

JUnit

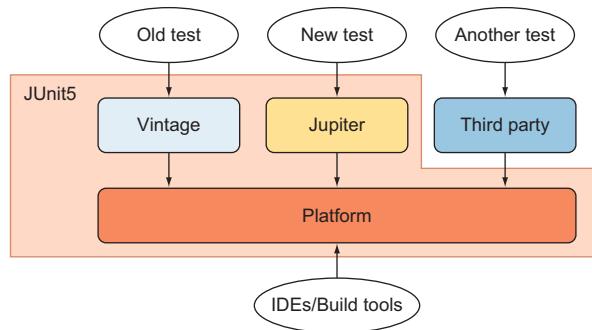
IN ACTION

THIRD EDITION

Cătălin Tudose



MANNING



JUnit in Action

THIRD EDITION

CĂTĂLIN TUDOSE



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2020 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.



Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Katie Sposato Johnson
Technical development editor: John Guthrie
Review editor: Mihaela Batinic
Production editor: Deirdre S. Hiam
Copy editor: Tiffany Taylor
Proofreader: Katie Tennant
Technical proofreader: David Cabrero
Typesetter and cover designer: Marija Tudor

ISBN 9781617297045

Printed in the United States of America

*This book is dedicated to all those people who made it possible:
family, friends, colleagues, professors, students.*

Cătălin Tudose

contents

preface *xiii*
acknowledgments *xv*
about this book *xvi*
about the author *xix*
about the cover illustration *xx*

PART 1 JUNIT 1

1 *JUnit jump-start* 3

- 1.1 Proving that a program works 4
- 1.2 Starting from scratch 6
 - Understanding unit testing frameworks* 8 □ *Adding unit tests* 9
- 1.3 Setting up JUnit 9
- 1.4 Testing with JUnit 12

2 *Exploring core JUnit* 15

- 2.1 Core annotations 16
 - The @DisplayName annotation* 19 □ *The @Disabled annotation* 20
- 2.2 Nested tests 21
- 2.3 Tagged tests 23
- 2.4 Assertions 25

2.5	Assumptions	29
2.6	Dependency injection in JUnit 5	31
	<i>TestInfoParameterResolver</i>	31
	<i>TestReporterParameterResolver</i>	32
	<i>RepetitionInfoParameterResolver</i>	33
2.7	Repeated tests	33
2.8	Parameterized tests	35
2.9	Dynamic tests	38
2.10	Using Hamcrest matchers	40

3 JUnit architecture 47

3.1	The concept and importance of software architecture	48
	<i>Story 1: The telephone directories</i>	48
	<i>Story 2: The sneakers manufacturer</i>	49
3.2	The JUnit 4 architecture	49
	<i>JUnit 4 modularity</i>	50
	<i>JUnit 4 runners</i>	50
	<i>JUnit 4 rules</i>	52
	<i>Shortcomings of the JUnit 4 architecture</i>	58
3.3	The JUnit 5 architecture	58
	<i>JUnit 5 modularity</i>	58
	<i>JUnit Platform</i>	59
	<i>JUnit Jupiter</i>	60
	<i>JUnit Vintage</i>	60
	<i>The big picture of the JUnit 5 architecture</i>	60

4 Migrating from JUnit 4 to JUnit 5 63

4.1	Migration steps between JUnit 4 and JUnit 5	64
4.2	Needed dependencies	65
4.3	Annotations, classes, and methods	67
	<i>Equivalent annotations, classes, and methods</i>	67
	<i>Categories vs. tags</i>	71
	<i>Migrating Hamcrest matcher functionality</i>	77
	<i>Rules vs. the extension model</i>	78
	<i>Custom rules</i>	82

5 Software testing principles 87

5.1	The need for unit tests	88
	<i>Allowing greater test coverage</i>	88
	<i>Increasing team productivity</i>	88
	<i>Detecting regressions and limiting debugging</i>	88
	<i>Refactoring with confidence</i>	89
	<i>Improving implementation</i>	90
	<i>Documenting expected behavior</i>	90
	<i>Enabling code coverage and other metrics</i>	91
5.2	Test types	92
	<i>Unit testing</i>	92
	<i>Integration software testing</i>	93
	<i>System software testing</i>	93
	<i>Acceptance software testing</i>	94

5.3	Black-box vs. white-box testing	94
	<i>Black-box testing</i>	95
	<i>White-box testing</i>	95
	<i>Pros and cons</i>	96

PART 2 DIFFERENT TESTING STRATEGIES 99

6 Test quality 101

6.1	Measuring test coverage	102
	<i>Introduction to test coverage</i>	102
	<i>Tools for measuring code coverage</i>	103
6.2	Writing testable code	108
	<i>Understanding that public APIs are contracts</i>	108
	<i>Reducing dependencies</i>	108
	<i>Creating simple constructors</i>	110
	<i>Following the Law of Demeter (Principle of Least Knowledge)</i>	110
	<i>Avoiding hidden dependencies and global state</i>	112
	<i>Favoring generic methods</i>	113
	<i>Favoring composition over inheritance</i>	114
	<i>Favoring polymorphism over conditionals</i>	114
6.3	Test-driven development	116
	<i>Adapting the development cycle</i>	116
	<i>Doing the TDD two-step</i>	117
6.4	Behavior-driven development	117
6.5	Mutation testing	118
6.6	Testing in the development cycle	119

7 Coarse-grained testing with stubs 123

7.1	Introducing stubs	124
7.2	Stubbing an HTTP connection	125
	<i>Choosing a stubbing solution</i>	128
	<i>Using Jetty as an embedded server</i>	128
7.3	Stubbing the web server resources	130
	<i>Setting up the first stub test</i>	130
	<i>Reviewing the first stub test</i>	133
7.4	Stubbing the connection	134
	<i>Producing a custom URL protocol handler</i>	134
	<i>Creating a JDK HttpURLConnection stub</i>	135
	<i>Running the test</i>	136

8 Testing with mock objects 138

8.1	Introducing mock objects	139
8.2	Unit testing with mock objects	139

- 8.3 Refactoring with mock objects 143
 - Refactoring example* 144 ▪ *Refactoring considerations* 145
- 8.4 Mocking an HTTP connection 147
 - Defining the mock objects* 147 ▪ *Testing a sample method* 148
 - Try #1: Easy refactoring technique* 149 ▪ *Try #2: Refactoring by using a class factory* 151
- 8.5 Using mocks as Trojan horses 154
- 8.6 Introducing mock frameworks 156
 - Using EasyMock* 157 ▪ *Using JMock* 162 ▪ *Using Mockito* 166

9 *In-container testing* 170

- 9.1 Limitations of standard unit testing 171
- 9.2 The mock-objects solution 172
- 9.3 The step to in-container testing 174
 - Implementation strategies* 174 ▪ *In-container testing frameworks* 175
- 9.4 Comparing stubs, mock objects, and in-container testing 175
 - Stubs evaluation* 175 ▪ *Mock-objects evaluation* 176
 - In-container testing evaluation* 177
- 9.5 Testing with Arquillian 178

