to be `public`; indeed, it is recommended to use the *default package visibility* (that is, package-private), which improves the readability of code.

In the rest of this section, we show some new exciting features of JUnit 5 by adapting the tests we have written so far, using the same classes `Bank` and `BankAccount`. We also stress the differences between some mechanisms concerning JUnit 4.



When running tests from Eclipse, you need to make sure that the JUnit launch configuration specifies “JUnit 5” as the “Test runner”; otherwise, only JUnit 4 tests will be executed. This is required only if you had previously created that launch configuration for running JUnit 4 tests. Otherwise, Eclipse will use the JUnit 5 test runner automatically if it detects that at least one of the tests uses JUnit 5. Remember that the JUnit 5 test runner will be able to execute also JUnit 4 tests.

First, the annotations `@Before` and `@After` have been replaced by `@BeforeEach` and `@AfterEach`, respectively (in the package `org.junit.jupiter.api`), with the same semantics. Similarly, `@BeforeAll` and `@AfterAll` must be used instead of `@BeforeClass` and `@AfterClass`, respectively.

We have seen that JUnit 4 assert methods can also be passed a string message as the first argument. In JUnit 5, such a string message is passed as the last argument. Thus, the following JUnit 4 assertion:

```

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 }
```

becomes in JUnit 5

```

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

```

When we need to specify a more meaningful error message in case of assertion failure, in JUnit 5, we can pass a `Supplier<String>` instead of a string, that is, a lambda returning a string. For example,

```

1  @Test
2  void testIdsAreIncremental() {
3      BankAccount firstAccount = new BankAccount();
4      BankAccount secondAccount = new BankAccount();
5      assertTrue(firstAccount.getId() < secondAccount.getId(),
6          () ->
7              "Ids were expected to be incremental, but" +
8              firstAccount.getId() + " is not less than " +
9              secondAccount.getId());
10 }

```

The result will be the same, but since a lambda expression is only executed when needed (only when the assertion fails), if the creation of the message is expensive, with this solution, the string will only be created in case of failure.

Moreover, the lazy evaluation nature of lambdas is also valuable when the failure message makes sense only in case of an assertion failure. For example, we want to ensure that an object (e.g., a `BankAccount`) is `null` in a given situation. If it is not `null`, we want to show a failure message with some information computed by calling methods on such an object. We cannot do something like this:

```

1  assertNull(account,
2      "Instead of null it is an account with id: " + account.getId());

```

Since when the `account` object is `null`, as desired, the creation of the failure message will throw a `NullPointerException`. We should then do something more involved, which also makes the test less readable with distracting details, e.g.,

```
1 String failureMessage = "";
2 if (account != null)
3     failureMessage = "Instead of null it is an account with id: " + account.getId();
4 assertNull(account, failureMessage);
```

Instead, using a lambda expression, we can write:

```
1 assertNull(account,
2     () -> "Instead of null it is an account with id: " + account.getId());
```

The body of the lambda will be executed only in case of assertion failure, i.e., when the object is not null.

The @Test annotation no longer has parameters. For example, we cannot use expected anymore to specify that an exception is expected as we have done in the previous examples (Section [Exception testing](#)). We should use instead assertThrows, but remember that in JUnit 5 this method is defined in org.junit.jupiter.api.Assertions.

In Section [Exception testing](#) we have also seen an example of JUnit 4 org.junit.Rule. In JUnit 5, the mechanism of Rule is not present. In JUnit 4, rules were meant as an extension mechanism. JUnit 5 provides a new extension mechanism based on the @ExtendWith annotation. We will see an example of the use of such an extension mechanism in Chapter [Mocking](#), Section [Mockito and JUnit 5](#).

As said in Section [Structure of a test](#), a test method should test a single behavior of the SUT, but in some cases, several assertions must be written in the same test method to verify a single behavior. We have already seen such an example:

```
1 @Test
2 void testDepositWhenAmountIsNegativeShouldThrow() {
3     BankAccount bankAccount = new BankAccount();
4     IllegalArgumentException e = assertThrows(IllegalArgumentException.class,
5         () -> bankAccount.deposit(-1));
6     assertEquals("Negative amount: -1.0", e.getMessage());
7     assertEquals(0, bankAccount.getBalance(), 0);
8 }
```

In this test, we want to make sure that when we pass a negative argument to deposit:

- An IllegalArgumentException is thrown, and
- the exception has an expected message (showing the illegal value), and
- that the balance is not changed

In Section *Structure of a test*, we also stressed that JUnit aborts the execution of a test method as soon as an assertion fails. This also holds in JUnit 5. This might be unfavorable in some cases when we want to assert several conditions.

In the above example, it would make no sense to go on in case `assertThrows` fails. However, once the exception is thrown, we want to verify both assertions: they verify two conditions related to two different elements (the exception and the account object). If the exception message is wrong, AND the account balance is modified, we will only see the failure for the first assertion, knowing nothing about the other assertion. We will fix the first problem, run the test, and then realize that the other assertion also fails. JUnit 5 introduced the assertion method `assertAll`. This method takes a variable number of lambdas, which are meant to perform an assertion, and executes them all, even in case of failures in some of them. In case of failures, it reports details about each failed assertion.

Thus, in our case, we can write the above test as follows:

```

1  @Test
2  void testDepositWhenAmountIsNegativeShouldThrow() {
3      BankAccount bankAccount = new BankAccount();
4      IllegalArgumentException e = assertThrows(IllegalArgumentException.class,
5          () -> bankAccount.deposit(-1));
6      assertAll(
7          () -> assertEquals("Negative amount: -1.0", e.getMessage()),
8          () -> assertEquals(0, bankAccount.getBalance(), 0)
9      );
10 }
```

In JUnit 5, you can provide custom display names for test classes and methods using the annotation `@DisplayName`. Display names are helpful for test reporting, e.g., in the Eclipse JUnit view. The exciting feature of display names is that they are Java strings, so they are allowed to contain spaces, special characters, and even emojis, differently from names of classes and methods, which must respect strict Java rules.

For example, we can write:

```

1  @DisplayName("Tests for BankAccount")
2  class BankAccountTest {
3      ...
4      @Test
5      @DisplayName("A positive number is automatically assigned as id")
6      void testIdIsAutomaticallyAssignedAsPositiveNumber() { ... }
```

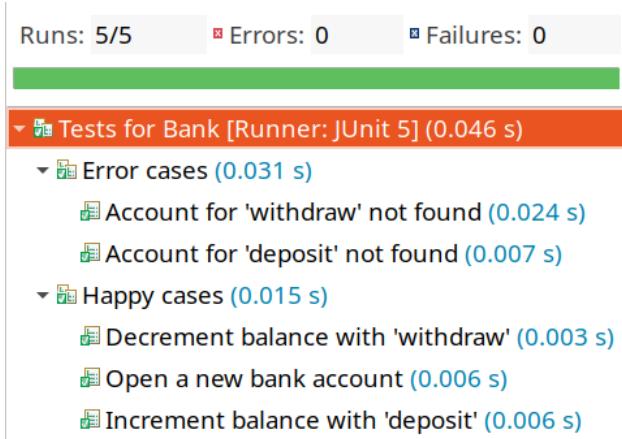
Another interesting new feature in JUnit 5 is the concept of **nested tests** allow the developer to group several groups of somehow related tests. The idea is to annotate a non-static inner class with `@Nested`. (There is no limit for the depth of the nesting, so you can have test classes nested in nested

test classes and so on.) Using `@Nested` combined with `@DisplayName` makes test reporting in IDEs even better.

For example, we can group tests related to “happy cases” and tests related to “exceptional cases” for our `Bank` class (see the complete code in the source code of the example — the body of the test methods is the same as the ones already seen before):

```
1  @DisplayName("Tests for Bank")
2  class BankNestedExampleTest {
3  ...
4      @BeforeEach
5      void setup() {...}
6
7      @Nested
8      @DisplayName("Happy cases")
9      class HappyCases {
10          @Test @DisplayName("Open a new bank account")
11          void testOpenNewAccountShouldReturnAPositiveIdAndStoreTheAccount() {...}
12
13          @Test @DisplayName("Increment balance with 'deposit'")
14          void testDepositWhenAccountIsFoundShouldIncrementBalance() {...}
15
16          @Test @DisplayName("Decrement balance with 'withdraw' ")
17          void testWithdrawWhenAccountIsFoundShouldDecrementBalance() {...}
18      }
19
20      @Nested
21      @DisplayName("Error cases")
22      class ExceptionalCases {
23          @Test @DisplayName("Account for 'deposit' not found")
24          void testDepositWhenAccountIsNotFoundShouldThrow() {...}
25
26          @Test @DisplayName("Account for 'withdraw' not found")
27          void testWithdrawWhenAccountIsNotFoundShouldThrow() {...}
28      }
29  ...
30 }
```

When run from Eclipse, the JUnit view will show the nested tests in a tree, which will make the grouping easy to spot:



The JUnit view with nested tests

Of course, you can use nested tests for proper grouping as you see fit.

In some of the following chapters, we will see other new features and mechanisms of JUnit 5.

# 4. TDD

We briefly described TDD in Chapter [Testing](#), Section [Test-Driven Development \(TDD\)](#). In this chapter, we go into details of the Test-Driven Development methodology ([Bec02](#)) and see a few examples in Java with JUnit.

## 4.1 Introduction to TDD

The central concept behind TDD is the so-called Red-Green-Refactor cycle. Moreover, we should follow some other techniques to use the TDD methodology correctly. We describe all these topics in this introductory section and the rest of the chapter.

### 4.1.1 The Red-Green-Refactor cycle

At the heart of TDD, there is the following iterative process, called the *Red-Green-Refactor* cycle:

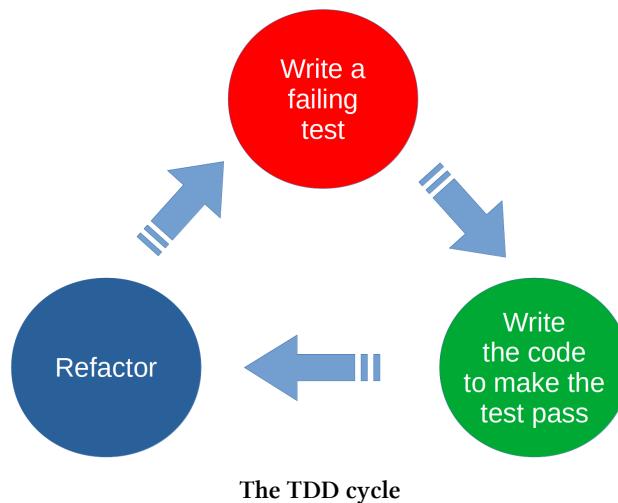
**Red** Write a test for a feature yet to be implemented and ensure it fails.

**Green**

Write only the smallest amount of code to make the test pass. Make sure that also existing tests still pass.

**Refactor**

If needed, refactor the code to clean it up and ensure all tests still pass.



We have already seen that the test must fail in the **red** state: since the test is written before the actual implementation, it is supposed to fail; otherwise, the test is a false positive. Recall that the test should fail for the expected reason; otherwise, the test will not be helpful.

Then, to pass the **green** state, we must write *only* the code to make the test pass. This means that we could also write it in a “dirty” way just to make the test pass. We’re even allowed to copy and paste code! However, we can only start a new cycle once we have refactored the code to make it clean. For example, suppose you copied and pasted code. In that case, this is the time to refactor the code, probably by introducing some common superclass or implementing some delegation after extracting the common code into another class. Of course, after refactoring, we must ensure we did not break anything; thus, we run all the tests. If all tests pass, then we can start a new cycle.

Note that in the **refactor** state, we can only refactor the code: we can change its structure without changing its behavior. This implies that we cannot add new functionality in this state. In TDD, you can add a new functionality only after you have a failing test for a new feature.

Thus, when in the red state, the code is expected not to work. When in the green state, the code works as expected but is not necessarily clean. Finally, since all the current features are tested, we have enough confidence to refactor and clean the code.

Note that while the red and the green states are mandatory, the refactor step is optional. Indeed, we could have written clean code right away. If not, we should immediately refactor the code before starting a new cycle. Delaying refactoring will probably make it harder to refactor at a later stage.

Once all the cycle steps are finished, we start a new cycle. The whole process might seem too long, but once you get familiar with TDD, you will see that you will only write a few lines of code at each step. Thus, a cycle will only last a few minutes, if not seconds. If that does not happen, then the scope of the test will probably be too broad; that is, you are trying to solve a problem that should be split into smaller sub-problems, each one to be solved in a different cycle.

Writing the test before writing the implementation code ensures that the code we write is implicitly testable and that every line of code we write gets tested since we write it after the test. Thus, we will be forced to write modular code since we must write testable code. With TDD, you will focus on requirements (i.e., the tests) before writing the code.

Moreover, as hinted above, you will be forced to focus on tiny problems instead of implementing the whole picture immediately. This should ensure a clean and clear code design, avoiding unnecessary features, and it is known as the **KISS** principle (**Keep It Simple, Stupid**), [https://en.wikipedia.org/wiki/KISS\\_principle](https://en.wikipedia.org/wiki/KISS_principle). This principle states that most systems work best if they are kept simple rather than made complicated. Using TDD correctly leads us to adhere to such a principle. After all, it is much easier to maintain a code with a simple design and implementation.

Thus, in general, it is crucial to be able first to understand how to decompose a problem into smaller sub-problems. The problem can also be implementing an entire application: you need to understand how this can be decomposed into several small components easily tested in isolation. Each component can then be implemented with TDD. As noted above, if a TDD cycle for implementing a single component functionality takes too long, it is better to split further that component or its features.

Writing unit tests afterward, i.e., after the code has already been written, might be hard, if not impossible. Moreover, you will write tests that only verify that your code works in a way you already know. This makes writing tests less sensible, and your tests will be biased since they will only tend to verify the current behavior of your code. On the contrary, writing tests first, before the code, will specify how your code is supposed to work, and you will write the code accordingly. As anticipated, it might be hard to write tests for existing code, which might have been written in such a way that it takes work to test. Writing such tests would first require refactoring the existing code. However, refactoring code without the safety net of tests is dangerous. Before refactoring, you need to have some tests for the code being refactored. And this generates a loop.

### 4.1.2 The three laws of TDD

With TDD, you write tests and production code at the same time. You switch from the test code to the production code continuously. The cadence of this switch between writing the test and writing the code to make it pass is well-described through the *The three laws of TDD*, codified by Robert C. Martin in <http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>. The three laws also describe the TDD process and force us to change only one thing at a time.

These are the three laws:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail, and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

The first law states that you can only write production code once you have a failing unit test. The second law tells you that as soon as a unit test fails, you should stop and write production code to make it pass. Note that the second law specifies that a test is considered to fail even if it does not compile. Indeed, when you start writing the test, the production code does not exist yet; for example, the test refers to a Java class that does not exist or calls a method that does not exist yet, etc. As soon as the test does not compile because of a missing class, you should stop writing the test and create the class to make it compile. Similarly, for a method that does not exist. When the test compiles, it should perform some assertions that are expected to fail since they refer to functionality not yet implemented in the production code. Thus you have to stop writing tests and implement the production code to make the test pass. Then, the third law tells you to write only enough code to make the failing test pass. No additional code should be added once the previously failing test succeeds. Note that this also includes the creation of a missing class or a missing method just to make the test compile. As soon as the test compiles because you added a method, stop writing production code and go back to writing the test.

These laws are meant to lead you into a development cycle that should be very short. During the cycle, you will have to switch between the test code and production code, probably several times.

For this reason, it is crucial to use an IDE, which allows you to switch quickly between the tests and the code. The laws will also force you to concentrate on a tiny and straightforward problem at a time (the one that makes the test fail). The IDE should also be used to quickly create a missing class or a missing method, e.g., by using a quick fix (see Chapter [Eclipse](#), Section [Quick Fix](#)).

Note that at each cycle, all tests pass. If something breaks on the next cycle, going back to a working state should be straightforward by just undoing what has been done in the current cycle. Moreover, by following the three laws, the thing that broke the code and must be undone will probably consist of a minimal number of lines of code.

It is likely to get things to work immediately by proceeding in small steps. Indeed, a symptom that you are correctly following this methodology is that you should not need to debug the code often. Vice versa, not following the TDD discipline would probably imply that you will try to solve bigger problems altogether, increasing the chances that you will fail and leading to long debugging sessions.

In the rest of this chapter, we see a few examples implemented with TDD. Besides the “Red-Green-Refactor” cycle and the three laws of TDD, we see a few strategies for performing the cycle, respecting the three laws. Such strategies are meant to guide us from one state to the next in the TDD cycle.

### 4.1.3 Remove duplication

The three laws described above are meant to put you on the right track when applying the TDD process of the red-green-refactor. However, there are still a few mechanisms and processes to consider.

Indeed, one might be tempted to use TDD to write test and production code like the following one:

```
1 // pseudo code
2
3 @Test
4 public void testFactorial() {
5     assertEquals(1, factorial(0));
6     assertEquals(1, factorial(1));
7     assertEquals(2, factorial(2));
8     assertEquals(6, factorial(3));
9     assertEquals(24, factorial(4));
10 }
11
12 public int factorial(int i) {
13     if (i == 0) return 1;
14     if (i == 1) return 1;
15     if (i == 2) return 2;
16     if (i == 3) return 6;
17     if (i == 4) return 24;
```

```
18     return -1;  
19 }
```

First, ending up with this code means considering the tests the final goal and writing production code with the only aim of making the tests pass. Of course, this makes no sense. Tests are a means to write code that works, not the other way around.

Moreover, if you end up with a production code like the one above, you did not apply any refactoring: you ignored the third step of the red-green-refactor cycle. Indeed, Kent Beck ([Bec02](#)) initially describes the refactor step as “Eliminate duplication”. Duplication must be eliminated through refactoring, not only in the production code. We must eliminate duplication between the test and the production code. In the code above, we left a lot of duplicated code: number literals for input and output appear both in tests and in the production code! Thus, refactoring and removing duplication implies making the production code more generic while writing subsequent tests. While we write tests, the tests get *more specific*. On the contrary, while we write production code to make the specific tests pass, we must make the production code *more generic*.

The last concept is at the heart of the next section, showing a few suggestions for performing the Red-Green-Refactor cycle.

## 4.2 The Transformation Priority Premise

In the article <https://blog.cleancoder.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html> Robert C. Martin stresses that to make the currently failing test pass, you must apply **transformations**, which are meant to change the code’s behavior (thus, they are the counterparts of refactoring, which must not change the behavior of the code). When applying transformations, we should follow this rule:

“As the tests get more specific, the code gets more generic.”

In particular, the article suggests that transformations have a preferred ordering (**priority**), aiming to implement the red-green-refactor cycle correctly. We already know that with TDD, we should always make the smallest change. However, the order in which we apply the changes is also essential. The priority tells us about the preferred ordering for applying transformations, which also implies an ordering for the tests to write (since we must first write a test before applying any transformation).

Here we report only a part of the transformations listed in the article mentioned above (the article admits that the presented transformations might be incomplete):

- (*{}*->*null*) no code at all->code that returns *null*
- (*null*->*constant*)
- (*constant*->*scalar*) replacing a constant with a variable or an argument
- (*statement*->*statements*)

- (*unconditional->if*)
- (*statement->recursion*)
- (*if->while*)
- (*expression->function*) replacing an expression with a function or algorithm

All these transformations turn the code's behavior from specific to something more generic. The transformations are also ordered by their complexity, from simple to complex.

The main idea described in the article is that simpler transformations should be preferred to more complex ones. In particular, more complex transformations should be applied only after applying the simpler ones. This also prescribes the order of the tests to write: we should first write failing tests that are expected to be satisfied by applying simpler transformations. When tests become more specific, we apply more complex transformations that turn the code into something more generic. So, when making a test pass, we try to do so with simpler transformations (higher in the list) than those that are more complex (lower in the list).

Summarizing, this ordering of transformations should put you on the right track for applying TDD correctly and proficiently. Moreover, the ordering should reduce the probability of getting to a point where one test causes you to rewrite an entire method (the so-called *impasse*).

### 4.2.1 A first example of TDD

In this first example, we aim at implementing a `leftTrim` method for strings, returning a new string with any leading whitespace removed.<sup>1</sup>



Such a method can be found in the standard Java libraries. We use it here as a simple running example to familiarize ourselves with TDD.

Since we'll use TDD from scratch, we'll write tests in the source folder `tests`, and we'll create the class and the method to implement in the source folder `src`, using the tools of Eclipse (shown in Chapter [Eclipse](#)).

Refer to Chapter [JUnit](#), Section [A first example](#), for the procedure for creating a JUnit test case in a Java project and for adding JUnit 4 to your build path.

---

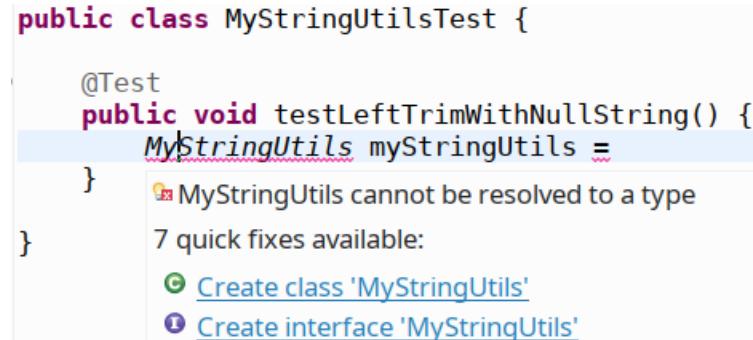
<sup>1</sup>The fact that the returned string is a new instance is implied by the Java `String` class' immutability, so that we won't test this aspect.

```

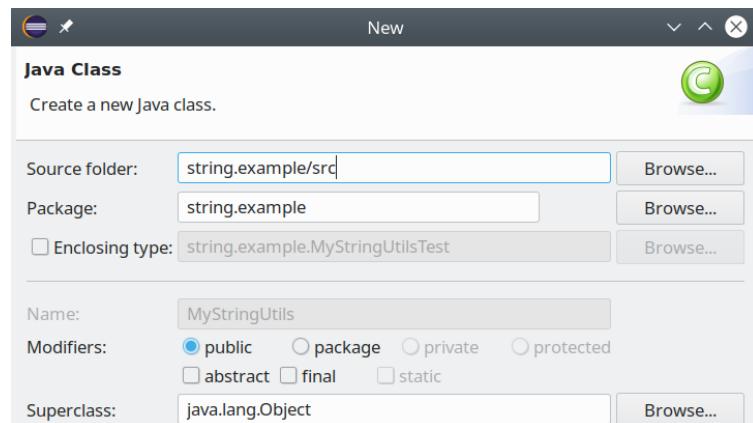
1 package string.example;
2 ...
3 public class MyStringUtilsTest {
4
5     @Test
6     public void testLeftTrimWithNullString() {
7         MyStringUtils myStringUtils =
8     }
9
10 }

```

The test does not compile: `MyStringUtils` does not exist yet. We stop writing the test and make it compile. Use the Eclipse quickfix (see Chapter [Eclipse](#), Section [Quick Fix](#)) to create the class (you can hover on the error, click on the marker in the editor's rules or, even better, use the shortcut **Ctrl + 1**):



Select the “Create class ‘`MyStringUtils`’”. The dialog for creating a new class will appear. The `MyStringUtils` class, which is our SUT, must be created in the source folder `src`, so make sure you change the “Source folder” in the dialog before pressing “Finish”:



Let’s go back to the test case class (you can use the Eclipse shortcut **Ctrl + F6** or **Alt + left/right** to navigate among the opened editors quickly). Let’s go on writing the test now that it compiles:

```

1  @Test
2  public void testLeftTrimWithNullString() {
3      MyStringUtils myStringUtils = new MyStringUtils();
4      String result = myStringUtils.leftTrim(null);
5  }

```

The test does not compile since there is no `leftTrim` method in the created `MyStringUtils` class. Again, we'll use the quickfix to create such a method.



Note that while writing the test, we're implicitly designing how our code should look like.

```

public String leftTrim(Object object) {
    // TODO Auto-generated method stub
    return null;
}

```

Navigate through parts of the automatically created method.

method is created in `MyStringUtils`, the editor is automatically opened, and several parts of the method, like its accessibility level, return type, parameter's type, and name, can be navigated using `Tab` and can be modified (refer to Chapter [Eclipse](#), Section [Editable proposals](#)). Thus, press `TAB` until you get to the parameter's type and change it from `Object` to `String`, press `TAB` again, and change the parameter's name to something more sensible, e.g., `input`.<sup>2</sup>

```

1 package string.example;
2
3 public class MyStringUtils {
4
5     public String leftTrim(String input) {
6         return null;
7     }
8
9 }

```



Remember that quickfixes that add members to an existing class, like the one we have just applied, do not automatically save the class.

---

<sup>2</sup>To help Eclipse understand our original intention, we can explicitly cast the `null` argument to `String` in the test method.

Save the file `MyStringUtils.java`, and the test now compiles. We applied the rule ( $\{f\} \rightarrow \text{null}$ ) (Section [The Transformation Priority Premise](#)). Indeed, Eclipse implemented the method by returning the default value for the return type, `null` in this case.

Let's go back to the test and write an assertion:

```
1  @Test
2  public void testLeftTrimWithNullString() {
3      MyStringUtils myStringUtils = new MyStringUtils();
4      String result = myStringUtils.leftTrim(null);
5      assertNull(result);
6 }
```

The test passes since the default implementation returning `null` satisfies our expectations.

In this first case, we wrote an assertion that already succeeds without modifying our production code. However, we wrote the production code (the method `leftTrim` with the default implementation) because the test failed (it did not compile). Of course, this is not a false positive, and we can be sure of that since the case is trivial. Moreover, we did not violate the first rule of TDD (Section [The three laws of TDD](#)): we wrote production code only to make a failing unit test pass. The body of the method `leftTrim` has been generated by Eclipse when creating the method through the quick fix. Eclipse generated the body with the default value for the return type (`null` in this case), which, in this specific case, is enough to make our assertion succeed right away. We could have also removed the generated return statement:

```
1  public String leftTrim(String input) {
2      // does not compile
3 }
```

The code would not compile, but the test compiles fine anyway. Eclipse allows you to run the test code even if the referred code does not compile. This would lead to a failure in the test at runtime since the executed code has compilation errors. So we would still be able to have a failure in the written assertion and then write the return statement to make the test succeed. However, this would only require us some more time without any benefit. As said above, what we did with the generated return statement is not a violation of the three laws.

The current version of `leftTrim` is far from complete, but it behaves correctly when the argument is `null`.

Let's refactor the test:

- Create a setup method annotated with `@Before`.
- Move the variable declaration `myStringUtils` and its initialization into the new method (use `Alt + Up Arrow` to move the statement into the method); the test does not compile.

- Convert the local variable into a class field with the Eclipse refactoring tool **Convert Local Variable to Field** specifying to initialize it in the current method. The field becomes our test fixture. The test now compiles again.
- Inline the local variable `result`, with the refactoring **Inline**.

```

1  public class MyStringUtilsTest {
2
3      private MyStringUtils myStringUtils;
4
5      @Before
6      public void setup() {
7          myStringUtils = new MyStringUtils();
8      }
9
10     @Test
11     public void testLeftTrimWithNullString() {
12         assertNull(myStringUtils.leftTrim(null));
13     }
14 }
```

The test still passes. Let's add another test for the empty string:

```

1  @Test
2  public void testLeftTrimWithEmptyString() {
3      assertEquals("", myStringUtils.leftTrim(""));
4 }
```

The test fails because our method always returns `null`. We can make this test pass with the following implementation, that is, by applying the rule (*null->constant*):

```

1  public String leftTrim(String input) {
2      return "";
3 }
```

The test passes, but the previous test now fails. Thus, this code is now too specific since it works only for the empty string and is not generic enough to handle the `null` case.

We could write an `if` statement, applying (*unconditional->if*). However, let's see if there's a simpler transformation rule to make the code generic enough to pass the two tests. We can apply (*constant->scalar*), returning the `input` parameter:

```

1 public String leftTrim(String input) {
2     return input;
3 }
```

Both tests pass, and the code is slightly more generic: it can handle both the `null` and the empty string cases.

It is crucial to understand that also this solution would make both tests pass:

```

1 public String leftTrim(String input) {
2     // too specific, not generic enough
3     if (input == null)
4         return null;
5     return "";
6 }
```

But this code would be too specific, violating the rule that the code should become more generic while the tests become more specific.

Now we could write a test with an input string that does not contain any leading white spaces expecting the same string as a result; we would also be sure that the new test would pass without touching the existing code. This could signify that we are not choosing the next test correctly: let's write a failing test that forces us to make the code more generic. The simplest failing test that we can write that also requires a minimal change to our code to make it more generic is with a string with one single leading space:

```

1 @Test
2 public void testLeftTrimWithOneLeadingSpace() {
3     assertEquals("abc", myStringUtils.leftTrim(" abc"));
4 }
```

The test fails because our method always returns the passed string. To make it pass without affecting the existing tests, we can add an `if` to handle the first two simple special cases (which can be seen as the base cases for the method we want to implement), applying the rule (*unconditional->if*):

```

1 public String leftTrim(String input) {
2     if (input == null || input.isEmpty())
3         return input;
4     return input.substring(1);
5 }
```

Note that an alternative could have been the one that proceeds in tiny steps:

```
1 public String leftTrim(String input) {  
2     if (input == null || input.isEmpty())  
3         return input;  
4     // duplication: refactor!  
5     return "abc";  
6 }
```

But once again, this would not be generic enough. Indeed, with this solution, the current TDD cycle would not be over: we should refactor to remove the duplication. In fact, our code and our test both contain the duplicate value "abc". So we would refactor to the version that invokes `substring`.

Note also that we wrote minimal code to make the test pass. Our code is more generic than the previous version, but it's still too specific: it works only for input strings with a single leading space (see the hardcoded `1` passed to `substring`).

What could be the next failing test that requires minimal generalization? It is now a good moment to test our method with a string that contains no leading space. We expect the following test to fail:

```
1 @Test  
2 public void testLeftTrimWithNoLeadingSpace() {  
3     assertEquals("abc", myStringUtils.leftTrim("abc"));  
4 }
```

Because the result would be "bc", since we always blindly skip the first character, even when it's not a space character. We expect to solve this with an `if` statement.



Note that an alternative failing test would be the one that passes a string with several leading spaces. To make such a test pass, we expect to use a loop. If we want to use the priorities of the transformations, we must avoid this path because adding an `if` is simpler than directly adding a loop. Of course, we know that at some point, the introduced `if` will have to be turned into a loop because, for generalizing our code, we'll have to use a more complex transformation later.

We make all tests pass with the following generalization of our code (applying the rule (*statement->statements*), since from a single `if` statement, we get to two `if` statements):

```

1 public String leftTrim(String input) {
2     if (input == null || input.isEmpty())
3         return input;
4     if (input.charAt(0) == ' ')
5         return input.substring(1);
6     return input;
7 }
```

Let's take some time to refactor the code to get rid of the standalone constants 0 and 1. Note that we are refactoring and are not changing the behavior. Thus, we do not apply transformations. Since we foresee loops, we use a variable that correlates those 0 and 1.

First, we can express 1 in terms of 0 with a trivial refactoring (which obviously does not break anything):

```

1 public String leftTrim(String input) {
2     if (input == null || input.isEmpty())
3         return input;
4     if (input.charAt(0) == ' ')
5         return input.substring(0 + 1); // express 1 in terms of 0
6     return input;
7 }
```

Then we refactor 0 to a local variable (**Refactor → Extract Local Variable...**, checking the box “Replace all occurrences...”); let's call that `beginIndex` since we plan to use it to specify the initial index of the substring:

```

1 public String leftTrim(String input) {
2     if (input == null || input.isEmpty())
3         return input;
4     int beginIndex = 0;
5     if (input.charAt(beginIndex) == ' ')
6         return input.substring(beginIndex + 1);
7     return input;
8 }
```

All tests are still green.

The code in the current form should tell us that we could make it even cleaner: we could always return the result of `substring` based on the value of `beginIndex`: if the `if` is not executed, then we return a substring starting from the very first character, that is, the original string, otherwise, we increment `beginIndex` in the `if` branch:

```

1 public String leftTrim(String input) {
2     if (input == null || input.isEmpty())
3         return input;
4     int beginIndex = 0;
5     if (input.charAt(beginIndex) == ' ')
6         beginIndex++;
7     return input.substring(beginIndex);
8 }
```

All tests are still green.

Are we moving to deal with several leading spaces, or could we write another failing test requiring minimal transformation? A minimal transformation could make the `if` condition more complex and generic. In our current implementation, we only remove a “space character,” but there are other white space characters, e.g., “tabulation character”. Indeed we expect this test to fail:

```

1 @Test
2 public void testLeftTrimWithOneLeadingTab() {
3     assertEquals("abc", myStringUtils.leftTrim("\tabc"));
4 }
```

We could add another condition to the `if` or replace the current `if` condition expression with an invocation to a utility function that detects whether a character is to be considered white space. We go for the latter and rely on the standard method `Character.isWhitespace`. This is the transformation (*expression->function*):

```

1 public String leftTrim(String input) {
2     if (input == null || input.isEmpty())
3         return input;
4     int beginIndex = 0;
5     if (Character.isWhitespace(input.charAt(beginIndex)))
6         beginIndex++;
7     return input.substring(beginIndex);
8 }
```

Now all tests pass.

Now we can concentrate on dealing with several leading white space characters, which can be handled with the transformation (*if->while*). At this point, it makes sense to use such a complex transformation since we have already used simpler ones. First, the failing test:

```

1  @Test
2  public void testLeftTrimWithSeveralSpaces() {
3      assertEquals("abc", myStringUtils.leftTrim(" abc"));
4  }

```

We can pass this test by transforming the `if` to `while`:

```

1  public String leftTrim(String input) {
2      if (input == null || input.isEmpty())
3          return input;
4      int beginIndex = 0;
5      while (Character.isWhitespace(input.charAt(beginIndex)))
6          beginIndex++;
7      return input.substring(beginIndex);
8  }

```

We previously cleaned our code by turning constants into variables and applied the transformations according to their priority. Thus, the transformation introducing a loop only consists of using `while` instead of `if`!

The next (failing) test is the one with an input consisting of white spaces only:

```

1  @Test
2  public void testLeftTrimWithAllSpaces() {
3      assertEquals("", myStringUtils.leftTrim(" "));
4  }

```

The test fails with a `StringIndexOutOfBoundsException` since we blindly scan the string for white characters without checking whether the string has been thoroughly scanned. The transformation adds this check to the `while`. Note that this could be seen as an instance of (*statement->statements*) though instead of adding a statement, we add a condition to an existing one:

```

1  public String leftTrim(String input) {
2      if (input == null || input.isEmpty())
3          return input;
4      int beginIndex = 0;
5      while (beginIndex < input.length() &&
6             Character.isWhitespace(input.charAt(beginIndex)))
7          beginIndex++;
8      return input.substring(beginIndex);
9  }

```

All tests pass.

Let's take some time to refactor the current code: let's save the length of the string into a local variable, again with the refactor "Extract Local Variable...":

```

1 public String leftTrim(String input) {
2     if (input == null || input.isEmpty())
3         return input;
4     int beginIndex = 0;
5     final int length = input.length();
6     while (beginIndex < length &&
7             Character.isWhitespace(input.charAt(beginIndex)))
8         beginIndex++;
9     return input.substring(beginIndex);
10 }

```

We should also realize that the initial check `input.isEmpty` is now useless since the rest of the code seamlessly handles the case when the string is empty (thanks to our final transformation). Let's remove it and verify that all tests still pass:

```

1 public String leftTrim(String input) {
2     if (input == null)
3         return input;
4     int beginIndex = 0;
5     final int length = input.length();
6     while (beginIndex < length &&
7             Character.isWhitespace(input.charAt(beginIndex)))
8         beginIndex++;
9     return input.substring(beginIndex);
10 }

```

This concludes our implementation.

All the tests we have written are helpful for our code's correctness, and none should be removed. Further tests would not (and should not) increase our confidence that the implemented method is correct; indeed, we tested all paths. Further tests we can add for other inputs and expected outputs are expected to succeed immediately. It makes sense to add further tests only if they are helpful for further documenting the method behavior.

For example, if we doubt it's clear that our method removes all kinds of white spaces, even when they are mixed, we can add another test for documenting the behavior better:

```

1 @Test
2 public void testLeftTrimWithSeveralKindsOfWhiteSpaces() {
3     // For documentation only: not used in TDD
4     assertEquals("abc", myStringUtils.leftTrim("\t abc"));
5 }

```

Again, since we are not adding production code without a failing test, we are not violating the first TDD law.

## 4.3 Three strategies for the green state

In this section, we see another example of TDD. In particular, we show a technique that can be used when applying TDD to derive a general solution to the problem we have to implement.

Kent Beck, in his book ([Bec02](#)), describes three strategies to get to the green state:

### Fake it

First, return a constant to make the test pass and gradually replace constants with variables.

### Use obvious implementation

Directly write the actual implementation.

### Triangulation

Generalize code when you have two examples or more.

We concentrate on the third strategy, called “triangulation,” by analogy to radar triangulation. The idea is to try to derive a general solution, which you still don’t know, once you have enough examples (experimental data).

We implicitly used these strategies in the previous example, but we will make them more explicit in the following example.

### 4.3.1 The factorial example with TDD

We want to implement a class that computes the factorial with TDD. Recall that the factorial of a natural number,  $n!$  is defined as follows:

- $0! = 1$
- $n! = n * (n - 1)!$  for  $n > 0$

So we would already know how to implement the factorial. However, let’s pretend that we don’t know the definition of the factorial, but we only know a few experimental data about its behavior:

- $\text{factorial}(0) = 1$
- $\text{factorial}(1) = 1$
- $\text{factorial}(2) = 2$
- $\text{factorial}(3) = 6$
- $\text{factorial}(4) = 24$
- $\text{factorial}(5) = 120$
- $\text{factorial}(\text{negative number}) = \text{undefined}$

We want to get to the general implementation of the factorial using these data (in the shape of tests) using triangulation.

Following the initial steps seen in the previous example, create a new Java project and

- Create a new JUnit 4 test;
- Start implementing a test and create the corresponding SUT Java class and method;
- Refactor the test till you get to this point:

```
1 package factorial.example;
2 ...
3 public class FactorialTest {
4
5     private Factorial factorial;
6
7     @Before
8     public void setup() {
9         factorial = new Factorial();
10    }
11
12    @Test
13    public void testBaseCase() {
14        assertEquals(1, factorial.compute(0));
15    }
16
17 }
```

Creating Factorial and its method compute with Eclipse tools, you should have this initial implementation:

```
1 package factorial.example;
2
3 public class Factorial {
4
5     public int compute(int n) {
6         return 0;
7     }
8
9 }
```

This test is expected to fail.

Now we must write in the Factorial class *only* the code to make this test succeed:

```
1 public int compute(int n) {  
2     return 1;  
3 }
```

Let's rerun the test, and this time the test succeeds, as we expected.

The following new test is expected to succeed with the current implementation:

```
1 @Test  
2 public void testFactorialOf1() {  
3     assertEquals(1, factorial.compute(1));  
4 }
```

The next test using our experimental data is expected to fail:

```
1 @Test  
2 public void testFactorialOf2() {  
3     assertEquals(2, factorial.compute(2));  
4 }
```

This requires us to generalize a bit our implementation:

```
1 public int compute(int n) {  
2     if (n < 2)  
3         return 1;  
4     return 2;  
5 }
```

This makes all the test pass, but there's a duplication that we can remove by refactoring (the result 2 is both in the test and in the code): we change the returned 2 to n, and the tests still pass:

```
1 public int compute(int n) {  
2     if (n < 2)  
3         return 1;  
4     return n;  
5 }
```

The next failing test is the following one:

```
1 @Test
2 public void testFactorialOf3() {
3     assertEquals(6, factorial.compute(3));
4 }
```

We already know the implementation of the factorial, and we can now write the “obvious implementation”. However, as said in the introduction to this example, let’s try to use “triangulation” with the existing tests. This is the code that we can write at the moment to make all the tests pass:

```
1 public int compute(int n) {
2     if (n < 2)
3         return 1;
4     return n * (n - 1);
5 }
```

This does not contain duplication with the test code; it is not too specific, yet not wholly generic. Moreover, we applied transformation rules preferring simple ones to more complex ones.

The next failing test should lead us to the final implementation:

```
1 @Test
2 public void testFactorialOf4() {
3     assertEquals(24, factorial.compute(4));
4 }
```

We could try to make this test pass with this code:

```
1 public int compute(int n) {
2     if (n < 2)
3         return 1;
4     return n * (n - 1) * (n - 2);
5 }
```

However, this breaks the test for the factorial of 2 (it returns 0 instead of 2).

Now we can spot the general pattern, and we use recursion:

```

1 public int compute(int n) {
2     if (n < 2)
3         return 1;
4     return n * compute(n - 1);
5 }
```

All tests pass.

We could write the test for 5, expecting 120. We could do that for documentation, but it should not increase our confidence.



## Remove redundant tests

Now that we derived the general solution thanks to our experimental data (the tests we wrote so far), we could also remove some tests that can now be seen as redundant. For example, `testFactorialOf3` is implied by `testFactorialOf4` now that we implemented the factorial. This is not a strict requirement, and we can still keep all the tests that led us to the general solution.

However, we are not done yet, since we should consider possible negative inputs. Passing a negative number should lead to an exception, e.g., an `IllegalArgumentException`.

First, we write the test using one of the mechanisms for testing exceptions that we saw in Chapter [JUnit](#), Section [Exception testing](#). For example,

```

1 @Test
2 public void testNegativeInput() {
3     IllegalArgumentException thrown = assertThrows(IllegalArgumentException.class,
4         () -> factorial.compute(-1));
5     assertEquals("Negative input: -1", thrown.getMessage());
6 }
```

This fails since `compute` still returns 1.

We fix this as follows:

```

1 public int compute(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException("Negative input: " + n);
4     if (n < 2)
5         return 1;
6     return n * compute(n - 1);
7 }
```

Now, since we have tests that act as a “safety net”, we could further refactor our implementation and optimize that: we can implement the factorial computation in an iterative way, e.g., with a “for loop”. This should make our tests still succeed. This is left as an exercise.

## This is not a formal proof

In this section, by using triangulation, we got to the algorithm for implementing the factorial by taking tiny steps following the suggested ordering of transformations. At each step, we generalized the implementation up to the final version. Note that two tests were not enough. Indeed, triangulation mentions “two examples or more”. The triangulation technique is beneficial if you get stuck trying to reach the general implementation at some point. However, this technique should not be used to design an algorithm. Designing an algorithm is not a trial and error, with guesses and attempts. It requires formal proof stating its correctness. Indeed, in this example, we knew the final implementation. We just used the example to demonstrate triangulation. In general, when trying to solve a problem, first look for a possible existing algorithm.

## 4.4 Small or big steps?

As we saw, TDD is about focusing on small problems. However, as stressed by Kent Beck ([Bec02](#)), TDD does not force you always to take “tiny” steps. With TDD, you are aware that you can proceed in small steps.

You might also take bigger steps without violating the TDD principles and the above rules. The important thing is that you learn how to proceed in small and simple steps. When you are in situations you are confident enough to proceed by bigger steps, you can (and probably should) do that. When you are in trouble with the current task or unsure how to proceed, you know that you can (and should) proceed in small steps.

As we saw in Section [\*Three strategies for the green state\*](#), we can also write the “obvious implementation” if we know that and if we have a failing test. If we do not know the general implementation, we can use “triangulation”, as we demonstrated in Section [\*The factorial example with TDD\*](#).

As another example, we want to implement the method `deposit` of the class `BankAccount` of Chapter [\*JUnit\*](#) using TDD. Before the method `deposit` is created, we write the very first test for the “happy” case of `deposit` that is when `deposit` should effectively increment the `balance` like this:

```
1 @Test
2 public void testDepositWhenAmountIsCorrectShouldIncreaseBalance() {
3     // setup
4     BankAccount bankAccount = new BankAccount();
5     bankAccount.setBalance(5);
6     // exercise
7     bankAccount.deposit(10);
8     // verify
9     assertEquals(15, bankAccount.getBalance(), 0);
10 }
```

It might make no sense to create the method `deposit` with an initial trivial implementation setting `balance` to 15 just to make the test pass:

```
1 public void deposit(double amount) {
2     balance = 15;
3 }
```

In such situations, where the code to make the test pass is evident, we can directly implement it in the very first place. In the end, it's just incrementing a field using the passed parameter:

```
1 public void deposit(double amount) {
2     balance += amount;
3 }
```

After all, we're still following TDD, and we wrote only the code to make the test pass and remove duplication.

The same would not hold if we implemented the whole logic of `deposit` directly without having a (failing) test for the case of a negative amount:

```
1 public void deposit(double amount) {
2     if (amount < 0) {
3         throw new IllegalArgumentException("Negative amount: " + amount);
4     }
5     balance += amount;
6 }
```

That would violate the laws of TDD.

## 4.5 Tests and induction (part 1)

This section shows an alternative implementation of the factorial with TDD. As mentioned, TDD does not necessarily force us to proceed in tiny steps. As long as we can write a new failing test and make it pass, we should be fine.

For example, we could use the recursive definition of the factorial (that is, by induction) both for the tests and the implementation. Thus, differently from Section [The factorial example with TDD](#), in this case, we use our existing knowledge of the definition of the factorial.

For the sake of brevity, we directly show the tests, which are meant to be written progressively:

```

1 ...
2 private Factorial factorial;
3
4 @Before
5 public void setup() {
6     factorial = new Factorial();
7 }
8
9 @Test
10 public void testBaseCase() {
11     assertEquals(1, factorial.compute(0));
12 }
13
14 @Test
15 public void testInductiveCase() {
16     assertEquals(5 * factorial.compute(4), factorial.compute(5));
17 }
18
19 @Test
20 public void testNegativeInput() {
21     // omitted, as before
22 ...

```

The first two tests mimic the definition by induction of the factorial. They are also more expressive than the tests written in the previous section. Of course, the `testInductiveCase` is not exhaustive. We cannot write a test that says “for any input greater than 0...”, but at least we can specify the behavior of the factorial correctly with input 5:

“the factorial of 5 must be  $5 * \text{the factorial of } 4$ ”.

The test also performs an assertion in an inductive way. It is as if we are applying the induction hypothesis in a proof by induction.

With these tests, using TDD, we should get to this implementation:

```

1 public int compute(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException("Negative input: " + n);
4     if (n == 0)
5         return 1;
6     return n * compute(n - 1);
7 }
```

This is similar to the previous section, and verifying that they are equivalent is easy. The base case is implemented with an explicit test for 0 instead of checking  $n < 2$  since we followed the `testBaseCase` specification. Similarly, the `testInductiveCase` is expressive enough to make us write the final general solution directly.



There's still a little flaw in the test we wrote, relying on induction. Can you spot it?

We'll get back to that in Chapter *Mutation Testing*, Section *Tests and induction (part 2)*.

Once again, since we have the “safety net” provided by tests, we could refactor our implementation in an iterative way, e.g., with a loop. This should make our tests still succeed. Note that if we implement the factorial iteratively without recursion, our test for the inductive case is still specified recursively by induction. This shows that the specifications (tests) do not need to force the actual implementation. Or, put another way, the implementation does not have to mimic the specifications as long as the results are the same.

## 4.6 TDD with JUnit 4 Parameterized Tests

JUnit 4 provides the mechanism of **Parameterized Tests**. This way, a test case is instantiated several times with a predefined set of arguments, which typically consist of pairs of inputs and expected outputs (but of course, it can be any tuple whose elements have a semantic for that specific test case). For each instantiation, JUnit will run all the test methods. The idea is to specify the data values for the test case instantiation and then write a reduced number of test methods (typically just one).

To use this mechanism, we must

- Annotate the test case with `@RunWith(Parameterized.class)`;
- Implement a public static method annotated with `@Parameters`, which returns the data values for creating the test case instances. This must return a collection of data values: a `Collection<Object[]>`, a collection of arrays of `Object`.
- Thus, each element of the collection is, in turn, an array, and each element of the array will be passed to the constructor of the test case. The constructor must be created accordingly to receive such arguments and store them in the fields of the test case.

The procedure might sound cumbersome, and indeed it is!



A JUnit 4 **runner** is responsible for running tests. We never had to specify any runner since we relied on the default JUnit 4 runner. JUnit 4 itself provides alternative runners like `Parameterized`. We will see additional examples of custom runners provided by testing libraries in Chapter [Mocking](#), Section [Alternative ways of initializing mocks and other elements](#), in Chapter [UI tests](#), Section [UI unit tests](#), and in Chapter [End-to-end tests](#), Section [BDD with Cucumber](#).



JUnit 5 replaces the mechanisms of runners and rules with a single mechanism, the `extension` API, which can be used with the annotation `@ExtendWith`. Moreover, you can specify multiple extensions with `@ExtendWith`. On the contrary, in JUnit 4, you can only specify one runner with the annotation `@RunWith`. We will see an example of such an extension mechanism in Chapter [Mocking](#), Section [Mockito and JUnit 5](#).

Let's apply this mechanism to the `MyStringUtils` example:

```
1 package string.example;
2
3 import static org.junit.Assert.*;
4
5 import java.util.Arrays;
6 import java.util.Collection;
7
8 import org.junit.Test;
9 import org.junit.runner.RunWith;
10 import org.junit.runners.Parameterized;
11 import org.junit.runners.Parameterized.Parameters;
12
13 @RunWith(Parameterized.class)
14 public class MyStringUtilsParameterizedTest {
15
16     @Parameters(name = "{index}: leftTrim({0})={1}")
17     public static Collection<Object[]> data() {
18         return Arrays.asList(
19             new Object[][] {
20                 {null, null},
21                 {"", ""},
22                 {" abc", "abc"},
23                 {"abc", "abc"},
24                 {"\tabc", "abc"},
```

```

25         {" abc", "abc"},  

26         {" ", ""},  

27         {"\t abc", "abc"}  

28     ));  

29 }  

30  

31 private MyStringUtils myStringUtils;  

32 private String input;  

33 private String expected;  

34  

35 public MyStringUtilsParameterizedTest(String input, String expected) {  

36     this.input = input;  

37     this.expected = expected;  

38     myStringUtils = new MyStringUtils();  

39 }  

40  

41 @Test  

42 public void testLeftTrim() {  

43     assertEquals(expected, myStringUtils.leftTrim(input));  

44 }  

45 }

```

In the above test, we used the name argument of `@Parameters` to quickly identify the individual test cases during the execution. The argument can contain *placeholders* such as `{index}` (the current parameter index) and `{n}`, representing the nth parameter value. Note that there is one single test method since JUnit will now create an instance of the test case for each pair of arguments.

When running this test from Eclipse, the JUnit view should show something like

```

1 MyStringUtilsParameterizedTest  

2 [0: leftTrim(null)=null]  

3     testLeftTrim[0: leftTrim(null)=null]  

4     [1: leftTrim()=]  

5         testLeftTrim[1: leftTrim()=]  

6     [2: leftTrim( abc)=abc]  

7         testLeftTrim[2: leftTrim( abc)=abc]  

8 ...

```

The tree view's nesting reflects how the mechanism of parameterized works, as described above.

We have just seen an example of JUnit 4 parameterized tests. How could we use this mechanism for TDD? The above test case example shows the final result. When applying TDD, we could employ the parameterized test mechanism by first setting up the test case according to the rules of parameterized tests. Then, we create a failing test by specifying data values for which we still do not have an

implementation. We make it succeed by implementing only the code for handling the new data values without breaking the tests for the existing data values. Thus, differently from what we have done up to now, for each TDD cycle, instead of a failing test method, we write data values to make the test fail.

While this process, relying on JUnit 4 parameterized tests, fits the TDD methodology, we should note that now we may experience some drawbacks that limit the effectiveness of TDD.

First, setting up a JUnit 4 parameterized test case is not straightforward, as we have just seen. In particular, you probably forget how to do that the next time you have to set up a parameterized test case. This slows down at least the very first TDD cycle.

Moreover, the JUnit 4 parameterized test mechanism is not as flexible as expected. This is more evident if we try to apply this mechanism to the factorial example we saw in Section [The factorial example with TDD](#). The resulting test case looks like the following one:

```
1 @RunWith(Parameterized.class)
2 public class FactorialParameterizedTest {
3
4     @Parameters(name = "{index}: factorial({0})={1}")
5     public static Collection<Object[]> data() {
6         return Arrays.asList(
7             new Object[][] {
8                 { 0, 1 },
9                 { 1, 1 },
10                { 2, 2 },
11                { 3, 6 },
12                { 4, 24 },
13            });
14     }
15
16     private Factorial factorial;
17     private int input;
18     private int expected;
19
20     public FactorialParameterizedTest(int input, int expected) {
21         this.input = input;
22         this.expected = expected;
23         factorial = new Factorial();
24     }
25
26     @Test
27     public void testFactorial() {
28         assertEquals(expected, factorial.compute(input));
29     }
```

```

30
31 // unfortunately, this is executed over and over again for each
32 // { input, expected }
33 @Test
34 public void testNegativeInput() {
35     IllegalArgumentException thrown = assertThrows(IllegalArgumentException.class,
36         () -> factorial.compute(-1));
37     assertEquals("Negative input: -1", thrown.getMessage());
38 }
39 }
```

In fact, as explained in the comment of `testNegativeInput`, that specific test method is executed for each data value pair `{ input, expected }`, although, in that test method, the data values (the fields `input` and `expected`) are not even used at all. This can be seen in the JUnit view:

```

1 FactorialParameterizedTest
2   [0: factorial(0)=1]
3     testFactorial[0: factorial(0)=1]
4     testNegativeInput[0: factorial(0)=1]
5   [1: factorial(1)=1]
6     testFactorial[1: factorial(1)=1]
7     testNegativeInput[1: factorial(1)=1]
8   ...
```

This is definitely a waste of resources, and for sure, it does not sound right at all.

Thus, JUnit 4 parameterized tests can be used for TDD, but only when we can express each scenario to test in a single parameterized test. This makes it hard to use it when testing exceptional cases. Of course, one could put the exceptional cases in a separate test case, but still, this is far from optimal.

As we see in Section [TDD with JUnit 5 Parameterized Tests](#), JUnit 5 has a more powerful and flexible mechanism for a parameterized test, which can effectively help when using TDD.

## 4.7 TDD with JUnit 5 Parameterized Tests

The mechanism of parameterized tests is also present in JUnit 5. However, parameterized tests in JUnit 5 are much more powerful than JUnit 4 parameterized tests (see Section [TDD with JUnit 4 Parameterized Tests](#)). In JUnit 5, single test methods are parameterized: a parameterized test method is declared just like a standard test method. Still, it has to be annotated with `@ParameterizedTest` (`org.junit.jupiter.params.ParameterizedTest`) instead of `@Test`. Moreover, at least one `source` must be specified that is meant to provide the arguments for each invocation. Such arguments will be consumed in the parameterized test method.

For example, the source can be specified with `@ValueSource`, with a single array of literal values (of basic types, of `String` or `Class`). Thus, this can only provide a single argument for the parameterized test method. For other mechanisms for specifying the source of parameterized tests, we refer to the official documentation: <https://junit.org/junit5/docs/current/user-guide>.

In this section, we use `@CsvSource` to specify the arguments of our parameterized tests. This annotation allows us to specify argument lists as comma-separated string values. If a string value with a comma inside has to be specified, the string must be enclosed in single quotes (''). An empty string is specified with a quoted empty value (' '). A `null` reference is specified with an empty value.

Let's apply this mechanism to the `MyStringUtils` example (The optional `name` argument of `@ParameterizedTest` allows us to identify the individual test cases during the execution quickly; the placeholders are similar to the ones we saw in Section [TDD with JUnit 4 Parameterized Tests](#)):

```
1 package string.example;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 import org.junit.jupiter.api.BeforeEach;
6 import org.junit.jupiter.params.ParameterizedTest;
7 import org.junit.jupiter.params.provider.CsvSource;
8
9 class MyStringUtilsJupiterTest {
10
11     private MyStringUtils myStringUtils;
12
13     @BeforeEach
14     void setup() {
15         myStringUtils = new MyStringUtils();
16     }
17
18     @ParameterizedTest(name = "{index}: '{0}' => '{1}'")
19     @CsvSource({
20         "", "",
21         "", "", "",
22         "' abc', abc",
23         "abc, abc",
24         "'\tabc', abc",
25         "' abc', abc",
26         "' ', ''",
27         "'\t abc', abc"
28     })
29     void testLeftTrim(String input, String expected) {
30         assertEquals(expected, myStringUtils.leftTrim(input));
```

```
31     }
32 }
```

Thus, differently from the parameterized test version of Section [TDD with JUnit 4 Parameterized Tests](#), this version with JUnit 5 is easy to set up, write and read. Note the use of single quotes, particularly the first value representing two `null` references. Thus, this mechanism can be fruitfully used for the TDD cycles. Note also that the methods such as `@BeforeEach` are still being used to initialize the fixture.

When running this test from Eclipse, the JUnit view should show something like

```
1 MyStringUtilJupiterTest
2   testLeftTrim(String, String)
3     1: 'null' => 'null'
4     2: '' => ''
5     3: ' abc' => 'abc'
6   ...
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

Since we parameterized a single test method, this mechanism can also be used for test cases with exceptional cases, like the factorial example we saw in Section [The factorial example with TDD](#):

```
1 class FactorialJupiterTest {
2
3   private Factorial factorial;
4
5   @BeforeEach
6   void setup() {
7     factorial = new Factorial();
8   }
9
10  @ParameterizedTest
11  @CsvSource({
12    "0, 1",
13    "1, 1",
14    "2, 2",
15    "3, 6",
16    "4, 24"
17  })
18  void testFactorial(int input, int expected) {
19    assertEquals(expected, factorial.compute(input));
20  }
21
22  @Test
```

```

23 void testFactorialNegativeInput() {
24     IllegalArgumentException thrown = assertThrows(IllegalArgumentException.class,
25         () -> factorial.compute(-1));
26     assertEquals("Negative input: -1", thrown.getMessage());
27 }
28 }
```

The parameterized test `testFactorial` is separate from the standard test for the negative input: the latter is executed only once, as it should be.

## 4.8 Testing abstract classes?

It might be the right moment to address the question of testing abstract classes. An abstract class, by its nature, cannot be instantiated. How do we create a SUT in our test case? This problem can be easily circumvented by creating an implementation of the abstract class that we use only in our tests. What if the abstract class has some abstract methods?<sup>3</sup> We should provide an implementation for such methods in our concrete class that we use in the tests. How do we implement such methods? With an empty implementation? What if the concrete methods of the abstract class call abstract methods? Can we test those concrete methods with our empty implementation of abstract methods? Things are getting kind of strange now.

The question of this section is quite similar to what we saw in Chapter [JUnit](#), Section [Testing private methods?](#). In that section, we hinted that private methods should be tested indirectly through accessible methods of the same class. Moreover, we said private methods should “appear” in our tested class during a refactoring (e.g., with **Extract Method**...). In this chapter, this makes much more sense since we saw that refactoring is the last phase of every Red-Green-Refactor cycle. Thus, the contents of an extracted private method have already been tested.

We can apply the same idea now for abstract classes. You don’t write an abstract class right away. You implement a concrete class with TDD, and then you “extract” the abstract superclass (e.g., with the refactoring **Extract Superclass**), possibly by declaring some methods as abstract in the extracted superclass (this can be done during the mentioned refactoring). Of course, the original concrete class has already implemented those abstract methods. In the end, the abstract class is already tested by the tests of the concrete class.

Suppose the extracted abstract methods are meant to be protected (e.g., they are meant to be called from public methods of the class). That is similar to private methods: they are made protected during refactoring and are called by public methods that have already been tested.

Now we have an abstract class and a concrete class thoroughly tested. What if we now need another concrete subclass? Again we can apply TDD. We could also start by copying the whole test case class of the previous concrete class into a new test case class. Then in the new test case, we mention the yet non-existing new concrete class; we create it as a subclass of the abstract class and make sure

---

<sup>3</sup>Recall that this is not always the case in Java: a class can be declared as `abstract` even without abstract methods.

that all tests pass. The new concrete subclass is probably slightly different from the original concrete class. We update the expectations of a test, ensure it fails, and update the new concrete subclass to make the test pass. And we go on like that, still following the TDD cycle.

## 4.9 Writing a test that succeeds

We conclude this chapter with some summary reflections on the cases when we write a test that succeeds immediately with the production code we have already written.

As we said in this chapter, the three laws prevent us from writing any production code without a failing test. Moreover, we should write only enough production code to make the failing test succeed without breaking existing tests. If we want to write a failing test that instead succeeds, we are writing the test in the wrong way since it's a false positive. We expect it to fail because it should test a situation that our code cannot handle while it succeeds: the test must be wrong. If the test is correct, we have a terrible grasp of our code! In any case, a test that passes when it should fail is an alarm bell.

On the other hand, the three laws do not prevent us from writing a test we expect to pass because it is meant to verify a scenario our production code can already handle. But why are we doing that?

If we are applying triangulation, then we are verifying that our production code, in the current state, can behave correctly with our experimental data (the tests), and we're fine. If the test is meant to document a case that might not be immediately evident by looking at the other tests, then we're okay again.

But what if we are doing that because we are not sure of what's going on in our production code? Then, probably, we are not following the TDD methodology correctly. We are not focusing on small problems. We are trying to deal with a problem that's too big to handle in the part of the code we are working on. Alternatively, we are probably not following the suggestions of the Transformation Priority Premise. We applied complex transformations before simple ones, which might lead to a more complicated code with a flow that's hard to understand and follow.

Summarizing, writing a test that succeeds right away is acceptable if it's done for good reasons and with knowledge of the cause.

Note that we are still talking about unit tests and TDD. Things are different for integration tests (Chapter [Integration tests](#)) and e2e tests (Chapter [End-to-end tests](#)). These are meant to verify that when composing single components that have already been written and tested in isolation (possibly by using mocking, Chapter [Mocking](#)) everything still works as expected. This means that integration and e2e tests are expected to succeed right away. If that's not the case, then something is wrong with the tests (these tests are more complex to write than unit tests), or something is wrong in some of the integrated components, probably because some of them not respecting the contracts expected by the other components. Finally, recall from Chapter [Testing](#), Section [Automated tests](#), that UI tests can also be unit tests and can be used for implementing the user interface with TDD (Chapter [UI tests](#)). When UI tests are unit tests, what we saw in this chapter is still applicable and should be followed.

# 5. Code coverage

*Code coverage* measures how many lines of code are executed while the application runs. When code coverage is measured, and the running application is a test suite, then code coverage will tell you how many parts of your code are being executed by your test suite. We can also use the term *test coverage* in this case. However, from now on, we will use the term *code coverage* to refer to *test coverage*.

In some sense, code coverage will give you an idea of how many parts of your code are tested. Of course, this statement highly relies on your tests being written correctly; that is, they effectively test your code. We'll get back to this later in this chapter. A code base with high test coverage, measured as a percentage, has had many parts executed during testing. High test coverage lowers the chance of containing undetected software bugs differently from a program with low test coverage.

Code coverage is collected by using a specialized tool. Such a tool typically *instruments* the binaries to keep track of executed statements. It also keeps track of possible branch instructions, like `if-then-else`, loops, switches, etc., so it can tell you whether all branches are covered. The same holds for complex expressions like *boolean expressions*: the tool tells you whether all the possibilities have been covered.

In this chapter, we will use the mainstream Java tool for code coverage, JaCoCo, and its integration with Eclipse, EclEmma.

## 5.1 JaCoCo and EclEmma

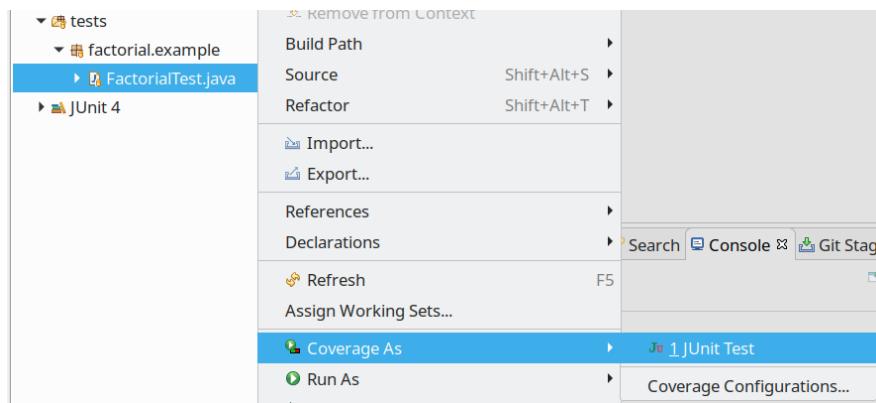
There are a few tools for computing code coverage. In Java, the mainstream one is **JaCoCo** (Java Code Coverage), <https://www.eclemma.org/jacoco/>. JaCoCo implements a Java agent<sup>1</sup> to execute any Java application (in particular, JUnit tests) and provides a report with all the lines of code executed during the application. If there are branch instructions, like `if-then-else`, it also reports if all branches are executed during the application.

The JaCoCo plugin for Eclipse is **EclEmma**: <https://www.eclemma.org/> and can be installed from the update site <http://update.eclemma.org/> (the feature to install is called “**EclEmma Java Code Coverage**”). 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”:

---

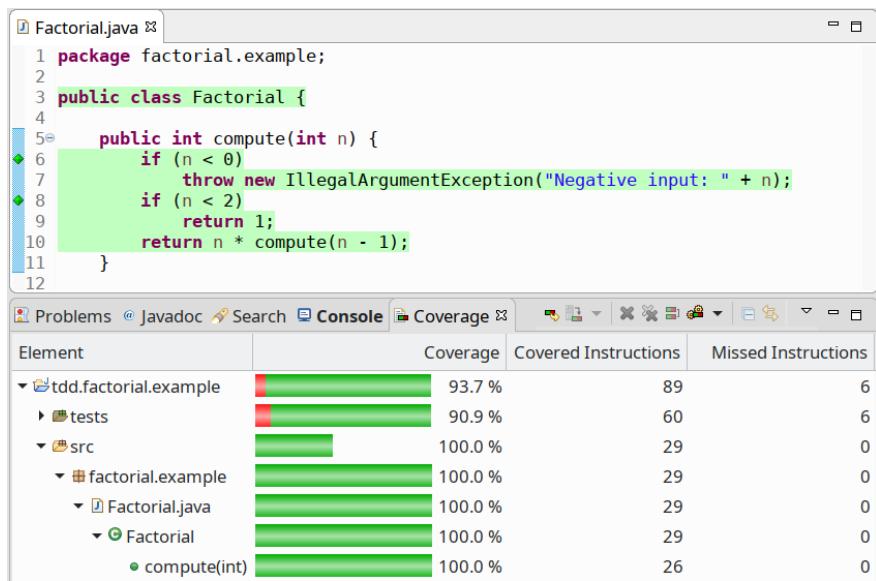
<sup>1</sup>Generally, a *Java agent* is a JVM “plugin” that relies on the Java Instrumentation API.



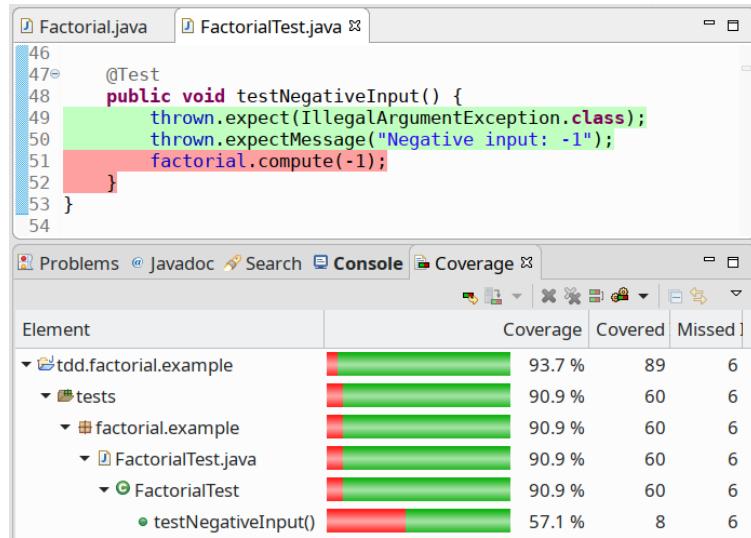
Tests will execute as before, but this time, JaCoCo will keep track of executed instructions and will show a report in the view “Coverage”. Java editors will also be colored accordingly:

- “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:

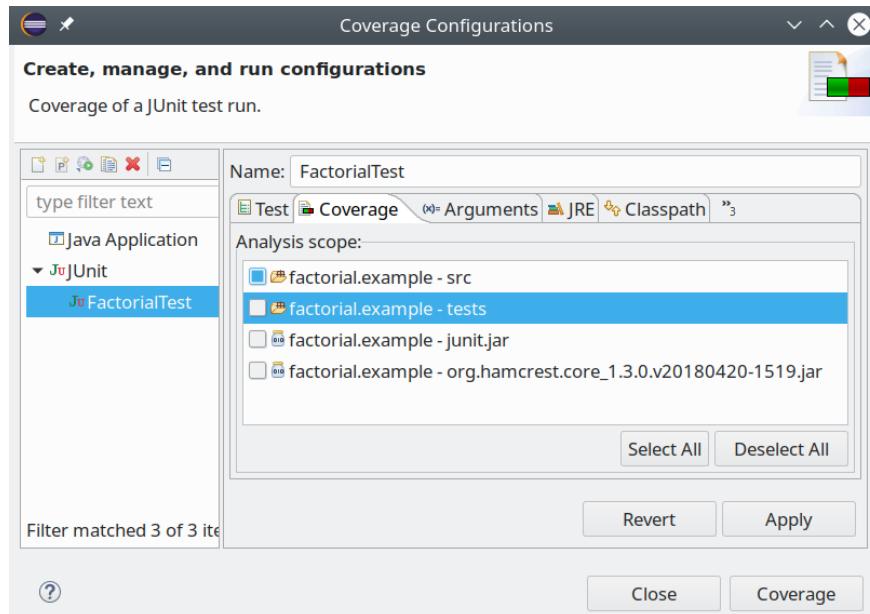


Note that the test class will have a red color for the `testNegativeInput` as if that test was never executed:



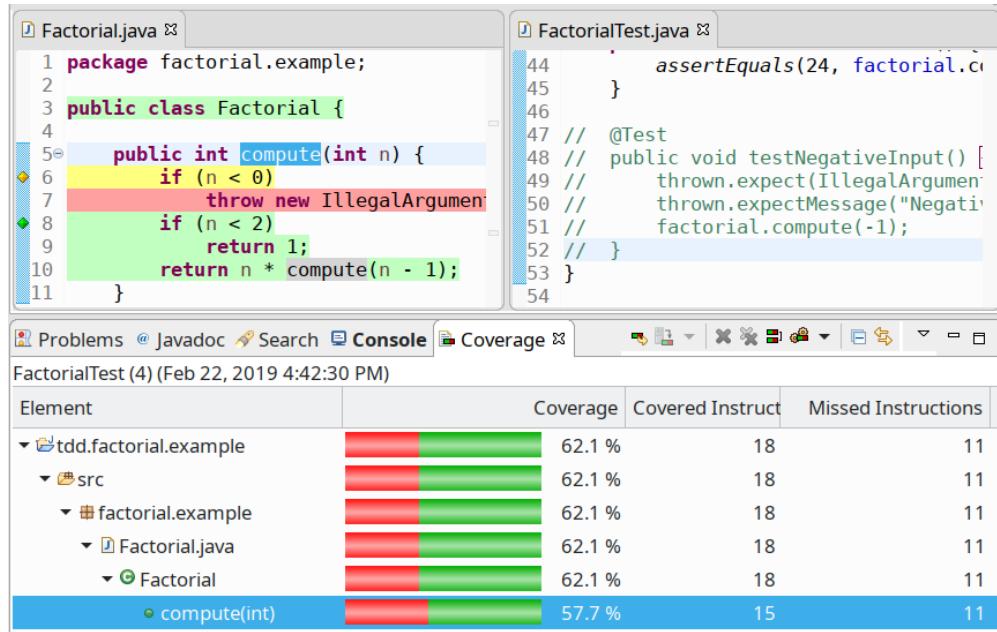
This is a limitation of JaCoCo that does not consider method invocations as covered when they throw an exception. (On the contrary, a `throw` statement is correctly covered when executed.) This is not a problem since we should exclude test classes from code coverage: we must concentrate on SUT classes.

In EclEmma, we can configure such an exclusion in the Run Configuration of the test. In the tab “Coverage,” you can refine the scope of the analysis by excluding tests directories:



Rerun the coverage, and you’ll see that tests will not be considered in the report. Once configured correctly, you may want to save the run configuration of the coverage (and tests) in the Eclipse project (refer to Chapter [Eclipse](#), Section [Eclipse Run configurations](#)). You’ll still have your custom launch if you import that project into another workspace.

Let's experiment with code coverage: let's uncomment the test for the exception and rerun the code coverage. You'll see that one line will not be covered, and the `if` will have an uncovered branch:



JaCoCo is smart enough to ignore some parts of the code that could not be executed during tests. For example, private empty no-arg constructors are automatically ignored.

## 5.2 How code coverage can help

Let's go back to our first Bank example we saw in Chapter [JUnit](#).

If we run the tests with EclEmma, we can verify that `BankAccount` is fully covered, but the same does not hold for `Bank`:

```
private BankAccount findBankAccountById(int bankAccountId) {
    return bankAccounts.stream()
        .filter(a -> a.getId() == bankAccountId)
        .findFirst()
        .orElseThrow(
            () -> new NoSuchElementException("No account found with id: " + bankAccountId));
}
```

1 of 2 branches missed.  
Press 'F2' for focus

This might look strange since we wrote a test for the case when the account is found and for the case when the account is not. However, the code coverage tells us that our tests cover only one case of the boolean expression. For sure, the case when the boolean expression is true is covered, but why is the other case uncovered?

These are the two tests that should cover both cases:

```

1  @Test
2  public void testDepositWhenAccountIsNotFoundShouldThrow() {
3      thrown.expect(NoSuchElementException.class);
4      thrown.expectMessage("No account found with id: 1");
5      bank.deposit(1, 10);
6  }
7
8  @Test
9  public void testDepositWhenAccountIsFoundShouldIncrementBalance() {
10     // setup
11     BankAccount testAccount = createTestAccount(INITIAL_BALANCE);
12     bankAccounts.add(testAccount);
13     // exercise
14     bank.deposit(testAccount.getId(), AMOUNT);
15     // verify
16     assertEquals(INITIAL_BALANCE+AMOUNT, testAccount.getBalance(), 0);
17 }
```

The filter predicate never evaluates to false: in the “happy” test, we use a list with a single account, which is the one that should be found; thus, the filter predicate evaluates to true. In the other case, we use an empty list; thus, the filter predicate is never executed.

To have full coverage of both branches, we can add another test account to the list, ensuring that the account to be found is not the first one. The filter predicate will be evaluated as false and true (in this order).

You might think this is overkill, and we do this only to have full coverage (more on 100% code coverage in the next section).

However, let’s deal with this situation from another perspective and let’s do an experiment: let’s comment the call to `filter` in the `Bank.findBankAccountById` method:

```

1  private BankAccount findBankAccountById(int bankAccountId) {
2      return bankAccounts.stream()
3          // Intentionally introduced bug: no filtering
4          // .filter(a -> a.getId() == bankAccountId)
5          .findFirst()
6          .orElseThrow(
7              () -> new NoSuchElementException
8                  ("No account found with id: " + bankAccountId));
9 }
```

Run the `BankTest`, and everything is still green! Despite introducing a bug in our implementation, our tests still succeed! That is because we are not testing the “happy” case correctly, and code coverage gave us a hint about the missing covered branch.

Let's modify the "happy" case test as follows: we add another account to the list after the one that should be found (local variables have been renamed for better understanding):

```
1 @Test
2 public void testDepositWhenAccountIsFoundShouldIncrementBalance() {
3     // setup
4     BankAccount toBeFound = createTestAccount(INITIAL_BALANCE);
5     bankAccounts.add(toBeFound);
6     BankAccount another = createTestAccount(INITIAL_BALANCE);
7     bankAccounts.add(another);
8     // exercise
9     bank.deposit(toBeFound.getId(), AMOUNT);
10    // verify
11    assertEquals(INITIAL_BALANCE+AMOUNT, toBeFound.getBalance(), 0);
12 }
```

Rerun the tests with the bugged `Bank.findBankAccountById`; everything is still green! Since the account we expect to find is the first element in the list, the test succeeds even without filtering. We still need to test the behavior correctly.

As proof, remove the bug from `Bank.findBankAccountById`, run the tests with code coverage, and still, one branch of the boolean expression is not covered (again, the one evaluating to false).

Let's fix the test by making sure that the account to be found is not the first one:

```
1 @Test
2 public void testDepositWhenAccountIsFoundShouldIncrementBalance() {
3     // setup
4     // first, an account that does not match...
5     BankAccount another = createTestAccount(INITIAL_BALANCE);
6     bankAccounts.add(another);
7     // ...then the one we want to modify
8     BankAccount toBeFound = createTestAccount(INITIAL_BALANCE);
9     bankAccounts.add(toBeFound);
10    // exercise
11    bank.deposit(toBeFound.getId(), AMOUNT);
12    // verify
13    assertEquals(INITIAL_BALANCE+AMOUNT, toBeFound.getBalance(), 0);
14 }
```

Run the tests with code coverage; now everything is fully covered. Re-introduce the bug in `Bank.findBankAccountById`, and this test will fail.

To summarize, when code coverage detects lines of code that are not covered or missing branches in boolean expressions or branching statements you should treat it as an “alarm bell”, especially when you thought that you had tested all the possible paths.

In Chapter [Mutation Testing](#), we will use a tool that automatically performs modifications like the bug we introduced in `Bank.findBankAccountById` and automatically runs the whole test suite, making sure that at least one test fails.



It could be worthwhile to modify also the test `testWithdrawWhenAccountIsFoundShouldDecrementBalance` along the same line.

## 5.3 Code coverage percentage

It is best practice to have a high code coverage percentage. For big projects, you can read in many places that you should aim for 80% code coverage and that reaching 100% is not that important.

Now I do not want to start a flame war, but my opinion is that **you should always aim at 100% code coverage**. Well, 100% code coverage of the *important* code, that is, code that contains some “logic”.

For example, you should not even test classes that contain only fields, getters, and setters with no logic (e.g., they do not contain branch instructions). These are known as *JavaBeans* or *POJOs* (Plain Old Java Objects). Typical examples are the classes that represent your application’s *Domain Model* objects that could also be stored in a database.

Such classes could, however, contain methods such as `toString`, `equals`, and `hashCode`. If you have such methods generated by some tools (e.g., Eclipse can generate such methods for you based on the fields of the classes), you could avoid testing these methods. Otherwise, you could write tests for these methods, but for sure, it is pointless to have test methods for getters and setters.

Thus, these classes might be less than 100% code covered. In this case, you can exclude them from the code coverage. We saw above that we can define the source folders that make the scope of the code coverage analysis. When we start using Maven, we will see how to configure the JaCoCo Maven plugin to exclude some classes from the code coverage (see Chapter [Maven](#), Section [Configuring the JaCoCo Maven plugin](#)).

Remember also that test classes themselves should be excluded from the code coverage.

Following TDD correctly makes it easy to reach 100% code coverage. This is because you will be forced to write code that you can easily test. On the contrary, thoroughly testing existing classes written without TDD might be more complicated. That is why in this book, we foster the use of TDD. In the end, TDD is about not writing any code without a failing test. The TDD cycle is tight and leads you to cover one code branch at a time, including error scenarios (e.g., throwing exceptions). Thus, by following TDD, any code that needs to be covered is covered immediately.

Finally, 100% code coverage should be reached with unit tests only; that is, integration tests should not increase code coverage. In particular, reaching 100% code coverage for a single class should be done only with the corresponding test case. That is to say, a test case for class A should not contribute to reaching 100% code coverage of another class B.

Of course, in the classes we have seen so far, it is easy to write unit tests in an isolated way, even in the presence of collaborator classes. We will see in another chapter (Chapter [Mocking](#)) that when a class has more involved collaborators and dependencies, we can still write isolated unit tests (reaching 100% code coverage) by using *mock* objects: we replace collaborators with “fake implementations” that we can easily stub to test our SUT.

If you achieve that, it is a good sign that you’re on the right track. In my experience, these methodologies always allow me to have modular and clean code (which is also maintainable) and well-tested, with 100% code coverage.

A final remark is crucial: 100% code coverage is a good starting point, but it does not guarantee that your code is well-tested.

We will also see in Chapter [Mutation Testing](#), that even 100% code coverage does not guarantee that your code is correct. In particular, you should always ask yourself, “do my tests effectively test all the behavior of my code?”. After all, code coverage only checks that code is executed when running tests, but if your tests do not perform the correct assertions, then code coverage is completely meaningless!

On the other hand, not reaching 100% code coverage can signal that your code is less modular and maintainable. In particular, you probably need to follow TDD correctly.

Finally, suppose you manage to reach 100% code coverage. In that case, if in the future you modify your code (by changing something or adding new features) and your code coverage decreases, it is much easier to spot it: it’s easier to note that from 100%, it decreases to 99.9% than, say, from 85% to 84.9%.

# 6. Mutation Testing

In the previous chapter, we saw that code coverage is a useful metric, which can help to discover parts of the main code that is never executed during tests. However, we also anticipated that it cannot guarantee that the quality of your tests is good. In particular, we suggested that you should aim at 100% code coverage (of the important code). In any case, 100% code coverage does not necessarily mean that the code is 100% tested: your tests could cover every line of your code, but they might perform the wrong assertions or no assertion at all.

Summarizing, low code coverage should give you a cause for concern, but 100% code coverage should not necessarily make you feel safe.

In the previous chapters, to have better confidence about the quality of our tests, we manually modified the SUT in some crucial parts, and we checked whether our tests would still be green. For example, we changed an increment operator `+=` to a simple assignment `=` and we removed the call to some methods. If our tests are still green with such crucial changes, then our tests do not effectively test the behavior of the SUT. However, doing such experiments manually takes too much time and should be repeated for every change in the SUT or in the tests.

**Mutation testing** frameworks perform such operations automatically. For this reason, such frameworks help you evaluate the quality of your tests.

The idea is to “mutate” the SUT according to several mutation strategies. Every single mutation of the SUT generates a mutated version of the SUT called **mutant**. Then, the whole test suite is run using such a mutant. If at least one test fails when using the mutant then the mutant is said to be **killed** and we are fine: our tests (at least one) detected such a change. If all the tests still succeed using a mutant then the mutant is said to be **survived** and we are in trouble: our tests do not completely test all the complexity of the code since they did not detect the change.

Mutants are generated using **mutators**, which are mutation patterns applied to the main code. Mutators perform changes of the shape:

- change `&&` into `||` in a boolean expression (or vice-versa)
- replace a boolean expression with `true` (or `false`)
- remove an assignment to a member variable
- modify literals (e.g., turning `0` to `1` or vice-versa)
- completely remove a method call
- etc.

In this book, we will use **PIT**, <https://pitest.org/>, one of the mainstream Java mutation testing frameworks. In particular, there is an Eclipse plugin, **Pitclipse**, which can be installed from the update site that you find on the project’s website <https://github.com/pitest/pitclipse>; you need to

install the feature “Pitclipse UI”. In the rest of this chapter, we will use PIT from Eclipse through its plugin.

PIT comes with a predefined set of mutators, and additional custom mutators can be implemented.

The process implemented by PIT can be sketched as follows:

- For each class of the main code
  - For each mutator
    - \* Create a mutant
    - \* Run the tests using the mutant
    - \* Aggregate result

It should be obvious that mutation testing requires a lot of time to execute, due to the above procedure. Thus, it is not meant to be used continuously while developing. It is meant to be used periodically and possibly from a continuous integration server (Chapter *Continuous Integration*).

In Chapter *Maven*, Section *Configuring the PIT Maven plugin*, we will see how to run PIT during a Maven build.

## 6.1 The first example with PIT

Let’s start with a very simple example, which is a simplified variant of the BankAccount we used in Chapter *JUnit*:

```
1 package testing.example.bank;
2
3 public class SimpleBankAccount {
4
5     private double balance = 0;
6
7     public SimpleBankAccount() {
8
9
10    public SimpleBankAccount(double balance) {
11        this.balance = balance;
12    }
13
14    public double getBalance() {
15        return balance;
16    }
17
18    public void deposit(double amount) {
```

```
19     if (amount <= 0) {
20         throw new IllegalArgumentException("Negative amount: " + amount);
21     }
22     balance += amount;
23 }
24 }
```

This code seems “innocent” and we can easily verify that the following test case gives us 100% code coverage:

```
1 package testing.example.bank;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Test;
6
7 public class SimpleBankAccountTest {
8
9     @Test
10    public void testDepositWhenAmountIsCorrectShouldIncreaseBalance() {
11        SimpleBankAccount bankAccount = new SimpleBankAccount(5);
12        bankAccount.deposit(10);
13        assertEquals(15, bankAccount.getBalance(), 0);
14    }
15
16    @Test
17    public void testDepositWhenAmountIsNegativeShouldThrow() {
18        // setup
19        SimpleBankAccount bankAccount = new SimpleBankAccount();
20        try {
21            // exercise
22            bankAccount.deposit(-1);
23            fail("Expected an IllegalArgumentException to be thrown");
24        } catch (IllegalArgumentException e) {
25            // verify
26            assertEquals("Negative amount: -1.0", e.getMessage());
27            assertEquals(0, bankAccount.getBalance(), 0);
28        }
29    }
30
31 }
```

However, this does not prove that our tests effectively test the whole behavior; moreover, this does

not even exclude possible implementation errors in our SUT.

Indeed there's an error: the condition

```
1 if (amount <= 0)
```

should have been

```
1 if (amount < 0)
```

to be consistent with the message of the thrown exception

```
1 throw new IllegalArgumentException("Negative amount: " + amount);
```

Just as an experiment (we performed similar experiments also in Chapter [JUnit](#)), let's change the condition in the `if` (from `<=` to `<`), and let's run the tests. The tests are still green, but that's a bad thing! Indeed, we changed our code and the unit tests did not fail. This means that our unit tests do not test correctly the whole behavior of the SUT (despite the 100% code coverage).

A mutation testing framework performs such modifications automatically and checks whether the tests fail or succeed. If they fail then we're OK, otherwise, we have to improve our tests.

We now use the tool for mutation testing, PIT, to detect such problems in our tests.

Spotted problems in our tests usually correspond to bugs in our SUT or to problems in the implementation of tests.

We run `SimpleBankAccountTest`, using PIT: right-click on the test case and select **Run As → PIT Mutation Test**.<sup>1</sup>



## Mutation testing requires a green suite.

You can use PIT for running mutation testing only when you have a green test suite: all your tests must succeed before running PIT, or it will immediately stop with an error.

Now, PIT will perform several mutations on the SUT. Each mutation will give birth to a “mutant”. For each mutant PIT will run all the tests against the mutant, and in the end, it will provide us with a report about survived and killed mutants.

As anticipated, if all the mutants are KILLED then we're OK. Possible SURVIVED mutants are reported (the actual mutator applied to the code is also specified) and we must check whether the problem is in the tests or in the code.

---

<sup>1</sup>For a possible bug in PIT, the very first execution might end with a `NullPointerException` exception in the console. Just run the test again, in case.



A mutant can survive with the state “NO\_COVERAGE” in case the initial test suite itself did not enjoy 100% code coverage.

We see the execution on the “Console” view. Then, we use the view “PIT Mutations”, where KILLED and SURVIVED mutants are shown. Double-clicking on an element of this view will lead us in the editor in correspondence of the mutation representing that specific mutant. In our example, one mutant SURVIVED:

```

20  public double getBalance() {
21      return balance;
22  }
23
24  public void deposit(double amount) {
25      if (amount <= 0) {
26          throw new IllegalArgumentException("Negative amount: " + a
27      }
28      balance += amount;
29  }

```

Problems Console Coverage PIT Mutations PIT Summary

- \* SURVIVED (1)
  - mutation.testing.initial.example (1)
    - testing.example.bank (1)
      - testing.example.bank.SimpleBankAccount (1)
        - 25: changed conditional boundary
  - \* KILLED (3)
    - mutation.testing.initial.example (3)
      - testing.example.bank (3)
        - testing.example.bank.SimpleBankAccount (3)
          - 21: replaced return of double value with -(x + 1) for testing/example/bank/SimpleBar
          - 25: negated conditional
          - 28: Replaced double addition with subtraction

One mutant survived

You can see that the mutant that survived is the one related to the “wrong” condition in the `if (amount <= 0)`: the mutant consists of the same code where the conditional boundary has been changed from `<=` to `<`. The mutant survives because our tests are still green with that mutation. Indeed, we don’t have a test for the boundary case.

It is straightforward to verify that the other 3 mutants are killed by our test suite:

- if the value returned by `getBalance` is changed then both tests will fail;
- if the condition `amount <= 0` is negated then both tests will fail;
- if the statement `balance += amount` is changed to `balance -= amount`, then at least `testDepositWhenAmountIsCorrectShouldIncreaseBalance` fails.

Note that for each line of our SUT, several mutations can be applied, as in the case of `if (amount <= 0)`: from this line PIT generates two mutants, one of which survives.



Of course the mutation that changes the conditional boundary works for all conditions that have boundaries, e.g., from  $\leq$  to  $<$ , from  $<$  to  $\leq$ , etc.

When this happens, we should take some time to realize what is going on. We don't have a test for the boundary case, so it's quite easy to add such a test. When we write such a test we realize that we have a bug in the SUT: the case when the amount is 0 is legal according to the message of our exception. (Alternatively, we could decide that it is correct that 0 is an illegal value, and we should update the message of the thrown exception accordingly.)

For example, we write the new test:

```
1 @Test
2 public void testDepositWhenAmountIsZeroIsOk() {
3     SimpleBankAccount bankAccount = new SimpleBankAccount(5);
4     bankAccount.deposit(0);
5     assertEquals(5, bankAccount.getBalance(), 0);
6 }
```

We run the test, and this fails since an exception is thrown. We realize that the bug is in the wrong condition in the SUT and we update the code of the SUT accordingly:

```
1 if (amount < 0)
```

and now all tests are green.

If we run PIT again, this time no mutant will survive: the mutant that changes  $<$  to  $\leq$  will make this new test fail, and the mutant is killed.



PIT also provides in Eclipse the view "PIT Summary", which provides an HTML report that can be navigated as a standard HTML web site.

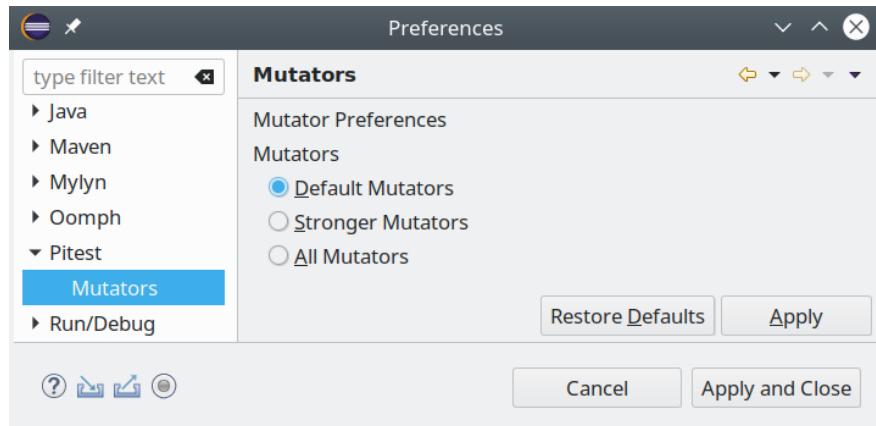
## Bugs are likely to live in untested code

This example shows that even in the presence of 100% code coverage we cannot be sure that our code is bug-free if we are not sure about the quality of our tests. In this example, our tests did not test the overall behavior of our code. Tests did not verify all the paths in our code. The bug was indeed in a path we did not test. In this example, we did not apply TDD. Indeed, TDD highly reduces the risk of having untested scenarios, like the one in this example, and thus it reduces the presence of bugs.

## 6.2 Enabling additional mutators

By default, only a reduced set of mutators is enabled in PIT, in order to avoid long-running mutation testing sessions.

In Eclipse, additional mutators can be enabled using the corresponding preference page:



For example, let's enable "Stronger Mutators" and let's run PIT again on this simple example. In this example, no further mutators are being used. (Recall that some mutators can be applied only on specific parts of code and on specific instructions and statements.)

Let's enable "All Mutators" and let's run PIT again. Now PIT will require some more time to execute since it generates many more mutants (and remember that for each mutant all tests are executed).

We now have several additional survived mutants. For example, "Removed assignment to member variable balance" (actually two versions of this mutant) and "Substituted 0.0. with 1.0" both related to the member variable initialization:

```
1 private double balance = 0;
```

This should look odd since when testing the case of the negative amount, we verify that the balance is not modified and thus it stays to the initial amount 0.

You can see that several mutants with the same mutation can refer to the same line in the source file. As stated in the [PIT documentation about mutators](#)<sup>2</sup>, to increase the performance and reduce the overhead of mutant generation, mutations are performed on the byte code generated by the compiler rather than on the source files. This might lead to equivalent mutants for the same source file line number.

In particular, when instance variables are explicitly initialized with the default value (as in the example above) equivalent mutants will be created. Indeed, if the assignment to 0 is removed from the balance initialization, the field will still be automatically initialized by the virtual machine to 0.

---

<sup>2</sup><http://pitest.org/quickstart/mutators/>

You can also note that this might lead to false positives like the “Substituted 0.0. with 1.0” above. If you look at the “PIT Mutations” view, such a mutant is both killed and survived.

In such cases, you can either ignore such mutators or simply remove the redundant initialization to the default value:

```
1 private double balance;
```

Now, we got rid of all the versions of the mutants “Removed assignment to member variable balance” and “Substituted 0.0. with 1.0”.

When not using the Java default value for basic types, in the presence of constructors, the actual initialization should be moved inside the constructors to avoid duplicate mutants and false positives.

For example, let’s assume that we want the default initial balance to be 10:

```
1 private double balance = 10;
```

The test for the negative amount should of course be updated, e.g., as follows:

```
1 @Test
2 public void testDepositWhenAmountIsNegativeShouldThrow() {
3     // setup
4     SimpleBankAccount bankAccount = new SimpleBankAccount(0);
5     ...
```

With all mutators enabled, we will have duplicate survived mutants: two versions of “Removed assignment to member variable balance” and two versions of “Substituted 10.0. with 1.0” both related to the member variable initialization.

A test asserting the initial default value is required:

```
1 @Test
2 public void testInitialBalance() {
3     assertEquals(10, new SimpleBankAccount().getBalance(), 0);
4 }
```

But this will kill only one version of the duplicate mutants.

Moving the initialization to the default initial value inside the no-arg constructor will avoid the two remaining survived mutants:

```

1 public class SimpleBankAccount {
2
3     private double balance;
4
5     public SimpleBankAccount() {
6         balance = 10;
7     }

```

Several mutants still survive, but they are all related to incrementing and decrementing local variables, parameters, and fields.

Let's have a look at the two mutants "Decremented/Incremented double field balance" related to the getter method:

```

1 public double getBalance() {
2     return balance;
3 }

```

The two mutations perform a post-decrement and post-increment of the field `balance` in the method. Indeed, we have no test verifying that after the getter method is called the field is not changed.<sup>3</sup>

Let's try adding such a test, for example:

```

1 @Test
2 public void testGetBalanceDoesNotChangeTheField() {
3     SimpleBankAccount bankAccount = new SimpleBankAccount();
4     assertEquals(10, bankAccount.getBalance(), 0);
5     // the field hasn't changed after calling the getter method
6     assertEquals(10, bankAccount.getBalance(), 0);
7 }

```

Now the previous two mutants related to the getter method are effectively killed. However, does it really make sense to add such a test? Is the added test easily comprehensible? (Indeed, we left a comment to explain what we are trying to verify.) You probably agree that the answer is "no". Recall that frameworks such as mutation testing frameworks aim at helping the developers in finding possible problems in their code and in their tests. They are tools, they are not the final goal.

Even the move of the field initialization from the declaration to the constructor was done mainly to kill the additional mutants, not because there was a bug in our code. Of course, in that case, the survived mutants showed us that we did not have tests for the initial default value of the balance. However, an initial empty balance (i.e., = 0) did not necessarily require to be tested.

---

<sup>3</sup>Note that on the contrary the mutants performing a pre-increment and pre-decrement on the same line of code are effectively killed since they would make the tests using `getBalance` fail.

If we now have a look at the remaining survived mutants we see that they are all related to post-increment and post-decrement of the parameter `amount` in the `throw` statement and in the `balance += amount` statement:<sup>4</sup>

```

1 public void deposit(double amount) {
2     if (amount < 0) {
3         throw new IllegalArgumentException("Negative amount: " + amount);
4     }
5     balance += amount;
6 }
```

There is not much we can do to kill such mutants because after using `amount` in those statements we forget about them, so we could not kill them in our tests. They can be considered false positives.

Summarizing, if you want to enable all mutators, you should be prepared to analyze the results very carefully and decide whether it makes sense to adjust the code and tests (without decreasing readability and cleanliness of code and tests) and whether you're in the presence of false positives.

Finally, you should take into account that by enabling all mutators you also enable “experimental” mutators, which might also contain bugs.<sup>5</sup>

## 6.3 Further experiments with PIT

Let's experiment some more with PIT, by using it with the projects we have implemented in the previous chapters.

### 6.3.1 Tests and induction (part 2)

Let's start with the alternative Factorial example we saw in Chapter *TDD*, Section *Tests and induction (part 1)*.

We are pretty confident that no mutant will survive with the current implementation of `Factorial` and `FactorialTest` (remember that we have 100% code coverage and that we implemented that example with TDD). However, with default mutators we have a survived mutant:

- “replaced int return with 0 for factorial/example/Factorial::compute → SURVIVED”

related to the `return` statement performing the recursive call:

---

<sup>4</sup>Once again mutants related to increments and decrements (both pre- and post-) of the parameter `amount` in the `if` statement are killed since they would make at least one of our tests fail.

<sup>5</sup>Indeed the mutators we have just seen, called “Unary Operator Insertion”, are experimental ones, as documented here [https://pitest.org/quickstart/mutators/#EXPERIMENTAL\\_UOI](https://pitest.org/quickstart/mutators/#EXPERIMENTAL_UOI).