PART 3 WORKING WITH JUNIT 5 AND OTHER TOOLS 187

10 *Running JUnit tests from Maven 3* 189

- 10.1 Setting up a Maven project 190
- 10.2 Using the Maven plugins 193
 - Maven compiler plugin* 194 ▪ *Maven Surefire plugin* 195
 - Generating HTML JUnit reports with Maven* 197
- 10.3 Putting it all together 198
- 10.4 Maven challenges 203

11 *Running JUnit tests from Gradle 6* 205

- 11.1 Introducing Gradle 206
- 11.2 Setting up a Gradle project 207
- 11.3 Using Gradle plugins 212

11.4 Creating a Gradle project from scratch and testing it with JUnit 5 213

11.5 Comparing Gradle and Maven 218

12 *JUnit 5 IDE support* 219

12.1 Using JUnit 5 with IntelliJ IDEA 220

12.2 Using JUnit 5 with Eclipse 226

12.3 Using JUnit 5 with NetBeans 232

12.4 Comparing JUnit 5 usage in IntelliJ, Eclipse, and NetBeans 238

13 *Continuous integration with JUnit 5* 240

13.1 Continuous integration testing 241

13.2 Introducing Jenkins 243

13.3 Practicing CI on a team 246

13.4 Configuring Jenkins 251

13.5 Working on tasks in a CI environment 255

PART 4 WORKING WITH MODERN FRAMEWORKS AND JUNIT 5 261

14 *JUnit 5 extension model* 263

14.1 Introducing the JUnit 5 extension model 264

14.2 Creating a JUnit 5 extension 264

14.3 Writing JUnit 5 tests using the available extension points 268

Persisting passengers to a database 268 ▪ *Checking the uniqueness of passengers* 277

15 *Presentation-layer testing* 281

15.1 Choosing a testing framework 282

15.2 Introducing HtmlUnit 282

A live example 282

15.3 Writing HtmlUnit tests 284

HTML assertions 284 ▪ *Testing for a specific web browser* 285

Testing more than one web browser 285 ▪ *Creating standalone tests* 286 ▪ *Testing forms* 288 ▪ *Testing JavaScript* 290

- 15.4 Introducing Selenium 294
- 15.5 Writing Selenium tests 295
 - Testing for a specific web browser* 297 ▪ *Testing navigation using a web browser* 298 ▪ *Testing more than one web browser* 299
 - Testing Google search and navigation using different web browsers* 301 ▪ *Testing website authentication* 303
- 15.6 HtmlUnit vs. Selenium 309

16 *Testing Spring applications* 311

- 16.1 Introducing the Spring Framework 311
- 16.2 Introducing dependency injection 312
- 16.3 Using and testing a Spring application 317
 - Creating the Spring context programmatically* 317 ▪ *Using the Spring TestContext framework* 321
- 16.4 Using SpringExtension for JUnit Jupiter 322
- 16.5 Adding a new feature and testing it with JUnit 5 325

17 *Testing Spring Boot applications* 333

- 17.1 Introducing Spring Boot 334
- 17.2 Creating a project with Spring Initializr 334
- 17.3 Moving the Spring application to Spring Boot 337
- 17.4 Implementing a test-specific configuration for Spring Boot 339
- 17.5 Adding and testing a new feature in the Spring Boot application 342

18 *Testing a REST API* 348

- 18.1 Introducing REST applications 349
- 18.2 Creating a RESTful API to manage one entity 350
- 18.3 Creating a RESTful API to manage two related entities 357
- 18.4 Testing the RESTful API 364

19 *Testing database applications* 369

- 19.1 The database unit testing impedance mismatch 370
 - Unit tests must exercise code in isolation* 370 ▪ *Unit tests must be easy to write and run* 370 ▪ *Unit tests must be fast to run* 371

19.2	Testing a JDBC application	372
19.3	Testing a Spring JDBC application	381
19.4	Testing a Hibernate application	388
19.5	Testing a Spring Hibernate application	393
19.6	Comparing the approaches for testing database applications	400

PART 5 DEVELOPING APPLICATIONS WITH JUNIT 5 ... 403

20 *Test-driven development with JUnit 5* 405

20.1	TDD main concepts	406
20.2	The flight-management application	407
20.3	Preparing the flight-management application for TDD	412
20.4	Refactoring the flight-management application	416
20.5	Introducing new features using TDD	420
	<i>Adding a premium flight</i>	420
	<i>Adding a passenger only once</i>	429

21 *Behavior-driven development with JUnit 5* 434

21.1	Introducing behavior-driven development	435
	<i>Introducing a new feature</i>	435
	<i>From requirements analysis to acceptance criteria</i>	436
	<i>BDD benefits and challenges</i>	437
21.2	Working BDD style with Cucumber and JUnit 5	437
	<i>Introducing Cucumber</i>	438
	<i>Moving a TDD feature to Cucumber</i>	439
	<i>Adding a new feature with the help of Cucumber</i>	447
21.3	Working BDD style with JBehave and JUnit 5	454
	<i>Introducing JBehave</i>	455
	<i>Moving a TDD feature to JBehave</i>	455
	<i>Adding a new feature with the help of JBehave</i>	461
21.4	Comparing Cucumber and JBehave	469

22 *Implementing a test pyramid strategy with JUnit 5* 471

22.1	Software testing levels	472
22.2	Unit testing: Basic components working in isolation	473
22.3	Integration testing: Units combined into a group	483

22.4 System testing: Looking at the complete software 491

Testing with a mock external dependency 491 ▪ *Testing with a partially implemented external dependency* 495 ▪ *Testing with the fully implemented external dependency* 498

22.5 Acceptance testing: Compliance with business requirements 501

appendix A Maven 513

appendix B Gradle 520

appendix C IDEs 525

appendix D Jenkins 528

index 531

****preface****

I am fortunate to have been in the IT industry for almost 25 years. I started programming in C++ and Delphi, and that is how I spent my student years and the first years of my career. I made the step from my mathematics background as a teenager to computer science and always kept both studies in mind. In 2000, my attention turned for the first time to the Java programming language, which was very young, but many people were predicting a great future for it. I was part of a team developing online games, using a particular technology: applets, which were extremely fashionable during those years. Our team spent some time developing and some more time testing, which was mainly done manually: we played together in the network and tried to discover the corner cases by ourselves. We hadn't heard about JUnit or test-driven development, which were in the pioneering stages.

After 2004, Java dominated about 90% of my work life. It was the dawn of a new era for me, and things like code refactoring, unit testing, and test-driven development became part of my normal professional life. Nowadays, I cannot imagine a project (even if it is a smaller one) without automated testing, and neither can Luxoft, the company I work for. My fellow developers talk about how they do automated testing in their current work, what the client expectations are, how they measure and increase code coverage, and how they analyze the quality of tests. Not only are unit testing and test-driven development at the heart of the conversation, but so also is behavior-driven development. We now cannot imagine shipping a product to fulfill market expectations without solid tests: an actual pyramid of unit tests, integration tests, system tests, and acceptance tests.