```
1 public int compute(int n) {  
2     if (n < 0)  
3         throw new IllegalArgumentException("Negative input: " + n);  
4     if (n == 0)  
5         return 1;  
6     return n * compute(n - 1);  
7 }
```

This is surprising, since breaking such a returned value should break the test `testInductiveCase`. Let's find out what is going on.



Recall that we anticipated in Section *Tests and induction (part 1)*, Chapter *TDD*, that there was a flaw in the test for the non-base case.

What such a mutation does should be clear. However, just for learning how to find the meaning of a PIT mutation, let's have a look at the “Console” view in Eclipse. We find the interesting section corresponding to the above survived mutant:

```
1 > org.pitest.mutationtest.engine.gregor.mutators.PrimitiveReturnsMutator  
2 >> Generated 2 Killed 1 (50%)  
3 > KILLED 1 SURVIVED 1 TIMED_OUT 0 NON_VIABLE 0  
4 > MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0  
5 > NO_COVERAGE 0
```

Let's look at the documentation of this mutator: <https://pitest.org/quickstart/mutators/>: “Replaces int, short, long, char, float and double return values with 0.” This means that this line

```
1 return n * compute(n - 1);
```

is mutated as follows:

```
1 return 0;
```

Thus, calling `compute` with an argument greater than 0 will always return 0. Now let's have a look at the relevant test:

```

1  @Test
2  public void testInductiveCase() {
3      assertEquals(5 * factorial.compute(4), factorial.compute(5));
4  }

```

We defined the expectation in terms of the recursive nature of the factorial. In Chapter [TDD](#), Section [Tests and induction \(part 1\)](#), we thought it would be a good idea since the factorial has a recursive definition. Since tests are specifications, it makes sense to make the test reflect this recursive nature. However, this means that, with the above mutation, `testInductiveCase` will execute `assertEquals(5 * 0, 0)` which will NOT fail, since both `compute(5)` and `compute(4)` will yield `0`. The mutant survives.

Note that if we apply PIT with all mutators to the first factorial implementation of Chapter [TDD](#), Section [The factorial example with TDD](#), we would not have such survived mutants. In that very first example, we did not use induction in the tests.

Now, when implementing a recursive method it is helpful to reason by induction. We first deal with the base case (or the base cases, depending on the function we are implementing), then we deal with the general (recursive) case. We assume that the recursive call, on a smaller input, (where “smaller” depends once again on the context) behaves correctly. This is known as the *induction hypothesis*. Relying on such an assumption, we use the returned value to complete the method. For example, for the `compute` method, we assume that the recursive call `compute(n-1)` correctly computes the factorial of `n-1` and we simply return the multiplication of `n` and the value returned by the recursive call. However, when writing a test, we cannot assume the induction hypothesis, since in the test we are not implementing that method: we are verifying the method that we are implementing.

Instead, we must assert the induction hypothesis as well:

```

1  @Test
2  public void testInductiveCase() {
3      // our specification assumes that factorial.compute(4)
4      // yields the correct value, i.e., 24
5      // thus we must assert this induction hypothesis
6      assertEquals(24, factorial.compute(4));
7      //...otherwise a mutant returning 0 will survive!
8      assertEquals(5 * factorial.compute(4), factorial.compute(5));
9  }

```

This will kill the survived mutant above and correctly specifies the recursive behavior of the factorial, asserting also the induction hypothesis.

Note how the mutation testing framework revealed the problem in our original test. In particular, enabling all the mutators allowed us to spot this testing problem.

We can now verify that if we enable all mutators, the survived mutants can be ignored.

## 6.3.2 Mutations and iterative solutions

Let's go on experimenting: at the end of In Chapter *TDD*, Section *Tests and induction (part 1)*, we suggested that the implementation of `compute` could be turned into an iterative implementation, without tail recursion.

For example, we could modify it as follows and verify that all tests are still green (and we still have 100% code coverage):<sup>6</sup>

```

1 public int compute(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException("Negative input: " + n);
4     if (n == 0)
5         return 1;
6     int result = 1;
7     while (n >= 2) {
8         result *= n--;
9     }
10    return result;
11 }
```

Default mutators generate no survived mutants. If we enable the stronger ones, we get a survived mutant, “removed conditional - replaced equality check with false”, that is, the one that turns:

```
1 if (n == 0)
```

into

```
1 if (false)
```

It is easy to verify that the case when `n == 0` is included in the rest of the method (the `while` loop). So we can remove that `if` statement:

---

<sup>6</sup>The reader will excuse the bad practice of directly modifying the passed parameter. This code is not meant to be definitive anyway.

```

1 public int compute(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException("Negative input: " + n);
4     int result = 1;
5     while (n >= 2) {
6         result *= n--;
7     }
8     return result;
9 }
```

This kills the survived mutant. Running PIT with all mutators enabled, with the new fixed version of `testInductiveCase` leads to no additional interesting survived mutants. The survived mutant “Substitute 2 with 1” shows that we could perform an additional iteration since it is safe to multiply by 1. However, we can consider our solution an optimization that avoids a useless loop iteration so we decide to ignore that survived mutant. In any case, when using all mutators, the mutation testing execution takes much longer than before. Since mutations also change the boundary of the loop and remove the loop’s body decrement operations, some tests against mutants terminate with a timeout (the loop would never end with mutants). Mutants that lead to timeout are not considered survived, so we are fine with that.

Let’s revert `testInductiveCase` so that it does not assert the induction hypothesis. Now many more mutants survive. In particular, the interesting one is “replaced int return with 0 for factorial/example/Factorial::compute → SURVIVED”, similar to what we saw in the previous section. Thus, despite the recursive or iterative implementation, our test needs to assert the induction hypothesis: mutations to the loop violate the assumption of the inductive step.

Let’s go back to the “fixed” version of `testInductiveCase`.

Thus, mutation testing in the presence of explicit loops tends to lead to timeouts. This also increases the time it takes to execute mutation testing, even with a small example like this one. We could refactor our factorial implementation to use Java 8 streams: this will allow us to get rid of tail recursion while keeping the iterative implementation:

```

1 import java.util.stream.IntStream;
2 ...
3 public int compute(int n) {
4     if (n < 0)
5         throw new IllegalArgumentException("Negative input: " + n);
6     return IntStream.rangeClosed(2, n).reduce(1, (x, y) -> x * y);
7 }
```

The tests are still green, with 100% code coverage. Now, no mutant survives with stronger mutators. With all mutators, we get some survived mutants that we can ignore. In any case, with this solution, the number of generated mutants is much lower than with the solution based on the `while` loop. Moreover, even with all mutators, we get rid of timeouts during PIT execution.

Just for experimenting, let's revert `testInductiveCase` once again so that it does not assert the induction hypothesis. Even with default mutators, we get survived mutant survive: the induction hypothesis really needs to be asserted.

### 6.3.3 Mutation testing on the MyStringUtils example

Let's experiment with mutation testing on our `MyStringUtils` class that we saw in Chapter [TDD](#), Section [A first example of TDD](#).

In our last refactoring, we removed the initial check `input.isEmpty` since we did not need that anymore. Let's try and put it back:

```
1 public String leftTrim(String input) {  
2     if (input == null || input.isEmpty())  
3         return input;  
4     ...  
5 }
```

Let's enable stronger mutators and run PIT. We have a survived mutant “removed conditional - replaced equality check with false → SURVIVED”. This proves that checking `input.isEmpty` is useless in this method.

### 6.3.4 Mutation testing on the Bank example

Let's experiment with mutation testing on our `Bank` and `BankAccount` classes. We start with the default mutators (so you might need to reset the corresponding property in Eclipse). We start from the project we implemented in Chapter [JUnit](#), Section [A first example](#).

You must remember that if you run PIT with a single test case class in a project where there are several SUT classes (and several test cases), then PIT will mutate not only the SUT of the single test case but all the main code classes. This will probably lead to several survived mutants: a single test case deals only with a single SUT, so mutants of other classes will survive. If you have a project with several classes you should run PIT with all the test cases. In this example, this can be achieved by right-clicking on the tests source folder (or on the whole project) and select **Run As → PIT Mutation Test**. Alternatively, if you want to concentrate on a single class and the corresponding test case, you will have to tweak the launch configuration, using the text field “Excluded classes”. For example, for `BankTest` we can use the exclusion `*BankAccount`, while for `BankAccountTest` we can use the exclusion `*Bank`. In both cases, test classes should also be explicitly excluded. In fact, by default, PIT launch configurations have `*Test` in “Excluded classes”. Summarizing, the “Excluded classes” should be `*BankAccount, *Test` and `*Bank, *Test`, respectively. Concentrating on a single class and excluding the other ones has the benefit to speed up the whole mutation process.

Let's concentrate on running PIT for `BankAccountTest`. We can verify that `BankAccount` enjoys 100% code coverage, but we'll see in a minute that our tests do not completely test the behavior of this class.

We have 3 survived mutants “changed conditional boundary” related to the boolean conditions `amount < 0` (in methods `deposit` and `withdraw`) and `balance - amount < 0` (in method `withdraw`). As we hinted in Chapter [JUnit](#), in the box *Have we now tested everything correctly?*, `BankAccountTest` does not have tests for the corner cases, that is, the boundary conditions `amount == 0` and `balance - amount == 0`. Thus, the corresponding mutants will survive.

These 3 additional tests cover the boundary conditions and will kill the above 3 mutants:

```

1  @Test
2  public void testDepositWhenAmountIsZeroShouldBeAllowed() {
3      BankAccount bankAccount = new BankAccount();
4      bankAccount.setBalance(INITIAL_BALANCE);
5      bankAccount.deposit(0);
6      assertEquals(INITIAL_BALANCE, bankAccount.getBalance(), 0);
7  }
8
9  @Test
10 public void testWithdrawWhenAmountIsZeroShouldBeAllowed() {
11     BankAccount bankAccount = new BankAccount();
12     bankAccount.setBalance(INITIAL_BALANCE);
13     bankAccount.withdraw(0);
14     assertEquals(INITIAL_BALANCE, bankAccount.getBalance(), 0);
15 }
16
17 @Test
18 public void testWithdrawWhenBalanceIsEqualToAmountShouldBeAllowed() {
19     BankAccount bankAccount = new BankAccount();
20     bankAccount.setBalance(INITIAL_BALANCE);
21     bankAccount.withdraw(INITIAL_BALANCE);
22     assertEquals(0, bankAccount.getBalance(), 0);
23 }
```

Stronger mutators do not generate new survived mutants, and all mutators generate the typical survived mutants that we decided to ignore as we saw in Section [Enabling additional mutators](#).



Note that with all mutators enabled, even for a simple class like `BankAccount`, we have more than 100 generated mutants.

Let’s now concentrate on `BankTest` (again, you might want to tweak the launch configuration as hinted above). With the default mutators only, we have two survived mutants:

- “removed call to testing/example/bank/BankAccount::setBalance → SURVIVED”

- “replaced boolean return with true for testing/example/bank/Bank::lambda\$0 → SURVIVED”

The first one is related to this method:

```

1 public int openNewBankAccount(double initialBalance) {
2     BankAccount newBankAccount = new BankAccount();
3     newBankAccount.setBalance(initialBalance);
4     bankAccounts.add(newBankAccount);
5     return newBankAccount.getId();
6 }
```

Indeed, in the test for `openNewBankAccount` we do not verify that the initial balance is set:

```

1 @Test
2 public void testOpenNewAccountShouldReturnAPositiveIdAndStoreTheAccount() {
3     int newAccountId = bank.openNewBankAccount(0);
4     assertTrue("Unexpected non positive id: " + newAccountId, newAccountId > 0);
5     assertEquals(newAccountId, bankAccounts.get(0).getId());
6 }
```

The mutant survives: if the call to `setBalance` is removed our test still succeeds.

Let's fix this by adding this assertion in the test (and we also pass an initial positive balance):

```

1 @Test
2 public void testOpenNewAccountShouldReturnAPositiveIdAndStoreTheAccount() {
3     int newAccountId = bank.openNewBankAccount(INITIAL_BALANCE);
4     assertTrue("Unexpected non positive id: " + newAccountId, newAccountId > 0);
5     BankAccount newAccount = bankAccounts.get(0);
6     assertEquals(newAccountId, newAccount.getId());
7     assertEquals(INITIAL_BALANCE, newAccount.getBalance(), 0);
8 }
```

The first mutant is now killed.

The other survived mutant is related to the lambda passed to `filter` in this method:

```

1 private BankAccount findBankAccountById(int bankAccountId) {
2     return bankAccounts.stream()
3         .filter(a -> a.getId() == bankAccountId)
4         .findFirst()
5         .orElseThrow(
6             () -> new NoSuchElementException
7             ("No account found with id: " + bankAccountId));
8 }
```

The mutated version is

```
1 ...filter(a -> true)...
```

Actually we were already aware of this problem in our tests, see Chapter [Code coverage](#), Section [How code coverage can help](#). In that section, we detected the problem in our tests because code coverage told us that one of the branches of the boolean expression was not covered. In that section we simulated what mutation testing does automatically: we removed the call to `filter` and our tests still succeeded (which corresponds to the mutant `filter(a -> true)`). We already know that to test correctly the code we must use a list with another bank account before the one that we want to find:

```

1 @Test
2 public void testDepositWhenAccountIsFoundShouldIncrementBalance() {
3     // setup
4     BankAccount another = createTestAccount(0);
5     bankAccounts.add(another);
6     BankAccount toBeFound = createTestAccount(INITIAL_BALANCE);
7     bankAccounts.add(toBeFound);
8     // exercise
9     bank.deposit(toBeFound.getId(), AMOUNT);
10    // verify
11    assertEquals(INITIAL_BALANCE+AMOUNT, toBeFound.getBalance(), 0);
12 }
```

Now all the mutants related to the `filter` operation are killed.

## 6.4 Narrowing mutations

As we saw in the introduction of this chapter and throughout the chapter, mutation testing requires a lot of time to execute, due to the steps of the mutation testing procedure. Thus, it is crucial to spend some time narrowing the scope of the mutation testing procedure.

In Section [\*Enabling additional mutators\*](#), we also saw that enabling all mutators might lead to a huge number of false positives, not to mention that enabling more mutators make the mutation testing procedure even much longer.

Finally, in Section [\*Mutation testing on the Bank example\*](#) we saw that it is better to focus on a specific SUT and its tests, instead of running the mutation testing procedure on the whole project. In more advanced projects, it might also be the case to exclude completely certain classes from mutation testing, possibly concentrating on the classes with an involved and important logic.

The features provided by Pitclipse that we used in this chapter are not as evolved as the features provided by the core project PIT. For example, despite the configuration mechanisms we have used in this chapter, with Pitclipse it is not possible to enable specific mutators (but only the predefined sets “Defaults”, “Stronger” and “All”). It is however possible to exclude specific classes (as we did in Section [\*Mutation testing on the Bank example\*](#)) and specific methods.

In Chapter [\*Maven\*](#), Section [\*Configuring the PIT Maven plugin\*](#), we will see how to exploit the advanced configuration features of PIT itself for narrowing the mutation testing procedure during a Maven build.

# 7. Maven

Up to now, we have always used Eclipse to compile and run tests. We have also used Eclipse to handle the project's dependencies, modifying the project's classpath.

When we needed additional libraries (that is JARs) we relied on Eclipse. If the library is part of Eclipse itself, like JUnit, that is quite easy. Otherwise, we had to manually download JARs from the Internet and add them to our projects in Eclipse by also setting the classpath. Adding JARs to the classpath is easy anyway with Eclipse. The hardest problem with the process of downloading JARs is dealing with *transitive dependencies*, that is, dependencies of dependencies (recursively).

Dealing with transitive dependencies might become a nightmare since they must be resolved manually. First, you need to get to know all the external JARs needed by the libraries you want to use. Then, you need to download them as well, until all dependencies are resolved and available in your project.

Up to now, we were lucky since the external JARs had no further dependencies (or if they had, 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 that we needed.

We chose to add the JARs directly to the project itself. When working on a team, the fact that external libraries are part of the project relieves other developers from performing all the above manual steps, again and again, each time they want to set up the development environment. However, having external libraries as part of your codebase is not ideal either.

In Chapter [JUnit](#), Section [Keeping test code separate from main code](#), we showed the importance of keeping the main code separate from the test code and that this separation must concern also the corresponding dependencies. In Section [Export a runnable JAR](#) we also saw how to export a JAR with all the dependencies. After having separated 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 is self-contained. However, if we want to distribute a library, including dependencies in the JAR might be overwhelming. For libraries, we should then document the needed dependencies, since the users of our JAR will have to add to the classpath also the dependencies of our library. Again, this is cumbersome for the users of our library.

In any case, dealing with dependencies manually is an effort and it's error-prone! Not to mention that it does not scale to complex programs with lots of dependencies.

We will soon deal with “Continuous Integration servers”, Chapter [Continuous Integration](#), which automatically

- Compile the whole program
- Run all the tests
- Keep track of possible problems with reports about:
  - Compilation errors
  - Failed tests
  - Code coverage
  - Code quality

Continuous Integration relies on a *build automation* tool, that is a “command line” tool that can perform all the above tasks automatically. In particular, such a tool should mimic what we usually do in Eclipse but automatically:

- Download all the dependencies of the project
- Compile the whole application
- Run all the tests
- Generate reports

Build automation and Continuous Integration is a requirement even for middle-size projects. When there are many tests and some of them are integration tests, which require much more time to run than unit tests, during a TDD cycle, a developer executes only the unit tests that concern the current SUT. A single developer can also run all the unit tests of the application, since they are fast, while all the other long-running tests are executed by the Continuous Integration server.

Maven comes to help us in all the above issues, aiming at making all the above processes as automatic as possible, starting from a configuration file defining both the dependencies of the project and the build automation infrastructure.

## 7.1 Introduction to Maven

Maven is a tool for building and managing Java-based projects. Maven aims at simplifying the build process and dependency management. In particular, it focuses on applying patterns to a project’s build infrastructure and on fostering best practices in the building process. The result should be an easy to reproduce building process that makes the following steps automatic:

- Download all the dependencies of the project
- Compile the whole application
- Run all the tests
- Generate reports

Maven also fosters a development process based on automatic tests: unit tests are part of the normal Maven build lifecycle. Moreover, testing best practices are also taken into consideration: test code is by default kept in a separate source folder. If test case classes follow a standard naming convention, they are automatically discovered and executed during a standard Maven build.

Indeed, Maven relies on **Convention over Configuration** mechanisms. If you follow these conventions, there is not much to manually configure. These conventions, besides the above-mentioned naming conventions on the tests, are applied also to the project's directory structure (which we'll see in the rest of this chapter). Once you learned these conventions, you can easily navigate any other project that uses Maven.

Instead of writing commands for compiling the sources and run the tests, just follow the conventions. Maven will be able to run the corresponding commands for compiling and running the tests.

Of course, in case these conventions cannot be applied to a project, customization points are available to tweak the build process. However, Maven is not famous for its flexibility. In particular, the overall build process is structured in specific *lifecycles* and *phases*, which are rigid and cannot be modified. The basic idea is that you can hook specific tasks to the lifecycles' phases but not much more. This rigidity has a huge benefit: once you get familiar with Maven's lifecycles and phases, the build will work automatically, and configuring new Maven projects will be much easier. Moreover, you will not have to take care of the build process. Thus, if you follow Maven's rules, there should be no bad surprises. On the contrary, when using more flexible build systems, you will have full control of the build process. You will be responsible for the overall process and you will have to deal with problems that come from this flexibility.

The extensibility of Maven relies on plugins. Some of such plugins are part of the Maven distribution and are enabled by default. Other plugins can be enabled in the project. There are tons of such plugins for the most common tasks for Java projects.

Maven relies on software repositories for plugins and Java JARs. In particular, there is a default central repository, which contains Maven plugins and most Java libraries. As we will see, once dependencies are specified with certain *coordinates* in the configuration file, they will be downloaded automatically by Maven from such repositories.

A Maven project must be described and configured in a single file, which must be placed in the root folder of the project, named `pom.xml`. This is the XML representation of the **Project Object Model (POM)**.<sup>1</sup>

We will now see the main features of Maven, with several examples. We will then use Maven in the rest of the book.

## 7.2 Maven installation

Maven is a command-line Java program. It is available for any platform that can execute the JVM. It can be downloaded from <https://maven.apache.org/download.cgi>. It is just a matter of extracting

---

<sup>1</sup>The complete structure of the POM can be found here: <http://maven.apache.org/pom.html>.

the archive somewhere in your system and of setting the system PATH accordingly.

Alternatively, you may want to use SdkMan also for installing Maven (see Chapter *Introduction*, Section *Install the JDK*).

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 on 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 the M2Eclipse project are summarized on its main web page:

- Launching Maven builds from within Eclipse
- Dependency management for Eclipse build path based on the `pom.xml`
- Resolving Maven dependencies from the Eclipse workspace without installing to the local Maven repository (this will be clearer later in this chapter)
- Automatic downloading of the required dependencies from the remote Maven repositories
- Wizards for creating new Maven projects, `pom.xml` and to enable Maven support on plain Java project



Maven needs Internet access to download its artifacts, like plugins and dependencies. All the downloaded artifacts will be cached in the local file system, but the first time you use Maven, be prepared to wait until everything is downloaded.

## 7.3 Let's get started with a Maven archetype

**Maven archetypes** are template projects, which are useful when you create new projects since they provide the typical structure for that kind of project. This way, the initial structure of the Maven project and its POM are created through the archetype and you do not need to create the typical initial configuration over and over again when you start a new project.

Typical examples of archetypes are “simple Java project”, “simple Java web application”, etc. We will often use the “maven-archetype-quickstart”, which creates a simple Java Maven project.

### 7.3.1 Create a Maven project from the command line

If you installed the Maven command line binary, you can create a Java Maven project with the following command (lines are split using \):

```
1 mvn archetype:generate \
2   -DarchetypeGroupId=org.apache.maven.archetypes \
3   -DarchetypeArtifactId=maven-archetype-quickstart \
4   -DarchetypeVersion=1.1 \
5   -DgroupId=com.mycompany.app \
6   -DartifactId=my-app \
7   -DinteractiveMode=false
```

The values of the properties `archetypeGroupId`, `archetypeArtifactId`, and `archetypeVersion` are related to the archetype we want to use. The values of the properties `groupId` and `artifactId` are related to your project that you are about to create (we'll talk about these properties later in Section [Maven coordinates \(GAV\)](#)).



For the didactic purpose, we use an old version of this archetype 1.1 (the most recent version is 1.4). This allows us to start from a simple project that we gradually improve.

The above command will first download a few things from the Internet, and then create the following directory and file structure:

```
1 └── my-app
2     ├── pom.xml
3     └── src
4         ├── main
5             ├── java
6                 └── com
7                     └── mycompany
8                         └── app
9                             └── App.java
10    └── test
11        ├── java
12            └── com
13                └── mycompany
14                    └── app
15                        └── AppTest.java
```

We will get into the details of the structure of the created project in the next sections.

### 7.3.2 Import a Maven project in Eclipse

As we anticipated, a Maven project is configured in the file `pom.xml`. Of course, there is no trace of Eclipse project metadata, thus, Eclipse is not able to directly open such a Maven project. However,

thanks to the M2E plugin, Eclipse can import the Maven project and automatically derive and create its own project metadata.

Select **File → Import... → Maven → Existing Maven Projects** and specify the root path of the project created through the archetype above. The specified path can also contain several Maven projects and Eclipse can import them all. Once the root path is specified, detected projects are shown for selection. Make sure the one we created is selected and press “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.

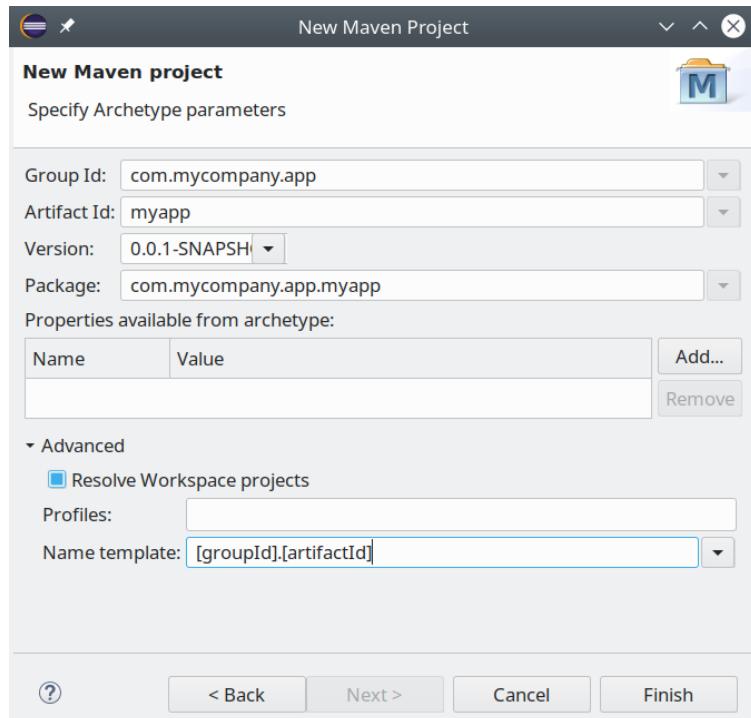
Note that the name of the Eclipse project, by default, will be the same as the `artifactId` specified when creating the project through the archetype.

### 7.3.3 Create a Maven project from Eclipse using an archetype

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

Select **File → New Project... → Maven → Maven Project** and choose a possibly different location for the created project. Make sure that the checkbox for skipping archetype selection is not checked and press “Next”. The very first time, the huge list of available archetypes has to be downloaded from the Internet. Use the “Filter:” box and type `maven-archetype-quickstart`. Make sure to uncheck “Show the last version of Archetype only” and choose the one with “Group Id” `org.apache.maven.archetypes` and “Version” `1.1` (the same we used when we created the project from the command line). Then provide the `groupId` and `artifactId`. If you imported in Eclipse the project created from the command line you must provide a different `artifactId`. In fact, as seen in the previous section, by default, M2E creates an Eclipse project with the same name as the `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.



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

## 7.4 Structure of a Maven project

We saw that the simple Java project created with the archetype already separates the source folder for the main code (`src/main/java`) from the one for the tests (`src/test/java`). These source folders are the default ones, thus the `pom.xml` does not specify them (remember that Maven relies on conventions over configurations).

Eclipse extracts this (default) information from the Maven project and configures the two folders above as Eclipse source folders. Moreover, the source folder for tests is automatically marked as a test source folder. Recall that in Chapter [JUnit](#) in Section [Keeping test code separate from main code](#), we had instead to configure this manually. This is one of the benefits of Maven, which is already “test-oriented”.

What's in `src/main/java` will end up in the final package of the application, while what's in `src/test/java` is test code and will be excluded from the final package.

The `pom.xml` must be placed in the root folder of what is meant to contain a Maven project.

## 7.5 Java settings

At the time of writing, by default, a Maven project is automatically configured to be compatible with Java 5. This default is again automatically used by Eclipse to configure the project metadata. Both in the imported project and the one created from 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 features that are present in Java 5.

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

Thus, 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 compilation settings will be used. This means that if we use Java 8 lambdas in our project, after changing the Eclipse settings, the project will compile in Eclipse, but it won’t compile with Maven.

The solution is to do the other way round. Thus, we first change the `pom.xml` in order to enable Java 8 compiler level (recall that we are using JDK 8 already) by adding the following two properties in the `pom.xml` file (we’ll see Maven properties in more details in Section [Properties](#)):

```
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>
```

And then we force Eclipse to update the project metadata from the changed `pom.xml`: right-click on the project and select **Maven** → **Update Project...**, in the dialog make sure that the project you want to update is selected and that the checkbox “Update project configuration from `pom.xml`” is selected. You will see that Eclipse updated the metadata of the project by setting the Java compilation level to 8, JavaSE-1.8.

## 7.6 Dependencies

One of the main features of Maven is its management of dependencies. We have already seen that when running Maven from the command line, Maven automatically downloads the artifacts that it needs to execute, that is, the JARs of Maven plugins. Besides that, Maven also automatically downloads dependencies specified in the `pom.xml`, that is, the Java libraries that your project needs to compile and run the tests.

Both plugins and dependencies are stored in Maven repositories, which are basically websites that serve Maven artifacts like JARs. By default, Maven will use the main repository, **Maven Central**, which stores most of all the possible Java JARs (additional repositories can be specified in the `pom.xml`, in case).

As mentioned before, Maven needs an Internet connection. The downloaded artifacts are cached in the local disk, by default in the home directory in the subdirectory `.m2`. The same JARs will not be downloaded again, if already cached. However, the first time you'll have to wait for all dependencies to be downloaded.

This implies that there must be a way to uniquely identify Maven plugins and jars and to select a specific version of these artifacts, as we will see in the next section.

The other important feature is that Maven automatically handles the transitivity of dependencies. In your project you only need to specify your first level dependencies and Maven will automatically resolve possible dependencies of dependencies transitively.

## 7.6.1 Maven coordinates (GAV)

Every Maven artifact is univocally determined by its coordinates, so-called **GAV**: `groupId, artifactId` and `version`.

The `groupId` identifies an organization or a project, for example, `org.mockito`, `org.apache.maven.plugins`, etc. The `artifactId` identifies that specific artifact within the organization or project, for example, `mockito-core`, `maven-compiler-plugin`, etc. The `version` identifies the specific version of that artifact (within that group).

These coordinates must always be specified for the Maven plugins, for the project's dependencies and your projects.

Concerning the `groupId`, usually, the Java convention for naming packages is used, <https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html>. For example, the artifacts of the Apache Software Foundation have the `groupId` that starts with `org.apache`. The convention is not always followed: JUnit itself simply uses `junit` as its artifacts `groupId`.

## 7.6.2 Dependencies in the POM

The dependencies of the project are specified in the `pom.xml` in the root level section `dependencies`, each one in a separate `dependency` element. Each such element must provide the coordinates and the `scope` of the dependency.

The scope has an impact on the classpath where the dependency will be available and on the transitivity of the dependency.

By default, the scope is set to `compile`: the dependency will be available in all classpaths of a project and it will be propagated to dependent projects. This means that the dependency will be a transitive dependency of your project. The scope `test` indicates a dependency that is not required for the

application itself, but only for the tests: it will be only available for the test compilation and execution phases. Moreover, this scope is not transitive.

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

```
1 <dependencies>
2   <dependency>
3     <groupId>junit</groupId>
4     <artifactId>junit</artifactId>
5     <version>3.8.1</version>
6     <scope>test</scope>
7   </dependency>
8 </dependencies>
```

Thus, this project is using JUnit 3.8.1 for testing.<sup>2</sup>

Recall that in Chapter [JUnit](#), Section [Keeping test code separate from main code](#), we configured the Eclipse project's classpath so that some dependencies were available only for tests. In Maven this is handled through the above concept of scope.

The two above-mentioned scopes are the only ones we'll need for the moment. For further and complete details about the available scopes, we refer to <http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>.

### 7.6.3 SNAPSHOT

If we use dependencies with a version that ends with `-SNAPSHOT` then we are using versions still under development: That version has not been officially released. Its implementation is not to be considered final, nor necessarily stable. Developers can release several different snapshots in a single day. Maven has a period after which it checks if there are 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`, since we are still working on their implementation. We could already deploy the snapshot version to a remote repository (including the Maven central repository).

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

---

<sup>2</sup>We'll switch to JUnit 4 in a moment.

## 7.6.4 Version ranges

Maven also supports version ranges for dependencies. For example, this allows you to automatically build your project when a new version of a dependency is released, provided the new version is in the range you specified.

However, to keep the build reproducible, you should avoid using version ranges. Your project might suddenly start to break (i.e., it doesn't compile anymore or tests start to fail) due to a new version of one of the dependencies. Similarly, using snapshots for dependencies should be based on a very well-motivated reason.

In Section [Updating dependencies](#) we will use a Maven plugin that automatically checks for new versions of our dependencies and possibly automatically updates them in the POM.

## 7.7 Eclipse m2e

The Maven integration in Eclipse provided by m2e consists of a bridge between Maven's dependency management and the Eclipse compiler.

This means that when you are developing a Maven project in Eclipse you are still using the Eclipse compiler and its automatic building mechanisms. The project is still compiled by Eclipse, not by Maven. However, Eclipse's project classpaths are automatically computed by Eclipse using the Maven project's dependencies.

In Chapter [JUnit](#), 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's classpath accordingly. When we're 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 such dependencies.

For example, with the `pom.xml` of the project 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 in the `pom.xml` JUnit is specified as a "test" dependency, the JAR is colored as an Eclipse test dependency (see Chapter [JUnit](#), Section [Keeping test code separate from main code](#)).

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 triggers the update of the classpath, after downloading the new JARs, if not already cached. As an Eclipse plugin, m2e provides a dedicated multi-tab editor for the `pom.xml` file.

For the moment, let's use the "pom.xml" tab of the editor, which allows us to directly modify the 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>
```



The XML editor provides context-sensitive code completion, so just inserting <, a character, and invoking the content assist will give you some suggestions. Choose an XML tag and the corresponding closing tag will be automatically added.

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 see also `hamcrest-core-1.3.jar`, which, as we learned in Chapter [JUnit](#), Section [Using Hamcrest matchers](#), is a transitive dependency of JUnit 4 (that was not the case for the older JUnit 3). As a transitive dependency of a test dependency, it’s also colored as a test dependency.

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

In the m2e preferences ([Preferences → Maven](#)) you can specify whether sources of the JARs should be automatically downloaded, when a dependency is downloaded. This allows you to have Java sources of the dependencies available to be inspected by expanding the “Maven Dependencies” node or by navigating to a Java type of a dependency. If this preference is not set, the first time you try to inspect the type of a dependency the sources will be automatically downloaded anyway (so you’ll experience a delay the first time you open a Java type of a dependency).



## Output folders in a Maven project

Another difference between a standard Eclipse project and an Eclipse Maven project is how output folders are managed. By default, in the former, the class files are generated in the `bin` directory. In the latter, following the Maven convention, the class files of the main code are generated in the `target/classes` directory and the class files of the test code are generated in the `target/test-classes` directory.

## 7.8 Properties

We have already used the properties section when we switched the Java compilation level to version 8 (Section [Java settings](#)).

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

```
1 <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 you configure the plugin's behavior by specifying a value for that property in the `<properties>` section.

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 precedence. For example, for the Maven compiler plugin, `<maven.compiler.source>` is the user property corresponding to the parameter `<source>`.<sup>3</sup>

Once a property is defined, either directly in the `<properties>` section or implicitly by some default Maven configurations, it can be used in the rest of the POM with the syntax  `${key}` . Referring to a property that has never been defined results in an error reported by Maven.

Using properties instead of hardcoded values in the POM has the same benefit as using variables or constants in a Java program: you get more consistency, you avoid copy and paste and allow for future changes.

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

```
1 <properties>
2 ...
3   <junit.version>4.13</junit.version>
4 </properties>
5
6 <dependencies>
7   <dependency>
8     <groupId>junit</groupId>
9     <artifactId>junit</artifactId>
10    <version>${junit.version}</version>
11    <scope>test</scope>
12  </dependency>
13 </dependencies>
```

---

<sup>3</sup>Such information can be found on the plugin's home page, for example, for the Maven Compiler plugin you can have a look at <https://maven.apache.org/plugins/maven-compiler-plugin/compile-mojo.html>.

Properties can also be overridden dynamically when running a Maven build on the command line (see Section [Build with Maven](#)) with the standard Java syntax

```
1 -Dkey=value
```

If you need several dependencies with the same groupId you likely want all those artifacts with the same version. Using properties for the versions fosters consistency and allows you to easily switch to another version by changing only one property. Using properties allows your Maven build to be parametric.

Maven itself defines several properties for the paths of source folders, output folders, resource folders, etc. These properties can be used to refer to folders relative to the above standard folders.

These are few examples of the available properties:

`${project.basedir}`

the root folder of the project, that is, the location of the `pom.xml` file

`${project.build.directory}`

by default the target folder, where all artifacts are generated.

`${project.build.outputDirectory}`

by default the `target/classes` folder, where all main Java code is compiled.

`${project.build.sourceDirectory}`

by default the `src/main/java` folder.

## 7.9 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 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 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 (see the box [Output folders in a Maven project](#)), that is, `target/classes` and `target/test-classes`, respectively:

```

1 target
2   └─ classes
3     |   └─ afile.txt
4     └─ com
5       └─ mycompany
6         └─ app
7           └─ myapp
8             └─ App.class
9   └─ test-classes
10    └─ afile.txt
11    └─ com
12      └─ mycompany
13        └─ app
14          └─ myapp
15            └─ 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.

### 7.9.1 Layout of an Eclipse project

Note that when Eclipse updates the classpath of the project (the file `.classpath`), e.g., as the result of **Maven → Update Project...**, new elements are appended to the current classpath settings. The classpath contains also entries like the “Maven Dependencies”.

Visually, in the “Package Explorer” or the “Project Explorer”, the elements of the project are shown in the order they are present in the classpath.

Updating the project after adding the resources folder will then show the source folders for Java code, the Maven dependencies and then the source folders of the resources.

- ▶ src/main/java
- ▶ src/test/java
- ▶ JRE System Library [JavaSE-1.8]
- ▶ Maven Dependencies
- ▶ src/main/resources
- ▶ src/test/resources

Layout after updating the project

This order can be changed, using the property of the project “Java Build Path”: the tab “Order and Export” allows you to move up or down the source folders.

- ▶ src/main/java
- ▶ src/main/resources
- ▶ src/test/java
- ▶ src/test/resources
- ▶ JRE System Library [JavaSE-1.8]
- ▶ Maven Dependencies

Layout after changing the order

This is useful if you prefer to have source folders organized in the Eclipse view.

## 7.9.2 Create a Maven project from Eclipse without an archetype

You can also create a Maven project without an archetype directly from Eclipse. Again, Eclipse will also automatically import the project in the workspace and create its metadata, as usual.

Select **File → New Project... → Maven → Maven Project** and choose a possibly 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` and press “Finish”.

This time the `pom.xml` is minimal and contains no dependency information.

Eclipse also already creates the source folders for main and test code and also for main and test resources. All such folders are empty.

## 7.10 The Eclipse m2e pom.xml multi-tab editor

As already sketched above, the Eclipse m2e editor for the `pom.xml` file is a multi-tab editor.

Besides allowing you to directly edit the XML contents, each tab allows you to visually specify some contents of the `pom.xml` and, when saving, it will automatically modify the XML contents accordingly. Modifying the XML file automatically updates the visual contents of the other tabs of the editor, unless there are validation errors.

The other tabs are now briefly described:

### Overview

deals with the main sections of the `pom.xml` including the GAV of the current project, properties (Section *Properties*), parent POM and modules (see Section *Parent and modules*) and other information of the project.

### Dependencies

allows you to see, modify, manage and add dependencies without manually editing the XML file.

### Dependency Hierarchy

shows the possible transitive dependencies of your first-level dependencies. For example, when using JUnit 4.13 we can see in this tab that it transitively depends on Hamcrest Core.

## Effective POM

shows the final effective POM contents as derived by Maven (see Section [Effective POM](#)).

## 7.11 The bank example as a Maven project

Let's create a Maven project where we'll recreate the bank example we saw at the end of Chapter [JUnit](#), in particular, the one using AssertJ and Log4j2.

The procedure for creating the project is the same as the one we have already seen, but skipping the archetype. We use `com.example` and `bank-example` for the group and the artifact, respectively.

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

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.example</groupId>
4   <artifactId>bank-example</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6   <properties>
7     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
8     <maven.compiler.source>1.8</maven.compiler.source>
9     <maven.compiler.target>1.8</maven.compiler.target>
10    </properties>
11 </project>
```

Then, we update the Eclipse project with the changed Maven configuration (**Maven → Update Project...**).

From the original bank example, let's copy the whole package `testing.bank.example` contents into the new project source folder `src/main/java`.

The code does not compile since it misses Log4j. Now that we can use the dependency management of Maven, we don't have to download manually any JAR and update the classpath: we just specify the Maven GAV coordinates for Log4j. We could use the <https://search.maven.org/> web page for looking up the coordinates or check whether the project's website provides such coordinates. In this case, we found them here: <https://logging.apache.org/log4j/2.x/maven-artifacts.html>. Let's add them to our POM:

```
1 <dependencies>
2   <dependency>
3     <groupId>org.apache.logging.log4j</groupId>
4     <artifactId>log4j-api</artifactId>
5     <version>2.13.0</version>
6   </dependency>
7   <dependency>
8     <groupId>org.apache.logging.log4j</groupId>
9     <artifactId>log4j-core</artifactId>
10    <version>2.13.0</version>
11  </dependency>
12 </dependencies>
```

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 is available to this project. We find the Maven coordinates for AssertJ from its home page <https://assertj.github.io/doc/#assertj-core-qucik-start> and we already know how to add JUnit:

```
1 <dependency>
2   <groupId>junit</groupId>
3   <artifactId>junit</artifactId>
4   <version>4.13</version>
5   <scope>test</scope>
6 </dependency>
7 <dependency>
8   <groupId>org.assertj</groupId>
9   <artifactId>assertj-core</artifactId>
10  <version>3.15.0</version>
11  <scope>test</scope>
12 </dependency>
```

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



In the original example, we used to have a custom Log4j configuration for test code, where the logging level was set to DEBUG. This is a candidate for being a test resource file: copy the `log4j2.xml` file from the `tests` folder of the original example to the folder `src/test/resources` of this project. Run the tests and now also debug logs appear in the console.

Note that the test code and the test dependencies in the Eclipse projects are automatically configured by only specifying the Maven dependencies with the correct scope.

Now compare this with the cumbersome manual procedure of downloading JARs, adding them to the project, configure the classpath, configure the separation of main code from test code that we did in Chapter [JUnit](#).

## 7.12 Build with Maven

Up to now, by relying on Eclipse m2e, we have 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.



The build directory must be excluded/ignored when using a Version Control System (see Chapter [Git](#)).

To fully understand the process of building with Maven, it is crucial to get acquainted with the Maven lifecycles.

### 7.12.1 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 running a Maven build you typically specify the name of the phase(s) to execute, which will be taken from the corresponding lifecycles.



Custom lifecycles can be implemented and existing lifecycles can be extended as well. This is usually required only if you need to implement a new packaging type. We will not deal with these extended mechanisms in this book.

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

```
1 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`.



Unfortunately, lifecycles, phases, and goals might have the same name.

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.<sup>4</sup>

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.

---

<sup>4</sup>See [https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle\\_Reference](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference) for the full reference.

**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 detail 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 the 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.<sup>5</sup>

The bindings can be inspected by running this Maven command

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

For example,

```
1 mvn help:describe -Dcmd=clean
2
3 [INFO] 'clean' is a phase within the 'clean' lifecycle, which has the following phas\
4 es:
5 * pre-clean: Not defined
6 * clean: org.apache.maven.plugins:maven-clean-plugin:2.5:clean
7 * 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.



Maven commands must be run on the folder where the `pom.xml`. Alternatively, the location of the `pom.xml` can be specified with the command line option `--file` or `-f`.

---

<sup>5</sup>See <http://maven.apache.org/ref/current/maven-core/default-bindings.html>.

## 7.12.2 Run Maven from the command line

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:

```
1 mvn clean
2
3 [INFO] Scanning for projects...
4 [INFO]
5 [INFO] ---< com.example:bank-example >---
6 [INFO]
7 [INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ bank-example ---
8 [INFO] Deleting <full path>/bank-example/target
9 [INFO] -----
10 [INFO] BUILD SUCCESS
```



When invoking Maven from the command line on a project that is currently opened in Eclipse, it is better to make sure Eclipse synchronizes with the file system, e.g., by refreshing the project and by performing a clean build (**Project → Clean...**).

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`):

```
1 mvn clean compile
2
3 [INFO] Scanning for projects...
4 [INFO]
5 [INFO] ---< com.example:bank-example >---
6 [INFO] Building bank-example 0.0.1-SNAPSHOT
7 [INFO] ---[ jar ]---
8 [INFO]
9 [INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ bank-example ---
10 [INFO] Deleting <full path>/bank-example/target
11 [INFO]
12 [INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ bank-example -\
13 --
14 [INFO] Using 'UTF-8' encoding to copy filtered resources.
```

```
15 [INFO] Copying 1 resource
16 [INFO]
17 [INFO] --- maven-compiler-plugin:3.1:compile (@ bank-example ---
18 [INFO] Changes detected - recompiling the module!
19 [INFO] Compiling 3 source files to <full path>/bank-maven-example/target/classes
```

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`.



When Java sources are compiled, Maven uses the default Java compiler of 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 and it 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/maven-compiler-plugin/non-javac-compilers.html>.

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.

```
1 mvn test
2
3 [INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ bank-example -\
4 --
5 [INFO] Using 'UTF-8' encoding to copy filtered resources.
6 [INFO] Copying 1 resource
7 [INFO]
8 [INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ bank-example ---
9 [INFO] Nothing to compile - all classes are up to date
10 [INFO]
11 [INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ bank-e\
12 xample ---
13 [INFO] Using 'UTF-8' encoding to copy filtered resources.
14 [INFO] Copying 1 resource
15 [INFO]
16 [INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ bank-exAMPL\
17 e ---
18 [INFO] Changes detected - recompiling the module!
19 [INFO] Compiling 2 source files to <full path>/bank-example/target/test-classes
20 [INFO]
21 [INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ bank-example ---
22 [INFO] Surefire report directory: <full path>/bank-example/target/surefire-reports
23
24 -----
25 T E S T S
26 -----
27 Running testing.example.bank.BankTest
28 2018-10-12 14:31:22,970 [main] DEBUG testing.example.bank.Bank - Success: withdraw(1\
29 , 5.00)
30 2018-10-12 14:31:23,003 [main] INFO testing.example.bank.Bank - New account opened \
31 with id: 2
32 2018-10-12 14:31:23,017 [main] DEBUG testing.example.bank.Bank - Success: deposit(3, \
33 5.00)
34 Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.378 sec
35 Running testing.example.bank.BankAccountTest
36 Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
37
38 Results :
39
40 Tests run: 12, Failures: 0, Errors: 0, Skipped: 0
41
42 [INFO] BUILD SUCCESS
```



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

### 7.12.3 Run Maven goals

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

```
1 <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.

```
1 <groupId>:<artifactId>:<goal>
```

For the standard Maven plugins<sup>6</sup>, the group can be omitted:

```
1 <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

```
1 <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`.<sup>7</sup>

This is based on a convention for naming Maven plugins:

---

<sup>6</sup>that is, for plugins with groupIds of `org.apache.maven.plugins` or `org.codehaus.mojo`.

<sup>7</sup>See <https://maven.apache.org/guides/introduction/introduction-to-plugin-prefix-mapping.html> for further details.

**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 on the 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`):

```
1 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`<sup>8</sup>, 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 website will be generated by default in the `target/site` directory since such web contents are meant to be included in the website generated during the `site` lifecycle. 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`):

---

<sup>8</sup>This parses the generated `TEST-*.xml` files under `target/surefire-reports` (generated by the `maven-surefire-plugin` that we have already seen) and renders them as a web site.

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

## 7.12.4 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 (see Section [Maven profiles](#)) and other configurations. Remember that this is an Eclipse run configuration, and, as such, once created it can be stored in the project (see Chapter [Eclipse](#), Section [Eclipse Run configurations](#)).

The output of the build will be shown in the “Console” view.



The label “Goals” in the launch configuration is meant both for goals and phases, and, in general, further options to be passed to Maven can be specified there, including values for properties with the syntax `-D<key>=<value>`.

## 7.12.5 Offline mode

A Maven build can be executed in **offline** mode, with the command-line option `-o`, `--offline` or, in Eclipse, by selecting the corresponding checkbox in the launch configuration.

This way, Maven will not contact any remote repository. This mode is suitable when there is no Internet connection. Of course, this assumes that all the dependencies and plugins are already available in the local repository.

Before starting working offline, you can use the following command

```
1 mvn dependency:go-offline
```

which ensures that all the dependencies and plugins are downloaded in the local repository.

## 7.12.6 Updating dependencies

We have seen in Section [Version ranges](#) that it is better to have full control on the versions of the dependencies of a Maven project. However, we would like to have an automatic means to check whether a new version of one of our dependencies is available, without manually checking the corresponding websites.

This mechanism is provided by the plugin `org.codehaus.mojo:versions-maven-plugin`.

Since this plugin has a “standard” groupId we can call its goals from the command line using the short syntax.

For example, by running this command from the Maven bank example project, we will check whether there is a new release for our dependencies (at the time of writing, all dependencies could be updated):

```
1 mvn versions:display-dependency-updates
```

We can have the plugin automatically update the POM with newer versions by running

```
1 mvn versions:use-latest-versions
```

If the version of a dependency is stored in a property (see Section [Properties](#)), for the update, we use the goal `versions:update-properties`.

When updating the POM, a backup file is also generated, `pom.xml.versionsBackup`.

## 7.13 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 (Section [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.

### 7.13.1 Packaging “jar”

The default, if not specified, is `jar`. This is explicit in the projects created starting from the archetype “simple Java project”. 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.

## 7.13.2 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 several sub-projects (Maven modules) together.

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 (we will also see an example later in this chapter).

## 7.14 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:

```
1 <parent>
2   <groupId>...</groupId>
3   <artifactId>...</artifactId>
4   <version>...</version>
5   <relativePath>...</relativePath> <!-- optional -->
6 </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.,

```
1 <parent>
2   <groupId>...</groupId>
3   <artifactId>...</artifactId>
4   <version>...</version>
5   <relativePath>../my.parent/pom.xml</relativePath>
6 </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

```
1 <parent>
2   <groupId>com.mycompany.app.</groupId>
3   <artifactId>my-app-parent</artifactId>
4   <version>1.0.0-SNAPSHOT</version>
5 </parent>
6 <groupId>com.mycompany.app</groupId>
7 <artifactId>my-app</artifactId>
8 <version>1.0.0-SNAPSHOT</version>
```

One can simply write:

```
1 <parent>
2   <groupId>com.mycompany.app.</groupId>
3   <artifactId>my-app-parent</artifactId>
4   <version>1.0.0-SNAPSHOT</version>
5 </parent>
6 <artifactId>my-app</artifactId>
```



Typically you use a parent POM that is a project of yours. However, this is not always the case. You can use any POM as the parent POM, by using its Maven coordinates, provided it is available from a Maven repository (e.g., by default from the Maven central repository). For example, Spring Boot, <https://spring.io/projects/spring-boot>, a Java framework for web application development, provides a parent POM with many sensible defaults for configurations, plugins, and dependency management. Typically, a project that uses Spring Boot uses the Maven artifact `spring-boot-starter-parent` as the parent POM.

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 `Object` class 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 into 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 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 has in common the configuration of a **Parent pom**.



POM inheritance is a single inheritance mechanism: you can specify one single parent POM.

### 7.14.1 Let's create a parent project and a module project

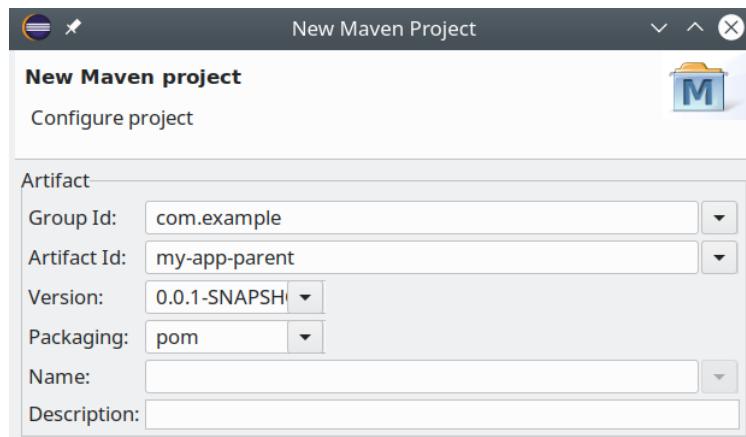
We will now create a parent project and two module projects. For simplicity, the module projects will be in two subfolders of the parent project folder.



The concept of the “Maven module” does not necessarily correspond to the concept of the “Java module” introduced in Java 9. In particular, in this book, we will not deal with Java 9 modules.

Select **File → New Project... → Maven → Maven Project** and choose a possibly 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:

```
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 </project>
```

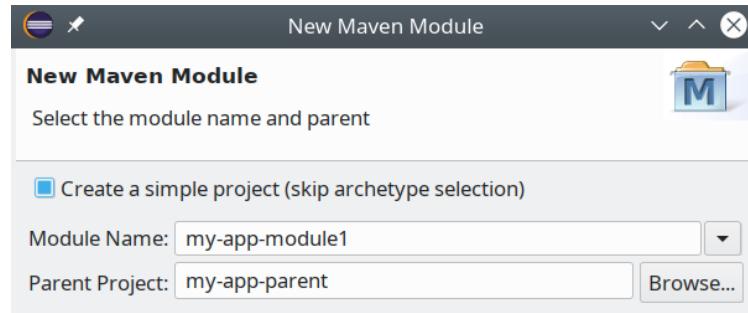
The parent project also contains the empty folder `src/site`; since we won't create a site for our Maven projects, you can delete that folder.

Now we create the first module project.

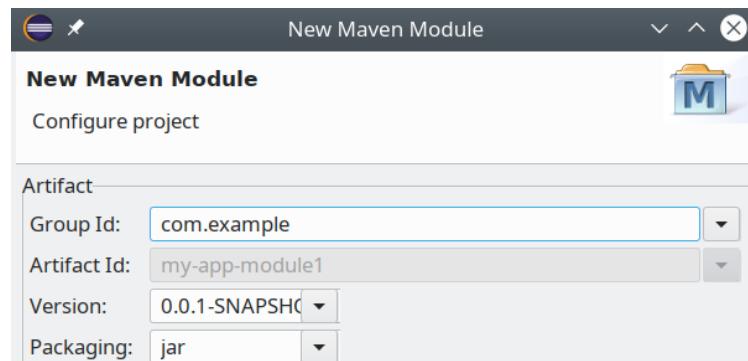
As usual, Eclipse can infer default settings when actions are run as context menus when a project or a folder is selected. Since we want to create a module of a parent project, we select the parent

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:

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

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

```
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  <properties>
11    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
12    <!-- Change the Java compilation level to 8 -->
13    <maven.compiler.source>1.8</maven.compiler.source>
14    <maven.compiler.target>1.8</maven.compiler.target>
15  </properties>
16  <dependencies>
17    <dependency>
18      <groupId>junit</groupId>
19      <artifactId>junit</artifactId>
20      <version>4.13</version>
21      <scope>test</scope>
22    </dependency>
23  </dependencies>
24 </project>
```

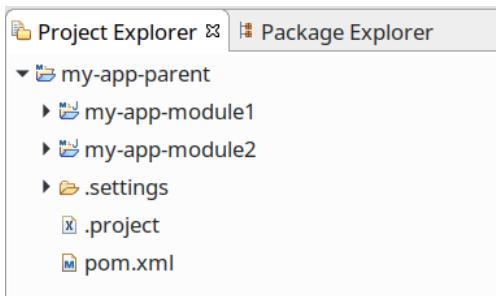
We update the Maven projects. We can do that starting from the parent project: right-click on the parent project and select **Maven → Update Project...**, in the dialog make sure that both the parent

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. This submodule will inherit all the configurations from the parent POM as well.



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 “Hierarchical” presentation of nested projects so that you avoid the above duplicate folders: from the menu of the view 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 (see Section [Properties](#)):

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <parent>
4     <groupId>com.example</groupId>
5     <artifactId>my-app-parent</artifactId>
6     <version>0.0.1-SNAPSHOT</version>
7   </parent>
8   <artifactId>my-app-module2</artifactId>
9   <dependencies>
```

```
10 <dependency>
11   <groupId>${project.groupId}</groupId>
12   <artifactId>my-app-module1</artifactId>
13   <version>${project.version}</version>
14 </dependency>
15 </dependencies>
16 </project>
```

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:

```
1 package org.my.app.module2;
2
3 public class App {
4     public static void main(String[] args) {
5         org.my.app.module1.App.main(args);
6     }
7 }
```

## 7.14.2 Effective POM

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).

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

```
1 mvn help:effective-pom
```

In Eclipse, the m2e editor provides a dedicated tab (of course, read-only), see Section [The Eclipse m2e pom.xml multi-tab editor](#).

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.

### 7.14.3 Project aggregation and project inheritance

Let's now examine how a multi-module project behaves when building with Maven.

We saw that settings and dependencies from the parent POM are inherited in the module projects.

This means that we can build the `my-app-module1` by itself, e.g., with `mvn test`. It works because the dependency to JUnit is inherited from the parent POM.

Let's try to do the same for the `my-app-module2` module project. The Maven build fails at the very beginning with the following error:

```
1 [INFO] -----< com.example:my-app-module2 >-----
2 [INFO] Building my-app-module2 0.0.1-SNAPSHOT
3 [INFO] -----[ jar ]-----
4 [WARNING] The POM for com.example:my-app-module1:jar:0.0.1-SNAPSHOT is missing, no d\
5 ependency information available
6 [INFO] -----
7 [INFO] BUILD FAILURE
```

As we saw in the previous section, although `my-app-module2` inherits from the parent POM all the configurations and dependencies such as JUnit, it does not inherit the `<modules>` section, thus Maven is not able to resolve the dependency `my-app-module1`. In particular Maven searches for such a dependency in the local cache and the remote central repository.

We could solve the problem by first “installing” `my-app-module1` in the local repository: in the `my-app-module1` folder, we run this command

```
1 mvn install
```

As we saw in Section [Maven lifecycles](#) this phase comes after compilation, testing, and packaging, and will copy the created jar into the local Maven cache repository (the directory `.m2` in the home folder):

```
1 ...
2 compilation and testing
3 ...
4 [INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ my-app-module1 ---
5 [INFO] Building jar: <full path> /my-app-parent/my-app-module1/target/my-app-module1\
6 -0.0.1-SNAPSHOT.jar
7 [INFO]
8 [INFO] --- maven-install-plugin:2.4:install (default-install) @ my-app-module1 ---
9 [INFO] Installing <full path>/my-app-parent/my-app-module1/target/my-app-module1-0.0\
10 .1-SNAPSHOT.jar to <full path>/.m2/repository/com/example/my-app-module1/0.0.1-SNAPS\
```

```
11 HOT/my-app-module1-0.0.1-SNAPSHOT.jar
12 [INFO] Installing <full path>/my-app-parent/my-app-module1/pom.xml to <full path>/.m\
13 2/repository/com/example/my-app-module1/0.0.1-SNAPSHOT/my-app-module1-0.0.1-SNAPSHOT\
14 .pom
15 [INFO] -----
16 [INFO] BUILD SUCCESS
```

Note that also the POM is copied into the local repository since that one contains all the metadata information needed by Maven to resolve possible transitive dependencies. Indeed, the previous build failure message was related to the missing POM of the dependency `my-app-module1`.

Let's now try to build again the `my-app-module2` module project. The build still fails with a different error:

```
1 [ERROR] Failed to execute goal on project my-app-module2: Could not resolve dependen\
2 cies for project com.example:my-app-module2:jar:0.0.1-SNAPSHOT: Failed to collect de\
3 pendencies at com.example:my-app-module1:jar:0.0.1-SNAPSHOT: Failed to read artifact\
4 descriptor for com.example:my-app-module1:jar:0.0.1-SNAPSHOT: Could not find artifa\
5 ct com.example:my-app-parent:pom:0.0.1-SNAPSHOT
```

Maven can now find the locally installed `my-app-module1` but it fails to resolve its parent `my-app-parent`, which is referred to in the installed POM of `my-app-module1`. Note that Maven, when resolving `my-app-module1` found in the local repository, does not use the `my-app-parent` of the current project `my-app-module2`.

Thus, when building single projects that are part of a multi-module project, the inheritance of the parent POM is taken into consideration for computing the effective POM, but dependencies among modules of the same project must all be found in the local repository.

However, when building a multi-module project, we can rely on project aggregation mechanism. This mechanism in Maven is called **reactor**<sup>9</sup>.

This mechanism is already setup in our parent POM, thanks to the `<modules>` section, which lists all the module projects of the multi-module project. It is only a matter of running the Maven build starting from the parent POM project, and not on a single project basis like we did in the initial part of this section.

Thus, let's try to run this Maven command from the `my-app-parent` project:

---

<sup>9</sup>See <https://maven.apache.org/guides/mini/guide-multiple-modules.html>.

```
1 mvn test
2
3 [INFO] Scanning for projects...
4 [INFO] -----
5 [INFO] Reactor Build Order:
6 [INFO]
7 [INFO] my-app-parent [pom]
8 [INFO] my-app-module1 [jar]
9 [INFO] my-app-module2 [jar]
10 [INFO]
11 ...
12 [INFO] ----- < com.example:my-app-module1 > -----
13 [INFO] Building my-app-module1 0.0.1-SNAPSHOT [2/3]
14 [INFO] ----- [ jar ] -----
15 ... compile and test ...
16 [INFO]
17 [INFO] ----- < com.example:my-app-module2 > -----
18 [INFO] Building my-app-module2 0.0.1-SNAPSHOT [3/3]
19 [INFO] ----- [ jar ] -----
20 ... compile and test ...
21 [INFO]
22 [INFO] Reactor Summary:
23 [INFO]
24 [INFO] my-app-parent ..... SUCCESS
25 [INFO] my-app-module1 ..... SUCCESS
26 [INFO] my-app-module2 ..... SUCCESS
27 [INFO]
28 [INFO] BUILD SUCCESS
```

This time all the module projects are built together, thus `my-app-module2` can be compiled since its dependency `my-app-module1` is resolved correctly as part of the same reactor.

In particular, the Maven reactor performs the following actions:

- Collects all the modules to build
- Sorts the modules into the correct build order
- Builds the modules in order

Note that since modules within a multi-module project can depend on each other, like in our example, the reactor sorts all the projects so that any project is built before it is required.

This implies two things:

- In our parent POM we could exchange the order of the listed modules, since the reactor will reorder them anyway;

- cycles in the dependencies are not allowed.

Let's run `mvn install` again, but this time on the parent project: all the modules (including the parent POM) will be installed in the local repository.

After that, we can also run a Maven build on the single `my-app-module2` module project, e.g., `mvn test`; this time the build will succeed, since all the dependencies will be resolved using the local cache. In particular, `my-app-module1` and transitively the referred parent will be resolved from the local repository.

All the above mechanisms also imply that the parent POM does not necessarily need to be also the project aggregator. Although, typically, the parent POM also lists all the modules, the two concepts can be kept separate:

- a parent POM contains all the common dependencies and configurations of several projects, but it does not contain the `<modules>` section;
- a POM project is used as the reactor aggregator, listing all the modules that must be built together. Several aggregator projects can also be defined, for example, to build only a subset of all the modules of the same multi-module project, as long as all the needed dependencies are part of the same subset.

The example `maven-multimodule-aggregator-example` contains a multi-module Maven project. The modules start with `my-app2-*` so that you can import both these projects and the ones of the previous example in the Eclipse workspace.

In this example,

- the parent project does not contain physically the submodule projects, which are at the same level as the parent in the filesystem;
- the parent project does NOT contain the `<modules>` section, thus it is used by the sub-modules only for the POM inheritance;
- there is a separate aggregator project, `my-app2-aggregator`, which only serves as the aggregator to build the multi-module project altogether; the directory of this project is at the same level as the other projects.

Module projects refer to the parent POM using also the `<relativePath>` tag:

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <parent>
4     <groupId>com.example</groupId>
5     <artifactId>my-app2-parent</artifactId>
6     <version>0.0.1-SNAPSHOT</version>
7     <relativePath>../my-app2-parent/pom.xml</relativePath>
8   </parent>
9   <artifactId>my-app2-module1</artifactId>
10 </project>
```

The aggregator POM is as simple as this:

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.example</groupId>
4   <artifactId>my-app2-aggregator</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6   <packaging>pom</packaging>
7   <modules>
8     <module>../my-app2-module1</module>
9     <module>../my-app2-module2</module>
10    <module>../my-app2-parent</module>
11  </modules>
12 </project>
```

We can note:

- There is no relation between the parent POM and the aggregator POM;
- In the aggregator, we can list modules in any order: Maven reactor will reorder them in case any project must be built first;
- The modules to build are referred to using a relative path since they do not live physically in subfolders;
- The parent POM is explicitly listed: this ensures that an `install` or a `deploy` phase will correctly copy in the local or remote repository also the parent POM, which is needed when resolving the modules that refer to the parent (see the error we got when we installed only the first module above in this section).

Now if we want to build the multi-module project, it makes no sense to run the Maven build from the parent project: we must run it from the aggregator project.

## 7.14.4 Build a single module

If we run the `install` phase on a multi-module project, we know that the artifacts and the corresponding metadata (POMs) are copied into the local Maven repository. After that, we can build a single module.

This can be done by running the Maven build from the single module project, as we did in the previous section, or by running the build from the aggregator project, with additional command-line options. Two useful command line options are:<sup>10</sup>

### **-rf, -resume-from**

Resumes reactor from the specified project.

### **-pl, -projects**

Builds only the specified reactor projects.

These options are useful when you have a multi-module project with lots of sub-modules: running the entire build might take minutes. Being able to build (or re-build) single modules will save a lot of time.

Of course, this makes sense if you plan to run Maven builds locally a lot. In general, when using Eclipse for development, running a Maven build does not make much sense, especially when we adopt Continuous Integration systems (Chapter [Continuous Integration](#)).

## 7.14.5 Dependency management

As we saw in the previous sub-sections, a dependency in the parent is inherited in all sub-modules, and this fosters consistencies among the sub-modules of a multi-module project.

But, what if only some modules need that dependency? Recall that a “compile” scope dependency will be a transitive dependency of that artifact once released.

If we still want to configure the dependency in a single place (parent POM) but make sure that a dependency is used only in the sub-modules that effectively use that, then we configure dependencies in the parent POM in the section `<dependencyManagement>`. In that section, we specify the version and other configurations of dependencies. In the sub-modules, when we need that dependency, we specify that as usual in the `<dependencies>` section, but we only mention the `groupId` and the `artifactId`.

For example, in the parent POM of one of our multi-module projects we can specify

---

<sup>10</sup>For more information see <https://blog.sonatype.com/2009/10/maven-tips-and-tricks-advanced-reactor-options/>.

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

Remember that defining a dependency in `<dependencyManagement>` only configures it, but it does not include (or use) it in any build phase. Thus, the sub-modules tests will not compile at the moment. In the sub-modules, inheriting from this POM, we need to specify the JUnit dependency (note that version and scope are not repeated):

```
1 <dependencies>
2   <dependency>
3     <groupId>junit</groupId>
4     <artifactId>junit</artifactId>
5   </dependency>
6 </dependencies>
```

## 7.14.6 Bill of Material (BOM)

A Maven project can **import** managed dependencies, i.e., the ones in `<dependencyManagement>`, from other projects. Note that this mechanism is different from POM inheritance, which, as already said, is single inheritance.

This import mechanism is useful when we need a few artifacts from a library and we must make sure to use the same versions of those artifacts. For example, in Section *The bank example as a Maven project*, we are using two artifacts from the project Log4J:

```
1 <dependencies>
2   <dependency>
3     <groupId>org.apache.logging.log4j</groupId>
4     <artifactId>log4j-api</artifactId>
5     <version>2.13.0</version>
6   </dependency>
7   <dependency>
8     <groupId>org.apache.logging.log4j</groupId>
9     <artifactId>log4j-core</artifactId>
10    <version>2.13.0</version>
11  </dependency>
12  ...
13 </dependencies>
```

and we must make sure to use the same versions.

Of course, we could enforce this by using a property for the version (as we did in Section [Properties](#)), but this assumes that Log4j uses the same versions for all its sub-modules. In any case, we would need an easier mechanism to keep the versions of a project's dependencies in sync with the versions distributed in the library.

For this reason, a multi-module project can provide a special POM, called **Bill of Materials (BOM)**, whose only aim is to define a `<dependencyManagement>` section with all its modules' versions in sync (possibly also for its transitive dependencies). In your project, possibly in your parent POM if you have a multi-module project, such a dependency is specified in the `<dependencyManagement>` section with the scope `import`.

For example, Log4j is a multi-module library. We only use the two above artifacts from Log4j. Log4j provides a BOM artifact, `log4j-bom`.

In the bank example we specify this:

```
1 <dependencyManagement>
2   <dependencies>
3     <dependency>
4       <groupId>org.apache.logging.log4j</groupId>
5       <artifactId>log4j-bom</artifactId>
6       <version>2.13.0</version>
7       <scope>import</scope>
8       <type>pom</type>
9     </dependency>
10    </dependencies>
11 </dependencyManagement>
```

And then we simplify the dependencies for Log4j by removing the versions:

```
1 <dependencies>
2   <dependency>
3     <!-- version taken from the imported BOM -->
4     <groupId>org.apache.logging.log4j</groupId>
5     <artifactId>log4j-api</artifactId>
6   </dependency>
7   <dependency>
8     <!-- version taken from the imported BOM -->
9     <groupId>org.apache.logging.log4j</groupId>
10    <artifactId>log4j-core</artifactId>
11  </dependency>
12  ...
13 </dependencies>
```



if you keep the versions also in the standard `<dependencies>` section, after importing a BOM for those artifacts, you will get a warning from the m2e Eclipse editor: Duplicating managed version 2.13.0 for log4j-api.

Switching the version of an imported BOM automatically allows us to make sure all the used artifacts will have their versions in sync.

The BOM can also be used in your projects with the same aim. Not to mention that you could have in the same reactor modules with different parents but importing the same BOM of yours.



By looking at the “Effective POM” you can see that all the dependencies specified in the imported BOM are visible in the `<dependencyManagement>` section.

## 7.14.7 How Eclipse resolves the projects in the workspace

Once again, let's stress that when using Eclipse the effective compilation and classpath is handled completely by Eclipse, taking into consideration Maven dependencies and automatically downloading them if not already cached.

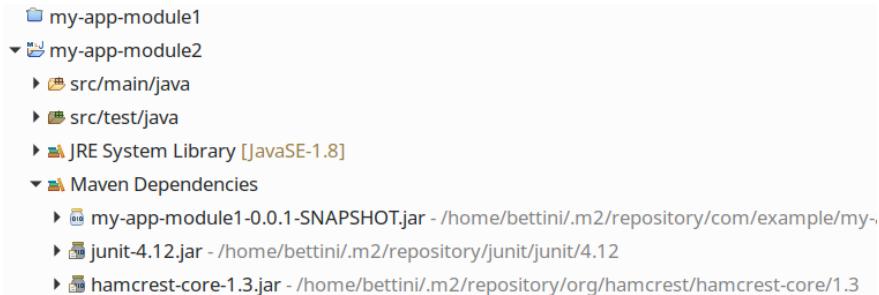
If a project like `my-app-module2` (taken from the previous sections) depends on another project, such as `my-app-module1`, which is opened in the workspace, then the opened project is used for dependency resolution. This is evident in the Eclipse project's layout (“Maven Dependency” node):



Dependency resolved using the opened workspace project

Thus, running `mvn install` is usually not required when developing with Eclipse.

In any case, after installing the modules of the multi-module project, we could also close the `my-app-module1` project: Eclipse will then resolve the dependency using the JAR installed in the local repository:



Dependency resolved using the installed JAR if the project is closed

## 7.15 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:<sup>11</sup>

---

<sup>11</sup>In this book, we will not use the lifecycle `site`, thus, we will only take into consideration **build plugins**, which will be executed during the build and are configured in the `<build>` section; we will skip **reporting plugins** which will be executed during the site generation and are configured in the `<reporting>` section.

```
1 <build>
2   <plugins>
3   ...
4   </plugins>
5 </build>
```

Each plugin is configured in an element `<plugin>`.

The configuration of a plugin typically consists of specifying

- its Maven coordinates (Section [Maven coordinates \(GAV\)](#));
- 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:

```
1 <plugin>
2   <groupId>...</groupId>
3   <artifactId>...</artifactId>
4   <version>...</version>
5   <executions>
6     <execution>
7       <id>...</id>
8       <phase>phase to bind the goal to</phase>
9       <goals>
10      <goal>plugin goal</goal>
11      </goals>
12      <configuration>
13        ... configuration for this execution only
14      </configuration>
15      </execution>
16      <execution>
17        ... as above
18      </execution>
```

```
19  </executions>
20  <configuration>
21      ... configuration common to all executions
22  </configuration>
23 </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” (Section [“Effective POM”](#)):

```
1 <plugin>
2     <artifactId>maven-compiler-plugin</artifactId>
3     <version>3.1</version>
4     <executions>
5         <execution>
6             <id>default-compile</id>
7             <phase>compile</phase>
8             <goals>
9                 <goal>compile</goal>
10                </goals>
11        </execution>
12        <execution>
13            <id>default-testCompile</id>
14            <phase>test-compile</phase>
15            <goals>
16                <goal>testCompile</goal>
17                </goals>
18        </execution>
19    </executions>
20 </plugin>
```

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

## 7.15.1 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 (see Section [Java settings](#)).

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

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

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

```
1 <build>
2   <plugins>
3     <plugin>
4       <artifactId>maven-compiler-plugin</artifactId>
5       <version>3.1</version>
6       <configuration>
7         <source>1.8</source>
8         <target>1.8</target>
9       </configuration>
10      </plugin>
11    </plugins>
12  </build>
```

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

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

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.

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:

```
1 <plugin>
2   <artifactId>maven-compiler-plugin</artifactId>
3   <version>3.1</version>
4   <executions>
5     <execution>
6       <id>default-compile</id>
7       <phase>compile</phase>
8       <goals>
9         <goal>compile</goal>
10      </goals>
11      <configuration>
12        <source>1.8</source>
13        <target>1.8</target>
14      </configuration>
15    </execution>
16    <execution>
17      <id>default-testCompile</id>
18      <phase>test-compile</phase>
19      <goals>
20        <goal>testCompile</goal>
21      </goals>
22      <configuration>
23        <source>1.8</source>
24        <target>1.8</target>
25      </configuration>
26    </execution>
27  </executions>
28  <configuration>
29    <source>1.8</source>
30    <target>1.8</target>
31  </configuration>
32 </plugin>
```

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.

## 7.15.2 Generate 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 the 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:

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-source-plugin</artifactId>
4   <version>3.0.1</version>
5   <executions>
6     <execution>
7       <id>attach-sources</id>
8       <!-- the phase would not be required in this case: goal jar of
9           this plugin already binds by default to the "package" phase -->
10      <phase>package</phase>
11      <goals>
12        <goal>jar</goal>
13      </goals>
14    </execution>
15  </executions>
16 </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:

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-javadoc-plugin</artifactId>
4   <version>3.0.1</version>
5   <executions>
6     <execution>
7       <id>attach-javadocs</id>
8       <goals>
9         <goal>jar</goal>
10      </goals>
11    </execution>
```

```
12    </executions>
13  </plugin>
```

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 that we created in the previous sections, by running `mvn package` on the parent project the two sub-modules will contain the additional JARs:

```
1 my-app-module1-0.0.1-SNAPSHOT.jar
2 my-app-module1-0.0.1-SNAPSHOT-javadoc.jar
3 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.

### 7.15.3 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`:

```
1 package com.mycompany.app;
2
3 public class App {
4     public static void main(String[] args) {
5         System.out.println("Hello World!");
6     }
7 }
```

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`:

```
1 java -jar <fullpath>/target/my-app-1.0-SNAPSHOT.jar
2
3 no main manifest attribute, in <fullpath>/target/my-app-1.0-SNAPSHOT.jar
```



The **manifest** is a special file with meta-information about the files packaged in a JAR file. For further information about it see <https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html>.

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, to run a Java application packaged in a 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:

```
1 java -cp <fullpath>/target/my-app-1.0-SNAPSHOT.jar com.mycompany.app.App  
2  
3 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>`:

```
1 <build>  
2   <plugins>  
3     <plugin>  
4       <artifactId>maven-jar-plugin</artifactId>  
5       <version>2.4</version>  
6       <configuration>  
7         <archive>  
8           <manifest>  
9             <mainClass>com.mycompany.app.App</mainClass>  
10            </manifest>  
11          </archive>  
12        </configuration>  
13      </plugin>  
14    </plugins>  
15  </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:

```
1 java -jar <fullpath>/target/my-app-1.0-SNAPSHOT.jar
2
3 Hello World!
```

## 7.15.4 Creating a FatJar

Let's go back to the bank-example Maven project. This also has a Java class with the `main` method, `testing.example.bank.app.Main`:

```
1 package testing.example.bank.app;
2
3 import java.util.ArrayList;
4 import org.apache.logging.log4j.LogManager;
5 import org.apache.logging.log4j.Logger;
6 import testing.example.bank.Bank;
7
8 public class Main {
9
10    private static final Logger LOGGER = LogManager.getLogger(Main.class);
11
12    public static void main(String[] args) {
13        LOGGER.info("App started");
14        Bank bank = new Bank(new ArrayList<>());
15        int bankAccountId = bank.openNewBankAccount(10);
16        bank.deposit(bankAccountId, 20);
17        LOGGER.info("App terminated");
18    }
19 }
```

Let's create the jar for this project, with `mvn package`.

Let's try to run from the command line the Java application by running the generated jar, which can be found in `target/bank-example-0.0.1-SNAPSHOT.jar`:

```

1 java -cp <fullpath>/target/bank-example-0.0.1-SNAPSHOT.jar testing.example.bank.app.\
2 Main
3
4 Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/logging/log4j/\
5 LogManager
6         at testing.example.bank.app.Main.<clinit>(Main.java:12)
7 Caused by: java.lang.ClassNotFoundException: org.apache.logging.log4j.LogManager
8         at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
9         at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
10        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:349)
11        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
12        ... 1 more

```

This is because log4j is not part of the generated JAR. Thus, this time, configuring the generated JAR with information about the main class would not solve the above problem.

Note that in the above command, when we run our application we provide a complete classpath. If we also specify the absolute paths of the two log4j JARs, which are in our local .m2 Maven cache folder, then everything works (the complete path can be easily retrieved from Eclipse, expanding the node “Maven Dependencies”, selecting the log4j JAR and using the contextual menu “Copy Qualified Name”):

```

1 java -cp <fullpath>/target/bank-example-0.0.1-SNAPSHOT.jar:<fullpath>/log4j-api-2.1\
2 3.0.jar:<fullpath>/log4j-core-2.13.0.jar testing.example.bank.app.Main
3
4 [main] INFO testing.example.bank.app.Main - App started
5 [main] INFO testing.example.bank.Bank - New account opened with id: 1
6 [main] INFO testing.example.bank.app.Main - App terminated

```

It is worth mentioning that the `log4j2.xml` file, as a resource file, is correctly inserted in the generated JAR, thus at run-time Log4j correctly finds it.

Indeed, the dependencies of a Maven project are not added to the generated JAR: This is standard for Maven artifacts since they are meant to be consumed by other Maven projects (and Maven automatically knows which dependencies to download). But if we want to create a JAR representing a self-contained Java application to be executed from the command line, we must create a jar with all the dependencies: a so-called **fatjar** (also known as **uber-jar**).

In order to create a fatjar, with all the dependencies (of course, only the compile dependencies, not the testing dependencies) we use `maven-assembly-plugin`, specifying the `<descriptorRef> jar-with-dependencies`. We also specify the name of our main class, so that this information will be stored in the manifest of the jar, just like we did in the previous section. By default, this plugin has no goal bound to any phase, so we need to bind its goal `single` to the right phase, which can be package:

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.apache.maven.plugins</groupId>
5       <artifactId>maven-assembly-plugin</artifactId>
6       <executions>
7         <execution>
8           <phase>package</phase>
9           <goals>
10          <goal>single</goal>
11        </goals>
12        <configuration>
13          <descriptorRefs>
14            <descriptorRef>jar-with-dependencies</descriptorRef>
15          </descriptorRefs>
16          <archive>
17            <manifest>
18              <mainClass>testing.example.bank.app.Main</mainClass>
19            </manifest>
20          </archive>
21        </configuration>
22      </execution>
23    </executions>
24  </plugin>
25 </plugins>
26 </build>
```



Why didn't we specify the version of this plugin? Indeed we get no warning when running the Maven build due to the missing version. The reason why we can skip the version for this plugin will be clear later, in Section [The \*pluginManagement\* element](#).

Let's run `mvn package`. This time, an additional JAR will be created, `bank-example-0.0.1-SNAPSHOT-jar-with-dependencies.jar`, which includes also the Java classes of all the compile-time dependencies of the project, that is, the Log4j classes. Indeed, this JAR is almost 2 Mb, while the standard JAR is about 6 Kb.

Let's run this JAR directly from the command line and our Java application is correctly executed:

```
1 java -jar <fullpath>/target/bank-example-0.0.1-SNAPSHOT-jar-with-dependencies.jar
2
3 [main] INFO testing.example.bank.app.Main - App started
4 [main] INFO testing.example.bank.Bank - New account opened with id: 1
5 [main] INFO testing.example.bank.app.Main - App terminated
```

Since we are interested in build automation, we would like to have an automatic procedure for checking that the generated JAR can be executed as a standalone, self-contained JAR, as opposed to manually checking from the command line. We can use the `org.codehaus.mojo:exec-maven-plugin` that allows you to execute from Maven any Java application during the build.

To configure this plugin, we must consider that the fatjar must be already built, so we need to bind this plugin to a phase executed after “package”. We bind this plugin’s goal to the phase “verify” which we know is executed after “package”. This also respects our intention of “verifying” that the JAR with dependencies is valid.

```
1 <plugin>
2   <groupId>org.codehaus.mojo</groupId>
3   <artifactId>exec-maven-plugin</artifactId>
4   <version>1.6.0</version>
5   <executions>
6     <execution>
7       <id>run-jar-with-dependencies</id>
8       <phase>verify</phase>
9       <goals>
10      <goal>exec</goal>
11    </goals>
12    <configuration>
13      <executable>java</executable>
14      <arguments>
15        <argument>-jar</argument>
16        <argument>${project.build.directory}/${project.artifactId}-${project.versi\
17 on}-jar-with-dependencies.jar</argument>
18      </arguments>
19    </configuration>
20  </execution>
21 </executions>
22 </plugin>
```

Note that we refer to the JAR using Maven properties, so that the build is reproducible in any environment.

Let’s run `mvn verify` and the build will succeed, since the JAR with dependencies can be executed.

Try and remove “-jar-with-dependencies” from `<argument>`: this will execute the standard JAR, which does not specify the main class nor it contains the dependencies, and the build will fail.

Another plugin to create a fatjar, but with more configuration options is `maven-shade-plugin`. For example, it allows you to “relocate” the classes and packages just to avoid conflicts when the JAR is used in other projects. For more details see <https://maven.apache.org/plugins/maven-shade-plugin/index.html>.

## 7.15.5 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>`:

```
1 <build>
2   <pluginManagement>
3     <plugins>
4       <plugin>
5         <artifactId>maven-compiler-plugin</artifactId>
6         <configuration>
7           <source>1.8</source>
8           <target>1.8</target>
9         </configuration>
10        </plugin>
11      </plugins>
12    </pluginManagement>
13  </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.

In Section *Creating a FatJar* we configured the `maven-assembly-plugin` without specifying the version. That’s because the version of that plugin is already configured in the `<pluginManagement>` section that is inherited. Verify that by looking at the “Effective POM”:

```
1 <pluginManagement>
2   <plugins>
3     <plugin>
4       <artifactId>maven-assembly-plugin</artifactId>
5       <version>2.2-beta-5</version>
6     </plugin>
7     ...
8   </plugins>
9 </pluginManagement>
```

This means that if we simply want to use a newer version of one of the standard plugins that are already bound to lifecycle phases, we just have to specify the version in the `<pluginManagement>`. We will still use the original configuration of the plugin.

For example, we could force newer versions of the `compiler` plugin and the `surefire` plugin like that:

```
1 <build>
2   <pluginManagement>
3     <plugins>
4       <plugin>
5         <artifactId>maven-compiler-plugin</artifactId>
6         <version>3.8.0</version>
7       </plugin>
8       <plugin>
9         <artifactId>maven-surefire-plugin</artifactId>
10        <version>2.22.1</version>
11      </plugin>
12    </plugins>
13  </pluginManagement>
14  ...
```

This strategy also allows us to lock down plugins versions and avoid using the default versions of plugins, which depend on the current Maven installation. This way, we have full control over the versions of the plugins.



If you use a more recent version, e.g., 1.4, of the `maven-archetype-quickstart`, you will see that the generated POM has a big `<pluginManagement>` to lock down main plugins versions.

Moreover, if we plan to run a Maven plugin goal only from the command line explicitly, we can still use the `<pluginManagement>` to specify the configuration of the plugin. Such a plugin configuration in our POM will be used when running its goals from the command line. Thus, in the configuration,

we never bind the plugin's goals to any phase, but we configure the main parts of the plugin, e.g., its version.

For example, in Section [Run Maven goals](#), we ran these goals with an explicit version for 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
```

We can avoid specifying that version by adding this configuration to the <pluginManagement> section

```
1 <build>
2   <pluginManagement>
3     <plugins>
4       <plugin>
5         <artifactId>maven-site-plugin</artifactId>
6         <version>3.7.1</version>
7       </plugin>
8     ...
9   ...
```

Now we can simply run

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

And maven-site-plugin version 3.7.1 will be used.

## 7.15.6 Configuring the PIT Maven plugin

We saw mutation testing with PIT in Chapter [Mutation Testing](#). PIT also provides a Maven plugin, org.pitest:pitest-maven.

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:

```

1 <build>
2   <pluginManagement>
3     <plugins>
4       ...
5       <plugin>
6         <groupId>org.pitest</groupId>
7         <artifactId>pitest-maven</artifactId>
8         <version>1.5.2</version>
9         <configuration>
10           <targetClasses>
11             <!-- excludes testing.example.bank.app.Main -->
12             <param>testing.example.bank.Bank*</param>
13           </targetClasses>
14           <targetTests>
15             <param>testing.example.bank.*</param>
16           </targetTests>
17           <mutators>
18             <mutator>DEFAULTS</mutator>
19           </mutators>
20           <mutationThreshold>80</mutationThreshold>
21         </configuration>
22       </plugin>
23     ...

```

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 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:

```
1 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.

Note that, despite having configured the `pitest-maven` plugin, we still need to use the `groupId:artifactId:goal` form, because this plugin does not respect the naming convention  `${prefix}-maven-plugin`.

In the generated report you will find several survived mutants. That is expected since we copied the Java files from the bank example of Chapter [JUnit](#); indeed, in Chapter [Mutation Testing](#), Section [Mutation testing on the Bank example](#), we learned how to fix the problems with survived mutants.

Try and change the `mutationThreshold` to 90 and the build will fail with such a message:

```
1 Mutation score of 80 is below threshold of 90
```

In the `<mutators>` section we can enable a set of mutators (as we did with `DEFAULTS`), single mutators by using the corresponding ID (mutators and their IDs can be found here <http://pitest.org/quickstart/mutators/>) and a combination of both.

For example,

```
1 ...
2 <mutators>
3   <mutator>DEFAULTS</mutator>
4   <mutator>NON_VOID_METHOD_CALLS</mutator>
5 </mutators>
6 ...
```

## 7.15.7 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. 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:

```
1 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 `.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.