I was also fortunate to get in contact with Manning after having already developed three courses about automated testing for Pluralsight. I didn't have to start this book from scratch: the second edition was already a best seller. But it was written for 2010

and JUnit 4, and 10 years look like centuries in the IT field! I made the big step to JUnit 5 and present-day hot technologies and working methodologies. Unit testing and JUnit have come a long way since their early days when I began working with them. The concept is simple, but careful consideration and planning are required when migrating from JUnit 4 to 5. The book effectively provides that information, with many practical examples. I hope that this approach will help you decide what to do when you face new situations in your current work.

acknowledgments

The Manning team helped to create a high-level book, and I am looking forward to more opportunities of this kind.

I would like to thank my professors and colleagues for all their support during the years and to the many participants in my face-to-face or online courses – they represented a stimulus for me in achieving top quality work and always looking for improvement. Thanks to the co-authors of the book, Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory, for strong first editions that represented a good foundation. I hope to meet all of you in person some day. Best thoughts for my colleague and friend Vladimir Sonkin, with whom I share the steps in investigating new technologies.

I would also like to thank the staff at Manning: acquisition editor Mike Stephens, project editor Deirdre Hiam, development editor Katie Sposato Johnson, review editor Mihaela Batinic, technical development editor John Guthrie, technical proofer David Cabrero, senior technical development editor Al Scherer, copyeditor Tiffany Taylor, and proofreader Katie Tennant.

To all the reviewers: Andy Keffallas, Becky Huett, Burk Hufnagel, Conor Redmond, David Cabrero Souto, Ernesto Arroyo, Ferdinando Santacroce, Gaurav Tuli, Greg Wright, Gualtiero Testa, Gustavo Filipe Ramos Gomes, Hilde Van Gysel, Ivo Alexandre Costa Alves Angélico, Jean-François Morin, Joseph Tingsanchali, Junilu Lacar, Karthikeyarajan Rajendran, Kelum Prabath Senanayake, Kent R. Spillner, Kevin Orr, Paulo Cesar, Dias Lima, Robert Trausmuth, Robert Wenner, Sau Fai Fong, Shawn Ritchie, Sidharth Masaldaan, Simeon Leyzerzon, Srihari Sridharan, Thorsten P. Weber, Vittorio Marino, Vladimír Oraný, and Zorodzayi Mukuya. Your suggestions helped make this a better book.

about this book

JUnit in Action is a book about creating safe applications and how to greatly increase your development speed and remove much of the debugging nightmare—all with the help of JUnit 5 with its new features, and other tools and techniques that work in conjunction with JUnit 5.

The book focuses first on understanding the who, what, why, and how of JUnit. The first few chapters should convince you of the capabilities and power of JUnit 5. Following that, I take a deep dive into working effectively with JUnit 5: migrating from JUnit 4 to JUnit 5, testing strategies, working with JUnit 5 and different tools, working with modern frameworks, and developing applications with JUnit 5 according to present-day methodologies.

Who should read this book

This book is for application developers who are already proficient in writing Java Core code and are interested in learning how to develop safe and flexible applications. You should be familiar with object-oriented programming and have at least a working knowledge of Java. You will also need a working knowledge of Maven and be able to build a Maven project and open a Java program in IntelliJ IDEA, edit it, and launch it in execution. Some of the chapters require basic knowledge about technologies like Spring, Hibernate, REST, and Jakarta EE.

How this book is organized: A roadmap

This book has 22 chapters in five sections. Part 1 presents the JUnit 5 essentials:

- Chapter 1 gives you a quick introduction to the concepts of testing—knowledge you need to get started. You will get straight to the code, seeing how to write and execute a very simple test and see its results.

- Chapter 2 discusses JUnit in detail; you will see JUnit 5's capabilities and walk through the code that puts them in practice.
- Chapter 3 looks at the JUnit architecture.
- Chapter 4 discusses how to move from JUnit 4 to JUnit 5 and how to migrate projects between these versions of the framework.
- Chapter 5 is dedicated to tests as a whole. The chapter describes different kinds of tests and the scenarios to which they apply. It also discusses different levels of testing and the best scenarios in which to execute those tests.

Part 2 presents different testing strategies:

- Chapter 6 is dedicated to analyzing test quality. It introduces concepts such as code coverage, test-driven development, behavior-driven development, and mutating testing.
- Chapter 7 is dedicated to stubs, taking a look at a solution to isolate the environment and make tests seamless.
- Chapter 8 explains mock objects, providing an overview of how to construct and use them.
- Chapter 9 describes a different technique: executing tests in a container.

Part 3 shows how JUnit 5 works with other tools:

- Chapter 10 provides a very quick introduction to Maven and its terminology.
- Chapter 11 guides you through the same concepts, this time using another popular tool called Gradle.
- Chapter 12 investigates the way you can work with JUnit 5 by using the most popular IDEs today: IntelliJ IDEA, Eclipse, and NetBeans.
- Chapter 13 is devoted to continuous integration tools. This practice, which is highly recommended by extreme programmers, helps you maintain a code repository and automate the build on it.

Part 4 shows how JUnit 5 works with modern frameworks:

- Chapter 15 introduces HtmlUnit and Selenium. You will see how to test the presentation layer with these tools.
- Chapters 16 and 17 are dedicated to testing one of the most useful frameworks today: Spring. Spring is an open source application framework and inversion of control container for the Java platform. It includes several separate frameworks, including the Spring Boot convention-over-configuration solution for creating applications that you can run directly.
- Chapter 18 examines testing REST applications. Representational State Transfer is an application program interface that uses HTTP requests to GET, PUT, PATCH, POST, and DELETE data.

- Chapter 19 discusses alternatives for testing database applications, including JDBC, Spring, and Hibernate.

Part 5 shows how JUnit 5 works with modern software development methodologies:

- Chapter 20 discusses project development using one of today's popular development techniques: test-driven development.
- Chapter 21 discusses developing projects using behavior-driven development. It shows how to create applications that address business needs: applications that not only do things right but also do the right thing.
- Chapter 22 shows how to build a test pyramid strategy with the help of JUnit 5. It demonstrates testing from the ground level (unit testing) to the upper levels (integration testing, system testing, and acceptance testing).

In general, you can read this book from one chapter to the next. But, as long as you master the essentials presented in part 1, you can jump directly to any chapter that addresses your current needs.

About the code

This book contains (mostly) large blocks of code, rather than short snippets. Therefore, all the code listings are annotated and explained. In some chapters, annotations in listings and their explanations in text are marked with a number and the prime character to indicate comparisons with lines in similar listings. You can find the full source code for all these examples by downloading it from GitHub at <https://github.com/ctudose/junit-in-action-third-edition>.

liveBook discussion forum

Purchase of *JUnit in Action, Third Edition*, includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum, go to <https://livebook.manning.com/#!/book/junit-in-action-third-edition/discussion>. You can also learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/#!/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking them some challenging questions lest their interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the author

CĂTĂLIN TUDOSE is born in Pitești, Argeș, Romania.