You can import a `.exec` file in Eclipse EclEmma to see the report in the “Coverage” view: select `File → Import... → Run/Debug → Coverage Session` and fill the dialog pages with information (including the source folder that represents the scope of the coverage session).

If we tried to run the `jacoco` goals in the “shorter” form from the command line:

```
1 mvn clean jacoco:prepare-agent test
```

We would get this error:

```
1 [ERROR] No plugin found for prefix 'jacoco' in the current project and in the plugin\
2 groups [org.apache.maven.plugins, org.codehaus.mojo] available from the repository\
3 s...
```

Indeed, nothing about this plugin is configured in the POM, and we already know that Maven will be able to run, using the “shorter” form, only the goals of the Maven plugins belonging to the groupIds mentioned in the error.

Thus, we can configure the plugin in the `<pluginManagement>` section with its coordinates and then run the shorter form of the goal:

```
1 <build>
2   <pluginManagement>
3     <plugins>
4       <plugin>
5         <groupId>org.jacoco</groupId>
6         <artifactId>jacoco-maven-plugin</artifactId>
7         <version>0.8.5</version>
8       </plugin>
9     ...
```

and use the shorter form when executing its goal:

```
1 mvn clean jacoco:prepare-agent test
```

Another goal of this plugin is report, which takes the generated .exec file and creates a website with code coverage report (it also creates the report in the shape of an XML and CSV file). Let's run this goal after the phase test (recall that it is in this phase that the jacoco.exec file is generated). Please keep in mind that if the prepare-agent goal was configured to create the .exec file in another output folder, the goal report should be configured accordingly (see the configuration options here: <https://www.eclemma.org/jacoco/trunk/doc/report-mojo.html>).

```
1 mvn clean jacoco:prepare-agent test jacoco:report
```

The HTML website (and the XML and CSV files) are generated by default in target/site/jacoco. You can examine the website with any browser. Of course, the Main class is not covered and the Bank class is not fully covered.<sup>12</sup>

Note that the Maven plugin of JaCoCo automatically excludes test classes from the code coverage and report, while when using EclEmma from Eclipse, we had to manually exclude test classes (see Chapter [Code coverage](#)).

Now we want to use another goal, check, which checks whether some code coverage criteria have been met. This goal has a required parameter, <rules>, which specifies the above-mentioned criteria.<sup>13</sup> Thus, we cannot simply invoke this goal, without a configured execution for the goal, even if not bound to any phase. In this example, we require a line coverage minimum of 50% for every package:

```
1 <build>
2   <pluginManagement>
3     <plugins>
4       <plugin>
5         <groupId>org.jacoco</groupId>
6         <artifactId>jacoco-maven-plugin</artifactId>
7         <version>0.8.5</version>
8         <executions>
9           <execution>
10             <id>jacoco-check</id>
11             <goals>
12               <goal>check</goal>
13             </goals>
14             <configuration>
15               <rules>
16                 <rule>
17                   <element>PACKAGE</element>
18                   <limits>
19                     <limit>
```

---

<sup>12</sup>We dealt with full code coverage of the bank example in Chapter [Code coverage](#), Section [How code coverage can help](#).

<sup>13</sup>See <https://www.eclemma.org/jacoco/trunk/doc/check-mojo.html>.

```

20          <counter>LINE</counter>
21          <value>COVEREDRATIO</value>
22          <minimum>0.50</minimum>
23          </limit>
24      </limits>
25  </rule>
26  </rules>
27 </configuration>
28 </execution>
29 </executions>
30 </plugin>
31 ...

```

If we want to invoke this configured goal from the command line, we also mention the id of the execution:

```

1 mvn clean jacoco:prepare-agent test jacoco:report jacoco:check@jacoco-check
2 ...
3 [INFO] --- jacoco-maven-plugin:0.8.5:check (jacoco-check) @ bank-example ---
4 [INFO] Loading execution data file <fullpath>/target/jacoco.exec
5 [INFO] Analyzed bundle 'bank-example' with 3 classes
6 [WARNING] Rule violated for package testing.example.bank.app: lines covered ratio is \
7 0.00, but expected minimum is 0.50
8 [INFO] -----
9 [INFO] BUILD FAILURE
10 [INFO] -----
11 [ERROR] Failed to execute goal org.jacoco:jacoco-maven-plugin:0.8.5:check (jacoco-ch\
eck) on project bank-example: Coverage checks have not been met.

```

As expected, the build fails, because this goal detected that the rule is violated for the package `testing.example.bank.app` (recall that its only Java class is `Main`, which is not tested). On the contrary, while the `testing.example.bank.Bank` is known not to be fully covered, its coverage ratio meets the criteria specified in the goal configuration.



If we gave to the execution the id `default-cli`, then we could have invoked the goal without the id, since that is the default execution id.

Let's exclude `Main` from the code coverage. We can do that in the `<configuration>` section with the element `<exclude>` outside any execution. Such exclusion will be used by all the goals:

```
1 ...
2 <groupId>org.jacoco</groupId>
3 <artifactId>jacoco-maven-plugin</artifactId>
4 <version>0.8.5</version>
5 <configuration>
6   <excludes>
7     <exclude>**/Main.*</exclude>
8   </excludes>
9 </configuration>
10 <executions>
11   ...
```

If we now run the above Maven command, we can see that the `Main` class is excluded from the code coverage, from the coverage report and from the `check` goal (2 classes are analyzed instead of 3 as before). Thus, `check` will not make the build fail now:

```
1 [INFO] --- jacoco-maven-plugin:0.8.5:check (jacoco-check) @ bank-example ---
2 [INFO] Loading execution data file <fullpath>/target/jacoco.exec
3 [INFO] Analyzed bundle 'bank-example' with 2 classes
4 [INFO] All coverage checks have been met.
5 [INFO] -----
6 [INFO] BUILD SUCCESS
7 [INFO] -----
```

Up to now, we configured the JaCoCo Maven plugin in the `<pluginManagement>` section and we called its goals directly from the command line.

We could enable the goals `prepare-agent`, `report` and `check` by binding them to appropriate lifecycle phases. Even better, we can simply list the goals as executions, since those goals have default phases for executions. Of course, these default phases are sensible for the aims of the goals:

- `prepare-agent` Binds by default to the lifecycle phase `initialize`, which is executed very early, for sure before `test`;
- `report` and `check` bind by default to the lifecycle phase `verify`.

Thus, we add the executions of `prepare-agent` and `report`, knowing that they will be bound by default to the corresponding phases:

```

1 ...
2 <groupId>org.jacoco</groupId>
3 <artifactId>jacoco-maven-plugin</artifactId>
4 <version>0.8.5</version>
5 <configuration>
6   <excludes>
7     <exclude>**/Main.*</exclude>
8   </excludes>
9 </configuration>
10 <executions>
11   <execution>
12     <goals>
13       <!-- binds by default to the phase "initialize" -->
14       <goal>prepare-agent</goal>
15       <!-- binds by default to the phase "verify" -->
16       <goal>report</goal>
17     </goals>
18   </execution>
19   <execution>
20     <id>jacoco-check</id>
21     <goals>
22       <!-- binds by default to the phase "verify" -->
23       <goal>check</goal>
24     </goals>
25   <configuration>
26   ...

```

We have separate executions because for the goal `check` we have to specify a configuration that makes sense only for that goal.

Recall that this plugin is configured in `<pluginManagement>`, thus it is not enabled by default during the build. We can then enable the plugin in the `<build>` section:

```

1 <build>
2   <plugins>
3     ...
4     <plugin>
5       <!-- configured in pluginManagement -->
6       <groupId>org.jacoco</groupId>
7       <artifactId>jacoco-maven-plugin</artifactId>
8     </plugin>

```

Now, running the `verify` phase will make sure that the tests are run with the JaCoCo agent, a report of coverage is generated and coverage criteria are checked.

Separating the configuration of a plugin in `<pluginManagement>` and later enabling the plugin in the `<build>` section is useful:

- in a multi-module project for configuring a plugin in a single place, once and for all, and for enabling the plugin only in specific sub-modules;
- when a plugin is enabled only in some circumstances, which are configured as “profiles”, as shown in the Section [Maven profiles](#).

## 7.16 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

```
1 <activation>
2   <jdk>[1.9,)</jdk>
3 </activation>
```

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

```
1 <activation>
2   <os>
3     <family>mac</family>
4   </os>
5 </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, also with executions, in the `<pluginManagement>` section, see Section [Configuring the JaCoCo Maven plugin](#)) and another one where the PIT Maven plugin has the goal `mutationCoverage` bound to the phase `verify` (already configured, without executions, in the `<pluginManagement>` section, see Section [Configuring the PIT Maven plugin](#)). 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`:

```
1 <profiles>
2   <profile>
3     <id>jacoco</id>
4     <build>
5       <plugins>
6         <plugin>
7           <!-- configured in pluginManagement -->
8           <groupId>org.jacoco</groupId>
9             <artifactId>jacoco-maven-plugin</artifactId>
10            </plugin>
11        </plugins>
12      </build>
13    </profile>
14    <profile>
15      <id>mutation-testing</id>
16      <build>
17        <plugins>
18          <plugin>
19            <!-- configured also in pluginManagement -->
20            <groupId>org.pitest</groupId>
21            <artifactId>pitest-maven</artifactId>
22            <executions>
23              <execution>
24                <goals>
25                  <goal>mutationCoverage</goal>
```

```
26      </goals>
27      <phase>verify</phase>
28    </execution>
29  </executions>
30  </plugin>
31 </plugins>
32 </build>
33 </profile>
34 </profiles>
```

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:

```
1 mvn clean verify -Pjacoco,mutation-testing
```

Of course, now that we have the profile `mutation-testing`, we could also configure the execution of the PIT Maven plugin in the `<pluginManagement>` section and we could then simply activate the plugin in the `mutation-testing` profile. This is left as an exercise.

The simplest activation policy is the following one:

```
1 <activation>
2   <activeByDefault>true</activeByDefault>
3 </activation>
```

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 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:

```
1 <profile>
2   <id>jacoco</id>
3   <activation>
4     <!-- Activated when the system property "skipCoverage" is not defined at
5       all, thus, implicitly active by default -->
6     <property>
7       <name>!skipCoverage</name>
8     </property>
9   </activation>
10 ...
11 ...
12 ...
```

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:

```
1 <profile>
2   <id>jacoco</id>
3   <activation>
4     <!-- Activated when the system property "skipCoverage" is not defined at
5       all or is defined with a value which is not "true", thus, implicitly active
6       by default -->
7     <property>
8       <name>skipCoverage</name>
9       <value>!true</value>
10    </property>
11  </activation>
12 ...
13 ...
14 ...
```

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`.

### 7.16.1 Don't abuse profiles

Maven profiles are a powerful and useful tool, especially when we need to customize our builds. However, it is also easy to misuse them and abuse them, resulting in builds that are hard to run and easy to fail.

First of all, we have just seen that using a profile to specify some configurations that by default should always be enabled is not a good idea. Indeed, `<activeByDefault>` does not work as expected, and achieving the desired goal requires some more involved solutions that might easily become hard to maintain. In the end, if a configuration must be active by default it should not be put in a profile: it should stay in the main part of the POM.

Moreover, profiles should be used to extend the build in some circumstances and not to provide some configurations that are essential to the build. This point can be easily shown with an example: if our build succeeds with the following command (where `myprofile` is defined in one of our POMs)

```
1 mvn clean verify -Pmyprofile
```

but it fails with the following command

```
1 mvn clean verify
```

then we have a terrible problem! Remember that *our build must succeed when no profile is activated*. Profiles must not be required to make the build pass.

The situation is even worse in case a profile requires another profile to be activated. In particular, remember that there is no way for a profile to activate another profile.

For example, let's consider these profiles, where the actual configurations and executions of plugins are not shown:

```
1 <!-- Example of wrong profiles -->
2 <profiles>
3   <profile>
4     <id>p1</id>
5     <!-- configuration and executions of plugin1 -->
6   </profile>
7   <profile>
8     <id>p2</id>
9     <!-- configuration and executions of plugin2,
10      which requires plugin1 -->
11   </profile>
12 </profiles>
```

Let's assume that `plugin2` requires that `plugin1` is executed, for example, because it needs some data produced by `plugin1`. (We will see a real example of such a situation in Chapter [Continuous Integration](#), Section [Let's revise our build process](#).) Then the build will fail or it will not behave as expected if we activate the profile `p2` but we forget to activate the profile `p1`. Even if we remember to activate them both, the command for the build will not reveal that the first profile is activated because it is needed by the second profile. Thus, situations like this should be completely avoided.

If the first profile is not meant to be used by itself, then we should have a single profile with both plugins.

If the first profile is meant to be used also without the second profile, but the second profile requires the plugin configured and activated in the first profile, then we definitely must add the configuration and the activation of the first plugin also in the second profile. This would make the second profile self-contained:

```
1 <!-- Fixed profiles -->
2 <profiles>
3   <profile>
4     <id>p1</id>
5     <!-- configuration and executions of plugin1 -->
6   </profile>
7   <profile>
8     <id>p2</id>
9     <!-- configuration and executions of plugin1 -->
10    <!-- configuration and executions of plugin2,
11        which requires plugin1 -->
12  </profile>
13 </profiles>
```

Keep in mind that even if we activate both profiles, Maven will merge all the configurations, so the fact that the first plugin is configured and activated in the same way in both profiles will not be a problem.

Of course now we have some duplication because of the configuration of the first plugin that occurs in both profiles. However, we know how to fix that: the configuration and the executions of the first plugin can be placed in the main `<pluginManagement>` section (outside profiles) and in both profiles we simply activate the plugin (i.e., just mention its `groupId` and `artifactId`):

```
1 <!-- Better profiles -->
2 <build>
3   <pluginManagement>
4     <!-- configuration and executions of plugin1 -->
5   </pluginManagement>
6 </build>
7
8 <profiles>
9   <profile>
10    <id>p1</id>
11    <!-- activation of plugin1 -->
12  </profile>
13  <profile>
14    <id>p2</id>
15    <!-- activation of plugin1 -->
16    <!-- configuration and executions of plugin2,
17        which requires plugin1 -->
18  </profile>
19 </profiles>
```

Now the only parts that are duplicated are the two lines for activating the first plugin. We can live

with that! Of course, we could do that same for the second plugin, but in this example, this is not strictly required.

Thus, in general, profiles must not be considered as a means to circumvent the strict Maven lifecycle structure.

## 7.17 The bank example as a multi-module project

Let us now see a complete example of a multi-module project where we try to use all the concepts explained in the chapter. Once again, we will use the bank example for the Java files and tests, but we'll split it into several sub-modules.

This structure of this multi-module project is probably exaggerated for its actual code, but for the sake of demonstration and learning, we will try to separate concepts as much as possible.

### bank-bankaccount-module

Contains only the implementation of the `BankAccount` and the corresponding tests. This module uses JUnit and AssertJ as test dependencies

### bank-bank-module

Contains only the implementation of the `Bank` and the corresponding tests. This uses `BankAccount` thus it depends on the previous module. This module uses Log4j as a `compile` dependency and JUnit and AssertJ as test dependencies.

### bank-app

Contains the `Main` class, thus it depends on the previous module (and, transitively and automatically, on the `bank-bankaccount-module`, though it never uses a `BankAccount` explicitly). This module contains no test and it is only meant to create a standalone executable JAR. Note that also this module needs Log4j as a `compile` dependency. This module has no explicit parent, but it imports the managed dependencies of the BOM (see below).

### bank-parent

The parent POM for this multi-module project, where common dependencies and common plugins are configured, but not automatically enabled by default. Indeed, the modules will enable them explicitly by relying on their configurations inherited from the parent POM.

### bank-bom

The BOM project, where the dependencies for `bank-bankaccount-module`, `bank-bank-module`, and their `compile` dependencies are kept in sync.

### bank-report

A project whose only aim is to create an aggregated coverage report of the two main modules of the project.

### 7.17.1 The bank BOM

This project is meant to be a BOM for the entire multi-module project, thus, it aims to configure the `compile` dependencies (it could also configure the `test` dependencies but for the sake of demonstration, we do that in the parent POM).

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.example.bank</groupId>
4   <artifactId>bank-bom</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6   <packaging>pom</packaging>
7
8   <properties>
9     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
10    <log4j.version>2.13.0</log4j.version>
11  </properties>
12
13  <dependencyManagement>
14    <dependencies>
15      <dependency>
16        <groupId>${project.groupId}</groupId>
17        <artifactId>bank-bankaccount-module</artifactId>
18        <version>${project.version}</version>
19      </dependency>
20      <dependency>
21        <groupId>${project.groupId}</groupId>
22        <artifactId>bank-bank-module</artifactId>
23        <version>${project.version}</version>
24      </dependency>
25      <dependency>
26        <groupId>org.apache.logging.log4j</groupId>
27        <artifactId>log4j-bom</artifactId>
28        <version>${log4j.version}</version>
29        <scope>import</scope>
30        <type>pom</type>
31      </dependency>
32    </dependencies>
33  </dependencyManagement>
34 </project>
```

Note that this BOM keeps in sync the versions of our modules (by using the standard Maven properties). This way, if we release our artifacts, clients can simply import this BOM, specifying the BOM version, and will be able to use our artifacts with the same versions. If one of our modules depends on another module of this project, it can simply specify such a dependency without the version by relying on this BOM. Log4j is also configured, by importing its BOM. Note that we use a property for specifying the version of Log4j so that we could easily build and test our project with a different version by overriding such property from the command line.

## 7.17.2 The bank parent POM

The parent POM is meant to be used by the sub-projects that need to share plugin configurations, besides dependencies. The parent itself uses the BOM as its parent so that the managed dependencies of the BOM become managed dependencies of the child projects.

Most of the configuration of the plugins of this parent is similar to the configurations we saw in the previous sections, so we will omit many parts in this section (refer to the actual source code of this example):

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <parent>
4     <groupId>com.example.bank</groupId>
5     <artifactId>bank-bom</artifactId>
6     <version>0.0.1-SNAPSHOT</version>
7     <relativePath>../bank-bom</relativePath>
8   </parent>
9   <artifactId>bank-parent</artifactId>
10  <packaging>pom</packaging>
11
12 <properties>
13   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
14   <maven.compiler.source>1.8</maven.compiler.source>
15   <maven.compiler.target>1.8</maven.compiler.target>
16   <junit.version>4.13</junit.version>
17   <assertj.version>3.15.0</assertj.version>
18 </properties>
19
20 <dependencies>
21   <dependency>
22     <groupId>junit</groupId>
23     <artifactId>junit</artifactId>
24     <version>${junit.version}</version>
25     <scope>test</scope>
26   </dependency>
27   <dependency>
28     <groupId>org.assertj</groupId>
29     <artifactId>assertj-core</artifactId>
30     <version>${assertj.version}</version>
31     <scope>test</scope>
32   </dependency>
33 </dependencies>
```

```
34
35 <build>
36   <pluginManagement>
37     <plugins>
38       <plugin>
39         <artifactId>maven-compiler-plugin</artifactId>
40         <version>3.8.0</version>
41         ... lock versions of main plugins as already seen
42       <plugin>
43         <groupId>org.pitest</groupId>
44         <artifactId>pitest-maven</artifactId>
45         ... similar to the configuration already seen
46       <plugin>
47         <groupId>org.jacoco</groupId>
48         <artifactId>jacoco-maven-plugin</artifactId>
49         ... similar to the configuration already seen
50         ... but without excluding the Main class (see below)
51     </plugins>
52   </pluginManagement>
53 </build>
54
55 <profiles>
56   <profile>
57     <id>jacoco</id>
58     ... as already seen
59   </profile>
60   <profile>
61     <id>mutation-testing</id>
62     ... as already seen
63   </profile>
64 </profiles>
65 </project>
```

Besides inheriting the managed dependencies from the BOM, this parent defines the test dependencies that are meant to be used by all its children.

### 7.17.3 The bankaccount project

This module is a child of the parent POM (but not physically in the directory structure). It only contains the implementation of BankAccount and its tests (already seen in this chapter). Test dependencies are inherited by the parent POM. This module does not require any other compile dependencies. The POM is thus as simple as this:

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <parent>
4     <groupId>com.example.bank</groupId>
5     <artifactId>bank-parent</artifactId>
6     <version>0.0.1-SNAPSHOT</version>
7     <relativePath>../bank-parent</relativePath>
8   </parent>
9   <artifactId>bank-bankaccount-module</artifactId>
10 </project>
```

Note that this project does not need any resources.

#### 7.17.4 The bank project

This module is another child of the parent POM and it depends on the bankaccount module since the class Bank and its tests use BankAccount. This module also requires Log4j. Such dependencies are simply mentioned without the version since they have been configured in the parent POM (through the BOM). This project also has resource directories (as already seen), for the configuration of logging.

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <parent>
4     <groupId>com.example.bank</groupId>
5     <artifactId>bank-parent</artifactId>
6     <version>0.0.1-SNAPSHOT</version>
7     <relativePath>../bank-parent</relativePath>
8   </parent>
9   <artifactId>bank-bank-module</artifactId>
10
11  <dependencies>
12    <dependency>
13      <groupId>${project.groupId}</groupId>
14      <artifactId>bank-bankaccount-module</artifactId>
15    </dependency>
16    <dependency>
17      <groupId>org.apache.logging.log4j</groupId>
18      <artifactId>log4j-api</artifactId>
19    </dependency>
20    <dependency>
21      <groupId>org.apache.logging.log4j</groupId>
```

```
22      <artifactId>log4j-core</artifactId>
23    </dependency>
24  </dependencies>
25 </project>
```

## 7.17.5 The bank report project

This is a simple POM project whose only aim is to use the JaCoCo report-aggregate goal. This will collect all the .exec files found in the dependencies of this project and create an aggregated report (again, HTML, XML, CSV). Thus, this project must specify as dependencies the projects involved in the coverage aggregated report.

This project is also a child of the parent POM, thus it inherits the main JaCoCo plugin configuration. It extends such a configuration by binding the report-aggregate goal to the verify phase. This binding is specified in the profile jacoco.



Note that the same profile id can be used across several projects. Activating a profile in a Maven build will activate that profile in all projects that define it.

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <parent>
4     <groupId>com.example.bank</groupId>
5     <artifactId>bank-parent</artifactId>
6     <version>0.0.1-SNAPSHOT</version>
7     <relativePath>../bank-parent</relativePath>
8   </parent>
9   <artifactId>bank-report</artifactId>
10  <packaging>pom</packaging>
11
12  <dependencies>
13    <dependency>
14      <!-- version taken from the bom -->
15      <groupId>${project.groupId}</groupId>
16      <artifactId>bank-bankaccount-module</artifactId>
17    </dependency>
18    <dependency>
19      <!-- version taken from the bom -->
20      <groupId>${project.groupId}</groupId>
21      <artifactId>bank-bank-module</artifactId>
22    </dependency>
```

```

23   </dependencies>
24
25   <profiles>
26     <profile>
27       <id>jacoco</id>
28       <build>
29         <plugins>
30           <plugin>
31             <!-- main configuration in pluginManagement -->
32             <groupId>org.jacoco</groupId>
33             <artifactId>jacoco-maven-plugin</artifactId>
34             <executions>
35               <execution>
36                 <id>report-aggregate</id>
37                 <phase>verify</phase>
38                 <goals>
39                   <goal>report-aggregate</goal>
40                 </goals>
41               </execution>
42             </executions>
43           </plugin>
44         </plugins>
45       </build>
46     </profile>
47   </profiles>
48 </project>

```



This goal is especially useful when tests are written in a project different from the one containing the SUTs. In such a case, you must specify the projects with the SUTs as compile dependencies and the ones containing the tests (and the code coverage execution data) as test dependencies.

## 7.17.6 The bank app project

This project is NOT a child of the parent POM. It only contains the `Main` class with the `main` method using a `Bank` object. Although it does not inherit from the parent POM, it imports the BOM in the `<dependencyManagement>` section, thus all the versions are of the common dependencies are already configured. In particular, this project depends on the `bank` module (and transitively on the `bankaccount` module) and on Log4j (thus it contains the resource `log4j2.xml`). In particular, it only depends on `log4j-core`, since `log4j-api` is a transitive dependency of `log4j-core`.<sup>14</sup>

---

<sup>14</sup>Indeed, we could have omitted `log4j-api` in the other examples as well, but for the sake of demonstration we specified them both.

The POM is also configured to create a FatJar and to execute it during the `verify` phase, as we have already seen in Section [Creating a FatJar](#).

This project has no test, thus it does not depend on JUnit and AssertJ. Since this project does not inherit from the parent POM, these test dependencies are not present at all.

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.example.bank</groupId>
4   <artifactId>bank-app</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6
7   <properties>
8     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
9     <maven.compiler.source>1.8</maven.compiler.source>
10    <maven.compiler.target>1.8</maven.compiler.target>
11  </properties>
12
13  <dependencyManagement>
14    <dependencies>
15      <dependency>
16        <groupId>com.example.bank</groupId>
17        <artifactId>bank-bom</artifactId>
18        <version>0.0.1-SNAPSHOT</version>
19        <scope>import</scope>
20        <type>pom</type>
21      </dependency>
22    </dependencies>
23  </dependencyManagement>
24
25  <dependencies>
26    <dependency>
27      <!-- version taken from the bom -->
28      <groupId>com.example.bank</groupId>
29      <artifactId>bank-bank-module</artifactId>
30    </dependency>
31    <dependency>
32      <!-- version taken from the bom -->
33      <groupId>org.apache.logging.log4j</groupId>
34      <artifactId>log4j-core</artifactId>
35    </dependency>
36  </dependencies>
37
```

```
38 <build>
39   <plugins>
40     <plugin>
41       <groupId>org.apache.maven.plugins</groupId>
42       <artifactId>maven-assembly-plugin</artifactId>
43       ... as already seen
44     <plugin>
45       <groupId>org.codehaus.mojo</groupId>
46       <artifactId>exec-maven-plugin</artifactId>
47       ... as already seen
48   </plugins>
49 </build>
50 </project>
```



Whether to make a project a child of a parent POM is up to you. You should consider tradeoffs between repeating a few configurations (like the Java compilation level) and inheriting unwanted dependencies and configurations.

## 7.17.7 The aggregator project

Finally, we have a separate project for building all the projects of this multi-module project (or only a subset). Note that this aggregator is completely independent of the parent project.

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.example.bank</groupId>
4   <version>0.0.1-SNAPSHOT</version>
5   <artifactId>bank-aggregator</artifactId>
6   <packaging>pom</packaging>
7
8   <modules>
9     <module>./bank-bom</module>
10    <module>./bank-parent</module>
11    <module>./bank-bankaccount-module</module>
12    <module>./bank-bank-module</module>
13  </modules>
14
15  <profiles>
16    <profile>
17      <id>app</id>
```

```
18      <modules>
19          <module>../bank-app</module>
20      </modules>
21  </profile>
22  <profile>
23      <id>jacoco</id>
24      <modules>
25          <module>../bank-report</module>
26      </modules>
27  </profile>
28 </profiles>
29 </project>
```

In a standard build, only the main modules are built. To build the bank app we have a profile, `id app`, which adds the `bank-app` project to the reactor (thus the `<modules>` specified in a profile are added to the main `<modules>` section). When the `jacoco` profile is activated we also add `bank-report` to the reactor (it makes no sense to build that project if we are not interested in code coverage).

Thus, this project is used to build the whole project, by possibly activating profiles. Of course, the profile `mutation-testing` can be activated even if not explicitly defined in this aggregator project.

As usual, several profiles can be activated at once. For example, the following command launched from the aggregator project builds all the projects, with code coverage and mutation testing (applied only on the projects where the corresponding plugins are active):

```
1 mvn clean verify -Papp,jacoco,mutation-testing
```

## 7.18 JUnit 5 and Maven

In order to use JUnit 5 in a Maven project, we need to add the corresponding dependency in the POM:

```
1 <dependency>
2     <groupId>org.junit.jupiter</groupId>
3     <artifactId>junit-jupiter</artifactId>
4     <version>5.7.0</version>
5     <scope>test</scope>
6 </dependency>
```

This will provide us with the Jupiter API, the runners, and also additional mechanisms like parameterized tests (see, for instance, Chapter [TDD](#), Section [TDD with JUnit 5 Parameterized Tests](#)).

The `maven-surefire-plugin`, which is responsible for running JUnit tests, supports JUnit 5 since version 2.22.0. This might not be the default version that comes with our Maven installation. The symptom will be that no JUnit 5 tests will be executed during the build. You might want to inspect the effective POM and see the version of such a plugin that comes by default in your Maven installation. However, as said before, it is better to have full control over the versions of Maven plugins. For this reason, we must make sure that `maven-surefire-plugin` is at least 2.22.0.<sup>15</sup> We have already done that in the `<pluginManagement>` section, as shown in Section [Plugin management](#):

```

1 <build>
2   <pluginManagement>
3     <plugins>
4       ...
5         <!-- Minimum version required to run JUnit 5 tests is 2.22.0 -->
6         <plugin>
7           <artifactId>maven-surefire-plugin</artifactId>
8           <version>2.22.1</version>
9         </plugin>
10        </plugins>
11      </pluginManagement>
12      ...

```

If in our project we have both JUnit 4 and JUnit 5 tests, then we have to add another dependency to the POM, in order to execute also JUnit 4 tests with the JUnit 5 vintage engine (see Chapter [JUnit](#), Section [JUnit 5](#)):

```

1 <dependency>
2   <groupId>org.junit.vintage</groupId>
3   <artifactId>junit-vintage-engine</artifactId>
4   <version>5.7.0</version>
5   <scope>test</scope>
6 </dependency>

```

This dependency will also add the JUnit 4 JAR so that JUnit 4 tests can be compiled.

JaCoCo can handle JUnit 5 tests out of the box. On the contrary, the PIT Maven plugin (Section [Configuring the PIT Maven plugin](#)) requires an additional dependency, `pitest-junit5-plugin`, to handle JUnit 5 tests (note that the version of the dependency is not the same as the one of the main `pitest-maven` plugin):

---

<sup>15</sup>If for any reason we wanted to use a previous version of `maven-surefire-plugin`, then we would have to specify `junit-platform-surefire-provider` as a dependency of the plugin.

```
1 <build>
2   <pluginManagement>
3     <plugins>
4       ...
5       <plugin>
6         <groupId>org.pitest</groupId>
7         <artifactId>pitest-maven</artifactId>
8         <version>1.5.2</version>
9         <dependencies>
10        <dependency>
11          <groupId>org.pitest</groupId>
12          <artifactId>pitest-junit5-plugin</artifactId>
13          <version>0.12</version>
14        </dependency>
15      </dependencies>
16      <configuration>
17      ...
```

## 7.19 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 following 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 data 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 released artifacts are meant to stay forever.

The deployment phase is out of the scope of this book.

## 7.20 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):

```
1 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, 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.,

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

## 7.21 Beyond Java 8

If we want to develop a Java application that uses Java 8 features, like lambdas, we need at least Java 8 installed in our system. However, we could use a more recent version of Java, say Java 11, to develop a Java application that is expected to run also on Java 8 (as the minimal version).

In this chapter, we learned that we can enforce the Java compilation setting. For example, even if we are using Java 11, we force the Java compiler to produce byte code that can be run also on Java 8:

```

1 <properties>
2   <maven.compiler.source>1.8</maven.compiler.source>
3   <maven.compiler.target>1.8</maven.compiler.target>
4 </properties>
```

With such a setting, which is also used by Eclipse to configure the Java 8 compatibility level, if we try to use a Java feature introduced after Java 8, like the `var` keyword (introduced in Java 10), we will get a compilation error. Even if we are using Java 11, we can be sure we will not use, inadvertently, features that are not available in Java 8.<sup>16</sup>

Unfortunately, setting the source and target options does not guarantee that our code runs on a JRE with the specified version. The pitfall is the unintended usage of APIs that only exist in later JREs. In fact, the Java compiler compiles against the most recent version of the platform APIs. If we made such a mistake, our compiled code would fail at runtime with a linkage error.

For example, with the above settings, we can write this Java code:

```

1 import java.util.List;
2
3 public class App {
4     public static void main( String[] args ) {
5         List<String> words = List.of("Hello", "World!");
6         words.forEach(System.out::println);
7     }
8 }
```

Here we are only using Java 8 linguistic features and the target option ensures that the byte code can be executed with Java 8.

Let's now try to run the produced `.class` file with a Java 8 virtual machine. We get this error:

```

1 Exception in thread "main" java.lang.NoSuchMethodError: java.util.List.of(Ljava/lang\
2 /Object;Ljava/lang/Object;)Ljava/util/List;
3         at com.mycompany.app.App.main(App.java:13)
```

In fact, `List.of` is an API introduced in Java 9, thus it is not present in the Java 8 runtime library.

To avoid these problems, in Java 9, <http://openjdk.java.net/jeps/247>, the `-N` argument of the Java compiler has been introduced. This makes sure we use only the APIs present in version N.

---

<sup>16</sup>Eclipse also provides a quickfix of the shape “Change project compliance and JRE to 10”. However, this is not our goal. We want to use Java 11 to develop a Java 8 application.

Otherwise, a compilation error is issued. The `maven-compiler-plugin` can use this flag since version 3.6.0, and it is configurable with the property `maven.compiler.release`.

Thus, we must make sure to use this version of the compiler plugin (we configure it in the `<pluginManagement>` section). Then, we use the `maven.compiler.release`, which replaces the previous properties `maven.compiler.source` and `maven.compiler.target`:

```
1 <properties>
2   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3   <maven.compiler.release>8</maven.compiler.release>
4 </properties>
5
6 <build>
7   <pluginManagement>
8     <plugins>
9       <plugin>
10      <!-- maven.compiler.release introduced in 3.6.0 -->
11      <artifactId>maven-compiler-plugin</artifactId>
12      <version>3.6.0</version>
13    </plugin>
14  </plugins>
15 </pluginManagement>
16 </build>
```

This instructs the Java compiler to check for API usage that is not part of the Java 8 platform.

If we now run the Maven build, the compiler will issue an error:

```
1 [ERROR] src/main/java/com/mycompany/app/App.java:[13,34] cannot find symbol
2   symbol:   method of(java.lang.String,java.lang.String)
3   location: interface java.util.List
```

Let's update the Eclipse project and the error appears also in Eclipse.



In Eclipse, independently from Maven, this could be manually turned on in the preferences (i.e., globally in the workspace or for a single project) in the section **Java Compiler**, option “Use ‘–release’ option”.

Note that the `release` argument has been introduced in the Java compiler in version 9. Running the Maven build with JDK 8 will make the build fail with this error:

```
1 [ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.6.0:\n2 compile (default-compile) on project:\n3 Fatal error compiling: invalid flag: --release
```

As a final note, in Eclipse you will get the following warnings:

```
1 Build path specifies execution environment JavaSE-1.8. There are no JREs installed i\\\n2 n the workspace that are strictly compatible with this environment.\n3\n4 The compiler compliance specified is 1.8 but a JRE 11 is used
```

You can either disable these warnings or fix them by using the preference **Java → Installed JREs → Execution Environment**. You can configure JavaSE-1.8 to point to the installed Java 11.

## 7.22 Reproducibility in the build

When applying build automation, e.g., with Maven, it is crucial to ensure a high level of reproducibility. Compiling and testing a specific version of the codebase should issue the same results.

Maven implements reproducibility by fostering the explicit version specification of dependencies and plugins. Using the Maven wrapper, described in the previous section, increments reproducibility by using a specific Maven version (with all its default plugins' versions bound to that specific version of the Maven distribution).

# 8. Mocking

In real-life applications, even a single class may delegate some tasks to other types. That is, such a class has dependencies (or collaborators). For example, it could query a database or it could use a remote service.

We want to write unit tests to test a single component in isolation, but how do we deal with dependencies? Tests might become complex: we should deal with a real database and this requires time; we should interact with a (remote) server during the tests and this requires an Internet connection.

This violates the main ideas of unit tests. Indeed, unit tests should test a single unit independently from the other dependencies. Moreover, unit tests should be fast, and dealing with a real database or a real remote service would undermine the speed of unit tests.

Besides, we might not have actual implementations for dependencies: They could be implemented by other team members and their implementations are not finished yet. We would be blocked!

If we want to be able to write unit tests for a single component in isolation, we must first start with an assumption: We must assume that the dependencies of the SUT:

- are implemented correctly (even by someone else) and
- they do what they're meant to do.

For internal dependencies, we assume they're already tested, or they will be tested when implemented. Indeed, we want to be able to test our SUT even if its dependencies are not finished yet. For external dependencies, we trust their implementation.

Summarizing, in any case, when writing unit tests, actual implementations of dependencies are not our business!

Recall that unit tests are meant to make us concentrate on that specific unit. Thus we must make sure that that unit behaves correctly, assuming that all its collaborators behave correctly. Will the untested component still work when integrated with real dependencies? This is not an issue that unit tests are meant to solve. In fact, we'll then have to write a different kind of test (Chapter [Integration tests](#)).

For unit tests, we should get rid of actual implementations of dependencies, and replace them with some fake implementations that only “impersonate” and simulate the real implementations. We directly manipulate such replacements to recreate the context that allows us to write unit tests for our SUT. Such replacements are called **test doubles**. They are implementations that stand-in for the real implementations. There are several categories of test doubles ([Mes07](#)). In this book we concentrate on **mocks**.

Thanks to mocks, we don't depend on components developed by other members and we can work on our classes even if dependencies are not yet implemented. Moreover, we can avoid depending on a real database or a remote service.

Since we assume that dependencies behave correctly, we make mocks do what we need them to do to test a specific scenario of our SUT. And we test that our SUT works correctly when dependencies act in a specific way. Since we do not use real implementations of dependencies, making them behave in a specific way for our testing scenario should be straightforward.

Our unit tests will be as fast as they should be.

Of course, the possibility of replacing actual dependencies with mocks relies on the fact that our SUT expects the actual implementation to be passed from the outside, e.g., at constructor time. Otherwise, it won't be easy (if possible at all) to replace dependencies with mocks. Moreover, the SUT should interact with dependencies through interfaces (or abstract classes). Once again, TDD will force us to keep our code modular, or we won't be able to easily test it.

To make the mocks behave in the desired way, we **stub** their methods that our SUT calls, so that they behave as we need them to behave in that specific scenario. The idea is that the stubbed implementation is only useful for our tests.

For example,

- Our SUT queries the database, through an interface `Database` using its method `getAllElements()`;
- We want to test the behavior of the SUT when the database returns a list with a single element;
- We mock the database and we stub its method `getAllElements()` so that it returns a list with a single element; then we test that our SUT does what we expect to do.

If the SUT uses the dependency through an interface and gets the actual implementation of the dependency passed from the outside, we can do mocking and stubbing manually.

For example, let's say that this is the available interface

```
1 public interface Database {  
2     List<String> getAllElements();  
3 }
```

and this is the component we want to test

```
1 public class MyComponent {
2     private Database database;
3
4     public MyComponent(Database database) {
5         this.database = database;
6     }
7
8     public Object myMethod() {
9         Object result = null;
10        List<String> list = database.getAllElements();
11        // do something with list and prepare the result ...
12        return result;
13    }
14 }
```

We could write a unit test for the scenario when the database returns a list with a single element as follows:

```
1 public class MyComponentTest {
2     @Test
3     public void testMyMethodWhenDBReturnsOneElement() {
4         Database database = new Database() {
5             @Override
6             public List<String> getAllElements() {
7                 List<String> list = new ArrayList<String>();
8                 list.add("foo");
9                 return list;
10            }
11        };
12        MyComponent myComponent = new MyComponent(database);
13        Object result = myComponent.myMethod();
14        // do assertions on the result
15    }
16 }
```

We manually create a fake implementation (mock) of the `Database` and we stub its method. Note that we can do that because our SUT expects the actual implementation to be provided by clients.

Manual mocking and stubbing is then possible but

- it requires some additional code in the tests (and this was a really simple case; what if the interface has many methods?);

- the test tends to be less readable since the mocking and stubbing is not directly evident by having a look at the test (this could be avoided by creating the mock and stub in a separate method, but again this requires some additional programming).

On the contrary, we would like to have a mechanism to easily perform mocking and stubbing, keeping the tests readable and concentrating on our SUT and its tests, instead of wasting time writing code for the mock implementation. Recall that TDD cycles should be fast.

For these reasons, in this chapter, we use a mocking framework, which solves all the above problems.

## 8.1 Testing state and testing interactions

Up to now, we have always written tests for code that

- returns something, i.e., methods with a return type;
- changes the state of the instances of the SUT.

The tests for this kind of code **test the state** of our code: they verify that

- the value returned by a method under test is as expected or
- after calling a method under test the state of an object changed as expected or
- a combination of both.

However, when writing tests for a SUT interacting with collaborators, we typically also need to **test the interaction** of our code with collaborators. That is, the tests verify that our code under test calls specific methods of collaborators

- a specific number of times (or never, in certain circumstances)
- passing specific arguments.

For example, our component processes elements returned by the database and then calls another method on the database `update(arg)`. We want to test that such a method is called only once with specific arguments. Alternatively, we must test that it is never called under specific circumstances.

This would require us implementing more complex mocks, and stubbing their methods so that they keep track of the invocation and the passed arguments. This involves more programming effort, and tests would tend to be less readable.

For example, given this interface

```
1 public interface Database {  
2     List<String> getAllElements();  
3     void update(String arg);  
4 }
```

We should manually mock the Database, stub its methods and in particular, keep track of invocation of update. This could be a possible implementation of the test:

```
1 public class MyComponentTest {  
2     private final static class MockedDatabase implements Database {  
3         String arg = null;  
4  
5         @Override  
6         public List<String> getAllElements() {  
7             List<String> list = new ArrayList<String>();  
8             list.add("foo");  
9             return list;  
10        }  
11  
12        @Override  
13        public void update(String arg) {  
14            // keep track of the passed argument  
15            this.arg = arg;  
16        }  
17    }  
18  
19    @Test  
20    public void testMyMethodCallsUpdateWithExpectedArg() {  
21        MockedDatabase database = new MockedDatabase();  
22        MyComponent myComponent = new MyComponent(database);  
23        myComponent.myMethod();  
24        // check that the method update has been called  
25        assertEquals("bar", database.arg);  
26        // or that it has never been called  
27        assertNull(database.arg);  
28    }  
29 }
```

Note the effort required to keep track of method invocation; in particular, in this implementation, we do not keep track of the times the method is called. Moreover, it is not completely clear from the assertions that we want to verify the interactions between our SUT and the mocked dependency.

A mocking framework solves also the above problems, by providing an easy and readable way to verify the interactions with mocks.

## 8.2 Mockito: an overview

In this book, we'll use **Mockito**, one of the most used frameworks for mocking, <https://site.mockito.org/>, which is easy to use and allows us to write very readable tests.

In this section, we'll give a very brief introduction to the features of Mockito. The next section will show many more features with a running example.

In a Maven project we add this (test) dependency:

```
1 <dependency>
2   <groupId>org.mockito</groupId>
3   <artifactId>mockito-core</artifactId>
4   <version>3.2.4</version>
5   <scope>test</scope>
6 </dependency>
```

### 8.2.1 Creating a mock

Creating a mock with Mockito is straightforward:

```
1 import static org.mockito.Mockito.*;
2
3 ...
4
5 Database db = mock(Database.class);
```

You can avoid the “static import” if you set the “Favorites” as shown in Chapter *Eclipse*, Section *Content Assist*.

The `mock` method, given any Java class or interface, creates a mocked implementation. The default implementation of the methods simply returns the default values of returned types; void methods are simply empty. Methods returning a collection will be implemented returning an empty collection.

Compare this with the manual implementation of the fake database seen in the introduction.

### 8.2.2 Method stubbing

On a mocked object, methods can be stubbed by using the fluent API of Mockito (recall the `import static` above):

```
1 List<String> list = new ArrayList<String>();
2 list.add("foo");
3 when(db.getAllElements()).thenReturn(list);
```

Note that this API is statically typed. For example, returning a `List<Integer>` would result in a compilation error.

### 8.2.3 Interaction verification

After exercising our methods (which are meant to interact with a mocked object), we can verify that some methods have been called (one or a specific amount of times) on the mocked object with specific arguments (or without specifying the actual arguments, but only their types). Alternatively, we can verify that some methods have never been called.

```
1 verify(db).update("bar");
2 verify(db, times(2)).update(anyString());
3 verify(db, never()).update(anyString());
```

You see that verifying interactions is a matter of using the fluent API of Mockito. Compare this with our previous manual implementation of interaction verification.

## 8.3 Mockito: a tutorial

We will show the main features of Mockito with a running example. Besides the Maven dependency for Mockito shown in the previous section, we'll also use AssertJ in our tests as done previously (so make sure to add it as a test Maven dependency).

We want to implement features for paying employees. Such employee objects are stored in a database and the payment takes place through a remote service. In this example, the database and the bank service provide an interface. We also assume that there is no implementation available of such interfaces. We will mock the interfaces when implementing our SUT with TDD.

In this example, we are going to test both the state and the interactions. Testing both aspects is crucial to verify the correct behavior of the code we are going to write. As you will see, testing the state is possible because the method we are going to write returns something. However, this is not enough for verifying the correct behavior, since most of the logic of our method will be delegated to collaborators, thus we must make sure that the interactions with collaborators are correct. Probably, that's the most interesting part. We could implement our method as `void` and see that it would still be valuable, thanks to the interactions with collaborators. However, for demonstration, we implement it so that it returns something to see how to test both the state (with already known mechanisms) and the interactions (with Mockito).

Let's say that there are already some existing classes and interfaces in our project.

The `Employee` is a class for our domain model

```
1 package com.example;
2
3 public class Employee {
4
5     private String id;
6     private double salary;
7     private boolean paid;
8
9     public Employee(String id, double salary) {
10         this.id = id;
11         this.salary = salary;
12     }
13
14     // getters and setters... and possibly equals and hashCode
15 }
```



Classes for representing domain model objects are very good candidates for being excluded from code coverage: their methods have no logic (`equals` and `hashCode` can be generated with the IDE), so we should not even test them.

The interface `BankService` abstracts from the actual implementation of such a service, which is meant to access the bank remotely.

```
1 package com.example;
2
3 public interface BankService {
4     void pay(String id, double amount);
5 }
```

Our domain model objects are stored in a database. We abstract from the database implementation, using the `EmployeeRepository` interface:

```
1 package com.example;
2
3 public interface EmployeeRepository {
4     List<Employee> findAll();
5 }
```

## The Repository pattern

The Repository pattern ([Eva03](#)) is a way to implement the Data Access Layer (DAL). In a layered architecture, meant to increment the modularity of the code base, this layer implements the logic for accessing the database. The main feature of the Repository pattern is that it acts as a collection of objects of a certain type. Typically a repository provides methods for accessing and modifying such a collection, abstracting from the actual storage. Note that it also abstracts from the details for mapping Java objects to database records. In Chapter *Integration tests*, we will write an implementation of a repository.

In this very simple example, our repository only provides a method for accessing such a collection; in a real application, it should also provide methods for saving objects and for querying the collection.

Let's start creating a test case, `EmployeeManagerTest`, following the procedures already seen in the previous chapters.

```
1 package com.example;
2
3 import static org.assertj.core.api.Assertions.*;
4
5 import org.junit.Test;
6
7 public class EmployeeManagerTest {
8
9     @Test
10    public void testPayEmployeesWhenNoEmployeesArePresent() {
11        EmployeeManager employeeManager = new EmployeeManager();
12        int number_of_payments = employeeManager.payEmployees();
13        assertThat(number_of_payments).isZero();
14    }
15
16 }
```

The test does not compile since we still haven't implemented our SUT `EmployeeManager`; using Eclipse quick fixes, as shown before:

- create the class `EmployeeManager` in the same package in the source folder `src/main/java`;
- create the method `payEmployees` in that class.

With the default implementation, the test will compile and succeed.



For saving some space we wrote the whole test before implementing the SUT. By following TDD strictly, we should have stopped as soon as the test did not compile for the missing `EmployeeManager` class, create it, and then go on writing the rest of the test until we got the next compilation error.

Before going on, let's refactor the test, by putting the creation of the SUT in a `@Before` method. Use the Eclipse tools for such a refactoring as already seen in the previous chapters.



Use the “Find Actions” (`Ctrl + 3`) for searching for the appropriate refactoring, e.g., “Convert Local Variable to Field” (see Chapter *Eclipse*, Section *Find Actions*).

```
1 public class EmployeeManagerTest {  
2  
3     private EmployeeManager employeeManager;  
4  
5     @Before  
6     public void setup() {  
7         employeeManager = new EmployeeManager();  
8     }  
9  
10    @Test  
11    public void testPayEmployeesWhenNoEmployeesArePresent() {  
12        int number_of_payments = employeeManager.payEmployees();  
13        assertThat(number_of_payments).isZero();  
14    }  
15  
16 }
```

Let's now start writing tests and code for our manager when it interacts with the database. We must create a mock of `EmployeeRepository`, pass it to the constructor of our SUT, and use it in our method. The mocks we'll create in our test will be fixtures of the test as well; the initialization will then be performed in the `setup` method:

```
1 import static org.mockito.Mockito.*;
2 ...
3 public class EmployeeManagerTest {
4
5     private EmployeeManager employeeManager;
6
7     private EmployeeRepository employeeRepository;
8
9     @Before
10    public void setup() {
11        employeeRepository = mock(EmployeeRepository.class);
12        employeeManager = new EmployeeManager(employeeRepository);
13    }...
```

The test class does not compile because there is no such constructor for our SUT. We create it with the quick fix, and then, by using “Quick Assist” (Ctrl+1) on the new constructor’s parameter, we assign the parameter to a new field (see Chapter *Eclipse*, Section *Quick Assist*):

```
1 public class EmployeeManager {
2
3     private EmployeeRepository employeeRepository;
4
5     public EmployeeManager(EmployeeRepository employeeRepository) {
6         this.employeeRepository = employeeRepository;
7     }
8     ...
```

The test class now compiles.

Let’s write a test for the case when the database contains one employee:

```
1 @Test
2 public void testPayEmployeesWhenOneEmployeeIsPresent() {
3     assertThat(employeeManager.payEmployees()).isEqualTo(1);
4 }
```

The test is expected to fail. In fact, `payEmployees` does not currently use the repository.

We must stub the method `EmployeeRepository.findAll` so that it returns a list with a single employee.

```
1 import static java.util.Arrays.asList;
2 ...
3 @Test
4 public void testPayEmployeesWhenOneEmployeeIsPresent() {
5     List<Employee> employees = new ArrayList<>();
6     employees.add(new Employee("1", 1000));
7     when(employeeRepository.findAll()).thenReturn(employees);
8     assertThat(employeeManager.payEmployees()).isEqualTo(1);
9 }
```

The method `payEmployees` of our SUT can then be updated so that it retrieves the list of employees and returns the size of the list. This should be enough for making the test pass.

```
1 public int payEmployees() {
2     return employeeRepository.findAll().size();
3 }
```

All tests pass now.

Let's refactor the test class so that also the list is a fixture of our test case; the stubbing of the method `EmployeeRepository.findAll` will take place in the `setup` method. This way, the tests will simply add elements to the list `employees` as they see fit:

```
1 // example of WRONG refactoring
2 public class EmployeeManagerTest {
3     ...
4     private List<Employee> employees;
5
6     @Before
7     public void setup() {
8         employees = new ArrayList<>();
9         employeeRepository = mock(EmployeeRepository.class);
10        when(employeeRepository.findAll()).thenReturn(employees);
11        employeeManager = new EmployeeManager(employeeRepository);
12    }
13    ...
14    @Test
15    public void testPayEmployeesWhenOneEmployeeIsPresent() {
16        // why do we add something to this list?!
17        employees.add(new Employee("1", 1000));
18        assertThat(employeeManager.payEmployees()).isEqualTo(1);
19    }...
```

With this refactoring, we avoid in the next test methods to stub `findAll`. The method is stubbed before any test method by always returning the `employees` list and in test methods we simply add employees directly to that list. Unfortunately, the test method (and the test methods we are going to write) will be less readable. In fact, it will be harder to understand how the mocked dependency is setup. We will have to go to the `setup` method o understand that what is added to the list will be the result of the stubbed method. Tests must be clean code, but it is even more important that tests are immediately readable, even at the price of a few code duplication (as shown also in Chapter [JUnit](#), Section [Beware of code duplication removal in tests](#)).

For the above reasons, we undo that refactoring, and we will always make the stubbing explicitly in the single tests. We can clean up the code anyway, by inlining the creation of the list when we stub. This way, the test is readable and the stubbing is evident:

```
1 import static java.util.Arrays.asList;
2 ...
3 public class EmployeeManagerTest {
4     ...
5
6     @Before
7     public void setup() {
8         employeeRepository = mock(EmployeeRepository.class);
9         employeeManager = new EmployeeManager(employeeRepository);
10    }
11    ...
12    @Test
13    public void testPayEmployeesWhenOneEmployeeIsPresent() {
14        when(employeeRepository.findAll())
15            .thenReturn(asList(new Employee("1", 1000)));
16        assertThat(employeeManager.payEmployees()).isEqualTo(1);
17    }...
```

Note that mocking in the setup method is admissible since mocking is part of the setup anyway.

It might also be the case to refactor the very first test, and make the stubbing with an empty list explicit, even it is redundant since Mockito would already stub the method like that by default. The explicit stubbing with an empty list would make our test clearer:

```
1 import static java.util.Collections.emptyList;
2 ...
3 public class EmployeeManagerTest {
4     ...
5
6     @Test
7     public void testPayEmployeesWhenNoEmployeesArePresent() {
8         when(employeeRepository.findAll())
9             .thenReturn(emptyList());
10        assertThat(employeeManager.payEmployees())
11            .isZero();
12    }...
```

The goal of our SUT method `payEmployees` is that it should pay all employees of the database by calling the `BankService`. Thus, we mock the `BankService` and we pass it to our SUT. Once again, to make the test compile, we must update the constructor, and assign the new parameter to a new field.

```
1 public class EmployeeManagerTest {
2     ...
3     private BankService bankService;
4
5     @Before
6     public void setup() {
7         employeeRepository = mock(EmployeeRepository.class);
8         bankService = mock(BankService.class);
9         employeeManager = new EmployeeManager(employeeRepository, bankService);
10    }...
11
12 public class EmployeeManager {
13
14     private EmployeeRepository employeeRepository;
15     private BankService bankService;
16
17     public EmployeeManager(EmployeeRepository employeeRepository,
18         BankService bankService) {
19         this.employeeRepository = employeeRepository;
20         this.bankService = bankService;
21     }...
```

Then we enhance our test so that it verifies that the method `BankService.pay` is effectively called with the expected arguments (the id of the employee and its salary):

```

1  @Test
2  public void testPayEmployeesWhenOneEmployeeIsPresent() {
3      when(employeeRepository.findAll())
4          .thenReturn(asList(new Employee("1", 1000)));
5      assertThat(employeeManager.payEmployees()).isEqualTo(1);
6      verify(bankService).pay("1", 1000);
7  }

```

The test fails: Mockito will tell us that there was no interaction:

```

1 Wanted but not invoked:
2 bankService.pay("1", 1000);
3 -> at testPayEmployeesWhenOneEmployeeIsPresent(EmployeeManagerTest.java)
4 Actually, there were zero interactions with this mock.

```

Let's fix our SUT so that this test passes:

```

1 public int payEmployees() {
2     List<Employee> employees = employeeRepository.findAll();
3     Employee employee = employees.get(0);
4     bankService.pay(employee.getId(), employee.getSalary());
5     return employees.size();
6 }

```

Now the test passes, but the previous test fails with an `IndexOutOfBoundsException`. Let's fix the method to make the two tests pass.

```

1 public int payEmployees() {
2     List<Employee> employees = employeeRepository.findAll();
3     if (!employees.isEmpty()) {
4         Employee employee = employees.get(0);
5         bankService.pay(employee.getId(), employee.getSalary());
6     }
7     return employees.size();
8 }

```



As done before, we apply TDD by proceeding with little steps, and by applying transformations. We always start with simple transformations and then proceed with more complex transformations. As you may expect, that `if` will become a `while` in the next TDD cycle (see Chapter [TDD](#)).

Let's write a test for the generic case when the database contains more than one element:

```
1 @Test
2 public void testPayEmployeesWhenSeveralEmployeesArePresent() {
3     when(employeeRepository.findAll())
4         .thenReturn(asList(
5             new Employee("1", 1000),
6             new Employee("2", 2000)));
7     assertThat(employeeManager.payEmployees()).isEqualTo(2);
8     verify(bankService, times(2)).pay(anyString(), anyDouble());
9 }
```



verify(mock) without further arguments is equivalent to verify(mock, times(1)).

We verify that the BankService is used exactly twice. For the moment we do not verify the actual arguments passed, by only checking the types of the passed arguments. The test will fail:

```
1 org.mockito.exceptionsverification.TooLittleActualInvocations:
2 bankService.pay(<any string>, <any double>);
3 Wanted 2 times:
4 -> at testPayEmployeesWhenSeveralEmployeeArePresent(EmployeeManagerTest.java)
5 But was 1 time:
6 -> at payEmployees(EmployeeManager.java)
```

Let's transform our method by turning the if into a while:

```
1 public int payEmployees() {
2     List<Employee> employees = employeeRepository.findAll();
3     for (Employee employee : employees) {
4         bankService.pay(employee.getId(), employee.getSalary());
5     }
6     return employees.size();
7 }
```

All tests pass now.

What if we wanted to verify the two method invocations on the BankService with actual arguments?

This is a possible solution, where we are not interested in verifying the order of the invocations (note that we intentionally verify the two invocations in reverse order):

```
1  @Test
2  public void testPayEmployeesWhenSeveralEmployeesArePresent() {
3      when(employeeRepository.findAll())
4          .thenReturn(asList(
5              new Employee("1", 1000),
6              new Employee("2", 2000)));
7      assertThat(employeeManager.payEmployees()).isEqualTo(2);
8      verify(bankService).pay("2", 2000);
9      verify(bankService).pay("1", 1000);
10     verifyNoMoreInteractions(bankService);
11 }
```

The last verification also makes sure that the two invocations of `pay` are the only interactions with the `BankService`.

## VerificationCollector

By default, a failure in a Mockito verification will make the test fail immediately, just like standard JUnit assert methods.

This behavior can be changed by using this Mockito JUnit rule:

```
1  import org.mockito.junit.MockitoJUnit;
2  import org.mockito.junit.VerificationCollector;
3  ...
4  @Rule
5  public VerificationCollector collector = MockitoJUnit.collector();
```

In this case, all verifications in the current test method are executed, and possible failures are reported only at the end of the test method.

If we want to verify also the order of invocation, we use the `inOrder` method to create an `InOrder` object, that we then use to perform the verifications:

```
1 import org.mockito.InOrder;
2 ...
3 @Test
4 public void testPayEmployeesInOrderWhenSeveralEmployeeArePresent() {
5     // an example of invocation order verification
6     when(employeeRepository.findAll())
7         .thenReturn(asList(
8             new Employee("1", 1000),
9             new Employee("2", 2000)));
10    assertThat(employeeManager.payEmployees()).isEqualTo(2);
11    InOrder inOrder = inOrder(bankService);
12    inOrder.verify(bankService).pay("1", 1000);
13    inOrder.verify(bankService).pay("2", 2000);
14    verifyNoMoreInteractions(bankService);
15 }
```

Note that now we verify that employees are paid in the order they are retrieved from the list. Try to invert the two `verify`s and the test will fail.

The `inOrder` method can be used with several mocks to verify the effective interactions with several mocks, as shown in this “artificial” test:

```
1 @Test
2 public void testExampleOfInOrderWithTwoMocks() {
3     // Just an example of invocation order verification on several mocks
4     when(employeeRepository.findAll())
5         .thenReturn(asList(
6             new Employee("1", 1000),
7             new Employee("2", 2000)));
8     assertThat(employeeManager.payEmployees()).isEqualTo(2);
9     InOrder inOrder = inOrder(bankService, employeeRepository);
10    inOrder.verify(employeeRepository).findAll();
11    inOrder.verify(bankService).pay("1", 1000);
12    inOrder.verify(bankService).pay("2", 2000);
13    verifyNoMoreInteractions(bankService);
14 }
```

This verifies that our SUT first interacts with the repository and then with the bank service. This test is for demonstration only and has no effective value for our project.

In case we need to perform further assertions on the arguments passed to the mocked objects methods we can use `ArgumentCaptor`:

```
1 import org.mockito.ArgumentCaptor;
2 ...
3 @Test
4 public void testExampleOfArgumentCaptor() {
5     // Just an example of ArgumentCaptor
6     when(employeeRepository.findAll())
7         .thenReturn(asList(
8             new Employee("1", 1000),
9             new Employee("2", 2000)));
10    assertThat(employeeManager.payEmployees()).isEqualTo(2);
11    ArgumentCaptor<String> idCaptor = ArgumentCaptor.forClass(String.class);
12    ArgumentCaptor<Double> amountCaptor = ArgumentCaptor.forClass(Double.class);
13    verify(bankService, times(2)).pay(idCaptor.capture(), amountCaptor.capture());
14    assertThat(idCaptor.getAllValues()).containsExactly("1", "2");
15    assertThat(amountCaptor.getAllValues()).containsExactly(1000.0, 2000.0);
16    verifyNoMoreInteractions(bankService);
17 }
```

Note that the AssertJ `containsExactly` also checks the order of the expected elements passed as arguments.

### 8.3.1 Spy

Sometimes we might need to use a real implementation class in our tests as a dependency for our SUT, but we still want to be able to stub its methods using the mechanisms of Mockito and to verify invocations on such a real implementation. We can do that with Mockito by “spying” on a real object.

Instead of using `mock` we use `spy` on an already created instance.

When invoking methods on a spied objects, the real methods are called, unless these are manually stubbed.

In our tests, we have been using real `Employee` instances. We do not need to stub its methods, but we might want to verify that some of its methods are called, with specific arguments.



Stubbing or verifying on an object that is not mocked, nor spied, raises an exception at run-time.

For example, let's say that once paid, an employee should have the boolean flag, `paid`, set to true. We could simply assert its field value with the getter, `isPaid`. However, we can express the expected behavior better by verifying that after invoking our SUT method, the method `setPaid(true)` has been called on the spied employee.

Of course, we could create a mock `Employee`, but then we should also stub all the getters that are called by our SUT. This makes no sense when we have an implementation. Moreover, it makes no sense to mock domain model classes (see also Section [What to mock](#)).

We then write this test, where we verify that `setPaid(true)` is called on the spied employee after paying the employee through the `BankService` (note the use of `InOrder`, explained above):

```
1  @Test
2  public void testEmployeeSetPaidIsCalledAfterPaying() {
3      Employee employee = spy(new Employee("1", 1000));
4      when(employeeRepository.findAll())
5          .thenReturn(asList(employee));
6      assertThat(employeeManager.payEmployees()).isEqualTo(1);
7      InOrder inOrder = inOrder(bankService, employee);
8      inOrder.verify(bankService).pay("1", 1000);
9      inOrder.verify(employee).setPaid(true);
10 }
```

The test fails, as expected. Let's make it work by fixing our method:

```
1  public int payEmployees() {
2      List<Employee> employees = employeeRepository.findAll();
3      for (Employee employee : employees) {
4          bankService.pay(employee.getId(), employee.getSalary());
5          employee.setPaid(true);
6      }
7      return employees.size();
8  }
```

Once again, note that the behavior specified by this test is finer than just checking whether the field `paid` is true: we want to verify that the field is set to `true` with its setter. The field could have already been `true`, and just checking its value with the getter is not what we want to make sure it happens.

For example, this test would succeed even without calling `setPaid` in `payEmployees`:

```

1  @Test
2  public void wrongTest() {
3      Employee employee = new Employee("1", 1000);
4      // set paid to true in advance
5      employee.setPaid(true);
6      when(employeeRepository.findAll())
7          .thenReturn(asList(employee));
8      assertThat(employeeManager.payEmployees()).isEqualTo(1);
9      verify(bankService).pay("1", 1000);
10     // the following will be true even if payEmployees
11     // never calls setPaid(true)
12     assertThat(employee.isPaid()).isTrue();
13 }

```

### 8.3.2 Spying and stubbing

We must pay attention when stubbing methods of a spied object. Suppose `o` is a spied object. If we do

```
1  when(o.m()).thenReturn(...)
```

we must take into consideration that the original method `m()` is called before our stubbed version. This could be fine in most cases. For example<sup>1</sup>, this succeeds:

```

1  @Test
2  public void testIsEmptyOnSpiedArrayList() {
3      // this has nothing to do with our case-study
4      // it's just a demonstration of spy and stubbing
5      List<String> list = spy(new ArrayList<>());
6      when(list.isEmpty()).thenReturn(false);
7      assertThat(list.isEmpty()).isFalse();
8 }

```

But if we write

---

<sup>1</sup>Just for the sake of demonstration of the current section subject, we'll spy an `ArrayList`. As said in Section [What to mock](#), this should be avoided in general.

```
1 @Test
2 public void testGetOnSpiedArrayList() {
3     // this has nothing to do with our case-study
4     // it's just a demonstration of spy and stubbing
5     List<String> list = spy(new ArrayList<>());
6     when(list.get(0)).thenReturn("foo");
7     // doesn't even get here
8     assertThat(list.get(0)).isEqualTo("foo");
9 }
```

we get an exception during the stubbing:

```
1 java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
2     at java.util.ArrayList.rangeCheck(ArrayList.java:657)
3     at java.util.ArrayList.get(ArrayList.java:433)
```

During the stubbing, `get(0)` is called on the original list, which is empty.

To avoid these problems on spied objects when stubbing, we should use a different stubbing mechanism:

```
1 doReturn(...).when(o).m()
```

instead of

```
1 when(o.m()).thenReturn(...)
```

Indeed, this test now succeeds:

```
1 @Test
2 public void testGetOnSpiedArrayList() {
3     // this has nothing to do with our case-study
4     // it's just a demonstration of spy and stubbing
5     List<String> list = spy(new ArrayList<>());
6     doReturn("foo").when(list).get(0);
7     assertThat(list.get(0)).isEqualTo("foo");
8 }
```

This alternative mechanism for stubbing can be used also with standard mocks, not only with spied objects.

However, `doReturn` is not statically checked concerning types: if we return an object of the wrong type we get an exception at runtime.

For example, this will throw a `ClassCastException`:

```

1  @Test
2  public void testGetWrongTypeOnEmptyOnSpiedArrayList() {
3      // this has nothing to do with our case-study
4      // it's just a demonstration of spy and stubbing
5      List<Employee> list = spy(new ArrayList<>());
6      // doReturn is not statically checked...
7      doReturn("foo").when(list).get(0);
8      // here we'll get a ClassCastException at runtime
9      Employee actual = list.get(0);
10     assertThat(actual).isEqualTo("foo");
11 }

```

Using `when...` would instead be statically checked by the compiler and this test, in fact, does not even compile:

```

1  @Test
2  public void testGetWrongTypeOnEmptyOnSpiedArrayList() {
3      // this has nothing to do with our case-study
4      // it's just a demonstration of spy and stubbing
5      List<Employee> list = spy(new ArrayList<>());
6      // COMPILE ERROR:
7      // The method thenReturn(Employee) is not applicable for the arguments (String)
8      when(list.get(0)).thenReturn("foo");
9      // here we'll get a ClassCastException at runtime
10     Employee actual = list.get(0);
11     assertThat(actual).isEqualTo("foo");
12 }

```

### 8.3.3 Stubbing and exceptions

As we saw in the previous chapters, unit tests should test all the paths of the SUT methods, including cases when exceptions are thrown. When using collaborators, we should also be able to reproduce cases when such collaborators can throw exceptions. Doing so without mocking might be hard. With mocking frameworks like Mockito instead, reproducing cases with exceptions is just a matter of stubbing methods of mocks accordingly.

With Mockito, we can do something like

```
1  when(o.m()).thenThrow(...new exception...);
```

In general, this stubbing syntax cannot be applied for `void` methods. For such cases, we must use the alternative syntax, which we have already seen in the previous section.

```
1 doThrow(...new exception...).when(o).m()
```

For example, the `BankService` methods are expected to throw exceptions, especially since this interface represents a remote service, which has to deal with network connections that are known to raise exceptions. To keep things simple, let's assume that `BankService.pay` can throw runtime exceptions, that is, unchecked exceptions.

For example, we write a test for the case when `BankService.pay` throws a `RuntimeException`.

```
1 @Test
2 public void testPayEmployeesWhenBankServiceThrowsException() {
3     Employee employee = spy(new Employee("1", 1000));
4     when(employeeRepository.findAll())
5         .thenReturn(asList(employee));
6     doThrow(new RuntimeException())
7         .when(bankService).pay(anyString(), anyDouble());
8     // number of payments must be 0
9     assertThat(employeeManager.payEmployees()).isZero();
10    // make sure that Employee.paid is updated accordingly
11    verify(employee).setPaid(false);
12 }
```

This test fails, with a `RuntimeException`, since our method does not catch it. Let's select the expressions in the loop inside `payEmployees` and select the context menu **Surround With** → **Try/catch Block** (or the shortcut **Shift+Alt+Z** and then **Try/catch Block**). We handle the exception in the `catch` block by logging it (we assume we are using Log4j in this project as well; otherwise, simply print the stack trace):

```
1 public int payEmployees() {
2     List<Employee> employees = employeeRepository.findAll();
3     for (Employee employee : employees) {
4         try {
5             bankService.pay(employee.getId(), employee.getSalary());
6             employee.setPaid(true);
7         } catch (RuntimeException e) {
8             LOGGER.error("Failed payment of " + employee, e);
9         }
10    }
11    return employees.size();
12 }
```

The test still fails, because we simply return the size of the list of employees, while we must keep track of the actual payments:

```

1 public int payEmployees() {
2     List<Employee> employees = employeeRepository.findAll();
3     int payments = 0;
4     for (Employee employee : employees) {
5         try {
6             bankService.pay(employee.getId(), employee.getSalary());
7             employee.setPaid(true);
8             payments++;
9         } catch (RuntimeException e) {
10             LOGGER.error("Failed payment of " + employee, e);
11         }
12     }
13     return payments;
14 }
```

Now the assertion on the returned integer passes, but the verification of `setPaid(false)` fails. Let's fix this as well:

```

1 public int payEmployees() {
2     List<Employee> employees = employeeRepository.findAll();
3     int payments = 0;
4     for (Employee employee : employees) {
5         try {
6             bankService.pay(employee.getId(), employee.getSalary());
7             employee.setPaid(true);
8             payments++;
9         } catch (RuntimeException e) {
10             LOGGER.error("Failed payment of " + employee, e);
11             employee.setPaid(false);
12         }
13     }
14     return payments;
15 }
```

Now the test succeeds.

### 8.3.4 Subsequent stubbing

Indeed, the test for the exception thrown by the `BankService` does not say anything about the case when an exception is thrown when paying a single employee but other employees can be paid successfully. Actually, our method already handles correctly this scenario: it catches the exception and tries to pay the other employees.

Let's write a test documenting this scenario. To do that, we can use another useful feature of Mockito: subsequent stubbing. Indeed, we can chain the stubbed implementations of a method. This is shown by the following artificial test:

```
1 @Test
2 public void testSubsequentStubbing() {
3     // this has nothing to do with our case-study
4     // it's just a demonstration of spy and stubbing
5     when(employeeRepository.findAll())
6         // only on first invocation
7         .thenReturn(Collections.emptyList())
8         // for subsequent invocations
9         .thenReturn(asList(new Employee("1", 1000)));
10    assertThat(employeeRepository.findAll()).isEmpty();
11    assertThat(employeeRepository.findAll()).hasSize(1);
12    assertThat(employeeRepository.findAll()).hasSize(1);
13 }
```

Of course, the same mechanism is available for the alternative syntax

```
1 doReturn(...).doReturn(...).when(o).m()
```

This is useful for recreating a scenario when the first invocation should throw an exception while other invocations should run without exceptions. We can use doNothing for subsequent stubbed calls, for example.

```
1 @Test
2 public void testOtherEmployeesArePaidWhenBankServiceThrowsException() {
3     Employee notToBePaid = spy(new Employee("1", 1000));
4     Employee toBePaid = spy(new Employee("2", 2000));
5     when(employeeRepository.findAll())
6         .thenReturn(asList(notToBePaid, toBePaid));
7     doThrow(new RuntimeException())
8         .doNothing()
9         .when(bankService).pay(anyString(), anyDouble());
10    // number of payments must be 1
11    assertThat(employeeManager.payEmployees()).isEqualTo(1);
12    // make sure that Employee.paid is updated accordingly
13    verify(notToBePaid).setPaid(false);
14    verify(toBePaid).setPaid(true);
15 }
```

As anticipated above, this test is expected to succeed with our current implementation.

### 8.3.5 Argument matchers

We saw that we can use actual expressions for arguments when stubbing method calls. We also used some predefined matchers such as `anyString()`, `anyDouble()`, etc. If we want to specify that we want to match any object of a given class, e.g., `Employee`, we use the matcher `isA`:

```
1 isA(Employee.class)
```

Alternatively, we can write a custom matcher by implementing `org.mockito.ArgumentMatcher` and passing an instance of that implementation to `org.mockito.ArgumentMatchers.argThat`. Since `ArgumentMatcher` is a functional interface, a lambda can be specified instead.

For example, this is a variant of the previous test, where, instead of subsequent stubbing, we use an argument matcher to throw an exception:

```
1 import static org.mockito.Mockito.ArguementMatchers.argThat;
2 ...
3 @Test
4 public void testArgumentMatcherExample() {
5     // equivalent to the previous test, with argument matcher
6     Employee notToBePaid = spy(new Employee("1", 1000));
7     Employee toBePaid = spy(new Employee("2", 2000));
8     when(employeeRepository.findAll())
9         .thenReturn(asList(notToBePaid, toBePaid));
10    doThrow(new RuntimeException())
11        .when(bankService).pay(
12            argThat(s -> s.equals("1")),
13            anyDouble());
14    // number of payments must be 1
15    assertThat(employeeManager.payEmployees()).isEqualTo(1);
16    // make sure that Employee.paid is updated accordingly
17    verify(notToBePaid).setPaid(false);
18    verify(toBePaid).setPaid(true);
19 }
```

## 8.4 Alternative ways of initializing mocks and other elements

For the moment we initialize mocks in the `@Before` method and we pass the mocks to our SUT instance. Mockito provides an easier (and more readable) way of initializing mocks, using Java

annotations. This mechanism also allows us to automatically initialize our SUT instance with such mocks. These annotations can be used to easily create spied objects and argument captors.

We will demonstrate these annotations by creating an alternative test case, `EmployeeManagerAlternativeTest`. Besides the use of such annotations, we will also see how this impacts the single test methods. We will write in this new test case class a few tests we have already seen to show the difference.

The idea is to annotate the test case fields that are used as mocks with the annotation `@Mock`:

```
1 ...
2 import org.mockito.Mock;
3
4 public class EmployeeManagerAlternativeTest {
5
6     @Mock
7     private EmployeeRepository employeeRepository;
8
9     @Mock
10    private BankService bankService;
11 ...
```

Then, the fixture field for our SUT is annotated with `@InjectMocks`:

```
1 import org.mockito.InjectMocks;
2 ...
3 @InjectMocks
4 private EmployeeManager employeeManager;
5 ...
```

Then, in our `@Before` method, we just need to ask Mockito to use these annotations to create mocks and inject them in our SUT by calling `MockitoAnnotations.initMocks` (the stubbing can be done as usual once we are sure that mocks have been initialized):

```
1 ...
2 import org.mockito.MockitoAnnotations;
3 ...
4 @Before
5 public void setup() {
6     MockitoAnnotations.initMocks(this);
7 }
```

Compare this setup with the one we had previously written. Summarizing, Mockito will create the mocks for each field of the test case class that is annotated with `@Mock`; then, it will create an instance

of fields annotated with `@InjectMocks`, by “injecting” the available mocks into the SUT, by looking at its constructor (it could also directly inject the mocks in the SUT fields, or by using SUT setters).<sup>2</sup>

In version 3.4 of Mockito, `initMocks` has been deprecated in favor of the new `openMocks` method. This new method returns a `java.lang.AutoCloseable`, which should be closed properly, e.g., in an `@After` method:

```

1 ...
2 import org.mockito.MockitoAnnotations;
3 ...
4 private AutoCloseable closeable;
5
6 @Before
7 public void setup() {
8     closeable = MockitoAnnotations.openMocks(this);
9 }
10
11 @After
12 public void releaseMocks() throws Exception {
13     closeable.close();
14 }
```

Mockito provides other annotations for its mechanisms. For example, by annotating a field with `@Captor` we have an `ArgumentCaptor` automatically created when calling `initMocks` (or `openMocks` when using newer versions of Mockito). Then, when we need the argument captors in the tests we just used the fields, without having to explicitly create them as we used to do before:

```

1 ...
2 import org.mockito.ArgumentCaptor;
3 ...
4 @Captor
5 private ArgumentCaptor<String> idCaptor;
6
7 @Captor
8 private ArgumentCaptor<Double> amountCaptor;
9 ...
10
11 @Test
12 public void testExampleOfArgumentCaptor() {
13     // Just an example of ArgumentCaptor
14     when(employeeRepository.findAll())
        .thenReturn(asList(
```

---

<sup>2</sup>Recall that we had already mentioned dependency injection in Chapter [JUnit](#), box [Dependency Injection](#). In a later chapter, Chapter [Learning tests](#), Section [Dependency Injection with Google Guice](#), we will see another mechanism for dependency injection.

```
15         new Employee("1", 1000),  
16         new Employee("2", 2000)));  
17     assertThat(employeeManager.payEmployees()).isEqualTo(2);  
18     // ArgumentCaptor<String> idCaptor = ArgumentCaptor.forClass(String.class);  
19     // ArgumentCaptor<Double> amountCaptor = ArgumentCaptor.forClass(Double.class);  
20     verify(bankService, times(2)).pay(idCaptor.capture(), amountCaptor.capture());  
21     assertThat(idCaptor.getAllValues()).containsExactly("1", "2");  
22     assertThat(amountCaptor.getAllValues()).containsExactly(1000.0, 2000.0);  
23     verifyNoMoreInteractions(bankService);  
24 }  
25 ...
```

Similarly to `@Mock`, the annotation `@Spy` can be used to annotate test case fields. Mockito, when calling `MockitoAnnotations.initMocks` (or `openMocks` when using newer versions of Mockito), will create real instances, unless the fields are not already initialized, and then create the corresponding spied objects:

```
1 ...  
2 import org.mockito.Spy;  
3 ...  
4 @Spy  
5 private Employee notToBePaid = new Employee("1", 1000);  
6 ...  
7 @Spy  
8 private Employee toBePaid = new Employee("2", 2000);  
9 ...  
10 @Test  
11 public void testOtherEmployeesArePaidWhenBankServiceThrowsException() {  
12     when(employeeRepository.findAll())  
13         .thenReturn(asList(notToBePaid, toBePaid));  
14     doThrow(new RuntimeException())  
15         .doNothing()  
16         .when(bankService).pay(anyString(), anyDouble());  
17     // number of payments must be 1  
18     assertThat(employeeManager.payEmployees()).isEqualTo(1);  
19     // make sure that Employee.paid is updated accordingly  
20     verify(notToBePaid).setPaid(false);  
21     verify(toBePaid).setPaid(true);  
22 }
```

Tests that were using spied objects can now simply use the fields annotated with `@Spy`.

Instead of manually calling `MockitoAnnotations.initMocks` (or `openMocks` when using newer versions of Mockito), in our `@Before` method we can use the specific Mockito JUnit runner:<sup>3</sup>

```
1 import org.junit.runner.RunWith;
2 import org.mockito.junit.MockitoJUnitRunner;
3
4 @RunWith(MockitoJUnitRunner.class)
5 public class EmployeeManagerAlternativeTest {
6 ...
7     @Before
8     public void setup() {
9         // Mockito annotations are automatically processed
10    }
11 ...
```

The variant `MockitoJUnitRunner.StrictStubs.class` automatically detects stubbing argument mismatches.

Instead of using the runner, we can have Mockito annotations automatically initialized by using the Mockito JUnit rule (once again, with the settings for detecting stubbing argument mismatches):

```
1 ...
2 import org.junit.Rule;
3 import org.mockito.junit.MockitoJUnit;
4 import org.mockito.junit.MockitoRule;
5 import org.mockito.quality.Strictness;
6
7 public class EmployeeManagerAlternativeTest {
8
9     @Rule
10    public MockitoRule rule =
11        MockitoJUnit.rule().strictness(Strictness.STRICT_STUBS);
12
13     @Before
14     public void setup() {
15         // Mockito annotations are automatically processed
16    }
17 ...
```

---

<sup>3</sup>We have first seen JUnit runners in Chapter [TDD](#), Section [TDD with JUnit 4 Parameterized Tests](#).

## 8.5 Mockito BDD style

Mockito provides an alternative API based on the **Behavior-Driven Development (BDD)** style (see Chapter *Testing*, Section *Behavioral-Driven Development (BDD)*), which fosters the writing of tests with 3 sections: **given**, **when**, **then**. This should increment the readability of tests.

Currently, with Mockito we implemented these 3 sections

- Given: stubbing with `when...then`
- When: calling methods on our SUT
- Then: standard assertions and verifications on mocks with `verify`

This style is not strictly compliant with BDD, even because terms like “when” and “then” are used with a different meaning with respect to the BDD terms.

To use the BDD style with Mockito, we need to use the static methods of `BDDMockito` instead of Mockito:

```
1 import static org.mockito.BDDMockito.*;
```



`BDDMockito` extends `Mockito`, thus, importing the static import above, with wildcards, still allows you to use all the static methods `Mockito`, and thus, to mix the two styles.

With the BDD style, instead of

```
1 when(employeeRepository.findAll()).thenReturn(employees);
```

we write

```
1 given(employeeRepository.findAll()).willReturn(employees);
```

For verifications, instead of

```
1 verify(bankService).pay("1", 1000);
```

we write

```
1 then(bankService).should().pay("1", 1000);
```

Note that in the BDD style, the API reflects the BDD terms.

In the source code of the example, the class `EmployeeManagerBDDTest` is the BDD-style version of the tests we have already written. These are a few examples of the previous tests written with the BDD style:

```
1  @Test
2  public void testPayEmployeesWhenSeveralEmployeeArePresent() {
3      given(employeeRepository.findAll())
4          .willReturn(asList(
5              new Employee("1", 1000),
6              new Employee("2", 2000)));
7      assertThat(employeeManager.payEmployees()).isEqualTo(2);
8      then(bankService).should().pay("2", 2000);
9      then(bankService).should().pay("1", 1000);
10     then(bankService).shouldHaveNoMoreInteractions();
11 }
12
13 @Test
14 public void testExampleOfInOrderWithTwoMocks() {
15     // Just an example of invocation order verification on several mocks
16     given(employeeRepository.findAll())
17         .willReturn(asList(
18             new Employee("1", 1000),
19             new Employee("2", 2000)));
20     assertThat(employeeManager.payEmployees()).isEqualTo(2);
21     InOrder inOrder = inOrder(bankService, employeeRepository);
22     then(employeeRepository).should(inOrder).findAll();
23     then(bankService).should(inOrder).pay("1", 1000);
24     then(bankService).should(inOrder).pay("2", 2000);
25     then(bankService).shouldHaveNoMoreInteractions();
26 }
27
28 @Test
29 public void testExampleOfArgumentCaptor() {
30     // Just an example of ArgumentCaptor
31     given(employeeRepository.findAll())
32         .willReturn(asList(
33             new Employee("1", 1000),
34             new Employee("2", 2000)));
35     assertThat(employeeManager.payEmployees()).isEqualTo(2);
36     then(bankService).should(times(2))
37         .pay(idCaptor.capture(), amountCaptor.capture());
38     assertThat(idCaptor.getAllValues()).containsExactly("1", "2");
39     assertThat(amountCaptor.getAllValues()).containsExactly(1000.0, 2000.0);
40     then(bankService).shouldHaveNoMoreInteractions();
41 }
42
43 @Test
```

```
44 public void testOtherEmployeesArePaidWhenBankServiceThrowsException() {  
45     given(employeeRepository.findAll())  
46         .willReturn(asList(notToBePaid, toBePaid));  
47     willThrow(new RuntimeException())  
48         .willDoNothing()  
49         .given(bankService).pay(anyString(), anyDouble());  
50     // number of payments must be 1  
51     assertThat(employeeManager.payEmployees()).isEqualTo(1);  
52     // make sure that Employee.paid is updated accordingly  
53     then(notToBePaid).should().setPaid(false);  
54     then(toBePaid).should().setPaid(true);  
55 }
```

## 8.6 Mockito and JUnit 5

If we use JUnit 5 (see Chapter [JUnit](#), Section [JUnit 5](#) and Chapter [Maven](#), Section [JUnit 5 and Maven](#)), we can use the specific JUnit 5 extension for Mockito, `MockitoExtension`, after adding this dependency:

```
1 <dependency>  
2     <groupId>org.mockito</groupId>  
3     <artifactId>mockito-junit-jupiter</artifactId>  
4     <version>${mockito.version}</version>  
5     <scope>test</scope>  
6 </dependency>
```

The `MockitoExtension` replaces the JUnit 4 runner that we have seen above, `MockitoJUnitRunner`. This way, we can simply write:

```
1 import org.junit.jupiter.api.extension.ExtendWith;  
2 import org.mockito.junit.jupiter.MockitoExtension;  
3 ...  
4 @ExtendWith(MockitoExtension.class)  
5 class EmployeeManagerJupiterTest {  
6  
7     @Mock  
8     private EmployeeRepository employeeRepository;  
9  
10    @Mock  
11    private BankService bankService;  
12}
```

```
13  @InjectMocks
14  private EmployeeManager employeeManager;
15  ...
16  // no need to initMocks or openMocks in a @BeforeEach method
```

## 8.7 Stubbing with answers

Up to now, we have always stubbed a method on a mocked (or spied) object by returning a value or by throwing an exception.

For returning a value we used `thenReturn` or, in the alternative syntax, `doReturn`. When doing that, we should keep in mind that we specify a returned value, which is evaluated once and for all at the time of stubbing. Thus, when calling the stubbed method, we will always get the same value.

If we want to return different values for different invocations, we saw that we can use the subsequent stubbing capabilities of Mockito (Section [Subsequent stubbing](#)) or argument matchers (Section [Argument matchers](#)).

But what if we need the stubbed method to return something that is based on the runtime value passed to the stubbed method? What if we must decide at runtime what the stubbed method should do?

First of all, when this happens, one should be sure that the design of the code and the tests cannot be changed to avoid such situations. If that is not the case, we can use the Mockito capability of **stubbing with callbacks**, that is, with instances of `Answer`. The `Answer` specifies an action (code block) that is executed when the stubbed method is called. When the stubbed method has to return something, the value returned by that action is returned. Instead of `thenReturn` we use `thenAnswer`. In the alternative syntax, instead of `doReturn` we use `doAnswer`.

The interface `Answer<T>` has this single abstract method:

```
1 T answer(InvocationOnMock invocation) throws Throwable
```

Thus, we can pass a lambda to `thenAnswer` and `doAnswer`. The parameter of type `InvocationOnMock` contains information about the called method, including the receiver (the mock) and the passed arguments. We access such information using the API of `InvocationOnMock`.

When stubbing a `void` method with `doAnswer` we should keep in mind that the lambda should return something anyway, though its value will not be used. In such a case, the lambda simply returns `null`. The `T` type parameter will be instantiated with type `Void` (the Java class representing `void`).

For example, in Section [Argument matchers](#), we used argument matchers to make the stubbed method throw an exception only when a give argument is passed:

```

1  @Test
2  public void testArgumentMatcherExample() {
3      // equivalent to the previous test, with argument matcher
4      Employee notToBePaid = spy(new Employee("1", 1000));
5      Employee toBePaid = spy(new Employee("2", 2000));
6      when(employeeRepository.findAll())
7          .thenReturn(asList(notToBePaid, toBePaid));
8      doThrow(new RuntimeException())
9          .when(bankService).pay(
10              argThat(s -> s.equals("1")),
11              anyDouble());
12     // number of payments must be 1
13     assertThat(employeeManager.payEmployees()).isEqualTo(1);
14     // make sure that Employee.paid is updated accordingly
15     verify(notToBePaid).setPaid(false);
16     verify(toBePaid).setPaid(true);
17 }

```

We could write an equivalent test by using `Answer` as follows:

```

1  @Test
2  public void testDoAnswerExample() {
3      // equivalent to the previous test, with Answer
4      Employee notToBePaid = spy(new Employee("1", 1000));
5      Employee toBePaid = spy(new Employee("2", 2000));
6      when(employeeRepository.findAll())
7          .thenReturn(asList(notToBePaid, toBePaid));
8      doAnswer(invocation -> {
9          if (invocation.getArgument(0, String.class).equals("1"))
10              throw new RuntimeException();
11          return null;
12      }).when(bankService).pay(anyString(), anyDouble());
13     // number of payments must be 1
14     assertThat(employeeManager.payEmployees()).isEqualTo(1);
15     // make sure that Employee.paid is updated accordingly
16     verify(notToBePaid).setPaid(false);
17     verify(toBePaid).setPaid(true);
18 }

```

Note how we use the API of `InvocationOnMock` for accessing the runtime information about the passed arguments. Unfortunately, when using `Answer` we have to give up the static type checks.

Mockito provides some additional utility functional interfaces `Answer1`, `Answer2`, `Answer3`, etc. representing answers for a single argument invocation, for a two-argument invocation, three-

argument invocation, etc. These are not subtypes of `Answer`, so we don't pass instances of such interfaces directly: we wrap them in a call to `org.mockito.AdditionalAnswers.answer`.

The abstract method of such functional interfaces does not take an `InvocationOnMock` parameter: it takes the corresponding number of parameters based on generic types. For example, for `Answer2` we have:

```
1 public interface Answer2<T, A0, A1> {
2     T answer(A0 argument0, A1 argument1) throws Throwable;
3 }
```

Thus, we enjoy static type safety. The previous test could be rewritten as follows (of course, a lambda with two parameters will correspond to an `Answer2`):

```
1 import static org.mockito.AdditionalAnswers.answer;
2 ...
3 @Test
4 public void testDoAnswer2Example() {
5     // equivalent to the previous test, with Answer2
6     Employee notToBePaid = spy(new Employee("1", 1000));
7     Employee toBePaid = spy(new Employee("2", 2000));
8     when(employeeRepository.findAll())
9         .thenReturn(asList(notToBePaid, toBePaid));
10    doAnswer(answer((String id, Double amount) -> {
11        if (id.equals("1"))
12            throw new RuntimeException();
13        return null;
14    })).when(bankService).pay(anyString(), anyDouble());
15    // number of payments must be 1
16    assertThat(employeeManager.payEmployees()).isEqualTo(1);
17    // make sure that Employee.paid is updated accordingly
18    verify(notToBePaid).setPaid(false);
19    verify(toBePaid).setPaid(true);
20 }
```

However, using this strategy, we cannot access all the additional information provided at runtime by `InvocationOnMock` (e.g., the called method, the receiver of the invocation, etc).

In general, the implementations of answers should not contain too much logic, or, put another way, they should contain only the minimal logic that is needed for that specific test. Remember that you are not implementing the mock objects, you just stub their methods for testing the SUT that is using the mock objects.

For example, let's say that we enhance our repository with another typical method:

```
1 Optional<Employee> findById(String id);
```

If we need to stub this method in our tests, it makes no sense to use answers for simulating a possible simplified implementation of `findById`:

```
1 @Test
2 public void exampleOfBadStubbingWithAnswer() {
3     when(employeeRepository.findAll())
4         .thenReturn(asList(
5             new Employee("1", 1000),
6             new Employee("2", 2000)));
7     when(employeeRepository.findById(anyString()))
8         .thenAnswer(answer(id ->
9             employeeRepository.findAll()
10            .stream()
11            .filter(e -> e.getId().equals(id))
12            .findFirst()
13        )));
14     // call the SUT that uses EmployeeRepository.findById("1")
15     // do verifications
16 }
```

Instead, we stub `findById` for this specific test as simply as follows:

```
1 @Test
2 public void exampleOfGoodStubbing() {
3     when(employeeRepository.findById("1"))
4         .thenReturn(Optional.of(new Employee("1", 1000)));
5     // call the SUT that uses EmployeeRepository.findById("1")
6     // do verifications
7 }
```

## 8.7.1 Another example: Transactions

Stubbing with answers is particularly useful when the stubbed method of a mock has to do something with arguments like calling a method on one of the arguments. For example, the stubbed method receives a lambda and is supposed to call the passed lambda in the middle of more complex logic. When the mock object calls the lambda it is supposed to pass some arguments to the lambda. Such arguments will probably be mock objects themselves. Our SUT passes the lambda to the mock object's stubbed method. When testing our SUT we are not interested in the logic of the stubbed method (as we said in this chapter, we must assume that our collaborators behave correctly).

However, to test our SUT we must verify that the passed lambda interacts correctly with the received arguments (as we said, these might be mocks themselves). To do such verifications, the mock object's stubbed method must execute the passed lambda. Thus, we stub the mock object's method accordingly, and to do that we need to stub it with an `Answer`.

To show such a scenario, we imagine adding **transactions** to our application. Transactions are not the responsibility of repositories. Transactions should be implemented in a higher layer, which is responsible for

1. create a transaction
2. create a repository instance inside such a transaction
3. call a code block (e.g., a lambda) passing such a repository instance
4. if something goes wrong during the execution of the code block then roll-back the transaction
5. finally close the transaction (and the repository instance)

How the transaction is created and how the repository is actually wrapped inside a transaction highly depend on the particular database implementation. Thus, we won't see an implementation of such transaction management in this chapter. However, as usual, we abstract from such an implementation and we define an interface for handing transactions in our application as follows:

```
1 package com.example;
2
3 public interface TransactionManager {
4     <T> T doInTransaction(TransactionCode<T> code);
5 }
```

Where `TransactionCode` is a functional interface we define as follows:

```
1 package com.example;
2
3 import java.util.function.Function;
4
5 @FunctionalInterface
6 public interface TransactionCode<T> extends Function<EmployeeRepository, T> {
7
8 }
```

This functional interface represents a lambda that takes an `EmployeeRepository` and returns a generic `T`.

To make the example slightly more interesting we extend our repository interface with a method for updating the database:

```

1 public interface EmployeeRepository {
2     List<Employee> findAll();
3     Employee save(Employee e);
4 }
```

We can implement a “transactional” version of our employee manager using a TransactionManager. For simplicity, we are not considering runtime exceptions in this version and we implement payEmployees as a void method, so that we will not have to test the state but only the interactions (see Section *Testing state and testing interactions*):<sup>4</sup>

```

1 package com.example;
2
3 import java.util.List;
4
5 public class TransactionalEmployeeManager {
6
7     private TransactionManager transactionManager;
8     private BankService bankService;
9
10    public TransactionalEmployeeManager(TransactionManager transactionManager,
11                                         BankService bankService) {
12        this.transactionManager = transactionManager;
13        this.bankService = bankService;
14    }
15
16    public void payEmployees() {
17        List<Employee> employees =
18            transactionManager.doInTransaction(EmployeeRepository::findAll);
19        for (Employee employee : employees) {
20            bankService.pay(employee.getId(), employee.getSalary());
21            employee.setPaid(true);
22            transactionManager.doInTransaction(
23                employeeRepository -> employeeRepository.save(employee)
24            );
25        }
26    }
27
28 }
```

Now the employee manager does not use directly an EmployeeRepository: it uses the TransactionManager and performs the access to the repository passing lambdas to doInTransaction. The passed lambdas then use the EmployeeRepository that is received as a parameter.

---

<sup>4</sup>We show the code first and then write the tests, but, as usual, the code has been implemented with TDD, by writing the tests first.

In the tests, we still use mocks, but this time, we have to stub the method `TransactionManager.doInTransaction` accordingly: we must execute the passed lambda passing a mock repository. This will allow us to verify the behavior (by testing interactions) of the lambdas of our employee manager. If we did not stub `doInTransaction`, the lambdas passed to such a method would not be executed and we would not be able to verify the interactions of such lambdas with the repository. This means that we would not be able to test a big part of the logic of our employee manager.

This could be the setup for the tests:

```
1 public class TransactionalEmployeeManagerTest {
2
3     private TransactionalEmployeeManager employeeManager;
4
5     // mocks
6     private TransactionManager transactionManager;
7     private EmployeeRepository employeeRepository;
8     private BankService bankService;
9
10    @Before
11    public void setup() {
12        employeeRepository = mock(EmployeeRepository.class);
13        transactionManager = mock(TransactionManager.class);
14        // make sure the lambda passed to the TransactionManager
15        // is executed, using the mock repository
16        when(transactionManager.doInTransaction(any()))
17            .thenAnswer(
18                answer((TransactionCode<?> code) -> code.apply(employeeRepository)));
19        bankService = mock(BankService.class);
20        employeeManager =
21            new TransactionalEmployeeManager(transactionManager, bankService);
22    }
23    ...
```

You see that the stubbing of `TransactionManager.doInTransaction` uses a Mockito answer, since we need to execute code based on the runtime value of the argument passed to the stubbed method.

We stub `TransactionManager.doInTransaction` for each test in the `@Before` method. In Section [Mockito: a tutorial](#) we said that it is not a good idea to stub in the `setup` method. In Chapter [JUnit](#), Section [Beware of code duplication removal in tests](#), we also said that code duplication is admissible in tests if it makes tests more readable. However, in this case, it makes sense to perform such a stubbing in the `setup` method, since it's the same for all the tests and it's a detail that, if written in all test methods, would not increase their readability.

Now we can verify the behavior of our transactional employee manager as usual using Mockito mechanisms:

```
1 @Test
2 public void testPayEmployeesWhenSeveralEmployeesArePresent() {
3     Employee employee1 = new Employee("1", 1000);
4     Employee employee2 = new Employee("2", 2000);
5     when(employeeRepository.findAll())
6         .thenReturn(asList(employee1, employee2));
7     employeeManager.payEmployees();
8     verify(bankService).pay("2", 2000);
9     verify(bankService).pay("1", 1000);
10    verify(employeeRepository).save(employee1);
11    verify(employeeRepository).save(employee2);
12    // also verify that all repository interactions pass through transactions
13    verify(transactionManager, times(3)).doInTransaction(any());
14 }
```

In the current shape, the `TransactionalEmployeeManager` executes each single database operation in a separate transaction: the full list of employees is retrieved in a transaction and every single employee is updated in a transaction. This might not be ideal, since, after retrieving the list of employees, the database might change. Thus when paying and updating employees we might refer to stale data. An alternative solution could be executing the full body of the method `payEmployees` in a single atomic transaction so that we can be sure that all the operations will be performed consistently.

Thus, we modify the last part of the previous test (the stubbing with an answer does not require any change):

```
1 @Test
2 public void testPayEmployeesWhenSeveralEmployeesArePresent() {
3     ...
4     // also verify that a single transaction is executed
5     verify(transactionManager, times(1)).doInTransaction(any());
6 }
```

We make sure it fails. Then we update our code accordingly:

```
1 public void payEmployees() {
2     transactionManager.doInTransaction(
3         employeeRepository -> {
4             List<Employee> employees = employeeRepository.findAll();
5             for (Employee employee : employees) {
6                 bankService.pay(employee.getId(), employee.getSalary());
7                 employee.setPaid(true);
8                 employeeRepository.save(employee);
9             }
10            return null;
11        }
12    );
13 }
```

Now the test passes. Note that since the type of the lambda passed to `doInTransaction` is not `void`, we have to return `null`.

In any case, please keep in mind that it is up to you to decide the right granularity of transactions. Executing long-running operations like the ones of `payEmployees` in a single transaction might have other drawbacks, like keeping the database locked to other transactions. Moreover, you should take into consideration also the granularity of a possible *rollback* of a transaction. In this chapter, we used transactions only as an example for showing Mockito answers.

## 8.8 Fakes

As we said at the beginning of this chapter, there are several kinds of test doubles. Here we focused on mocks, which allow us to stub methods with fixed return values or with answers. Mocks also keep track of method invocations and allow us to easily test interactions.

We now briefly describe another kind of test double: **fakes**. Fakes are effective implementations, but they are usually simplified. For sure, they are much simpler than the real implementations. They could be useful for testing in isolation because they replace a dependency/collaborator with a much lighter-weight implementation. Moreover, such fake implementations are not meant to be used in production and they are useful only for testing purposes.

For example, in the context of the application of this chapter, we could implement the `EmployeeRepository` interface with a concrete class representing an in-memory database. So we write a class and implement the methods of the implemented interface. We don't simply stub its methods after creating a mock. This implementation is a fake since it is a concrete implementation but it is much simpler than a real implementation.

This could a possible implementation of `EmployeeRepository` representing a fake:

```
1 public class EmployeeInMemoryRepository implements EmployeeRepository {  
2  
3     private List<Employee> employees;  
4  
5     public EmployeeInMemoryRepository(List<Employee> employees) {  
6         this.employees = employees;  
7     }  
8  
9     @Override  
10    public List<Employee> findAll() {  
11        return employees;  
12    }  
13  
14    @Override  
15    public Employee save(Employee employee) {  
16        ListIterator<Employee> listIterator = employees.listIterator();  
17        while (listIterator.hasNext()) {  
18            if (listIterator.next().getId().equals(employee.getId())) {  
19                listIterator.set(employee);  
20                return employee;  
21            }  
22        }  
23        employees.add(employee);  
24        return employee;  
25    }  
26}
```

This simulates the behavior of a database, but it does not persist any data on disk. For example, it simulates the update of an `Employee` record by using the `employee id` as a primary key (as it would happen in a real database).

You note that implementing a fake is not as easy as creating a mock and stub its methods. Of course, it's much easier than implementing a repository that uses a real database.

The other crucial difference with respect to mocks is that a fake implementation must have its tests!<sup>5</sup> Indeed, it's a fake implementation but it must behave correctly. It must simulate correctly the behavior of the real implementation, though in a much simpler way.

Writing unit tests for our SUT using a fake is much easier and comfortable than using a real implementation but is also much more complex than using Mockito mechanisms for mocking and stubbing. When stubbing methods we only need to care of returning a value that is useful for a specific testing scenario. When implementing a fake, we must make it behave correctly in all scenarios where the real implementation could be used. When stubbing a method, it's quite easy to

---

<sup>5</sup>See the sources of the example of this chapter for the tests of `EmployeeInMemoryRepository`.

satisfy the assumptions that the real implementation works correctly. When implementing a fake, we must make sure that it works correctly, indeed we must test the fake implementation.

Finally, it is not straightforward to verify interactions with a fake, unless we manually implement verification mechanisms in the fake itself.

Indeed, using fakes, instead of mocks, in tests make the tests closer to integration tests, rather than to real unit tests. In Chapter [Integration tests](#), Section [Use an in-memory database](#), we use an in-memory implementation of a MongoDB database, which can be seen as a “fake”. In that section, we also see that using a fake might not be a good idea, since it might make our tests less reliable and might give us false confidence in the quality of our tests.

## 8.9 What to mock

We conclude this chapter with a few suggestions on when to use mocks and when mocks should be avoided.

In general, you should mock only your dependencies, as we did in this chapter.

3rd party libraries, that is, external dependencies, should not be mocked. Mocking a 3rd party interface might still be acceptable. Mocking a 3rd party class might be really hard: you have to stub methods accordingly. To do that, you need to know the internal details of such methods. This might require some time and effort, losing the benefits of Mocking, which should allow you to quickly write tests. Moreover, stubbing external class methods that require knowledge of internal details, might lead to fragile tests, which will fail if such internal details change in the future. We will see an example of how hard is to mock a 3rd party Java database API, and why it should be avoided in Chapter [Integration tests](#), Section [Unit tests with databases](#).

In general, the use of 3rd party code should be confined in a wrapper that you own (possibly consisting of an interface and an implementation class).<sup>6</sup> All of your code should rely on such a wrapper interface, instead of calling 3rd party code directly. This way, you can mock your wrapper interface when testing your other classes. This also makes your code independent from the implementation of the wrapper class, and switching to another 3rd party library for a specific task is just a matter of using another wrapper implementation respecting the wrapper interface.

But then you will have to test the wrapper implementation at some point. In such cases, it might be worthwhile to test the wrapper implementation directly with the 3rd party code, without any mocking.

In the example of this chapter, we applied this technique by defining the `EmployeeRepository` interface, which abstracts from the actual database implementation. Later, we'll implement such an interface by using a real database. All of our code will still use this interface, thus abstracting from the implementation of the database. This allowed us to mock this interface in our unit tests. Moreover, using such an abstraction makes the tests and the code robust to changes of the 3rd party implementation classes.

---

<sup>6</sup>The wrapper can be seen as an instance of the `Adapter` design pattern or the `Facade` design pattern ([GHJV95](#)).

Even mocking value objects can be seen as an anti-pattern. For example, we never mocked `Employee` objects. The only reason to mock values of the domain model is that creating such objects is hard, e.g., their constructors require many arguments. This is a symptom that your domain classes need to be refactored. For example, instead of a constructor with many arguments, you could use the **Builder** design pattern ([GHJV95](#)) to make the creation of value objects easier.

After all, mocking aims at writing your tests quickly and at making the setup of the execution environment of your SUT easy. If the SUT requires some files, it shouldn't be hard to create them in a temporary directory and remove them when the test ends. Mocking files could be more difficult. Moreover, mocking relies on reflection, and the use of the real file system during tests could be more efficient than the use of mocked files with heavy use of reflection.

# 9. Git

Even when developing simple applications, you might want to keep track of all the changes you made to your code. This can be useful to go back to a previous version of the codebase. This mechanism is called **Version Control**.

You can implement a manual mechanism for version control. Each time you make important changes you create a copy of the whole directory of the project with a new name. Drawbacks of this approach should be immediately evident: you waste a lot of space in the disk and it is not easy to go back in time. In particular, it is not easy to understand which files have changed in which version.

A **Version Control System (VCS)** allows you to easily keep track of the changes of a project. It keeps track of the “history” of a collection of files. Each element of the history is a “snapshot” of the files and you can switch to any of these snapshots of the history. With a VCS you can easily keep track of all the experiments done with your software and if anything goes wrong you can easily go back to a working version.

Moreover, a VCS is a fundamental mechanism for sharing your code and for working in a team. We will also see that the VCS is also fundamental for implementing Continuous Integration (Chapter *Continuous Integration*).

There are several VCS available, e.g., CVS, SVN, Mercurial, etc. In this book, we will use **Git**.

First, let's deal with some terminology of Version Control Systems:

## Repository

The history is stored in a repository.

## Commit

The history in the repository is made of commits. Each commit is a saved change and can concern several files. A commit is a snapshot of the whole project. When to commit? Each time we made changes that we consider worth saving.

**Tag** A commit can be tagged. For example, when we release a version, 1.3.2, we can tag that commit with something like rel\_1.3.2. This way, it's easy to go back to a specific tagged version in history.

## Branch

A branch is a separate development stream in the repository. We can work in separate branches for different features of our application. Branches can be **merged**, by handling possible conflicts during the merge.

Version Control Systems can be divided into two main categories:

## Centralized VCS

There is a single repository on a central server. Each commit made on the local computer is immediately sent to the remote repository. If you need a certain version (commit) you need to contact the server. The developer has only the current local version of the software. If the server crashes or there's no connection then you cannot access the history, you can't commit nor update to a specific version in the history. If the server's disk breaks the whole history is lost. If there's no backup, it is lost forever.

## Distributed VCS

There can be many clones of a repository on different machines. Each developer has a complete clone of the whole repository, i.e., the whole history is on the local computer of the developer. Each commit is done on the local copy. You can always access any version from the local repository. The update of the local copy from a remote repository is done with the operation called **fetch**. The update of the remote repository with the commits made locally is done with the operation called **push**. This means that you can work offline. Since each developer has the full history, it is hard to completely lose the history of a project, since the history is implicitly redundant. You can "connect" your local repository to any other clone of the same repository. Indeed, the history is the same in each clone (at the binary level). With a distributed VCS you commit often and you're not scared to experiment: it is easy to go back in time. Moreover, there's no single point of failure.

Git is probably the most popular Distributed Version Control System. It was created by Linus Torvalds in 2005 to manage the development of the Linux kernel. You can download the command line application from <https://git-scm.com/>. We will also use the Eclipse plugin for Git, EGit <https://www.eclipse.org/egit/>, which is usually already installed in all the Eclipse distributions.

## 9.1 Let's start experimenting with Git

In this section, we'll start learning Git by using the command line application, which can be downloaded for several operating systems from <https://git-scm.com/>. In that distribution, besides the command line application, also two GUI applications are included: `git gui` for performing the main tasks on the repository and `gitk` for browsing the history of the repository.

All git commands consist of the main program `git` followed by the actual command. Then, arguments for that specific command are specified. These commands must be executed from any directory of a git working tree.

### 9.1.1 Create a git repository

Create a directory and then create the repository in that directory with the command `git init`:

```
1 mkdir myrepo  
2 cd myrepo  
3 git init
```

The command `git init` will create a new git repository in the current directory.

The subdirectory `.git` will store the repository and the current directory (`myrepo` in this example) will contain the current version of the files, called **working directory**.

The working directory, also called the **working tree**, represents a specific version belonging to the history of the repository. You can checkout any version and the working directory will be updated. The working directory also represents the version you're currently working on including changes and modifications that you still haven't committed.

## 9.1.2 Git configuration

Before starting using Git, you should configure globally a few properties of your user, with the command `git config`

```
1 git config --global user.name "your name"  
2 git config --global user.email "your email"
```

From now on, every commit you do will have this information. That is, every commit you make from now on will be identified by the author with the specified name and email.

The above two commands will configure this information globally for your current user. Global configurations will be stored in your home directory (file `.gitconfig`).

Configurations can also be specified for a single git repository stored in the file `.git/config`.

### Dealing with line endings

When using Git to collaborate with other developers it is crucial to make sure Git is properly configured to handle line endings. In fact, different operating systems handle line endings differently. The installation procedure of Git should already set the correct configuration globally, using the property `core.autocrlf`, depending on the operating system. In Linux and Mac, it should be `core.autocrlf=input`. In Windows, it should be `core.autocrlf=true`.

## 9.1.3 States of the working tree

Each file of the working tree can be:

**Untracked**

not present in the repository

**Tracked**

present in the repository and not modified

**Staged**

modified and ready to be part of the next commit

**Dirty**

modified but not staged

Changes yet not committed can be reverted. The command `git status` provides you with information about the state of the files of the working tree and the repository.

The state **staged** will be clear in the next subsection, which shows how a commit is done in Git.

### 9.1.4 Perform a commit

In Git, performing a commit consists of two phases (differently from a centralized VCS like CVS and SVN):

- Add modified or untracked files to the **index**, that is, a **staging area** where the files taking parts of the next commit are temporarily stored. The command is `git add` followed by the paths of the files (regular expressions are supported).
- Commit the staging area. This will create the commit in the history, that is, a snapshot in the repository. The command is `git commit`.

You're allowed to stage files with different `git add` commands until you're ready to commit with `git commit`.

Let's create two files, `File1.txt` and `File2.txt` with some contents in the directory `myrepo`. Then, we check the status of the repository with `git status`:

```
1 $ git status
2
3 On branch master
4
5 No commits yet
6
7 Untracked files:
8   (use "git add <file>..." to include in what will be committed)
9
10    File1.txt
11    File2.txt
12
13 nothing added to commit but untracked files present (use "git add" to track)
```



Note that git commands will provide you with useful hints.

The two new files are indeed “Untracked” since they are not part of the repository: they are only present in the working tree.

Let’s add these two files to the staging area. We could list their names in the `git add` command. If we simply want to add any untracked and dirty files to the staging area we can specify “`.`”. Of course, Git is smart enough not to consider the contents of the directory `.git`. Then we query the status again.

```
1 $ git add .
2
3 $ git status
4
5 On branch master
6
7 No commits yet
8
9 Changes to be committed:
10  (use "git rm --cached <file>..." to unstage)
11
12      new file:   File1.txt
13      new file:   File2.txt
```



Note the suggested command to possibly unstage the files, `git rm --cached`.

Now we can perform the actual commit. Every commit should have a descriptive message, describing the contents of the commit. Commit messages are crucial for inspecting the history, thus, never specify an empty commit message. The message is specified with the command line argument `-m`. If you don’t specify this command line argument, the default system text editor is opened so that you can create the commit message there. Using the text editor for specifying the commit message is useful because, after the first line, which is the main commit message, you’re allowed to write as many lines as you want in the message.

```
1 $ git commit -m "Added some initial files"
2
3 [master (root-commit) b67b44b] Added some initial files
4   2 files changed, 2 insertions(+)
5   create mode 100644 File1.txt
6   create mode 100644 File2.txt
```

The (partial) hash value b67b44b uniquely identifies this commit in this repository.

In a distributed VCS, commits cannot be given an automatic incremented version (while this is typical of centralized VCS): If several developers make a commit in their local repository how could the version be incremented consistently? After all, there might not be a central repository, but several clones of the same repository. For this reason, in a distributed VCS, commits are identified by the **hash** of their contents.

Let's modify the first file, e.g., by adding some contents. If we execute `git status` again, we get

```
1 $ git status
2
3 On branch master
4 Changes not staged for commit:
5   (use "git add <file>..." to update what will be committed)
6   (use "git checkout -- <file>..." to discard changes in working directory)
7
8       modified:   File1.txt
9
10 no changes added to commit (use "git add" and/or "git commit -a")
```

The modified file is tracked in the repository, thus its state is now “dirty”, i.e., modified.



You can use `git diff` to view the differences of currently modified files in the working tree.

We can now commit our changes as done before:

```
1 $ git add .
2
3 $ git commit -am "A few changes to File1"
```

Let's add another file, say `File3.txt`, and commit (with `git add .` and then `git commit`, as before). Let's now physically remove it from the working tree. Querying the status of the repository will show:

```
1 $ git status
2
3 On branch master
4 Changes not staged for commit:
5   (use "git add/rm <file>..." to update what will be committed)
6   (use "git checkout -- <file>..." to discard changes in working directory)
7
8       deleted:    File3.txt
9
10 no changes added to commit (use "git add" and/or "git commit -a")
```

We then commit the removal. We could use `git rm` as suggested, or simply use again `git add .`: in this case, we stage the fact that the next commit will consist of the file removal:

```
1 $ git add .
2
3 $ git commit -am "Removed File3"
```

Of course, we could still go back in the history of the repository and retrieve the removed file.

### 9.1.5 Commit history

With the command `git log` you can see the history of all the commits. Each one will show the commit message and the author. Besides, the complete hash code of the commit is also shown. To refer to a commit with its hash code, it is enough to specify only a few characters of the hash code; if there is no ambiguity, this will suffice to refer to a commit.

### 9.1.6 Ignore files and directories

In any directory of the working tree, a file `.gitignore` can be created specifying files and directories (possibly using wildcards) that Git must ignore. Each line in this file is a pattern describing untracked files and directories that Git should ignore. Note that files that are already tracked will not be affected. If a file has already been added to the repository, it should first be removed (and the removal must be committed). Then, it can be ignored from then on.

Git will check whether a `.gitignore` file is present in the current directory or any of the parent directories, up to the root of the working tree. Patterns in the higher level files (up to the root of the working tree) will be overridden by patterns in lower level files, down to the directory containing the file. The patterns are meant to match relative to the location of the `.gitignore` file. Comment lines start with `#`.

It is crucial and best practice to include at least a `.gitignore` file in the root of the working tree, containing patterns for files generated as part of the project build. For the complete syntax of patterns, we refer to <https://git-scm.com/docs/gitignore>.

An example of a typical `.gitignore` file for a Maven Java project should at least be

```
1 # Maven output folder  
2 target/  
3  
4 # Backup files automatically generated by some editors  
5 *~
```

Similarly, for an Eclipse Java project (not Maven) the `.gitignore` file could contain

```
1 bin/
```

Or simply

```
1 *.class
```



Git is not meant for binary files. Of course, you could store binary files in a Git repository, but you should do that only if you know that they do not change often. The repository will require much space to keep all the versions of binary files. In a centralized VCS, this is not a big problem, since in your computer you only have the last version and only the server will waste disk space. In a distributed VCS instead, you clone the whole repository. When using the Maven wrapper, as shown in Chapter [Maven](#), Section [Maven wrapper](#), it makes sense to store the small JAR in the repository (it is small and it is not expected to change often).

## 9.1.7 Branches

Branches are the “killer feature” of Git: differently from other version control systems, branch management in Git is easy and efficient.

In particular, you have already seen that by default there is always a branch in the repository, called `master`.

The command to create a branch is `git branch <name of the branch>`. Once created, you need to “switch” to that branch with `git checkout <name of the branch>`. You can also create and checkout a branch with a single command: `git checkout -b <name of the branch>`. Once you switched to a branch, all commits will be part of that branch only.

The command `git branch` will show all the branches of the repository and mark with a \* the current branch.

A branch can be removed with `git branch -d <name of the branch>`. This will be allowed only if that branch has already been merged (we’ll see merging later). Using `-D` instead forces the branch removal anyway. Be careful: if the branch has not been merged, the whole history of that branch (i.e., all its commits) will be lost.

For example, let’s create a new branch `experiments` and checkout the new branch:

```

1 $ git checkout -b experiments
2
3 Switched to a new branch 'experiments'
```

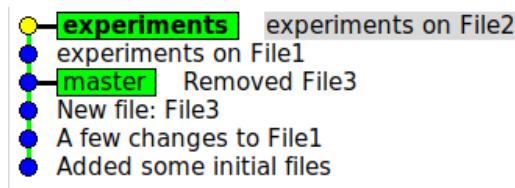
Let's run `git branch` to list all the branches (the current one has the \*):

```

1 $ git branch
2
3 * experiments
4   master
```

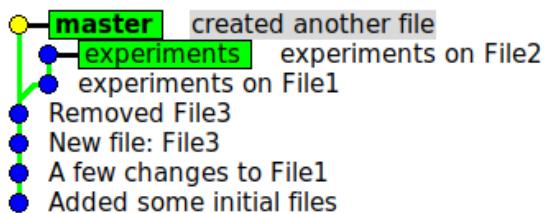
Let's modify the files and create two commits. These two commits will be part of the branch `experiments` only.

You can use the GUI program `gitk` to visualize the branches and the commits. In particular `gitk --all` will show all the branches. The current branch is in boldface.



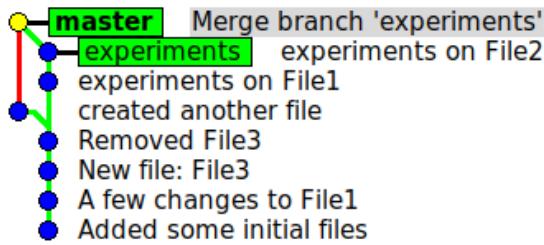
The state of the two branches after committing in the branch "experiments"

Let's get back to the "master" branch with `git checkout master`. The working tree will be restored with the contents of the master branch. Let's make a commit (after creating a new file in the working tree). The two branches now "diverge".



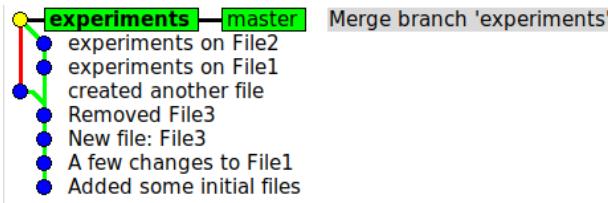
The branch "master" and "experiments" now diverge

We're on the `master` branch and we want to merge into `master` the changes of the branch `experiments`. This can be done with the command `git merge experiments`. If the commits of the two branches do not concern the same files, then the merge takes place without conflicts. The working tree will now have the changes of the commits of both branches. The merge itself will consist of a commit. The default system editor will popup proposing the default merge commit message, which can be accepted as it is by exiting the editor.



The branch “experiments” has been merged into “master”

If two branches do not “diverge” the merge will consist in a “fast forward”: the current branch is simply aligned with the other branch. Of course, the working tree will still consist of the changes of the two branches, but there will be no commit concerning the merge. The merge commit can be forced with the command line option `--no-ff`. For example, let’s checkout `experiments` branch and merge it with `master`. The `experiments` branch will be aligned with the branch `master` with a fast forward.



The branch “master” has been merged into “experiments” with fast forward

If the same files are modified by commits of two branches and the changes concern the same part of the file then merging will result in a conflict. This means that merging will be “suspended”. In the working tree, the files involved in the conflict will be annotated by Git with information about the conflicting sections, showing what’s in the current branch and what’s in the other branch.

These annotations have the following shape:

- 1 ... non-conflicting parts
- 2 <<<<< HEAD
- 3 Changes done in the current branch
- 4 =====
- 5 Changes done in the other branch
- 6 >>>>> experiments
- 7 ... non-conflicting parts

You must manually solve the conflict. You can decide to take only one branch’s version. Alternatively, you can take a few lines from one branch and other lines from the other branch. Of course, the annotations, that is, the lines starting with `<<<<<`, `=====` and `>>>>>` must also be manually removed. Then the file where you solved the conflict must be staged. After all conflicts in all the conflicting files are solved and staged, you can commit. This will finalize the suspended merging.

In such a case, the default commit message of the merge will list also the conflicting files. It could be useful to keep this information in the history of the repository.



If the conflicting files are source code, e.g., Java files, then the annotations created by Git will make the source file not compilable, as we will see later.



In the current git repository, try to modify and commit the same file in both branches, then merge and solve the conflict as described above.

## 9.1.8 Reset

You can reset the current branch to any point in history, with the command `git reset`. It is easier to do that with a GUI, like Gitk (or Eclipse, which we'll see later in this chapter): select the commit where you want to reset the current branch and choose the “Reset” context menu.

There are 3 types of reset:

**Soft** Leaves the working tree and the index untouched; but the current branch is now in the chosen commit.

**Mixed**

Leaves the working tree untouched, but the index is reset.

**Hard**

Resets both the working tree and the index; this corresponds to discarding all the changes both in the working tree and in the history. If there is no other branch with the commits from the current branch to the point where you reset to, then all those commits are lost forever. Thus a “hard reset” should be executed with extreme care.

## 9.1.9 Cherry-pick

You can cherry-pick a commit, that is, take a commit from another branch and bring it into the current branch: all the changes of the commit are applied to the current branch. As usual, this could lead to a conflict if the changes of the commit conflict with the ones done in the current branch.

The command is `git cherry-pick <id of the commit>`. As usual, the procedure is easier with a GUI.

If this succeeds, the two branches will have the same commit. In a future merge, those two commits will not generate a conflict.

This is useful, for example, when in a branch a bug has been fixed in a commit. We are not yet ready to merge the other branch in the current branch, but we could still benefit from that commit, so we cherry-pick it.

## 9.1.10 Stash

While working on a branch, we might need to temporarily switch to another branch but we don't want to lose uncommitted changes. We don't even want to commit the current changes since we haven't finished working on that. We can then **stash** our current changes (command `git stash create <message>`). The current changes will be saved in a stash area and the working tree is reset to the latest commit in the current branch. We can later retrieve the stashed changes and keep on working on that (`git stash apply`). We can have several stashes with different names.

# 9.2 Remote repositories

In this section, we start experimenting with “remote” Git repositories. In particular, in this section, we will not deal with repositories that are actually on another machine, but we'll use clones of the same repository in the same machine, in separate local directories.

## 9.2.1 Bare repository

When we initialize a Git repository in a directory, we can create a **bare repository**, by specifying `--bare` to the command `git init`. A bare repository contains only the Git repository and no working tree. In particular, there will be no `.git` subdirectory since the repository will be created in the same directory.

A bare repository is meant to be placed on a remote machine, accessible through the network, to be shared among developers. In this section, we use bare repositories to experiment with remote repositories.

Let's create a new bare repository:

```
1 $ mkdir myremoterepo  
2 $ cd myremoterepo  
3 $ git init --bare
```

Inspect the directory and note the differences with a standard Git repository we have created so far. We'll use this bare repository in the next subsections to experiment with remote repositories.

## 9.2.2 Adding a remote repository

We can “connect” our local Git repository to several “remote” repositories. Every remote repository has a *name* (which should be something meaningful for us to identify a remote repository) and a *URI* (Uniform Resource Identifier). A URI can have several forms depending on the underlying protocol for accessing such a remote repository. Later we will see some of such URIs when we access a real

remote repository on the Internet. For the moment, the URI of our remote repositories will be paths in our local file system.

The command to add a remote repository to the current repository is `git remote add <name> <uri>`.

Let's add the remote `myremoterepo` to our repository `myrepo`. From the directory of `myrepo`:

```
1 $ cd <full path>/myrepo
2 $ git remote add origin <full path>/myremoterepo
```

### 9.2.3 Initialize an empty remote repository

The remote repository is currently empty (that is, it has no history). To initialize it, we **push** from our local repository to the remote repository. With the push operation, all commits of our local repository will be sent to the remote repository. After this operation, the two repositories, `myrepo` and `myremoterepo`, will be clones.

The syntax is `git push <name of the remote> <name of the branch>`, which pushes the commits of a single branch. Alternatively, `git push <name of the remote> --all` pushes all branches.

From the directory of `myrepo`:

```
1 $ git push origin master
2
3 Enumerating objects: 21, done.
4 Counting objects: 100% (21/21), done.
5 Delta compression using up to 8 threads
6 Compressing objects: 100% (15/15), done.
7 Writing objects: 100% (21/21), 1.78 KiB | 454.00 KiB/s, done.
8 Total 21 (delta 5), reused 0 (delta 0)
9 To <URI of myremoterepo>
10 * [new branch]      master -> master
```

We are notified that on the remote repository a new branch `master` has been created with all our commits done on our local `master`.

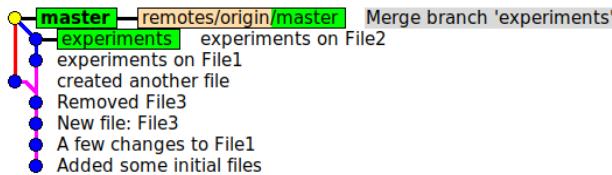
If we now run `git branch --all` we can see all the branches of our local repository. The option `--all` will list both remote-tracking branches and local branches:

```

1 $ git branch --all
2
3   experiments
4 * master
5   remotes/origin/master

```

The branch `remotes/origin/master` is a **remote-tracking branch**, that is, a pointer to the last known state of a branch of the remote repository `origin` (remember that this is the name we gave to the remote).



The branch “master” has been pushed to the remote repository

Note that we pushed only the branch `master`, thus the remote repository does not know anything of our local branch `experiments`. Of course, since we had already merged the commits of `experiments` into `master`, the merged commits are part of the history of the remote repository as well.

## 9.2.4 Cloning a remote repository

If we want to work on a project hosted on a Git repository, the first thing we have to do is cloning the remote Git repository. This will give us access to the whole history, stored in our local clone.

The syntax is `git clone <uri> {<local destination directory>}`. If the local destination is not specified, the clone will be stored in the current directory with the same name as the main directory of the remote repository. The URI of the remote repository will be automatically added as a remote repository with the default name `origin`.



You can limit the depth of the cloned history with `--depth <depth>`, but this will provide you with a clone with a history truncated to the specified number of commits.

Let’s clone the repository `myremoterepo` into another folder of our local filesystem, e.g., `myclonedrepo` (run this command from a directory that is not already a Git repository):

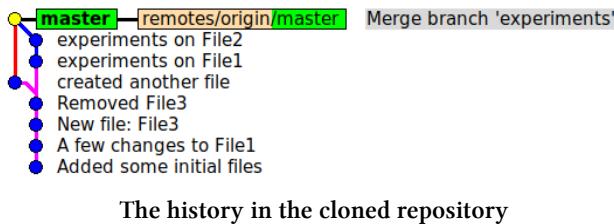
```

1 $ git clone <full path>/myremoterepo myclonedrepo
2
3 Cloning into 'myclonedrepo'...
4 done.

```

We now have the full history of the remote repository. The working tree will correspond to the latest commit on the `master` branch.

If we inspect the history of the cloned repository, there's no trace of the branch `experiments`. As we said in the previous subsection, we hadn't pushed that branch to the remote repository. Of course, since we had already merged the commits of `experiments` into `master`, the merged commits are part of the history of the remote repository as well.



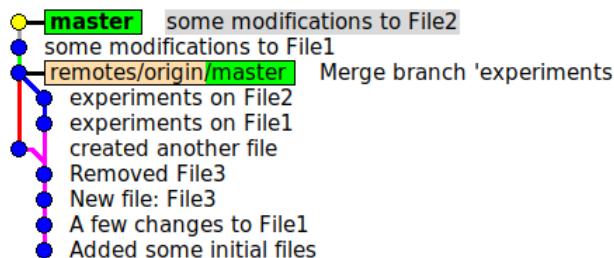
### 9.2.5 Push and fetch

We continue experimenting with simulating two developers working on the same project. Both developers work on their local cloned repositories: they modify files in their working trees, commit on their local clones and now and then they synchronize their cloned repository with the remote repository.



In this example, there is one single central remote repository. But due to the distributed nature of Git, you could have several remote repositories and all of them are clones.

Let's make some changes in `myclonedrepo` and perform a few commits. Recall that these commits are in the local clone, thus, they are not yet visible to other developers. As shown in the figure, the local `master` branch advanced, but the remote-tracking branch stays back.



The history in the cloned repository after a few commits

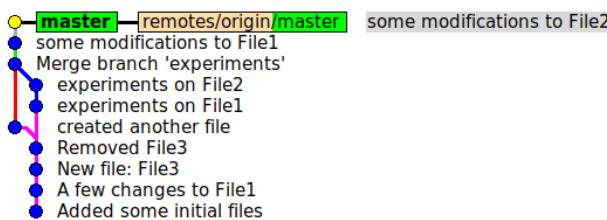
We can now push from our local repository to the remote repository (recall that by default the name is `origin`):

```

1 $ git push origin master
2
3 Enumerating objects: 9, done.
4 Counting objects: 100% (9/9), done.
5 Delta compression using up to 8 threads
6 Compressing objects: 100% (5/5), done.
7 Writing objects: 100% (6/6), 680 bytes | 680.00 KiB/s, done.
8 Total 6 (delta 0), reused 0 (delta 0)
9 To <full path>/myremoterepo
10   c39c89b..7cb5d5d  master -> master

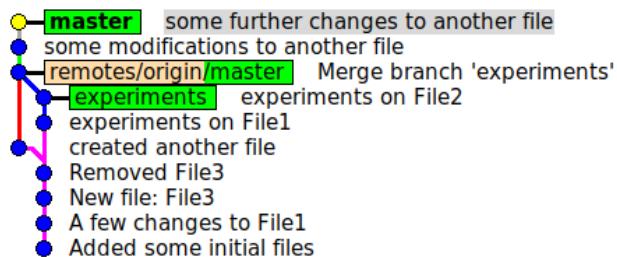
```

The remote `master` branch has been updated with our commits. Our local tracking branch has also been updated accordingly.



The history in the cloned repository after the push

Let's make some changes in `myrepo`. Remember that this other clone has not yet been updated with the new changes in the remote repository. To keep the example simple and to avoid conflicts, let's make these changes to other files.



The history in "myrepo" after a few commits

Let's try to push our changes to the remote repository:

```

1 $ git push origin master
2
3 To <full path>/myremoterepo
4 ! [rejected]      master -> master (fetch first)
5 error: failed to push some refs to '<full path>/myremoterepo'
6 hint: Updates were rejected because the remote contains work that you do
7 hint: not have locally. This is usually caused by another repository pushing
8 hint: to the same ref. You may want to first integrate the remote changes
9 ...

```

Not surprisingly, this fails. Our new commits in the `master` branch cannot be pushed to the remote repository's history, since, in the meantime, the remote history has evolved.

As suggested by the hint, we first need to

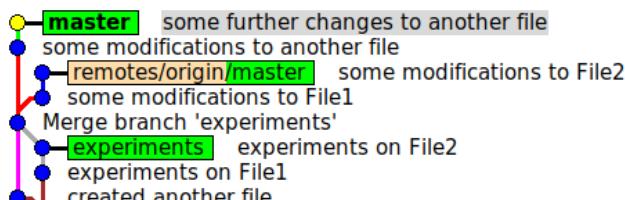
1. Fetch the remote changes, that is, update our `master` remote-tracking branch;
2. Integrate our `master` with the `master` remote-tracking branch, e.g., by merging;
3. Push our updated `master` branch (hoping that, in the meantime, no one else pushed to the remote repository again).

The syntax for fetching remote history is `git fetch <name of the remote>`:

```

1 $ git fetch origin
2
3 remote: Enumerating objects: 9, done.
4 remote: Counting objects: 100% (9/9), done.
5 remote: Compressing objects: 100% (5/5), done.
6 remote: Total 6 (delta 0), reused 0 (delta 0)
7 Unpacking objects: 100% (6/6), done.
8 From <full path>/myremoterepo
9   c39c89b..7cb5d5d  master      -> origin/master

```



The history in “myrepo” after fetching from the remote repository

Note that the fetch operation does not update any of our local branches: it only updates the remote-tracking branch.

Now we can merge the remote-tracking branch `remotes/origin/master` into our local `master` branch. Recall that this operation might be interrupted due to possible conflicts (in this very example,

we were careful not to create conflicting commits). In such a case, you should apply the manual merging procedure we have already shown.

After the merging, we can push.



Fetching and merging are two common operations and for this reason, Git provides the command `git pull <name of the remote> <name of the branch>` that does the two operations together. Again, this could generate a conflict during the merging.

## Forcing a push

You can force a push to a remote repository even when it would not be allowed because your repository is not up-to-date. To do that use the command line argument `--force`. Please be aware that this will rewrite the history of the remote repository, breaking all the existing clones, which will then have to be “realigned”.



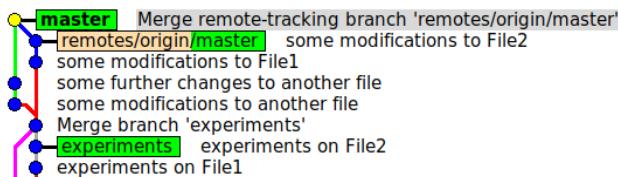
You can set as a remote also a Git repository that is not a bare repository. You can fetch from such a repository but you cannot push to a non-bare repository.

### 9.2.6 Rebasing

Given the current branch, say `R1`, and another branch, say `R2`, which are diverging, instead of merging `R2` into `R1`, we can **rebase** `R1` on top of `R2`: all commits in `R1` starting from the diverging point will be moved on top of `R2`. Of course, during the rebase conflicts may arise.

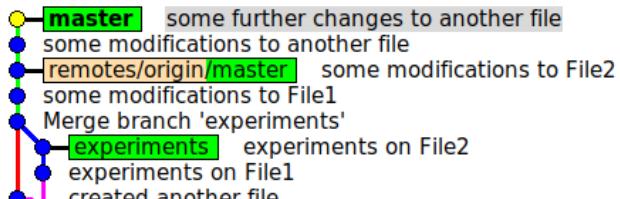
With rebasing, the history stays “linear” instead of “branched”.

For example, this is the history after merging the remote-tracking branch into our local branch:



The history if we merge with the remote-tracking branch

Instead, if we rebase our branch on top of the remote-tracking branch we get:



The history if we rebase on top of the remote-tracking branch

```

1 $ git rebase remotes/origin/master
2
3 First, rewinding head to replay your work on top of it...
4 Applying: some modifications to another file
5 Applying: some further changes to another file
  
```



The rebase operation “rewrites” the local history, starting from the diverging point. Do not rebase a branch that’s already been pushed, or you won’t be able to push the branch unless you force the push.

The rebase operation can also be done “interactively”, using the command-line argument `--interactive`. This will allow you to discard commits, change the commit messages or “squash” commits (that is, merge several commits into a single one). When done from the command line an editor is opened to perform all of the above operations. With Eclipse, the interactive rebase is done using a dedicated view with buttons for the above operations. This is however outside the scope of the book.

## 9.2.7 Pull requests

We now briefly introduce the main mechanisms to collaborate on a project with other developers. The same mechanism could also be used by a single developer as a development workflow.

Let’s say that Alice and Bob are working on the same project. They both have a local clone and a remote repository of the same project. Thus, the Git repositories are the same, but there does not necessarily exist a single central repository, since they both have their remote repositories, stored somewhere and accessible to them.

Alice pushed a new branch `feature1` on her remote repository with some new implemented feature and tells Bob to pull her branch `feature1` into his repository. This is known as **pull request**.

When performed manually, Bob would

1. add Alice’s remote repository as a remote repository of his own local repository,
2. fetch her branch `feature1`,
3. check that everything is OK and merge `feature1` on his working branch,

4. push the change to his own remote repository

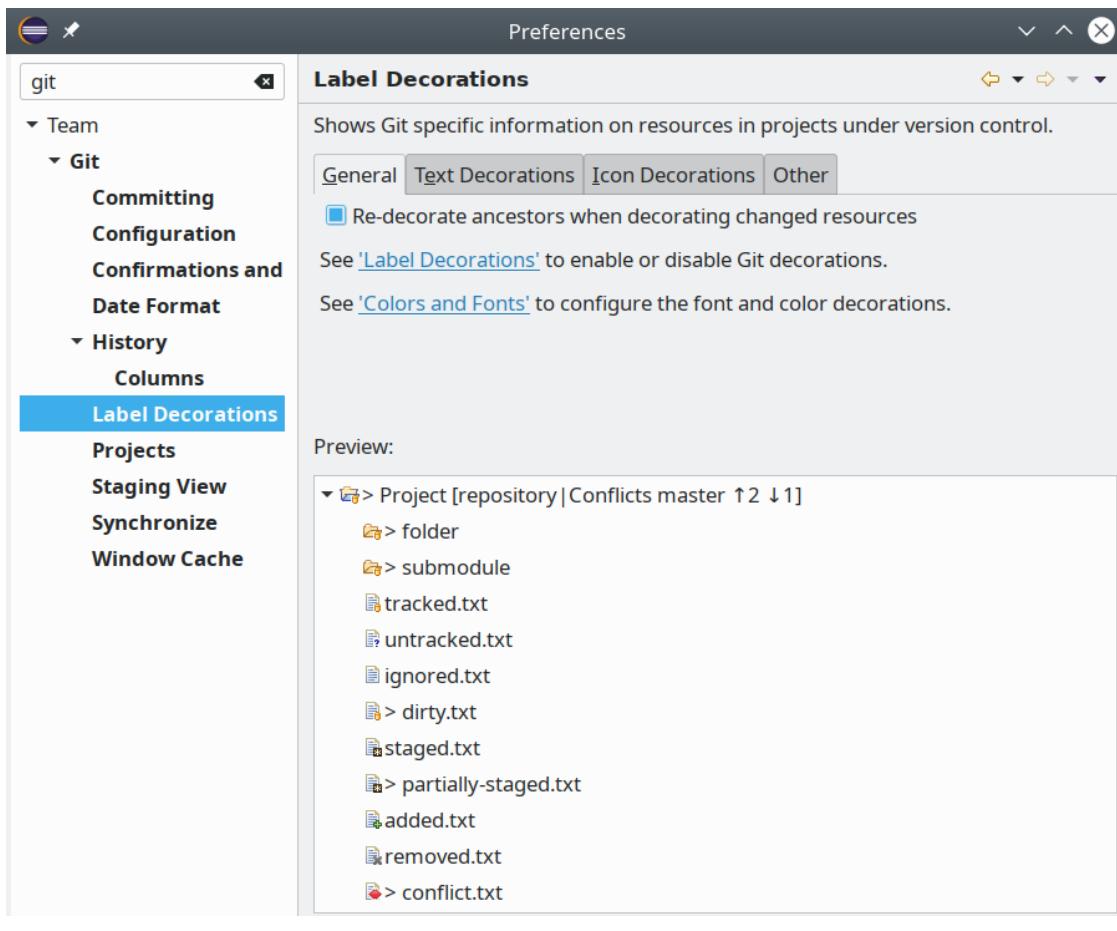
This mechanism is so popular and widely used that cloud collaborative platforms based on Git, like GitHub (which we'll see later in Section [GitHub](#)), provide some tools to easily handle a pull request from its creation to the final merge.

## 9.3 EGit (Git in Eclipse)

The integration of Git in Eclipse is provided by the EGit project, <https://www.eclipse.org/egit/>. The main plugins of EGit are already installed in the main Eclipse distributions (in particular, in the one for Java developers). Here we will only give a brief overview of EGit and we refer to the official documentation for further details, [https://wiki.eclipse.org/EGit/User\\_Guide](https://wiki.eclipse.org/EGit/User_Guide).

EGit provides several Eclipse views and the perspective "Git". We'll see some of these features in the next sections.

Moreover, when an Eclipse project is part of a Git repository, EGit uses label decorations to make it easy to spot the Git state of each single file in the project. As usual, the decorations and colors, and fonts can be customized:



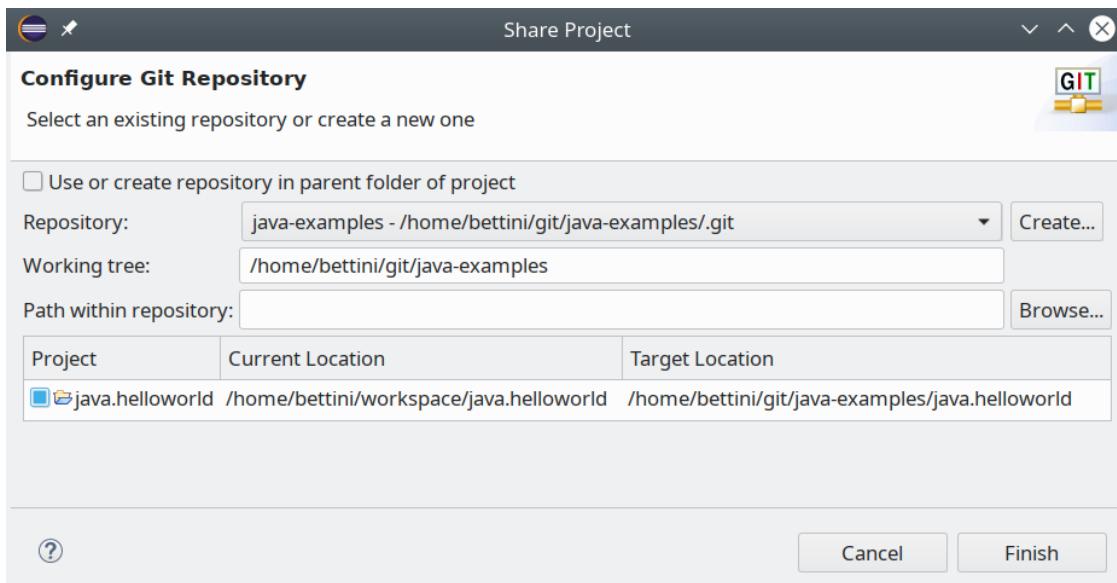
EGit label decorations in the project

### 9.3.1 Add an Eclipse project to a new Git repository

Let's create a Java project in Eclipse, e.g., `java.helloworld`. You can create it in the workspace (though we recommended not to do that in previous chapters, we'll fix that in a minute). Create a Java class in that project, e.g., `helloworld.HelloWorld`.

Use the context menu of the project **Team → Share Project...**. If you have other plugins for other version control systems, then make sure you select Git. In the "Share Project" dialog, press the button "Create...". In the dialog, specify the path for the Git repository. By default, the main directory for Git repositories is the `git` directory in your home folder. For example, choose `java-examples` as the final directory. Press "Finish". This has already initialized the empty Git repository in the specified folder.

The "Share Project" dialog now summarizes the following information:



This highlights the fact that the directory of the project, will be moved inside the directory hosting the Git repository, `java-examples`, in a subdirectory called `java.helloworld` (see the “Target Location”). Press “Finish” to make this happen.



EGit has a preference, set by default, **Team → Git → Projects** “Automatically ignore derived resources by adding them to `.gitignore`”, which automatically updates the `.gitignore` with patterns of the generated files (like `.class` files) or creates one if not already present.

The project has the label decorations of EGit, showing that all the files of the project are untracked in the Git repository. Note also that a `.gitignore` has been automatically created with the pattern for excluding the `bin` directory, where `.class` files are generated. Indeed, we do not want to store generated files in the history of the repository.

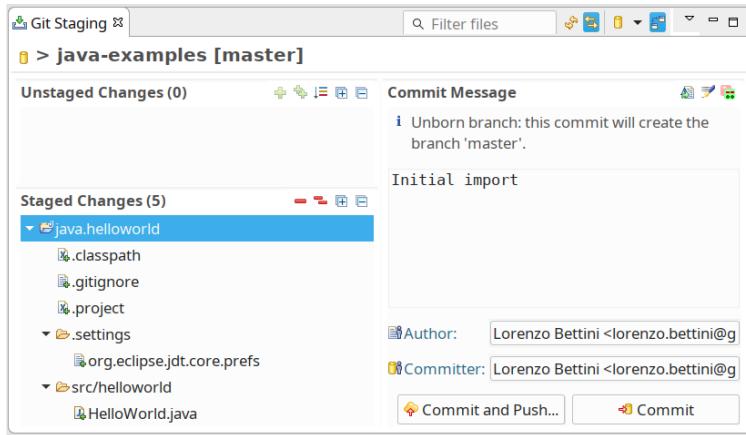
We now have to commit the files of the project. This can be done with the context menu **Team → Commit....** This will open the “Git Staging” view. Alternatively, simply manually open such a view.

The files you want to commit must be dragged from the “Unstaged Changes” area into the “Staged Changes” area. This can be done one by one if you want to commit only some of the files, or you can use the double plus toolbar to stage all untracked (and modified) files.



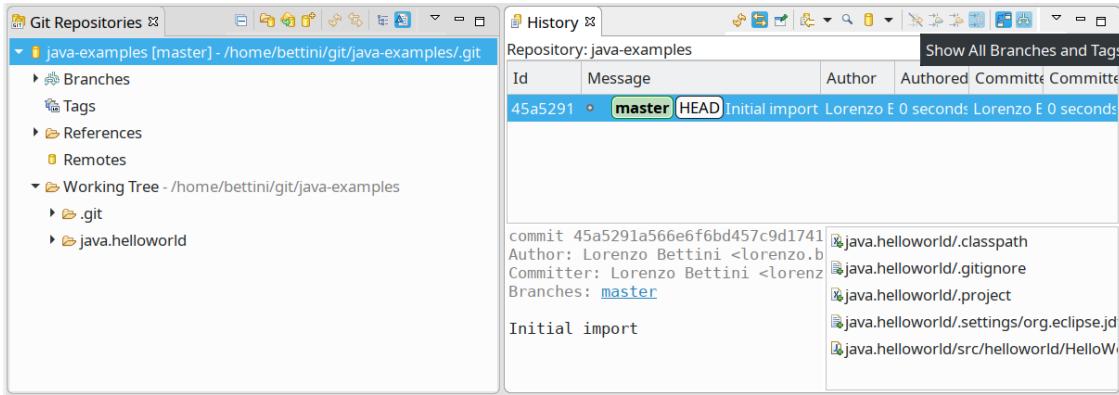
Use the view menu of the “Git Staging” view, and select **Presentation → Compact Tree** to see the untracked or modified files in the tree structure of the project.

Then write a commit message in the “Commit Message” area and press “Commit”:



The “Git Staging” view right before committing

The “Git Repositories” and “History” views (which are by default shown in the “Git” perspective) show all the Git repositories of the projects in the workspace and the history of the current repository. In the “History” views you might want to toggle the toolbar button “Show All Branches and Tags”, otherwise, only the current branch is shown in the history.

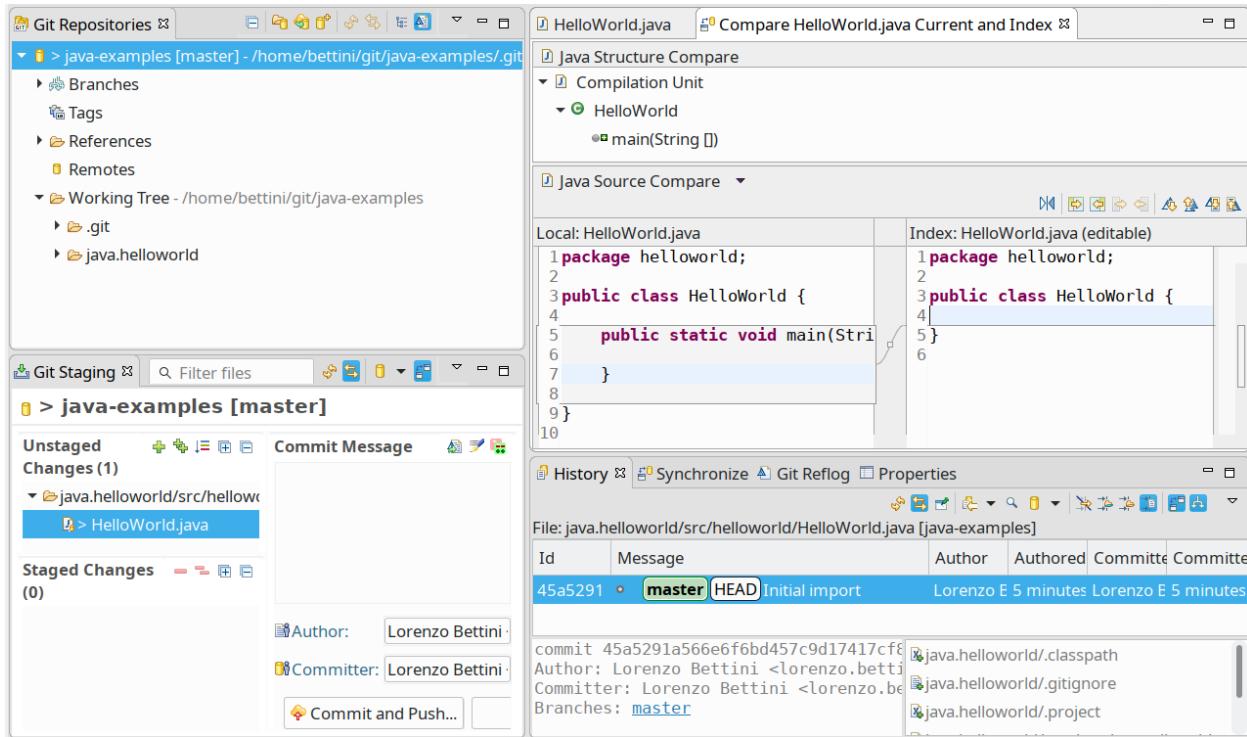


On the left of the “Show All Branches and Tags” toolbar button, you might want to also toggle the button “Compare Mode”. When this mode is enabled, double-clicking on a file of a commit shows the changes introduced in that commit, using the typical “Compare” view of Eclipse. When it is not enabled, double-clicking will simply show the contents of that file in that commit.

Try to add files and modify existing ones. The label decorators will update accordingly. Then, commit by following the same procedure.

If you remove a file from the project, the removed file will be automatically added to the “Staged Changes” area.

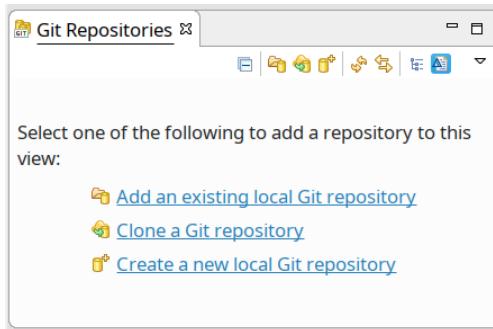
When something has changed in a project, the context menu **Compare With** allows you to compare the current file with the previous revision or with a specific commit. Double click one of the files in the “Unstaged Changes” or “Staged Changes” and the compare view of Eclipse will highlight the differences with the version of the file in the repository.



### 9.3.2 The Git Repositories view

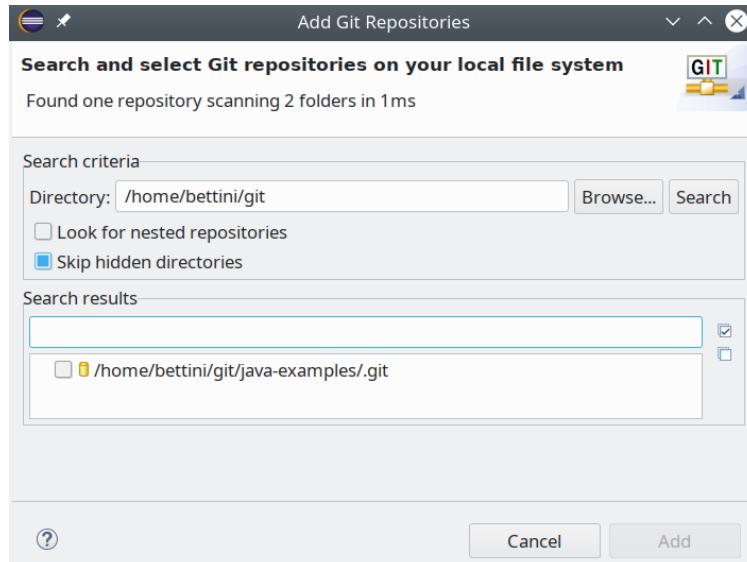
The “Git Repositories” view (present by default in the “Git” perspective) shows all the git repositories that are already known to Eclipse, e.g., because you created them from Eclipse, (as done in the previous section) or because you imported a project that is part of a Git repository (as shown in the next section).

This view provides a few tools as shown in the next screenshot (the screenshot assumes that no Git repository has been imported yet, but the toolbar buttons provide the same mechanisms like the ones shown here):



Besides “Clone a Git repository” and “Create a new local Git repository”, which should be straightforward to understand, the “Add an existing local Git repository” allows you to import in this Eclipse view an existing local Git repository.

For example, assuming we are in a fresh workspace, let's add the `java-examples` Git repository that we created in the previous section. By default, the dialog that appears shows all the Git repositories in the default Git directory (`git` in your home folder); of course, you can search for any Git repository on your hard disk. Select the repositories you want to import (in this example, the `java-examples` created in the previous section) and press "Add"



The view will be populated with the added Git repositories. You can then use the contextual menu to import the projects into Eclipse. The imported projects will be automatically known to be part of that git repository.

### 9.3.3 Importing an existing project

If you import an Eclipse project from a location that is part of a working tree of a Git repository, EGit automatically detects the project as part of a Git repository. The "Git Repositories" view will then be automatically populated with the corresponding Git repository and the imported project will be connected to that Git repository.

### 9.3.4 Branch operations

The history view of EGit provides many contextual menus to deal with branch operations, like switching, creating, removing, and merging branches and resetting the current branch to a specific commit.

For merging, you need to select the branch that you want to merge with the current branch and use the context menu "Merge". In case of conflicts, a dialog will appear with such an error. The "Git Staging" view will show the files with conflicts. Git has already added its annotations, thus Java files with conflict won't compile due to syntax errors:

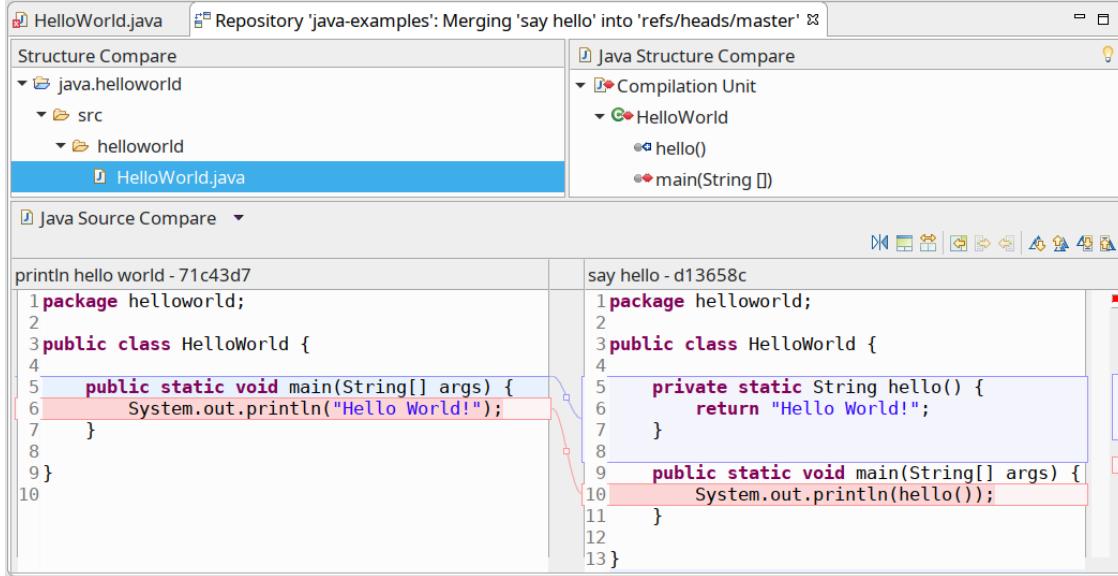
```

1 package helloworld;
2
3 public class HelloWorld {
4
5     <<<<< HEAD
6     public static void main(String[] args) {
7         System.out.println("Hello World!");
8     }
9
10    private static String hello() {
11        return "Hello World!";
12    }
13
14    public static void main(String[] args) {
15        System.out.println(hello());
16    }
17
18 }

```

Conflicts could be solved by simply replacing the file completely with our version or with the version of the other branch, using the context menu in the “Git Staging”, “Replace With”. Once replaced, the file with previous conflicts must be manually staged. This replacement operation should be used with care and only if you are sure that the complete replacement with one of the two versions makes sense.

Otherwise, the conflict must be handled manually. Editing the file with the annotations left by Git might not be ideal. An easier way is to use the context menu in the “Git Staging”, “Merge Tool”. This can be enabled also by double-clicking on a file with conflicts in such a view.



This view shows the two versions, highlighting the differing sections. In particular, the blue sections do not conflict while the red ones conflict. Hovering on the small boxes of the connecting lines, you can choose the blocks to take from the other branch. Alternatively, you can simply manually edit the left part (which corresponds to your current branch).

Once you’re happy, save the file (make sure it now compiles) and stage it. When all conflicts in

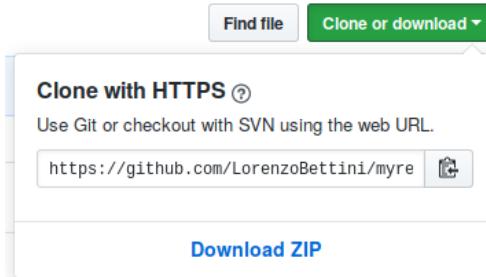
all files are solved, the “Commit Message” provides the default merging message, and by pressing “Commit” the merge is finalized.

### 9.3.5 Cloning from Eclipse

We have already seen how to clone a remote repository with the `git` command from the command line in Section [Cloning a remote repository](#). In that section, the remote repository was actually in our file system.

Let’s now clone a real remote Git repository directly from Eclipse and import the Eclipse projects in that git repository. We will clone a Git repository from GitHub, one of the mainstream Git hosting sites (we will get back to GitHub in Section [GitHub](#)).

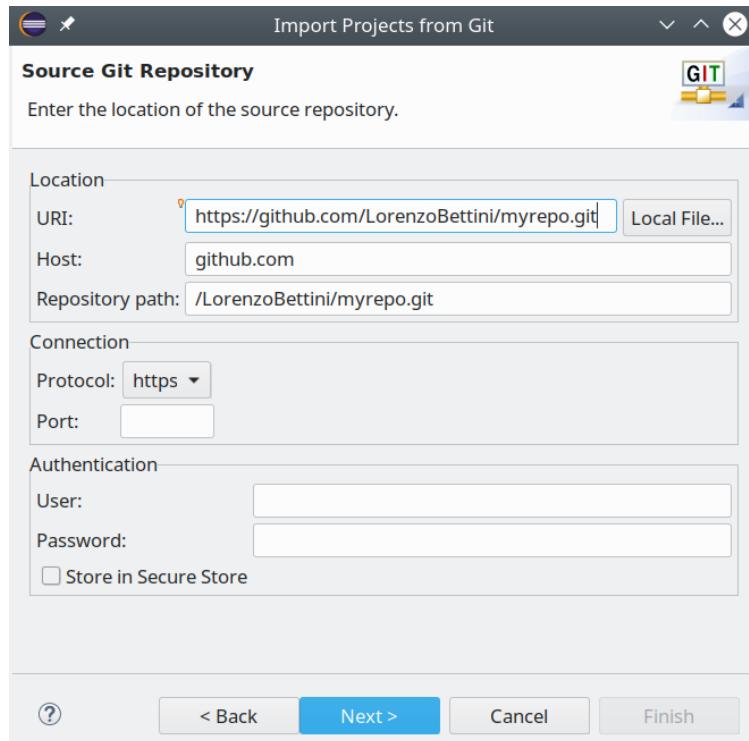
First of all, we need to know the URI of the repository we want to clone. Such a URI can be retrieved from the hosting site, once we landed on the project web page. For example, let’s say we want to clone the repository of the project <https://github.com/LorenzoBettini/myrepo>. The URI can be found by selecting “Clone or download”



The HTTPS URI is always available and should be used if you’re not the owner of that repository. For owners of the repository, the SSH URI can be used. The URI must be copied (e.g., by using the icon on the right of the URI) into your own clipboard.

The Git repository can be cloned from the EGit “Git Repositories” view and then contained project(s) imported from that view. In this example, we want to clone and import the projects altogether, thus we select the menu **File → Import... → Git → Projects from Git**. Then we select “Clone URI” and press “Next”. In the “URI” text field we paste the URI of the remote repository<sup>1</sup>

<sup>1</sup>If the URI has already been copied in the clipboard, Eclipse automatically pastes it into this dialog.



Press “Next” and you can select the branches. Press “Next” and specify the local path where the repository will be cloned (as usual, the default is the directory `git` in your home folder) and possibly other settings like the initial branch and the name for the remote. Press “Next” again and you can choose to import existing projects (if any) or other choices for the creation of new projects. On the final wizard page, a summary of the found projects is shown, and pressing “Finish” will start the cloning. When the clone is done, the selected projects are imported in Eclipse and automatically connected to the cloned Git repository.

## 9.4 GitHub

GitHub is one of the mainstream Git hosting sites. We will use it in this book.

GitHub is free for open-source projects and recently it started to provide private Git repositories for free (of course, there are payment plans for the enterprise as well).

### 9.4.1 Create a repository on GitHub

First of all, you have to create a new account on <https://github.com/> and follow the instructions to upload your SSH public key (you also find instructions to create a public-private key pair in case you have none).

Once you have your account and you are logged in, you can start creating a new Git repository from the GitHub web interface. Use the top right “plus” sign drop-down menu and select “New

repository". In this example, we will create a Git repository on GitHub for the Git repository we created and populated locally in the previous sections, `java-examples`. Choose a repository name (in this example we use the same name as the local repository directory, `java-examples`, but you're free to choose any name that makes sense for the repository). You can also provide a description, whether to make it public or private, and initialize it with a `README`, a `.gitignore`, and a license file. In the screenshot below I'm using my account; in your case, the "Owner" will be your account:

The screenshot shows the GitHub interface for creating a new repository. At the top, there are fields for 'Owner' (set to 'LorenzoBettini') and 'Repository name' (set to 'java-examples'). Below these, a note says 'Great repository names are short and memorable. Need inspiration? How'. A 'Description (optional)' field contains the text 'Some Java examples, just for experimenting with GitHub'. Under 'Visibility', the 'Public' option is selected, with the note 'Anyone can see this repository. You choose who can commit.' The 'Private' option is also shown. A checkbox for 'Initialize this repository with a README' is checked, with the note 'This will let you immediately clone the repository to your computer. Skip this step'. At the bottom, there are buttons for 'Add .gitignore: None' and 'Add a license: None', followed by a 'Create repository' button.

Once you're done, press "Create repository". The next page will give you the commands to add this remote repository to your existing git repository and to push (note that by default the URI is based on the SSH protocol). These commands are to be run from the directory of the Git repository you want to push remotely (<YourGitHubUser> will be your own GitHub account):

```
1 git remote add origin git@github.com:<YourGitHubUser>/java-examples.git  
2 git push -u origin master
```



GitHub has recently switched to naming the default branch as `main` instead of `master`. The examples in this book are still using `master` as the default branch.

When the push successfully terminates your local Git repository will be shared remotely on GitHub. Other users will be able to clone it. Other users won't be able to push to your repository unless you authorize them explicitly. This can be done in the "Settings" section, choosing "Collaborators".

Besides, contributions to GitHub repositories can be done as shown in the following sections.

## 9.4.2 GitHub pull requests

You can directly push to your remote GitHub repositories. However, you might want to get familiar with the usual GitHub (and other Git services) workflow, based on pull requests.

In this section, we describe the main development workflow fostered by GitHub (<https://guides.github.com/introduction/flow/>).

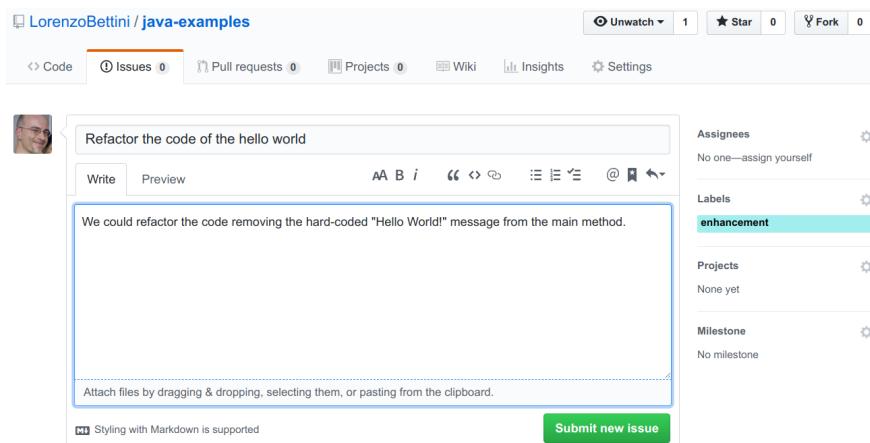
This workflow is based on two main mechanisms:

- Work on a feature of your project in a separate branch, assuming that what's in the `master` branch is stable and possibly ready to be deployed.
- When the branch implementing a feature is ready to be at least “reviewed”, create a **pull request** (PR for short); once the pull request is “accepted” the branch can be merged (either into another branch or into the `master` branch).

Thus, when you want to start working on a new feature of your project, create a branch in your local Git repository and start working on that branch. As we saw in this chapter, Git branches are lightweight and very easy to handle, thus you can freely experiment on the new branch, without being scared to break anything. In the worst case, if the work done on the branch is not worth being merged you can always remove that branch.

For example, let's say we want to refactor the Java code of our project `java.helloworld` that is part of the Git repository `java-examples` that we've just pushed to GitHub.

While not strictly required, it might be helpful to create an **issue** on the GitHub web interface (tab “Issues” on the GitHub project web page) to keep track of the work we plan to do. Remember that issues do not necessarily represent a **bug**: issues can be used to organize your work on a project, e.g., as a TODO list for the features you plan or wish to add to your project. You can use the “Labels” to give a category to an issue (e.g., in this case, we want to enhance the code; if the issue represented a bug, we would use the “bug” label):



When working on a team, it might be worthwhile to specify the “Assignees” of a specific issue, so that other members of the team know if anyone is already working on that issue.

Once created, an issue provides a way to add further comments to that issue. You can add comments yourself, or, when working on a team, other members can comment on that issue. Each issue is automatically assigned an incremental number as an identifier. This can be used to refer to the issue with a URL. For example, the first issue created can be referred to with the URL: “<https://github.com/<YourGitHubUser>/java-examples/issues/1>”.

On our local Git repository, we create a new branch, with a meaningful name, which is related to the work we plan to do on that branch. For example, in this case, we can call the branch `refactoring`. We create such a branch and we start working on that branch.



Create the branch using the context menu of the “History” view in Eclipse. By default, after the branch is created, the new branch automatically becomes the active branch.

Now we create a new Java class `HelloMessage` and we commit. We add a method `getMessage` to that class and we commit. We refactor the main method so that it creates an instance of the new class and calls `getMessage`.

Once we think that our branch is worth being merged, we push it to GitHub.

If a commit message contains the URL of an issue, GitHub adds a reference to that commit in the issue web page. For example, one of the commits of our branch has this message

```
1 created method HelloMessage.getMessage  
2  
3 https://github.com/LorenzoBettini/java-examples/issues/1
```

And the issue web page records this information (the link to the commit allows you to inspect the commit changes from GitHub):

## Refactor the code of the hello world #1

The screenshot shows a GitHub issue page for pull request #1. The title is "Refactor the code of the hello world #1". A green button labeled "Open" is visible. The issue was opened by LorenzoBettini 13 minutes ago with 0 comments. The first comment from LorenzoBettini 13 minutes ago suggests refactoring the code by removing the hard-coded "Hello World!" message from the main method. This comment has a reply from LorenzoBettini adding the "enhancement" label 13 minutes ago. Below this, another comment from LorenzoBettini 10 minutes ago adds a commit that referenced this issue. The commit message is "created method HelloMessage.getMessage". The commit hash is d6ed106.

Now we create a pull request, for merging this branch into master directly from GitHub.

After pushing a new branch, the GitHub web interface provides a shortcut to directly create a pull request for recently pushed branches:

The screenshot shows a GitHub repository page. At the top, there are summary statistics: 3 commits, 2 branches, 0 releases, and 1 contributor. Below this, a section titled "Your recently pushed branches:" lists two items: "refactoring" (less than a minute ago) and "java.helloworld". For each item, it shows the author's profile picture, the branch name, the commit message, and the time of the latest commit. A prominent yellow button labeled "Compare & pull request" is positioned above the list of branches. Below the list, there are buttons for "Create new file", "Upload files", "Find file", and "Clone or download".

It is then enough to use the button “Compare & pull request”. Alternatively, the button “New pull request” is always available to manually create one.



By default, the destination branch of a pull request is `master`. However, you are allowed to choose any destination branch, if you want to merge your branch into another branch different from `master`.

Once we selected “Compare & pull request”, we can give a title to the pull request (by default the name of the branch is used) and an additional description. If a pull request aims at implementing (or fixing) an issue, we can specify in the description the message

1 Closes #<number of the issue>

This way, when the pull request is merged, the mentioned issue is automatically closed, with a link to the pull request. In our example, we can add `Closes #1` (code completion is available in the description text area for easily referring to an existing issue).

Before creating the pull request we have the chance to review all the changes that will be implemented after merging the pull request. These are available scrolling down the web page (the differences can be seen using “Unified” or “Split”; try them both and choose your favorite one):

The screenshot displays two code diff panels side-by-side. The top panel shows the file `java.helloworld/src/helloworld/HelloMessage.java` with the following content:

```
+ package helloworld;
+
+ public class HelloMessage {
+     public String getMessage() {
+         return "Hello World!";
+     }
+ }
```

The bottom panel shows the file `java.helloworld/src/helloworld/HelloWorld.java` with the following content:

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

This is a very useful tool, which should be used even when you are the only one working on the project. Indeed, you have the chance to easily review all the changes of the commits of the pushed branch, all flattened as differences for each file involved in the commits. (Single commits can also be inspected):

If you spot any mistakes, you can fix them locally, commit, and push the branch again. It is then a matter of reloading the web page of the pull request to see the new commit(s).

Once you are ready, press “Create pull request”. The web page of a pull request is similar to the one of an issue. It is meant to stimulate a discussion with other members of a team. General comments to the pull request can be added with the text area at the end of the web page.

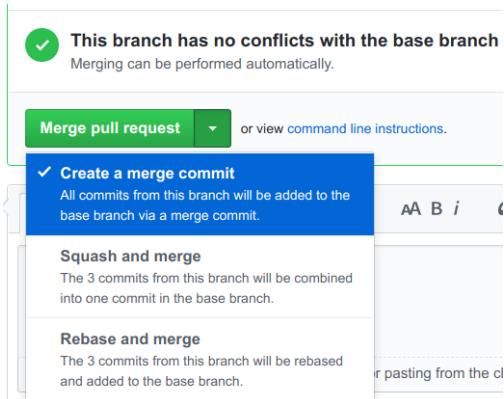
The “Files changed” tab allows anyone to see the differences applied when merging this pull request. This is the same mechanism available when creating the pull request. However, when the pull request is created, this view of difference allows others to “review” your changes by adding comments on every single line. The review can be created then by requesting changes or by simply approving the changes. Recall that changes can always be done to a pull request by committing changes to the branch and by pushing the branch with the updated changes: the pull request will be automatically updated with the new commits.

In the comments of issues and pull requests we can mention any GitHub user with the syntax `@<GitHubUser>` (code completion is available) so that the user will be notified (in the GitHub user’s notification section and by email). Similarly, reviewers can be selected directly from the pull request web page using the top right “Reviewers” section.

We will see this reviewing mechanism in the next section when contributing to a GitHub project of someone else.

GitHub automatically checks whether the branch of the pull request can be merged without conflicts. In that case, and in general, when you’re happy with a pull request, you use the “Merge pull request”

button.



Note that you have 3 possible ways of merging the pull request. The default one creates a merge commit. The other two ways are described in the selection list. The default one is the only one that does not rewrite the history of the GitHub repository. “Squash and merge” has the advantage to keep the history short and compact while “Rebase and merge” will keep it linear. Choose the one you prefer. In this example, we choose the default one.

Then, we must confirm the merging. Recall that, since we specified the `Closes #1` string in the pull request description, the corresponding issue will be automatically closed. The same holds if you specify such a string in any commit message of the branch of the pull request:

### Refactor the code of the hello world #1

Closed LorenzoBettini opened this issue an hour ago · 0 comments

LorenzoBettini commented an hour ago  
We could refactor the code removing the hard-coded "Hello World!" message from the main method.

LorenzoBettini added the `enhancement` label an hour ago

LorenzoBettini added a commit that referenced this issue an hour ago  
created method `HelloMessage.getMessage` ... d6ed106

LorenzoBettini referenced this issue 35 minutes ago  
**Refactoring #2**

LorenzoBettini closed this in #2 a minute ago

Also pull requests have an automatically assigned identifier. Links to pull requests work the same ways as the links to issues.

Once the pull request is merged, you can remove the merged branch from the GitHub repository, using the “Delete Branch” button in the pull request web page.

Now that the pull request is merged, the GitHub repository has been updated.<sup>2</sup> However, our local Git repository has to be manually updated.

The update of the local Git repository can be performed as follows:

1. Fetch from the remote repository;
2. Delete the local `master` branch (assuming you are still positioned on the branch `refactoring`);
3. Create a new `master` branch starting from the remote reference of `master` (and switch to the new `master` branch);
4. Remove both the local `refactoring` branch and the remote reference to `refactoring`.



If a branch is removed from a remote Git repository, fetching from the remote repository does not automatically remove the branch from the local Git repository. Even existing references to remote repository branches are not removed. Removing branches and references to remote branches has to be performed manually in the local repository. In Eclipse you can simply use the context menu of the “History” view.

### 9.4.3 Contributing to other projects

The workflow that we saw in the previous section can be used to contribute to GitHub projects that we do not own (or for which we are not enabled as a collaborator by the owner).

In this section, I will use two GitHub accounts I own for simulating the contribution to a GitHub repository:

- `LoreBett` is the account for contributing to the repository `java-examples`
- `LorenzoBettini` is the account owning the repository `java-examples`

#### 9.4.3.1 The contributor creates the PR

We now impersonate the contributor, i.e., the user `LoreBett`.

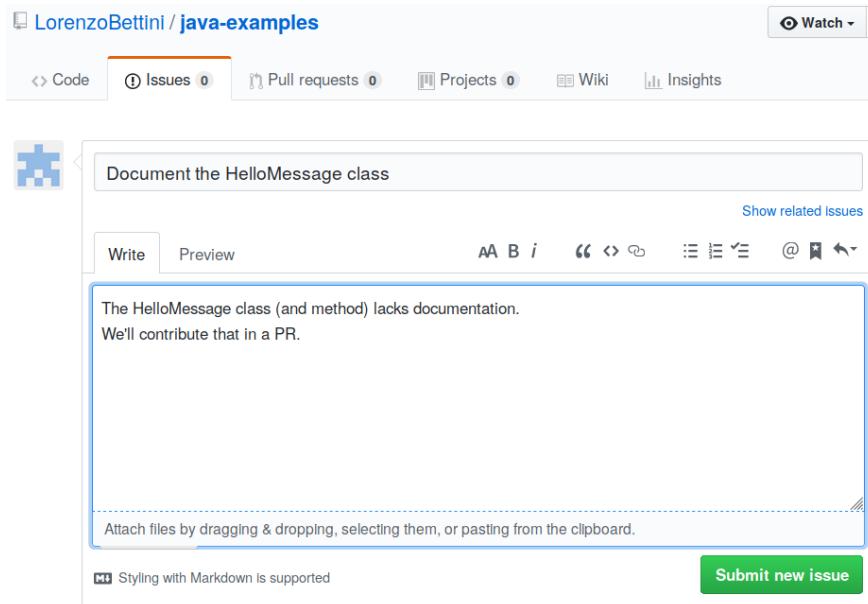
We want to contribute to the project hosted in the GitHub repository <https://github.com/LorenzoBettini/java-examples>.

For example, we note that the Java code does not contain documentation for the `HelloMessage` class and we would like to contribute it.

While not strictly required, we create an issue on the GitHub project page. For example, something like

---

<sup>2</sup>Note that the committer of the merge commit is GitHub itself.



Since we are not set as collaborators of that repository, we cannot directly push branches to that repository.

The first thing we must do is create a **fork** of that repository into our own GitHub user account. A fork is a clone of a repository. Once we created a fork into our GitHub account, we become the owner of such a fork, we can push to our fork and we can freely experiment without affecting the original project.

Let's log into our GitHub account with the browser. We visit the URL of the project we want to contribute and we press the top right button "Fork". Once the forking finishes (it should take a few seconds), we are automatically redirected to the web page of our fork. The web page explicitly reports that this is a fork of the original project.



The "Clone or download" button now provides the URI of our fork. We clone locally our Git repository following the same procedure we have already seen. Similarly, we import the Eclipse project contained in the Git repository.

We create a new branch, e.g., `document-hello-message` and we start working on the new branch. For example, we add the Javadoc to the `HelloMessage` in one commit, and the Javadoc to the method `getMessage` in another commit.

We then push this branch. Recall that we are going to push to our own GitHub repository, that is, to our fork, not to the original project's repository. On our fork web site, we find the button for creating a PR:

Some Java examples, just for experimenting with GitHub

Manage topics

7 commits 3 branches 0 releases 1 contributor

Your recently pushed branches:

document-hello-message (3 minutes ago)

Compare & pull request

Note that the PR spawns two different GitHub repositories:

### Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base fork: LorenzoBettini/java-examples base: master head fork: LoreBett/java-examples compare: document-hello-message

Able to merge. These branches can be automatically merged.

Indeed, we ask the owner of the original project to pull the changes from our fork's branch `document-hello-message` into the original repository `master` branch. Thus, GitHub allows contributors to easily implement the overall process of pull requests that we briefly describe in an informal way in Section [Pull requests](#).

As usual, we can provide a description and add the `Closes` string referring to the issue we previously created. Of course, we could take some time to review our own proposed changes and finally create the PR.

We then wait for the response from the owner of the original repository.

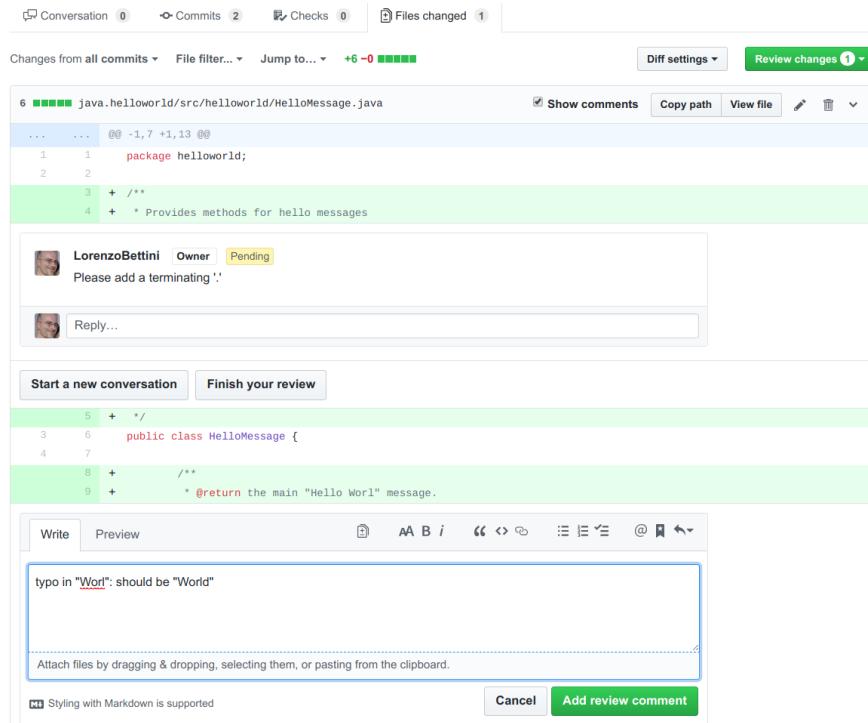
#### 9.4.3.2 The owner reviews the PR

We now impersonate the owner, i.e., the user `LorenzoBettini`.

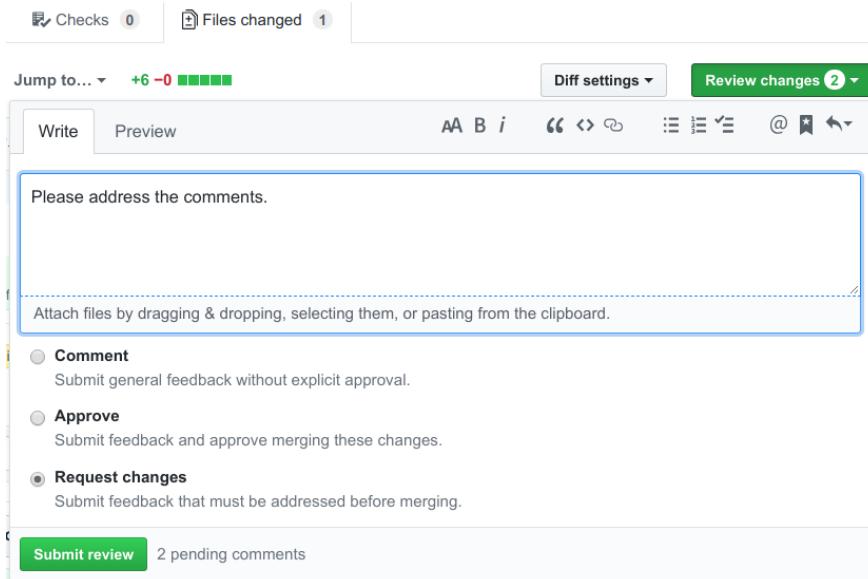
We receive a notification of a new PR from a contributor. We visit the web page of the PR. The page provides instructions (“command line instructions”) for pulling the proposed changes into our own local Git repository from the remote repository of the contributor, so that we can experiment locally with the proposed changes. This is useful in case the PR includes many changes to many parts of the project.

In any case, especially for small changes, we can review the proposed changes directly from the PR web page in the “Files changed” tab. As we saw before, this view provides a flat view of all the changes proposed in this PR. If we think that the PR needs some changes before being merged, we can put comments on every single line of this view, by pressing the pop-up + that appears by hovering on a single line. We can add a single comment or use “Start a review” so that we can specify many comments and request changes.

In this example, we create a first comment, start a review, and add another comment:



When we are done with our comments we select “Review changes”, specifying whether we are requesting changes to the contributor, by possibly adding additional general comments to the review:



This review, together with its detailed comments, becomes part of the discussion of the PR.

### 9.4.3.3 The contributor addresses the requested changes

Let's go back to impersonating the contributor, i.e., the user `LoreBett`.

We see that the owner requested some changes. We can see that by looking at the PR web page where each detailed comment is shown.

We implement the requested changes (or we can answer in the PR web page to a specific comment if we do not agree) locally on our project. Commit the changes and push our branch again to our remote fork. The original PR will be automatically updated with the new commit.

### 9.4.3.4 The owner merges the PR

Let's go back to impersonating the owner, i.e., the user `LorenzoBettini`.

We see that the contributor pushed further commits addressing our comments. We review again the proposed changes making sure all our concerns are addressed (alternatively, we can discuss in the PR requested changes that the contributor did not agree with).

If we find other issues to be solved in the PR, we start a new review cycle. Otherwise, we decide to merge the contributor's PR. We do that just like we did for our PR in the previous sections. It might be worthwhile to thank the contributor by adding a final comment to the PR.

### 9.4.3.5 The contributor updates the fork

Let's go back to impersonating the contributor, i.e., the user `LoreBett`.

Now that the PR has been merged into the original GitHub repository, our fork is not in sync with the original GitHub repository. If we plan to contribute to this repository in the future we need to keep our fork up to date with respect to the original repository.

To do that, we add the original repository URI as another remote repository of our local repository. Recall that the `origin` remote repository points to our own GitHub fork. We add this additional remote with another name. Typically, we use `upstream` to name the main remote repository of a project we are contributing to:

```
1 git remote add upstream https://github.com/LorenzoBettini/java-examples.git
```

Then, we fetch from `upstream` so that we have references to the remote branches of the original GitHub repository.

We then reset our local `master` branch to the `upstream/master`. Finally, we push our local `master` to our own GitHub forked repository (that is, the remote `origin`).

As usual, we can remove both locally and remotely the branch we used for the PR, `document-hello-message`, since now its commits are merged into `master`.

# 10. Continuous Integration

In the previous chapters, we cared about writing small and modular components, which are loosely coupled and thus easy to test. We aimed at having a test case for every single SUT (e.g., class). We learned that being able to write tests verifying single components in isolation, e.g., with mocking, allows us to keep our unit tests fast. Thus, it is rather cheap to run all the unit tests. Soon we will learn also about integration tests that verify the behavior of several components when wired together. These tests will be slower than unit tests.

In general, when working on an application made up of several components, during the typical development cycles, you work on a single part of the application, and you run the tests only of that part. However, at some point, you might also want to run all the tests of the application, and, since these might include integration tests, this process is expected to take some time: several minutes (even for small applications) if not hours (for big and real life applications).

For these reasons, the integration of all the components of the application, which, in the end, includes the whole application itself, might be tested only seldom. This is dangerous because problems due to the integration of components might be discovered too late, and fixing such integration problems might require too much time.

**Continuous Integration**, abbreviated as **CI**, ([Fow06](#)) is a software development practice consisting of integrating all the changes into the whole application or project continuously, several times a day. Such integration is meant to be built automatically, verifying that the compilation succeeds, and completely tested, running all the tests of the application. If a bug is introduced into the application by a single component, this is detected at the very first integration of that component in the application. Of course, CI relies on an automatic build mechanism (like Maven, Chapter [Maven](#)) for performing all its tasks. Moreover, a version control system (like Git, Chapter [Git](#)) is required to store the whole application codebase. Finally, the build process is delegated to a dedicated server, called **Continuous Integration server (CI server)**.

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

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.

Even if a complete integration build can take several minutes, the developers can continue working while the server executes the build. Periodically, the developers check the results provided by the CI server, and, in case of failure of a particular build, the breaking changes of the corresponding commit can be analyzed and fixed.

Besides building and running the tests, a CI build process typically executes additional checks on the code, like, e.g., creating a report of code coverage and executing further code analysis with code quality tools (see Chapter [Code Quality](#)). Again, these additional steps are usually skipped when developing, since 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 the developers through some web pages. The history of such reports will then be kept to have some statistics on the project reports.

In this book, we implement the CI process by using

### GitHub

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

### GitHub Actions

(<https://github.com/features/actions>) as the CI server, where we will use Maven to build and test our projects. GitHub Actions provides virtual execution environments for Linux OS (which we'll use by default), macOS, and Windows.

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

- We push to our Git repository on GitHub;
- This triggers a build on GitHub Actions;
- The project is built;
- The tests are run with code coverage;
- Code coverage results are sent to Coveralls;
- Code quality analysis is performed on SonarCloud;
- Results from all the remote services are sent back to GitHub, and will be used to annotate the commit and possibly the PR with such results (including failures).

## 10.1 Our running 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.

We create a Maven project in Eclipse with the archetype `maven-archetype-quickstart` as done before. We'll keep the POM to the minimum needed (we do not even change the Java compilation level, we only change the JUnit version as usual):

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3
4   <groupId>com.examples</groupId>
5   <artifactId>myproject</artifactId>
6   <version>0.0.1-SNAPSHOT</version>
7   <packaging>jar</packaging>
8
9   <properties>
10    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
11  </properties>
12
13  <dependencies>
14    <dependency>
15      <groupId>junit</groupId>
16      <artifactId>junit</artifactId>
17      <version>4.13</version>
18      <scope>test</scope>
19    </dependency>
20  </dependencies>
21 </project>
```



This example assumes that you are using Java 8. We'll test the project with other versions of Java later in this chapter.