He graduated with a degree in computer science in 1997, in Bucharest, and also completed a PhD in this field in 2006. He has more than 15 years of experience in the Java area. He took part in projects in telecommunications and finance, working as a senior software developer or technical team leader. He is currently acting as a Java and Web Technologies expert at Luxoft Romania.

He taught more than 2000 hours of courses and applications as a professor at the Faculty of Automation and Computers in Bucharest. He taught more than 4000 hours of Java courses at Luxoft, including the Corporate Junior Program, which has prepared about 50 new Java programmers in Poland. He also developed corporate courses on Java topics inside the company.

He taught online courses at UMGC (University of Maryland Global Campus): Computer Graphics with Java (CMSC 405), Intermediate Programming in Java (CMIS 242), Advanced Programming in Java (CMIS 440), Software Verification and Validation (SWEN 647), Database Concepts (IFSM 410), SQL (IFSM 411), Advanced Database Concepts (IFSM 420).

He has developed 5 courses for Pluralsight: TDD with JUnit 5; Java: BDD Fundamentals; Implementing A Test Pyramid Strategy in Java; Spring Framework: Aspect Oriented Programming with Spring AOP; and Migrating from the JUnit 4 to the JUnit 5 Testing Platform.

Besides the professional IT domain, he is interested in mathematics, world culture, and soccer. He is a lifelong supporter of his hometown team, FC Argeș Pitești.

about the cover illustration

The figure on the cover of *JUnit in Action, Third Edition* is captioned “Dame Walaque,” or Walaque lady. The illustration is taken from a collection of dress costumes from various countries by Jacques Grasset de Saint-Sauveur (1757-1810), titled *Costumes de Différents Pays*, published in France in 1797. Each illustration is finely drawn and colored by hand. The rich variety of Grasset de Saint-Sauveur’s collection reminds us vividly of how culturally apart the world’s towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

The way we dress has changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or countries. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Grasset de Saint-Sauveur’s pictures.

Part 1

JUnit

Welcome to *JUnit in Action*, which covers the JUnit framework, started by Kent Beck and Erich Gamma in late 1995. Ever since then, the popularity of the framework has been growing; now JUnit is the de facto standard for unit testing Java applications.

This book is the third edition. The first edition was a best seller, written by Vincent Massol and Ted Husted in 2003 and dedicated to version 3.x of JUnit. The second edition was also a best seller, written by Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory in 2010, and dedicated to version 4.x of JUnit.

In this edition, I cover version 5.x of JUnit—the newest version—and talk about lots of features included in it. At the same time, I focus on interesting details and techniques for testing your code: the architecture of the framework, test quality, mock objects, interaction with other tools, and JUnit extensions, as well as testing layers of your application, applying the test-driven development and behavior-driven development techniques, and so forth.

This part of the book explores JUnit itself. Chapter 1 gives you a quick introduction to the concepts of testing—knowledge you need to get started. I get straight to the code, showing you how to write and execute a very simple test and see its results. Chapter 2 introduces JUnit in detail. I show JUnit 5’s capabilities and walk through the code that puts them in practice. Chapter 3 looks at JUnit architectures, and chapter 4 discusses how to make the step from JUnit 4 to JUnit 5 and how to migrate projects between these versions of the framework. Chapter 5 is dedicated to tests as a whole. The chapter describes different kinds of tests and the scenarios to which they apply. I discuss different levels of testing and the best scenarios in which to execute those tests.

JUnit jump-start



This chapter covers

- Understanding JUnit
- Installing JUnit
- Writing your first tests
- Running tests

Never in the field of software development was so much owed by so many to so few lines of code.

—Martin Fowler

All code needs to be tested. During development, the first thing we do is run our own programmer's acceptance test. We code, compile, and run. When we run, we test. The test may just consist of clicking a button to see whether it brings up the expected menu or looking at a result to compare it with the expected value. Nevertheless, every day, we code, we compile, we run, and *we test*.

When we test, we often find issues, especially during early runs. So we code, compile, run, and test again.

Most of us quickly develop a pattern for our informal tests: add a record, view a record, edit a record, and delete a record. Running a little test suite like this by hand is easy enough to do, so we do it—over and over again.

Some programmers like doing this type of repetitive testing. It can be a pleasant break from deep thought and hardcoded. When our little click-through tests finally succeed, we have a real feeling of accomplishment (“Eureka! I found it!”).

Other programmers dislike this type of repetitive work. Rather than run the tests by hand, they prefer to create a small program that runs the tests automatically. Play-testing code is one thing; running automated tests is another.

If you’re a “play-test” developer, this book is for you. It shows you that creating automated tests can be easy, effective, and even fun.

If you’re already test-infected,¹ this book is also for you. We cover the basics in part 1 and move on to tough, real-life problems in parts 2–5.

1.1 **Proving that a program works**

Some developers feel that automated tests are essential parts of the development process: you cannot *prove* that a component works until it passes a comprehensive series of tests. In fact, two developers felt that this type of unit testing was so important that it deserved its own framework. In 1997, Erich Gamma and Kent Beck created a simple but effective unit testing framework for Java called *JUnit*: they were on a long plane trip, and it gave them something interesting to do. Erich wanted Kent to learn Java, and Erich was interested in knowing more about the SUnit testing framework that Kent created earlier for Smalltalk, and the flight gave them time to do both.

DEFINITION *Framework*—A semicomplete application that provides a reusable common structure to share among applications.² Developers incorporate the framework into their own applications and extend it to meet their specific needs. Frameworks differ from toolkits by providing a coherent structure rather than a simple set of utility classes. A framework defines a skeleton, and the application defines its own features to fill out the skeleton. The developer code is called appropriately by the framework. Developers can worry less about whether a design is good and focus more on implementing domain-specific functions.

If you recognize the names Erich Gamma and Kent Beck, that’s for a good reason. Gamma is one of the Gang of Four who gave us the now-classic *Design Patterns* book.³ Beck is equally well known for his groundbreaking work in the software discipline known as Extreme Programming (www.extremeprogramming.org).

¹ *Test-infected* is a term coined by Erich Gamma and Kent Beck in “Test-Infected: Programmers Love Writing Tests,” *Java Report* 3 (7), 37–50, 1998.

² Ralph Johnson and Brian Foote, “Designing Reusable Classes,” *Journal of Object-Oriented Programming* 1 (2): 22–35, 1988; www.laputan.org/drc/drc.html.

³ Erich Gamma et al., *Design Patterns* (Reading, MA: Addison-Wesley, 1995).

JUnit quickly became the de facto standard framework for developing unit tests in Java. Today, JUnit (<https://junit.org>) is open source software hosted on GitHub, with an Eclipse Public License. And the underlying testing model, known as *xUnit*, is on its way to becoming the standard framework for any language. xUnit frameworks are available for ASP, C++, C#, Eiffel, Delphi, Perl, PHP, Python, Rebol, Smalltalk, and Visual Basic—to name just a few.

The JUnit team didn't invent software testing or even unit tests, of course. Originally, the term *unit test* described a test that examined the behavior of a single unit of work: a class or a method. Over time, the use of the term *unit test* broadened. The Institute of Electrical and Electronics Engineers (IEEE), for example, has defined unit testing as “testing of individual hardware or software units *or groups of related units*” (emphasis added).⁴

In this book, we use the term *unit test* in the narrower sense to mean a test that examines a single unit in isolation from other units. We focus on the type of small, incremental test that programmers apply to their own code. Sometimes, these tests are called *programmer tests* to differentiate them from quality-assurance or customer tests (<http://c2.com/cgi/wiki?ProgrammerTest>).

Here is a generic description of a typical unit test from the perspective of this book: “Confirms that the method accepts the expected range of input and that the method returns the expected value for each input.” This description asks us to test the behavior of a method through its interface. If we give it value *x*, will it return value *y*? If we give it value *z* instead, will it throw the proper exception?

DEFINITION *Unit test*—A test that examines the behavior of a distinct unit of work. A *unit of work* is a task that is not directly dependent on the completion of any other task. Within a Java application, the distinct unit of work is often, but not always, a single method. In contrast, *integration tests* and *acceptance tests* examine how various components interact.

Unit tests often focus on testing whether a method is following the terms of its API contract. Like a written contract between people who agree to exchange certain goods or services under specific conditions, an *API contract* is a formal agreement made by the signature of a method. A method requires its callers to provide specific object references or primitive values and returns an object reference or primitive value. If the method cannot fulfill the contract, the test should throw an exception, and we say that the method has *broken* its contract.

DEFINITION *API contract*—A view of an application programming interface (API) as a formal agreement between the caller and the callee. Often, unit tests help define the API contract by demonstrating the expected behavior. The notion of an API contract arises from the practice of Design by Contract,

⁴ IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries (New York: IEEE, 1990).

popularized by the Eiffel programming language (<http://archive.eiffel.com/doc/manuals/technology/contract>).

In this chapter, we walk through creating a unit test from scratch for a simple class. We start by writing a test and its minimal runtime framework so you can see how things used to be done. Then we roll out JUnit to show you how the right tools can make life much simpler.

1.2 **Starting from scratch**

For our first example, we will create a very simple `Calculator` class that adds two numbers. Our calculator, shown in the following listing, provides an API to clients and does not contain a user interface. To test its functionality, we'll first create our own pure Java tests and later move to JUnit 5.

Listing 1.1 The Calculator class to be tested

```
public class Calculator {
    public double add(double number1, double number2) {
        return number1 + number2;
    }
}
```

Although the documentation isn't shown, the intended purpose of `Calculator`'s `add(double, double)` method is to take two doubles and return the sum as a double. The compiler can tell you that the code compiles, but you should also make sure it works at runtime. A core principle of unit testing is, "Any program feature without an automated test simply doesn't exist."⁵ The `add` method represents a core feature of the calculator. You have some code that allegedly implements the feature. What is missing is an automated test that proves the implementation works.

Isn't the add method too simple to break?

The current implementation of the `add` method is too simple to break with usual, everyday calculations. If `add` were a minor utility method, you might not test it directly. In that case, if `add` did fail, tests of the methods that used `add` would fail. The `add` method would be tested indirectly, but tested nonetheless. In the context of the calculator program, `add` is not just a method, but also a *program feature*. To have confidence in the program, most developers would expect there to be an automated test for the `add` feature, no matter how simple the implementation appears to be. In some cases, you can prove program features through automatic functional tests or automatic acceptance tests. For more about software tests in general, see chapter 5.

⁵ Kent Beck, *Extreme Programming Explained: Embrace Change* (Reading, MA: Addison-Wesley, 1999).

Yet testing anything at this point seems to be problematic. You do not even have a user interface with which to enter a pair of doubles. You could write a small command-line program that waited for you to type two double values and then displayed the result. Then, of course, you would also be testing your own ability to type a number and add the result yourself, which is much more than you want to do. You just want to know whether this unit of work actually adds two doubles and returns the correct sum. You do not want to test whether programmers can type numbers!

Meanwhile, if you are going to go to the effort of testing your work, you should also try to preserve that effort. It is good to know that the `add(double, double)` method worked when you wrote it. What you really want to know, however, is whether the method works when you ship the rest of the application or whenever you make a subsequent modification. If we put these requirements together, we come up with the idea of writing a simple test program for the `add` method.

The test program can pass known values to the method and see whether the result matches expectations. You can also run the program again later to be sure the method continues to work as the application grows. So what is the simplest possible test program you could write? What about this `CalculatorTest` program?

Listing 1.2 A simple test calculator program

```
public class CalculatorTest {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        double result = calculator.add(10, 50);
        if (result != 60) {
            System.out.println("Bad result: " + result);
        }
    }
}
```

`CalculatorTest` is simple indeed: it creates an instance of `Calculator`, passes two numbers to it, and checks the result. If the result does not meet your expectations, you print a message on standard output.

If you compile and run this program now, the test quietly passes, and all seems to be well. But what happens if you change the code so that it fails? You have to watch the screen carefully for the error message. You may not have to supply the input, but you are still testing your own ability to monitor the program's output. You want to test the code, not yourself!

The conventional way to signal error conditions in Java is to throw an exception. Let's throw an exception to indicate a test failure.

Meanwhile, you may also want to run tests for other `Calculator` methods that you have not written yet, such as `subtract` or `multiply`. Moving to a modular design will make catching and handling exceptions easier; it will also be easier to extend the test program later. The next listing shows a slightly better `CalculatorTest` program.

Listing 1.3 A (slightly) better test calculator program

```

public class CalculatorTest {

    private int nbErrors = 0;

    public void testAdd() {
        Calculator calculator = new Calculator();
        double result = calculator.add(10, 50);
        if (result != 60) {
            throw new IllegalStateException("Bad result: " + result);
        }
    }

    public static void main(String[] args) {
        CalculatorTest test = new CalculatorTest();
        try {
            test.testAdd();
        }
        catch (Throwable e) {
            test.nbErrors++;
            e.printStackTrace();
        }
        if (test.nbErrors > 0) {
            throw new IllegalStateException("There were " + test.nbErrors
                + " error(s)");
        }
    }
}

```



At ①, you move the test into its own `testAdd` method. Now it's easier to focus on what the test does. You can also add more methods with more unit tests later without making the `main` method harder to maintain. At ②, you change the `main` method to print a stack trace when an error occurs; then, if there are any errors, you end by throwing a summary exception.

Now that you have looked at a simple application and its tests, you can see that even this small class and its tests can benefit from the little bit of skeleton code you created to run and manage test results. But as an application gets more complicated and the tests become more involved, continuing to build and maintain a custom testing framework becomes a burden.

Next, we take a step back and look at the general case for a unit testing framework.

1.2.1 **Understanding unit testing frameworks**

Unit testing has several best practices that frameworks should follow. The seemingly minor improvements in the `CalculatorTest` program in listing 1.3 highlight three rules that (in my experience) all unit testing frameworks should follow:

- Each unit test should run independently of all other unit tests.
- The framework should detect and report errors test by test.
- It should be easy to define which unit tests will run.