The SUT and the corresponding test case are really simple:

```
1 package com.examples.myproject;
2
3 public class App {
4     public String sayHello() {
5         return "Hello";
6     }
7 }
```

```
1 package com.examples.myproject;
2 ...
3 public class AppTest {
4     private App app;
5
6     @Before
7     public void setup() {
8         app = new App();
9     }
10
11    @Test
12    public void testSayHello() {
13        assertEquals("Hello", app.sayHello());
14    }
15 }
```



Although we aim at running the build on the CI server, it is best to run the build at least once on the local machine, to make sure the build works locally. It might be hard to understand why a build fails on the CI server, thus it is better to make sure it works at least on our machine.

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

## 10.2 GitHub Actions

GitHub Actions allows us to execute build automation workflows that build and test the code in GitHub repositories. Such workflows run by default on GitHub-hosted virtual machines, but they can also run on machines that we host ourselves. We will only use GitHub-hosted virtual machines for our CI process.

We will have such CI workflows run automatically when we push a branch to GitHub and when we create a PR. In general, GitHub Actions workflows can be executed also when other events occur or on a set schedule.

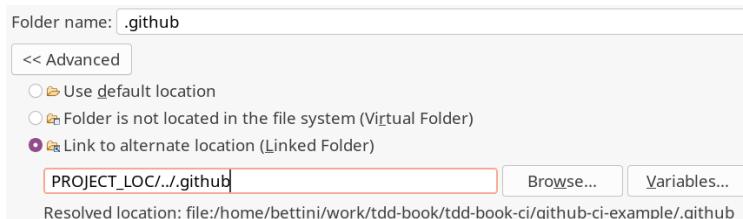
GitHub Actions workflows are specified using YAML, <https://yaml.org/>, which is a typical format for configuration files. The name is a recursive acronym: “YAML Ain’t Markup Language”. The website provides also the specification of the YAML language. In YAML whitespace characters are part of the syntax (differently, e.g., from Java). Indentation denotes the structure of the configuration. On the contrary, tab characters cannot be used as part of that indentation. Comments begin with

#. Associative array entries, key-value pairs, are denoted by `key: value`. List elements start with - with one member per line. Alternatively, list elements, separated by a comma, can be specified in square brackets [ . . . ]. We will see other features in the rest of the chapter when we need them for writing our GitHub Actions workflows.

GitHub Actions workflow files, with extension `.yml` or `.yaml`, are expected to be located in the directory `.github/workflows` in the root directory of the Git repository. As we will see, we can have as many workflow files as we want in that directory. They will all be inspected and possibly executed by GitHub Actions.

We could create the first workflow by using the link “Actions” in the GitHub web interface of our GitHub repository. GitHub web interface provides some “getting started” guides for programming languages and build tools, including Java and Maven. It creates a starting template and provides an on-line editor with some support for syntax highlighting and code completion. Then, it will allow us to commit the newly created workflow file in the Git repository. In this chapter, we manually create the workflow file in our local Git repository.

If we have a single project in the repository that is in its subdirectory, we do not have direct access to the `.github/workflows` directory from Eclipse, because we do not have direct access to the root folder of the Git repository. We could create YAML files in there from outside Eclipse and then edit them from Eclipse (with **File → Open File...**), or with the default system text editor. Alternatively, we can create in the project a “New Folder”, `.github`, which is a “Linked Folder”. From the “New Folder” dialog expand “Advanced” and select “Link to alternate location (Linked Folder)”. Click on “Variables...” and select “PROJECT\_LOC”. Finally, specify the path to the linked physical directory `.github` relative to the project’s directory. In this example, the relative path is `../.github`.



**Creating a “Linked Folder” from Eclipse**

If we did not create the physical directory in advance, when pressing “Finish”, the wizard will propose to create that for us. In the `.project` file, the following XML element will be created:<sup>1</sup>

---

<sup>1</sup>Once you have learned this procedure, you could also paste the XML snippet directly into your `.project` file, instead of using the wizard.

```
1 ...
2 </natures>
3 <linkedResources>
4   <link>
5     <name>.github</name>
6     <type>2</type>
7     <locationURI>PARENT-1-PROJECT_LOC/.github</locationURI>
8   </link>
9 </linkedResources>
10 </projectDescription>
```

Note that the reference is relative to the project's location, so this solution is portable to different computers and it will be independent of the absolute path of the project. In Eclipse, the "Project Explorer" will show the .github folder with an icon representing the link. (Remember that resources starting with . are hidden by default in Eclipse, but we have already seen how to make them visible.) Now, we can create the workflows subfolder of .github directly from Eclipse. Inside that folder, we create a YAML file from Eclipse as well, e.g., maven.yml.



You can use the Eclipse preference **Editors → File Associations**, "Open unassociated files with" for specifying "Search Marketplace", which should be enabled by default. The first time you open a file with an unknown extension in Eclipse, you get a proposal to search for an editor for that format in the Marketplace. For example, for YAML you can choose *Yaml Editor*, *YEdit* and others, like the editor provided by the project *Wild Web Developer*. The latter is also aware of the YAML structure of GitHub Actions workflows, so its code completion capabilities can be very useful.

These are the initial contents of the file

```
1 # This workflow will build a Java project with Maven
2 name: Java CI with Maven in Linux
3
4 on:
5   push:
6   pull_request:
7
8 jobs:
9   build:
10    runs-on: ubuntu-latest
11
12   steps:
13     - uses: actions/checkout@v2
14     - name: Set up JDK 8
```

```

15    uses: actions/setup-java@v1
16    with:
17      java-version: 1.8
18    - name: Build with Maven
19      run: mvn -f com.examples.myproject/pom.xml clean verify

```

Let's see what the elements of this workflow mean.

The `name`: of the workflow will appear in the Actions tab of the GitHub repository (see later in this section).

The `on`: part specifies when the workflow will be triggered. In the above configuration, as you can imagine, we request the workflow to be executed each time we push a commit on any branch and each time a pull request is created or updated (see also Section [Building pull requests](#)). With that respect, we have full control of such a specification. For example, we might want to build our project only when a push is performed on the `master` branch or on any branch with a name that starts with `task` like that:

```

1 # just an example
2 on:
3   push:
4     branches:
5       - master
6       - task*

```

The same specifications could be done also in the `pull_request`: part. For example, we might want to build pull requests that involve the `master` branch only. This is left as an exercise.

Workflows are made up of one or more **jobs**. Here we are specifying a single job, `build`. We specify the **runner** for this job by providing the name of one of the virtual environments provided by GitHub Actions. In this example, we specify the latest version of Ubuntu (see also Section [Building using different operating systems](#) and Section [Reproducibility in CI](#)). Each job in a workflow runs in a fresh virtual environment. Data produced in previous workflow runs are lost. We will see that we can use some mechanisms to reuse some of the data of the previous builds (Section [Caching dependencies](#) and Section [Storing artifacts](#)).

A job consists of an ordered list of **steps** that will be executed on the same runner of the job. Thus, the steps naturally share input and output data with the other steps of the same job. A step can be given a name that will be shown in the log of the workflow execution (see later in this section). A step can be a shell command like the `run`: step above. In this example, such a step executes the actual Maven build, as we would do on our computer. Please keep in mind that the used shell depends on the OS of the runner, in this example, we are using the Linux Bash shell.

Alternatively, a step can “use” an **action** (that is where the name “GitHub Actions” come from), with the keyword `uses`:. Actions are standalone commands and, to some extent, actions play the

same role as Maven plugins. In the above workflow, we are using two predefined available actions `actions/checkout@v2` and `actions/setup-java@v1`.

The first action is common to most workflows since it checks out (i.e., it clones) our GitHub repository on the runner. This will allow us to build and test the project hosted on this GitHub repository. The second action installs a specific version of the Java JDK on the current runner. Arguments to actions are specified with the keyword `with:`. The actual arguments depend on the specific action and can be found on the action's website. For example, for `setup-java` we specify the version of the JDK we want to install with the argument `java-version:`. Several predefined actions can be found in the GitHub Actions market place <https://github.com/marketplace/actions/>. You can also write your own actions.

An action can be specified with a few syntactic forms. The one we are using is `{owner}/{repo}@{ref}`, where “owner” is the owner of the GitHub repository hosting the action, specified by “repo”. The part after @ is a Git reference, that is, a *tag*, a *branch* or even a specific identifier of a commit. For instance, the GitHub repository of the `checkout` action is <https://github.com/actions/checkout>.

Let's add this file (stage) to the git repository, commit and push to GitHub. In this example, we specify the commit message “First GitHub workflow”.

In a few seconds, the workflow will start. We can verify that by navigating to “Actions”. Note the name we specified in the workflow “Java CI with Maven in Linux” and the message of the commit in the screenshot.

The screenshot shows the GitHub Actions interface. At the top, there is a navigation bar with links for Issues, Pull requests, Actions (which is highlighted in red), Projects, Wiki, Security, Insights, and Settings. Below the navigation bar, there is a section titled "Workflows" with a "New workflow" button and a "All workflows" button, which is highlighted with a blue background. To the right of this section, there is a feedback message: "Tell us how to make GitHub Actions work better for you with three quick ques". Below this, there is a search bar labeled "Filter workflows" and a section titled "All workflows" with a "1 result" indicator. The result is a card for a workflow named "First GitHub workflow", which was triggered by a commit to the "master" branch of the "Java CI with Maven in Linux" repository. The commit message was "Java CI with Maven in Linux #1: Commit 7037903 pushed by LorenzoBettini".

Our first workflow running on GitHub Actions

By clicking on the link of the commit we access the details of the running workflow and we can expand the steps to see the log, for example, of the actual Maven build, as shown in the following screenshot:



### Inspecting the log of the workflow

Note how, in the above screenshots, the strings “Java CI with Maven in Linux” and “build” match with the contents of the workflow we have written above. The same holds for the names used for steps. If we haven’t specified any name for a specific step, the log will show a default message (like “Run actions/checkout@v2”).

The green mark shows that the workflow has already finished successfully. If the workflow is still running we can follow the log of the Maven build (or of any other step in execution). By clicking on the “gear” icon on the top right we can also access the complete raw log or download an archive with the log so that we can inspect it locally. Please keep in mind that logs of the executed workflow will be removed automatically after some time.

From now on, every time we push commits to the GitHub repository of this project, the project will be automatically built by our GitHub Actions workflow.

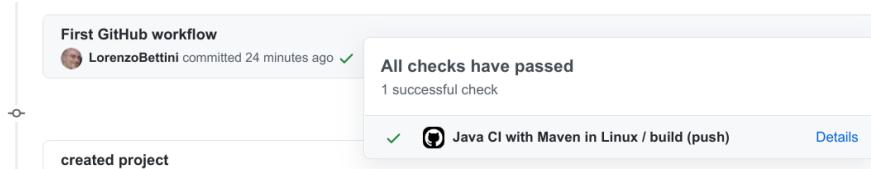


The workflow will always be executed on the last pushed commit of a branch; for example, if we create 3 commits locally and then we push, the 2 intermediate commits will not be built. If we want to run the CI process on every single commit, we must always push after committing.



A workflow can be manually restarted using the top right button “Re-run jobs”. That will always rebuild the very commit associated with that specific build.

The interaction between GitHub and GitHub Actions is bidirectional: GitHub Actions tells GitHub whether the build for a commit succeeded or not, and GitHub will show this information on the commit. If we go to the “commits” section of the GitHub repository we can see:

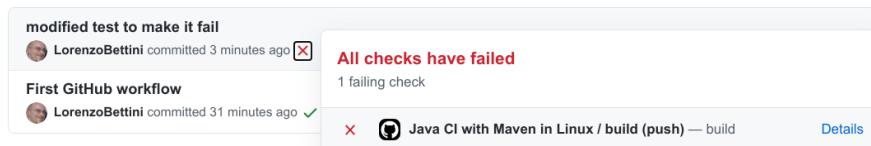


Positive feedback reported on the commit on GitHub

The “Details” link will bring us directly to the build job corresponding to that commit.

Depending on our settings, we will also get emails with the outcome of the build, e.g., when it fails.

Let’s modify the test to make it fail, commit and push. As expected the build fails and this is reported on the corresponding commit on GitHub:



Negative feedback reported on the commit on GitHub

Let’s fix the test, e.g., by reverting the commit, and push. Everything will get back to a green state.

## 10.2.1 Caching dependencies

We know that Maven automatically downloads dependencies and caches them in the .m2 in the home folder so that new dependencies are downloaded only once. However, each build job in GitHub Actions workflows is executed on a brand new virtual environment, and the state is not kept across jobs. Thus, cached Maven dependencies are lost. In fact, from the log of each build, we can see that the Maven dependencies are always downloaded from scratch. For a big project with many dependencies, this is a waste of time.

There is a specific action, `cache`, to cache data to be re-used in further builds of a project:

<https://docs.github.com/en/actions/guides/caching-dependencies-to-speed-up-workflows>.

We can use this action in the YAML file and configure it appropriately:

```

1 ...
2   with:
3     java-version: 1.8
4   - name: Cache Maven packages
5     uses: actions/cache@v2
6     with:
7       path: ~/.m2
8       key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
9       restore-keys: ${{ runner.os }}-m2-
10  - name: Build with Maven
11 ...

```

As shown above, we have to specify the path to store in the cache and to restore in the next builds. This can be an absolute path or relative to the working directory. Our configuration specifies to cache the whole directory `.m2` where Maven stores the downloaded dependencies and plugins. This cache will be reused on the next builds so that already downloaded dependencies will not be downloaded again. Note that if you are using the Maven wrapper (Chapter [Maven](#), Section [Maven wrapper](#)), the above directive will cache the Maven distribution used by the wrapper as well. We must also specify the key that identifies the cache. The key can use constant strings (like `-m2-`), variables (like the `runner.os`), and functions (like `hashFiles('**/pom.xml')` that computes a hash using all the POM files of the project). The `restore-keys` is optional: we can specify a list of alternative keys for the cache in case there is no exact match. For example, if we have already cached some contents and we change one of our POM, the cache key will be different due to the new hash value. Thus, there will be no exact match, but since we specify an alternative restore key without the hash value, we can still start from the previously cached values. Of course, the cache will then be updated with a new key if the build terminates successfully.

Let's commit and push. We can now verify that the build job will also store a cache with the `.m2` contents when the build terminates successfully. We can also verify that if we restart the build or we push some more commits, the cache will be used and Maven will not download artifacts from the Internet.



How the cache is used for Git branches is documented on the action's documentation site shown above. The documentation also specifies when GitHub will automatically remove the caches and the size limit for caches.

Note that if a cache is found matching exactly the cache key (**cache hit**), at the end of a successful job, the cache will not be updated at all, even if we had something new to store in the cache. For example, we use a function like `hashFiles('**/pom.xml')` in the cache key. This means that if we do not change any of our POM and we have a cache hit the cache will not be updated even if Maven had to download some new artifacts. This might happen, for example, if we change our workflow and we explicitly execute a new goal without changing our POM. The new artifacts downloaded for the new goal will not be saved in the cache and will have to be downloaded on the next builds.

Somehow, this strategy forces us to add a configuration in the POM, e.g., in the `<pluginManagement>` section, for the new plugins, we are going to use in our CI workflow: we change our POM with the configuration of the new plugins and, since a cache hit will not happen, the cache will be updated with the new downloaded artifacts. As said in Chapter [Maven](#), Section [Plugin management](#), it is important to lock down plugins versions and avoid using the default versions of plugins, which depend on the current Maven installation. This way, we have full control over the versions of the plugins.

However, this might still not be enough to fully enjoy the caching mechanism. Let's say we configure a new plugin in `<pluginManagement>` but we are still not using it. We push the commit, the POM will be different so we will not have a cache hit and a new cache will be saved. Unfortunately, since we are not using that plugin, no new Maven artifacts will be downloaded and stored in the cache.

Now we change our YAML workflow to enable that new plugin and push. The POM hasn't changed now and we will have a cache hit. New Maven artifacts will be downloaded for the new used plugin but the cache will not be updated.

Thus, we should use a cache key that is sensitive to changes that require updating the cache. From what we have just seen, hashing POM files might not be enough: we should hash also YAML files:

```
1 key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml', '**/*.yml') }}
```

```
2 restore-keys: ${{ runner.os }}-m2-
```

The `restore-keys` can stay the same. Remember that, as said above, if we don't have a cache hit but a cache matching the `restore-keys` specification is found, we will not start with an empty cache.

Depending on your language, platform, and build tool, you might have to adapt the above strategy.

Finally, it is worth knowing that we can use as many "cache" steps as we want in our workflows. Thus, we can create several caches with different keys.

## 10.3 Build matrix

One of the useful features of Continuous Integration servers is the ability to build and test our projects with different environments. In particular, we can build and test our projects with several versions of the programming language and with several operating systems and architectures that we do not even own.

GitHub Actions provides a mechanism called **build matrix** that allows us to run our workflows across multiple combinations of operating systems, platforms, and languages.

The build matrix is specified in the YAML workflow with the keywords `strategy` and `matrix`, and we specify the build options as an array. Basically, we provide a set of pairs of key-values. The GitHub Actions CI infrastructure will use every combination of the pairs and will spawn several parallel jobs starting from a single job definition in the workflow. Each job can access the current combination of values for the corresponding keys.

In the next sections, we will experiment with this mechanism.



GitHub automatically cancels all the in-progress jobs of a matrix if any job of the matrix fails. This mechanism is called "fail fast". We can specify `fail-fast: false` in the `strategy` section to avoid that and to allow all jobs of the same matrix to finish.

### 10.3.1 Building using different JDKs

We can use GitHub Actions build matrix to build and test our projects with different Java versions.

Indeed, in our first workflow, we used a specific GitHub action, `setup-java`, <https://github.com/actions/setup-java>, to specify the JDK to use. In this example, we use version 1.8. In our case, we modify our workflow as follows, so that we build and test our project with JDK 8 and JDK 11 (that is, the current LTS versions):

```
1 ...
2 jobs:
3   build:
4     runs-on: ubuntu-latest
5     strategy:
6       matrix:
7         # test against several Java versions:
8         java: [ 8, 11 ]
9
10    name: Build with Java ${{ matrix.java }}
11    steps:
12      - uses: actions/checkout@v2
13      - name: Set up JDK ${{ matrix.java }}
14        uses: actions/setup-java@v1
15        with:
16          java-version: ${{ matrix.java }}
17      - name: Cache Maven packages
18      ...
```

Note how we access the value of the current matrix configuration using the variable `matrix.java` in several parts of our job specification. The variable depends on the key used in the matrix configuration. In this example, it is `java`. If we specified, for example, `jdk: [ 8, 11 ]` then we should use the variable `matrix.jdk`.

In this example, we specify a major Java version, but the action allows us to have more control over the configuration of the JDK (see also Section [Reproducibility in CI](#)), as documented in the action's website shown above. The website provides information also on the JDK distributions available. It is also possible to manually install a specific JDK without using any specific action, that is, with a `run` step, but this is out of the scope of the book.

Commit and push. Now, instead of a single job for this commit, we will see two parallel jobs, each one using a JDK specified in the build matrix. To see the log of a specific job we need to click on the corresponding link. At the end of the log, we can see that the job using JDK 11 fails:

The screenshot shows a 'Summary' view of a CI pipeline. At the top, there's a 'Jobs' section with two entries: 'Build with Java 8' (marked with a green checkmark) and 'Build with Java 11' (marked with a red X). Below this, a message reads 'Multiple parallel jobs, one failing'.

If we look at the log of that job we can see the source of the failure:

```
1 Error: COMPILATION ERROR :  
2 Error: Source option 5 is no longer supported. Use 6 or later.  
3 Error: Target option 1.5 is no longer supported. Use 1.6 or later.
```

The error message should be clear: the project is by default configured to use Java 5 (recall that we did not specify the Java compilation level in the POM) and Java 11 does not support Java 5 anymore (actually, support for Java 5 has been removed since Java 9).

Of course, we did not aim at making our project compatible with Java 5 and we forgot to specify the Java 8 compilation level. So let's update our POM as we have done before with the corresponding settings

```
1 <properties>  
2 ...  
3   <maven.compiler.source>1.8</maven.compiler.source>  
4   <maven.compiler.target>1.8</maven.compiler.target>  
5 </properties>
```

Commit and push. Now both jobs will succeed.

Note that this CI process allowed us to build and test our project originally developed with Java 8 even with more recent versions, without manually verifying that on our system. We could also add additional versions of the JDK in our matrix. In particular, we could also use an “Early Access” release of the JDK (that is, a preview release). In the `setup-java` action an early access release typically consists of the major version and the suffix `-ea`.

Recall that each build will consist of several parallel jobs, one for each value in the matrix.



If two jobs are using the same cache key, like in this example, and they try to save a new cache at the same time, only one of them will succeed and in the other job's log a message will be shown "Unable to reserve cache with key ..., another job may be creating this cache." If the two jobs execute the same build process that should not be a problem. On the contrary, if one of the two jobs executes a slightly different build (e.g., by executing additional Maven goals, as we will see later starting from Section [Code coverage in CI](#)), the cache for that job with new downloaded artifacts might not be updated (see also Section [Caching dependencies](#)). It might be the case to provide two different cache keys for the two jobs. For example, we might use also the Java version in the key: `key: ${{ runner.os }}-m2-jdk${{ matrix.java }}-${{ hashFiles('**/pom.xml', '**/*.yml') }}`.

### 10.3.2 Building using different operating systems

Up to now, we have always used the default operating system provided by GitHub Actions, Linux. GitHub Actions also provides virtual machines for Windows and macOS.

The list of available environments and the installed software on each environment can be found here: <https://github.com/actions/virtual-environments>.

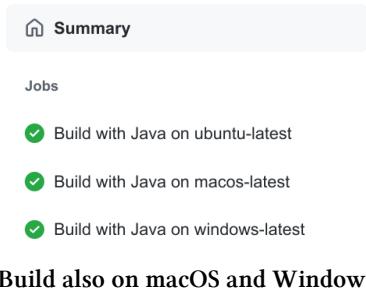
We can use the build matrix mechanism shown in the previous section to specify the different operating systems. Let's modify our workflow file as follows:

```
1 ...
2 jobs:
3   build:
4     runs-on: ${{ matrix.os }}
5     strategy:
6       matrix:
7         # test against several OSes:
8         os: [ubuntu-latest, macos-latest, windows-latest]
9
10    name: Build with Java on ${{ matrix.os }}
11    steps:
12      - uses: actions/checkout@v2
13      - name: Set up JDK 8
14        uses: actions/setup-java@v1
15        with:
16          java-version: 8
17      - name: Cache Maven packages
18        uses: actions/cache@v2
19        with:
20          path: ~/.m2
21          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
```

```
22     restore-keys: ${{ runner.os }}-m2-
23     ...
```

We reverted to a fixed version of Java while using a matrix for OSes. In particular, we changed `runs-on` so that it uses the matrix value `matrix.os`. Note that in the cache action we are still using the variable `runner.os` and not `matrix.os`. In fact, `runner.os` is automatically set by GitHub according to the OS specified in `runs-on`, so there's no need to update the cache step.

Commit and push. Now 3 jobs will be run for each build, one for each of the specified operating systems. The build succeeds on all the 3 environments:



This way, we can build and test our projects on 3 operating systems, that is, even in environments that we might not have available on our own computers.

### 10.3.3 Configuring the build matrix

We could also specify a more complex matrix using what we saw in the previous two sections together:

```
1  jobs:
2    build:
3      runs-on: ${{ matrix.os }}
4      strategy:
5        matrix:
6          # test against several OSes:
7          os: [ubuntu-latest, macos-latest, windows-latest]
8          # test against several Java versions:
9          java: [ 8, 11 ]
10
11        name: Build with Java ${{ matrix.java }} on ${{ matrix.os }}
12        steps:
13        - uses: actions/checkout@v2
14        - name: Set up JDK ${{ matrix.java }}
15          uses: actions/setup-java@v1
16          with:
```

```
17     java-version: ${{ matrix.java }}  
18     - name: Cache Maven packages  
19     ...
```

With this modification we will have a build matrix with 6 jobs, that is, all the combinations specified in the matrix.

For a very simple project like this one, building all the configurations takes only a few minutes. However, for a more complex project this build matrix could be too big, would require too much time, and would probably waste too much computing power (not our computer, but on GitHub infrastructure, which is provided for free).

We can then explicitly remove some specific configurations from the build matrix. For example, we might build with JDK 8 and 11 only on Linux, and on the other OSes we build only with JDK 8:

```
1 matrix:  
2     # test against several OSes:  
3     os: [ubuntu-latest, macos-latest, windows-latest]  
4     # test against several Java versions:  
5     java: [ 8, 11 ]  
6     # excludes JDK 11 on macOS and Windows  
7     exclude:  
8     - os: macos-latest  
9         java: 11  
10    - os: windows-latest  
11        java: 11
```

Alternatively, we can explicitly include a configuration in a smaller build matrix:

```
1 matrix:  
2     # test against several OSes:  
3     os: [ubuntu-latest, macos-latest, windows-latest]  
4     # test against several Java versions:  
5     java: [ 8 ]  
6     # includes JDK 11 on Linux  
7     include:  
8     - os: ubuntu-latest  
9         java: 11
```

The two above configurations will lead to the same result.

The `include` and `exclude` can be combined to create more complex configurations, keeping in mind that `include` specifications are processed after `exclude`.

## 10.4 Building pull requests

Note that in the YAML configuration we have this specification:

```
1 ...
2 on:
3   push:
4     pull_request:
5
6 jobs:
7 ...
```

Thus, the workflow will be triggered not only for standard pushes, but also for pull requests. In particular, by default, our workflow will build a pull request when it is opened, and whenever commits are added to the pull request. Instead, when a pull request is merged, this will correspond to a push on the base branch.

It is crucial to understand that when a pull request is built, the workflow will first perform a merge between the two branches involved in the pull request. This is quite useful, since, if the two branches have diverged in the meantime (but they can still be merged without conflicts), the workflow will build the project as if the two branches have already been merged. Thus, we get immediate feedback about what happens when the branches are finally merged.

Indeed, two divergent branches can be merged without conflicts, but the resulting code might not compile or existing tests might fail. Thus, the above behavior will allow us to avoid such bad situations, since, in case, the build of the pull request will fail.

### 10.4.1 Multiple workflows

We are free to add as many workflows as we want to the directory `.github/workflows` with any name. GitHub Actions will inspect them all and trigger them if the `on:` specification matches (or other explicit conditions are met).

Let's experiment with this feature. We create a new local branch, e.g., `multiple-workflows`, and switch to that branch.

We might want to refactor our continuous integration strategy as follows (this is just an example):

- we want to build on Linux with JDK 8 and JDK 11 always, that is, both on pushes and pull requests
- we want to build on macOS and Windows with JDK 8 only on pull requests.

Let's create a new YAML file, e.g., `pr-windows-mac.yml` inside `.github/workflows` with the following contents:

```
1 name: Java CI with Maven for PR
2
3 on:
4   pull_request:
5
6 jobs:
7   build:
8     runs-on: ${{ matrix.os }}
9     strategy:
10    matrix:
11      # test against several OSes:
12      os: [macos-latest, windows-latest]
13
14    name: Build with Java 8 on ${{ matrix.os }}
15    steps:
16    - uses: actions/checkout@v2
17    - name: Set up JDK 8
18      uses: actions/setup-java@v1
19      with:
20        java-version: 8
21    - name: Build with Maven
22      run: mvn -f com.examples.myproject/pom.xml clean verify
```

Note that the `on:` part specifies that this workflow must be triggered only on pull requests. For simplicity, we skipped the cache part.

Our previous `maven.yml` can now be changed as follows (it could also be renamed with a more sensible name if you want):

```
1 name: Java CI with Maven on Linux
2
3 on:
4   push:
5   pull_request:
6
7 jobs:
8   build:
9     runs-on: ubuntu-latest
10    strategy:
11      matrix:
12        # test against several Java versions:
13        java: [ 8, 11 ]
```

```
15   name: Build with Java ${{ matrix.java }} on Linux
16   steps:
17     - uses: actions/checkout@v2
18     - name: Set up JDK ${{ matrix.java }}
19       uses: actions/setup-java@v1
20       with:
21         java-version: ${{ matrix.java }}
22 ... as before
```

Let's commit and push (remember we are on a new branch). We can verify that the new workflow `pr-windows-mac.yml` is not triggered, since we performed a standard push. Instead, we have two jobs running on Linux with two different JDKs.

Now let's create a new PR (as shown in Chapter *Git*, Section *GitHub pull requests*).

This time, we will see 6 jobs for the build of the PR, as we expected. GitHub Actions directly participates in the GitHub PR, by communicating that while the jobs of the branch have already finished successfully, the jobs for the PR are still in progress:



GitHub Actions participates to the PR

Remember that the two jobs defined in `maven.yml` are executed also for the pull request, according to the `on:` specification.

When all the jobs terminate (hopefully successfully), the PR will be updated.



GitHub Actions notifies the PR that all is good

We can now merge the pull request. If you followed what we said in the previous paragraphs, you should know that after the merging of the pull request two jobs will start (the ones defined in `maven.yml`).

## 10.5 Storing artifacts

GitHub Actions allows us to persist some artifacts after a job has been completed. For example, we could use this mechanism to store an HTML report of our unit tests. This way, we can inspect the report by downloading the artifact from a specific workflow execution.



Such artifacts will be automatically removed after some time, thus, they cannot be used to publish released versions of our projects. There are other mechanisms provided by GitHub Actions for that, but we will not show them in this book.

First of all, we have to generate such a report and we know that we can do that using the `maven-surefire-report-plugin` as we have seen in Chapter [Maven](#), Section [Run Maven goals](#).

Thus, we modify our workflows accordingly. For simplicity, we modify only the workflow `maven.yml` (we do that in a separate branch, e.g., `surefire-report` that then we will merge with a PR):

```

1 - name: Build with Maven
2   run: >
3     mvn -f com.examples.myproject/pom.xml
4     clean verify
5     surefire-report:report-only site:site -DgenerateReports=false

```

Here we used the *folded style* of YAML, note the `>`, which allows us to break a long line into several lines. Newlines will be replaced with spaces.

For what we said in Section [Caching dependencies](#) we configure the new plugins we use in the POM:

```

1 <build>
2   <pluginManagement>
3     <plugins>
4       <plugin>
5         <artifactId>maven-surefire-report-plugin</artifactId>
6         <version>2.22.2</version>
7       </plugin>
8       <plugin>
9         <artifactId>maven-site-plugin</artifactId>
10        <version>3.9.1</version>
11      </plugin>
12    </plugins>
13  </pluginManagement>
14 </build>

```

Then, we add a final step using the predefined action to store artifacts, `upload-artifact`, by providing a name for the archive (we distinguish the archive according to the job's used JDK) and the path of the directory to archive (we know that the HTML report will be generated in the `target/site` directory):

```

1 - name: Archive JUnit Report
2   uses: actions/upload-artifact@v2
3   with:
4     name: surefire-report-jdk-${{ matrix.java }}
5     path: '**/target/site'

```

Note that since we start a value with \*, which is a special character in YAML, we must enclose the pattern in quotes.

Let's commit and push. When the job finishes, in the "Summary" of the executed workflow, scrolling down the page, we can find the links to download the ZIP files with the stored JUnit reports.

The screenshot shows the GitHub Actions "Summary" page. On the left, under "Jobs", there are two entries: "Build with Java 8 on Linux" and "Build with Java 11 on Linux", both marked with green checkmarks. On the right, under "Artifacts", it says "Produced during runtime". A table lists the artifacts:

Name	Size
surefire-report-jdk-11	24.3 KB
surefire-report-jdk-8	24.3 KB

#### GitHub Actions stored artifacts

Now, let's add a failing test, like the following one:

```

1 public class App2Test {
2     private App app;
3
4     @Before
5     public void setup() {
6         app = new App();
7     }
8
9     @Test
10    public void testSayHello() {
11        assertEquals("Hallo", app.sayHello());
12    }
13 }
```

Let's commit and push. Of course, we expect the build to fail. However, by looking at the log of the jobs we spot two problems:

- Since the test phase fails, the additional goals `surefire-report:report-only site:site` are not executed at all by Maven, so no HTML report is generated.
- The step "Archive JUnit Report" is not executed at all, since the previous step has failed and the workflow will stop immediately, skipping further steps.

In our scenario, this is not desired, since we usually need to inspect the test report when there's a test failure.

We can solve both problems by forcing the execution of steps, even if previous steps have failed, by using the condition `if:` and the function `always()`, which always returns true. In particular, we create a new step where we run the Maven goals `surefire-report:report-only site:site` after the standard Maven build terminates (even if it failed):

```

1 - name: Build with Maven
2   run: mvn -f com.examples.myproject/pom.xml clean verify
3 - name: Generate JUnit Report
4   run: >
5     mvn -f com.examples.myproject/pom.xml
6       surefire-report:report-only site:site -DgenerateReports=false
7   if: ${{ always() }}
8 - name: Archive JUnit Report
9   uses: actions/upload-artifact@v2
10  if: ${{ always() }}
11  with:
12    name: surefire-report-jdk-${{ matrix.java }}
13    path: '**/target/site'
```



You can imagine that in the second Maven step it is crucial not to call the `clean` phase or we remove the information with the test results that are needed to create the report.

Now, even in the presence of test failures, the test report is created and stored. We can download it, extract it, and inspect it on our computer:

The screenshot shows a CI pipeline interface with three stages:

- AppTest**: Shows a green checkmark icon, the test name `testSayHello`, and a duration of `0.003`.
- App2Test**: Shows a red X icon, the test name `testSayHello + [ Detail ]`, and a duration of `0.009`. Below it, the error message `expected:<H[a]llo> but was:<H[e]llo>` is displayed.
- Failure Details**: A summary page with links to [Summary], [Package List], and [Test Cases]. It lists the failed test `testSayHello` with the same error message and file location `com.examples.myproject.App2Test:17`.

Inspecting locally the downloaded JUnit Report



Note that, since we typically need to inspect the JUnit report only when there's a test failure, we might also change the above `if:` conditions by using `${{ failure() }}`, which evaluates to true when any previous step of a job has failed. This way, we store the report only when there are test failures.

We can now remove the failing test, commit, push, create a PR and merge it when all the jobs are successful.

## 10.6 Code coverage in CI

Now that we set up our CI process with GitHub Actions, we would also like to perform code coverage during the build with JaCoCo. We will experiment with code coverage in CI in a brand new local branch, e.g., `coverage`.

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:

```
1 <build>
2   <pluginManagement>
3     <plugins>
4     ...
5       <plugin>
6         <groupId>org.jacoco</groupId>
7         <artifactId>jacoco-maven-plugin</artifactId>
8         <version>0.8.6</version>
9         <executions>
10           <execution>
11             <goals>
12               <!-- binds by default to the phase "initialize" -->
13               <goal>prepare-agent</goal>
14               <!-- binds by default to the phase "verify" -->
15               <goal>report</goal>
16             </goals>
17           </execution>
18         </executions>
19       </plugin>
20     </plugins>
21   </pluginManagement>
22 </build>
```

Then we create a specific profile

```
1 <profiles>
2   <profile>
3     <id>jacoco</id>
4     <build>
5       <plugins>
6         <plugin>
7           <!-- configured in pluginManagement -->
8           <groupId>org.jacoco</groupId>
9           <artifactId>jacoco-maven-plugin</artifactId>
10          </plugin>
11        </plugins>
12      </build>
13    </profile>
14 </profiles>
```

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

```
1 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.

Now, we want to perform code coverage in the GitHub Actions workflow. It is just a matter of changing the Maven command in our YAML workflow. Of course, we do that only in our main workflow `maven.yml`, since we do not expect any code coverage changes if we build and test our project also on Windows and macOS (unless our program has some specific parts and tests that are used only with specific Java versions or operating systems):

```
1 run: >
2   mvn -f com.examples.myproject/pom.xml
3   clean verify -Pjacoco
```

However, this workflow is configured with a build matrix and it will execute two parallel jobs, one with Java 8 and one with Java 11. Again, it makes no sense to perform code coverage in both jobs.<sup>2</sup> To implement this optimization in our workflow, we can define a specific key in our build matrix with additional arguments to pass to the Maven build (in this case, `-Pjacoco`). We do that only in a matrix configuration, e.g., the one that uses Java 8. To do that, we make the build matrix explicit by listing all the configurations with `include` (see Section [Configuring the build matrix](#)):

```
1 strategy:
2   # test against several Java versions:
3   matrix:
4     include:
5       - java: 8
6         additional-maven-args: "-Pjacoco"
7       - java: 11
```

The matrix variable `matrix.additional-maven-args` will be defined only when building with Java 8. When building with Java 11 it will not be defined and accessing `matrix.additional-maven-args` will result in an empty value. The run configuration of our Maven build will then be modified accordingly:

```
1 run: >
2   mvn -f com.examples.myproject/pom.xml
3   clean verify ${{ matrix.additional-maven-args }}
```

<sup>2</sup>In a small project like the one we are using, the overhead of code coverage will not be noticeable, but in bigger projects, it might impact the performance of the build significantly.

Let's commit and push. We can verify that code coverage will be executed only in the job using Java 8. We now create a PR and when all checks succeed, we merge it.

We saw in Chapter [Maven](#), Section [Configuring the JaCoCo Maven plugin](#), that we can configure the JaCoCo Maven plugin so that it makes the build fail if the code coverage percentage is below a given threshold. We could configure our POM accordingly. This way, our CI process would detect violations of our code coverage rules.

This strategy might not be ideal for a few reasons. If such a violation happened our build would fail and we would get the failure feedback on GitHub, e.g., during a PR (see Section [Building pull requests](#)), but we should then inspect the job log to understand the reason of the failure. (Was it a compilation failure? A test failure? Or was it due to code coverage?) From the log, we would then try to understand why the code coverage decreased, and this might not be easy. We could then try to reproduce the failure on our local machine, but that is somehow against the CI process that we set up. We could store the JaCoCo HTML report as an artifact in our job (see Section [Storing artifacts](#)). We could then download the report on our computer and inspect it locally. While this strategy makes sense for detecting failing tests, it is not ideal for code coverage.

In fact, we would like to have an integrated mechanism in our CI process for getting feedback about code coverage. We will see such a possible mechanism in the next sections.



As an exercise, try to store the `jacoco` directory, containing also the HTML report, as another artifact (see Section [Storing artifacts](#)).

## 10.7 Code coverage with Coveralls

Coveralls, <https://coveralls.io/>, is a free service that keeps track of the code coverage of public projects hosted on Git servers, including GitHub. Coveralls itself does not perform any code coverage, it only keeps track of it. Thus, the code coverage must be performed separately and then the code coverage information must be sent to Coveralls. Coveralls can be used when building from our computer or from a CI build. In both cases, we need a token key available from the user profile on Coveralls site.

We will first use Coveralls from our local computer, and then we will use it from our GitHub Actions workflow.

### 10.7.1 Using Coveralls from our computer

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”, identified by a “+” icon. Use “Sync Repos” if the project is not yet available from the list. Once found, enable the project by setting the corresponding switch to “ON”. This operation must be done for every GitHub repository that we want to use in Coveralls.

We will experiment with Coveralls in a brand new local branch, e.g., `ci-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.

We have already configured JaCoCo in our project in Section [Code coverage in CI](#).

Now we configure the `coveralls-maven-plugin`. We configure it in the `<pluginManagement>`. We do not bind any goals to any phase, since we plan to run its goal, `coveralls:report`, directly from the Maven command line (note the specific additional dependency needed when building with a recent version of Java):

```
1 <build>
2   <pluginManagement>
3     <plugins>
4       ...
5       <plugin>
6         <groupId>org.eluder.coveralls</groupId>
7         <artifactId>coveralls-maven-plugin</artifactId>
8         <version>4.3.0</version>
9         <dependencies>
10           <!-- This is required when using JDK 9 or higher
11             since javax.xml.bind has been removed from the JDK -->
12           <dependency>
13             <groupId>javax.xml.bind</groupId>
14             <artifactId>jaxb-api</artifactId>
15             <version>2.3.1</version>
16           </dependency>
17         </dependencies>
18       </plugin>
19     </plugins>
20   </pluginManagement>
21 </build>
```

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` was generated in a different folder, the `coveralls-maven-plugin` should be configured using `<jacocoReport>` elements, e.g.,

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

Let's commit this change, e.g., with the message "configured coveralls-maven-plugin", and push it to GitHub.

If we still haven't done that, let's create the JaCoCo report:

```
1 mvn verify -Pjacoco
```

Now we try to send the coverage report to Coveralls from our local computer, using the coverage report (the `jacoco.xml` file) that we have just created.

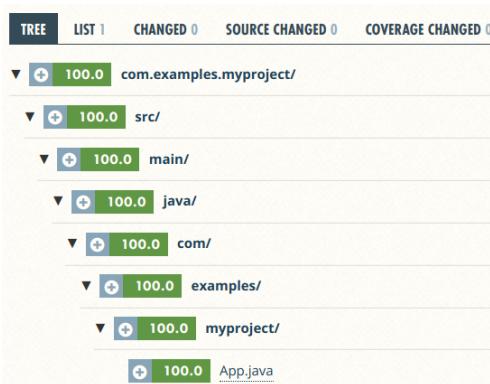
First of all, we have to use the token that Coveralls created and associated with this repository. On the "Settings" of this repository on Coveralls, we copy the "REPO TOKEN" (either on a local file or temporarily in memory). We can now try to use Coveralls from our local computer. Remember that we need the `jacoco.xml` file, so we just run the following goal without `clean`:

```
1 mvn coveralls:report -DrepoToken=<paste your repo token here>
```

We should see something like that:

```
1 [INFO] Processing coverage report from ...target/site/jacoco/jacoco.xml
2 [INFO] Successfully wrote Coveralls data in 170ms
3 [INFO] Gathered code coverage metrics for 1 source files with 7 lines of code:
4 [INFO] - 2 relevant lines
5 [INFO] - 2 covered lines
6 [INFO] - 0 missed lines
7 [INFO] Submitting Coveralls data to API
8 [INFO] https://coveralls.io/jobs/...
9 [INFO] *** It might take hours for Coveralls to update the actual coverage...
```

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



Our first coverage report on Coveralls

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



```
+ 100.0 /com.examples.myproject/src/main/java/com/examples/myproject/App.java
1 package com.examples.myproject;
2
3 public class App {
4     public String sayHello() {
5         return "Hello";
6     }
7 }
```

Code coverage on Coveralls



Coveralls shows colored sources if it finds the sources on the GitHub repository connected to this Coveralls repository. In particular, it uses the version of the sources that corresponds exactly to the current Git commit. For this reason, in this experiment from our local computer, we made sure to create a commit and push it to GitHub before running `coveralls:report`. If we forget to do that, Coveralls will not be able to show the sources with coverage information.

Coveralls communicates its results back to GitHub. The commit where Coveralls has been first used shows this new feedback (recall the commit message we used above):



Positive feedback from Coveralls reported on the commit on GitHub

Summarizing, this is what has just happened:

1. We created a commit and pushed it to GitHub;
2. We created the JaCoCo report on our computer and we sent it to Coveralls;
3. Coveralls stored the coverage report and sent feedback to GitHub for that specific commit.

Now we want the second step to be performed entirely on GitHub Actions. We will show that in the next section.



Experiment with the web links between Coveralls and GitHub. For example, from the feedback on the GitHub commit try to navigate to the corresponding Coveralls repository. Similarly, from the Coveralls report try to navigate to the corresponding commit on GitHub.

## 10.7.2 Using Coveralls from GitHub Actions

Our final goal is to send the coverage report to Coveralls from our CI infrastructure, that is, directly from the GitHub Actions workflow.

First of all, we must face the problem of the Coveralls repository token. How can we let the GitHub Actions workflow use our secret Coveralls repository token? We might be tempted to paste the repository token directly into the workflow YAML file. That would be a **terrible mistake**: we should never ever put our secrets (tokens or passwords) in clear text in a public GitHub repository! Anyone would then be able to use our secret tokens and passwords!

Fortunately, GitHub Actions provides a mechanism for the above issue: **encrypted secrets**: <https://docs.github.com/en/actions/reference/encrypted-secrets>. Such secrets are encrypted environment variables that we set in our own GitHub account settings or a specific GitHub repository that we own. Of course, such settings are visible and accessible only to us. Such environment variables are then available to use in GitHub Actions workflows. GitHub makes sure that the contents of such variables remain encrypted until we use them in a workflow.

In the “Settings” of our GitHub repository, we navigate to “Secrets” and we create a new repository secret. We specify COVERALLS\_TOKEN as the name and we paste our Coveralls repository token in the “Value” field. Finally, we press “Add secret”.

Now, we must use such an encrypted secret in our YAML workflow. Following the strategy described in Section [Code coverage in CI](#), we will use Coveralls only in a single matrix configuration, that is, the same where we perform code coverage. We define an **environment variable** in the corresponding step of the job, using the keyword `env`. We give the variable a name, e.g., `COVERALLS_REPO_TOKEN`, and we assign it the value of the secret we have created above that we access by its name prefixed with `secret`.<sup>3</sup>

```

1 - name: Build with Maven
2   run: >
3     mvn -f com.examples.myproject/pom.xml
4     clean verify ${{ matrix.additional-maven-args }}
5   env:
6     COVERALLS_REPO_TOKEN: ${{ secrets.COVERALLS_TOKEN }}
```

and we update our build matrix configuration, in particular the value for `additional-maven-args`:

```
1 additional-maven-args: "-Pjacoco -DrepoToken=$COVERALLS_REPO_TOKEN coveralls:report"
```

Let’s commit and push. In the log of the job using Java 8 we should see that code coverage is sent to Coveralls (the output is similar to what we saw when using Coveralls from our computer in the previous section). Just like in the previous section, Coveralls communicates its results back to GitHub.

---

<sup>3</sup>We will use this mechanism also in Chapter [Code Quality](#), Section [SonarCloud](#).



GitHub automatically hides secrets printed to the log of the job. You can verify that by looking at the log of the job. However, you should avoid manually printing secrets to the log or you will reveal their values.

Let's create a PR for this branch. We can see that Coveralls also participates in the PR with its feedback. We can merge the PR

### 10.7.3 Coverage threshold

In Section [Code coverage in CI](#), we said that we want a mechanism integrated with our CI process that notifies us if the code coverage goes below a given threshold. Coveralls allows us to specify a threshold for the code coverage in the repository setting. If this threshold is not respected, then Coveralls sends GitHub negative feedback. This is useful especially for PRs, to make sure that the code coverage threshold is respected, before merging a PR. Let's set a threshold for our example repository on Coveralls by clicking on the "Settings" link. Here we specify these settings (note that we can also specify that code coverage must not decrease by a specific percentage):

PULL REQUESTS ALERTS  
(Not available for all CI's)

LEAVE COMMENTS?  **ENABLED**

FORMAT:

USE STATUS API?  **ENABLED**

COVERAGE THRESHOLD FOR FAILURE  %

COVERAGE DECREASE THRESHOLD FOR FAILURE  %

**SAVE CHANGES**

Coveralls coverage settings for pull requests

Now let's experiment with this feature, by creating a new local branch in our repository, e.g., `coverage-experiments`. Let's also push this branch so that Coveralls get to know about the current coverage percentage (100%) of this branch, when the GitHub Actions workflow terminates.

Let's modify the SUT with a parameter and a condition; this condition is intentionally not covered by our tests:

```

1 package com.examples.myproject;
2
3 public class App {
4     public String sayHello(String name) {
5         if (name == null)
6             return "Hello";
7         else
8             return "Hello " + name;
9     }
10
11    public String sayHello() {
12        return sayHello(null);
13    }
14 }
```

Commit on the new branch, push the new branch on GitHub and create a PR. In the build log of the main configuration, we expect to see the uncovered parts detected:

```

1 [INFO] Gathered code coverage metrics for 1 source files with 14 lines of code:
2 [INFO] - 5 relevant lines
3 [INFO] - 4 covered lines
4 [INFO] - 1 missed lines
```

It might take some time before Coveralls sends its feedback to the GitHub PR. When this happens, we should see the negative feedback:



Coveralls notifies the PR that the coverage threshold has been violated

Coveralls detected that the coverage decreased from 100% to 80%.

In the Coveralls settings shown in the screenshot above, we also enabled the detailed comments. However, we see no such detailed comments in the PR. Coveralls must know the PR number to participate in the PR with detailed comments. Currently, the `coveralls-maven-plugin` is not able to retrieve the number of the PR automatically when running within the GitHub Actions workflow. So

we must retrieve it using the variable `github.event.pull_request.number` and pass it to the plugin with the parameter `pullRequest`:<sup>4</sup>

```
1 additional-maven-args: >
2   -Pjacoco
3   -DrepoToken=${COVERALLS_REPO_TOKEN}
4   -DpullRequest=${{ github.event.pull_request.number }}
5   coveralls:report
```

Let's commit and push. After the job of the PR finishes, we can see the detailed comment from Coveralls on the GitHub PR:

The screenshot shows a GitHub pull request page with a detailed coverage report from Coveralls. The report title is "Pull Request Test Coverage Report for Build #43". It includes a summary of changes and coverage statistics, followed by a table of coverage details for a specific file, and a breakdown of totals and changes from a base build.

Changes Missing Coverage	Covered Lines	Changed/Added Lines	%
com.examples.myproject/src/main/java/com/examples/myproject/App.java	3	4	75.0%

Totals	coverage: 80%
Change from base Build #39:	-20.0%
Covered Lines:	4
Relevant Lines:	5

#### Detailed comment from Coveralls

And the links will lead us to the Coveralls code coverage of the file with uncovered lines:

<sup>4</sup>When not building a PR, `github.event.pull_request.number` will not be defined and we pass an empty value to the parameter `pullRequest`, which is safe.

```

1 package com.examples.myproject;
2
3 public class App {
4     public String sayHello(String name) {
5         if (name == null)
6             return "Hello";
7         else
8             return "Hello " + name;
9     }
10
11     public String sayHello() {
12         return sayHello(null);
13     }
14 }

```

### Uncovered parts on Coveralls

The comment will be updated if we push other commits to the PR

Depending on your settings on your Coveralls account, you will also get an email about the violated coverage threshold condition.

Let's add the missing test in AppTest so that we cover the whole SUT:

```

1 @Test
2 public void testSayHelloWithName() {
3     assertEquals("Hello World", app.sayHello("World"));
4 }

```

Commit and push. The PR will be updated with the new commit. When the build ends, Coveralls will send positive feedback to the PR (all checks will now be green), and the comment from Coveralls will be updated accordingly (note the “edited” on the top of the comment):

**Pull Request Test Coverage Report for Build #46**

- 4 of 4 (100.0%) changed or added relevant lines in 1 file are covered.
- No unchanged relevant lines lost coverage.
- Overall coverage remained the same at 100.0%

Totals	coverage 100%
Change from base Build #39:	0.0%
Covered Lines:	5
Relevant Lines:	5

### Detailed comment from Coveralls

Summarizing, the CI process that we setup allows us to keep track of the status of our project: it builds correctly, tests are successful, some code quality criteria (like code coverage) are met. The build tools and other mechanisms are all integrated into this process. In particular, we have access to possible failed tests and we can easily spot parts of our code that are covered by our tests. In Chapter [Code Quality](#) we will further extend this CI process using another tool for keeping track of the code quality of our code.

## 10.8 Badges

Both GitHub Actions 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.



The README.md (or a README in any format, e.g., plain text) should always be present in a GitHub repository to describe the project and to provide some documentation.

The badge for a specific workflow in GitHub Actions can be generated by navigating to the workflow in the “Actions” tab, e.g., in this example, we select the workflow with the name “Java CI with Maven in Linux”. Then, we click on the top right button with “...” and we select “Create status badge”. Here we can also select the branch to associate to the build status badge. We then copy the status badge in Markdown format into the clipboard:

The screenshot shows the 'Create status badge' dialog box. At the top, there's a 'Branch' dropdown set to 'Default branch'. Below it is an 'Event' dropdown set to 'Default'. A text area contains the Markdown code for the badge: `![Java CI with Maven in Linux](https://github.com/LorenzoBettini/example/workflows/Java%20CI%20with%20Maven%20n%20Linux/badge.svg)`. At the bottom is a green button labeled 'Copy status badge Markdown'.

Getting the code for the badge

The Markdown code for the badge will be of the shape

```
1 ! [WORKFLOW_NAME](https://github.com/<OWNER>/<REPOSITORY>/workflows/<WORKFLOW_NAME>/b\\
2 badge.svg)
```

Then, create the `README.md` in the root of the Git repository. We can do it 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 have just copied. We can use the “Preview” tab to see how the `README.md` will be rendered.

We can also modify the Markdown code so that the badge also links to the “Actions” page of the repository:

```
1 [ ! [WORKFLOW_NAME](https://github.com/<OWNER>/<REPOSITORY>/workflows/<WORKFLOW_NAME>/\\
2 badge.svg)](https://github.com/<OWNER>/<REPOSITORY>/actions)
```

In the preview, we can verify that the badge is also clickable and will bring us to the corresponding “Actions” tab.

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 workflow fails the badge is automatically updated showing a failing status.

Similarly, on Coveralls, select “Embed” to get the Markdown code to paste in the `README.md`. The badge of Coveralls is already configured as a web link to the Coveralls web page for the project.

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:



The badges in the README file

## 10.9 Reproducibility in CI

As we saw in Chapter [Maven](#), Section *Reproducibility in the build*, we should aim at reproducibility in build automation. We also saw in that section (and in general in that chapter) how to have a high level of reproducibility. Building in a CI server like GitHub Actions poses additional challenges to reproducibility. Despite the versions of the Maven dependencies and the Maven plugins, our build will rely on the execution environment provided by the CI server.

For this reason, it might be worthwhile to have a look at the list of available environments and the installed software on each environment: <https://github.com/actions/virtual-environments>.

In this example, we are relying on the Maven installed in the virtual environment. This might not be a problem, but if we want to have control on the Maven version, we might want to use the Maven wrapper (Chapter [Maven](#), Section *Maven wrapper*).

In this chapter, we specified only the major version of Java to use in our builds (including using different Java versions). We could be more specific on the version using a full semantic version, e.g., 8.0.232 or ‘11.0.4’. Unless our project heavily relies on some Java internal features, (like reflection, classloaders, etc.) specifying only the major version should be enough. Further details on how to use a specific Java version can be found on the web page of the action: <https://github.com/actions/setup-java>.

Indeed, using the action `setup-java` is not even strictly necessary: the virtual environments provided by GitHub Actions already come with a few versions of Java installed. Depending on the environment (operating system) a specific version of Java is set as the default one.

We can verify that by creating a temporary new branch `environments-experiments` and by creating a brand new workflow `environments-experiments.yml` as simple as this:

```

1  on:
2    push:
3
4  jobs:
5    build:
6      runs-on: ubuntu-latest
7
8      name: Build with the default Java
9      steps:
10     - uses: actions/checkout@v2
11     - name: Show Java version
12       run: java -version
13     - name: Build with Maven
14       run: mvn -f com.examples.myproject/pom.xml clean verify

```

In this workflow, we don't use `setup-java` and we are relying on the default version of Java configured in the environment `ubuntu-latest`. We also show the current version of Java.

By looking at the log, at the time of writing, `ubuntu-latest` corresponds to `ubuntu-18.04` and this the output of printing the Java version:

```

1 openjdk version "1.8.0_275"
2 OpenJDK Runtime Environment (AdoptOpenJDK)(build 1.8.0_275-b01)
3 OpenJDK 64-Bit Server VM (AdoptOpenJDK)(build 25.275-b01, mixed mode)

```

If instead, we specify a specific Ubuntu version

```
1 runs-on: ubuntu-20.04
```

we see that the default Java version is different:

```
1 openjdk version "11.0.9.1" 2020-11-04
2 OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.9.1+1)
3 OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.9.1+1, mixed mode)
```

At some point, `ubuntu-latest` will correspond to `ubuntu-20.04` and thus also our workflow will silently switch accordingly to use a different Java version. This might break our builds.

This new workflow might also be slightly faster because we don't have to set up Java. However, we realize that it is much better to be in full control of the Java version for our builds, at least we should specify the major version of Java. This way, we reduce the chances of future failures by making the build as reproducible as possible.

## 10.10 Let's revise our build process

Now that our CI process has been set up, let's take some time to see whether we can improve our build process. We can do that in a brand new branch that we'll then push and finally merge with a PR.

First of all, do we really need to call the Maven phase `clean` when our project is built in the GitHub Actions virtual environments? We followed the best practice of not storing the `target` folder in the Git repository (see Chapter [Git](#), Section [Ignore files and directories](#)). Moreover, we know that when a job is executed by GitHub Actions it always uses a fresh new virtual environment (in fact, we are caching dependencies, see Section [Caching dependencies](#)). Thus, when the Maven build starts in GitHub Actions, there is nothing to clean! Running the phase `clean` is harmless, but it is also completely useless, and useless things only generate noise and misunderstanding. For this reason, we get rid of the `clean` phase in all our YAML files.

So the Maven command line

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

becomes

```
1 mvn -f com.examples.myproject/pom.xml verify ...
```

In the Git repository for this example, the Maven project is in a subdirectory, `com.examples.myproject`. In fact, we always use the `-f` command line argument when we invoke Maven. Indeed, Maven would search for a POM file in the current directory by default. To avoid the use of `-f`, which leads to a long command, we can use the `working-directory` keyword of `run`, which allows us to specify the working directory where the command will be executed.

For example, instead of

```
1 run: >
2   mvn -f com.examples.myproject/pom.xml verify ...
```

we can write

```
1 run: >
2   mvn verify ...
3 working-directory: com.examples.myproject
```

Note that the command is much simpler now. We could even get rid of the folded syntax `>`. Remember that we need to specify the `working-directory` in every `run:` command where we execute a Maven build:

```
1 - name: Build with Maven
2   run: >
3     mvn verify ${{ matrix.additional-maven-args }}
4   working-directory: com.examples.myproject
5 ...
6 - name: Generate JUnit Report
7   run: >
8     mvn surefire-report:report-only site:site -DgenerateReports=false
9   working-directory: com.examples.myproject
```

The subdirectory of our code is likely not to change, so we just need to make sure it correctly the first time. However, if we want to further enforce consistency, we could declare a job level environment variable:

```
1 jobs:
2   build:
3     runs-on: ubuntu-latest
4     env:
5       workdir: com.examples.myproject
6       ...
```

and then refer to that variable in all `working-directory` specifications:

```
1 - name: Build with Maven
2   run: >
3     mvn verify ${{ matrix.additional-maven-args }}
4   working-directory: ${{ env.workdir }}
5 ...
6 - name: Generate JUnit Report
7   run: >
8     mvn surefire-report:report-only site:site -DgenerateReports=false
9   working-directory: ${{ env.workdir }}
```

Finally, let's focus on the actual Maven command that we execute to send code coverage information to Coveralls. In the end, excluding additional properties like the `repoToken` and `pullRequest`, this is the interesting part:

```
1 mvn verify -Pjacoco coveralls:report
```

What happened if we forgot to enable the `jacoco` profile? That is,

```
1 mvn verify coveralls:report
```

We would get an error from the `coveralls-maven-plugin`:

```
1 I/O operation failed: No coverage report files found
```

This means that our Maven command relies on some required profiles: it would fail if we called the goal `coveralls:report` without enabling the profile `jacoco`. We have already seen such a situation in Chapter [Maven](#), Section [Don't abuse profiles](#).

We want our build to be as reliable as possible, minimizing the chances of build failures. If the goal `coveralls:report` requires a code coverage report generated by JaCoCo we can create a new additional profile in our POM, e.g., `coveralls`, which enables the `jacoco-maven-plugin` (already configured in the `<pluginManagement>` section) and explicitly binds the `coveralls:report` goal to the `verify` phase.<sup>5</sup>

---

<sup>5</sup>We need to specify a phase since the `coveralls:report` goal is not automatically bound to any phase.

```

1 <profile>
2   <id>coveralls</id>
3   <build>
4     <plugins>
5       <plugin>
6         <!-- configured in pluginManagement -->
7         <!-- JaCoCo report is required by coveralls-maven-plugin -->
8         <groupId>org.jacoco</groupId>
9         <artifactId>jacoco-maven-plugin</artifactId>
10        </plugin>
11        <plugin>
12          <!-- configured in pluginManagement -->
13          <groupId>org.eluder.coveralls</groupId>
14          <artifactId>coveralls-maven-plugin</artifactId>
15          <executions>
16            <execution>
17              <phase>verify</phase>
18              <goals>
19                <goal>report</goal>
20              </goals>
21            </execution>
22          </executions>
23        </plugin>
24      </plugins>
25    </build>
26  </profile>

```

Now, our Maven command simply becomes:

```
1 mvn verify -Pcoveralls -DrepoToken=... -DpullRequest=...
```

We do not have to explicitly enable the jacoco profile and then run the coveralls:report goal: we simply enable the coveralls profile and we cannot make mistakes.

The jacoco profile can still be there: we could use it to create a JaCoCo report locally on our computer if we want. But what happens if we enabled both the jacoco profile and the coveralls profile? They both enable jacoco-maven-plugin so would this make the build fail? No: Maven already handles the merging of plugin configurations, and in such a case jacoco-maven-plugin will be enabled only once, still taking the main configuration from the <pluginManagement> section.

We still have some duplication since this part

```
1 <plugin>
2   <groupId>org.jacoco</groupId>
3   <artifactId>jacoco-maven-plugin</artifactId>
4 </plugin>
```

is present both in the jacoco profile and in the coveralls profile. Well, this duplicated part is minimal and it is important to have full control over the requirements of each profile and goal, so that our build changes if we enable some profiles, and will not break if we forget to enable some profiles. It is also much clearer now what each profile does since each profile is now self-contained. Concerning duplication, the important thing is that the main configuration appears only once in the `<pluginManagement>` section.

If we did not want to have a separate profile for the `coveralls-maven-plugin`, then we could improve our build process by clearly separating the main Maven command, which is expected to generate the JaCoCo report using the default lifecycle, from a further Maven command that is only executing the `coveralls:report` goal without any lifecycle phase:

```
1 mvn verify -Pjacoco
2 mvn coveralls:report
```

This the same strategy we are using for generating the JUnit HTML report in a separate step where we only run specific goals without any phase. Of course, this step relies on some data already generated by a previous Maven command, which executes lifecycle phases.

```
1 mvn surefire-report:report-only site:site
```

Mixing the two strategies, as we were doing before revising our build process, is instead error-prone and less clear.

## 10.11 Further steps

Now it should be straightforward to experiment with this CI process with more complex projects. For example, try to put one of the previous examples of the book on GitHub and set up the CI process with GitHub Actions and Coveralls.

Another experiment could be to run mutation tests with PIT only on pull requests, making sure the build fails if there are survived mutants (see Chapter [Maven](#), Section [Configuring the PIT Maven plugin](#)). Unfortunately, in this case, there's no free service for storing reports of mutation tests, thus, one should temporarily store the report of PIT using the mechanism shown in Section [Storing artifacts](#).

Of course, if something fails on GitHub Actions, it might also make sense to run the build locally to better debug the source of the failure (being a compilation failure, a test failure, or a failure due to some report).

Moreover, both PIT HTML reports and JUnit HTML reports could be published on the website of the project, e.g., only when releasing a new version or on other specific events, e.g., when tagging a Git commit. GitHub provides webspace for the GitHub repositories. You may want to have a look at this service, called **GitHub Pages**, <https://pages.github.com/>. There is also a predefined action to publish something on GitHub Pages. Similarly, there is an action to perform a **release** directly on the GitHub website of the project. It is also possible to publish Maven artifacts to Maven Central from GitHub Actions. We leave all these subjects as exercises.

# 11. Docker

When your application needs to communicate with a server, like, e.g., a database, at some point of the development you will have to test your application against such a real server. In the next chapter, we will see this kind of test, called **integration tests**. For unit tests, it is recommended to isolate the SUT as much as possible. Moreover, application parts that rely on a server should be minimized and wrapped into only a few classes, to which your application code delegates, abstracting from the server details and API and keeping the door open to future changes or to an easy reconfiguration with a different server. While most of the application can then be tested in isolation, possibly with mocks, the parts that communicate with the server should be tested, at some point, with a real server.

Installing a server on your development machine is usually a bad idea. It overloads your system with additional configurations and wastes resources. The installation procedure might be automatic but it still requires some input from the user. Such an installation should then be reproduced on every development machine, possibly across team members' computers. To keep the development process reproducible, and avoid typical problems like "it works on my machine", the same server version with the same configurations should be installed on all machines, including the continuous integration server.

To avoid these problems, we could run the server required by our application, at least during development and in the CI server, in a virtualized environment. This way, we configure the virtual machine with the needed server, once and for all, sharing the virtual machine configuration among the team members and in the CI server. When we need the server we start the virtual machine. When we do not need the server anymore, we simply stop the virtual machine.

When using a virtual machine (like, e.g., VirtualBox, <https://www.virtualbox.org/>) you virtualize, install and run a full operating system (including the kernel). This requires lots of space on the host and starting a virtual machine takes a lot of time. Indeed, a complete OS has to boot inside the virtual machine. Moreover, you must allocate some resources (e.g., memory, CPU, etc.) of the host to the virtual machine in advance. Such resources are stolen from your main OS.

**Containers** are an alternative virtualization solution, which performs virtualization at the OS level: all containers run on the same operating system kernel, the one of the host, and are thus much more lightweight than virtual machines. For example, if the host runs Ubuntu, a container can execute RedHat but reusing the host Ubuntu kernel. This means that containers share the resources with the host and since they do not virtualize the kernel, starting a container is usually almost as fast as running a local process. Note however those containers are still isolated from each other and communication among containers has to be explicitly configured, as we will see later.

**Docker** is one of the mainstream programs for containers. Containers are created from **images** that specify their precise contents, e.g., the filesystem, the installed programs, and configurations.



Just like a process is a program in execution, a container is an image in execution. There can be many processes running the same program. There can be many containers executing the same image.

Images are often created by combining and modifying standard images downloaded from public repositories.

Using Docker has several advantages:

- Development and production environments can be easily reproduced;
- Everyone (both developers and final users) will use the same versions of used servers, e.g., MySQL, Apache, etc.;
- We can easily package an application with all dependencies and with the execution environment.

The architecture of Docker can be summarized as follows:

### Docker daemon

A machine (physical or virtual) running the **Docker engine**; this is also known as the “Docker host”.

### Docker client

A program configured to “talk” to the Docker engine.

Typically for development, that is, in our main scenario, the daemon and the client are on the same machine.

The client can build a new image and run an existing image in a container. It does so by sending requests to the Docker daemon. The Docker daemon is responsible for pulling from the Internet images that are not present locally. Downloaded images are then cached locally so that further requests of the same image will not require a further download. This is similar to Maven downloading dependencies and caching them locally. Images are stored in **Docker registries**, which are similar to Maven repositories. The default registry is **Docker Hub**, <https://hub.docker.com/>. Just like Maven artifacts, each image is available with several different versions.

A Docker image is a read-only template. On the contrary, containers executing an image are read-write. Docker uses a **union filesystem** that combines several layers in a stacked form. Thus, layers in an image are read-only and a container adds a read-write layer on top of the stack of the image. The crucial aspect of this layered file system of images is that common layers are reused among images. This is also taken into consideration for storing images locally and for downloading images: common layers are stored only once (saving disk space) and downloaded only once.

Images are identified by their name and **tag**. The tag specifies the version of the image and can contain letters, digits, underscores, periods, and dashes. If not specified, the latest version of the image is downloaded and used.

As we will see throughout the chapter and in the rest of the book, using Docker has several advantages. We list here the main ones for our context:

### Portability

Docker allows you to share the development environment, at least the needed services, with all different infrastructures, from the development team to the CI server, not to mention the final user execution environment.

### Quick and easy

By issuing simple commands, you can start new containers or stop and destroy existing ones. If we do not consider the time it takes to download new images the very first time, starting a container is almost as fast as starting a local process.

### Reproducibility

A Docker container running a specific image will behave the same way on any system that can run Docker. This enforces and implies consistency. Custom images can be built by writing instructions in a text file, the *Dockerfile* (see Section *The Dockerfile*). This file is distributed among team members and on the CI server so that all images built from the same Dockerfile will behave identically. Upgrading an image, that is, the service included in the image, simply consists of updating your *Dockerfile*.

### Isolation

Services running in containers are indeed isolated (“containerized”) so that they do not interfere with other containers’ services. As we will see in this chapter, communications among these services can be made explicit when required.

### Open-source

The code involved in Docker, including the files used to build the images (*Dockerfile*) are open source so that you can customize them to your need and contribute fixes and enhancements.

## 11.1 Let's get started with Docker

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

```
1 docker <command> <arguments>
```

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

```
1 docker <command> --help
```

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

```
1 sudo usermod -aG docker ${USER}
```

After that, you need to logout and login again. Then, you can run Docker with your user.

Containers are executed with the command `run` and the name of the image. Let's run the `hello-world` with Docker

```
1 $ docker run hello-world
2 Unable to find image 'hello-world:latest' locally
3 latest: Pulling from library/hello-world
4 ... feedback of downloading ...
5 Status: Downloaded newer image for hello-world:latest
6
7 Hello from Docker!
8
9 To generate this message, Docker took the following steps:
10 1. The Docker client contacted the Docker daemon.
11 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
12     (amd64)
13 3. The Docker daemon created a new container from that image which runs the
14     executable that produces the output you are currently reading.
15 4. The Docker daemon streamed that output to the Docker client, which sent it
16     to your terminal.
17
18 To try something more ambitious, you can run an Ubuntu container with:
19 $ docker run -it ubuntu bash
20 ...
```

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 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:

```
1 -i, --interactive          Keep STDIN open even if not attached
2 -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. Indeed, 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 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,

```
1 $ docker run --name my-ubuntu-container -it ubuntu /bin/bash
2 root@...:/# touch aFile.txt
3 root@...:/# exit
```

Now, we could try to rerun the very same container specifying its name:

```
1 $ docker run --name my-ubuntu-container -it ubuntu /bin/bash
2 docker: Error response from daemon: Conflict.
3 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 as an existing one. If we intend 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:

```
1 $ docker restart my-ubuntu-container
2 $ docker attach my-ubuntu-container
3 root@....:/# ls aFile.txt
4 aFile.txt
5 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. 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.

For example,

```
1 $ docker run --name myvolume -v /adirectory ubuntu
```

Creates a container with the 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:

```
1 $ 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.

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

```
1 $ 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.)



create the local directory in advance, or it will be created with root as the owner and you won't be able to easily write in there. In any case, if files are created from the container inside that directory they will be owned by root.

The command `docker images` can be used to list all the Docker images already downloaded. Note that the size of the images reported by the command is to be considered "virtual": common layers are shared among images.

## 11.2 Run a server in a container

We will use Docker in the next chapters to run tests that need access to a server. In the next sections and the next chapters, we will need to access a database server but in this chapter, we'll experiment with a web server, **Apache**.

First of all, we need to know the name of the Docker image for Apache. You can see that there are several Docker images on DockerHub for Apache. The default one is `httpd`. In general, the DockerHub page for an image also shows how the image can be used (see for example [https://hub.docker.com/\\_/httpd](https://hub.docker.com/_/httpd)).

It is crucial to understand that when a server is running inside a container, it is completely isolated. This means that from the outside we cannot simply access the server. For example, Apache runs by default on port 80. However, we can access that port only from the container. Of course, we would like to access the server running in the container from our system (e.g., like we do in the next chapters, from our tests).

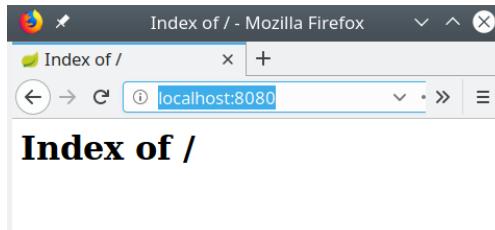
The `run` command provides a specific argument `-p`, `--publish` that publishes a container's port to the host. This is used as follows: `-p <local port>:<container port>`. If the container exposes `<container port>`, it will be visible from the host as `<local port>`. For example with `-p 8080:80` the container's port 80 will corresponds to the host port 8080. The server in the container listening on port 80 will be accessible from the host on 8080.

Let's create a directory, e.g., `myweb` in the current directory. Then we run Apache in a container, mapping the default path where Apache searches for HTML pages (that can be found on the image

documentation site, [https://hub.docker.com/\\_/httpd](https://hub.docker.com/_/httpd)) to our local `myweb` directory (the argument `-d` --detach runs the container in background; `docker run` will return after the container has started after printing the container ID):

```
1 $ docker run -dit --rm --name my-apache-app \
2   -v "$PWD"/myweb:/usr/local/apache2/htdocs/ -p 8080:80 httpd
```

Now, with a web browser let's visit <http://localhost:8080/>: we'll get the answer from Apache running in the container. The directory for HTML contents is empty thus we get an empty list:

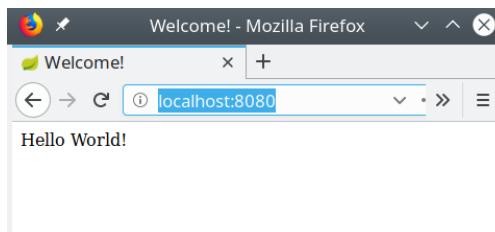


Output from Apache running in a container

Let's create a basic `index.html` inside our local `myweb` directory, e.g.,

```
1 <html>
2   <head>
3     <title>Welcome!</title>
4   </head>
5   <body>
6     Hello World!
7   </body>
8 </html>
```

Reload the page, and now Apache is serving this file:



Our local `index.html` served from Apache running in a container

Note that we did not install Apache locally: we are running it in a container.

Since the container is running in the background, if we need to terminate Apache we must do that with `docker stop` passing the ID that had been printed on the screen or the name we gave to the container.

## 11.3 Dockerize a 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.

### 11.3.1 The Java application to dockerize

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 simple “Hello World”.

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3
4   <groupId>com.examples</groupId>
5   <artifactId>hellodocker</artifactId>
6   <version>0.0.1-SNAPSHOT</version>
7   <packaging>jar</packaging>
8
9   <properties>
10    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
11    <maven.compiler.source>1.8</maven.compiler.source>
12    <maven.compiler.target>1.8</maven.compiler.target>
13  </properties>
14
15  <dependencies>
16    ... JUnit as usual
17  </dependencies>
18</project>
```

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:

```
1 java -cp target/hellodocker-0.0.1-SNAPSHOT.jar com.examples.hellodocker.App
```

### 11.3.2 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 an 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.



Never pass `/` as the context when you build an image: all your local file system contents will be sent to the Docker host!

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 name of the release, e.g., 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.



If you plan to create an image that requires installing additional packages, instead of proceeding directly with a `Dockerfile`, you might want to run the starting image in a container and then experiment with the installation steps until you are sure that all the required packages are installed.

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/helldocker-0.0.1-SNAPSHOT.jar /app/app.jar
11
12 CMD ["java", "-cp", "/app/app.jar", "com.examples.helldocker.App"]
```

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 the 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 command `docker build` 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/helldocker-0.0.1-SNAPSHOT.jar /app/app.jar
16 ...
17 Step 5/5 : CMD ["java", "-cp", "/app/app.jar", "com.examples.helldocker.App"]
18 ...
19 Successfully built ...
20 Successfully tagged java-hello-world:latest
```

We can manually test our image by running

```
1 $ docker run --rm java-hello-world
2 Hello World!
```

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:

```
1 $ docker build -t java-hello-world .
2 ...
3 Step 2/5 : RUN apt-get update && \
4         apt-get install -y openjdk-8-jdk && \
5         apt-get clean
6 ---> Using cache
7 Step 3/5 : RUN java -version
8 ---> Using cache
9 Step 4/5 : COPY /target/helldocker-0.0.1-SNAPSHOT.jar /app/app.jar
10 ---> Using cache
11 ...
```

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

```
1 ...
2 RUN apt-get update && \
3     apt-get install -y openjdk-8-jdk && \
4     apt-get clean
5 ...
```

into

```
1 ...
2 RUN apt-get update && apt-get install -y openjdk-8-jdk && \
3     apt-get clean
4 ...
```

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. 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 (all available tags and versions can be found at the Docker image website for this image, [https://hub.docker.com/\\_/openjdk/](https://hub.docker.com/_/openjdk/)).

Our `Dockerfile` can then be simplified as follows:

```
1 FROM openjdk:8
2
3 COPY /target/helldocker-0.0.1-SNAPSHOT.jar /app/app.jar
4
5 CMD ["java", "-cp", "/app/app.jar", "com.examples.helldocker.App"]
```

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/helldocker-0.0.1-SNAPSHOT.jar /app/app.jar
8 Step 3/3 : CMD ["java", "-cp", "/app/app.jar", "com.examples.helldocker.App"]
9 ...
10 Successfully built ...
11 Successfully tagged java-hello-world:latest

1 $ docker run --rm java-hello-world
2 Hello World!
```

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 command `docker build` we can see that it's less than 30kB). Let's create a `.dockerignore` (in the same directory of the `Dockerfile`) so that the sent contents will consist only of the jar file of the target directory (excluding its recursive contents, i.e., generated `.class` files and other files, like sources):

```
1 # Ignore everything...
2 **
3
4 # ...but the JAR file
5 !/target/*.jar
```

Let's run the build command again and this time we see that the sent contents are only the essential ones (of course, we also verify that we can still run our Docker image):

```
1 $ docker build -t java-hello-world .
2 Sending build context to Docker daemon 6.144kB
3 ...
```

## 11.4 Windows containers

On macOS and Windows, when we run Docker containers we actually run their services on a Linux kernel. Docker Desktop for Windows allows us to switch to **Windows containers** (there's a specific entry in the context menu of the Docker tray icon). This way, we can run services inside a Windows kernel.

This has two main consequences to be aware of:

- if we pull an existing image from DockerHub, that image must exist for the Windows architecture (`windows-amd64`);
- if we build a custom image with a `Dockerfile` (see Section [The Dockerfile](#)) the commands we specify with `RUN` are executed in a Windows shell, in a Windows operating system.

Concerning the first point, DockerHub automatically provides the requested image for the Windows architecture when we enabled Windows containers, as long as an image for such architecture exists. Otherwise, we get a failure.

The images we are using in this chapter for our example, `openjdk` (Section [The Dockerfile](#)) and `mongo` (Section [Docker networks](#)) are available also for the Windows architecture.

The same does not hold for `httpd` (Section [Run a server in a container](#)). If we run this command from Windows, after switching to Windows containers, we get this error:

```
1 C:\> docker pull httpd
2 Unable to find image 'httpd:latest' locally
3 latest: Pulling from library/httpd
4 no matching manifest for windows/amd64 ... in the manifest list entries.
```

We would have the same problem with this command, as we can expect:

```
1 C:\> docker run -it ubuntu bash
2 Unable to find image 'ubuntu:latest' locally
3 latest: Pulling from library/ubuntu
4 no matching manifest for windows/amd64 ... in the manifest list entries.
```

Note that if we try to run the `hello-world` container, which is available for Windows, we succeed, but the output is slightly different from the one we receive when we execute such a container in Linux or macOS. It shows that we are running an image for Windows. We highlight the different lines concerning the ones of Section [Let's get started with Docker](#):

```
1 ...
2 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3   (windows-amd64, nanoserver-1809)
4 ...
5 To try something more ambitious, you can run a Windows Server container with:
6 PS C:\> docker run -it mcr.microsoft.com/windows/servercore powershell
```

In particular, the last sentence suggests us to run a Windows server container with its shell (Powershell), instead of running the Bash shell inside an Ubuntu container.

The `Dockerfile` we are using in this chapter for our Java application works also when using Windows containers: it uses `openjdk` as the base image (which, as said above, exists for Windows

as well) and it does not RUN any specific Linux command. Of course, the initial version of our Dockerfile, where we were running apt commands would not work when using Windows containers.



In general, a Docker image for the Windows architecture is much bigger than the same image for Linux.

On our Windows computer, we are free to switch between Linux and Windows containers as we see fit. However, we should consider the above differences when using the Windows environment of GitHub Actions (Section [Using Docker in GitHub Actions](#)): where only Windows containers can be executed.

## 11.5 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 building the Docker image for our application every time we run the Maven build.

```
1 <profiles>
2   <profile>
3     <id>docker</id>
4     <build>
5       <plugins>
6         ...
```

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 the goal `stop` to the phase `verify` 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>
3   <artifactId>docker-maven-plugin</artifactId>
4   <version>0.34.1</version>
5   <configuration>
6     <showLogs>true</showLogs>
7     <verbose>build</verbose>
8     <images>
9       <image>
10      <name>java-hello-java</name>
11      <build>
12        <contextDir>${project.basedir}</contextDir>
13      </build>
14      <run>
15        <wait>
16          <log>Hello World!</log>
17          <time>10000</time>
18        </wait>
19      </run>
20      </image>
21    </images>
22  </configuration>
23  <executions>
24    <execution>
25      <id>docker-build-image</id>
26      <phase>package</phase>
27      <goals>
28        <goal>build</goal>
29      </goals>
30    </execution>
31    <execution>
32      <id>docker-verify</id>
33      <phase>verify</phase>
34      <goals>
35        <goal>start</goal>
36        <goal>stop</goal>
37      </goals>
38    </execution>
39  </executions>
40 </plugin>
```

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). Using `<verbose>build</verbose>` we require to print out Docker the build instructions.

The binding of goals to phases in the `<executions>` element has already been explained.

Let's now see in detail how the other parts of this plugin are configured. The `<images>` element lists all the Docker images involved in the build. The name of the image is specified by `<name>`. In this example, we only deal with our 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), with `<contextDir>`. (One can specify the `Dockerfile` to use with the argument `dockerFile`; by default, the `Dockerfile` found in the `contextDir` will be used.)

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, 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 from the command line or from Eclipse):

```
1 $ mvn clean verify -Pdocker
2 [INFO] --- docker-maven-plugin:0.34.1:build (docker-build-image) @ hellodocker ---
3 [INFO] Building tar: ...target/docker/java-hello-java/tmp/docker-build.tar
4 [INFO] DOCKER> [java-hello-java:latest]: Created docker-build.tar in 34 milliseconds
5 [INFO] DOCKER> Step 1/4 : FROM openjdk:8
6 [INFO] DOCKER> Step 2/4 : ARG jarToCopy
7 [INFO] DOCKER> Step 3/4 : COPY /target/$jarToCopy /app/app.jar
8 [INFO] DOCKER> Step 4/4 : CMD ["java", "-cp", "/app/app.jar", "com.examples.hellodoc\
9 ker.App"]
10 [INFO] DOCKER> Successfully built ...
11 [INFO] DOCKER> Successfully tagged java-hello-java:latest
12 [INFO] DOCKER> [java-hello-java:latest]: Built image sha256:...
13 [INFO]
14 [INFO] --- docker-maven-plugin:0.34.1:start (docker-verify) @ hellodocker ---
15 [INFO] DOCKER> [java-hello-java:latest]: Start container ab20b605bef8
16 [INFO] DOCKER> Pattern 'Hello World!' matched for container ab20b605bef8
17 ab20b6> Hello World!
18 [INFO] DOCKER> [java-hello-java:latest]: Waited on log out 'Hello World!' 509 ms
19 [INFO]
20 [INFO] --- docker-maven-plugin:0.34.1:stop (docker-verify) @ hellodocker ---
21 [INFO] DOCKER> [java-hello-java:latest]: Stop and removed container ... after 0 ms
```

We can see that the image is built (goal `build`) during the phase `package`. If this is the first time we build our Docker image, the base image `openjdk:8` will also be downloaded. During the phase `verify` 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. In the same phase, after the goal `start`, the goal `stop` is also executed: the container is stopped and removed.

When the plugin builds the Docker image, all the files and directories located in the `contextDir` directory will be added to the build context. In fact, this plugin does not consider the `.dockerignore`. You can have a look at the `target` subdirectory where the plugin stores the files for the build:

```
1 target/docker
2   └── build.timestamp
3     └── java-hello-java
4       ├── build
5         └── Dockerfile
6       └── tmp
7         └── docker-build.tar
8       └── 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 `contextDir` (that is, the project base directory), the file `.maven-dockerinclude`, listing only the files for the context build:

```
1 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:

```
1 COPY /target/helldocker-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>`:

```
1 FROM openjdk:8
2
3 ARG jarToCopy
4
5 COPY /target/$jarToCopy /app/app.jar
6
7 CMD ["java", "-cp", "/app/app.jar", "com.examples.helldocker.App"]
```

Now, when we build the image we must pass a value for `jarToCopy`:

```
1 docker build --build-arg jarToCopy=helldocker-0.0.1-SNAPSHOT.jar \
2     -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   <contextDir>${project.basedir}</contextDir>
4   <args>
5     <jarToCopy>${project.artifactId}-${project.version}.jar</jarToCopy>
6   </args>
7 </build>
8 ...
```

If we changed the project version or even the artifact identifier our Dockerfile would still be valid.

## 11.6 Using Docker in GitHub Actions

Let us now see how to build this project, which uses Docker, in a GitHub Actions workflow. Docker is already available in the Linux virtual environments provided by GitHub Actions.

As seen in Chapter [Continuous Integration](#), Section [GitHub Actions](#), we create a YAML file, e.g., `maven.yml`, in the directory `.github/workflows` of our Git repository. For this project we specify

```
1 name: Java CI with Maven and Docker in Linux
2
3 on:
4   push:
5     pull_request:
6
7 jobs:
8   build:
9     runs-on: ubuntu-latest
10
11   name: Build in Linux
12   steps:
13     - uses: actions/checkout@v2
14     - name: Set up JDK 8
15       uses: actions/setup-java@v1
16       with:
17         java-version: 1.8
18     - name: Cache Maven packages
19       uses: actions/cache@v2
20       with:
21         path: ~/.m2
22         key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml', '**/*.yml') }}
23         restore-keys: ${{ runner.os }}-m2-
24     - name: Build with Maven
25       run: mvn verify -Pdocker
26       working-directory: com.examples.helldocker
```

The contents of the YAML should be straightforward to understand after reading the Chapter *Continuous Integration*.



Note that we do not cache Docker images. Docker images are pulled quickly in GitHub Actions virtual environments (e.g., the `openjdk:8` base image is pulled in a few seconds), so for the moment we do not need to cache the Docker images.

The macOS virtual environment does not have Docker installed. It is possible to install Docker on the virtual environment before running the Maven build, but it is a cumbersome procedure. Fortunately, there is an action for that:

<https://github.com/marketplace/actions/setup-docker>. Such an action installs Docker: it downloads Docker for macOS, configures it, starts it, and waits for it to be up and running.

This is the YAML file for macOS (e.g., named `maven-mac.yml`), where only the interesting parts are shown (Maven caching is not shown):

```
1 ...
2 jobs:
3   build:
4     runs-on: macos-latest
5
6     name: Build in macOS
7     steps:
8     - uses: actions/checkout@v2
9     - name: Set up JDK 8
10    uses: actions/setup-java@v1
11    with:
12      java-version: 1.8
13    - name: Install Docker
14      uses: docker-practice/actions-setup-docker@master
15    - name: Build with Maven
16      run: mvn verify -Pdocker
17      working-directory: com.examples.helldocker
```

Unfortunately, only the above-mentioned procedure for Docker takes about 10 minutes. This time is required each time we run such workflow (besides the time for the other steps). Thus, running this workflow on macOS should be done only on rare occasions, and we should ask ourselves whether we need to run this workflow also on macOS.

The Windows virtual environment already provides Docker installed. Unfortunately, the standard configuration of Docker in Windows raises a few problems when used together with the `docker-maven-plugin`. In fact, by default, on Windows the Docker daemon is accessible through a *named pipe* instead of a *socket* (as in Linux and macOS). When the `docker-maven-plugin` goals are executed such pipe instances quickly become busy making the goals fail. For example, `build` might succeed, but `start` fails with errors of the shape:

```
1 Error: DOCKER> Cannot create docker access object [\\.\\pipe\\docker_engine (All pip\
2 e instances are busy)]
```

This might happen also when using Docker on your Windows computer. Typically, the problem is solved on a local computer by accessing the General Settings dialog of Docker for Windows and by enabling “Expose daemon on `tcp://localhost:2375...`”. On GitHub Actions, it is not that easy to do that, since we have to do that in a headless way and from the command line.

Thus, we first have to issue this shell command (the `-H` argument corresponds to the above setting in the dialog)

```
1 sc config docker binpath="dockerd.exe --run-service -H tcp://localhost:2375"
```

and then restart the Docker service:

```
1 restart-service *docker*
```

Then, we have to tell the Docker client and the docker-maven-plugin that the Docker daemon is listening on that TCP address. This can be done by setting the standard Docker environment variable DOCKER\_HOST (which is read by docker-maven-plugin as well).

Summarizing, this is the YAML file for Windows (e.g., named maven-windows.yml), where only the interesting parts are shown (Maven caching is not shown):

```
1 ...
2 jobs:
3   build:
4     runs-on: windows-latest
5     env:
6       DOCKER_HOST: tcp://localhost:2375
7
8     name: Build in Windows
9     steps:
10    - uses: actions/checkout@v2
11    - name: Set up JDK 8
12      uses: actions/setup-java@v1
13      with:
14        java-version: 1.8
15    - name: Docker config
16      run: >
17        sc config docker
18        binpath="dockerd.exe --run-service -H tcp://localhost:2375"
19    - name: Docker restart
20      run: restart-service *docker*
21    - name: Docker info
22      run: docker info
23    - name: Build with Maven
24      run: mvn verify -Pdocker
25      working-directory: com.examples.helldocker
```

Note that the step running docker info is just for debugging purposes, to verify that we can access the Docker daemon through the specified TCP address.