The “slightly better” test program comes close to following these rules but still falls short. For each unit test to be truly independent, for example, each should run in a different class instance.

1.2.2 Adding unit tests

You can add new unit tests by adding a new method and then adding a corresponding `try/catch` block to `main`. This is a step up but still short of what you would want in a real unit test suite. Experience tells us that large `try-catch` blocks cause maintenance problems. You could easily leave out a unit test and never know it!

It would be nice if you could just add new test methods and continue working, but if you did, how would the program know which methods to run? Well, you could have a simple registration procedure. A registration method would at least inventory which tests are running.

Another approach would be to use Java’s reflection capabilities. A program could look at itself and decide to run whatever methods follow a certain naming convention, such as those that begin with `test`.

Making it easy to add tests (the third rule in the earlier list) sounds like another good rule for a unit testing framework. The support code that realizes this rule (via registration or reflection) would not be trivial, but it would be worthwhile. You’d have to do a lot of work up front, but that effort would pay off each time you add a new test.

Fortunately, the JUnit team has saved you the trouble. The JUnit framework already supports discovering methods. It also supports using a different class instance and class loader instance for each test and reports all errors on a test-by-test basis. The team has defined three discrete goals for the framework:

- The framework must help us write useful tests.
- The framework must help us create tests that retain their value over time.
- The framework must help us lower the cost of writing tests by reusing code.

We’ll discuss these goals further in chapter 2.

Next, let’s see how to set up JUnit.

1.3 Setting up JUnit

To use JUnit to write your application tests, you need to know about its dependencies. You’ll work with JUnit 5, the latest version of the framework when this book was written. Version 5 of the testing framework is a modular one; you can no longer simply add a jar file to your project compilation classpath and your execution classpath. In fact, starting with version 5, the architecture is no longer monolithic (as discussed in chapter 3). Also, with the introduction of annotations in Java 5, JUnit has also moved to using them. JUnit 5 is heavily based on annotations—a contrast with the idea of extending a base class for all testing classes and using naming conventions for all testing methods to match the `testXYZ` pattern, as done in previous versions.

NOTE If you are familiar with JUnit 4, you may wonder what’s new in this version, as well as why and how to move toward it. JUnit 5 represents the next

generation of JUnit. You'll use the programming capabilities introduced starting with Java 8; you'll be able to build tests modularly and hierarchically; and the tests will be easier to understand, maintain, and extend. Chapter 4 discusses the transition from JUnit 4 to JUnit 5 and shows that the projects you are working on may benefit from the great features of JUnit 5. As you'll see, you can make this transition smoothly, in small steps.

To manage JUnit 5's dependencies efficiently, it's logical to work with the help of a build tool. In this book, we'll use Maven, a very popular build tool. Chapter 10 is dedicated to the topic of running JUnit tests from Maven. What you need to know now are the basic ideas behind Maven: configuring your project through the pom.xml file, executing the `mvn clean install` command, and understanding the command's effects.

NOTE You can download Maven from <https://maven.apache.org>. When this book was being written, the latest version was 3.6.3.

The dependencies that are always needed in the pom.xml file are shown in the following listing. In the beginning, you need only `junit-jupiter-api` and `junit-jupiter-engine`.

Listing 1.4 pom.xml JUnit 5 dependencies

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>
</dependency>
```

To be able to run tests from the command prompt, make sure your pom.xml configuration file includes a JUnit provider dependency for the Maven Surefire plugin. Here's what this dependency looks like.

Listing 1.5 Maven Surefire plugin configuration in pom.xml

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version>
        </plugin>
    </plugins>
</build>
```

As Windows is the most commonly used operating system (OS), our example configuration details use Windows 10, the latest version. Concepts such as the path, environment variables, and the command prompt also exist in other OSs; follow your documentation guidelines if you will be running the examples on an OS other than Windows.

To run the tests, the bin folder from the Maven directory must be on the OS path (figure 1.1). You also need to configure the JAVA_HOME environment variable on your OS to point to the Java installation folder (figure 1.2). In addition, your JDK version must be at least 8, as required by JUnit 5.

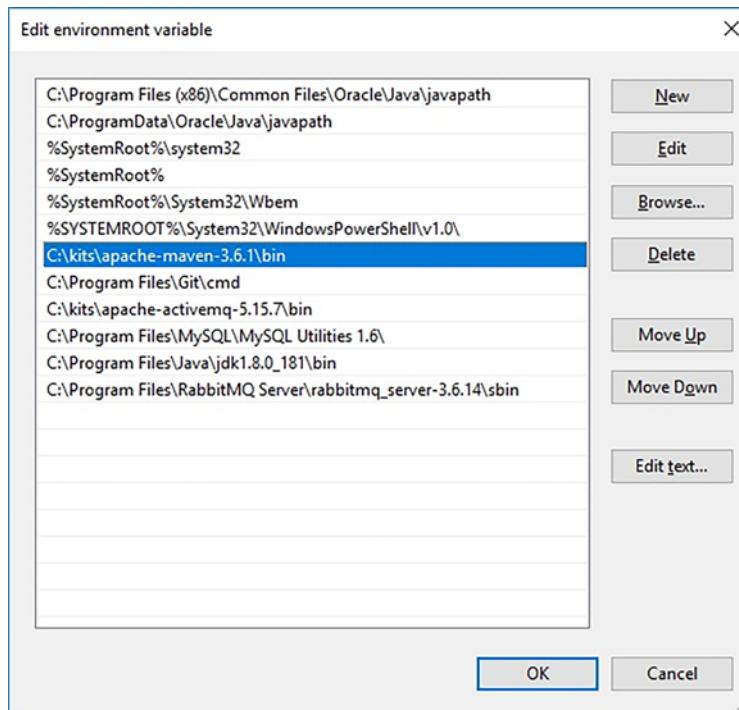


Figure 1.1 The configuration of the OS path must include the Apache Maven bin folder.

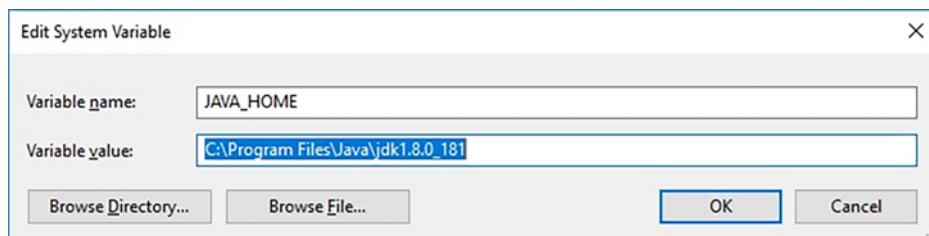
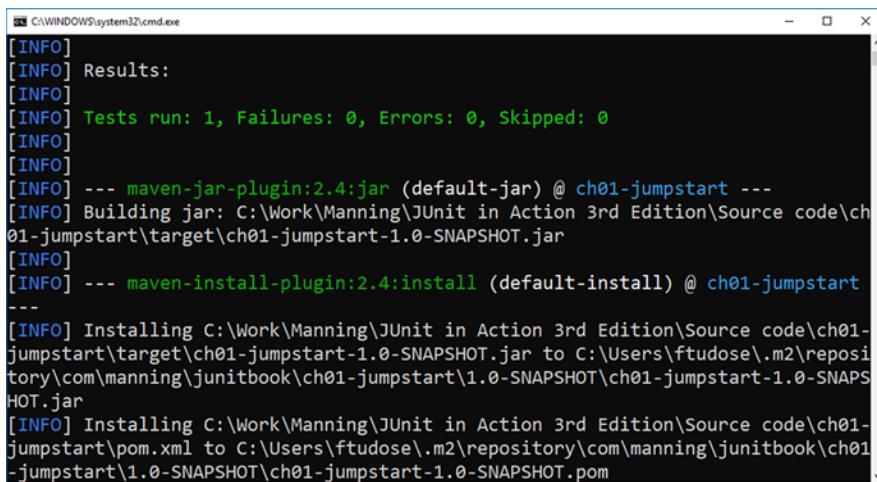


Figure 1.2 The configuration of the JAVA_HOME environment variable

You will need the source files from the chapter to get the results shown in figure 1.3. Open a command prompt into the project folder (the one containing the pom.xml file), and run this command:

```
mvn clean install
```

This command will take the Java source code, compile it, test it, and convert it into a runnable Java program (a jar file, in our case). Figure 1.3 shows the result of the test.

A screenshot of a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text:

```
[INFO] [INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ ch01-jumpstart ---
[INFO] Building jar: C:\Work\Manning\JUnit in Action 3rd Edition\Source code\ch01-jumpstart\target\ch01-jumpstart-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ ch01-jumpstart ---
[INFO] Installing C:\Work\Manning\JUnit in Action 3rd Edition\Source code\ch01-jumpstart\target\ch01-jumpstart-1.0-SNAPSHOT.jar to C:\Users\ftudose\.m2\repository\com\manning\junitbook\ch01-jumpstart\1.0-SNAPSHOT\ch01-jumpstart-1.0-SNAPSHOT.jar
[INFO] Installing C:\Work\Manning\JUnit in Action 3rd Edition\Source code\ch01-jumpstart\pom.xml to C:\Users\ftudose\.m2\repository\com\manning\junitbook\ch01-jumpstart\1.0-SNAPSHOT\ch01-jumpstart-1.0-SNAPSHOT.pom
```

Figure 1.3 Execution of the JUnit tests using Maven and the command prompt

Part 3 of the book provides more details about running tests with the Maven and Gradle build tools.

1.4 **Testing with JUnit**

JUnit has many features that make writing and running tests easy. You'll see these features at work throughout this book:

- Separate test class instances and class loaders for each unit test to prevent side effects
- JUnit annotations to provide resource initialization and cleanup methods: `@BeforeEach`, `@BeforeAll`, `@AfterEach`, and `@AfterAll` (starting from version 5); and `@Before`, `@BeforeClass`, `@After`, and `@AfterClass` (up to version 4)
- A variety of assert methods that make it easy to check the results of your tests
- Integration with popular tools such as Maven and Gradle, as well as popular integrated development environments (IDEs) such as Eclipse, NetBeans, and IntelliJ

Without further ado, the next listing shows what the simple Calculator test looks like when written with JUnit.

Listing 1.6 JUnit CalculatorTest program

```

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        double result = calculator.add(10, 50);
        assertEquals(60, result, 0);
    }
}

```

Running such a test with the help of Maven results in behavior similar to that shown in figure 1.3. The test is very simple. At ①, you define a test class. It's common practice to end the class name with `Test`. JUnit 3 required extending the `TestCase` class, but this requirement was removed with JUnit 4. Also, up to JUnit 4, the class had to be public; starting with version 5, the top-level test class can be public or package-private, and you can name it whatever you want.

At ②, you mark the method as a unit test method by adding the `@Test` annotation. In the past, the usual practice was to name test methods following the `testXYZ` pattern, as was required up to JUnit 3. Now that doing so is no longer required, some programmers drop the prefix and use a descriptive phrase as the method name. You can name your methods as you like; as long as they have the `@Test` annotation, JUnit will execute them. The JUnit 5 `@Test` annotation belongs to a new package, `org.junit.jupiter.api`, and the JUnit 4 `@Test` annotation belongs to the `org.junit` package. This book uses JUnit 5's capabilities except in some clearly emphasized cases (such as to demonstrate migration from JUnit 4).

At ③, you start the test by creating an instance of the `Calculator` class (the object under test). And at ④, as before, you execute the test by calling the method to test, passing it two known values.

At ⑤, the JUnit framework begins to shine! To check the result of the test, you call an `assertEquals` method, which you imported with a static import on the first line of the class. The Javadoc for the `assertEquals` method is

```

/**
 * Assert that expected and actual are equal within the non-negative delta.
 * Equality imposed by this method is consistent with Double.equals(Object)
 * and Double.compare(double, double). */
public static void assertEquals(
    double expected, double actual, double delta)

```

In listing 1.6, you pass these parameters to `assertEquals`:

```

expected = 60
actual = result
delta = 0

```

Because you pass the calculator the values 10 and 50, you tell `assertEquals` to expect the sum to be 60. (You pass 0 as `delta` because you are expecting no floating-point errors when adding 10 and 50, as the decimal part of these numbers is 0.) When you call the `calculator` object, you save the return value in a local double named `result`. Therefore, you pass that variable to `assertEquals` to compare with the expected value (60). If the actual value is not equal to the expected value, JUnit throws an unchecked exception, which causes the test to fail.

Most often, the `delta` parameter can be 0, and you can safely ignore it. This parameter comes into play with calculations that are not always precise, including many floating-point calculations. `delta` provides a range factor: if the actual value is within the range `expected - delta` and `expected + delta`, the test will pass. You may find this useful when you're performing mathematical computations with rounding or truncating errors or when you're asserting a condition about the modification date of a file, because the precision of these dates depends on the OS.

The remarkable thing about the JUnit `CalculatorTest` class in listing 1.6 is that the code is easier to write than the first `CalculatorTest` program in listings 1.2 or 1.3. In addition, you can run the test automatically through the JUnit framework.

When you run the test from the command line (figure 1.3), you see the amount of time it takes and the number of tests that passed. There are many other ways to run tests, from IDEs and from different build tools. This simple example gives you a taste of the power of JUnit and unit testing.

You may modify the `Calculator` class so that it has a bug—for example, instead of adding the numbers, it subtracts them. Then you can run the test and watch what the result looks like when a test fails.

In chapter 2, we will take a closer look at the JUnit framework classes (annotations and assertion mechanisms) and capabilities (nested and tagged tests, as well as repeated, parameterized, and dynamic tests). We'll show how they work together to make unit testing efficient and effective. You will learn how to use the JUnit 5 features in practice and differences between the old-style JUnit 4 and JUnit 5.