As anticipated in Section [Windows containers](#), the Docker program in the Windows environment of GitHub Actions can only run Windows containers.

This Windows workflow is faster than the macOS one, but it is slower than the Linux workflow. Thus, also the Windows workflow might be executed only on rare occasions.

We show the time it takes to execute the workflow on the three OSes (on a new branch `other-oses`) in the following screenshot: 1 minute in Linux, 3 minutes in Windows, and 15 minutes in macOS.

The screenshot shows three GitHub Actions workflow runs for Java CI with Maven and Docker. Each run has a green checkmark icon and the text "Also build on macOS and Windows". The first run is for Linux, showing a status of "other-oses" and a duration of 54s. The second run is for Windows, showing a status of "other-oses" and a duration of 2m 56s. The third run is for macOS, showing a status of "other-oses" and a duration of 15m 0s. All runs were pushed by LorenzoBettini.

GitHub Actions workflows on the 3 OSes

As suggested above, we might want to enable the workflows on macOS and Windows only on PR, for example.

## 11.7 Docker networks

In this section, we see how different containers can interact with each other.

As stated in the official Docker documentation, [https://docs.docker.com/config/containers/multi-service\\_container/](https://docs.docker.com/config/containers/multi-service_container/), a container should execute only one service. This means that the service running in a container can fork into multiple processes. Thus, a container running Apache is allowed to have several forked Apache processes. However, it should not run another service, e.g., a database.

Let's say we have a Java application that uses MongoDB, <https://www.mongodb.com>.<sup>1</sup> Such an application will have to interact with a MongoDB server. We can run MongoDB in a Docker container. If we have another container running our Java application, the two containers will have to interact. We will see that this interaction requires further configuration.

First of all, let's start a new Maven project. For this example, we will use

```
1 <groupId>com.examples</groupId>
2 <artifactId>hellodockernet</artifactId>
```

We need the `mongo-java-driver` dependency to interact with a MongoDB from Java:

---

<sup>1</sup>We will use MongoDB again in Chapter [Integration tests](#), where we will provide more details on this database.

```
1 <dependency>
2   <groupId>org.mongodb</groupId>
3   <artifactId>mongo-java-driver</artifactId>
4   <version>3.12.7</version>
5 </dependency>
```

The Java application simply connects to MongoDB, writes a document, and then reads it back printing the contents on the screen. The host of MongoDB can be passed as the first command-line argument and it defaults to “localhost”:

```
1 package com.examples.helldockernet;
2
3 import org.bson.Document;
4
5 import com.mongodb.MongoClient;
6 import com.mongodb.client.MongoCollection;
7 import com.mongodb.client.MongoDatabase;
8
9 /**
10  * Simple app accessing MongoDB.
11 */
12 public class SimpleMongoApp {
13     public static void main(String[] args) {
14         String mongoHost = "localhost";
15         if (args.length > 0)
16             mongoHost = args[0];
17         // default port for MongoDB is 27017
18         MongoClient mongoClient = new MongoClient(mongoHost);
19         MongoDatabase db = mongoClient.getDatabase("mydb");
20         MongoCollection<Document> collection = db.getCollection("examples");
21         Document doc = new Document("name", "Greeting")
22             .append("type", "Hello World!");
23         collection.insertOne(doc);
24         // should print "Hello World!"
25         System.out.println(collection.find().first().get("type"));
26         mongoClient.close();
27     }
28 }
```

For this very example, this is enough. In the next chapter, we will see a more complex example of a Java application that interacts with MongoDB, and, in particular, we will see how to use Docker to write integration tests for our application.

As expected, if we try to run this Java application, we see exceptions on the console since it cannot connect to any MongoDB server. Note that due to the way the Java Mongo driver API is designed, connection failures do not make the Java code exit immediately: we have to wait for a timeout exception. We will get back to this in Section [Control startup order of container](#).

Let's run MongoDB in a Docker container, publishing to the host the default MongoDB port with -p 27017:27017:

```
1 docker run -p 27017:27017 --rm mongo:4.4.3
```

Now the Java application correctly connects and finally prints “Hello World!” on the screen, together with some logging from the Mongo driver.

As seen in Chapter [Maven](#), Section [Creating a FatJar](#), we configure the POM in order to create a FatJar for our Java application, so that dependencies like the Mongo Java driver are packaged in the JAR, together with the information about the main class:

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-assembly-plugin</artifactId>
4   <executions>
5     <execution>
6       <phase>package</phase>
7       <goals>
8         <goal>single</goal>
9       </goals>
10      <configuration>
11        <descriptorRefs>
12          <descriptorRef>jar-with-dependencies</descriptorRef>
13        </descriptorRefs>
14        <archive>
15          <manifest>
16            <mainClass>com.examples.helldockernet.SimpleMongoApp</mainClass>
17          </manifest>
18        </archive>
19      </configuration>
20    </execution>
21  </executions>
22</plugin>
```

Let's run the Maven build (phase package) to create our FatJar.

Finally, we define the Dockerfile as follows (note that we use the wildcard for the FatJar, without ARG as in the previous example; moreover, we run the JAR directly with - jar; .dockerignore can be adapted accordingly):

```
1 FROM openjdk:8
2
3 COPY /target/*-jar-with-dependencies.jar /app/app.jar
4
5 CMD ["java", "-jar", "/app/app.jar"]
```

Let's build the Docker image for our Java application using another name `java-mongo-hello-world`:

```
1 $ docker build -t java-mongo-hello-world .
```

Now let's run the new image, making sure the MongoDB container is still running:

```
1 $ docker run --rm java-mongo-hello-world
2 ...
3 Exception in monitor thread while connecting to server localhost:27017
4 com.mongodb.MongoSocketOpenException: Exception opening socket
```

Thus, the container executing our Java application image cannot communicate with the MongoDB server running inside the other container. Even if we published the MongoDB container port to our host, other containers will not be able to use such published ports.

Indeed, we have to create a **Docker network** to allow two containers to communicate.

The first step is to create the network giving it a name, e.g., `mynetwork`:

```
1 $ docker network create mynetwork
```

Then we run a new MongoDB container with the option `--net` specifying the name of the Docker network we've just created. It is crucial, for what we want to achieve, to give a name to the MongoDB container (e.g., `mongodb`):

```
1 $ docker run --net=mynetwork -p 27017:27017 --rm --name=mongodb mongo:4.4.3
```

By running a container using a defined network, the container will enjoy automatic name resolution using DNS. The name of a container running in this network is mapped to the container's IP address in the network. For example, `mongodb` is the DNS name of the MongoDB container and it is mapped to the IP address of the container in this network. Other containers in this network can use `mongodb` to connect to the container.

To better understand this, let's run our Docker image with an attached terminal, overriding the default command with the Bash shell, in the same network. Once in the container, let's run the command `ping mongodb` and verify that `mongodb` is resolved to the IP address of the MongoDB container (of course, the IP address might differ in your system):

```
1 $ docker run -it --rm --net=mynetwork java-mongo-hello-world /bin/bash
2 root@...:/# ping mongodb
3 PING mongodb (172.23.0.2): 56 data bytes
4 ...
```

Thus, to make our application Docker container communicate with the MongoDB container we only have to pass our Java application the command argument `mongodb` as the first command-line argument. Recall that our Java `main` uses the first command-line argument, if passed, to connect to the MongoDB host. So let's change the `CMD` command in our `Dockerfile`:

```
1 CMD ["java", "-jar", "/app/app.jar", "mongodb"]
```

Rebuild the Docker image and run the container specifying the network:

```
1 $ docker run -it --rm --net=mynetwork java-mongo-hello-world
2 ...
3 Hello World!
```

This time, our Java application will be able to connect to the MongoDB container and we can see “Hello World!” printed in the container.

If we are not interested in accessing the MongoDB also from the host computer, we can avoid publishing the MongoDB container's ports and the container of our Java application will still be able to access MongoDB since it accesses it through the Docker network we created. Let's run MongoDB as follows:

```
1 $ docker run --net=mynetwork --rm --name=mongodb mongo:4.4.3
```

Then let's run our Docker image and verify that everything still works.

Now let's configure the Docker Maven plugin for this example: we want to build our Docker image and then, during the `verify` phase, we start both our container and the MongoDB container on the same network, and make sure that “Hello World!” appears on the log (as usual, this is done in a specific profile `docker`):

```
1 <plugin>
2   <groupId>io.fabric8</groupId>
3   <artifactId>docker-maven-plugin</artifactId>
4   <version>0.34.1</version>
5   <configuration>
6     <!-- Docker custom networks mentioned in the run image configurations
7         will be created automatically during "start" and removed during "stop". -->
8     <autoCreateCustomNetworks>true</autoCreateCustomNetworks>
9     <images>
10    <image>
11      <name>java-mongo-hello-java</name>
12      <build>
13        <contextDir>${project.basedir}</contextDir>
14      </build>
15      <run>
16        <network>
17          <mode>custom</mode>
18          <name>my-network</name>
19        </network>
20        <wait>
21          <log>Hello World!</log>
22          <time>10000</time>
23        </wait>
24        <dependsOn>
25          <container>mongo</container>
26        </dependsOn>
27        </run>
28      </image>
29      <image>
30        <name>mongo:4.4.3</name>
31        <alias>mongo</alias>
32        <run>
33          <network>
34            <mode>custom</mode>
35            <name>my-network</name>
36            <alias>mongodb</alias>
37          </network>
38          <wait>
39            <log>Waiting for connections</log>
40            <time>10000</time>
41          </wait>
42          </run>
43      </image>
```

```

44      </images>
45      </configuration>
46      <executions>
47          ... as in the previous example
48      </executions>
49  </plugin>

```

This configuration reflects the manual steps we performed from the command line. The `<network>` element specifies that the container(s) must be run in the custom network `my-network` (we don't reuse `mynetwork` to avoid conflicts). In particular, we must explicitly specify the `<alias>` in the MongoDB `<network>` configuration so that `mongodb` resolves to the IP of the MongoDB container in this custom network (from the command line we simply gave the container an explicit name `mongodb`). Note that we wait for the string "Waiting for connections" to appear in the log of the MongoDB container, to make sure that the MongoDB server is ready for accepting connections. (We will get back to this also in Section [Control startup order of container](#).) Also, the `<run>` element of our image refers to the custom network. Moreover, with `<dependsOn>` we tell the plugin that our container depends on the container running the image with alias `mongo` (specified in the `<alias>` element after the image `mongo:4.4.3`, not to be confused the `<alias>` in the `<network>` configuration). With `<dependsOn>` the goal `start` will ensure that all dependencies a container depends on are completely started (including all `<wait>` conditions) before the depending container is started.

Let's run the Maven build for the `docker` profile and verify that everything works:

```

1 $ mvn clean verify -Pdocker
2 [INFO] --- docker-maven-plugin:0.34.1:build (docker:build) @ hellodockernet ---
3 [INFO] DOCKER> [java-mongo-hello-java:latest]: Built image sha256:...
4 [INFO]
5 [INFO] --- docker-maven-plugin:0.34.1:start (docker:start) @ hellodockernet ---
6 [INFO] DOCKER> [mongo:4.4.3] "mongo": Start container ...
7 [INFO] DOCKER> Pattern 'Waiting for connections' matched
8 [INFO] DOCKER> [mongo:4.4.3] "mongo": Waited on log out 'Waiting for connections'
9 [INFO] DOCKER> [java-mongo-hello-java:latest]: Start container ...
10 [INFO] DOCKER> Pattern 'Hello World!' matched for container ...
11 [INFO] DOCKER> [java-mongo-hello-java:latest]: Waited on log out 'Hello World!'
12 [INFO]
13 [INFO] --- docker-maven-plugin:0.34.1:stop (docker:start) @ hellodockernet ---
14 [INFO] DOCKER> [java-mongo-hello-java:latest]: Stop and removed container ...
15 [INFO] DOCKER> [mongo:4.4.3] "mongo": Stop and removed container ...

```

Note the "mongo" that corresponds to the `<alias>` specified in the POM. In the configuration above, we wait for a specific string ("Waiting for connections") to appear in the log of the MongoDB container. Such a string highly depends on the specific version of the MongoDB image we are using. This solution works with version 4.4.3. In other versions, it might be different.<sup>2</sup> We know that to

---

<sup>2</sup>For example, in version 4.2.3 it was "waiting for connections on port" (note the different case of the first letter).

make our builds reproducible we must specify explicit versions of our dependencies and this holds also for Docker (see also Section *Reproducibility in Docker*).

In any case, we could make our waiting check less dependent on such low details that depend on the version of a server like MongoDB. For example, we could query the MongoDB port with the standard HTTP protocol. If we run our MongoDB container as we have already done

```
1 docker run -p 27017:27017 --rm mongo:4.4.3
```

we can then visit the URL <http://localhost:27017> with a browser (which by default uses the HTTP “GET” method) and we should have a positive response (i.e., the response code “200”), with a message saying “It looks like you are trying to access MongoDB over HTTP on the native driver port.” Alternatively, we could do this check from the command line with the program *curl*<sup>3</sup> (<https://curl.se/>, a tool for transferring data using several network protocols, including HTTP) and we get:

```
1 curl -i http://localhost:27017
2 HTTP/1.0 200 OK
3 Connection: close
4 Content-Type: text/plain
5 Content-Length: 85
```

Of course, this strategy requires mapping the MongoDB port in the container to a port on the local computer. Then, we can change our wait condition in the POM (we still use the timeout):

```
1 <image>
2   <name>mongo:4.4.3</name>
3   ...
4     <ports>
5       <port>27017:27017</port>
6     </ports>
7     <wait>
8       <http>
9         <url>http://localhost:27017</url>
10        <method>GET</method>
11        <status>200</status>
12      </http>
13      <time>10000</time>
14    </wait>
15  </run>
16 </image>
```

We could also map the container’s port to a random port in the local computer, using a mechanism provided by the Maven plugin:

---

<sup>3</sup>You might have to install that on your computer if you want to perform this experiment yourself. You don’t need it for the POM configuration we will show in the following.

```
1 <ports>
2   <port>mongo.port:27017</port>
3 </ports>
4 <wait>
5   <http>
6     <url>http://localhost:${mongo.port}</url>
7     <method>GET</method>
8     <status>200</status>
9   </http>
10  <time>10000</time>
11 </wait>
```

Instead of hard-coding the local port, we can specify the name of a (possibly undefined) Maven property, e.g., `mongo.port`. If the property is undefined (like in this example) when the `start` goal executes, a free random port will be dynamically selected by Docker and its value will be assigned to the property. The property can then be used later in the same POM.

If we start the Maven build we should in the log something like that (where the port 32772 will be different in further Maven builds):

```
1 [INFO] --- docker-maven-plugin:0.34.1:start (docker-verify) @ hellodockernet ---
2 [INFO] DOCKER> [mongo:4.4.3] "mongo": Start container ...
3 [INFO] DOCKER> [mongo:4.4.3] "mongo": Waiting on url http://localhost:32772
4 with method GET for status 200.
5 [INFO] DOCKER> [mongo:4.4.3] "mongo": Waited on url http://localhost:32772
```

This waiting solution forces us to map the container's port to a host port, just to check whether MongoDB is ready for connections. If we intend to make the MongoDB port exposed to the running host then there is no problem. However, in this example, we only want to make the container of our application communicate with the container of MongoDB. So, mapping the MongoDB port is needed only for the `<wait>` check.

To circumvent this problem, we can use another waiting mechanism provided by the Maven plugin:

```
1 <image>
2   <name>mongo:4.4.3</name>
3   ... no port is mapped
4   <wait>
5     <tcp>
6       <ports>
7         <port>27017</port>
8       </ports>
9     </tcp>
10    <time>10000</time>
```

```
11    </wait>
12    </run>
13  </image>
```

This way, the plugin will try to directly access the specified port of the container. Indeed, we don't need to map its port. This is the log of the Maven build (the IP 192.168.144.2 of the container will be different in further Maven builds):

```
1 INFO] --- docker-maven-plugin:0.34.1:start (docker-verify) @ hellodockernet ---
2 [INFO] DOCKER> [mongo:4.4.3] "mongo": Start container ed78dd291bfc
3 [INFO] DOCKER> [mongo:4.4.3] "mongo": Network mode: my-network
4 [INFO] DOCKER> [mongo:4.4.3] "mongo": Waiting for ports [27017]
5 directly on container with IP (192.168.144.2).
6 [INFO] DOCKER> [mongo:4.4.3] "mongo": Waited on tcp port '[/192.168.144.2:27017]'
```



Unfortunately, this last technique does not work on macOS. If we want our build to succeed also on macOS we have to revert to the previous mapped port solution for the `<wait>`.

## 11.8 Docker compose

If we want our Docker image to be used by others, we must specify in the documentation (e.g., on Docker Hub) all the steps to execute to make it work (create docker network, run MongoDB container, etc.). To make things easier, we can use **Docker Compose** that allows the users to run several containers and connect them ("orchestrating containers").



In Linux, Docker Compose is not part of the Docker package and must be installed as a separate program (see <https://docs.docker.com/compose/install/>).

The orchestration is specified in the file `docker-compose.yml` (using the YAML language, that we've already used for GitHub Actions workflows).

For our example, the file is:

```
1 version: '2'  
2  
3 services:  
4   app:  
5     image: java-mongo-hello-java  
6     networks:  
7       - my-network  
8     depends_on:  
9       - mongodb  
10    mongodb:  
11      image: mongo:4.4.3  
12      networks:  
13        - my-network  
14  
15 networks:  
16   my-network:  
17     driver: bridge
```

The details of a Docker Compose file can be found here <https://docs.docker.com/compose/>. The file for our example should be clear enough since it specifies the network configurations that we have already seen. This will start two containers: `app` based on our Docker image `java-mongo-hello-java` and `mongodb` based on the MongoDB image. Docker Container will automatically create a docker network (the name of the network is based on the name of the directory where the file is). Custom networks must be declared and explicitly used in the containers' configurations. Moreover, we specify that our container depends on the `mongodb` container. Note that `mongodb` is also the DNS name in the Docker network of the MongoDB service, which our Java application Docker image relies on. Since the image `java-mongo-hello-java` is not available on Docker Hub (we haven't published it), this file assumes that that image has already been built locally.



It is also possible to build an image from Docker Compose, with `build:` before starting the image. We refer to the official documentation for that.

This file `docker-compose.yml` can be placed in any directory of the hard disk, but it makes sense to store it in the base directory of our project.

From that directory, we run this command

```
1 docker-compose up
```

You should see the output of both containers, prefixed by their names. Finally, our container should output the "Hello World!" string.

Note that Docker Compose will not terminate until there is at least a running container. The MongoDB will keep on running, so we must explicitly terminate the program, e.g., with **Ctrl + C**, or, from another shell from the same directory with `docker-compose stop`.

Thus, with Docker Compose, we can simply distribute the `docker-compose.yml` and all the services of our application will be started and configured to talk to each other (of course, it makes sense to distribute such a file once we also released our Java application Docker image).

The Docker Maven plugin supports Docker Compose as well; we must specify an `<image>` element of the shape:

```
1 <image>
2   <alias>app</alias>
3   <external>
4     <type>compose</type>
5     <basedir>${project.basedir}</basedir>
6   </external>
7   <run>
8     <wait>
9       <log>Hello World!</log>
10      <time>10000</time>
11    </wait>
12  </run>
13 </image>
```

Note the use of the `<external>` type `compose` and the `<basedir>` where to find the compose file. The `<wait>` configuration is applied to the container in the compose file with the specified `<alias>` (recall that the container of our application is called `app` in the `docker-compose.yml` above).

If we now run the Maven build, we should see also the output related to our container specified in the compose file (besides our container started in the previous configurations):

```
1 [INFO] --- docker-maven-plugin:0.34.1:start (docker-verify) @ hellodockernet ---
2 [INFO] DOCKER> [mongo:4.4.3] "mongo": Start container ...
3 [INFO] DOCKER> [mongo:4.4.3] "mongo": Waiting on url http://localhost:32775
4 with method GET for status 200.
5 [INFO] DOCKER> [mongo:4.4.3] "mongo": Waited on url http://localhost:32775
6 [INFO] DOCKER> [mongo:4.4.3] "mongodb": Start container ...
7 [INFO] DOCKER> [java-mongo-hello-java:latest]: Start container ...
8 [INFO] DOCKER> Pattern 'Hello World!' matched for container ...
9 [INFO] DOCKER> [java-mongo-hello-java:latest]: Waited on log out 'Hello World!'
10 [INFO] DOCKER> [java-mongo-hello-java:latest] "app": Start container ...
11 [INFO] DOCKER> Pattern 'Hello World!' matched for container ...
12 [INFO] DOCKER> [java-mongo-hello-java:latest] "app": Waited on log out 'Hello World!'
```

```
13 [INFO]
14 [INFO] --- docker-maven-plugin:0.34.1:stop (docker-verify) @ hellodockernet ---
15 [INFO] DOCKER> [java-mongo-hello-java:latest] "app": Stop and removed container
16 [INFO] DOCKER> [java-mongo-hello-java:latest]: Stop and removed container
17 [INFO] DOCKER> [mongo:4.4.3] "mongodb": Stop and removed container
18 [INFO] DOCKER> [mongo:4.4.3] "mongo": Stop and removed container
```

In the log above “mongo” refers to the MongoDB container configured with the `<alias>` in the `<image>` section in the POM and started by the Maven plugin’s goal `start`. Indeed, we can see the `<wait>` condition applied to “mongo”. On the contrary, “mongodb” is the MongoDB container started from Docker Compose (see `mongodb`: in the `docker-compose.yml` above).



During this build, we will have two containers running MongoDB, each one on the same port 27017. Why isn’t there any port conflict? The two MongoDB servers are running in isolated containers in different networks and no port is mapped on the same port on the localhost. Thus, we have no conflict.

It might make sense to run this plugin configuration for Docker compose in a separate Maven profile, e.g., `docker-compose` to be run separately so that it is easier to understand the output of the build. However, keep in mind that in the current version the `docker-compose.yml` relies on the image of our application to be already built. Since we should keep Maven profiles independent from each other (see also Chapter [Maven](#), Section [Don’t abuse profiles](#)), if we want to have such a new profile, we should either build the image of our application in `docker-compose.yml` (as hinted above), or refactor our POM and the configuration of the `docker-maven-plugin`. This is left as an exercise.



Docker compose is available in GitHub Actions, thus we do not need to adjust our workflows.

We will use Docker compose again in Chapter [Code Quality](#), Section [Using SonarQube locally](#).

### 11.8.1 Control startup order of container

Let’s take some time to further understand how containers are started and how to make sure that containers are ready to cooperate.

In Section [Docker compose](#), we specify in the `docker-compose.yml` file that our `app`’s container `depends_on: mongodb`. This way, Docker Compose will start and stop containers in dependency order according to `depends_on`: The MongoDB container will be started before our Java application container. However, Docker Compose does not wait until a container is “ready”. Indeed, the concept of “readiness” cannot be generalized and highly depends on the service running in a container.

Does this mean that when our Java application’s container is started the MongoDB server might not yet be ready to accept connections? The answer is yes.

Depending on the speed of our computer, when running `docker-compose up` we might note in the log these lines (where `app_1` identifies log from our application and `mongodb_1` identifies log from the MongoDB server):<sup>4</sup>

```
1 mongodb_1 | ...Automatically disabling TLS 1.0, ...
2 ...
3 app_1      | INFO: Exception in monitor thread while connecting
4           to server mongodb:27017
5 app_1      | com.mongodb.MongoSocketOpenException: Exception opening socket
6 ...
7 app_1      | INFO: Cluster description not yet available. Waiting for 30000 ms
8           before timing out
9 ...
10 mongodb_1 | ..."Waiting for connections"...
11 ...
12 app_1      | INFO: Opened connection to mongodb:27017
13 app_1      | Hello World!
14 ...
```

This log tells us a few things:

- The MongoDB container is effectively started before our application’s container, but not before MongoDB waits for connections;
- The Mongo Java driver used by our application throws an exception because it cannot connect to MongoDB;
- However, this does not result in stopping our Java code: the Mongo Java driver is willing to try to connect again periodically, waiting at most 30 seconds before effectively failing (we have already anticipated that in Section [Docker networks](#));
- The MongoDB becomes ready for connections;
- Our Java application succeeds in accessing MongoDB and successfully terminates.

Indeed, in Section [Docker networks](#), when running the containers with the `docker-maven-plugin` we did not only specify `<dependsOn><container>mongo...</container>`: we also specified a wait condition on the MongoDB container (first, by waiting for the string “Waiting for connections” to appear in its log, and then by trying to connect to its port).

In version “2” of the syntax of Docker Compose, which is the one we are using in this example, there is no straightforward way to express such waiting conditions. Some more powerful mechanisms have been introduced in further versions, such as `healthcheck`, which, however, are not yet supported by the `docker-maven-plugin`.

Apparently, we do not need such checks. As seen above, the Mongo Java driver is already developed so that it attempts to re-establish a connection to the database after a failure within a specific timeout.

---

<sup>4</sup>Remember that this is not deterministic: it might happen on your computer always, sometimes, or it might not happen at all.

You might try and remove the `<wait>` element from the configuration of the MongoDB image from the POM (including the possibly mapped port). You can verify that the build succeeds anyway (of course, as long as the computer is fast enough to let MongoDB be ready for connections before the timeout of the Mongo Java driver elapses).

However, we might not always be that lucky. For example, if we used a SQL database and the corresponding Java libraries, such as JPA and Hibernate, we would get an immediate failure upon connection to the database if the database server is not ready. In such a case our Java application would terminate immediately.

For this reason, it might be worthwhile to spend some time learning a few possible techniques to deal with such situations, even if, as said above, in this example with MongoDB it would be redundant.

We have already seen how to specify a `<wait>` condition with the `docker-maven-plugin` in the configuration of the database container. As an alternative, we could remove such a condition and implement a waiting mechanism in the container of our Java application.

There are already a few utilities for this task. One of them is `wait-for-it`, <https://github.com/vishnubob/wait-for-it>, a BASH script for waiting on the availability of a port of a host. Thus, it is ideal for “synchronizing” the startup of interdependent Docker containers. This script only requires that BASH is available in the running container, which is the case for the `openjdk` base image that we use in the container of our Java application.

The script can be used like that:

```
1 ./wait-for-it.sh <host:port> --timeout=30 --strict -- <the application command>
```

The script will try to contact the “host:port” within the specified timeout and will fail if it does not succeed. If it succeeds, it will execute the command specified after the `--`.

In our example, we should do something like that in our Java container:

```
1 ./wait-for-it.sh mongodb:27017 --timeout=30 --strict -- java -jar ...etc.
```

The idea is to download the script `wait-for-it.sh`, make it executable, copy that into the Docker image of our Java application and adjust the `CMD`. We could also use another command in the `Dockerfile`: `ADD`, which is similar to `COPY` but it can also take a remote URL as the source. So we don’t have to put the BASH script into our sources and copy it with `CMD` and we can simply let the Docker build download such a script. This is the modified `Dockerfile`:

```
1 FROM openjdk:8
2
3 COPY /target/*-jar-with-dependencies.jar /app/app.jar
4
5 # Add wait-for-it and make it executable
6 ADD https://raw.githubusercontent.com/vishnubob/wait-for-it/master/wait-for-it.sh /
7 RUN chmod +x wait-for-it.sh
8
9 CMD [ "./wait-for-it.sh", "mongodb:27017", "--timeout=30", "--strict", "--", "java", \
10 "-jar", "/app/app.jar", "mongodb"]
```

If we now rebuild our Docker image we should see the additional steps (during the build the BASH script will be downloaded—the URL is the standard format for downloading a file directly from the master branch of a GitHub repository):

```
1 ...
2 [INFO] DOCKER> Step 3/5 : ADD https://raw.githubusercontent.com/vishnubob/wait-for-i\
3 t/master/wait-for-it.sh /
4 [INFO] DOCKER>   Downloading
5 [INFO] DOCKER> Step 4/5 : RUN chmod +x wait-for-it.sh
6 [INFO] DOCKER> Step 5/5 : CMD [ "./wait-for-it.sh", "mongodb:27017", "--timeout=30", \
7 "--strict", "--", "java", "-jar", "/app/app.jar", "mongodb"]
8 ...
```

Let's now use `docker-compose up`. We might see the following logs that show that MongoDB is started but it is not ready for connections; the `wait-for-it.sh` tries to access the service but it's not ready yet so it tries again later. Finally, the script succeeds and our Java application is started:

```
1 mongodb_1 | ..."Automatically disabling TLS 1.0"...
2 app_1      | wait-for-it.sh: waiting 30 seconds for mongodb:27017
3 ...
4 mongodb_1 | ..."Waiting for connections"...
5 ...
6 app_1      | wait-for-it.sh: mongodb:27017 is available after 1 seconds
7 ...
8 app_1      | INFO: Opened connection [connectionId{localValue:2, serverValue:3}] to m\
9 ongodb:27017
10 app_1     | Hello World!
```

As you can imagine, this solution does not work in Windows containers, which do not have BASH and the command `chmod`. If we need to target also Windows containers, we should resort to other solutions. For example, we could design our Java application itself to wait for the connection with

several attempts, before starting the main logic. This might also be the best solution since it will not only make our Docker image resilient to connection failures but it will make our Java application work in such situations even when not running inside a Docker container. Finally, such a strategy would be independent of the operating system. This is left as an exercise.

## 11.9 Docker in Eclipse

The Eclipse plugin for Docker is available from the official Eclipse update site: you need to install “Docker Tooling”. This provides an editor for `Dockerfile` and a launch configuration for building, including support for Docker Compose.

Moreover, it provides tools for pulling images and for starting containers, besides mechanisms for managing images and containers (e.g., listing and removal). These features are available in several views (which can be activated by switching to the “Docker Tooling” perspective):

### Docker Explorer

a tree listing all images and containers.

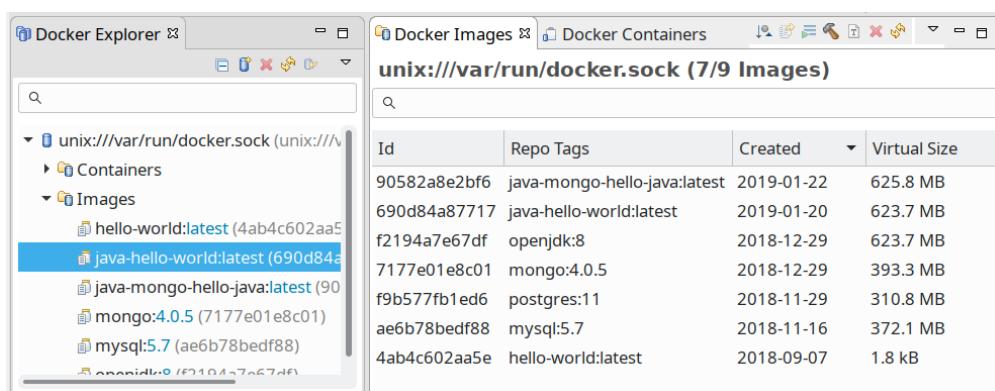
### Docker Images

a table listing all images.

### Docker Containers

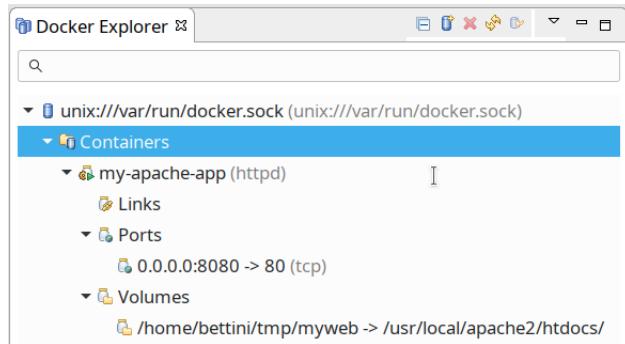
a table listing all containers.

The Docker Explorer should automatically connect to the Docker demon. If not, it allows you to specify such a connection.



The Docker Tooling perspective

Expanding a container node in the Docker Explorer shows information for running containers, such as exposed ports and volumes. For example, if we run Apache as shown in Section [Run a server in a container](#), we can see the exposed and published ports and volumes (in this example we mapped a local directory to a data volume):

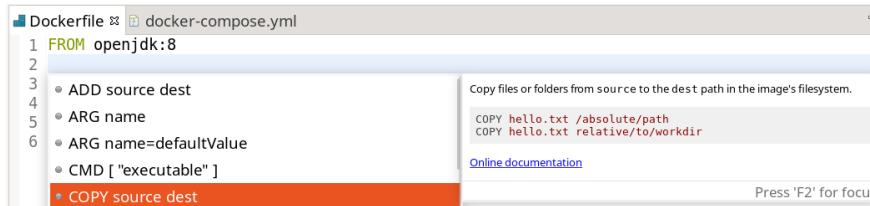


The Docker Explorer showing information of a running container

The context menu of images, and toolbar buttons, shown in these views allow you to start a container, showing dialogs in which the arguments for the `run` command can be specified. These dialogs are also pre-filled with information retrieved by the images (e.g., the default command, possible volumes used by the image, etc.).

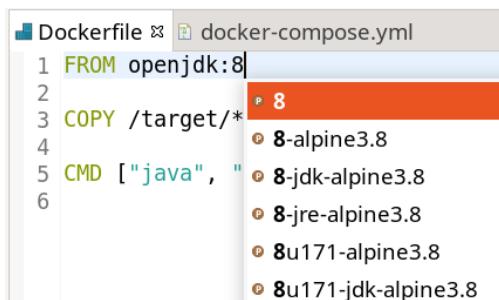
Run configurations for running containers, building images (from a `Dockerfile`), and running Docker Compose (from a `docker-compose.yml`), just like any Eclipse Run configuration, can be configured with arguments and possibly stored locally in the project folders (see Chapter *Eclipse*, Section *Eclipse Run configurations*).

Finally, rich-editor support for the files `Dockerfile` and `docker-compose.yml` is also provided, including syntax highlighting, validation, and code completion:



Code completion in Dockerfile for commands

Code completion is also provided for tags of images. In that case, the tool has to download information on available tags from the Internet, so the proposals might take some time to be shown:



Code completion in Dockerfile for image tags

## 11.10 Push to DockerHub

Publishing an image on DockerHub is quite easy:

1. You register on DockerHub
2. You build your image specifying your userid as the prefix of the image name, e.g., <your userid>/java-hello-world (only official images are without prefix)
3. You run the command `docker login` (you will have to enter your userid and your password)
4. You run the `docker push` command specifying the full image name

It is also possible to push an image on DockerHub directly from a GitHub Actions workflow. We refer to the official documentation for that and we leave this task as an exercise.

## 11.11 Reproducibility in Docker

We already described the issue of reproducibility in build automation and continuous integration in Chapter [Maven](#), Section [\*Reproducibility in the build\*](#), and in Chapter [Continuous Integration](#), Section [\*Reproducibility in CI\*](#).

Docker ensures that a container running a specific image will behave the same way on any system that can run Docker. However, this holds only if you specify the explicit version of the Docker image. Relying on `latest` (the default version if none is specified), undermines reproducibility. As we said above, a different version of MongoDB might print different strings in the log and this might break our build if we rely on wait conditions based on specific printed strings and we simply use the “latest” version of a Docker image.

For this reason, in this chapter, we always specified the version of the used images. For MongoDB, we specified a complete version, while for `openjdk` we only specified the major version, since we do not foresee problems if we will implicitly use in the future upgrades to version 8. However, we might also consider some optimizations when building our Docker images. In this example, for our Java application, we use `openjdk` as the base image. This means that our Docker image will include the whole JDK, that is JVM and the Java compiler. The Java compiler is useless in our case since we are not compiling our Java application inside the Docker container. We could then use a base image that only includes the JVM (i.e., the JRE—Java Runtime Environment). By looking at the website of the `openjdk` Docker image, we find that there is a specific version tag `openjdk:8-jre`. (Alternatively, we could use the code completion mechanisms provided by the Docker tooling in Eclipse, see Section [\*Docker in Eclipse\*](#).) As an exercise, try to switch to this new image and verify that everything still works. The image with the JDK is about 500 MB, while the one with only the JRE is about 270MB. These numbers refer to Linux images. For Windows containers, the numbers might be different.

# 12. Integration tests

While writing unit tests for single components in isolation is crucial for implementing the components with confidence, at some point we should also make sure that components still work correctly when integrated, possibly interacting with real external services that we mocked in unit tests. Tests that involve the integration of several components of an application (e.g., other classes of the same application, real classes from external libraries, additional external services like a database) are **integration tests**.

We can then summarize the main features of unit tests and integration tests as follows:

## Unit tests

test a single SUT in isolation: all possible dependencies are mocked. If a unit test fails, there is surely something wrong in the single component under test (if we do not consider possible mock and stubbing errors). They are meant to be really fast, as we have always seen in the previous chapters.

## Integration tests

test a component when integrated with at least a real dependency implementation. Typically they test the correct behavior of a component (or several components) when interacting with external services, like a database or a remote service. A failed integration test might also be due to a (breaking) change in the used external services. Integration tests are allowed (and usually expected) to be slower than unit tests. In particular, these tests, when running, also have to wait for the used external services to be up and running, and this adds some overhead.

For all the above properties, you should write many small and fast unit tests and only some more coarse-grained integration tests. We will go back to this point in the rest of the chapter.

It is also important to keep unit tests and integration tests separate, in particular, during the build execution. Moreover, unit tests are meant to be run after each change in the code, while integration tests might be run less often, especially if they require much more time than unit tests. Typically, integration tests are run in the CI server.

Finally, it usually makes no sense to run integration tests if unit tests fail: integration tests are expected to fail as well, since, if single components do not behave correctly in isolation and in a mocked environment, they are likely to fail when integrated together and with external services.

As we will see in the examples of this chapter, integration tests do not necessarily have to test the whole application: as long as at least a few real dependencies are used, we consider a test an integration test.

Note that we still use JUnit for writing integration tests. The “Unit” in the name is not meant to relegate JUnit to unit tests only.

Integration tests, besides external components, should connect our components (e.g., objects of our classes). Integration tests are meant to be written after the single components involved in the test has been fully tested, with unit tests. Thus, single components are known to work correctly in isolation and are now tested together. Usually, integration tests are expected to succeed right away, since they should verify that already tested components still work together.

## 12.1 Our running example

To get familiar with integration tests, we will implement a few simple classes that interact together in a **MVC architecture (Model-View-Controller)**.<sup>1</sup>

In this chapter we do not implement the “view” part: we will concentrate on the controller, which acts as a bridge between the view and the data store of our model. We will implement the view in Chapter *UI tests*.

As done in Chapter *Mocking*, Section *Mockito: a tutorial*, we use the **Repository** pattern ([Eva03](#)) for the access to the database. In that chapter, we did not implement the repository: in unit tests we were mocking and stubbing its methods. In this chapter, we use mocking for the repository when writing the unit tests for the controller. However, we also develop an implementation of the repository that will be used in integration tests.

In this example, we use Mockito and AssertJ as test dependencies, thus, make sure you add them to the POM, as we have already seen in previous chapters.

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    }
13
14    // getters, setters, equals, hashCode, toString...
15 }
```

The repository interface is as follows:

---

<sup>1</sup>Actually, we use a variant of the MVC, called Model-View-Presenter, <https://en.wikipedia.org/wiki/Model-view-presenter>. However, we still call it MVC in this book.

```
1 package com.examples.school.repository;
2 ...
3 public interface StudentRepository {
4     public List<Student> findAll();
5     public Student findById(String id);
6     public void save(Student student);
7     public void delete(String id);
8 }
```

The view interface is as follows:

```
1 package com.examples.school.view;
2 ...
3 public interface StudentView {
4     void showAllStudents(List<Student> students);
5     void showError(String message, Student student);
6     void studentAdded(Student student);
7     void studentRemoved(Student student);
8 }
```

Finally, we have the following implementation of the controller:<sup>2</sup>

```
1 package com.examples.school.controller;
2 ...
3 public class SchoolController {
4     private StudentView studentView;
5     private StudentRepository studentRepository;
6
7     public SchoolController(StudentView studentView,
8         StudentRepository studentRepository) {
9         this.studentView = studentView;
10        this.studentRepository = studentRepository;
11    }
12
13    public void allStudents() {
14        studentView.showAllStudents(studentRepository.findAll());
15    }
16
17    public void newStudent(Student student) {
18        Student existingStudent = studentRepository.findById(student.getId());
```

---

<sup>2</sup>We could have introduced an interface also for the controller, but for this project, the interface is not strictly required. We can still mock the controller when we need to, even if it is not an abstract type.

```
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 }
```

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.



The controller takes care of avoiding saving several students with the same id. Thus, the id is meant to represent the *primary key* in the database. We manually take care of the id in this example just for demonstration purposes and to make the example more interesting for the aims of this book. However, in general, primary key management should be delegated to the database itself. Similarly, several database operations, like querying and updating, should be executed in *transactions*, relying on transaction mechanisms provided by the database. This ensures several nice properties, including *atomicity*, in the presence of several clients using the same database. However, we will not deal with such features in this book, and we concentrate on other mechanisms.

The unit tests for the controller are written using Mockito:

```
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    private AutoCloseable closeable;
15
16    @Before
17    public void setup() {
18        closeable = MockitoAnnotations.openMocks(this);
19    }
20
21    @After
22    public void releaseMocks() throws Exception {
23        closeable.close();
24    }
25
26    @Test
27    public void testAllStudents() {
28        List<Student> students = asList(new Student());
29        when(studentRepository.findAll())
30            .thenReturn(students);
31        schoolController.allStudents();
32        verify(studentView).showAllStudents(students);
33    }
34
35    @Test
36    public void testNewStudentWhenStudentDoesNotExist() {
37        Student student = new Student("1", "test");
38        when(studentRepository.findById("1"))
39            .thenReturn(null);
40        schoolController.newStudent(student);
41        InOrder inOrder = inOrder(studentRepository, studentView);
42        inOrder.verify(studentRepository).save(student);
43        inOrder.verify(studentView).studentAdded(student);
```

```
44     }
45
46     @Test
47     public void testNewStudentWhenStudentAlreadyExists() {
48         Student studentToAdd = new Student("1", "test");
49         Student existingStudent = new Student("1", "name");
50         when(studentRepository.findById("1"))
51             .thenReturn(existingStudent);
52         schoolController.newStudent(studentToAdd);
53         verify(studentView)
54             .showError("Already existing student with id 1", existingStudent);
55         verifyNoMoreInteractions(ignoreStubs(studentRepository));
56     }
57
58     @Test
59     public void testDeleteStudentWhenStudentExists() {
60         Student studentToDelete = new Student("1", "test");
61         when(studentRepository.findById("1"))
62             .thenReturn(studentToDelete);
63         schoolController.deleteStudent(studentToDelete);
64         InOrder inOrder = inOrder(studentRepository, studentView);
65         inOrder.verify(studentRepository).delete("1");
66         inOrder.verify(studentView).studentRemoved(studentToDelete);
67     }
68
69     @Test
70     public void testDeleteStudentWhenStudentDoesNotExist() {
71         Student student = new Student("1", "test");
72         when(studentRepository.findById("1"))
73             .thenReturn(null);
74         schoolController.deleteStudent(student);
75         verify(studentView)
76             .showError("No existing student with id 1", student);
77         verifyNoMoreInteractions(ignoreStubs(studentRepository));
78     }
79 }
```

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 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.



The controller and the interfaces have been implemented with TDD, writing the unit tests first. Try to re-implement everything from scratch yourself (including the interfaces) starting from the unit tests.

## 12.2 Unit tests with databases

Now we want to write an implementation of `StudentRepository` using MongoDB, `StudentMongoRepository`.

We have already partly used this database in Chapter [Docker](#), Section [Docker networks](#). MongoDB, <https://www.mongodb.com/>, is a *NoSQL* database. It is document-oriented instead of relying on tables and relations like relational databases. We use it in this chapter since it is straightforward to setup. For example, differently from MySQL or PostgreSQL, it requires no configuration. Since the focus of the chapter is on integration tests, and not on database programming, using such a NoSQL database allows us to focus on the subject of the chapter, without spending too much time on the database layer. As we anticipated in the previous section, we skip additional advanced topics in database programming like *transactions* and automatic *primary key* management. Indeed, implementing an application relying on a database requires several tunings, which we will not see in this book.

Here we use only the very basic features of MongoDB. (We have already used MongoDB in Chapter [Docker](#), for presenting a use case of Docker.) To make the code understandable, we also give a very minimal introduction to the Mongo Java API.

First of all, we need to add this dependency to be able to use the Mongo Java API:

```
1 <dependency>
2   <groupId>org.mongodb</groupId>
3   <artifactId>mongo-java-driver</artifactId>
4   <version>3.12.7</version>
5 </dependency>
6 <dependency>
7   <!-- required to see Mongo Java Driver logs -->
8   <groupId>ch.qos.logback</groupId>
9   <artifactId>logback-classic</artifactId>
10  <version>1.2.3</version>
11 </dependency>
```

The logging dependency is required to see the logs of communications with the database.



When a unit test fails it is usually straightforward to understand the reason for the failure: dependencies are mocked and stubbed so that they behave as we want them to. On the other hand, when we communicate with a server, as we do in integration tests, it is harder to understand the reason for the failure: is it in our code or is something going wrong during the communication with the server? For this reason, having the logs of the communication is crucial in integration tests.

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.

```
1 // create the connection with the server
2 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-the-fly, without any schema):

```
1 MongoDB db = mongoClient.getDatabase("mydb");
2 MongoCollection<Document> collection = db.getCollection("examples");
```

Since MongoDB is document-oriented, we write and retrieve documents, which consist of key-value pairs. In our application they represent values of `Student` instances (the keys are the field names and the values are the field values):

```
1 // create the connection with the server
2 MongoClient mongoClient = new MongoClient(mongoHost);
3 MongoDB db = mongoClient.getDatabase("school");
4 MongoCollection<Document> collection = db.getCollection("student");
5 Document doc = new Document("id", "id")
6     .append("name", "A student");
7 collection.insertOne(doc);
8 collection.find().first().get("id");
```



To keep the example simple, we will not use POJO MongoDB features, <http://mongodb.github.io/mongo-java-driver/3.9/driver/getting-started/quick-start-pojo/>, that automatically serializes Java objects into documents and vice versa.

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.

```
1 package com.examples.school.repository.mongo;
2
3 import org.bson.Document;
4 import com.mongodb.MongoClient;
5 import com.mongodb.client.MongoCollection;
6 ...
7 public class StudentMongoRepository implements StudentRepository {
8
9     public static final String SCHOOL_DB_NAME = "school";
10    public static final String STUDENT_COLLECTION_NAME = "student";
11    private MongoCollection<Document> studentCollection;
12
13    public StudentMongoRepository(MongoClient client) {
14        studentCollection = client
15            .getDatabase(SCHOOL_DB_NAME)
16            .getCollection(STUDENT_COLLECTION_NAME);
17    }
18    ...
19 }
```



As usual, use Eclipse mechanisms: once the collection is retrieved, assign it to studentCollection, which is not declared; then use quickfix to create the field. Then, use refactoring for extracting the database and collection names.

Now we want to test this implementation with unit tests. We could have also implemented this class in a TDD fashion, but we still haven't seen how to write tests in the presence of a database.

### 12.2.1 Mock the MongoClient?

We could mock the implementation of the MongoClient API, but mocking its methods is not trivial, and, as said in Chapter [Mocking](#), Section [What to mock](#), we should not mock third party dependencies.

For example, let's say we want to mock MongoClient and stub the retrieval of an element, collection.find().first().get("id") so that it returns a name. This would require approximately all these lines:

```
1 // pseudo code, without generics
2 MongoClient mongoClient = mock(MongoClient.class);
3 MongoDatabase db = mock(MongoDatabase.class);
4 when(mongoClient.getDatabase("school")).thenReturn(db);
5 MongoCollection collection = mock(MongoCollection.class);
6 when(db.getCollection("student")).thenReturn(collection);
7 FindIterable iterable = mock(FindIterable);
8 when(collection.find()).thenReturn(iterable);
9 Document document = mock(Document);
10 when(iterable.first()).thenReturn(document);
11 when(document.get("id")).thenReturn("test name");
```

This makes no sense. It makes writing tests hard and error-prone. Not to mention that we have to know too many internal details of the Mongo API.

Moreover, we should ask ourselves whether mocking would make sense in this context anyway. If we consider the controller, its unit test must verify its behavior in terms of interaction with the view and the repository. It makes sense to mock and stub the repository and verify the interactions with the view since this is enough to test its behavior. For example, when the controller is required to add a new student and the repository does not contain a student with the same id, the controller must save the student in the repository and call a method on the view. Mockito is enough to test this behavior. Indeed, that's the only logic implemented by the controller.

On the contrary, the implementation of our repository has very little logic. It executes operations like reading contents from the database and converting them to domain model objects and vice versa. It also implements a few database queries. Its correct behavior highly depends on the correct serialization/deserialization of objects, and on how the database interprets queries. Mocking and stubbing the database would make our tests only verify that our repository calls specific methods on the MongoClient. It would not allow us to verify the serialization/deserialization correctness. We could not even have confidence in the correctness of queries.



Never ever mock third party API. Do not even spy third party API.

## 12.2.2 Use an in-memory database

If we do not want to interact with a real server for writing unit tests of our repository implementation, the closest thing to a mock implementation is using an in-memory implementation of the database, if available. Fortunately, there already exist projects that provide an in-memory implementation of a MongoDB server, without saving anything to disks and meant to be used for easy unit testing. We use one of such projects, [MongoDB Java Server](https://github.com/bwaldvogel/mongo-java-server), <https://github.com/bwaldvogel/mongo-java-server>. Such an in-memory implementation could be seen as a test double classified as a *fake* (see Chapter [Mocking](#), Section [Fakes](#)).

The idea is to start this version of MongoDB, which will run in memory, and then create a MongoClient that connects to this in-memory database:

```
1 MongoServer server = new MongoServer(new MemoryBackend());
2 // bind on a random local port
3 InetSocketAddress serverAddress = server.bind();
4 MongoClient client = new MongoClient(new ServerAddress(serverAddress));
```

So let's add this test dependency:

```
1 <dependency>
2   <groupId>de.bwaldvogel</groupId>
3   <artifactId>mongo-java-server</artifactId>
4   <version>1.11.1</version>
5   <scope>test</scope>
6 </dependency>
```

And create the StudentMongoRepositoryTest (use **Ctrl + 1** on StudentMongoRepository to create the JUnit test case):

```
1 package com.examples.school.repository.mongo;
2
3 import static org.assertj.core.api.Assertions.*;
4 import java.net.InetSocketAddress;
5 import org.bson.Document;
6 import com.mongodb.MongoClient;
7 import com.mongodb.ServerAddress;
8 import com.mongodb.client.MongoCollection;
9 import com.mongodb.client.MongoDatabase;
10 import de.bwaldvogel.mongo.MongoServer;
11 import de.bwaldvogel.mongo.backend.memory.MemoryBackend;
12 ...
13 public class StudentMongoRepositoryTest {
14
15     private static MongoServer server;
16     private static InetSocketAddress serverAddress;
17
18     private MongoClient client;
19     private StudentMongoRepository studentRepository;
20     private MongoCollection<Document> studentCollection;
21
22     @BeforeClass
```

```
23  public static void setupServer() {
24      server = new MongoServer(new MemoryBackend());
25      // bind on a random local port
26      serverAddress = server.bind();
27  }
28
29  @AfterClass
30  public static void shutdownServer() {
31      server.shutdown();
32  }
33
34  @Before
35  public void setup() {
36      client = new MongoClient(new ServerAddress(serverAddress));
37      studentRepository = new StudentMongoRepository(client);
38      MongoDatabase database = client.getDatabase(SCHOOL_DB_NAME);
39      // make sure we always start with a clean database
40      database.drop();
41      studentCollection = database.getCollection(STUDENT_COLLECTION_NAME);
42  }
43
44  @After
45  public void tearDown() {
46      client.close();
47  }
48  ...
49 }
```

Note that to make every single test independent from each other, it is crucial to always drop the database we are using in the tests, in case it was created by previous tests (indeed, we create the server only once). If we do not drop the database, all the tests will use the contents added by the previous tests and the tests cannot run independently from each other. Indeed, the database contents are part of the test fixture. The database must then always be in a known fixed state (i.e., empty in our case) before each test.

In the tests we are going to write, we will use the `studentCollection` instance to create the context for scenarios for testing our repository and for verifying its correct behavior. Recall that if we are testing (and implementing) 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. Most of all, we must test the logic of a method independently from other methods of the same SUT, as we have always done in unit tests.

Let's start with `findAll` when the collection is empty:

```
1 @Test
2 public void testFindAllWhenDatabaseIsEmpty() {
3     assertThat(studentRepository.findAll()).isEmpty();
4 }
```

As expected, this fails since `findAll` now returns null.

Note however that the console shows the log from the in-memory database. This shows at least that our in-memory database setup works.

Let's follow TDD strictly and proceed by a small step returning an empty collection just to make this test succeed:

```
1 @Override
2 public List<Student> findAll() {
3     return Collections.emptyList();
4 }
```

Let's proceed with the case when the database contains some students (we create a helper method that we'll reuse for other tests). We manually insert students in the database from the test and we verify that our repository correctly reads them:

```
1 @Test
2 public void testFindAllWhenDatabaseIsNotEmpty() {
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 private void addTestStudentToDatabase(String id, String name) {
12     studentCollection.insertOne(
13         new Document()
14             .append("id", id)
15             .append("name", name));
16 }
```

The test fails since our implementation always returns an empty list. Let's implement our method so that it reads documents from the database and converts them to `Student` objects:

```
1 import java.util.stream.Collectors;
2 import java.util.stream.StreamSupport;
3 ...
4 @Override
5 public List<Student> findAll() {
6     return StreamSupport.
7         stream(studentCollection.find().spliterator(), false)
8         .map(d -> new Student(""+d.get("id"), ""+d.get("name")))
9         .collect(Collectors.toList());
10 }
```

All tests pass.



This implementation assumes that the collection we are using only contains documents representing Student objects. In a real application, we should apply some filtering or rely on MongoDB POJO support for reading contents based on a Java type.

Let's proceed a little bit faster by implementing the tests for the two cases of `findById`:

```
1 @Test
2 public void testFindByIdNotFound() {
3     assertThat(studentRepository.findById("1"))
4         .isNull();
5 }
6
7 @Test
8 public void testFindByIdFound() {
9     addTestStudentToDatabase("1", "test1");
10    addTestStudentToDatabase("2", "test2");
11    assertThat(studentRepository.findById("2"))
12        .isEqualTo(new Student("2", "test2"));
13 }
```

The first one already succeeds as expected, while the second one fails.

Let's implement the method correctly. While doing that, we refactor the creation of a `Student` from a `Document` into a method that we reuse:

```
1  @Override
2  public List<Student> findAll() {
3      return StreamSupport.
4          stream(studentCollection.find().spliterator(), false)
5          .map(this::fromDocumentToStudent)
6          .collect(Collectors.toList());
7  }
8
9  private Student fromDocumentToStudent(Document d) {
10     return new Student(""+d.get("id"), ""+d.get("name"));
11 }
12
13 @Override
14 public Student findById(String id) {
15     Document d = studentCollection.find(Filters.eq("id", id)).first();
16     if (d != null)
17         return fromDocumentToStudent(d);
18     return null;
19 }
```

For finding a document matching a given criteria we use com.mongodb.client.model.Filters API.  
We proceed similarly for the remaining methods. These are the tests:

```
1  @Test
2  public void testSave() {
3      Student student = new Student("1", "added student");
4      studentRepository.save(student);
5      assertThat(readAllStudentsFromDatabase())
6          .containsExactly(student);
7  }
8
9  @Test
10 public void testDelete() {
11     addTestStudentToDatabase("1", "test1");
12     studentRepository.delete("1");
13     assertThat(readAllStudentsFromDatabase())
14         .isEmpty();
15 }
16
17 private List<Student> readAllStudentsFromDatabase() {
18     return StreamSupport.
19         stream(studentCollection.find().spliterator(), false)
```

```

20         .map(d -> new Student(""+d.get("id"), ""+d.get("name")))
21         .collect(Collectors.toList());
22     }

```

and these are the implementations:

```

1  @Override
2  public void save(Student student) {
3      studentCollection.insertOne(
4          new Document()
5              .append("id", student.getId())
6              .append("name", student.getName()));
7  }
8
9  @Override
10 public void delete(String id) {
11     studentCollection.deleteOne(Filters.eq("id", id));
12 }

```

There is a few code duplication between the tests and the code concerning reading from and writing to the database. This is expected if we do not want to use a method of the SUT for reading while testing a method of the SUT that writes, and the other way round. In any case, the SUT might also implement read and write operations differently from the test code in a real application.

Summarizing, we applied TDD for implementing our `StudentMongoRepository` by relying on an in-memory database.

Using an in-memory database is not always a good idea though. With mocking, we mock our own types and stub our own abstract methods assuming that the final implementation will be correct. We mock and stub our dependencies, assuming that their implementations are correct in the end. It will be then our responsibility to implement our dependencies correctly.

On the contrary, using an in-memory database means using an implementation of the database that does not necessarily correspond to the real database implementation. Typically, such in-memory databases do not implement all the real database features and mechanisms (the limitations of such in-memory databases are usually documented, see, e.g., <https://github.com/bwaldvogel/mongo-java-server>).

Our tests might succeed with an in-memory database but our code might fail when using the real database. Usually, when implementing and testing simple CRUD operations<sup>3</sup>, or simple queries, like in our case, the in-memory database is usually reliable. In any case, we need tests for crucial scenarios also with a real database.

Integration tests aim at solving these problems as well. Moreover, thanks to Docker, it is easy to write integration tests against a real database server, as we will see in the next sections.

---

<sup>3</sup>CRUD stands for “Create”, “Read”, “Update”, and “Delete”, the four basic functions of persistent storage.

## 12.3 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.

### 12.3.1 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>
13      <sources>
14        <source>src/it/java</source>
15      </sources>
16    </configuration>
17  </execution>
18 </executions>
19 </plugin>
```

Then we update the Maven project from Eclipse; the new folder will be now handled as a test source folder.

## 12.3.2 Integration tests with 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, PostgreSQL, and MongoDB. When there is no such a specific module, we can always run a generic container, using `GenericContainer`.

We first need to add this test dependency in our POM:

```
1 <properties>
2   <testcontainers.version>1.15.1</testcontainers.version>
3 ...
4 <dependency>
5   <groupId>org.testcontainers</groupId>
6   <artifactId>testcontainers</artifactId>
7   <version>${testcontainers.version}</version>
8   <scope>test</scope>
9 </dependency>
```

For demonstration, let's start writing an integration test for our `StudentMongoRepository`, `StudentMongoRepositoryTestcontainersIT`, in the folder `src/it/java`, by using the class `GenericContainer` provided by Testcontainers:

```
1 package com.examples.school.repository.mongo;
2
3 import org.junit.ClassRule;
4 import org.testcontainers.containers.GenericContainer;
5 ...
6 public class StudentMongoRepositoryTestcontainersIT {
7
8     @SuppressWarnings("rawtypes")
9     @ClassRule
10    public static final GenericContainer mongo =
11        new GenericContainer("mongo:4.4.3")
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 }
```

The `GenericContainer` will execute the specified image in a Docker container. Since it is also a JUnit rule, with the annotation `@ClassRule` we make sure the Docker container will be started before executing all tests and it will be automatically stopped and removed after all tests are run.



We first saw JUnit rules in Chapter [JUnit](#), Section *A first example*. `@ClassRule` is similar to `@Rule`, but its life cycle execution is on class level rather than on instance level, thus it corresponds to static methods annotated with `@BeforeClass`, <https://github.com/junit-team/junit4/wiki/rules>.

We configure such a container by exposing ports (in this case, the default port of MongoDB). 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 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 // testSave and testDelete are similar

```

In this simple example, we can simply copy them from `StudentMongoRepositoryTest`, together with private utility methods, changing test method names. Of course, there will be some code duplication. As we have already seen, e.g., in Chapter [JUnit](#), Section *Beware of code duplication removal in tests* and in Chapter [Mocking](#), this is admissible for tests, especially if they are more readable than a version relying on abstractions.

The `GenericContainer` from `Testcontainers` is the most general mechanism for running Docker containers. As said above, `Testcontainers` has many additional predefined modules for the most common services and databases. Such modules are available as separate additional dependencies. It recently introduced one for MongoDB. Thus, we add the corresponding dependency

```

1 <dependency>
2   <groupId>org.testcontainers</groupId>
3   <artifactId>mongodb</artifactId>
4   <version>${testcontainers.version}</version>
5   <scope>test</scope>
6 </dependency>

```

and we use the corresponding class `MongoDBContainer`. The initial part of the above test becomes:

```
1 package com.examples.school.repository.mongo;
2
3 import org.testcontainers.containers.MongoDBContainer;
4 ...
5 public class StudentMongoRepositoryTestcontainersIT {
6
7     @ClassRule
8     public static final MongoDBContainer mongo =
9         new MongoDBContainer("mongo:4.4.3");
```

Even in this simple example, there's a small improvement concerning the previous test, where we used `GenericContainer`. We did not have to map the standard port 27017 since that is taken care of automatically by `MongoDBContainer`. Moreover, `MongoDBContainer` also takes care of waiting for the MongoDB server to be ready for accepting connections before running our tests. Thus, especially in more complex scenarios, a `Testcontainers` module for a specific service makes it easier to configure the service running in a Docker container.



As documented in the `Testcontainers` module documentation, the module for MongoDB is still in an incubating phase and its API may change in the future in a non-backward compatible way.

If we use JUnit 5 (see Chapter [JUnit](#), Section [JUnit 5](#) and Chapter [Maven](#), Section [JUnit 5 and Maven](#)), we can use the specific JUnit 5 extension provided by `Testcontainers` after adding this dependency:

```
1 <dependency>
2     <groupId>org.testcontainers</groupId>
3     <artifactId>junit-jupiter</artifactId>
4     <version>${testcontainers.version}</version>
5     <scope>test</scope>
6 </dependency>
```

Now we can use the extension `@Testcontainers`. This annotation is meant to be used directly for annotating our JUnit 5 test case class, without using `@ExtendWith`. This extension automatically starts containers by using the fields annotated with `@Container`. Similarly to `@ClassRule` and `@Rule` annotations seen above, containers declared as static fields will be automatically started before executing all tests and they will be automatically stopped and removed after all tests are run, while containers declared as instance fields will be automatically started and stopped before and after every test method, respectively. This is the JUnit 5 version of the example we have seen before:

```
1 import org.testcontainers.junit.jupiter.Container;
2 import org.testcontainers.junit.jupiter.Testcontainers;
3 ...
4 @Testcontainers
5 class StudentMongoRepositoryTestcontainersJupiterIT {
6
7     @Container
8     static final MongoDBContainer mongo =
9         new MongoDBContainer("mongo:4.4.3");
10
11    private MongoClient client;
12    private StudentMongoRepository studentRepository;
13    private MongoCollection<Document> studentCollection;
14
15    @BeforeEach
16    void setup() {
17        ...
18    }
19
20    @AfterEach
21    void tearDown() {
22        ...
23    }
24 ...
```

Finally, with Testcontainers, the lifecycle of the Docker containers can also be managed manually in our tests: the `start` and `stop` methods of containers can be called appropriately in `@Before`, `@After`, `@BeforeClass`, and `@AfterClass` (or in the corresponding lifecycle methods of JUnit 5).



At the time of writing, Testcontainers does not support Windows Containers on Windows (see Chapter [Docker](#), Section [Windows containers](#)). We have to consider this especially if we build our project in the Windows virtual environment provided by GitHub Actions (Chapter [Continuous Integration](#)). We will get back to this in Section [Running integration tests in GitHub Actions](#).

### 12.3.3 Running integration 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 [https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle\\_Reference](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference) 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 because 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`.

Different 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-integration` 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>
```

Now, running `mvn test` will only run unit tests while running `mvn verify` will also run integration tests, after unit tests have passed.

### 12.3.4 Integration 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     private AutoCloseable closeable;
13
14     @Before
15     public void setUp() {
16         closeable = MockitoAnnotations.openMocks(this);
17         studentRepository = new StudentMongoRepository(new MongoClient("localhost"));
18         // explicit empty the database through the repository
19         for (Student student : studentRepository.findAll()) {
20             studentRepository.delete(student.getId());
21         }
22         schoolController = new SchoolController(studentView, studentRepository);
23     }
24     ... // @After as usual

```

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 interacts 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 }
```

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:

```
1 docker run -p 27017:27017 --rm mongo:4.4.3
```

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.



In this example we do not configure the `failsafe` plugin and the Docker plugin in a separate profile. Due to the way the plugins are bound to Maven lifecycle phases, if we do not want to run integration tests, but only unit tests, it is enough to run the Maven `test` phase instead of `verify`.

```
1 <plugin>
2   <groupId>io.fabric8</groupId>
3   <artifactId>docker-maven-plugin</artifactId>
4   <version>0.34.1</version>
5   <configuration>
6     <images>
7       <image>
8         <name>mongo:4.4.3</name>
9         <run>
10        <ports>
11          <port>27017:27017</port>
12        </ports>
13      </run>
14    </image>
15  </images>
16 </configuration>
17 <executions>
18   <execution>
19     <id>docker-start</id>
20     <phase>pre-integration-test</phase>
21     <goals>
22       <goal>start</goal>
23     </goals>
24   </execution>
25   <execution>
26     <id>docker-stop</id>
27     <phase>post-integration-test</phase>
28     <goals>
29       <goal>stop</goal>
30     </goals>
31   </execution>
32 </executions>
33 </plugin>
```

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 you previously started the MongoDB container from the command line, make sure you stop it before running the Maven build. Otherwise, you would end up with two containers mapped to the same localhost port, leading to a conflict, and a build failure.

If we now run

```
1 mvn 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

```
1 mvn clean verify \
2   surefire-report:report-only \
3   surefire-report:failsafe-report-only \
4   org.apache.maven.plugins:maven-site-plugin:3.7.1:site \
5   -DgenerateReports=false
```

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).



Now that we have configured the `docker-maven-plugin`, if we need to run the Docker container manually for running tests in Eclipse, we can simply invoke the `start` goal, e.g., `mvn docker:start`. The build succeeds and terminates but the Docker container is now running. When you do not need it anymore, remember to stop it with `mvn docker:stop`. Remember that `docker:start` is a goal of the Maven plugin, it has nothing to do with the execution's id. The same holds for `docker:stop`.

Using a fixed port on the localhost (27017 in this example) is not always the best solution. As we saw, if we have started the MongoDB Docker container manually from the command line we have to remember to stop it before running the Maven build, otherwise, we will get a conflict. Remember that the conflict is due to the fact we mapped the Docker container's port to the same port on the localhost. On the contrary, the internal ports of two MongoDB containers never conflict.

In Chapter [Docker](#), Section [Docker networks](#), we saw that the `docker-maven-plugin` allows us to map the container's port to a random port on the local computer.<sup>4</sup> also in that respect. Such a free random port will be dynamically selected by Docker and its value will be assigned to a Maven property, which can then be used later in the same POM. Let's use this technique in this example:

---

<sup>4</sup>Actually, this is a feature provided by Docker itself.

```
1 <image>
2   <name>mongo:4.4.3</name>
3   <run>
4     <ports>
5       <port>mongo.mapped.port:27017</port>
6     </ports>
7   </run>
8 </image>
```

We can then use the Maven property `mongo.mapped.port` in our POM. For example, we pass it as a system property to the integration tests that are run by the `maven-failsafe-plugin`. We do that in the configuration of the plugin using the element `<systemPropertyVariables>`:

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-failsafe-plugin</artifactId>
4   <version>2.22.1</version>
5   <configuration>
6     <systemPropertyVariables>
7       <mongo.port>${mongo.mapped.port}</mongo.port>
8     </systemPropertyVariables>
9   </configuration>
10  <executions>
11    ...
```

Now, our integration tests can access the system property `mongo.port`, which is initialized by Maven with the value of `mongo.mapped.port`. Thus, let's change our `SchoolControllerIT`, which relies on a Docker container that is already started, as follows:

```
1 public class SchoolControllerIT {
2 ...
3   private static int mongoPort =
4     Integer.parseInt(System.getProperty("mongo.port", "27017"));
5
6   @Before
7   public void setUp() {
8     closeable = MockitoAnnotations.openMocks(this);
9     studentRepository = new StudentMongoRepository(
10       new MongoClient(
11         new ServerAddress("localhost", mongoPort)));
12   ...
```

The integration test retrieves the mapped port of the container from the system property `mongo.port`. If not set, it defaults to 27017. This way, if we run the test manually from Eclipse after manually starting the Docker container (`docker run -p 27017:27017 --rm mongo:4.4.3`), then the test will still work. If we forget to stop the Docker container and we run the Maven build, everything will still succeed since the integration test will be passed the random mapped port of the container started during the Maven build.

This solution, besides the advantages hinted above, works also in operating systems where the port 27017 can be directly used. An example of such a situation is the Windows virtual environment of GitHub Actions (see also Section [Running integration tests in GitHub Actions](#)).

There is a small drawback to this solution. We can still manually start and stop the Docker container by invoking the Maven goals `docker:start` and `docker:stop`, respectively, but we ignore the number of the mapped port, and running that integration test from Eclipse will not work. We could have the value of the mapped port printed on the Console, e.g., by adding a `<wait>` condition as we saw in Chapter [Docker](#), Section [Docker networks](#). Then, we should always modify the Eclipse run configuration by specifying the port value with an explicit `-Dmongo.port=...`. However, this procedure is cumbersome, and each time we restart the container we should update the value of the passed property. Probably, in such a situation, it is better to manually start the Docker container from the command line with the explicit mapped port 27017, which will be used by default by our integration test when the property `mongo.port` is not set.

### 12.3.5 Running integration tests in GitHub Actions

Once the Maven build is set up, running this build in a GitHub Actions workflow is just a matter of creating the YAML file as we have already seen in Chapter [Continuous Integration](#). In Chapter [Docker](#), Section [Using Docker in GitHub Actions](#), we saw how to run such workflows also on macOS and Windows. The workflow files for the three operating systems can be found in the GitHub repository of this example.

In particular, we saw that we need to do a few adjustments especially for the Windows environment. In Section [Integration tests with Docker and Testcontainers](#), we said that at the time of writing Testcontainers does not support Windows Containers on Windows, that is, in the Windows environment of GitHub Actions. If we want the build of this project to succeed also when running on Windows, in GitHub Actions, we have to give up running integration tests that use Testcontainers.



Create a Maven profile that extends the configuration of the `maven-failsafe-plugin` by “excluding” the tests that use Testcontainers, by relying on their names. An example of test exclusion is shown in Chapter [End-to-end tests](#), Section [E2e tests for our application](#). Remember that the main configuration of the `maven-failsafe-plugin` is part of the main build section. Then, enable this profile in the workflow running in Windows. This is what is done in the Windows workflow in the GitHub repository of this example.

## 12.4 Unit or integration tests?

In this final section of this chapter, we further discuss unit and integration tests.

The question of the section has two aims.

First of all, we can ask ourselves whether the tests in `StudentMongoRepositoryTest`, using an in-memory database, are real unit tests. In the literature, you can find several places where these are called integration tests. Indeed, the database is not mocked: it is a real in-memory database. In this book (like in many other places in the literature) we will still consider them as unit tests, since the database, though not mocked, is still not the real database that will be used in production. On the contrary, `StudentMongoRepositoryTestcontainersIT` are considered effectively integration tests, since they communicate with a real database. Even though the database is running in a container, it is the real database, and, in production, Docker is typically used as well for running databases for the final application.

The second aim of the question is whether it makes sense to use an in-memory database for unit tests. Couldn't we write the tests for the repository implementation directly against a real database? We already mentioned that an in-memory database might behave differently from a real database since it does not implement all the features of the real database. Indeed, it is often advised to directly write integration tests for the database layer, against a real database, even because Docker makes this easy nowadays. In this book, we showed both tests with an in-memory database and with a real database. In the end, the final choice depends also on the level of database operations needed in the code. Simple CRUD operations can be reliably tested even with an in-memory database. Complex operations and queries are instead more reliably tested with a real database.

Finally, also the test execution time should be taken into consideration. Starting an in-memory database still requires some time. Unit tests, even when using an in-memory database, are not as fast as the usual unit tests relying on mocking. In our examples, running integration tests with `Testcontainers`, `StudentMongoRepositoryTestcontainersIT` is surely slower than running the unit tests with the in-memory database, due to the startup time of the container. However, the integration tests relying on an already running Docker container, `StudentControllerIT`, have an execution time comparable with the execution time of unit tests with the in-memory database. Moreover, they test against the real database, giving more confidence.

All the above issues must be taken into account when choosing the testing strategy for the database layer.

# 13. UI tests

In this chapter we deal with UI tests, that is, tests that verify the correct behavior of the user interface. In particular, we test the **GUI interface** (Graphical User Interface) implemented with the standard Java toolkit **Swing**.

UI tests are meant to verify that the user interface of an application works correctly. Such a verification usually requires simulating the user behavior. For example, when the user clicks on a button of the UI then we verify that something happens. When certain fields are not correctly filled, then some UI parts, e.g., buttons, should be disabled, etc. In general, we should also verify that the state of the UI changes according to some specific user interactions.

As we mentioned in the introductory Chapter [Testing](#), UI tests do not have to be on the highest level of the test pyramid. As we will see in the first sections of this chapter, we first write unit tests for our GUI in isolation, mocking the external dependencies (in this example, the controller). Then, we also write integration tests for our GUI, which will interact with real external components. We will see that the unit tests we write will be much different from the integration tests, even though they are about the same GUI component. In particular, the unit tests will verify the behavior of the GUI assuming that the external collaborators behave correctly. Then, in integration tests, we will be able to verify a complete user interaction flow, consisting of the collaboration of the GUI and the external components (i.e., the controller and the database.)

Writing UI tests, both unit and integration tests, completely manually might be very hard, since it requires programmatically trigger typical user interface events (e.g., mouse click, selection, text insertion, etc.). Fortunately, for Swing applications, there is a testing framework, **AssertJ Swing**, <http://joel-costigliola.github.io/assertj/assertj-swing.html>, that will allow us to easily write UI tests. It provides a fluent API to simulate user interactions and to easily verify the state of our Swing components.

Note that, in general, UI testing frameworks will not allow us to verify whether the UI “looks good”. If something in the UI code changes concerning the layout of the components (text fields, buttons, etc.) the tests might still succeed, even if we broke the nice layout. In the end, the beauty of the layout of the UI must be verified manually. Still, the correct behavior of the UI can be tested automatically, in a TDD way.

## 13.1 The running example

We continue from the example of the previous chapter. We concentrate on the development of the GUI (Graphical User Interface) by using the standard Java GUI **Swing**.

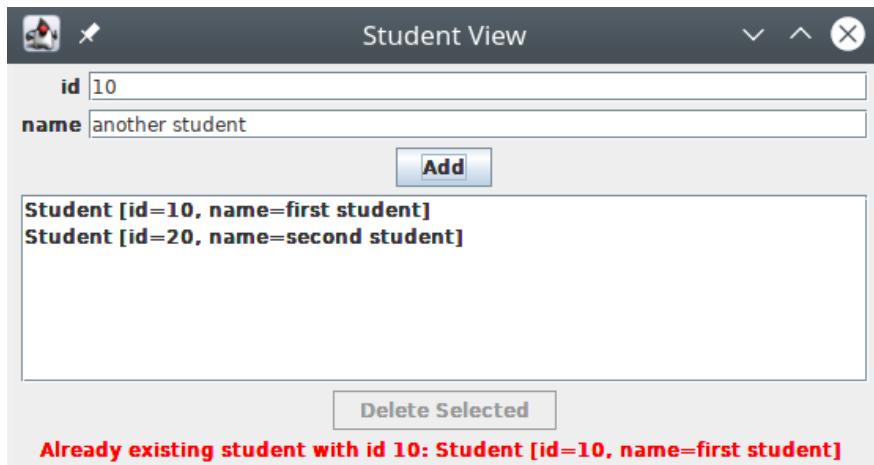
We implement the user interface in the class `Student Swing View`, which also implements the `StudentView` interface, introduced in Chapter [Integration tests](#).

We will not get into details on Swing programming: we will only see the parts that are relevant to our context. For further details, please see the official documentation <https://docs.oracle.com/javase/8/docs/technotes/guides/swing/index.html>.

Writing Java code for the GUI, e.g., with Swing, is usually cumbersome. It is quite hard to get the layout right by manually writing Java code directly dealing with the Swing API. For this reason, we strongly suggest you get familiar with a visual editor. We use **WindowBuilder**, the mainstream visual editor for Eclipse, <https://www.eclipse.org/windowbuilder/>. We use its **Swing Designer**, which can be installed from the main Eclipse update site, or by using the project's update site <https://download.eclipse.org/windowbuilder/latest/>.

WindowBuilder provides a WYSIWYG visual editor to create Swing complex graphical interfaces and generates the corresponding Java code. The generated Java code does not require any additional libraries to compile and run. WindowBuilder also interprets the Java code of the UI: you can customize the generated Java code and the WindowBuilder visual editor will re-interpret it. The WindowBuilder Java editor shows the Java code in the “Source” tab and the visual editor in the “Design” tab.

The final GUI of our application is shown in the next screenshot.<sup>1</sup>



The GUI of our application

The implementation of the GUI should have the following features:

- The “Add” button should be enabled only when both text fields are not blank. When pressed, it should call the controller’s method `newStudent` passing a `Student` object with the values specified in the text fields.
- The view’s interface method `showError` should show the error message and the information about the passed `Student` in the error label on the bottom of the GUI.
- The view’s interface methods `studentAdded` and `studentRemoved` should add and remove the passed `Student` to and from the list, respectively. Both methods should also clear the error label.

<sup>1</sup>At the end of Chapter [End-to-end tests](#), in Section [Change some low-level details](#), we will slightly change the representation of `Student` objects: instead of using the default `toString` representation, `Student [id=<id>, name=<name>]`, we will use the representation `<id> - <name>`.

- The “Delete Selected” button should be enabled only when a student is selected in the list. When pressed, it should call the controller’s method `deleteStudent`.

We will implement these low-level features in a TDD way, by first writing unit tests with **AssertJ Swing**. The controller will be mocked.

From the user’s point of view, the GUI will behave as follows:

- The “Add” button will be used to add a new Student to the database with the values specified in the text fields. If the operation succeeds, the Student will appear in the list.
- The “Delete Selected” button will be used to delete from the database the Student selected in the list. If the operation succeeds, the selected Student will disappear from the list.
- Errors will be shown at the bottom of the view.

We will verify these high-level features with integration tests.

As prescribed by the test pyramid, we will have several unit tests, each one for a scenario of the GUI. We will have only a few integration tests, which will not test (again) internal details of the GUI, like the enabled/disabled state of the buttons.

## 13.2 UI unit tests

In this section, we implement the GUI and its behavior and its unit tests, using **AssertJ Swing**, in a TDD way.

We replace the `assertj-core` with this dependency, which also includes `assertj-core`:

```
1 <dependency>
2   <groupId>org.assertj</groupId>
3   <artifactId>assertj-swing-junit</artifactId>
4   <version>3.17.1</version>
5   <scope>test</scope>
6 </dependency>
```

Note that **AssertJ Swing** has a lower version than **AssertJ Core**. However, it depends transitively on a higher version of `assertj-core`. The version of `assertj-core` included in `assertj-swing-junit` might not be aligned with the most recent version of `assertj-core` that we have used in previous examples.



## Bug in Windows 10

There is a [bug<sup>2</sup>](#) in the JDK in Windows 10, when using a scaling factor (typical of HiDPI displays). The bug concerns the Java AWT Robot, which is used by AssertJ Swing to simulate user interactions in the GUI. Thus, tests concerning mouse movements, like clicking on a button or selecting an element in a list, which we'll write in the rest of this chapter, are likely to fail in Windows 10 if you have a scaling factor different from 100%. The bug has been fixed in the JDK 11.0.1. If you do not want to switch to this more recent version of Java, you must use a 100% scaling factor, to run the examples of this chapter (and the next chapters) in Windows 10.



## AssertJ Swing in macOS Mojave

macOS Mojave has security/privacy mechanisms detecting software that tries to control the mouse and the keyboard, which AssertJ Swing does to simulate user interactions. The first time you run such tests, you will be asked to approve explicitly Java and Eclipse as applications that are allowed to control your computer.

First of all, let's create the `JFrame` of our GUI, `com.examples.school.view.swing.StudentSwipeView` by selecting **File → New → Other... → WindowBuilder → Swing Designer -> JFrame**.

The Java class generated by the WindowBuilder wizard also has a `main` method. We will later extract the `main` method into another class, but for the moment, we can use it to run the Java application just to see whether the GUI layout looks nice.

We make this Java class implement the interface `StudentView` and use Eclipse quickfix to create an empty implementation of the interface methods. We will implement these methods later, after writing the corresponding (failing) tests.



Get familiar with the WindowBuilder visual editor. For example, set the title of the frame to "Student View". The WindowBuilder visual editor will add the corresponding Java statement for setting the title. Run the Java application to see the title of the frame.



You can find a video showing how to create the entire GUI of this example with WindowBuilder at this URL: <https://youtu.be/omnoCxl-4wA>.

Let's create a test case for this class. We use the Quick Assist **Ctrl + 1** (see Section [Quick Assist](#), Chapter [Eclipse](#)) and we select "Create new JUnit test case for...". We create the test case in `src/test/java`.

We use the AssertJ Swing runner (which will allow us to create screenshots in case of failures, as we will see later) and extend the base class of AssertJ Swing, which performs most of the common setup:

<sup>2</sup><https://bugs.openjdk.java.net/browse/JDK-8196030>

```
1 package com.examples.school.view.swing;
2
3 import org.assertj.swing.junit.runner.GUITestRunner;
4 import org.assertj.swing.junit.testcase.AssertJSwingJUnitTestCase;
5 import org.junit.runner.RunWith;
6
7 @RunWith(GUITestRunner.class)
8 public class Student SwingViewTest extends AssertJSwingJUnitTestCase {
9
10    @Override
11    protected void onSetUp() {
12        // to implement
13    }
14
15 }
```

In the overridden setup method `onSetUp`, which is automatically called by the superclass before each test, we need to create an AssertJ Swing fixture to handle our Frame. The following shows the standard setup code for AssertJ Swing. In this setup code, we create our `Student SwingView` and store it into a field. The AssertJ Swing `FrameFixture` will then be used to interact with our view's controls (labels, text fields, buttons, etc.). The `Robot` passed to the fixture's constructor is taken from the superclass. The `robot` simulates user input on a Swing Component. The use of `GuiActionRunner.execute` will be clearer later, for the moment, it is enough to know that we must create our Swing frame in the lambda passed to `execute`.

```
1 import org.assertj.swing.edt.GuiActionRunner;
2 import org.assertj.swing.fixture.FrameFixture;
3
4 @RunWith(GUITestRunner.class)
5 public class Student SwingViewTest extends AssertJSwingJUnitTestCase {
6
7     private FrameFixture window;
8
9     private Student SwingView student SwingView;
10
11    @Override
12    protected void onSetUp() {
13        GuiActionRunner.execute(() -> {
14            student SwingView = new Student SwingView();
15            return student SwingView;
16        });
17        window = new FrameFixture(robot(), student SwingView);
18        window.show(); // shows the frame to test
```

```
19     }
20
21     @Test
22     public void test() {
23         // just to check the setup works
24     }
25 }
```

Our frame and the fixture will be recreated for each test method so that we always start with a fresh user interface. The base class `AssertJ Swing JUnit TestCase` takes care of closing the windows and cleaning up resources.

Note that we store both our view and the frame fixture. We will use the former for calling (and testing) the `StudentView`'s methods and the latter for interacting with the GUI components.

We also added a fake test method just to check whether we can run this test case. Let's run it as a JUnit test. We should see the window of our view quickly show and then close.



If you have already set up a GitHub Actions workflow for this project, committing and pushing this test will make the build fail. We will deal with that in Section [Running UI tests in GitHub Actions](#).

### 13.2.1 Testing the GUI controls

The first unit test that we could write is the one that verifies that all the controls of the user interface are present; in particular, we can also verify that some controls are enabled by default (e.g., the text fields), and that others are disabled by default (e.g., the buttons).

Now we start using the fixture to test the GUI. The fixture allows us to find controls in our GUI and then call the API to simulate a user interacting with the GUI. This way, we verify that our GUI behaves as we expect.

The fixture API lookup by default finds components by name. This requires us to assign with the Swing API method `Component.setName` explicit names to our controls. Alternatively, we can use custom matchers or the default matchers, such as, `JButtonMatcher`, `JLabelMatcher`, to lookup components by their text or by other criteria. AssertJ Swing API is meant to be fluent and readable. As usual, content assist is your friend when you call these API methods.

```
1 @Test @GUIUnitTest
2 public void testControlsInitialStates() {
3     window.label(JLabelMatcher.withText("id"));
4     window.textBox("idTextBox").requireEnabled();
5 }
```

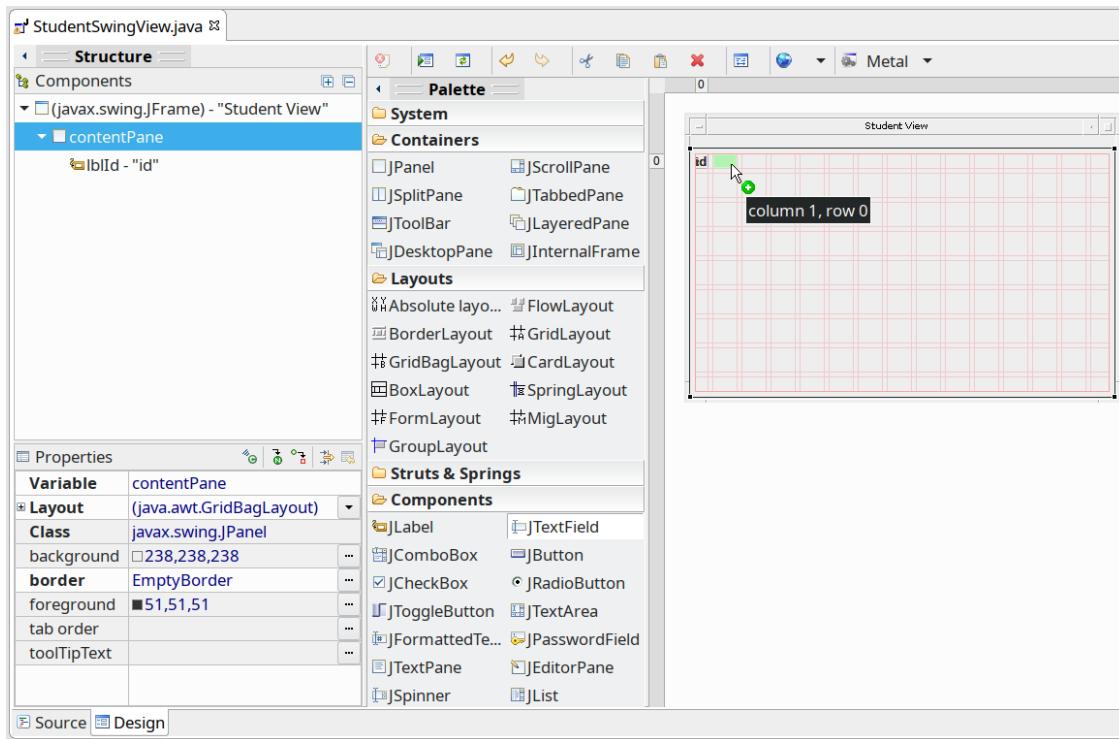


With `@GUIUnitTest` you tell AssertJ Swing to take a screenshot of the desktop when a JUnit GUI test fails. Screenshots of failed tests will be saved in the directory `failed-gui-tests` (in the directory where tests are executed).

The test fails since our frame does not contain any control yet. The trace of the failure shows enough information about the cause. It also shows the hierarchy of components in our empty frame:

```
1 org.assertj.swing.exception.ComponentLookupException: Unable to find component using\
2   matcher org.assertj.swing.core.matcher.JLabelMatcher[name=<Any>, text='id', require\
3   Showing=false].
4
5 Component hierarchy:
6 com.examples.school.view.swing.StudentSwipeView[name='frame0', title='Student View', \
7   enabled=true, visible=true, showing=true]
8   javax.swing.JRootPane[]
9     javax.swing.JPanel[name='null.glassPane']
10    javax.swing.JLayeredPane[]
11      javax.swing.JPanel[name=null]
```

Let's make this test pass by adding a label with the text "id" and a text field with name "idTextBox". We could write Java code directly or use WindowBuilder. The following screenshot shows WindowBuilder in action: we set the main layout of our frame to `GridBagLayout`, we have already added the label and now we place the text field.



#### Adding a JTextField with WindowBuilder

In case, we can rename the variables of the controls (WindowBuilder automatically gives name to Swing controls created, according to the type of the control, the text, etc.). By default, WindowBuilder creates fields in the Java class for the editing controls like text fields and local variables for controls such as labels. The name can be assigned to a control by using the `name` property, once “Show advanced properties” is selected in the “Properties” view.



Variables and fields can be renamed afterward directly from the Java editor. WindowBuilder always re-interprets the manually modified Java code.

The relevant Java parts (generated by WindowBuilder) are:

```

1 JLabel lblId = new JLabel("id");
2 contentPane.add(lblId, ...);
3
4 JTextField txtId = new JTextField();
5 txtId.setName("idTextBox");
6 contentPane.add(txtId, ...);

```

The test now passes.

Now we go on with the other Swing controls, by first writing the tests. Instead of proceeding by little steps, we write the whole test right away:

```
1 @Test @GUITest
2 public void testControlsInitialStates() {
3     window.label(JLabelMatcher.withText("id"));
4     window.textBox("idTextBox").requireEnabled();
5     window.label(JLabelMatcher.withText("name"));
6     window.textBox("nameTextBox").requireEnabled();
7     window.button(JButtonMatcher.withText("Add")).requireDisabled();
8     window.list("studentList");
9     window.button(JButtonMatcher.withText("Delete Selected")).requireDisabled();
10    window.label("errorMessageLabel").requireText(" ");
11 }
```

We make it pass by implementing the whole user interface of our frame. Note that controls are “enabled” by default. When we create the buttons we must make sure to set their `enabled` property to `false`, as requested by our test. The test for the label containing possible errors checks that it contains a single space (not that it is empty). Using a single space for labels that can be empty is a trick to make sure that the space for the label is always reserved from the very beginning of the application.



Even if this single test performs several assertions, it makes sense to group them in a single test: this test only aims at verifying the UI structure of our frame.

There are things that this test does not verify. For example, it is usual to have a `JList` inside a `JScrollPane` so that it is scrollable. Our test does not assert this. Similarly, it does not verify that the color of the error label is red. These properties are internal details that are not worthwhile to test. In general, UI tests do not necessarily test the positioning of the controls, neither they test the “beauty” of the final GUI. This would require complex inspection of the control hierarchy in a Swing component. Whether the final GUI looks nice has still to be manually checked, as anticipated in the introduction of this chapter.

In our frame, the Add button is disabled by default, and it should be enabled only after the user entered an id and a name in the text fields. Indeed, we want to avoid passing the controller and invalid `Student` object (i.e., empty id or empty name). Let’s express this requirement with this test:

```
1 @Test
2 public void testWhenIdAndNameAreNonEmptyThenAddButtonShouldBeEnabled() {
3     window.textBox("idTextBox").enterText("1");
4     window.textBox("nameTextBox").enterText("test");
5     window.button(JButtonMatcher.withText("Add")).requireEnabled();
6 }
```

This test fails as expected since we never change the state of the button. Note that `enterText` simulates the user entering some text in the text field. We have to implement this functionality by intercepting the event of the user typing some keys. We could do that with a Swing `DocumentListener`,

but for our goal, it is enough to intercept the event representing the user releasing a key after pressing a key in the text field. We can use WindowBuilder to create such an event listener on the text fields, with the context menu **Add event handler → key → keyReleased**. This will create the corresponding Java code. We do that first on the `txtId` field:

```
1 txtId.addKeyListener(new KeyAdapter() {
2     @Override
3     public void keyReleased(KeyEvent e) {
4         btnAdd.setEnabled(
5             !txtId.getText().isEmpty() &&
6             !txtName.getText().isEmpty()
7         );
8     }
9 });
```



Since we intercept the keyboard events in our frame, we must use the AssertJ Swing method `enterText` that simulates the user typing with the keyboard. The method `setText` would not work since it would directly set the text field contents, and our `KeyListener` would not intercept such events. If our frame considered also the possibility of the text field's contents to be set programmatically, then we should change our logic for intercepting the events (e.g., with the aforementioned document listener), and we should also change our tests by using `setText`.

Then we extract the anonymous class expression into a local variable that we set also on the `txtName`:

```
1 KeyAdapter btnAddEnabler = new KeyAdapter() {
2     @Override
3     public void keyReleased(KeyEvent e) {
4         btnAdd.setEnabled(
5             !txtId.getText().isEmpty() &&
6             !txtName.getText().isEmpty()
7         );
8     }
9 };
10 txtId.addKeyListener(btnAddEnabler);
11 ...
12 txtName.addKeyListener(btnAddEnabler);
```

The test now passes.

We actually want the button to be disabled even when the entered values only contain spaces. We express this requirement with this test (note the strings with a single space):

```

1  @Test
2  public void testWhenEitherIdOrNameAreBlankThenAddButtonShouldBeDisabled() {
3      JTextComponentFixture idTextBox = window.textBox("idTextBox");
4      JTextComponentFixture nameTextBox = window.textBox("nameTextBox");
5
6      idTextBox.enterText("1");
7      nameTextBox.enterText(" ");
8      window.button(JButtonMatcher.withText("Add")).requireDisabled();
9
10     idTextBox.setText("");
11     nameTextBox.setText("");
12
13     idTextBox.enterText(" ");
14     nameTextBox.enterText("test");
15     window.button(JButtonMatcher.withText("Add")).requireDisabled();
16 }

```

This fails because our implementation does not check for blank strings. We fix this by using `trim`:

```

1 KeyAdapter btnAddEnabler = new KeyAdapter() {
2     @Override
3     public void keyReleased(KeyEvent e) {
4         btnAdd.setEnabled(
5             !txtId.getText().trim().isEmpty() &&
6             !txtName.getText().trim().isEmpty()
7         );
8     }
9 };

```

All tests pass now.



## Don't interact with the computer

While running GUI tests with a testing framework like AssertJ Swing, you should not interact with the computer: don't press any key, don't use the mouse, etc. Otherwise, you will disturb the "robot" that simulates user interactions during the tests, leading to possible test failures. Even pressing `Caps Lock` might make your tests fail: `enterText("test")` will insert `TEST`, making the subsequent assertions fail.

Now we want to test that the "Delete Selected" button is enabled only when an element (i.e., a Student) is selected in the list. In this simple example, the `JList` is configured with the selection mode `ListSelectionModel.SINGLE_SELECTION` since we can only delete a single student.

To write this test, we must not use another method of the view to add elements to the list. As we have already seen, this should be avoided in unit tests. Moreover, we still haven't implemented such functionality in our view. On the contrary, we must manually add objects to the list.

To do that, we first configure the `JList` with a *list model*. We will create a default list model for storing our `Student` object. This is part of the MVC architecture of Swing itself (not related to the MVC architecture of our example). We will use the list model to store and remove `Student` objects. The `JList` graphical control will then automatically repaint itself with the new contents. We will need to access the list model from our tests, as we will see in a minute, thus we also create a package-private getter method. Note that the list model is a generic class. In our example, it will be a `DefaultListModel<Student>`. The `JList` will be also parameterized accordingly:

```
1 import javax.swing.DefaultListModel;
2 ...
3     private JList<Student> listStudents;
4     private DefaultListModel<Student> listStudentsModel;
5 ...
6     DefaultListModel<Student> getListStudentsModel() {
7         return listStudentsModel;
8     }
9 ...
10    public StudentSwipeView() {
11        ...
12        listStudentsModel = new DefaultListModel<>();
13        listStudents = new JList<>(listStudentsModel);
14        ...
```

Now we write our test: we manually add an object to the list, by using its underlying list model, then we select that element (using AssertJ Swing API) and we verify the enabled state of the button. Similarly, when there is no selection (for simplicity we test both situations in a single test):

```
1 @Test
2 public void testDeleteButtonShouldBeEnabledOnlyWhenAStudentIsSelected() {
3     studentSwipeView.getListStudentsModel().addElement(new Student("1", "test"));
4     window.list("studentList").selectItem(0);
5     JButtonFixture deleteButton =
6         window.button(JButtonMatcher.withText("Delete Selected"));
7     deleteButton.requireEnabled();
8     window.list("studentList").clearSelection();
9     deleteButton.requireDisabled();
10 }
```

This test fails with an error, unrelated to the fact that we still haven't implemented this feature:

```
1 org.assertj.swing.exception.EdtViolationException: EDT violation detected
2 ...
3     at javax.swing.DefaultListModel.addElement(DefaultListModel.java:367)
4     at com.examples.school.view.swing.Student SwingViewTest.testDeleteButtonShouldBeEn\
5 abledOnlyWhenAStudentIsSelected(Student SwingViewTest.java:72)
```

That is because AssertJ Swing checks that all operations on Swing components are executed on the special **Event Dispatch Thread (EDT)**, <https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html>. Our operation for adding an element to the list model violates this. AssertJ Swing provides the utility method (that we have already used in the setup) `GuiActionRunner.execute` that executes a lambda in the EDT. Note that this method waits until the lambda has finished its execution. This is useful since we don't have to manually write synchronization operations.

```
1 @Test
2 public void testDeleteButtonShouldBeEnabledOnlyWhenAStudentIsSelected() {
3     GuiActionRunner.execute(() ->
4         student SwingView.getListStudentsModel().addElement(new Student("1", "test")));
5     window.list("studentList").selectItem(0);
6     JButtonFixture deleteButton =
7         window.button(JButtonMatcher.withText("Delete Selected"));
8     deleteButton.requireEnabled();
9     window.list("studentList").clearSelection();
10    deleteButton.requireDisabled();
11 }
```



Of course, all AssertJ Swing API methods to access Swing components are already EDT-safe.

Now the test fails because we still haven't implemented this feature.

It is enough to implement a selection listener on the list (again, use WindowBuilder to create the initial Java code for the listener):

```
1 listStudents.addListSelectionListener(new ListSelectionListener() {
2     @Override
3     public void valueChanged(ListSelectionEvent e) {
4         btnDeleteSelected.setEnabled(listStudents.getSelectedIndex() != -1);
5     }
6 });
```

The test now passes. We can now refactor by converting the anonymous class to a lambda (using the Eclipse Quick Assist, **Ctrl + 1**).

## 13.2.2 Implementing the StudentView interface

We now write tests and implement the methods of the `StudentView` interface without having to deal with the controller. We just need to make sure that these methods correctly show something on the GUI controls (recall that they will be called from the outside, i.e., from the controller).

Let's start:

```
1  @Test
2  public void testsShowAllStudentsShouldAddStudentDescriptionsToTheList() {
3      Student student1 = new Student("1", "test1");
4      Student student2 = new Student("2", "test2");
5      GuiActionRunner.execute(() ->
6          student Swing View.showAllStudents(Arrays.asList(student1, student2))
7      );
8      String[] listContents = window.list().contents();
9      assertThat(listContents)
10         .containsExactly(student1.toString(), student2.toString());
11 }
```

Note that since the view methods we are testing are meant to do something with the Swing controls, we make sure they are executed on the EDT. The test fails. Let's fix it by implementing `showAllStudents`:

```
1  @Override
2  public void showAllStudents(List<Student> students) {
3      students.stream().forEach(listStudentsModel::addElement);
4  }
```

The test now passes.

These are the test for `showError` and its implementation:

```
1  @Test
2  public void testShowErrorShouldShowTheMessageInTheErrorMessageLabel() {
3      Student student = new Student("1", "test1");
4      GuiActionRunner.execute(
5          () -> student Swing View.showError("error message", student)
6      );
7      window.label("errorMessageLabel")
8          .requireText("error message: " + student);
9  }
```

```
1 @Override
2 public void showError(String message, Student student) {
3     lblErrorMessage.setText(message + ": " + student);
4 }
```

We follow a similar approach for `studentAdded` and `studentRemoved`. Besides updating the list, we expect these methods to reset the error label, which is meant to show only the error of the last operation. Thus, we verify the whole behavior of these methods in their tests:

```
1 @Test
2 public void testStudentAddedShouldAddTheStudentToTheListAndResetTheErrorLabel() {
3     Student student = new Student("1", "test1");
4     GuiActionRunner.execute(
5         () ->
6             student SwingView.studentAdded(new Student("1", "test1"))
7     );
8     String[] listContents = window.list().contents();
9     assertThat(listContents).containsExactly(student.toString());
10    window.label("errorMessageLabel").requireText(" ");
11 }
12
13 @Test
14 public void testStudentRemovedShouldRemoveTheStudentFromTheListAndResetTheErrorLabel\
15 () {
16     // setup
17     Student student1 = new Student("1", "test1");
18     Student student2 = new Student("2", "test2");
19     GuiActionRunner.execute(
20         () -> {
21             DefaultListModel<Student> listStudentsModel =
22                 student SwingView.getListStudentsModel();
23             listStudentsModel.addElement(student1);
24             listStudentsModel.addElement(student2);
25         }
26     );
27     // execute
28     GuiActionRunner.execute(
29         () ->
30             student SwingView.studentRemoved(new Student("1", "test1"))
31     );
32     // verify
33     String[] listContents = window.list().contents();
34     assertThat(listContents).containsExactly(student2.toString());
```

```
35     window.label("errorMessageLabel").requireText(" ");
36 }
```

Note that in the test for the removal, we create a brand new object to pass to `studentRemoved`. Removal from a list model relies on the `equals` method implementation. `Student` implements `equals` and it is not something that the frame should worry about. However, it does no harm to be pedantic in the test. Moreover, we make a little bit more explicit that the object passed to `studentRemoved` does not have to be the same object stored in the list model. Indeed, when dealing with databases, we cannot rely on object identities.

These are the implementations to make these tests pass

```
1 @Override
2 public void studentAdded(Student student) {
3     listStudentsModel.addElement(student);
4     resetErrorLabel();
5 }
6
7 @Override
8 public void studentRemoved(Student student) {
9     listStudentsModel.removeElement(student);
10    resetErrorLabel();
11 }
12
13 private void resetErrorLabel() {
14     lblErrorMessage.setText(" ");
15 }
```

### 13.2.3 Unit tests for the UI frame's logic

Now we have to implement and verify the interactions with the `SchoolController`. Since we are writing unit tests, we mock such a controller. Instead of passing it to the constructor of our view, we create a set method. This will be necessary when we write integration tests: when we instantiate the actual controller we have to pass the view to its constructor, thus we cannot pass the controller to the view's constructor, since it is not yet created: we will first create the view, create the controller passing the view to its constructor, and then set the controller in the view.

As usual, we first write the test and then add the setter and field using Eclipse quickfixes.

```
1 public class Student SwingView extends JFrame implements StudentView {  
2 ...  
3     private SchoolController schoolController;  
4 ...  
5     public void setSchoolController(SchoolController schoolController) {  
6         this.schoolController = schoolController;  
7     }  
8 ...  
  
1 public class Student SwingViewTest extends AssertJSwingJUnitTestCase {  
2 ...  
3     @Mock  
4     private SchoolController schoolController;  
5 ...  
6     private AutoCloseable closeable;  
7 ...  
8     @Override  
9     protected void onSetUp() {  
10         closeable = MockitoAnnotations.openMocks(this);  
11         GuiActionRunner.execute(() -> {  
12             student SwingView = new Student SwingView();  
13             student SwingView.setSchoolController(schoolController);  
14             return student SwingView;  
15         });  
16         window = new FrameFixture(robot(), student SwingView);  
17         window.show(); // shows the frame to test  
18     }  
19 ...  
20     @Override  
21     protected void onTearDown() throws Exception {  
22         closeable.close();  
23     }  
24 ...
```

The overridden `onTearDown` method is automatically called by the superclass after each test method.

We need to verify that our view delegates the insertion and removal of a student to the controller, by calling `newStudent` and `deleteStudent`, respectively. The controller's `newStudent` method should be called with a `Student` object created with the values entered in the two text fields. The controller's `deleteStudent` method should be called with the `Student` selected in the list. We have already tested that the buttons are enabled only when it makes sense to call the controller's methods.

Should we also verify that when we try to add a student with an existing id an error is shown in the error label? That is part of the controller's logic, so we don't test it in the unit tests of our frame.

When implementing the view we could ignore that in case of errors the controller will call our showError method. All we need to care about, when unit testing the frame, is that it implements the StudentView methods and that it does so by satisfying the expected behavior of the frame itself, not caring about the behavior of the controller. After all, we created these several layers and interfaces just to have loosely coupled components. Unit tests should respect the components' boundaries and care of the SUT only.

These are the tests for the interaction between our buttons and the controllers:

```
1  @Test
2  public void testAddButtonShouldDelegateToSchoolControllerNewStudent() {
3      window.textBox("idTextBox").enterText("1");
4      window.textBox("nameTextBox").enterText("test");
5      window.button(JButtonMatcher.withText("Add")).click();
6      verify(schoolController).newStudent(new Student("1", "test"));
7  }
8
9  @Test
10 public void testDeleteButtonShouldDelegateToSchoolControllerDeleteStudent() {
11     Student student1 = new Student("1", "test1");
12     Student student2 = new Student("2", "test2");
13     GuiActionRunner.execute(
14         () -> {
15             DefaultListModel<Student> listStudentsModel =
16                 student SwingView.getListStudentsModel();
17             listStudentsModel.addElement(student1);
18             listStudentsModel.addElement(student2);
19         }
20     );
21     window.list("studentList").selectItem(1);
22     window.button(JButtonMatcher.withText("Delete Selected")).click();
23     verify(schoolController).deleteStudent(student2);
24 }
```

These can be made pass by implementing the ActionListeners for the buttons with these lambdas:

```
1 ...
2 btnAdd.addActionListener(
3     e -> schoolController.newStudent(new Student(txtId.getText(), txtName.getText()))
4 );
5 ...
6 btnDeleteSelected.addActionListener(
7     e -> schoolController.deleteStudent(listStudents.getSelectedValue())
8 );
```

All tests pass.

We completed the implementation of our frame, fully covered by our unit tests. If the `main` method is still in our frame class, of course, that method is not covered.

## 13.3 Running UI tests in GitHub Actions

If we try to run AssertJ Swing tests in a Linux workflow in GitHub Actions with a configuration similar to the ones seen in the previous chapters, we will see they fail with such an exception:

```
1 Caused by: java.awt.HeadlessException:
2 No X11 DISPLAY variable was set,
3 but this program performed an operation which requires it.
4     at sun.java2d.HeadlessGraphicsEnvironment.getScreenDevices...
5     at org.assertj.swing.util.RobotFactory.newRobotInLeftScreen...
6     at org.assertj.swing.monitor.WindowStatus.<init>...
```

Indeed, AssertJ Swing tests need to run a graphical application, thus, in Linux, they need a graphical environment (the `X11` display server). In the Linux virtual environment provided by GitHub Actions, there's no such a graphical environment. We can then start a virtual X display server with `Xvfb` (X virtual framebuffer). This is a display server implementing the `X11` display server protocol, performing all graphical operations in virtual memory without showing any screen output. This virtual server does not require the computer it is running on to have any kind of screen or any input device.

The Linux virtual environment provided by GitHub Actions already provides `Xvfb`. In particular, it also offers the program `xvfb-run`, which runs a command in such a virtual X server environment. Thus, it is just a matter of changing the command for running the Maven build as follows:

```
1 ...
2 - name: Build with Maven
3   run: xvfb-run mvn verify
4   working-directory: com.examples.school
```

Now AssertJ Swing tests will be able to start the graphical application and will be able to run successfully.

The use of Xvfb is required only in the Linux virtual environment. In macOS and Windows virtual environments, a graphical environment is already provided and AssertJ Swing tests succeed right away.

In case your AssertJ Swing tests need to perform involved operations on the windows of the GUI under test, like, e.g., maximizing or minimizing a window, Xvfb might not be enough, and a more involved graphical environment configuration might be required. You can find directions in the AssertJ Swing official documentation: <https://joel-costigliola.github.io/assertj/assertj-swing-running.html>.

## 13.4 UI integration tests

Now we write integration tests for our GUI frame. These integration tests will use the `StudentSwipeView` together with a real `SchoolController` and a real `SchoolMongoRepository`. We can still use an in-memory database, since, for verifying the correct integration of the view and the controller it is not strictly required to use a real database.

These integration tests will allow us to verify that our GUI behaves correctly concerning the real controller's logic. Now that we are using a real controller we can verify scenarios for the GUI that we couldn't verify in the unit tests (and that we were not supposed to verify in the unit tests). For example,

1. an id and a name are inserted in the text fields
2. the “Add” button is clicked
3. the values of the added student should appear in the list

The “Add” button delegates to the controller's `newStudent` method (as already tested in the previous unit tests) and the controller, after adding the record in the database, calls the view's method `studentAdded`. Thus, the above scenario for our GUI can only be tested with integration tests. Indeed, it is a behavior that results from the interaction between the view and the controller.

The setup of our integration tests is as follows. It is a mixture of what we saw in Chapter [Integration tests](#), Section [Unit tests with databases](#) and in this chapter, Section [UI unit tests](#).

```
1  @RunWith(GUIRunner.class)
2  public class StudentSwingViewIT extends AssertJSwingJUnitTestCase {
3      private static MongoServer server;
4      private static InetSocketAddress serverAddress;
5
6      private MongoClient mongoClient;
7
8      private FrameFixture window;
9      private StudentSwingView studentSwingView;
10     private SchoolController schoolController;
11     private StudentMongoRepository studentRepository;
12
13     @BeforeClass
14     public static void setupServer() {
15         server = new MongoServer(new MemoryBackend());
16         // bind on a random local port
17         serverAddress = server.bind();
18     }
19
20     @AfterClass
21     public static void shutdownServer() {
22         server.shutdown();
23     }
24
25     @Override
26     protected void onSetUp() {
27         mongoClient = new MongoClient(new ServerAddress(serverAddress));
28         studentRepository = new StudentMongoRepository(mongoClient);
29         // explicit empty the database through the repository
30         for (Student student : studentRepository.findAll()) {
31             studentRepository.delete(student.getId());
32         }
33         GuiActionRunner.execute(() -> {
34             studentSwingView = new StudentSwingView();
35             schoolController =
36                 new SchoolController(studentSwingView, studentRepository);
37             studentSwingView.setSchoolController(schoolController);
38             return studentSwingView;
39         });
40         window = new FrameFixture(robot(), studentSwingView);
41         window.show(); // shows the frame to test
42     }
43 }
```

```
44     @Override
45     protected void onTearDown() {
46         mongoClient.close();
47     }...
```

We keep a reference also to the controller and the repository so that we can use them to create the correct scenarios for verifying the behavior of our frame. As usual, we make sure we always execute each test with a fresh database.

First of all, we verify that when the controller's method `allStudents` is called, the frame's list is populated with the students in the database:

```
1  @Test @GUIUnitTest
2  public void testAllStudents() {
3      // use the repository to add students to the database
4      Student student1 = new Student("1", "test1");
5      Student student2 = new Student("2", "test2");
6      studentRepository.save(student1);
7      studentRepository.save(student2);
8      // use the controller's allStudents
9      GuiActionRunner.execute(
10          () -> schoolController.allStudents());
11     // and verify that the view's list is populated
12     assertThat(window.list().contents())
13         .containsExactly(student1.toString(), student2.toString());
14 }
```

We use the repository to populate the database, then we call the controller's method and verify that the frame's list is populated.

Then, we verify that the view and the controller interact correctly when a student is added through the “Add” button.

```
1  @Test @GUIUnitTest
2  public void testAddButtonSuccess() {
3      window.textBox("idTextBox").enterText("1");
4      window.textBox("nameTextBox").enterText("test");
5      window.button(JButtonMatcher.withText("Add")).click();
6      assertThat(window.list().contents())
7          .containsExactly(new Student("1", "test").toString());
8  }
9
10 @Test @GUIUnitTest
11 public void testAddButtonError() {
```

```
12 studentRepository.save(new Student("1", "existing"));
13 window.textBox("idTextBox").enterText("1");
14 window.textBox("nameTextBox").enterText("test");
15 window.button(JButtonMatcher.withText("Add")).click();
16 assertThat(window.list().contents())
17     .isEmpty();
18 window.label("errorMessageLabel")
19     .requireText("Already existing student with id 1: "
20         + new Student("1", "existing"));
21 }
```

For the scenario where a student with the same id is already present, we need to first add a student to the database through the repository. Then we verify that the student does not appear in the list and that instead an error message is shown in the view.

Finally, we verify that the view and the controller interact correctly when a student is deleted through the “Delete” button.

```
1 @Test @GUITest
2 public void testDeleteButtonSuccess() {
3     // use the controller to populate the view's list...
4     GuiActionRunner.execute(
5         () -> schoolController.newStudent(new Student("1", "toremove")));
6     // ...with a student to select
7     window.list().selectItem(0);
8     window.button(JButtonMatcher.withText("Delete Selected")).click();
9     assertThat(window.list().contents())
10        .isEmpty();
11 }
12
13 @Test @GUITest
14 public void testDeleteButtonError() {
15     // manually add a student to the list, which will not be in the db
16     Student student = new Student("1", "non existent");
17     GuiActionRunner.execute(
18         () -> student SwingView.getListStudentsModel().addElement(student));
19     window.list().selectItem(0);
20     window.button(JButtonMatcher.withText("Delete Selected")).click();
21     assertThat(window.list().contents())
22         .containsExactly(student.toString());
23     window.label("errorMessageLabel")
24         .requireText("No existing student with id 1: " + student);
25 }
```

Note that in the first test we use the controller to add a student both to the database and to the view's list. In the second test, we have to simulate the case when a student is deleted from the view that does not exist in the database. To recreate such a scenario we manually add a student to the view list, but not to the database.

All these integration tests succeed as expected: we had previously tested the view, the controller, and the repository in unit tests in isolation. It makes sense that the integration tests succeed. Indeed, they should just verify the correct behavior of already unit tested components when composed together.

Note that it does not make sense to write integration tests for scenarios that only have to do with the view itself. For example, it does not make sense to write integration tests for the enabling of buttons: we have already tested these situations in the unit tests of the view, since they are not related to external dependencies of the view. Thus, as seen before, the number of integration tests is smaller than the number of unit tests.

In the integration test implemented above, the test fixture consists of all the components of our program, however, the SUT can still be considered the Swing view. Indeed, we verify its behavior when integrated with other real components of our program.

We could also write a kind of different integration test, where we verify that actions performed on the Swing view will lead to changes to the state of the database. We will write only two tests, one verifying the addition of a student and the other one verifying the deletion. Again, this integration test is a combination of the techniques already seen so far. In this particular test, we will use Testcontainers to start a real MongoDB server. We call such a test `ModelViewControllerIT`, since, as we will explain in a minute, it verifies the correct behavior of our MVC architecture:

```
1  @RunWith(GUIRunner.class)
2  public class ModelViewControllerIT extends AssertJSwingJUnitTestCase {
3      @ClassRule
4      public static final MongoDBContainer mongo =
5          new MongoDBContainer("mongo:4.4.3");
6
7      private MongoClient mongoClient;
8
9      private FrameFixture window;
10     private SchoolController schoolController;
11     private StudentMongoRepository studentRepository;
12
13     @Override
14     protected void onSetUp() {
15         mongoClient = new MongoClient(
16             new ServerAddress(
17                 mongo.getContainerIpAddress(),
18                 mongo.getFirstMappedPort()));
19         studentRepository = new StudentMongoRepository(mongoClient);
```

```
20      // explicit empty the database through the repository
21      for (Student student : studentRepository.findAll()) {
22          studentRepository.delete(student.getId());
23      }
24      window = new FrameFixture(robot(), GuiActionRunner.execute(() -> {
25          Student SwingView student SwingView = new Student SwingView();
26          schoolController =
27              new SchoolController(student SwingView, studentRepository);
28          student SwingView.setSchoolController(schoolController);
29          return student SwingView;
30      }));
31      window.show(); // shows the frame to test
32  }
33
34  @Override
35  protected void onTearDown() {
36      mongoClient.close();
37  }
38
39  @Test
40  public void testAddStudent() {
41      // use the UI to add a student...
42      window.textBox("idTextBox").enterText("1");
43      window.textBox("nameTextBox").enterText("test");
44      window.button(JButtonMatcher.withText("Add")).click();
45      // ...verify that it has been added to the database
46      assertThat(studentRepository.findById("1"))
47          .isEqualTo(new Student("1", "test"));
48  }
49
50  @Test
51  public void testDeleteStudent() {
52      // add a student needed for tests
53      studentRepository.save(new Student("99", "existing"));
54      // use the controller's allStudents to make the student
55      // appear in the GUI list
56      GuiActionRunner.execute(
57          () -> schoolController.allStudents());
58      // ...select the existing student
59      window.list().selectItem(0);
60      window.button(JButtonMatcher.withText("Delete Selected")).click();
61      // verify that the student has been deleted from the db
62      assertThat(studentRepository.findById("99"))
```

```
63     .isNull());  
64 }  
65  
66 }
```

In this integration test, the SUT can be seen as the combination of the view, the controller and the repository, since we verify the behaviors of all these components, when connected. We perform actions on the view through AssertJ Swing and the repository, and we perform verifications through the repository. Note that we do not keep a reference to the Swing view, since we interact with that only through AssertJ Swing.

This is quite similar to an end-to-end test. However, as we will see in Chapter [End-to-end tests](#), in e2e tests we usually interact only with the user interface of the application, while here we still use the components of the application. It is similar to an e2e test since we interact with the GUI. However, verifications are still performed by accessing the database through our repository component.

## 13.5 Multithreading

Up to now, our tests have always been deterministic. This was made easy by the fact that all our code is based on a single thread. When an application has to deal with multithreaded code, testing becomes harder and a few tests will become non-deterministic.

Of course, it is crucial to be able to isolate the concurrent parts of an application as much as possible and make sure to first implement and test the logic that is wrapped by the concurrent constructs.

However, at some point, we will have to deal with tests that involve concurrent threads.

Multithreading is somehow implicit in GUI applications, like the one we have developed in this chapter. Window toolkits, like Swing, typically handle user events in a separate concurrent thread, the Event Dispatch Thread (EDT). We dealt with EDT in Section [Testing the GUI controls](#). AssertJ Swing already handles synchronizations with this thread automatically, keeping tests deterministic as much as possible.

In our application, we never used multithreading ourselves, and this relieved us from problems in testing this application.

However, GUI applications should introduce multithreading explicitly at least in a few parts. For example, we know that all operations concerning GUI controls must take place in the EDT in Swing. When something is executed on the EDT, the Swing GUI is not responsive. For this reason, it is best practice to handle user events, which are known to possibly be long-running operations that might be blocking the UI, into a parallel thread.

In our application, when we click on the “Add” button, we delegate to the controller the insertion of a student in the database. Database operations might take some time. In the current implementation, during the database operation, the UI is not responsive. We could then call the controller method newStudent in a new parallel thread.

Before doing that, let's recall that the controller will then call the methods of the `StudentView` interface. If the controller's methods are called in another thread, then the `StudentView` methods interacting with the GUI controls, such as, `studentAdded` and `showError`, will be executed in a thread different from the EDT, leading to the problems we have already experimented. If we plan to call the controller in a separate thread, we have to implement the `StudentView` methods so that they are executed in the EDT.

First of all, let's recreate such a situation in our unit tests. In Section [\*Implementing the StudentView interface\*](#) we wrapped the call to `student SwingView.studentAdded` in a lambda executed through `GuiActionRunner.execute`, to avoid the `EdtViolationException`. Let's remove such a wrap and the test fails:

```

1  @Test
2  public void testStudentAddedShouldAddTheStudentToTheListAndResetTheErrorLabel() {
3      Student student = new Student("1", "test1");
4      student SwingView.studentAdded(new Student("1", "test1"));
5      String[] listContents = window.list().contents();
6      assertThat(listContents).containsExactly(student.toString());
7      window.label("errorMessageLabel").requireText(" ");
8  }

```

We now have to fix `studentAdded` so that its code (at least the parts interacting with Swing controls) is executed on the EDT. Swing provides a utility method to execute something in the EDT:  
`javax.swing.SwingUtilities.invokeLater(Runnable)`:

```

1  @Override
2  public void studentAdded(Student student) {
3      SwingUtilities.invokeLater(() -> {
4          listStudentsModel.addElement(student);
5          resetErrorLabel();
6      });
7  }

```

Now the test is back to green.

It is legal to call `student SwingView.studentAdded` directly from the test since its code will be executed in the EDT. (recall that the test code is executed in a separate thread, not on the EDT: it is AssertJ Swing that uses the EDT when interacting with the fixture).

We follow the same strategy for `showError`:

```

1  @Test
2  public void testShowErrorShouldShowTheMessageInTheErrorMessageLabel() {
3      Student student = new Student("1", "test1");
4      studentSwingView.showError("error message", student);
5      window.label("errorMessageLabel")
6          .requireText("error message: " + student);
7  }

1  @Override
2  public void showError(String message, Student student) {
3      SwingUtilities.invokeLater(() ->
4          lblErrorMessage.setText(message + ": " + student)
5      );
6  }

```

We can now safely use the controller in a new thread when implementing the event handler for the “Add” button (when the controller calls the view `studentAdded` its code will be executed in the EDT):

```

1  btnAdd.addActionListener(
2      e -> new Thread(() ->
3          schoolController.newStudent(
4              new Student(txtId.getText(), txtName.getText()))
5      ).start()
6  );

```

Our tests are still green... but are they still be reliable in the current form?

For example, let's take this test

```

1  @Test
2  public void testAddButtonShouldDelegateToSchoolControllerNewStudent() {
3      window.textBox("idTextBox").enterText("1");
4      window.textBox("nameTextBox").enterText("test");
5      window.button(JButtonMatcher.withText("Add")).click();
6      // schoolController.newStudent executed in a parallel thread now
7      verify(schoolController).newStudent(new Student("1", "test"));
8  }

```

What happens if the local machine is somehow overloaded with other processes and the thread executing `schoolController.newStudent` is not scheduled in time to fulfill the Mockito verification? The test will fail. However, it will not always fail. It depends on the load of the machine executing the tests. This becomes a **flaky test**, which non-deterministically fails/passes.

Let's try to recreate such a situation, e.g., by forcing a delay in the thread. Even 1 second delay is enough to our aim:

```
1 btnAdd.addActionListener(  
2     e -> new Thread(() ->  
3     {  
4         try {  
5             Thread.sleep(1000);  
6         } catch (InterruptedException e1) {  
7             }  
8         schoolController.newStudent(  
9             new Student(txtId.getText(), txtName.getText()));  
10    }  
11 ).start()  
12 );
```

The above test now fails.

When possible, these issues should be solved by adding synchronization mechanisms in the tests. In our example, however, this would require changing the main code just for testing purposes, thus we avoid it.

In our case, what we can do, is perform verifications using timeouts. The idea is that if verification fails, we should try to run it again a few more times. If it still fails after a given timeout, we consider the verification effectively failed.

Fortunately, Mockito provides verification mechanisms based on a timeout that automatically performs the above strategy.

```
1 private static final int TIMEOUT = 5000;  
2 ...  
3 @Test  
4 public void testAddButtonShouldDelegateToSchoolControllerNewStudent() {  
5     window.textBox("idTextBox").enterText("1");  
6     window.textBox("nameTextBox").enterText("test");  
7     window.button(JButtonMatcher.withText("Add")).click();  
8     verify(schoolController, timeout(TIMEOUT))  
9         .newStudent(new Student("1", "test"));  
10 }
```

The test now succeeds.

Note that a timeout does not make the test deterministic. It only makes it less likely to fail due to parallel threads issues. On the other hand, choosing a very big timeout has the drawback that, if the test must fail, it will fail only after the timeout is exceeded.

Now also integration tests related to the “Add” button will fail, due to our manual delay added in the parallel thread.

Let’s start with `StudentSwingViewIT`. For example, this test will now fail:

```
1 @Test @GUITest
2 public void testAddButtonError() {
3     studentRepository.save(new Student("1", "existing"));
4     window.textBox("idTextBox").enterText("1");
5     window.textBox("nameTextBox").enterText("test");
6     window.button(JButtonMatcher.withText("Add")).click();
7     assertThat(window.list().contents())
8         .isEmpty();
9     window.label("errorMessageLabel")
10        .requireText("Already existing student with id 1: "
11            + new Student("1", "existing"));
12 }
```

In fact, the error label is not updated on time.

AssertJ Swing provides mechanisms for testing with timeouts as well. In particular, we can pause until a specific condition is satisfied, within a given timeout. In the above test, we can wait until the error label contains some text, and then proceed with the previous verifications:

```
1 import static org.assertj.swing.timing.Pause.pause;
2 import static org.assertj.swing.timing.Timeout.timeout;
3 import org.assertj.swing.timing.Condition;
4 ...
5 private static final long TIMEOUT = 5000;
6 ...
7 @Test @GUITest
8 public void testAddButtonError() {
9     studentRepository.save(new Student("1", "existing"));
10    window.textBox("idTextBox").enterText("1");
11    window.textBox("nameTextBox").enterText("test");
12    window.button(JButtonMatcher.withText("Add")).click();
13    pause(
14        new Condition("Error label to contain text") {
15            @Override
16            public boolean test() {
17                return !window.label("errorMessageLabel")
18                    .text().trim().isEmpty();
19            }
20        }
21        , timeout(TIMEOUT));
22    assertThat(window.list().contents())
23        .isEmpty();
24    window.label("errorMessageLabel")
```

```
25     .requireText("Already existing student with id 1: "
26         + new Student("1", "existing"));
27 }
```

Now the test is back to green. The `pause` method above can be seen as a mechanism for forcing some form of synchronization.

The other failing integration test is this one:

```
1 @Test @GUITest
2 public void testAddButtonSuccess() {
3     window.textBox("idTextBox").enterText("1");
4     window.textBox("nameTextBox").enterText("test");
5     window.button(JButtonMatcher.withText("Add")).click();
6     assertThat(window.list().contents())
7         .containsExactly(new Student("1", "test").toString());
8 }
```

We can fix this by performing the assertion a few times until it is met or a timeout is reached.

For demonstration purposes, we use another testing framework, `Awaitility`, <https://github.com/awaitility/awaitility>, a small Java DSL for synchronizing asynchronous operations. This provides a fluent API for expressing expectations of an asynchronous system in a readable way. It also integrates nicely with `AssertJ`'s core assertions.

This is the Maven dependency:

```
1 <dependency>
2     <groupId>org.awaitility</groupId>
3     <artifactId>awaitility</artifactId>
4     <version>4.0.2</version>
5     <scope>test</scope>
6 </dependency>
```

And this is how we can use it to fix the above test:

```
1 import static org.awaitility.Awaitility.*;
2 import java.util.concurrent.TimeUnit;
3 ...
4 @Test @GUITest
5 public void testAddButtonSuccess() {
6     window.textBox("idTextBox").enterText("1");
7     window.textBox("nameTextBox").enterText("test");
8     window.button(JButtonMatcher.withText("Add")).click();
9     await().atMost(5, TimeUnit.SECONDS).untilAsserted(() ->
10         assertThat(window.list().contents())
11             .containsExactly(new Student("1", "test").toString())
12     );
13 }
```

Using the same approach, we also fix the corresponding integration test in `ModelViewControllerIT`. If we now have good confidence that our tests are more resistant to synchronization issues, we can remove the artificial `Thread.sleep` introduced above.



Following the same strategy, try to handle also the “Delete” button in an asynchronous thread and run in the EDT thread the code of the remaining methods of the interface `StudentView` in `Student SwingView`.

### 13.5.1 Race conditions in the application

Multithreading requires us to deal with possible race conditions in parts of the code that is executed by possible multiple concurrent threads.

In this example, the critical part is the method `newStudent` of the controller, which could be executed by several multiple threads in case the user quickly clicks on the “Add” button several times with the same student data:

```
1 public void newStudent(Student student) {
2     Student existingStudent = studentRepository.findById(student.getId());
3     if (existingStudent != null) {
4         studentView.showError("Already existing student with id " + student.getId(),
5             existingStudent);
6         return;
7     }
8
9     studentRepository.save(student);
10    studentView.studentAdded(student);
11 }
```

Between the check for an existing student and the actual insertion in the repository there might be context switches and two concurrent threads might end up inserting two students with the same id without any failure.

We can try to recreate such a race condition with a unit test, which spawns several concurrent threads that all call the method `newStudent`. Such a technique does not guarantee to recreate the race condition each time the test is run. However, chances are that the race condition takes place even with a small number of threads (10 in this case):

```
1 public class SchoolControllerRaceConditionTest {
2
3     @Mock
4     private StudentRepository studentRepository;
5
6     @Mock
7     private StudentView studentView;
8
9     @InjectMocks
10    private SchoolController schoolController;
11
12    private AutoCloseable closeable;
13
14    @Before
15    public void setUp() throws Exception {
16        closeable = MockitoAnnotations.openMocks(this);
17    }
18
19    // @After method as usual
20
21    @Test
22    public void testNewStudentConcurrent() {
23        List<Student> students = new ArrayList<>();
24        Student student = new Student("1", "name");
25        // stub the StudentRepository
26        when(studentRepository.findById(anyString()))
27            .thenAnswer(invocation -> students.stream()
28                        .findFirst().orElse(null));
29        doAnswer(invocation -> {
30            students.add(student);
31            return null;
32        }).when(studentRepository).save(any(Student.class));
33        // start the threads calling newStudent concurrently
34        List<Thread> threads = IntStream.range(0, 10)
```

```

35      .mapToObj(i -> new Thread(() -> schoolController.newStudent(student)))
36      .peek(t -> t.start())
37      .collect(Collectors.toList());
38      // wait for all the threads to finish
39      await().atMost(10, SECONDS)
40      .until(() -> threads.stream().noneMatch(t -> t.isAlive()));
41      // there should be a single element in the list
42      assertThat(students)
43      .containsExactly(student);
44  }
45
46 }
```

We mock and stub the repository so that it uses a list as a fake database (we do not need the view, but we have to mock it since the controller calls its methods). Note that we must use `thenAnswer` for stubbing: the passed lambda is executed each time the stubbed method is called by the controller. We introduce Mockito “answers” in Chapter [Mocking](#), Section [Stubbing with answers](#). To our aim, it is enough to return the first element of the list (if any) for stubbing `findById`. For stubbing `save` we follow a similar strategy, by simply adding the `Student` stored in the local variable. Since `save` is a void method, we must use the `doAnswer` mechanism for stubbing (see Chapter [Mocking](#), Section [Stubbing and exceptions](#) for details about stubbing void methods). Then we create a list of 10 threads all calling `newStudent` on the same controller and we start the threads.<sup>3</sup> After all threads have finished the execution, we check that the list (the fake database) contains only one instance of the student.

With the current implementation of `newStudent` this test is likely to fail: the list will contain more than one student.



On a fast machine, the test might also succeed now and then. Of course, increasing the number of concurrent threads increases the chances to make this test fail. In some cases, we might also get an exception in the stubbed `findById` due to a concurrent modification of the list while the stream is used. Moreover, when the test fails, the list contains more than one student, but not necessarily 10 students. This is expected and implied by the non-deterministic nature of multithreaded code.

In this example, fixing the race condition is just a matter of making the controller method `synchronized`:

```
1 public synchronized void newStudent(Student student) { ... }
```

The test now succeeds.

---

<sup>3</sup>Recall that `peek` returns a stream consisting of the elements of the stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

Of course, in more complex situations, recreating a race condition might be harder. Similarly, avoiding race conditions might be harder as well.



Apply the same strategy also for `deleteStudent`.

### 13.5.2 Race conditions in the database

In the previous section, we dealt with possible race conditions in a single instance of the application. As anticipated in Chapter [Integration tests](#), Section [Our running example](#), in the presence of several clients using the same database, we should use additional mechanisms for *primary key* and *atomic transaction* management provided by the database and its API.

Synchronizing access to the controller's methods is enough to avoid race conditions in a single application, assuming that all components use the same shared instance of the controller. However, if there are several instances of the application, then each instance will use its instance of the controller. If the instances of the application connect to the same database, then synchronized access to a controller instance is not enough to avoid race conditions at the database level. The result is that the database might end up containing several students with the same id without any failure.

Similarly to what we did in the previous section, we can try to recreate such a race condition, this time with an integration test, which assumes there's a MongoDB running (e.g., in a Docker container). This integration test uses a real `StudentMongoRepository` and spawns several concurrent threads that all call the method `newStudent` on a controller, but this time each thread uses its instance of `SchoolController`. Again, such a technique does not guarantee to recreate the race condition each time the test is run. However, chances are that the race condition takes place even with a small number of threads (10 in this case):

```
1  /**
2   * Communicates with a MongoDB server on localhost; start MongoDB with Docker with
3   * docker run -p 27017:27017 --rm mongo:4.4.3
4  */
5  public class SchoolControllerRaceConditionIT {
6      @Mock
7      private StudentView studentView;
8
9      private StudentRepository studentRepository;
10
11     private AutoCloseable closeable;
12
13     @Before
14     public void setUp() {
```

```
15     closeable = MockitoAnnotations.openMocks(this);
16     MongoClient client = new MongoClient("localhost");
17     MongoDatabase database = client.getDatabase(SCHOOL_DB_NAME);
18     // make sure we always start with a clean database
19     database.drop();
20     studentRepository = new StudentMongoRepository(client);
21 }
22
23 // @After method as usual
24
25 @Test
26 public void testNewStudentConcurrent() {
27     Student student = new Student("1", "name");
28     // start the threads calling newStudent concurrently
29     // on different SchoolController instances, so 'synchronized'
30     // methods in the controller will not help...
31     List<Thread> threads = IntStream.range(0, 10)
32         .mapToObj(i -> new Thread(
33             () ->
34                 new SchoolController(studentView, studentRepository)
35                     .newStudent(student)))
36         .peek(t -> t.start())
37         .collect(Collectors.toList());
38     // wait for all the threads to finish
39     await().atMost(10, SECONDS)
40         .until(() -> threads.stream().noneMatch(t -> t.isAlive()));
41     // there should be a single element in the list
42     assertThat(studentRepository.findAll())
43         .containsExactly(student);
44 }
45 }
```

As in the previous section, this test is likely to fail: the list returned by `findAll` contains more than one student.

To avoid such a situation we have to rely on some database mechanisms. For MongoDB, we can set a **unique index** on the collection. A unique index ensures that the indexed fields do not store duplicate values, thus, it enforces uniqueness for the indexed fields.

We modify the setup to specify that in the student collection the `id` field must contain unique value:

```

1  @Before
2  public void setUp() {
3      closeable = MockitoAnnotations.openMocks(this);
4      MongoClient client = new MongoClient("localhost");
5      MongoDatabase database = client.getDatabase(SCHOOL_DB_NAME);
6      // make sure we always start with a clean database
7      database.drop();
8      MongoCollection<Document> studentCollection =
9          database.getCollection(STUDENT_COLLECTION_NAME);
10     // A unique index ensures that the indexed field
11     // (in this case "id") does not store duplicate values:
12     studentCollection.createIndex(
13         Indexes.ascending("id"), new IndexOptions().unique(true));
14     studentRepository = new StudentMongoRepository(client);
15 }

```

With such a modification, the test now fails before getting to the assertion. In fact, when two controllers get into the race condition, the subsequent insertion of the student with duplicate id will raise an exception:

```

1 com.mongodb.MongoWriteException: E11000 duplicate key error collection:
2 school.student index: id_1 dup key: { id: "1" }

```

Thus, the index specification prevents having a database with several students with the same id.

Of course, that exception should be caught somewhere and the application user should be notified somehow. We will not do that, but we can modify the test to mimic this idea:

```

1 new Thread(
2     () -> {
3         try {
4             new SchoolController(studentView, studentRepository)
5                 .newStudent(student);
6         } catch (MongoWriteException e) {
7             // E11000 duplicate key error collection:
8             // school.student index: id_1 dup key: { id: "1" }
9             e.printStackTrace();
10        }
11    })

```

Now the test succeeds. The console will show the printed exceptions and the database will contain one single student with the specified id. Only one thread will “win the race” and manage to insert the student with id 1.

In general, exceptions like `MongoWriteException` should not be caught by the code calling the `SchoolController` methods. In fact, our components abstract from implementation details and `MongoWriteException` is specific to MongoDB. Such exceptions should be caught by the class that connects directly to the MongoDB database. In this example, a good candidate would be `StudentMongoRepository`, which should then wrap such a specific exception into an application exception (e.g., we could introduce the exception `DuplicateStudentIdException`, which is application specific and independent from the database).

The final step would be to use the MongoDB API for transactions, which would take care of dealing with such race conditions without getting to a state where a `MongoWriteException` is even thrown. The transaction mechanisms would basically serialize concurrent executions of the code of `newStudent`, leading to locking and synchronization at the database level, not only at the application level. Such locking and synchronization are handled by the MongoDB server and by the implementation of the Java MongoDB transaction API, relying on the unique index specifications. As we have already said, MongoDB transactions are out of the scope of this book.

In Chapter [Mocking](#), Section [Another example: Transactions](#), we gave some hints on a possible way to structure the transaction management through a general interface, `TransactionManager`. One could reuse such a “pattern” in this example and provide an implementation using the MongoDB transaction API. In that case, the controller should not access the repository directly: it should do that through the transaction manager.

Finally, database configurations, like the unique index we set in the above test, should be enforced at the database level from the application. For example, the `main` method (which we still haven’t implemented) could check whether we are using a brand new database and in case configure the unique index. Alternatively, we might delegate such a responsibility to the database administrator. In our tests, we might create a custom MongoDB Docker image, which configures the database with such an index. We could implement a `Dockerfile` starting from the official MongoDB image and perform such database configurations (refer to the documentation of the Docker MongoDB image for the details on how to configure the MongoDB databases and collections). Our tests should then run Docker containers using our custom MongoDB image.

## 13.6 Improvements

Several improvements can be made to this simple application. We just hint at a few of them that we leave as an exercise. These are meant to be implemented with the testing techniques seen so far.

- If we follow the approach to handle events that deal with the database in a separate thread, we should also notify the user that something is happening after a button has been pressed, e.g., by using an additional label somewhere in the GUI. Of course, the label should also be cleared when the long-running operation (executed by a parallel thread) terminates.
- If we are trying to delete a student that is not available in the database anymore, when showing the error in the label, we should also take the chance to update the list by removing the

non-existent student (we will implement this mechanism in Chapter [End-to-end tests](#), Section [Starting from the high-level specifications](#)).

- Similarly, if we try to add a student with an existing id, when showing the error, we should also update the list with such an existing student, if not already present in the list.
- After the button “Add” is clicked, we should clear the text fields in case the operation terminates successfully.
- We should provide mechanisms for editing and updating an existing student. For this task, some additional methods must be added to the existing interfaces and classes.

Note that the addition of a student with an existing id and the deletion of a student that does not exist in the database are likely to happen if there are several instances of this application, possibly on different computers, using the same MongoDB database (see also Section [Race conditions in the database](#)). In such situations, some listener mechanisms might be useful: when the database changes, clients are notified and can update their knowledge of the current state of the database. Also for such mechanisms, we should rely on the features provided by the database we are using.

Of course, in the current form, the application is far from being complete. We should add other domain model concepts such as Teacher, School, etc., and develop features, including the GUI, also for these concepts. We leave this as an exercise as well.

# 14. End-to-end tests

In this chapter, we deal with end-to-end tests. Recall from Chapter [Testing](#), Section [Automated tests](#), that in e2e tests, the whole application is tested. These tests verify that all the components interact correctly, to accomplish the main goals of the application.

We will first write e2e tests directly in Java. Then, we will use the BDD framework **Cucumber** for writing high-level specifications for our e2e tests (see Chapter [Testing](#), Section [Behavioral-Driven Development \(BDD\)](#)).

Of course, it makes no sense to test the same behaviors both with manual Java e2e tests and with Cucumber. We do that in this chapter only for showing two different techniques.

To some extent, e2e tests are a special kind of integration tests where all the system components are integrated. However, in the integration tests, we have implemented in the previous chapters, the test fixture consisted of a few instances of our classes, manually wired together, to verify their correct interactions. In e2e tests, instead, we will run verifications directly on the whole running application. As we anticipated in Chapter [Testing](#), e2e tests interact directly with the user interface of the application. In our example, e2e tests interact with the GUI using AssertJ Swing (Chapter [UI tests](#)). In previous integration tests involving the GUI, we still directly called methods on instances of our classes. In e2e tests, we will never call the methods of our classes directly. We will never even mention our Java classes explicitly.

To achieve that, we need to start our application somehow. In desktop applications, this can be achieved by adding a `main` method that creates the GUI and all the needed instances (controller, repository, etc.). As we will see, AssertJ Swing supports launching an application from its `main` method.

To create the contexts for our tests, we will still need direct access to the database. However, differently from the integration test `Student Swing View IT` of Chapter [UI tests](#), where we were using `StudentMongoRepository`, in e2e tests we will use directly a `MongoClient` instance.

## 14.1 The running example

Once again, we start from the example of the previous chapter, where we had already implemented the GUI, tested both with unit and integration tests. For simplicity, we will not consider the multithreading features we partly implemented in Chapter [UI tests](#), Section [Multithreading](#). Thus, in this chapter, we start from the version of the application we had implemented right before that section.

We first refactor the `StudentMongoRepository` so that it accepts as constructor's parameters also the name of the database and the name of the collection. This will allow us to use different databases and collections in our tests.

```
1 public StudentMongoRepository(MongoClient client,
2     String databaseName, String collectionName) {
3     studentCollection = client
4         .getDatabase(databaseName)
5         .getCollection(collectionName);
6 }
```

A few tests will have to be adjusted, passing the database name and the collection name. Try to adjust all the tests (then have a look at the source code of this example).

We extract the `main` method from the view into a separate class, `School SwingApp`. We will use the extracted `main` body to implement the instructions for starting our view with all the wired collaborators (controller, repository, etc.). This could be an initial main class:

```
1 package com.examples.school.app.swing;
2 ...
3 public class School SwingApp {
4
5     public static void main(String[] args) {
6         EventQueue.invokeLater(() -> {
7             try {
8                 String mongoHost = "localhost";
9                 int mongoPort = 27017;
10                if (args.length > 0)
11                    mongoHost = args[0];
12                if (args.length > 1)
13                    mongoPort = Integer.parseInt(args[1]);
14                StudentMongoRepository studentRepository =
15                    new StudentMongoRepository(
16                        new MongoClient(new ServerAddress(mongoHost, mongoPort)),
17                        "school", "student");
18                Student SwingView studentView = new Student SwingView();
19                SchoolController schoolController =
20                    new SchoolController(studentView, studentRepository);
21                studentView.setSchoolController(schoolController);
22                studentView.setVisible(true);
23                schoolController.allStudents();
24            } catch (Exception e) {
25                e.printStackTrace();
26            }
27        });
28    }
29 }
```

Besides the standard statements for starting a Swing application, we need to initialize the controller, the repository and the MongoClient. Finally, we set the controller in the view and show it. We are also using the args argument, containing the command line arguments, to make our application able to connect to any MongoDB server, given the host and the port (the defaults are still “localhost:27017”). We can start this application, after launching the MongoDB in a Docker container. We can manually verify that everything works.



You may now realize that this is the first time we launch our application for manual testing.

In the previous chapters, we were able to test all the aspects in isolation, through unit tests, and then to verify that the integration of our tested components still work, through integration tests.

To make the example more interesting, we would like to handle further command-line arguments for specifying the database name and the collection name. We could do that manually, by parsing the args parameter, but that would become cumbersome. Instead, we use the **Picocli**, <https://github.com/remkop/picocli>, a Java command line parser. Picocli allows us to specify the command line arguments by defining fields in a Java class with its annotations. Picocli initializes such annotated fields from the command line arguments, converting the input to strongly typed values in the fields. We refer to its documentation for all the details, <https://picocli.info/>.

We add the Picocli dependency to our POM:

```
1 <dependency>
2   <groupId>info.picocli</groupId>
3   <artifactId>picocli</artifactId>
4   <version>4.6.1</version>
5 </dependency>
```

Our main application accepts the command line arguments for specifying the MongoDB host and port and for the database and collection names. Otherwise, it relies on some default values.

```
1 package com.examples.school.app.swing;
2 ...
3 import java.util.concurrent.Callable;
4 import picocli.CommandLine;
5 import picocli.CommandLine.Command;
6 import picocli.CommandLine.Option;
7
8 @Command(mixinStandardHelpOptions = true)
9 public class SchoolSwingApp implements Callable<Void> {
10
11   @Option(names = { "--mongo-host" }, description = "MongoDB host address")
12   private String mongoHost = "localhost";
```

```

13
14     @Option(names = { "--mongo-port" }, description = "MongoDB host port")
15     private int mongoPort = 27017;
16
17     @Option(names = { "--db-name" }, description = "Database name")
18     private String databaseName = "school";
19
20     @Option(names = { "--db-collection" }, description = "Collection name")
21     private String collectionName = "student";
22
23     public static void main(String[] args) {
24         new CommandLine(new SchoolSwingApp()).execute(args);
25     }...

```

We modify the code for starting our GUI frame is as follows, using the command line arguments stored in the fields of this class:

```

1  @Override
2  public Void call() throws Exception {
3      EventQueue.invokeLater(() -> {
4          try {
5              StudentMongoRepository studentRepository =
6                  new StudentMongoRepository(
7                      new MongoClient(new ServerAddress(mongoHost, mongoPort)),
8                      databaseName, collectionName);
9              StudentSwingView studentView = new StudentSwingView();
10             SchoolController schoolController =
11                 new SchoolController(studentView, studentRepository);
12             studentView.setSchoolController(schoolController);
13             studentView.setVisible(true);
14             schoolController.allStudents();
15         } catch (Exception e) {
16             ...
17         }
18     });
19     return null;
20 }

```

We can experiment manually with the application, passing a few arguments or stick with the defaults. In any case, remember to start MongoDB in a Docker container accordingly before running this Java application.

Thanks to Picocli our application accepts standard arguments like `--help` and the result will be the

typical output showing all the arguments and their descriptions (generated automatically using the specified annotations):

```
1 Usage: <main class> [-hV] [--db-collection=<collectionName>]
2                               [--db-name=<databaseName>] [--mongo-host=<mongoHost>]
3                               [--mongo-port=<mongoPort>]
4   --db-collection=<collectionName>
5           Collection name
6   --db-name=<databaseName>
7           Database name
8 -h, --help      Show this help message and exit.
9   --mongo-host=<mongoHost>
10          MongoDB host address
11   --mongo-port=<mongoPort>
12          MongoDB host port
```

Picocli will also terminate the application if we pass a wrong argument, e.g., an unknown or misspelled option:

```
1 Unknown option: '--mango-port'
```

The same happens if we pass a value for an option that is not valid for that option's type. For example, if we pass `--mongo-port=hello` we get:

```
1 Invalid value for option '--mongo-port': 'hello' is not an int
```



Throughout the book we stressed that each class should get its dependencies (i.e., the actual implementation of dependencies) *injected* from the outside. This is the crucial point, which allowed us to test and implement each component in isolation. This technique is known as **Dependency Injection** and up to now, we have implemented dependency injection manually. In unit tests, it was easy to inject dependencies, even without using Mockito dependency injection mechanisms: it was enough to create mocks and pass them to the SUT constructor. In integration tests, things became a little more complex: we had to connect a few real instances. In the `main` method, we have to connect all the components and this is evident from the long sequence of statements creating instances and passing them to the constructors or setter methods. It is crucial to write these statements correctly and in the right order to avoid problems at run-time. In a later chapter, Chapter [Learning tests](#), Section [Dependency Injection with Google Guice](#), we will use a dependency injection framework that will allow us to get rid of the complex statements for initializing and wiring all our objects.

## 14.2 E2e tests for our application

First of all, let's store all e2e tests into a separate test folder. We extend the configuration of the build-helper-maven-plugin plugin that we created in Chapter *Integration tests*, Section *Source folder for integration tests*, with the additional test source folder src/e2e/java (let's first create this folder in our project):

```
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>
13      <sources>
14        <source>src/it/java</source>
15        <source>src/e2e/java</source>
16      </sources>
17    ...
```

Then we update the Maven project from Eclipse; the new folder will be now handled as a test source folder.

We also configure failsafe to run e2e tests, which will have the E2E suffix in the Java file name, in a separate execution, just to keep e2e tests easily recognizable during the Maven build. Since E2E is not a standard suffix, we need to configure failsafe to run tests with this pattern. We also exclude the Java classes ending with IT, which are already executed during the default failsafe execution:

```
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       <id>default-it</id>
8       <goals>
9         <goal>integration-test</goal>
```

```
10      <goal>verify</goal>
11  </goals>
12 </execution>
13 <execution>
14   <id>e2e-tests</id>
15  <goals>
16    <goal>integration-test</goal>
17    <goal>verify</goal>
18  </goals>
19  <configuration>
20    <excludes>
21      <exclude>**/*IT.java</exclude>
22    </excludes>
23    <includes>
24      <include>**/*E2E.java</include>
25    </includes>
26  </configuration>
27 </execution>
28 </executions>
29 </plugin>
```

AssertJ Swing supports launching an application from its `main` method. In particular, we can launch the application programmatically either by specifying the Java class or its fully qualified name. We go for the latter. Furthermore, we can pass arguments that will populate the `args` argument passed to `main`. We pass the host and the IP of the Docker container running MongoDB and we also specify the database and collection names (instead of relying on the default names). In this example, we will start the Docker container from the test with Testcontainers (see Chapter [Integration tests](#), Section [Integration tests with Docker and Testcontainers](#)). The application is started using the AssertJ Swing API of the class `ApplicationLauncher`. Then, the API of `WindowFinder` is used to look up the frame of our view by title:

```
1 package com.examples.school.view.swing;
2
3 import static org.assertj.swing.launcher.ApplicationLauncher.*;
4 ...
5 @RunWith(GUIRunner.class)
6 public class SchoolSwingAppE2E extends AssertJSwingJUnitTestCase {
7
8     @ClassRule
9     public static final MongoDBContainer mongo =
10         new MongoDBContainer("mongo:4.4.3");
11
12     private static final String DB_NAME = "test-db";
```

```
13  private static final String COLLECTION_NAME = "test-collection";
14
15  private MongoClient mongoClient;
16
17  private FrameFixture window;
18
19  @Override
20  protected void onSetUp() {
21      String containerIpAddress = mongo.getContainerIpAddress();
22      Integer mappedPort = mongo.getFirstMappedPort();
23      mongoClient = new MongoClient(containerIpAddress, mappedPort);
24      // always start with an empty database
25      mongoClient.getDatabase(DB_NAME).drop();
26      // add some students to the database
27      addTestStudentToDatabase("1", "first student");
28      addTestStudentToDatabase("2", "second student");
29      // start the Swing application
30      application("com.examples.school.app.swing.School SwingApp")
31          .withArgs(
32              "--mongo-host=" + containerIpAddress,
33              "--mongo-port=" + mappedPort.toString(),
34              "--db-name=" + DB_NAME,
35              "--db-collection=" + COLLECTION_NAME
36          )
37          .start();
38      // get a reference of its JFrame
39      window = WindowFinder.findFrame(new GenericTypeMatcher<JFrame>(JFrame.class) {
40          @Override
41          protected boolean isMatching(JFrame frame) {
42              return "Student View".equals(frame.getTitle()) && frame.isShowing();
43          }
44      }).using(robot());
45  }
46
47  @Override
48  protected void onTearDown() {
49      mongoClient.close();
50  }
51
52  private void addTestStudentToDatabase(String id, String name) {
53      mongoClient
54          .getDatabase(DB_NAME)
55          .getCollection(COLLECTION_NAME)
```

```

56     .insertOne(
57         new Document()
58             .append("id", id)
59             .append("name", name));
60     }
61 }
```

In the setup, we make sure we always start from an empty database and we also add two records in the database. We do that by manually using a MongoClient. We could have done that by using a StudentMongoRepository, which could be part of the test fixture. However, as mentioned above, in e2e tests, we should avoid using components of the application directly. Thus, for learning purposes, let's try to avoid that and manually interact with the database.

These two records in the database are part of the test fixture; each test can rely on such two records and can use them for verifications.

We write a first test that verifies that when the application starts it shows the current elements in the database:

```

1 @Test @GUITest
2 public void testOnStartAllDatabaseElementsAreShown() {
3     assertThat(window.list().contents())
4         .anySatisfy(e -> assertThat(e).contains("1", "first student"))
5         .anySatisfy(e -> assertThat(e).contains("2", "second student"));
6 }
```

Let's refactor the test a bit and extract constants for the strings of the database elements.

```

1 private static final String STUDENT_FIXTURE_1_ID = "1";
2 private static final String STUDENT_FIXTURE_1_NAME = "first student";
3 ...
4 addTestStudentToDatabase(STUDENT_FIXTURE_1_ID, STUDENT_FIXTURE_1_NAME);
5 addTestStudentToDatabase(STUDENT_FIXTURE_2_ID, STUDENT_FIXTURE_2_NAME);
6 ...
7 @Test @GUITest
8 public void testOnStartAllDatabaseElementsAreShown() {
9     assertThat(window.list().contents())
10        .anySatisfy(e -> assertThat(e)
11                    .contains(STUDENT_FIXTURE_1_ID, STUDENT_FIXTURE_1_NAME))
12        .anySatisfy(e -> assertThat(e)
13                    .contains(STUDENT_FIXTURE_2_ID, STUDENT_FIXTURE_2_NAME));
14 }
```

Let's write the tests for the “Add” button. For simulating the case when a student with the same id is already present we can rely on the two students already present in the database:

```

1  @Test @GUITest
2  public void testAddButtonSuccess() {
3      window.textBox("idTextBox").enterText("10");
4      window.textBox("nameTextBox").enterText("new student");
5      window.button(JButtonMatcher.withText("Add")).click();
6      assertThat(window.list().contents())
7          .anySatisfy(e -> assertThat(e).contains("10", "new student"));
8  }
9
10 @Test @GUITest
11 public void testAddButtonError() {
12     window.textBox("idTextBox").enterText(STUDENT_FIXTURE_1_ID);
13     window.textBox("nameTextBox").enterText("new one");
14     window.button(JButtonMatcher.withText("Add")).click();
15     assertThat(window.label("errorMessageLabel").text())
16         .contains(STUDENT_FIXTURE_1_ID, STUDENT_FIXTURE_1_NAME);
17 }
```

Note that in the second test, we only verify that an error is shown. We do not verify that the student we were trying to add is not effectively shown in the list. Since this is meant to be a high-level test, we can concentrate on the important aspect, i.e., the shown error. We have already verified that the list is not updated in the previous chapter in the integration test of the view (Chapter [UI tests](#), Section [UI integration tests](#)). In that integration test, it was easy to verify that, since we had started with an empty list and we verified that the list was still empty. Moreover, in that case, things were easier since we were still using other components of our application directly (in that case we were using the `StudentMongoRepository`). In these e2e tests, instead, we are not using any class of our application directly; moreover, the database has already been set up with 2 records. Thus, verifying that the list has not changed would require verifying that none of the list elements match the student we are trying to add. This shows that writing high-level e2e tests is harder than writing unit and integration tests, as we anticipated in Chapter [Testing](#).

Finally, let's write the tests for the “Delete” button.

```

1  @Test @GUITest
2  public void testDeleteButtonSuccess() {
3      window.list("studentList")
4          .selectItem(Pattern.compile("." + STUDENT_FIXTURE_1_NAME + "."));
5      window.button(JButtonMatcher.withText("Delete Selected")).click();
6      assertThat(window.list().contents())
7          .noneMatch(e -> e.contains(STUDENT_FIXTURE_1_NAME));
8  }
9
10 @Test @GUITest
```

```
11 public void testDeleteButtonError() {
12     // select the student in the list...
13     window.list("studentList")
14         .selectItem(Pattern.compile(".*" + STUDENT_FIXTURE_1_NAME + ".*"));
15     // ... in the meantime, manually remove the student from the database
16     removeTestStudentFromDatabase(STUDENT_FIXTURE_1_ID);
17     // now press the delete button
18     window.button(JButtonMatcher.withText("Delete Selected")).click();
19     // and verify an error is shown
20     assertThat(window.label("errorMessageLabel").text())
21         .contains(STUDENT_FIXTURE_1_ID, STUDENT_FIXTURE_1_NAME);
22 }
23
24 private void removeTestStudentFromDatabase(String id) {
25     mongoClient
26         .getDatabase(DB_NAME)
27         .getCollection(COLLECTION_NAME)
28         .deleteOne(Filters.eq("id", id));
29 }
```

Note that to make the tests slightly higher-level, we select from the list using a `java.util.regex.Pattern`. The case for simulating a wrong removal is once again harder to write. We must manually remove from the database the student corresponding to the selected element in the list, to recreate the error scenario.

Summarizing, in these e2e tests, we never refer to the Java classes of our application. To further stress this, we launched our GUI application through AssertJ Swing by specifying the fully qualified name of the main class, not its Java type. Our e2e tests interact with our application only through the AssertJ Swing fixture, that is, through the application GUI.

## 14.3 BDD with Cucumber

In this section we implement BDD tests using **Cucumber**, <https://cucumber.io/>. You can also install the Cucumber Eclipse plugin from <https://cucumber.io/cucumber-eclipse/update-site>. See Chapter *Testing*, Section *Behavioral-Driven Development (BDD)*, for a brief introduction on BDD.

In a Cucumber **feature file** (with extension `.feature`) you describe your tests in a descriptive language (similar to English). This DSL (Domain Specific Language) is called **Gherkin**. It serves as an automation test script as well as live specification documents. A feature file should describe a single feature of the system or a particular aspect of a feature. It provides a high-level description of a software feature, and groups related scenarios.

Each feature file begins with the keyword `Feature:`, which defines the logical functionality tested by this feature file: it describes a software feature that is going to be tested by the scenarios. After

Feature: you provide a short text describing the feature. Additional free-form text can be added to add more description.

A **scenario** defines examples of the expected behavior. A scenario can be seen as equivalent to a test in our regular development process. Each scenario starts with the keyword `Scenario`:. After the keyword, you provide a short text describing the scenario. A scenario consists of **steps** and can be broken down into three parts:

- An initial context (the precondition to the test), represented by the **Given** keyword. Given steps describe the initial context of the test.
- An event, represented by the **When** keyword. These steps describe an event, or an action, e.g., the user interacting with the application.
- An expected outcome (the verification of the test), represented by the **Then** keyword. These steps describe an expected outcome, using assertions.

Moreover, additional Given, When, and Then steps can be defined using **And** and **But**.

The feature file only provides the descriptions of scenarios and steps, but, as we will see later, we have to implement additional mechanisms to effectively run the tests.

Before starting using Cucumber, let's configure a few things in our project.

As done for the previous e2e tests, we want to create our BDD tests in a separate test source folder, `src/bdd/java`. Cucumber feature files should be stored as resources, so we also create a separate test resource folder `src/bdd/resources`. Let's first create these two folders in our project.

Similarly to what we have done before we extend the configuration of the `build-helper-maven-plugin`. This time, we also use its goal `add-test-resource` for adding a test resource folder:

```
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>
13      <sources>
14        <source>src/it/java</source>
15        <source>src/e2e/java</source>
```

```

16      <source>src/bdd/java</source>
17    </sources>
18  </configuration>
19 </execution>
20 <execution>
21   <id>add-test-resource</id>
22   <phase>generate-test-resources</phase>
23   <goals>
24     <goal>add-test-resource</goal>
25   </goals>
26   <configuration>
27     <resources>
28       <resource>
29         <directory>src/bdd/resources</directory>
30       </resource>
31     </resources>
32   </configuration>
33 </execution>
34 ...

```

Then we update the Maven project from Eclipse; the new folders will be now handled as a test source folder and a test resource folder, respectively.

Similarly to what we have done for e2e tests, we also configure `failsafe` to run BDD tests in a separate execution, just to keep bdd tests easily recognizable during the Maven build. We use BDD as the suffix for our Java BDD tests:

```

1 <execution>
2   <id>bdd-tests</id>
3   <goals>
4     <goal>integration-test</goal>
5     <goal>verify</goal>
6   </goals>
7   <configuration>
8     <excludes>
9       <exclude>**/*IT.java</exclude>
10      </excludes>
11      <includes>
12        <include>**/*BDD.java</include>
13      </includes>
14    </configuration>
15 </execution>

```

We add the dependencies for using Cucumber in Java with JUnit:

```
1 <cucumber.version>5.5.0</cucumber.version>
2 ...
3 <dependency>
4   <groupId>io.cucumber</groupId>
5   <artifactId>cucumber-java</artifactId>
6   <version>${cucumber.version}</version>
7   <scope>test</scope>
8 </dependency>
9 <dependency>
10  <groupId>io.cucumber</groupId>
11  <artifactId>cucumber-junit</artifactId>
12  <version>${cucumber.version}</version>
13  <scope>test</scope>
14 </dependency>
```

In `src/bdd/resources` we create a new feature file, e.g., `student_view.feature` (if you installed the Eclipse plugin, the file will be automatically populated with some example contents; just remove all the contents). We add these contents:

```
1 Feature: Student Application Frame
2   Specifications of the behavior of the Student Application Frame
3
4   Scenario: The initial state of the view
5     Given The database contains a student with id "1" and name "first student"
6     When The Student View is shown
7     Then The list contains an element with id "1" and name "first student"
```

To effectively run the tests, Cucumber relies on a testing framework, and as usual, we will use JUnit (see the dependency `cucumber-junit` above). Thus, we create a new Java class in `src/bdd/java` (recall that we configured `failsafe` to run files ending with BDD):

```
1 package com.examples.school.bdd;
2
3 import org.junit.runner.RunWith;
4
5 import io.cucumber.junit.Cucumber;
6 import io.cucumber.junit.CucumberOptions;
7
8 @RunWith(Cucumber.class)
9 @CucumberOptions(features = "src/bdd/resources", monochrome = true)
10 public class SchoolSwingAppBDD {
11
12 }
```

We need to specify the Cucumber JUnit runner, `io.cucumber.junit.Cucumber`, with the JUnit annotation `@RunWith`.<sup>1</sup> The `@CucumberOptions` annotation is used to provide options to the Cucumber runner, e.g., where the feature files can be found (we also use `monochrome` so that the output from the tests in the console will not use ANSI colors).



Cucumber also provides support for JUnit 5 but we will not show that in this book.

The Cucumber JUnit runner will try to execute all the scenarios of the feature files found in the specified directory (with the option `features` above). For each scenario, it will execute the steps in order. Recall that in this context, a Scenario represents a test.

What is still missing is the Java code implementing the tests, that is, in this context, the steps of the feature file. When running a Cucumber test, Cucumber will scan the current directory for Java files containing step definitions. Alternatively, the location where to look for step definition can be specified with the option `glue` in `@CucumberOptions`. A **Step Definition** is simply a Java method with an annotation and an expression that matches a scenario's step in the feature file. Thus, step definitions map (or “glue”, in the Cucumber’s terminology) each Gherkin step into runnable Java code that implements the actions that should be performed by the step. Step definitions wire the specification to the implementation.

Currently, we have no such step definition in any Java file.



The class annotated with the Cucumber JUnit runner is used only for running the test with JUnit: step definitions are written in a separate file. This shows that a Cucumber test is much different from the JUnit tests we have written so far, and it has a different lifecycle, as we will see in more details in this section.

However, when Cucumber encounters a Gherkin step without a matching step definition, it will print a step definition Java snippet, which can be used as a starting point for new step definitions.

By default, a missing step definition will not make the JUnit test fail (unless you specify `strict=true` in the `@CucumberOptions`). Let’s run this test. The JUnit view reports that the test succeeds. The console shows that the scenario and the steps are undefined. The output in the Console also gives us some suggestions on how to proceed:

---

<sup>1</sup>We have already seen other JUnit runners in Chapter [TDD](#), Section [TDD with JUnit 4 Parameterized Tests](#), in Chapter [Mocking](#), Section [Alternative ways of initializing mocks and other elements](#), and in Chapter [UI tests](#), Section [UI unit tests](#).

```
1 Undefined scenarios:
2   src/bdd/resources/student_view.feature:4 # The initial state of the view
3
4   1 Scenarios (1 undefined)
5   3 Steps (3 undefined)
6
7 You can implement missing steps with the snippets below:
8
9 @Given("The database contains a student with id {string} and name {string}")
10 public void the_database_contains_a_student_with_id_and_name(String string, String s\
11 tring2) {
12     // Write code here that turns the phrase above into concrete actions
13     throw new io.cucumber.java.PendingException();
14 }
15
16 @When("The Student View is shown")
17 public void the_Student_View_is_shown() {
18     // Write code here that turns the phrase above into concrete actions
19     throw new io.cucumber.java.PendingException();
20 }
21
22 @Then("The list contains an element with id {string} and name {string}")
23 public void the_list_contains_an_element_with_id_and_name(String string, String stri\
24 ng2) {
25     // Write code here that turns the phrase above into concrete actions
26     throw new io.cucumber.java.PendingException();
27 }
```



The JUnit view will as usual show a tree with feature nodes and scenario nodes. We can pass an option through `@CucumberOptions` so that the tree will also show every single step. However, this will make JUnit report the number of steps as the number of executed tests, which might be misleading (a test should correspond to a scenario). Note also that when running Cucumber tests, the JUnit view has a few limitations: in case of several features, you can re-run a single feature with the tree's context menu, but you cannot re-run a single scenario. Moreover, double-clicking on any node will not bring you to any Java file (it will show a dialog with an error).

We create a new Java class, e.g., `SchoolSwingAppSteps`, where we are going to implement the step definitions and paste the methods suggested in the above output (use Eclipse “Organize Imports” to automatically add the missing imports):

```
1 package com.examples.schoolbdd.steps;
2
3 import io.cucumber.java.PendingException;
4 import io.cucumber.java.en.Given;
5 import io.cucumber.java.en.Then;
6 import io.cucumber.java.en.When;
7
8 public class SchoolSwingAppSteps {
9
10    @Given("The database contains a student with id {string} and name {string}")
11    public void the_database_contains_a_student_with_id_and_name(
12        String string, String string2) {
13        // Write code here that turns the phrase above into concrete actions
14        throw new PendingException();
15    }
16
17    @When("The Student View is shown")
18    public void the_Student_View_is_shown() {
19        // Write code here that turns the phrase above into concrete actions
20        throw new PendingException();
21    }
22
23    @Then("The list contains an element with id {string} and name {string}")
24    public void the_list_contains_an_element_with_id_and_name(
25        String string, String string2) {
26        // Write code here that turns the phrase above into concrete actions
27        throw new PendingException();
28    }
29 }
```

It is important to note that Cucumber captures regular expressions such as strings and integers in the steps of the feature file. These will become parameters of the step definition methods and Cucumber will pass the corresponding arguments automatically during the execution. This means that two steps in a feature file differing only for the values of the captured regular expressions can be implemented with a single step definition. Later, we will see examples of two steps in a feature file differing only for the values of the captured regular expressions.

Let's run the test again. This time we see that the step definition is found but it is still pending (since we throw a PendingException):

```
1 Pending scenarios:  
2 src/bdd/resources/student_view.feature:4 # The initial state of the view  
3  
4 1 Scenarios (1 pending)  
5 3 Steps (2 skipped, 1 pending)  
6  
7 io.cucumber.java.PendingException: TODO: implement me  
8 ...
```

As we said above, the Cucumber JUnit runner does not fail if there are undefined or pending steps. Of course, the JUnit runner will fail if there's a failed assertion during the execution of a step definition.

Now, we will implement a step definition method at a time, in order. You see that in Cucumber, most of the Java code implementing tests is not in the Java class run by the Cucumber runner, but in the Java class containing the step definition files.

Most of the Java code used for implementing the steps is based on the code we have already seen in the previous chapters, so we will not comment it unless strictly required.



We will not use Testcontainers in this example. Since the test code is in a file different from the one for running the tests, it is not straightforward to setup Testcontainers. Thus, the Cucumber tests we will see in the following assume that you have already started a container with MongoDB with the usual arguments, e.g.,

`docker run -p 27017:27017 --rm mongo:4.4.3.` Alternatively, since in this project we have configured the docker-maven-plugin so that it starts the MongoDB container with the appropriate port mapping specification, if we need to run the Docker container manually for running tests in Eclipse, we can simply run `mvn docker:start`. When you do not need it anymore, remember to stop it with `mvn docker:stop`. However, this can be done only if we do not configure the plugin to map the container's port to a local random port (see Chapter [Integration tests](#), Section [Integration tests with Docker and Maven](#)).

We use a MongoClient for connecting to the MongoDB server. We implement the step `The database contains a student with id "..." and name "..."` as follows:

```
1 package com.examples.schoolbdd.steps;  
2  
3 import org.bson.Document;  
4  
5 import com.mongodb.MongoClient;  
6  
7 import io.cucumber.java.After;  
8 import io.cucumber.java.Before;  
9 import io.cucumber.java.PendingException;  
10 import io.cucumber.java.en.Given;
```

```
11 import io.cucumber.java.en.Then;
12 import io.cucumber.java.en.When;
13
14 public class SchoolSwingAppSteps {
15
16     private static final String DB_NAME = "test-db";
17     private static final String COLLECTION_NAME = "test-collection";
18
19     private MongoClient mongoClient;
20
21     @Before
22     public void setUp() {
23         mongoClient = new MongoClient();
24         // always start with an empty database
25         mongoClient.getDatabase(DB_NAME).drop();
26     }
27
28     @After
29     public void tearDown() {
30         mongoClient.close();
31     }
32
33     @Given("The database contains a student with id {string} and name {string}")
34     public void the_database_contains_a_student_with_id_and_name(
35         String id, String name) {
36         mongoClient
37             .getDatabase(DB_NAME)
38             .getCollection(COLLECTION_NAME)
39             .insertOne(
40                 new Document()
41                     .append("id", id)
42                     .append("name", name));
43     }
44     ...
45 }
```

Note that the annotations `@Before` and `@After` in the above code come from Cucumber (`io.cucumber.java.Before` and `io.cucumber.java.After`, respectively); they are not the usual JUnit annotations we have used so far. That is because the lifecycle of a Cucumber test is different from the standard JUnit lifecycle, as already anticipated. The standard JUnit annotations `@BeforeClass` and `@AfterClass` can be used in the main JUnit test class (the one run by the Cucumber runner) and they are respected by Cucumber. Instead, if you need to execute something before and after each scenario, you need to use the Cucumber annotations `@Before` and `@After`. These are called **hooks** (we refer to <https://docs.cucumber.io/cucumber/api> for further information about hook usage).

This step now succeeds:

```
1 1 Scenarios (1 pending)
2 3 Steps (1 skipped, 1 pending, 1 passed)
```

Now we have to implement the step definitions related to the GUI. We will still use AssertJ Swing. Up to now, we have used `AssertJSwingJUnitTestCase` as the base class of our AssertJ Swing tests. We know that this base class takes care of most of the plumbing involved when writing a GUI test. We cannot use such a base class now, since the AssertJ Swing test code will be in the Java file implementing step definitions, not in the Java file for running the tests. Thus, we have to manually write the setup and teardown operations for AssertJ Swing.

First of all, we need to install the `FailOnThreadViolationRepaintManager`, which checks that all access to Swing components is performed in the EDT. We do that once and for all in a `@BeforeClass` static method of the main JUnit test class:

```
1 package com.examples.schoolbdd;
2 ...
3 import org.assertj.swing.edt.FailOnThreadViolationRepaintManager;
4 ...
5 @RunWith(Cucumber.class)
6 @CucumberOptions(features = "src/bdd/resources", monochrome = true)
7 public class SchoolSwingAppBDD {
8     @BeforeClass
9     public static void setUpOnce() {
10         FailOnThreadViolationRepaintManager.install();
11     }
12 }
```

Then, in the `SchoolSwingAppSteps` we clean up the AssertJ Swing FrameFixture in the Cucumber's `@After` annotated method. We start the main application as before (see Section [E2e tests for our application](#)). When looking for our frame, we cannot rely on the `robot()` method, which we previously inherited from `AssertJSwingJUnitTestCase`. We need to create a Robot manually with `BasicRobot.robotWithCurrentAwtHierarchy()`:

```

1 ...
2 public class SchoolSwingAppSteps {
3 ...
4     private static final String DB_NAME = "test-db";
5     private static final String COLLECTION_NAME = "test-collection";
6
7     private FrameFixture window;
8
9     @After
10    public void tearDown() {
11        mongoClient.close();
12        // the window might be null if the step for showing the view
13        // fails or it's not executed
14        if (window != null)
15            window.cleanUp();
16    }
17 ...
18 @When("The Student View is shown")
19 public void the_Student_View_is_shown() {
20     // start the Swing application
21     application("com.examples.school.app.swing.SchoolSwingApp")
22         .withArgs(
23             "--db-name=" + DB_NAME,
24             "--db-collection=" + COLLECTION_NAME
25         )
26         .start();
27     // get a reference of its JFrame
28     window = WindowFinder.findFrame(new GenericTypeMatcher<JFrame>(JFrame.class) {
29         @Override
30         protected boolean isMatching(JFrame frame) {
31             return "Student View".equals(frame.getTitle()) && frame.isShowing();
32         }
33     }).using(BasicRobot.robotWithCurrentAwtHierarchy());
34 }
35 ...
36 }
```

Note that in our previous e2e tests, we were starting the application and creating the `FrameFixture` in the setup phase. In this example, this is part of a step implementation.

The step also succeeds.

Finally, we implement the last step definition, simply using AssertJ Swing API as done before:

```

1 @Then("The list contains an element with id {string} and name {string}")
2 public void the_list_contains_an_element_with_id_and_name(String id, String name) {
3     assertThat(window.list().contents())
4     .anySatisfy(e -> assertThat(e).contains(id, name));
5 }
```

Now, all steps succeed and the scenario passes:

```

1 1 Scenarios (1 passed)
2 3 Steps (3 passed)
```

The current scenario takes into consideration only one existing student in the database. We could be a little bit more expressive by considering two students initially in the database. Of course, we need to update also our expectations. To do that, we use And to add one step in the initial context and one step in the expectations:

```

1 Feature: Student Application Frame
2   Specifications of the behavior of the Student Application Frame
3
4 Scenario: The initial state of the view
5   Given The database contains a student with id "1" and name "first student"
6   And The database contains a student with id "2" and name "second student"
7   When The Student View is shown
8   Then The list contains an element with id "1" and name "first student"
9   And The list contains an element with id "2" and name "second student"
```

As said above, two steps in a feature file differing only for the values of the captured regular expressions can be implemented with a single step definition. In our example, the following two steps will both match the step definition method that we have already implemented `the_database_contains_a_student_with_id_and_name(String id, String name)`:

```

1 Given The database contains a student with id "1" and name "first student"
2 And The database contains a student with id "2" and name "second student"
```

The same holds for the And step after Then and the corresponding implemented step definition method.



Prepositions and adverbs (such as, Given and And) are ignored by Cucumber when looking up step definitions. That is why the two steps above can be implemented with a single step definition.

Thus, we can run `SchoolSwingAppBDD` again, and everything still succeeds. Of course, Cucumber will now show the two additional passed steps. Indeed, the first and the last step definition methods in `SchoolSwingAppSteps` will now be invoked twice, with different arguments:

```

1 1 Scenarios (1 passed)
2 5 Steps (5 passed)

```

### 14.3.1 Data tables

For the sake of learning, let's use another feature of Gherkin: **data tables**, which are useful for passing a list of values to a step definition. In our current scenario, we have two steps that differ for the passed values. We can rewrite those two steps with a single one, followed by a data table:

```

1 Scenario: The initial state of the view
2   Given The database contains the students with the following values
3     | 1 | first student |
4     | 2 | second student |
5   When The Student View is shown
6   ...

```

Note that we don't need to specify values in double quotes (if we did, the double quotes would be part of the values). This should improve the readability of our specifications.

Of course, if we run the test now, we have a missing step definition (now that the step has changed, our current step definition does not match this new step anymore):

```

1 You can implement missing steps with the snippets below:
2
3 @Given("The database contains the students with the following values")
4 public void the_database_contains_the_students_with_the_following_values(io.cucumber\
5 .datatable.DataTable dataTable) {
6   // Write code here that turns the phrase above into concrete actions
7   // For automatic transformation, change DataTable to one of
8   // E, List<E>, List<List<E>>, List<Map<K,V>>, Map<K,V> or
9   // Map<K, List<V>>. E,K,V must be a String, Integer, Float,
10  // Double, Byte, Short, Long, BigInteger or BigDecimal.
11  //
12  // For other transformations you can register a DataTableType.
13  throw new io.cucumber.java.PendingException();
14 }

```

In the presence of a data table, the step definition requires some more work, as shown in the comment of the snippet generated by the Cucumber runner. We will deal with that in a minute.

First of all, we remove the previous step implementation, which is not needed anymore:<sup>2</sup>

---

<sup>2</sup>While it does no harm to have step implementations for steps that are not present in any scenarios, these implementations would only clutter the tests.

```

1 // remove this method
2 @Given("The database contains a student with id {string} and name {string}")
3 public void the_database_contains_a_student_with_id_and_name(String id, String name) \
4 ...

```

and replace it with this one:

```

1 @Given("The database contains the students with the following values")
2 public void the_database_contains_the_students_with_the_following_values(
3     List<List<String>> values) {
4     values.forEach(
5         v -> mongoClient // v is a List<String>
6             .getDatabase(DB_NAME)
7             .getCollection(COLLECTION_NAME)
8             .insertOne(
9                 new Document()
10                .append("id", v.get(0))
11                .append("name", v.get(1)))
12    );
13 }

```

We changed the parameter of the method from `DataTable` to `List<List<String>>` (it is one of the suggestions of the comment in the snippet) for holding the values of the table specified in the feature file. With such a parameter, Cucumber will pass the contents of the data table in this data structure. The elements of the outer list will contain the rows, while elements of the inner list will contain the values of the columns for each row. Thus, our method will receive this representation of the table specified in the feature file:

```

1 [
2     [ "1", "first student" ],
3     [ "2", "second student" ]
4 ]

```

In our step definition, we use the passed list of list of strings accordingly to setup the database.

Let's use a data table also for the verification on the list. Our scenario now looks like this:

```

1 Scenario: The initial state of the view
2   Given The database contains the students with the following values
3     | 1 | first student |
4     | 2 | second student |
5   When The Student View is shown
6   Then The list contains elements with the following values
7     | 1 | first student |
8     | 2 | second student |

```

And we update our step definition class accordingly:

```

1 // remove this method
2 @Then("The list contains an element with id {string} and name {string}")
3 public void the_list_contains_an_element_with_id_and_name(String id, String name) {
4 ...
5
6 // add this method
7 @Then("The list contains elements with the following values")
8 public void the_list_contains_elements_with_the_following_values(
9   List<List<String>> values) {
10  values.forEach(
11    v -> assertThat(window.list().contents())
12      .anySatisfy(e -> assertThat(e).contains(v.get(0), v.get(1)))
13  );
14 }

```

Let's move on to a new scenario:

```

1 Scenario: Add a new student
2   Given The Student View is shown
3   When The user enters the following values in the text fields
4     | id | name           |
5     | 1 | a new student |
6   And The user clicks the "Add" button
7   Then The list contains elements with the following values
8     | 1 | a new student |

```

If we run the tests now, we see that two steps are missing:

```
1 You can implement missing steps with the snippets below:  
2  
3 @When("The user enters the following values in the text fields")  
4 public void the_user_enters_the_following_values_in_the_text_fields(io.cucumber.data\  
5 DataTable dataTable) {  
6 ...  
7  
8 @When("The user clicks the {string} button")  
9 public void the_user_clicks_the_button(String string) {  
10 ...
```

The last `Then` was already mentioned in the previous scenario and the corresponding step definition has already been implemented. The `Given The Student View is shown` does not require a new step definition. We reuse the one already implemented for the previous scenario's step `When The Student View is shown`. The new step for the "Add" button is parameterized over the text of the button. Thus, we can reuse it later for the "Delete Selected" button.

In the data table of the step `When The user enters the following values...` we specify a table with headers. This makes the specification clearer. However, if we specified `List<List<String>>` as the parameter type, as we did before, the method would get this representation of the table:

```
1 [  
2   [ "id", "name" ],  
3   [ "1", "a new student" ]  
4 ]
```

This is not comfortable to use in the method: we should skip the first list and we could not use the values of the headers. To use the headers of the data table (actually, to interpret the values of the first row as headers), we use another data structure for implementing the step: `List<Map<String, String>>`. The key in the map is the header and the value is the column value. With this type, the method gets this representation:

```
1 [  
2   { "id": "1", "name": "a new student" }  
3 ]
```

This allows us to use the header values for looking up the text fields in the view:

```
1 @When("The user enters the following values in the text fields")
2 public void the_user_enters_the_following_values_in_the_text_fields(
3     List<Map<String, String>> values) {
4     values
5         .stream() // each element is a Map<String, String>
6         .flatMap(m -> m.entrySet().stream())
7         .forEach(
8             e -> window
9                 .textBox(e.getKey() + "TextBox")
10                .enterText(e.getValue()))
11     );
12 }
13
14 @When("The user clicks the {string} button")
15 public void the_user_clicks_the_button(String buttonText) {
16     window.button(UIButtonMatcher.withText(buttonText)).click();
17 }
```

Remember that in the code above each call `m.entrySet().stream()` returns a `Stream<Entry<String, String>>` and `flatMap` flattens all such streams into a single `Stream<Entry<String, String>>`.

The scenario now succeeds.

The next scenario is as follows:

```
1 Scenario: Add a new student with an existing id
2     Given The database contains the students with the following values
3         | 1 | first student |
4     And The Student View is shown
5     When The user enters the following values in the text fields
6         | id | name           |
7         | 1 | a new student |
8     And The user clicks the "Add" button
9     Then An error is shown containing the following values
10    | 1 | first student |
```

Let's add the only missing step definition:

```

1 @Then("An error is shown containing the following values")
2 public void an_error_is_shown_containing_the_following_values(
3     List<List<String>> values) {
4     assertThat(window.label("errorMessageLabel").text())
5         .contains(values.get(0));
6 }
```

We could also write the scenario for removal through the “Delete” button, following a similar approach. We leave this as an exercise.

Another feature of Cucumber that we will not cover in this book is **tags**: they allow you to organize features and scenarios and are useful for running a subset of scenarios and for scoping hooks to a subset of scenarios. We refer to the official documentation: <https://docs.cucumber.io/cucumber/api>.

### 14.3.2 Step decoupling

Step definitions are not bound to a particular feature file or scenario. Indeed, the only thing used by Cucumber is the step definition’s expression. As mentioned above, Cucumber scans all the Java classes for the step definitions. It is good practice to organize steps by domain concept, using domain-related names, rather than names related to features or scenarios. This will increase step reusability across possible several feature files.

In the previous example, we implemented all the steps in a single file `SchoolSwingAppSteps`. This file contains step implementations for two different domain concepts: the database and the view.

Thus, we could split the step definitions into two different Java files, and name them according to the corresponding domain concept: `DatabaseSteps` and `StudentSwingViewSteps`. In these two files, we copy the steps we had previously implemented in the single Java file (which we’ll then remove):

```

1 public class DatabaseSteps {
2
3     static final String DB_NAME = "test-db";
4     static final String COLLECTION_NAME = "test-collection";
5
6     private MongoClient mongoClient;
7
8     @Before
9     public void setUp() {
10         mongoClient = new MongoClient();
11         ...
12     }
13
14     @After
15     public void tearDown() ...
```

```
16  
17 @Given("The database contains the students with the following values")  
18 public void the_database_contains_the_students_with_the_following_values(  
19     List<List<String>> values) {  
20     values.forEach...  
  
1 public class Student SwingViewSteps {  
2  
3     private FrameFixture window;  
4  
5     @After  
6     public void tearDown() ...  
7  
8     @When("The Student View is shown")  
9     public void the_Student_View_is_shown() ...
```

All the scenarios still pass.

### 14.3.3 High-level specifications

The feature file we wrote in the previous sections allowed us to get familiar with a few interesting features of Cucumber.

The scenarios that are written in Cucumber (and implemented in Java) are more readable than the e2e tests that we wrote manually in Java (Section *E2e tests for our application*). However, they still contain a few internal details that would not make them feasible for the end-user. The specifications we wrote could be fine for the developers, but they still expose a few low-level concepts.

Let's create a new file `src/bdd/resources/student_view_high_level.feature`. We define a new Feature and the following first scenario (we intentionally skipped the specification of the initial state of the application since it is not interesting in this case):

```
1 Feature: Student View High Level  
2   Specifications of the behavior of the Student View  
3  
4   Background:  
5     Given The database contains a few students  
6     And The Student View is shown  
7  
8   Scenario: Add a new student  
9     Given The user provides student data in the text fields  
10    When The user clicks the "Add" button  
11    Then The list contains the new student
```

In this example we use the Background section: a Background is run before each scenario but after any Before hook. It must be specified in a feature file before the first Scenario and you can only have one set of Background steps per feature.

The Cucumber runner will run both features, possibly reusing already implemented steps. For this feature, we reuse a few existing steps.

We need to implement the step `The database contains a few students`. We do that in the class `DatabaseSteps` (we factored out the method for adding records directly to the database and we use a few constants; this is similar to what we did in Section [E2e tests for our application](#)):

```
1 public class DatabaseSteps {
2     static final String STUDENT_FIXTURE_1_ID = "1";
3     static final String STUDENT_FIXTURE_1_NAME = "first student";
4     static final String STUDENT_FIXTURE_2_ID = "2";
5     static final String STUDENT_FIXTURE_2_NAME = "second student";
6     ...
7     @Given("The database contains a few students")
8     public void the_database_contains_a_few_students() {
9         addTestStudentToDatabase(STUDENT_FIXTURE_1_ID, STUDENT_FIXTURE_1_NAME);
10        addTestStudentToDatabase(STUDENT_FIXTURE_2_ID, STUDENT_FIXTURE_2_NAME);
11    }
12
13    private void addTestStudentToDatabase(String id, String name) {
14        mongoClient
15            .getDatabase(DB_NAME)
16            .getCollection(COLLECTION_NAME)
17            .insertOne(
18                new Document()
19                    .append("id", id)
20                    .append("name", name));
21    }
22 }
```

The other step in the Background has already been implemented and will be reused. The same holds for the step for clicking the “Add” button. We then need to implement the remaining steps in the class `Student Swing View Steps`:

```

1 public class Student SwingViewSteps {
2 ...
3     @Given("The user provides student data in the text fields")
4     public void the_user_provides_student_data_in_the_text_fields() {
5         window.textBox("idTextBox").enterText("10");
6         window.textBox("nameTextBox").enterText("new student");
7     }
8
9     @Then("The list contains the new student")
10    public void the_list_contains_the_new_student() {
11        assertThat(window.list().contents())
12            .anySatisfy(e -> assertThat(e).contains("10", "new student"));
13    }

```

Note that this time the specifications in the new feature file are high-level and they do not mention any specific value. Still, they are comprehensible. Actually, for the final user, they are much more readable than the previous specifications. On the other hand, the implementation of the step definitions requires some more work. Indeed, now the step definitions have to use values explicitly and consistently.

The next high-level scenario can be defined as follows:

```

1 Scenario: Add a new student with an existing id
2   Given The user provides student data in the text fields, specifying an existing id
3   When The user clicks the "Add" button
4   Then An error is shown containing the name of the existing student

```

We need to implement two step definitions:

```

1 @Given("The user provides student data in the text fields, specifying an existing id\
2 ")
3 public void the_user_provides_student_data_in_the_text_fields_specifying_an_existing\
4 _id() {
5     window.textBox("idTextBox").enterText(DatabaseSteps.STUDENT_FIXTURE_1_ID);
6     window.textBox("nameTextBox").enterText("new student");
7 }
8
9 @Then("An error is shown containing the name of the existing student")
10 public void an_error_is_shown_containing_the_name_of_the_existing_student() {
11     assertThat(window.label("errorMessageLabel").text())
12         .contains(DatabaseSteps.STUDENT_FIXTURE_1_NAME);
13 }

```

This scenario passes.

Finally, let's write the two scenarios for the "Delete" button:

```

1 Scenario: Delete a student
2   Given The user selects a student from the list
3   When The user clicks the "Delete Selected" button
4   Then The student is removed from the list
5
6 Scenario: Delete a not existing student
7   Given The user selects a student from the list
8   But The student is in the meantime removed from the database
9   When The user clicks the "Delete Selected" button
10  Then An error is shown containing the name of the selected student

```

Note that the step definition for "Delete Button" does not have to be implemented: the one for "Add" is reused, since that step is parameterized over the text of the button.

For the first scenario, we need these two additional step definitions in the class Student Swing View Steps

```

1 @Given("The user selects a student from the list")
2 public void the_user_selects_a_student_from_the_list() {
3     window.list("studentList")
4     .selectItem(Pattern.compile("." + DatabaseSteps.STUDENT_FIXTURE_1_NAME + "."));
5 }
6
7 @Then("The student is removed from the list")
8 public void the_student_is_removed_from_the_list() {
9     assertThat(window.list().contents())
10    .noneMatch(e -> e.contains(DatabaseSteps.STUDENT_FIXTURE_1_NAME));
11 }

```

For the last scenario we need an additional step definition in the class Database Steps, manually removing a student (we remove the first one):

```

1 @Given("The student is in the meantime removed from the database")
2 public void the_student_is_in_the_meantime_removed_from_the_database() {
3     mongoClient
4         .getDatabase(DB_NAME)
5         .getCollection(COLLECTION_NAME)
6         .deleteOne(Filters.eq("id", STUDENT_FIXTURE_1_ID));
7 }

```

We need one last step definition in the class Student Swing View Steps:

```

1 @Then("An error is shown containing the name of the selected student")
2 public void an_error_is_shown_containing_the_name_of_the_selected_student() {
3     assertThat(window.label("errorMessageLabel").text())
4         .contains(DatabaseSteps.STUDENT_FIXTURE_1_NAME);
5 }
```

Once again, since there are no captured expressions in the steps of these scenarios, when implementing the step definitions in Java we need to consistently use values and constants, possibly across step definitions in different Java files.

Summarizing, in the end, it depends on the target audience how high-level the Cucumber specifications need to be.

In particular, while the initial feature file can be written by the application developer, the second high-level feature file could be written by anyone.

## 14.4 Starting from the high-level specifications

In this book, we have always started from unit tests when applying TDD. We then wrote integration tests for components already unit tested. Finally, we tested the whole application.

As anticipated in Chapter *Testing*, Section *Behavioral-Driven Development (BDD)*, BDD typically fosters starting from multiple high-level specifications before starting the actual coding. Thus, while TDD is an *inside-out* process, beginning with unit tests and gradually integrate towards the final user interface, BDD is often understood as *outside-in*: we start with high-level specifications and go inside towards the implementation of single units. Indeed, TDD is meant for developers and it is meant to have a very fast cycle, while BDD is for everyone (including the final users) and has a much slower cycle: for each BDD scenario, we might have to write many TDD unit tests.

We now try to experiment with this different process with our current application. We change the specification for the “Delete” button a bit: if the student to delete is not found, it makes sense to remove it from the list of the GUI. Indeed, such a scenario implies that the information of the GUI is stale with respect to the database contents.<sup>3</sup> We do that in the file `student_view_high_level.feature`:

```

1 Scenario: Delete a not existing student
2   Given The user selects a student from the list
3   But The student is in the meantime removed from the database
4   When The user clicks the "Delete Selected" button
5   Then An error is shown containing the name of the selected student
6   And The student is removed from the list # new specification
```

---

<sup>3</sup>We anticipated this improvement in Chapter *UI tests*, Section *Improvements*.

The added step has already a step definition (it is the same as the “happy” scenario). If we now run the Cucumber tests, this step will fail, as expected: in the current implementation, only an error is shown, but the student still stays in the list.

We have to work on this new feature and we do that with TDD starting from unit tests.

From Chapter *Integration tests* we recall that the `StudentView` interface has a single method `showError` for all kinds of errors. This is meant to be called from the `StudentController` when something goes wrong. Currently, it is called both when we try to add a student with an already existing id and when we try to delete a student that does not exist.

The removal of the student from the list has to be implemented by the view. Currently, the only way to know whether we have to remove a student that does not exist is to inspect the error message passed by the controller. This however would couple the view implementation to the controller implementation: if the message in the controller changes, the view has to be updated accordingly. This breaks our modular architecture.

We can change the API of the `StudentView` by adding a new method `showErrorStudentNotFound` to be called by the controller when a student does not exist, instead of the generic `showError`.<sup>4</sup>

Let’s start with the unit test of the controller

`SchoolControllerTest.testDeleteStudentWhenStudentDoesNotExist()`. We modify it as follows:

```

1  @Test
2  public void testDeleteStudentWhenStudentDoesNotExist() {
3      Student student = new Student("1", "test");
4      when(studentRepository.findById("1")).
5          thenReturn(null);
6      schoolController.deleteStudent(student);
7      // call the new method instead of showError
8      verify(studentView)
9          .showErrorStudentNotFound("No existing student with id 1", student);
10     verifyNoMoreInteractions(ignoreStubs(studentRepository));
11 }
```

This does not compile, since the method `showErrorStudentNotFound` is not yet in the `StudentView` interface. Let’s add it with the Eclipse quickfix:

```

1  public interface StudentView {
2      ...
3      void showErrorStudentNotFound(String message, Student student);
4  }
```

---

<sup>4</sup>If in the future we implement some mechanisms for updating an existing student (both in the controller and in the GUI) we could use this new method also for the error situation when the user tries to update a student that is not present anymore.

The test compiles and we can run it. Note that in the project there will be compilation errors in `StudentSwipeView` since it has to implement the new abstract method. However, this does not prevent us from running `SchoolControllerTest` (a dialog will appear telling us that there are compilation errors in the project, but we can go on with “Proceed”).

The test fails, since the controller is not yet calling that new method. Let’s fix it by updating the corresponding `StudentController` method:

```

1 public void deleteStudent(Student student) {
2     if (studentRepository.findById(student.getId()) == null) {
3         studentView.showErrorStudentNotFound(
4             "No existing student with id " + student.getId(),
5             student);
6         return;
7     }
8     ... // as before

```

All tests in `SchoolControllerTest` are back to green.

Let’s write the failing unit test for the `StudentSwipeView`, in `StudentSwipeViewTest` (note that currently, all the tests in this test case are still green: even though `StudentSwipeView` does not compile, tests never call the missing method they all succeed):

```

1 @Test
2 public void testShowErrorStudentNotFound() {
3     Student student1 = new Student("1", "test1");
4     Student student2 = new Student("2", "test2");
5     GuiActionRunner.execute(
6         () -> {
7             DefaultListModel<Student> listStudentsModel =
8                 studentSwipeView.getListStudentsModel();
9             listStudentsModel.addElement(student1);
10            listStudentsModel.addElement(student2);
11        }
12    );
13    GuiActionRunner.execute(
14        () -> studentSwipeView.showErrorStudentNotFound("error message", student1)
15    );
16    window.label("errorMessageLabel")
17        .requireText("error message: " + student1);
18    // new assertion
19    assertThat(window.list().contents())
20        .containsExactly(student2.toString());
21 }

```

This test fails due to unresolved compilation problems, as expected. Let's add the missing method in `StudentSwipeView`, using the Eclipse quickfix.

With the empty implementation, the test fails when looking for the error message in the label.

Let's modify the default implementation like this:

```
1 @Override
2 public void showErrorStudentNotFound(String message, Student student) {
3     lblErrorMessage.setText(message + ":" + student);
4 }
```

The test now fails because the student is still on the list. Let's make the test pass like this:

```
1 @Override
2 public void showErrorStudentNotFound(String message, Student student) {
3     lblErrorMessage.setText(message + ":" + student);
4     listStudentsModel.removeElement(student);
5 }
```

All unit tests are green.

Let's run the integration tests. One fails, `StudentSwipeViewIT.testDeleteButtonError()`. In fact, in that test we were verifying that the student not found was still in the list:

```
1 @Test @GUI
2 public void testDeleteButtonError() {
3     // manually add a student to the list, which will not be in the db
4     Student student = new Student("1", "non existent");
5     GuiActionRunner.execute(
6         () -> studentSwipeView.getListStudentsModel().addElement(student));
7     window.list().selectItem(0);
8     window.button(JButtonMatcher.withText("Delete Selected")).click();
9     // this assertion now fails
10    assertThat(window.list().contents())
11        .containsExactly(student.toString());
12    window.label("errorMessageLabel")
13        .requireText("No existing student with id 1: " + student);
14 }
```

We just need to update the assertion accordingly:

```
1 assertThat(window.list().contents()).isEmpty();
```

The e2e tests are still green (we were not verifying the state of the list in such a situation).

Finally, the Scenario: Delete a not existing student now succeeds.

## 14.5 Change some low-level details

As another experiment, let's try to change something in the GUI implementation. High-level tests, like the e2e and Cucumber tests, should not be affected by a low-level change. Unit (and possibly integration) tests, related to the changed class, are instead expected to have failures.

For example, at the moment, when we display Student, both in the list and the error label, we rely on its `toString`, which will then display a student with the string `Student [id=<id>, name=<name>]`. It would be better to display a student by showing the id and the name in a different format, e.g., `<id> - <name>`.

Let's change this in the `Student SwingView`. We add a utility method

```
1 private String getDisplayString(Student student) {  
2     return student.getId() + " - " + student.getName();  
3 }
```

We use it for displaying error messages in the error label:

```
1 @Override  
2 public void showError(String message, Student student) {  
3     lblErrorMessage.setText(message + ": " + getDisplayString(student));  
4 }  
5 ...  
6 @Override  
7 public void showErrorStudentNotFound(String message, Student student) {  
8     // similar  
9 }
```

We also use it for displaying elements in the list (this is achieved by implementing a customized `ListCellRenderer`; for details, please have a look at the Swing documentation):

```

1 listStudents.setCellRenderer(new DefaultListCellRenderer() {
2     @Override
3     public Component getListCellRendererComponent(JList<?> list, Object value,
4         int index, boolean isSelected, boolean cellHasFocus) {
5         Student student = (Student) value;
6         return super.getListCellRendererComponent(list,
7             getDisplayString(student),
8             index, isSelected, cellHasFocus);
9     }
10 });

```

What happens to our test suite?

We expect to have lots of failures in unit tests of `Student Swing View Test`, where we verified the strings by exact match. In unit tests, it made sense to be precise on our verifications, using knowledge of the internal details.

We then have to update the failing tests. For example, this test

```

1 @Test
2 public void testShowErrorShouldShowTheMessageInTheErrorMessageLabel() {
3     Student student = new Student("1", "test1");
4     GuiActionRunner.execute(
5         () -> studentSwingView.showError("error message", student)
6     );
7     window.label("errorMessageLabel")
8     .requireText("error message: " + student);
9 }

```

must be updated as follows:

```

1 @Test
2 public void testShowErrorShouldShowTheMessageInTheErrorMessageLabel() {
3     Student student = new Student("1", "test1");
4     GuiActionRunner.execute(
5         () -> studentSwingView.showError("error message", student)
6     );
7     window.label("errorMessageLabel")
8     .requireText("error message: 1 - test1");
9 }

```

We have to update a few failing tests in the integration tests of the view, `Student Swing View IT`, along the same line. In those tests, we were verifying the integration of our view with the real controller,

and it made sense to verify internal details of the view. If however, this does not look right in these integration tests, we might switch to higher-level verifications (along the line of e2e tests).

Since we wrote e2e tests and Cucumber tests in a high-level way, this should not break such tests. Indeed, in those tests we were not verifying the exact string (in list and error label): we were only verifying that the id and the name of the student were present. Indeed, these tests are still green.

# 15. Code Quality

In this chapter, we apply several static analysis mechanisms to keep track of the code quality of our Java code. These static analysis mechanisms are complementary to the ones already performed by the Java compiler. Code quality measures the reliability and maintainability of an application during the project's lifespan. The primary rationale behind this is that a low-quality codebase is usually more expensive to maintain. To this aim, we will use SonarQube and its ecosystem. We will first use SonarQube locally, and then we will use SonarCloud, a SonarQube instance in the cloud, from a GitHub Actions workflow.

SonarQube is an open-source platform for *continuous inspection of code quality*. It implements several mechanisms for static analysis to detect issues that undermine the code quality of a project (it supports several programming languages including Java).

SonarQube defines a model for code quality consisting of several measures. The main three measures are

## Reliability

issues in this domain are **bugs**: your software might behave differently from what you expect.

## Security

issues in this domain are **vulnerabilities**: your software might be subject to breaches and attacks.

## Maintainability

issues in this domain are **code smells**: your software might be hard to maintain.

SonarQube also detects other problems like duplicated code and provides a web interface with reports on the measures mentioned above and unit tests and code coverage. It keeps track of the history of the above metrics and provides evolution graphs. SonarQube provides fully automated analysis and integration with Maven, and, as we will see later, it also provides an Eclipse plugin and SonarCloud, a cloud service based on SonarQube.

SonarQube also provides an estimation of the time that it might take to solve detected problems. The sum of such estimations represents the **technical debt** (a metaphor created by Ward Cunningham, <http://c2.com/doc/oopsla92.html>). This is similar to a financial debt, which has to be repaid with interest. Writing software without code quality best practices makes the software less maintainable leading to technical debt. Paying off the debt means fixing possible code quality problems. This debt might keep increasing over time, just like interest. The higher the debt, the harder it will be to pay it off.

Indeed, SonarQube fosters fixing code quality problems as soon as possible. This is called the *water leak paradigm*: water leaks should be fixed soon. For this reason, SonarQube emphasizes

the code quality of changed and added code (with respect to the one already present in the project's repository). In SonarQube, the *Leak* is a built-in concept. After the first analysis, as seen in this chapter, further analysis will provide yellow areas focusing on new issues.



SonarQube changes very frequently. If you install a new version, the web interface might be slightly different concerning the following descriptions. The same holds for rules and quality gates and profiles. New rules can be added in newer versions or renamed, while others can be deprecated.

## 15.1 Using SonarQube locally

SonarQube is available as a standalone JAR that can be downloaded and run on the local computer. SonarQube needs a database to store the history of the analysis of projects.

By default, SonarQube uses an embedded H2 database unsuitable for production. For production, you should connect SonarQube to an actual database. SonarQube can use most mainstream databases. Indeed, when using H2, if you update SonarQube to a newer version, the database schema cannot be updated, and all the previous data will be lost.

For running SonarQube locally, using the Docker image is the easiest solution. In particular, a few `docker-compose.yml` files are available from this page <https://github.com/SonarSource/docker-sonarqube/blob/master/examples.md>. We use the `docker-compose.yml` that starts the SonarQube container and connects it to a PostgreSQL container.

In particular, this is the `docker-compose.yml` we are going to use:

```
1  version: "2"
2
3  services:
4    sonarqube:
5      image: sonarqube:7.9-community
6      depends_on:
7        - db
8      ports:
9        - "9000:9000"
10     networks:
11       - sonarnet
12     environment:
13       - sonar.jdbc.url=jdbc:postgresql://db:5432/sonar
14     volumes:
15       - sonarqube_conf:/opt/sonarqube/conf
16       - sonarqube_data:/opt/sonarqube/data
```

```
17      - sonarqube_extensions:/opt/sonarqube/extensions
18
19 db:
20   image: postgres:11.1
21   networks:
22     - sonarnet
23   environment:
24     - POSTGRES_USER=sonar
25     - POSTGRES_PASSWORD=sonar
26   volumes:
27     - postgresql:/var/lib/postgresql
28     - postgresql_data:/var/lib/postgresql/data
29
30 networks:
31   sonarnet:
32     driver: bridge
33
34 volumes:
35   sonarqube_conf:
36   sonarqube_data:
37   sonarqube_extensions:
38   postgresql:
39   postgresql_data:
```



Make sure you read the “Docker Host Requirements” section at [https://hub.docker.com/\\_/sonarqube/](https://hub.docker.com/_/sonarqube/). For example, for Linux, you probably need to run this command as a superuser (e.g., with sudo): `sudo sysctl -w vm.max_map_count=262144`. Otherwise, SonarQube might not be able to run.

We specify the versions of the two images to make our SonarQube installation reproducible. This file will also instruct Docker to create the data volumes so that both the SonarQube installation (including its plugins) and the PostgreSQL database will be persisted and available for the subsequent runs. (refer to Chapter [Docker](#) for further details about Docker networks, volumes and Docker compose).

Copy the `docker-compose.yml` to a local directory and run `docker-compose` from that directory. After the images are downloaded and the containers start, you should see this string in the log, “SonarQube is up”, stating that SonarQube is ready. The very first time, after downloading Docker images, some additional time is required to set up the SonarQube database.

Now SonarQube is up and running and can be accessed from <http://localhost:9000>.



If we update SonarQube, e.g., by pulling a new version, we might have to update the database schema. This is done automatically by SonarQube: we just have to open the URL <http://localhost:9000/setup> and follow the instructions.

Before analyzing projects, let's explore the web interface of SonarQube and its main settings.

Select “Quality Profiles,” and you can see that SonarQube already has a few default plugins. We'll use the Java plugin and the corresponding quality profile. In “Rules,” we can see all the rules defined by the plugins; select Java on the left panel to see only Java code quality rules.

Let's log in as an administrator to configure a few things. The default admin user is “admin” with the password “admin” (of course, it is better to change the password immediately). In “Quality Gates,” we can see the thresholds that will be used to give **grades** to the projects (the best grade is A). Navigate to **Administration → Marketplace → Installed** to see all installed plugins and to “Updates Only” to check for possible updates. Updating the Java-related plugins, such as “Java Code Quality and Security” and “JaCoCo,” might be good. After an update, we must restart SonarQube directly from the web interface. Using the tab “All”, you can install additional plugins directly from the Marketplace.

## 15.2 Analyze a project

We now perform the analysis of a project using Maven. We use the example of the previous chapter. For simplicity and to speed up the build, we can use a project version without the BDD tests.

We run the Maven build invoking the goal `sonar` of the Maven SonarQube plugin, `org.sonarsource.scanner.maven directly:sonar-maven-plugin` (The name is `sonar` since the original name of SonarQube was “Sonar”). For the moment, we only run unit tests.

```
1 mvn test sonar:sonar
```

The goal invocation works even without configuring the Maven plugin in the POM, taking the latest version. However, as usual, it is recommended to lock down versions of the plugin in the `<pluginManagement>` section (see Chapter [Maven](#), Section [Plugin management](#)), for example:

```
1 <pluginManagement>
2   <plugins>
3     <plugin>
4       <groupId>org.sonarsource.scanner.maven</groupId>
5       <artifactId>sonar-maven-plugin</artifactId>
6       <version>3.8.0.2131</version>
7     </plugin>
8   </plugins>
9 </pluginManagement>
```

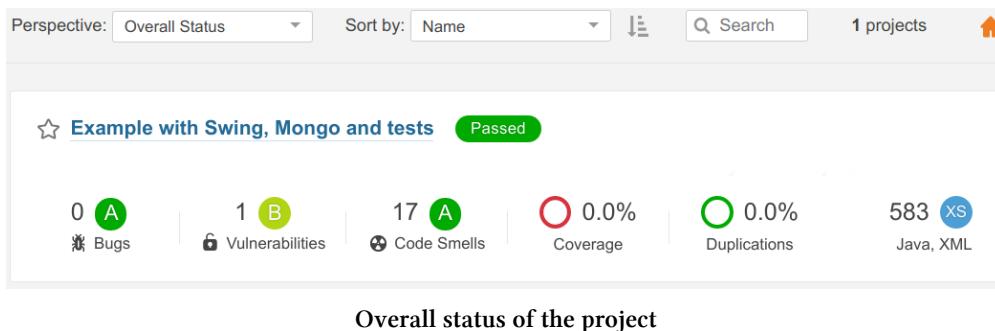
When the build finishes, we should see this output:

```
1 [INFO] ANALYSIS SUCCESSFUL, you can browse http://localhost:9000/dashboard?id=...name...
2   of your project...
```

Note that SonarQube automatically detected that our project is a Java project. It also detected that we use XML (the `pom.xml`). Thus, SonarQube will analyze both the Java and the XML files in search of possible problems according to the corresponding quality profiles. This can be seen in the log:

```
1 [INFO] Quality profile for java: Sonar way
2 [INFO] Quality profile for xml: Sonar way
```

Let's open the URL <http://localhost:9000> in the browser. We can see the result of our project analysis in the "Projects" section.



You can experiment with perspectives using the dropdown combo (the default is "Overall Status"). The project's name is from the `<name>` element specified in the POM. In our example, we have the following:

```
1 <name>Example with Swing, Mongo and tests</name>
```

The name is taken from the `artifactId` if the element is not specified.



Colors are used by SonarQube to highlight the quality of projects' metrics: dark green is the best, and red is the worst.

If we click on the project name, we land on the project's "Overview". This overview gives us an idea of possible problems detected by SonarQube. SonarQube also shows the estimated technical debt based on the detected code quality problems.

Let us concentrate on the "Issues" tab. (We'll deal with 0% code coverage percentage in a minute.)

On the left panel, we can filter the issues according to the **type** (Bug, Vulnerability, Code Smell, Security Hotspot) and **severity** (Blocker, Critical, Major, Minor, Info).

For example, in our project, there is one vulnerability of minor severity:

The screenshot shows a software interface for managing code quality issues. At the top, there's a navigation bar with tabs: Overview, Issues (which is selected), Measures, Code, Activity, and Administration. Below the navigation is a toolbar with buttons for 'My Issues' (selected), 'All', 'Bulk Change', and various selection tools. A status bar at the top right indicates '1 / 1 issues | 10min effort'. The main area displays a single issue from 'src/.../com/examples/school/app/swing/SchoolSwingApp.java'. The issue is a 'Vulnerability' of 'Minor' severity, labeled 'Use a logger to log this exception.' It includes a 'See Rule' link and a detailed description: 'last year ▾ L47 🔍 cwe, error-handling, owasp-a3'. The left sidebar contains filters for 'Type' (Bug, Vulnerability, Code Smell, Security Hotspot) and 'Severity' (Blocker, Critical, Major, Minor, Info). A note says 'Ctrl + click to add to selection'.

### A vulnerability issue of minor severity

If we click on the issue, we see the part of the code affected:

This screenshot shows the detailed view of the vulnerability. On the left, it shows '1 / 1 issues' and the file 'src/.../school/app/swing/SchoolSwingApp.java'. The specific issue is highlighted with a red border. The code snippet is as follows:

```

39  lore...
40
41  lore...
42
43
44
45
46
47

```

The line 'e.printStackTrace()' is underlined with a red arrow, indicating it is the part of the code affected by the issue. The right side of the interface shows the rest of the code and the same vulnerability summary as the previous screenshot.

### The part of the code affected by the issue

This rule has an estimated 10-minute effort (indeed, it is pretty straightforward to solve this issue, as we will see in a minute).

If we click on the "See Rule," we have a description of the rule that has been violated:

The screenshot shows a SonarQube analysis result for a Java project. The rule is titled "Throwable.printStackTrace(...) should not be called". It is categorized as a Vulnerability (Minor) and is associated with CWE-311. The rule was available since July 08, 2019, by SonarAnalyzer (Java). The ID is java:S1148. The description states that `Throwable.printStackTrace(...)` prints a `Throwable` and its stack trace to some stream, defaulting to `System.Err`, which could inadvertently expose sensitive information. It recommends using loggers instead. A note says the rule raises an issue when `printStackTrace` is used without arguments. A "Noncompliant Code Example" is provided:

```
try {  
    /* ... */  
} catch(Exception e) {  
    e.printStackTrace();      // Noncompliant  
}
```

A "Compliant Solution" is also shown:

```
try {  
    /* ... */  
} catch(Exception e) {  
    LOGGER.log("context", e);  
}
```

### The description of the violated rule

A rule description typically describes the primary intent of the rule, the reasons why the code violates the rule, and shows a few examples of non-compliant code and how to make the code compliant with the rule. The ID of the rule is in the top right corner of the description. For example, the ID of this rule is `java:S1148`. We will use the ID later to exclude rules from the analysis (Section [False positives and rule exclusion](#)).

We could use Log4J to log that exception as we have done in other parts of the book, or, for this project, we can rely on the standard Java logging mechanisms available in the package `java.util.logging`:

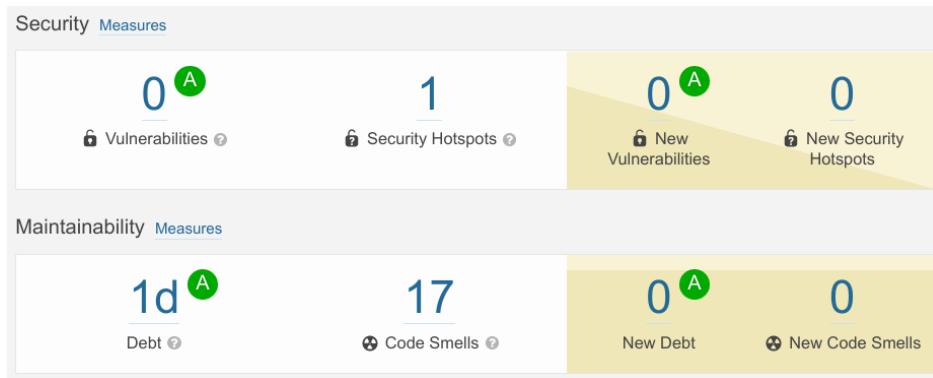
```
1 ...  
2 } catch (Exception e) {  
3     Logger.getLogger(getClass().getName())  
4         .log(Level.SEVERE, "Exception", e);  
5 }
```

If we rerun the analysis, we can see that the vulnerability issue has been solved.



Since, in this case, we have just modified a Java source, we can run the single goal `sonar:sonar` instead of executing the whole phase test.

SonarQube started to keep track of our project, showing issues' trends in the overview (refer to the concept of “leak” in the introduction of this chapter). The “Security Rating” has changed from B to A:



The vulnerability issue has been solved

We have several *code smells* (i.e., something in the source code that possibly indicates a more profound problem). First, let's concentrate on the critical one. The violated rule is "Fields in a Serializable class should either be transient or serializable". The rule states that "Fields in a Serializable class must themselves be either Serializable or transient even if the class is never explicitly serialized or deserialized".

```
src/.../com/examples/school/view/swing/StudentSwipeView.java

28     public class StudentSwipeView extends JFrame implements StudentView {
29 ...
30         private static final long serialVersionUID = 1L;
31 ...
32         private JPanel contentPane;
33         private JTextField txtId;
34         private JTextField txtName;
35         private JButton btnAdd;
36         private JList<Student> listStudents;
37         private JScrollPane scrollPane;
38         private JButton btnDeleteSelected;
39         private JLabel lblErrorMessage;
40 ...
41         private DefaultListModel<Student> listStudentsModel;
42 ...
43         private SchoolController schoolController;
```

Make "schoolController" transient or serializable. ...

6 days ago L43 !  
! Code Smell ! Critical O Open Not assigned 30min effort  
! cwe, serialization

The part of the code affected by the issue

Either we make the class `SchoolController` serializable, or we turn the instance variable in the view transient; we go for the latter:

```
1 private transient SchoolController schoolController;
```

This change will solve the issue after we run the analysis.

When looking at the log, we can spot the following line:

<sup>1</sup> [WARNING] SonarScanner will require Java 11 to run starting in SonarQube 8.x

Since we are using SonarQube 7.9, we can ignore this warning. However, when we use SonarCloud in Section [SonarCloud](#), which uses SonarQube 8, we will have to deal with that.

### 15.2.1 False positives and rule exclusion

On `Student SwingView`, we get two code smells in several parts.

First of all, “Inheritance tree of classes should not be too deep”, stating that “This class has 6 parents which is greater than 5 authorized.” (the same holds for the custom `DefaultListCellRenderer` we implemented in Chapter [End-to-end tests](#), Section [Change some low-level details](#)). However, there’s not much we can do about that: our class extends the Swing base class `JFrame`; thus, our class is not violating this rule, and we could not fix it in any way. We thus decide to ignore this issue in this class.<sup>1</sup> We could do that from the SonarQube interface using the dropdown menu as shown in the screenshot:



Manual resolution of an issue

However, if we switch to another SonarQube server, we should repeat this operation. We can then configure the POM to ignore specific rules on specific classes, as we will see in a minute.

This file also has several code smells due to the rule “Local variable and method parameter names should comply with a naming convention” that suggests “Rename this local variable to match the regular expression `^[a-zA-Z][a-zA-Z0-9]*$`.” This issue is due to the way WindowBuilder generated local variables for things such as `GridBagConstraints`:

<sup>1</sup> `GridBagConstraints gbc_scrollPane = new GridBagConstraints();`

We can rename all those local variables, but again, we can ignore this issue on the code generated by WindowBuilder.

To disable a SonarQube rule in the POM, we first need to know its ID. As we have already seen, we retrieve the ID by clicking on “See Rule”: the ID is on the top right corner of the description. For example, for “Inheritance tree of classes should not be too deep,” the ID is `java:S110`.

Then, we must specify a few properties in our POM:

<sup>1</sup>Note that the rule has an estimated effort of 4h30m. As you can imagine, re-structuring a class hierarchy is a time-consuming task.

```
1 <!-- For each rule to be ignored, add another eXX and two corresponding  
2 properties with eXX in the name as below -->  
3 <sonar.issue.ignore.multicriteria>e11</sonar.issue.ignore.multicriteria>  
4 <!-- Disable rule for "Inheritance tree of classes should not be too  
5 deep" (just an example) -->  
6 <sonar.issue.ignore.multicriteria.e11.ruleKey>  
7   java:S110  
8 </sonar.issue.ignore.multicriteria.e11.ruleKey>  
9 <sonar.issue.ignore.multicriteria.e11.resourceKey>  
10  **/*.java  
11 </sonar.issue.ignore.multicriteria.e11.resourceKey>
```

The e11 is an identifier we choose for this exclusion, which we then have to use in the other properties. We must specify the id of the rule with the property that ends with ruleKey. We must specify the Java sources that should not be considered for that rule with the property that ends with resourceKey. The other sources that do not match the specified pattern will still be analyzed according to the rule. In our example, we exclude that rule for all Java files.

If we rerun the analysis, the code smell issues related to that rule will disappear. Note that the technical dept dropped down from 1 day to 18 minutes. As noted above, the rule we excluded has an estimated effort of 4h30m.

Let's also exclude the other rule, "Local variable and method parameter names should comply with a naming convention", whose ID is java:S117. This time we exclude it only for the Student Swing View. We have to choose another identifier, e.g., e12, and proceed as above:

```
1 <sonar.issue.ignore.multicriteria>e11,e12</sonar.issue.ignore.multicriteria>  
2 ...  
3 <!-- Disable rule for "Local variable and method parameter names should  
4 comply with a naming convention" (just an example) -->  
5 <sonar.issue.ignore.multicriteria.e12.ruleKey>  
6   java:S117  
7 </sonar.issue.ignore.multicriteria.e12.ruleKey>  
8 <sonar.issue.ignore.multicriteria.e12.resourceKey>  
9   **/Student Swing View.java  
10 </sonar.issue.ignore.multicriteria.e12.resourceKey>
```

Another way to exclude SonarQube rules is to add the comment // NOSONAR at the end of the line with the issue. This will suppress *all* the issues, from now on, which might be found on the line. Thus, while this is probably the most effortless mechanism to disable rules, it might be dangerous if not used carefully: many real problems on that line will go completely unnoticed. Indeed, SonarQube provides a specific rule with the id java:NoSonar, which raises an issue when // NOSONAR is used. Of course, this is not enabled by default in the default profile.



In general, you must have a very good reason to disable a SonarQube rule! Do not disable a rule just to get to 0 technical debt.

## 15.2.2 Analysis of test code

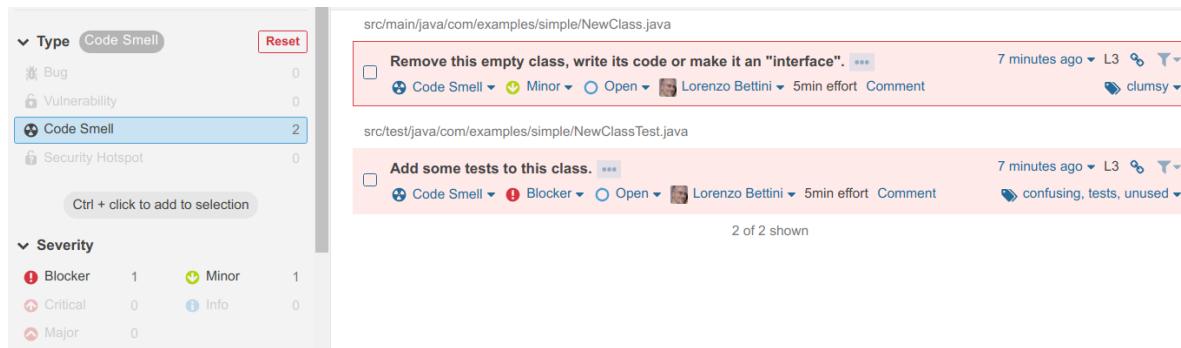
SonarQube analyzes test code as well but applies different rules. Indeed, code quality metrics and issues are different in the main and test codes.

Indeed, a few code smells are still detected in our test class `Student Swing View Test`. SonarQube shows that a few test methods in this class do not perform any assertion. The rule is “Tests should include assertions” (ID `java:S2699`), and the suggestion is to “Add at least one assertion to this test”. Indeed our test methods in `Student Swing View Test` actually assert something: they assert the presence of Swing controls using `AssertJ` Swing methods. However, SonarQube does not know that these `AssertJ` Swing methods effectively assert something.<sup>2</sup> Thus, in `Student Swing View Test`, we do not violate the abovementioned rule. We have to consider these issues as false positives.

As done before, we exclude this rule for the `Student Swing View Test` test class (see the sources of this example).

To show another example of the different rules applied for main and test code, let’s create an empty class `NewClass` in `src/main/java` and an empty class `NewClassTest` in `src/test/java`. Both classes are empty; thus, SonarQube will detect this as a code smell. However, the violated rules are different, and so are the severities.

Let’s run the analysis and see the results:



Different rules for main and test code

Having a Test class without any test method is much worse than having an empty class in the main code. The resolution of these issues is also different.

SonarQube is also aware of the mainstream test libraries, like `AssertJ`, and raises issues if we do not use the methods of such libraries correctly. This includes cases where we can assert the same thing using different assertion methods. Still, we should prefer a specific assertion method since it will

<sup>2</sup>Indeed, as we see in the following paragraphs, SonarQube is aware of mainstream testing libraries (see the rule description), including `AssertJ`, but unfortunately, it is not aware of `AssertJ` Swing.

provide a better error message, simplify the debugging process, and make the test more readable. For example, let's consider this test code:

```
1 assertThat(bankAccount.getId()).isGreaterThan(0);
```

SonarQube will raise an issue on `isGreaterThan(0)`, suggesting to “use `isPositive()` instead”. This rule is “Chained AssertJ assertions should be simplified to the corresponding dedicated assertion”. This rule applies in several usages of the assertion methods of AssertJ. For example, `isNull()` should be preferred to `isEqualTo(null)`, `isNotPositive()` should be preferred to `isLessThanOrEqualTo(0)`, etc. We have already seen this best practice in Chapter [JUnit](#), Section [Using AssertJ](#).

Similarly, when using mechanisms such as `Thread.sleep` in the test code to introduce some delays and synchronization, SonarQube will suggest using a dedicated testing library for testing concurrent code, such as Awaityility. (We used this library in Chapter [UI tests](#), Section [Multithreading](#).)

### 15.2.3 Security hotspots

If we rerun the analysis, our technical debt drops to 0!

However, there is still an issue reported on our `SchoolSwingApp`, “Using command line arguments is security-sensitive”. In Chapter [End-to-end tests](#), we added the `main` method using command line arguments. The rule states:

Command-line arguments can be dangerous, just like any other user input. They should never be used without being first validated and sanitized. [...] This code flags all program entry points (main methods) when command line arguments are used. The goal is to guide security code reviews.

This issue is a *security hotspot*, not a *vulnerability*. While vulnerabilities are points in the code open to attack, security hotspots are security-sensitive pieces of code that an expert should carefully review to verify that the sensitive piece of code is used most safely. More information about security reports can be found here: <https://docs.sonarqube.org/latest/user-guide/security-reports/>.

In our simple application, we can also decide to ignore that one.

### 15.2.4 Code coverage in SonarQube

SonarQube also keeps track of code coverage. In this example, we had never enabled code coverage with JaCoCo. Thus, SonarQube cannot find any information on code coverage. Indeed, SonarQube analyzes code coverage information produced by the build; it does not perform any code coverage itself.

Now that we solved all our issues, SonarQube focuses on the fact that we do not have any code coverage, and the quality gate now fails.



Quality gate failed due to missing coverage

We have already seen how to configure the JaCoCo Maven plugin in a separate profile in Chapter *Maven*, Sections [Configuring the JaCoCo Maven plugin](#) and [Maven profiles](#).

Thus, we configure the plugin in a separate profile, e.g., `jacoco`, similarly to what we have done before (see the sources of the example of this chapter). We only need to configure its goal `prepare-agent`. We also added the goal `report` to ensure that code coverage is executed correctly.



A few releases ago, SonarQube stopped inspecting the JaCoCo binary `.exec` files to collect code coverage. It now inspects the JaCoCo's XML coverage report, which is why it is crucial to enable the goal `report` of the JaCoCo Maven plugin.

```
1 <plugin>
2   <groupId>org.jacoco</groupId>
3   <artifactId>jacoco-maven-plugin</artifactId>
4   <version>0.8.5</version>
5   <configuration>
6     <excludes>
7       <exclude>**/model/*.*</exclude>
8       <exclude>**/SchoolSwingApp.*</exclude>
9     </excludes>
10    </configuration>
11    <executions>
12      <execution>
13        <goals>
14          <goal>prepare-agent</goal>
15          <goal>report</goal>
16        </goals>
17      </execution>
18    </executions>
19  </plugin>
```

Note that we exclude the model class `Student` since it's a domain model class (see Chapter [Code coverage](#), Section [Code coverage percentage](#)) with no logic to test. We also exclude the Java class with the `main` method since performing code coverage on that class makes no sense.

We verify that the resulting code coverage report in the directory `target/site/jacoco` (with the above exclusions) is 100% with

```
1 mvn verify -Pjacoco
```

Remember that the goal report is generated during the phase verify.

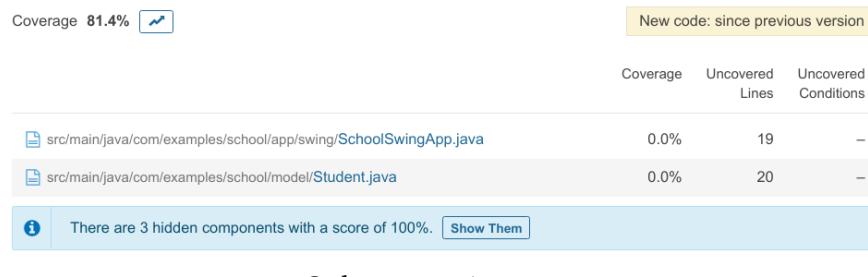
Let's now perform a new analysis with SonarQube enabling this profile (also, in this case, since we have just generated the JaCoCo report, we could simply run the single goal sonar:sonar):

```
1 mvn verify -Pjacoco sonar:sonar
```

In the log, we can see that SonarQube detected our code coverage data:

```
1 [INFO] Sensor JaCoCo XML Report Importer [jacoco]
2 [INFO] Sensor JaCoCo XML Report Importer [jacoco] (done)
```

However, SonarQube does not report 100% of code coverage. SonarQube automatically excludes the test code from the code coverage percentage. However, it still considers the Student class and the SchoolSwingApp, despite our exclusions in the JaCoCo plugin configuration in the POM:



Code coverage is not 100%

Indeed, SonarQube does not consider the exclusions configured for JaCoCo. We must manually force the coverage exclusions with the SonarQube property sonar.coverage.exclusions:

```
1 <sonar.coverage.exclusions>
2   **/model/*.*,
3   **/SchoolSwingApp.*
4 </sonar.coverage.exclusions>
```

If we rerun the analysis, we can see that SonarQube now shows 100% code coverage, excluding the files as specified by the new properties.

## 15.2.5 JUnit reports

Independently from code coverage, SonarQube only counts and shows the reports generated by surefire, that is, unit tests. Let's make SonarQube keep track of integration tests as well. In that case, we must set the following property explicitly in the POM, specifying the directories of reports of surefire and failsafe, separated by a comma:

```
1 <sonar.junit.reportPaths>
2   ${project.build.directory}/surefire-reports,
3   ${project.build.directory}/failsafe-reports
4 </sonar.junit.reportPaths>
```

SonarQube will still report all the tests in the “Unit Tests” section. There is no way to make SonarQube distinguish between unit tests and integration tests.

All the properties set in the POM could be configured also from the project’s settings in SonarQube. As we mentioned above, such settings would be persisted only on that specific installation of SonarQube. It is, however, useful to know how to specify such settings in SonarQube because it is the best way to get to know about the corresponding property keys to set in the POM.

In **Administration → General Settings → Analysis Scope**, we find settings (and properties) like “Coverage Exclusions” (and the property key `sonar.coverage.exclusions`) that we have already used. Here we can also specify “Duplications Exclusions”, i.e., the files that should be ignored by duplication detection (property key `sonar.cpd.exclusions`). Moreover, we can exclude entire files from any analysis (property key `sonar.exclusions`).

In **Administration → General Settings → Java**, we find settings that are specific to Java, for example, “JaCoCo Reports” (which we did not need to set since we used the default paths) and the “JUnit Report Paths” (which we have set above, using the property key `sonar.junit.reportPaths`).

## 15.3 SonarCloud

SonarCloud, <https://sonarcloud.io/>, is a SonarQube instance in the cloud, which is free for public projects.

As done with Coveralls, you access SonarCloud with your GitHub account. Then, you authorize SonarCloud to access some information about your GitHub account.

You can then create new “Organizations” or simply use the default organization represented by your account (which is of the shape `<YourGitHubUsername>-github`).

You can analyze a new project with SonarCloud from your computer by using the + sign in the SonarCloud web interface once logged in and by choosing the manual setup. Instead, we will use SonarCloud from our GitHub Actions workflows (Chapter [Continuous Integration](#), Section [GitHub Actions](#)).



There is a particular action, <https://github.com/SonarSource/sonarcloud-github-action>, to scan the code with SonarCloud from a GitHub Actions workflow. However, this cannot be used when we build our project with Maven.

Similar to what we have done in Chapter [Continuous Integration](#), Section [Code coverage with Coveralls](#), we need to use an authentication token from SonarCloud. We create a user authentication

token for your account on SonarCloud, navigating to <https://sonarcloud.io/account/security>. Please remember that once the token is generated and shown in the SonarCloud web interface, you will not be able to see it again, so you must make sure to copy that in a secret place on your computer.

As already stressed in Chapter *Continuous Integration*, Section *Using Coveralls from GitHub Actions*, the token must not be put as clear text in the GitHub repository. As done in the mentioned chapter, in the “Settings” of our GitHub repository, we navigate to “Secrets” and create a new repository secret. We specify SONAR\_TOKEN as the name (the name must be exactly like that), and we paste our SonarCloud token in the “Value” field. Finally, we press “Add secret”.

As anticipated in Section *Analyze a project*, SonarCloud uses SonarQube 8, requiring at least Java 11 to run the analysis. This does not mean we have to configure our Maven project to use Java 11. It just means that to run the build, we have to use Java 11. (At least, we need Java 11 when running the goal `sonar:sonar`).

The workflow file for this example is the following (where the value for the property `sonar.organization` must be replaced with your organization; note that we must also specify the URL of the SonarQube instance using the property `sonar.host.url`, which, by default, is `localhost:9000`):

```
1  name: Java CI with Maven, Docker, and SonarCloud in Linux
2
3  on:
4    push:
5      pull_request:
6
7  jobs:
8    build:
9      runs-on: ubuntu-latest
10
11   steps:
12     - uses: actions/checkout@v2
13     - name: Set up JDK 11
14       uses: actions/setup-java@v1
15       with:
16         java-version: 11
17     - name: Cache Maven packages
18       uses: actions/cache@v2
19       with:
20         path: |
21           ~/m2
22           ~/sonar/cache
23       key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml', '**/*.{yml}') }}
24       restore-keys: ${{ runner.os }}-m2-
25
26   - name: Build with Maven and SonarCloud
27     run: >
```

```

27      xvfb-run mvn verify -Pjacoco sonar:sonar
28      -Dsonar.organization=lorenzobettini-github
29      -Dsonar.host.url=https://sonarcloud.io
30      working-directory: com.examples.school
31      env:
32          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
33          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

Note that it is worthwhile to cache also `~/.sonar/cache`, where SonarQube caches a few downloaded artifacts for performing the analysis. (The YAML syntax `|` allows us to specify several entries.) We must also use `GITHUB_TOKEN`, a secret authentication token that GitHub provides.

As we will see later, SonarCloud can also analyze single branches. However, this is enabled only after performing at least one analysis of the master branch.

Pushing the change to GitHub will trigger the workflow. At the end of the build, the analysis using SonarCloud will be performed, and the results will be available on SonarCloud.

You can see this warning in the build log:

```

1 [WARNING] Shallow clone detected, no blame information will be provided. You can con\
2 vert to non-shallow with 'git fetch --unshallow'.
3 [WARNING] Missing blame information for the following files:
4 ...
5 [WARNING] This may lead to missing/broken features in SonarQube
```

By default, the action `checkout` only fetches one commit when cloning the GitHub repository. To fix this, you can run `git fetch --unshallow` in a step to get the full history. Alternatively, you can disable the shallow clone by specifying the following argument to the action:

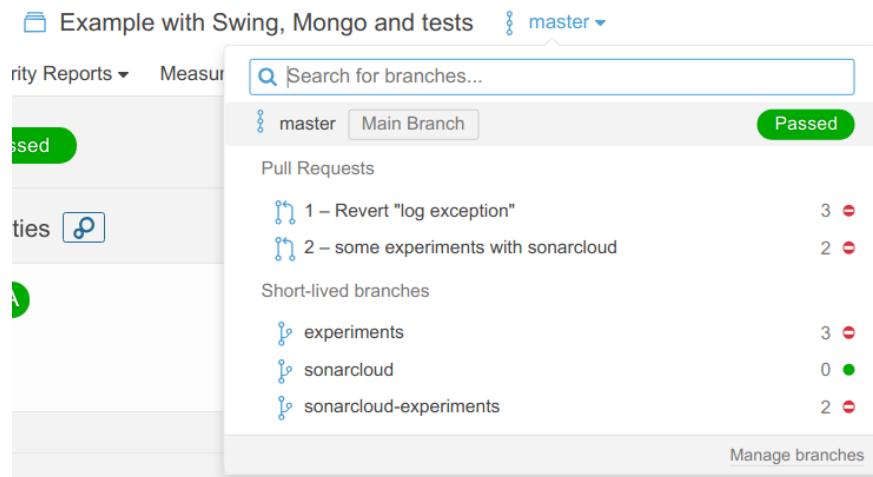
```

1 - uses: actions/checkout@v2
2   with:
3     # Shallow clones should be disabled for better relevancy of SonarQube analysis
4     fetch-depth: 0
```

SonarCloud comes with a built-in mechanism to analyze branches and pull requests as soon as they get created (first, you need to install the **SonarCloud application** on your GitHub organization, <https://github.com/apps/sonarcloud>). In particular, for new branches, SonarCloud will consider only the code that has been added/changed in the branch. The same happens when analyzing pull requests. The analysis of branches and pull requests are accessible using the dropdown menu, as shown in the following screenshot:<sup>3</sup>

---

<sup>3</sup>See the official documentation, <https://docs.sonarqube.org/latest/branches/overview/>, for the concepts of **long-lived** and **short-lived** branches.



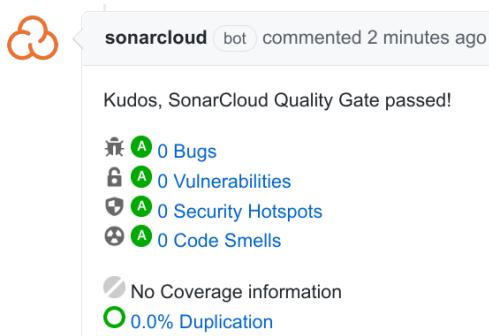
Branches and pull requests in SonarCloud

SonarCloud can also provide feedback on a PR (similar to the feedback of Coveralls, see Chapter *Continuous Integration*, Section *Code coverage with Coveralls*). To activate analysis on pull requests, we must install the SonarCloud application on the GitHub organization. It is enough to navigate to the URL <https://github.com/apps/sonarcloud> and follow the installation instructions. After that, new pull requests will get feedback from SonarCloud as well. Of course, in case the code of the pull request introduces new SonarQube issues, the feedback from SonarCloud will be negative. A “Details” link will allow us to see the issues with the code of the pull request.



For security reasons, at the moment, SonarCloud will inspect only internal pull requests of your repository: pull requests built from forks will not be inspected.

When providing feedback on a PR and in general when performing the analysis of the code of a separate branch, SonarCloud will only take into consideration the “new” and “changed” code. This way, SonarCloud will ensure you do not introduce issues with the new commits. For example, if you have many issues and technical debt in the main branch’s code, you can still get positive feedback from SonarCloud on a PR as long as you do not introduce new issues. Even code coverage is reported only on the new code. The idea is that, even if you have technical debt, you should not introduce a new one. Indeed, as said in this chapter’s introduction, SonarQube emphasizes the code quality of changed and added code.



Feedback on a PR from SonarCloud

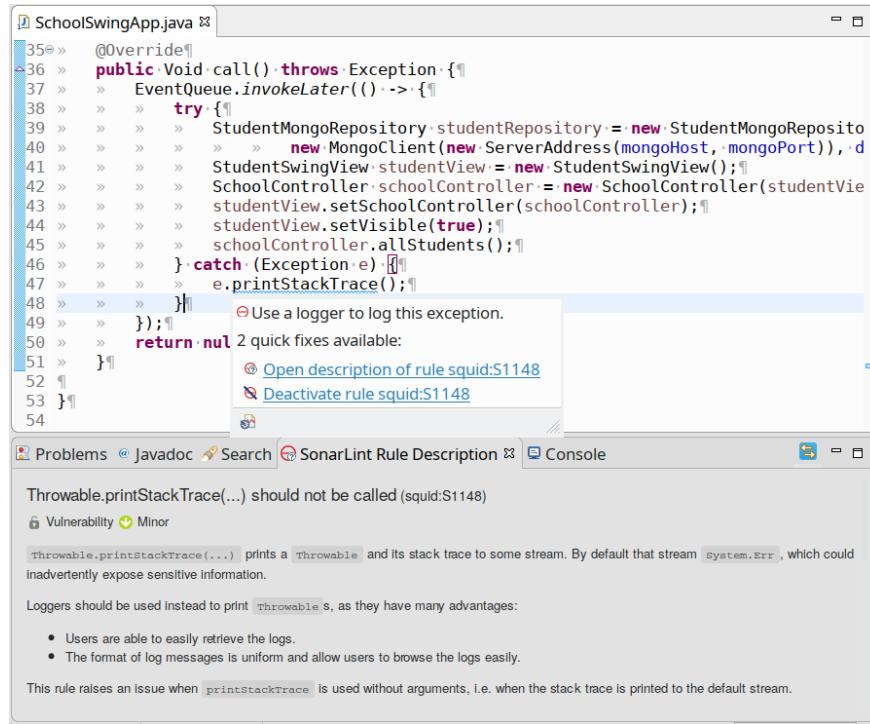
As with other services like GitHub Actions and Coveralls, SonarCloud provides **badges** to put, for example, in the README of the GitHub repository of the analyzed project. In the “Overview” of the project, the button “Get project badges” allows us to create several badges (e.g., “Quality Gate Status”, “Bugs”, “Code Smells”, “Coverage”, “Duplicated Lines”, “Technical Debt”, etc.). It is just a matter of copying the Markdown code and pasting it into the README of the GitHub repository.

As we have seen in other parts of the book, it does not make sense to analyze our code with SonarCloud from all the possible multiple GitHub Actions workflows. We can do that in the main workflow, the one that is executed on all pushes and pull requests, or choose a different strategy.

## 15.4 SonarLint (IDE integration)

SonarLint, <https://www.sonarlint.org/>, is a plugin for several IDEs, including Eclipse, which detects quality issues as you write code. It acts like a spell checker, and SonarLint “squiggles” the parts of code (and files) that violate code quality rules.

The Eclipse plugin can be installed from the Market Place or the update site <https://eclipse-uc.sonarlint.org/>. Then, SonarLint automatically analyzes code as you write in Eclipse, providing means for opening rule descriptions and deactivating a specific rule. In the screenshot, we can see the minor vulnerability issue due to `printStackTrace` in our example:



### Issue detected by SonarLint

SonarLint is not a complete replacement for SonarQube, since SonarLint only searches for issues (Bug, Code Smells, etc.). Still, it does not perform advanced analysis like code duplication detection, as SonarQube does. However, it is a valuable development tool since it allows you to discover issues early before running a complete SonarQube analysis.

# 16. Learning tests

When we integrate some third-party package (a library or a framework) into our code, we should be already familiar with such a third-party code developed (and hopefully well tested) by others.

We can read the documentation of the third-party package and have a look at the examples. However, it is crucial to be really confident in the use of such an external code before integrating it into our application.

We could first write a few examples, independently from our code, using the third-party package with some very simple classes, for experimenting and learning. Should we then write a `main` method running our examples and manually verify that the outcome is as expected? Why not using automated tests?

These tests are not meant to verify the behavior of our code, which, in this context, is meant to be just an excuse for learning the third-party code. These tests are meant to explore our understanding of the third-party code. Indeed, these tests are usually called **learning tests** ([Bec02](#), [Mar08](#)).

These tests are more than that and they have an important value. They verify that our expectations on the behavior of the third-party code are correct. Once they have been written, they can be run against a new release of the third-party code, to make sure that our expectations still hold, before using the new release in our code.

In this chapter, we write learning tests to get familiar with the **Dependency Injection** framework **Google Guice**. Afterward, we use this framework in our code, in particular, in the Swing application we developed in the previous chapters.

## 16.1 Dependency Injection with Google Guice

Throughout the book, we stressed over and over again that a class that depends on components should depend on abstractions, not on real implementations, as much as possible.

We followed this practice extensively in our examples, defining interfaces for our components. The only exception was the `SchoolController`, which does not extend an abstract class nor implements an interface. However, this has not prevented us to be able to write unit tests for every single component in isolation, using mocks.

Using mocks for testing and implementing each component in isolation has been possible not mainly thanks to the abstractions. We were able to do that because our classes never instantiate a dependency directly. Our controller does not instantiate the view nor the repository. The controller receives such implementations from outside, as constructor's parameters. The view has a setter method for receiving the controller.

Thus, each class gets its dependencies (i.e., the actual implementation of dependencies) *injected* from the outside. This is the crucial point, which allowed us to test and implement each component in isolation.

This technique is known as **Dependency Injection**, which we have already anticipated in other chapters, Chapter [JUnit](#) and Chapter [Mocking](#). Dependency Injection ([Fow04](#), [PP18](#)) is a design pattern that aims at removing hard-coded dependencies. With this pattern, the implementations of dependencies are *injected* from outside so that the effective dependency resolution will take place at runtime, not at compile-time. The goal of this pattern is to decouple objects so that no client code has to be modified simply because an object it depends on needs to be changed to a different one. In turns, dependency injection supports the **Dependency Inversion Principle** ([Mar00](#)), that is, making high-level components independent of the low-level component implementation details.

Up to now, we have implemented dependency injection manually. In unit tests, we mocked dependencies of the SUT and we passed the mock objects to the constructor of the SUT when creating its instances. We have already used a form of automatic dependency injection, relying on Mockito mechanisms for dependency injection. In Chapter [Mocking](#), Section [Alternative ways of initializing mocks and other elements](#), we used Mockito annotations such as `@Mock` and `@InjectMocks` on the test fixture fields and let Mockito automatically inject the mocked objects with `MockitoAnnotations.initMocks(this)`.

In unit tests, it was easy to inject dependencies, even without using Mockito's dependency injection mechanisms: it was enough to create mocks and pass them to the SUT constructor.

In integration tests, things became a little more complex. If we have a look at the typical integration test setup steps for our view, which we have written in the previous chapters, they typically have the following shape

```
1 mongoClient = new MongoClient(new ServerAddress(serverAddress));
2 studentRepository =
3     new StudentMongoRepository(mongoClient,
4         SCHOOL_DB_NAME, STUDENT_COLLECTION_NAME);
5 student SwingView = new Student SwingView();
6 schoolController = new SchoolController(student SwingView, studentRepository);
7 student SwingView.setSchoolController(schoolController);
```

The steps required to wire all the components together are:

1. Create the `MongoClient`;
2. Create the `StudentMongoRepository` passing the `MongoClient`;
3. Create the `Student SwingView` (not fully initialized yet);
4. Create the `SchoolController` passing the `Student SwingView` and the `StudentMongoRepository`;
5. Conclude the initialization of the `Student SwingView` setting the `SchoolController`.

Thus, even in a simple application like the one we have developed, initializing all the objects, injecting all the dependencies, might not be trivial. The steps must also be executed in the right order, possibly dealing with cyclic dependencies, like the one between the view and the controller, which we have to manually handle. In particular, to break the mutual dependency we had to add a setter in the view. This has the drawback that once the view is created with its constructor it is not yet ready to be used until the controller is set. Forgetting the call to the setter method will make the application completely misbehave, due to several `NullPointerExceptions`.

This is where a dependency injection framework, like Guice, comes in to make the initialization of the *graph of objects* straightforward and easily configurable at runtime. Indeed, Guice heavily relies on Java reflection for creating instances and wire them together.

Before starting using Guice in our application, we will write learning tests to get familiar with this framework and to acquire confidence in our understanding of such a framework.

We create a brand new Maven Java project, where the `src/main/java` will be empty and we write only learning tests in `src/test/java`. Besides JUnit, this is the Maven dependency for using Google Guice:

```
1 <dependency>
2   <groupId>com.google.inject</groupId>
3   <artifactId>guice</artifactId>
4   <version>4.2.3</version>
5 </dependency>
```

We will use very simple classes, with no logic, for experimenting with Guice. We will not make the instance variables of these classes private: they do not have to be clean code, we will use them only for learning tests, thus it makes sense to allow the tests to directly access instance variables of objects.

### 16.1.1 Guice main concepts

Let's start writing a Java test case that we use to write our first learning tests. Since these tests are meant only for experimenting and learning Guice, we write the Java classes used in the tests directly in the test case, as `private static` classes.

```
1 package guice.learningtests;
2
3 public class GuiceLearningTest {
4 ...
5 }
```

First of all, Guice relies on Java annotations. Thus, we must annotate our class' members that are meant to be handled by Guice with the `@Inject` annotation, `com.google.inject.Inject`.<sup>1</sup>

For example, we have this class

```
1 private static class MyService {}
```

We want an instance of this class injected into another class through its constructor. We annotate the constructor with `@Inject`:

```
1 import com.google.inject.Inject;
2
3 private static class MyClient {
4     MyService service;
5
6     @Inject
7     public MyClient(MyService service) {
8         this.service = service;
9     }
10 }
```

Guice will be able to create an instance of `MyClient` and to figure out how to inject into its constructor annotated with `@Inject` an instance of `MyService`. If `MyService` had further dependencies, Guice would first resolve those dependencies, before passing the resulting object into `MyClient`'s constructor.

Now the question is: “how can we create an instance of `MyClient` with Guice?”

To create an instance with Guice we need a `com.google.inject.Injector` first. An `Injector` is created using the static method `Guice.createInjector`, which requires an argument of type `com.google.inject.Module`. A Guice module contributes configuration information. For this very first example, we will not need to specify any configuration.

We must either implement the `Module` interface or simply extend the abstract class (which already implements the interface `Module`), `AbstractModule`. This abstract class does not require implementing any method. We still need to create an anonymous inner class extending `AbstractModule`, since in Java you cannot instantiate an abstract class directly, even if it has no abstract methods.

Once we have an `Injector`, we can create an instance using its method `getInstance`, passing the Java `Class` of the object we want to create.

Summarizing, for the above code, these are the steps to create an instance of `MyClient` with Guice:

---

<sup>1</sup>Java itself has a specification for dependency injection, `JSR-330`, <https://javax-inject.github.io/javax-inject/>, defining its own `@Inject` annotation, `javax.inject.Inject`. Guice can handle such annotations as well, almost interchangeably, as detailed here: <https://github.com/google/guice/wiki/JSR330>. In this chapter, we use `com.google.inject.Inject`.

```
1 @Test
2 public void canInstantiateConcreteClassesWithoutConfiguration() {
3     Module module = new AbstractModule() {};
4     Injector injector = Guice.createInjector(module);
5     MyClient client = injector.getInstance(MyClient.class);
6     assertNotNull(client.service);
7 }
```

Without any further configuration, Guice can instantiate a `MyService` (and inject it into a `MyClient`) since `MyService` is a concrete class and `MyClient` specifies such a concrete class in its constructor.

This test passes.

Let's now consider an interface for the service

```
1 private static interface IMyService {}
2 private static class MyService implements IMyService {}
```

And a variant of the client that abstracts from the actual implementation of the service:

```
1 private static class MyGenericClient {
2     IMyService service;
3
4     @Inject
5     public MyGenericClient(IMyService service) {
6         this.service = service;
7     }
8 }
```

Now, Guice will not be able to instantiate such a class without further configuration. The call to `getInstance` will throw an exception:

```
1 @Test
2 public void injectAbstractType() {
3     Module module = new AbstractModule() {};
4     Injector injector = Guice.createInjector(module);
5     MyGenericClient client = injector.getInstance(MyGenericClient.class);
6     assertNotNull(client.service);
7 }
```

The exception contains the following information:

```
1 Exception in thread "main" com.google.inject.ConfigurationException:  
2 Guice configuration errors:  
3  
4 1) No implementation for IMyService was bound.  
5     while locating IMyService  
6         for the 1st parameter of MyGenericClient.<init>  
7     while locating MyGenericClient
```

In our module, we must implement the method `configure` and specify the **bindings**: a binding is meant to bind an abstract type to a concrete type. In the method `configure` bindings are specified using the fluent API of Guice, e.g.,

```
1 bind(AbstractType.class).to(ConcreteType.class)
```

Of course, this API makes use of Java generics and it is statically typed. Thus, `ConcreteType` must be a subtype of `AbstractType` or a compiler error will be issued.

Let's update our test and also verify the type of the actual injected instance:

```
1 @Test  
2 public void injectAbstractType() {  
3     Module module = new AbstractModule() {  
4         @Override  
5         protected void configure() {  
6             bind(IMyService.class).to(MyService.class);  
7         }  
8     };  
9     Injector injector = Guice.createInjector(module);  
10    MyGenericClient client = injector.getInstance(MyGenericClient.class);  
11    assertNotNull(client.service);  
12    assertEquals(MyService.class, client.service.getClass());  
13 }
```

Besides binding a type to a concrete type, we can also bind a type to an instance. In this case, each time Guice needs to instantiate the abstract type, it will always reuse the same instance. This can be verified by this test:

```
1  @Test
2  public void bindToInstance() {
3      Module module = new AbstractModule() {
4          @Override
5          protected void configure() {
6              bind(IMyService.class).toInstance(new MyService());
7          }
8      };
9      Injector injector = Guice.createInjector(module);
10     MyGenericClient client1 = injector.getInstance(MyGenericClient.class);
11     MyGenericClient client2 = injector.getInstance(MyGenericClient.class);
12     assertNotNull(client1.service);
13     assertSame(client1.service, client2.service);
14 }
```

## 16.1.2 Singleton

By default, Guice returns a new instance each time it supplies a value. This behavior is configurable via **scopes**. For example, Guice also provides the annotation `com.google.inject.Singleton` to be used on implementation classes when you want only one instance per Injector. Alternatively, scopes can also be configured in bind statements. Let's write another test case, `GuiceSingletonLearningTest` to experiment with singletons (`IMyService`, `MyService` and `MyClient` are as in the previous test case):

```
1  @Test
2  public void bindToSingleton() {
3      Module module = new AbstractModule() {
4          @Override
5          protected void configure() {
6              bind(IMyService.class).to(MyService.class);
7              bind(MyService.class).in(Singleton.class);
8          }
9      };
10     Injector injector = Guice.createInjector(module);
11     MyClient client1 = injector.getInstance(MyClient.class);
12     MyClient client2 = injector.getInstance(MyClient.class);
13     assertNotNull(client1.service);
14     assertSame(client1.service, client2.service);
15 }
16
17 @Test
18 public void singletonPerInjector() {
```

```
19     Module module = new AbstractModule() {
20         @Override
21         protected void configure() {
22             bind(IMyService.class).to(MyService.class).in(Singleton.class);
23         }
24     };
25     assertNotSame(
26         Guice.createInjector(module)
27             .getInstance(MyClient.class)
28             .service,
29         Guice.createInjector(module)
30             .getInstance(MyClient.class)
31             .service
32     );
33 }
```

We verify that a singleton is not like a global Java singleton (implemented manually): singleton in Guice means the same instance per injector. We also took the chance to learn that we can bind an abstract type to a concrete type and then bind the concrete type to a scope or that we can do both operations in a single bind statement. The two modules above are equivalent.



Try to write a learning test that verifies the behavior when using Guice `@Singleton` annotation.

### 16.1.3 Field and method injection

Guice can also directly *inject fields* (even private ones) annotated with `@Inject`. It can also *inject methods* annotated with `@Inject`. Dependencies take the form of parameters. Guice will first instantiate via injection the arguments of such parameters, before invoking the method. These methods must not necessarily be *setter* methods (the method name is not taken into consideration) and they can have any number of parameters.

Let's create another test case Java class, e.g., `GuiceFieldAndMethodInjectionLearningTest`. Here are two learning tests for these mechanisms (and the classes used in these tests):

```
1 private static class MyClientWithInjectedField {
2     @Inject
3     IMyService service;
4 }
5
6 private static class MyClientWithInjectedMethod {
7     IMyService service;
8
9     @Inject
10    public void init(IMyService service) {
11        this.service = service;
12    }
13 }
14
15 @Test
16 public void fieldAndMethodInjection() {
17     Module module = new AbstractModule() {
18         @Override
19         protected void configure() {
20             bind(IMyService.class).to(MyService.class);
21         }
22     };
23     Injector injector = Guice.createInjector(module);
24     MyClientWithInjectedField client1 =
25         injector.getInstance(MyClientWithInjectedField.class);
26     MyClientWithInjectedMethod client2 =
27         injector.getInstance(MyClientWithInjectedMethod.class);
28     assertNotNull(client1.service);
29     assertNotNull(client2.service);
30 }
```

Method and field injection can be used to initialize an existing instance. Thus, we can create an instance manually through its no-arg constructor and then inject members into this instance using `Injector.injectMembers`:

```
1  @Test
2  public void injectMembers() {
3      Module module = new AbstractModule() {
4          @Override
5          protected void configure() {
6              bind(IMyService.class).to(MyService.class);
7          }
8      };
9      Injector injector = Guice.createInjector(module);
10     MyClientWithInjectedField client1 = new MyClientWithInjectedField();
11     injector.injectMembers(client1);
12     MyClientWithInjectedMethod client2 = new MyClientWithInjectedMethod();
13     injector.injectMembers(client2);
14     assertNotNull(client1.service);
15     assertNotNull(client2.service);
16 }
```

Method and field injections may be *optional*: Guice will silently ignore them when the dependencies are not available. To use optional injection, use `@Inject(optional=true)`. This allows you to use a dependency when it exists and to fall back to a default otherwise.

## What to prefer

In general, constructor injection is to be preferred, since it allows you to declare `final` fields. Moreover, it allows you to create instances directly, without Guice, and this might be very useful in testing, using mocking, for example. However, constructor injection cannot be *optional*. Method injection is useful when you need to create an instance manually, not with Guice. It is also useful for optional or changeable dependencies. Field injection has the most compact syntax, but you will be able to create instances only with Guice (or with reflection).

### 16.1.4 Providers

If we need to control the creation of a dependency of a type `T`, instead of injecting an instance of type `T`, we can inject a `com.google.inject.Provider<T>` and use its method `get` to create an instance of the dependency.

Let's create another test case Java class, e.g., `GuiceProviderLearningTest`, and write a learning test. In a class that we instantiate with Guice, we inject a `Provider` and later use the provider to create an instance:

```
1 private static interface IMyService {}  
2 private static class MyService implements IMyService {}  
3  
4 private static class MyClientWithInjectedProvider {  
5     @Inject  
6     Provider<IMyService> serviceProvider;  
7  
8     IMyService getService() {  
9         return serviceProvider.get();  
10    }  
11 }  
12  
13 @Test  
14 public void injectProviderExample() {  
15     Module module = new AbstractModule() {  
16         @Override  
17         protected void configure() {  
18             bind(IMyService.class).to(MyService.class);  
19         }  
20     };  
21     Injector injector = Guice.createInjector(module);  
22     MyClientWithInjectedProvider client =  
23         injector.getInstance(MyClientWithInjectedProvider.class);  
24     assertNotNull(client.getService());  
25 }
```

If we need to have full control over the creation of an instance, for example, because it represents a dependency to be injected in other classes, but its class does not have `@Inject` annotations (e.g., it comes from a third-party package) we can bind its type to a `Provider`.

Such a binding looks like this

```
1 bind(Type.class).toProvider(.../* provider for Type */...)
```

The argument of `toProvider` must be a valid `Provider` for the type. We could write a class implementing the `Provider` interface. Being a functional interface, we can also pass directly a lambda.

For example, continuing writing inside `GuiceProviderLearningTest`, assuming we created a file in `src/test/resources/afile.txt`:

```
1 private static class MyFileWrapper {
2     @Inject
3     File file;
4 }
5
6 @Test
7 public void providerBinding() {
8     Module module = new AbstractModule() {
9         @Override
10        protected void configure() {
11            bind(File.class)
12                .toProvider(
13                    () -> new File("src/test/resources/afile.txt"));
14        }
15    };
16    Injector injector = Guice.createInjector(module);
17    MyFileWrapper fileWrapper = injector.getInstance(MyFileWrapper.class);
18    assertTrue(fileWrapper.file.exists());
19 }
```

## 16.1.5 Binding annotations

In case we need multiple bindings for the same type we can use a built-in Guice **binding annotation** like `@Named` (`com.google.inject.name.Named`), which takes a string as an argument. This allows us to inject, for example, different String objects (with different `@Named` annotations). We write this test in another test case `GuiceBindingAnnotationTest`:

```
1 ...
2 import com.google.inject.name.Named;
3 import com.google.inject.name.Names;
4 ...
5 private static class MyFileWrapper {
6     File file;
7
8     @Inject
9     public MyFileWrapper(@Named("PATH") String path, @Named("NAME") String name) {
10         file = new File(path, name);
11     }
12 }
13
14 @Test
15 public void bindingAnnotations() {
```

```
16     Module module = new AbstractModule() {
17         @Override
18         protected void configure() {
19             bind(String.class)
20                 .annotatedWith(Names.named("PATH"))
21                 .toInstance("src/test/resources");
22             bind(String.class)
23                 .annotatedWith(Names.named("NAME"))
24                 .toInstance("afile.txt");
25         }
26     };
27     Injector injector = Guice.createInjector(module);
28     MyFileWrapper fileWrapper = injector.getInstance(MyFileWrapper.class);
29     assertTrue(fileWrapper.file.exists());
30 }
```

Note that the compiler cannot check the strings used in @Named annotations and it is easy to misspell one of the strings. It is better to define our own binding annotation. To do that, it is enough to define a new Java annotation, using Guice @BindingAnnotation, specify the targets (in this example, fields, parameters and methods) and @Retention(RUNTIME):

```
1 import static java.lang.annotation.ElementType.*;
2 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3 import java.lang.annotation.Retention;
4 import java.lang.annotation.Target;
5 import com.google.inject.BindingAnnotation;
6 ...
7 @BindingAnnotation
8 @Target({ FIELD, PARAMETER, METHOD })
9 @Retention(RUNTIME)
10 private static @interface FilePath {}
11
12 @BindingAnnotation
13 @Target({ FIELD, PARAMETER, METHOD })
14 @Retention(RUNTIME)
15 private static @interface FileName {}
16
17 private static class MyFileWrapper2 {
18     File file;
19
20     @Inject
21     public MyFileWrapper2(@FilePath String path, @FileName String name) {
22         file = new File(path, name);
```

```

23     }
24 }
25
26 @Test
27 public void customBindingAnnotations() {
28     Module module = new AbstractModule() {
29         @Override
30         protected void configure() {
31             bind(String.class)
32                 .annotatedWith(FilePath.class)
33                 .toInstance("src/test/resources");
34             bind(String.class)
35                 .annotatedWith(FileName.class)
36                 .toInstance("afile.txt");
37         }
38     };
39     Injector injector = Guice.createInjector(module);
40     MyFileWrapper2 fileWrapper = injector.getInstance(MyFileWrapper2.class);
41     assertTrue(fileWrapper.file.exists());
42 }
```

## 16.1.6 Overriding bindings

If we extend one of our existing modules and in the derived module we try to provide a new binding for a type already bound in the superclass module we get a runtime exception. Instead of relying on class inheritance, Guice allows us to override bindings by composing module instances with the `Modules` API:

```
1 Modules.override(new DefaultModule()).with(new CustomModule())
```

Let's write a new test case for these learning tests, `GuiceModulesOverrideLearningTest`:

```

1 @Test
2 public void modulesOverride() {
3     Module defaultModule = new AbstractModule() {
4         @Override
5         protected void configure() {
6             bind(IMyService.class).to(MyService.class);
7         }
8     };
9     Injector injector = Guice.createInjector(defaultModule);
```

```

10 MyClient client1 = injector.getInstance(MyClient.class);
11 MyClient client2 = injector.getInstance(MyClient.class);
12 // not singleton
13 assertNotSame(client1.service, client2.service);
14 Module customModule = new AbstractModule() {
15     @Override
16     protected void configure() {
17         bind(MyService.class).in(Singleton.class);
18     }
19 };
20 injector = Guice.createInjector(
21     Modules.override(defaultModule).with(customModule)
22 );
23 client1 = injector.getInstance(MyClient.class);
24 client2 = injector.getInstance(MyClient.class);
25 assertNotNull(client1.service);
26 // now it is singleton
27 assertEquals(client1.service, client2.service);
28 }
```

We have overridden the binding for `MyService` so that it is now a singleton.

### 16.1.7 Factories and AssistedInject

There might be cases when we want a few dependencies to be injected by Guice and other dependencies to be provided directly when creating an instance. We can use the Factory pattern for this. For example, we define an interface for our factory with a `create` method accepting the parameters that will be provided directly, implement the factory that takes some injected dependencies and create the instance with the injected dependencies and the ones passed to the `create` method.

Guice can create the implementation of our factory interface on-the-fly. This mechanism, known as `AssistedInject`, is an extension to the Guice core and it is part of a separate JAR:

```

1 <dependency>
2   <groupId>com.google.inject.extensions</groupId>
3   <artifactId>guice-assistedinject</artifactId>
4   <version>4.2.3</version>
5 </dependency>
```

Let's write a new test case `GuiceAssistedInjectLearningTest`. This is a learning test showing how this mechanism works (we intentionally use interfaces and classes with names "view", "controller" and "repository" because we aim at recreating a scenario similar to the one of our application we developed in the previous chapters):

```
1 import com.google.inject.Injector;
2 import com.google.inject.FactoryModuleBuilder;
3 ...
4 private static interface IMyView {}
5 private static class MyView implements IMyView {}
6
7 private static interface IMyRepository {}
8 private static class MyRepository implements IMyRepository {}
9
10 private static interface IMyController {}
11 private static class MyController implements IMyController {
12     IMyView view;
13     IMyRepository repository;
14
15     @Inject
16     public MyController(
17         @Assisted IMyView view, // from the instance's creator
18         IMyRepository repository // from the Injector
19     ) {
20         this.view = view;
21         this.repository = repository;
22     }
23 }
24
25 private static interface MyControllerFactory {
26     IMyController create(IMyView view);
27 }
28
29 @Test
30 public void assistedInject() {
31     Module module = new AbstractModule() {
32         @Override
33         protected void configure() {
34             bind(IMyRepository.class).to(MyRepository.class);
35             install(new FactoryModuleBuilder()
36                 .implement(IMyController.class, MyController.class)
37                 .build(MyControllerFactory.class));
38         }
39     };
40     Injector injector = Guice.createInjector(module);
41     MyControllerFactory controllerFactory =
42         injector.getInstance(MyControllerFactory.class);
43     MyController controller = (MyController) controllerFactory.create(new MyView());
```

```
44     assertNotNull(controller.view);
45     assertNotNull(controller.repository);
46 }
```

The Guice AssistedInject mechanism automatically maps the factory's create method's parameters to the corresponding @Assisted parameters in the implementation class' constructor. Other constructor arguments will be injected as usual. It is then enough to use the Guice API using FactoryModuleBuilder to have Guice implement a factory implementation automatically.

## 16.1.8 Cyclic dependencies

Cyclic dependencies should in general be avoided. There are cases, though, when they are implied by a pattern, like in our MVC architecture that we developed in the previous chapters. By using Guice providers and AssistedInject we can get around the cyclic dependencies.

Let's recreate a context similar to our MVC application in a new test case

GuiceCyclicDependenciesLearningTest. For saving space, we will not show here interfaces and classes that are equal to the ones of the previous test case, unless when useful for better comprehension:

```
1 private static class MyView implements IMyView {
2     IMyController controller;
3
4     public void setController(IMyController controller) {
5         this.controller = controller;
6     }
7 }
8
9 private static class MyController implements IMyController {
10    IMyView view;
11    IMyRepository repository;
12
13    @Inject
14    public MyController(
15        @Assisted IMyView view, // from the instance's creator
16        IMyRepository repository // from the Injector
17    ) {
18        this.view = view;
19        this.repository = repository;
20    }
21 }
22
23 private static interface MyControllerFactory {
```

```
24     IMyController create(IMyView view);
25 }
26
27 private static class MyViewProvider implements Provider<MyView> {
28     @Inject
29     private MyControllerFactory controllerFactory;
30
31     @Override
32     public MyView get() {
33         // manually solve the cycle
34         MyView view = new MyView();
35         view.setController(controllerFactory.create(view));
36         return view;
37     }
38 }
39
40 @Test
41 public void cyclicDependencies() {
42     Module module = new AbstractModule() {
43         @Override
44         protected void configure() {
45             bind(IMyRepository.class).to(MyRepository.class);
46             bind(MyView.class).toProvider(MyViewProvider.class);
47             install(new FactoryModuleBuilder()
48                 .implement(IMyController.class, MyController.class)
49                 .build(MyControllerFactory.class));
50         }
51     };
52     Injector injector = Guice.createInjector(module);
53     MyView view = injector.getInstance(MyView.class);
54     assertEquals(view, ((MyController) view.controller).view);
55     assertNotNull(((MyController) view.controller).repository);
56 }
```

Note that the view still has a setter method (though it never uses inject annotations). However, the correct initialization is now performed internally by a provider (appropriately used in the module's bindings). When we create an instance of the view with Guice, the view is ready to be used, since the cyclic dependencies have already been resolved.

However, we have direct access to the view only. The controller is hidden in the view. If we need to use the controller directly, the view must somehow expose it, e.g., with a getter.

## 16.1.9 Provides

In case a Provider is only a few lines of code long, but it still needs some injected dependencies, instead of implementing a Provider class, we can simply create a method in the module, annotated with `@Provides` (`com.google.inject.Provides`). The name of the method is not relevant. The method's return type will be the bound type. Dependencies can be injected as parameters to the method.

For example, this test is an alternative to the previous one, where we do not use a custom provider. We simply define a `@Provides` method in the module:

```
1  @Test
2  public void providesMethod() {
3      Module module = new AbstractModule() {
4          @Override
5          protected void configure() {
6              bind(IMyRepository.class).to(MyRepository.class);
7              install(new FactoryModuleBuilder()
8                  .implement(IMyController.class, MyController.class)
9                  .build(MyControllerFactory.class));
10             // don't bind the provider here
11         }
12
13         @Provides // the parameter will be injected
14         MyView view(MyControllerFactory controllerFactory) {
15             MyView view = new MyView();
16             view.setController(controllerFactory.create(view));
17             return view;
18         }
19     };
20     Injector injector = Guice.createInjector(module);
21     MyView view = injector.getInstance(MyView.class);
22     assertEquals(view, ((MyController) view.controller).view);
23     assertNotNull(((MyController) view.controller).repository);
24 }
```

## 16.2 Apply our learnings

Now that we are quite confident that we learned how Guice works, we might be ready to use this dependency injection framework in our application.

We start from the application we developed in Chapter [End-to-end tests](#).

Of course, we add the Guice Maven dependencies that we used in the previous section.

Then, we prepare a few custom binding annotations, which we plan to use for the MongoDB host and port and the database and collection names:

```
1 package com.examples.school.guice;
2 ...
3 @BindingAnnotation
4 @Target({ FIELD, PARAMETER, METHOD })
5 @Retention(RUNTIME)
6 public @interface MongoHost {
7 }
```

Along the same line, we also define MongoPort, MongoDbName and MongoCollectionName.

Then, we use @Inject and some binding annotations in the StudentMongoRepository:

```
1 @Inject
2 public StudentMongoRepository(MongoClient client,
3     @MongoDbName String databaseName,
4     @MongoCollectionName String collectionName) {
5     studentCollection = client
6         .getDatabase(databaseName)
7         .getCollection(collectionName);
8 }
```

In the controller, we use @Assisted for the view (we will have to deal with the cyclic dependency), while the repository will be injected directly:

```
1 @Inject
2 public SchoolController(@Assisted StudentView studentView,
3     StudentRepository studentRepository) {
4     this.studentView = studentView;
5     this.studentRepository = studentRepository;
6 }
```

When the cyclic dependency is removed, we will only have direct access to the StudentSwipeView. Thus, we add a getter for retrieving the controller (which will be useful in tests) and we add a method start that will make the view visible and call the allStudents method of the controller:

```
1 public SchoolController getSchoolController() {
2     return schoolController;
3 }
4
5 public void start() {
6     setVisible(true);
7     schoolController.allStudents();
8 }
```

Note that these new methods and the Guice annotations will not break any existing test. We were already passing dependencies via constructors and the Guice annotations do not disturb the existing code that manually instantiates our objects.

We create a factory for our controller, that will be used with AssistedInject:

```
1 package com.examples.school.guice;
2 ...
3 public interface SchoolControllerFactory {
4     SchoolController create(StudentView view);
5 }
```

We create a custom Guice module for our application, which can be used in the main class and in integration tests.

We implement a fluent API (inspired by the **Builder** pattern) for setting the fields with the values for our custom binding annotations (the fields are initialized with a default value). Moreover, we configure the main bindings and write two @Provides methods, for the MongoClient and our Swing view:

```
1 package com.examples.school.guice;
2 ...
3 public class SchoolSwingMongoDefaultModule extends AbstractModule {
4     private String mongoHost = "localhost";
5     private int mongoPort = 27017;
6     private String databaseName = "school";
7     private String collectionName = "student";
8
9     public SchoolSwingMongoDefaultModule mongoHost(String mongoHost) {
10         this.mongoHost = mongoHost;
11         return this;
12     }
13
14     // mongoPort(), databaseName() and collectionName() are similar...
15 }
```

```
16  @Override
17  protected void configure() {
18      bind(String.class).annotatedWith(MongoHost.class).toInstance(mongoHost);
19      bind(Integer.class).annotatedWith(MongoPort.class).toInstance(mongoPort);
20      bind(String.class).annotatedWith(MongoDbName.class).toInstance(databaseName);
21      bind(String.class).annotatedWith(MongoCollectionName.class)
22          .toInstance(collectionName);
23
24      bind(StudentRepository.class).to(StudentMongoRepository.class);
25
26      install(new FactoryModuleBuilder()
27          .implement(SchoolController.class, SchoolController.class)
28          .build(SchoolControllerFactory.class));
29  }
30
31  @Provides
32  MongoClient mongoClient(@MongoHost String host, @MongoPort int port) {
33      return new MongoClient(host, port);
34  }
35
36  @Provides
37  Student SwingView studentView(SchoolControllerFactory schoolControllerFactory) {
38      Student SwingView view = new Student SwingView();
39      view.setSchoolController(schoolControllerFactory.create(view));
40      return view;
41  }
42 }
```

When writing the above module, we applied all the things we learned by writing learning tests in the previous section.

We do not need to explicitly bind `MongoClient` since we define a `@Provides` method. We do not need to bind `StudentView` to `Student SwingView` because when creating a `Student SwingView` in the `@Provides` method we use the `SchoolControllerFactory` explicitly.

Let's use this module right away in our main class, to create a view with Guice and start it:

```
1  @Command(mixinStandardHelpOptions = true)
2  public class SchoolSwingApp implements Callable<Void> {
3
4      @Option(names = { "--mongo-host" }, description = "MongoDB host address")
5      private String mongoHost = "localhost";
6      ... // as before
7      public static void main(String[] args) {
8          new CommandLine(new SchoolSwingApp()).execute(args);
9      }
10
11     @Override
12     public Void call() throws Exception {
13         EventQueue.invokeLater(() -> {
14             try {
15                 Guice.createInjector(
16                     new SchoolSwingMongoDefaultModule()
17                         .mongoHost(mongoHost)
18                         .mongoPort(mongoPort)
19                         .databaseName(databaseName)
20                         .collectionName(collectionName))
21                     .getInstance(StudentSwingView.class)
22                     .start();
23             } catch (Exception e) {
24                 ...
25             }
26         });
27         return null;
28     }
29 }
```

We have our e2e (and BDD) tests to verify that we did not break anything. All the tests still pass.

Of course, Guice required us to add a few annotations and methods. Moreover, the creation of the Guice module with the correct bindings requires some attention and effort. However, when this is done, the creation of the components of our application is straightforward. All the components will be automatically wired together by Guice, following our bindings.

Using Guice in our unit tests will provide no further benefit, since we rely on Mockito for injecting mocks.

We can use Guice in our integration tests, in particular, in `StudentSwingViewIT`, which we first created in Chapter [UI tests](#), Section [UI integration tests](#).

We @Inject a few fields for our test fixture

```
1 public class Student SwingViewIT extends AssertJSwingJUnitTestCase {  
2     ...  
3     @Inject  
4     private MongoClient mongoClient;  
5  
6     @Inject  
7     private Student SwingView student SwingView;  
8  
9     @Inject  
10    private StudentMongoRepository studentRepository;  
11  
12    // this will be retrieved from the view  
13    private SchoolController schoolController;  
14    ...
```

We inject such fields in the @Before method.

```
1 @Override  
2 protected void onSetUp() {  
3     final Module moduleForTesting =  
4         Modules  
5             .override(new School SwingMongoDefaultModule())  
6             .with(new AbstractModule() {  
7                 @Override  
8                 public void configure() {  
9                     bind(MongoClient.class)  
10                        .toInstance(new MongoClient(new ServerAddress(serverAddress)));  
11                }  
12            });  
13     final Injector injector = Guice.createInjector(moduleForTesting);  
14     GuiActionRunner.execute(() -> {  
15         injector.injectMembers(this);  
16         // explicit empty the database through the repository  
17         for (Student student : studentRepository.findAll()) {  
18             studentRepository.delete(student.getId());  
19         }  
20         schoolController = student SwingView.getSchoolController();  
21         return student SwingView;  
22     });  
23     window = new FrameFixture(robot(), student SwingView);  
24     window.show(); // shows the frame to test  
25 }
```

Here we use `Modules.override` starting from the `SchoolSwingMongoDefaultModule`: we configure the creation of the `MongoClient` since in this test we are using the in-memory database. The binding to a specific instance in the new module in this test overrides the binding in the default module (the `@Provides` method).

All the integration tests are still green.

If we had to apply Guice directly to our project, experimenting with Guice in our project, we would have a hard time and wasted a lot of time.

Instead, we learned Guice with learning tests, experimenting in a very simple context. Afterward, applying Guice to our project has been rather easy.

# Bibliography

- [App15] Frank Appel. *Testing with JUnit*. Packt Publishing, 2015.
- [Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.
- [Eva03] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley, 2003.
- [FG15] Viktor Farcic and Alex Garcia. *Test-Driven Java Development*. Packt Publishing, 2015.
- [Fow04] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. <https://www.martinfowler.com/articles/injection.html>, 2005.
- [Fow05] Martin Fowler. *FluentInterface*. <https://www.martinfowler.com/bliki/FluentInterface.html>, 2005.
- [Fow06] Martin Fowler. *Continuous Integration*. <https://www.martinfowler.com/articles/continuousIntegration.html>, 2006.
- [Fow18] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2nd edition, 2018.
- [Fra16] Nicolas Frankel. *Integration Testing from the Trenches*. Leanpub, 2016.
- [Gar17] Boni Garcia. *Mastering Software Testing with JUnit 5*. Packt Publishing, 2017.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [Mar00] Robert C. Martin. *Design Principles and Design Patterns*, 2000. PDF available through <https://en.wikipedia.org/wiki/SOLID>.
- [Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
- [Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.
- [PP18] Krunal Patel and Nilang Patel. *Java 9 Dependency Injection*. Packt, 2018.
- [Voc18] Ham Vocke. *The Practical Test Pyramid*. <https://www.martinfowler.com/articles/practical-test-pyramid.html>, 2018.