Summary

This chapter has covered the following:

- Why every developer should perform some type of test to see if code actually works. Developers who use automatic unit tests can repeat these tests on demand to ensure that new code works *and does not break existing tests*.
- Writing simple unit tests, which are not difficult to create without JUnit.
- As tests are added and become more complex, writing and maintaining tests becomes more difficult.
- Introduction to JUnit as a unit testing framework that makes it easier to create, run, and revise unit tests.
- Stepping through a simple JUnit test.

Exploring core JUnit



This chapter covers

- Understanding the JUnit life cycle
- Working with the core JUnit classes, methods, and annotations
- Demonstrating the JUnit mechanisms

Mistakes are the portals of discovery.

—James Joyce

In chapter 1, we decided that we need a reliable, repeatable way to test programs. Our solution is to write or reuse a framework to drive the test code that exercises our program API. As our program grows, with new classes and new methods added to the existing classes, we need to grow our test code as well. Experience has taught us that classes sometimes interact in unexpected ways, so we need to make sure we can run all of our tests at any time, no matter what code changes have taken place. But how do we run multiple test classes? How do we find out which tests passed and which ones failed?

In this chapter, we look at how JUnit provides the functionality to answer those questions. The chapter begins with an overview of the core JUnit concepts: the test

class, methods, and annotations. Then we take a detailed look at the many testing mechanisms of JUnit 5 and the JUnit life cycle.

This chapter is written in the practical spirit of the Manning “in Action” series, looking mainly at the usage of the new core features. For comprehensive documentation of each class, method, and annotation, please visit the JUnit 5 user guide (<https://junit.org/junit5/docs/current/user-guide>) or the JUnit 5 Javadoc (<https://junit.org/junit5/docs/current/api>).

The chapter introduces Tested Data Systems Inc., an example company that uses the testing mechanisms. Tested Data Systems is an outsourcing company that runs several Java projects for a few customers. These projects use different frameworks and different build tools, but they have something in common: they need to be tested to ensure the high quality of the code. Some older projects are running their tests with JUnit 4; newer ones have already started using JUnit 5. The engineers have decided to acquire in-depth knowledge of JUnit 5 and to transmit it to the projects that need to move from JUnit 4 to JUnit 5.

2.1 Core annotations

The `CalculatorTest` program from chapter 1, shown in the following listing, defines a test class with a single test method, `testAdd`.

Listing 2.1 `CalculatorTest` test case

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        double result = calculator.add(10, 50);
        assertEquals(60, result, 0);
    }
}
```

These are the most important concepts:

- A *test class* may be a top-level class, a static member class, or an inner class annotated as `@Nested` that contains one or more test methods. Test classes cannot be abstract and must have a single constructor. The constructor must have no arguments, or arguments that can be dynamically resolved at runtime through dependency injection. (We discuss the details of dependency injection in section 2.6.) A test class is allowed to be package-private as a minimum requirement for visibility. It is no longer required that test classes be public, as was the case up to JUnit 4.x. The Java compiler will supply a no-args constructor for `CalculatorTest` because we do not define any constructors for the class.

- A *test method* is an instance method that is annotated with `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, or `@TestTemplate`.
- A *life cycle method* is a method that is annotated with `@BeforeAll`, `@AfterAll`, `@BeforeEach`, or `@AfterEach`.

Test methods must not be abstract and must not return a value (the return type should be `void`).

The source files accompanying this book contain the code for all the examples. To use the imported classes, methods, and annotations needed for the test in listing 2.1, you'll need to declare their dependencies. Most projects use a build tool to manage them. (We have chosen to use Maven, as discussed in chapter 1. Chapter 10 covers running JUnit tests from Maven.)

You need to carry out only basic tasks in Maven: configure your project through the `pom.xml` file, execute the `mvn clean install` command, and understand the command's effects. The next listing shows the minimal JUnit 5 dependencies to be used in the `pom.xml` Maven configuration file.

Listing 2.2 pom.xml JUnit 5 dependencies

```

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId> ← 1
  <version>5.6.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId> ← 2
  <version>5.6.0</version>
  <scope>test</scope>
</dependency>

```

This shows that the minimal needed dependencies are `junit-jupiter-api` 1 and `junit-jupiter-engine` 2.

JUnit creates a new instance of the test class before invoking each `@Test` method to ensure the independence of test methods and prevent unintentional side effects in the test code. Also, it is a universally accepted fact that the tests must produce the same result independent of the order of their execution. Because each test method runs on a new test class instance, you cannot reuse instance variable values across test methods. One test instance is created for the execution of each test method, which is the default behavior in JUnit 5 and all previous versions.

If you annotate your test class with `@TestInstance(Lifecycle.PER_CLASS)`, JUnit 5 will execute all test methods on the same test instance. A new test instance will be created for each test class when using this annotation.

Listing 2.3 shows the use of the JUnit 5 life cycle methods in the `lifecycle.SUTTest` class. One of the projects at Tested Data Systems is testing a system that will start

up, receive regular and additional work, and close itself. The life cycle methods ensure that the system is initializing and then shutting down before and after each effective test. The test methods check whether the system receives regular and additional work.

Listing 2.3 Using JUnit 5 life cycle methods

```
class SUTTest {
    private static ResourceForAllTests resourceForAllTests;
    private SUT systemUnderTest;

    @BeforeAll
    static void setUpClass() {
        resourceForAllTests =
            new ResourceForAllTests("Our resource for all tests");
    }

    @AfterAll
    static void tearDownClass() {
        resourceForAllTests.close();
    }

    @BeforeEach
    void setUp() {
        systemUnderTest = new SUT("Our system under test");
    }

    @AfterEach
    void tearDown() {
        systemUnderTest.close();
    }

    @Test
    void testRegularWork() {
        boolean canReceiveRegularWork =
            systemUnderTest.canReceiveRegularWork();

        assertTrue(canReceiveRegularWork);
    }

    @Test
    void testAdditionalWork() {
        boolean canReceiveAdditionalWork =
            systemUnderTest.canReceiveAdditionalWork();

        assertFalse(canReceiveAdditionalWork);
    }
}
```

The diagram illustrates the execution flow of JUnit 5 life cycle methods. It starts with the `@BeforeAll` method (1), which is a static method. This is followed by the `@BeforeEach` method (3), which is executed before each test. Then, the `@Test` methods (5) are executed. After the tests, the `@AfterEach` method (4) is executed, and finally the `@AfterAll` method (2), which is a static method.

Following the life cycle of the test execution, you see that

- The method annotated with `@BeforeAll` (1) is executed once: before all tests. This method needs to be static unless the entire test class is annotated with `@TestInstance(Lifecycle.PER_CLASS)`.
- The method annotated with `@BeforeEach` (3) is executed before each test. In our case, it will be executed twice.

- The two methods annotated with `@Test` ⑤ are executed independently.
- The method annotated with `@AfterEach` ④ is executed after each test. In our case, it will be executed twice.
- The method annotated with `@AfterAll` ② is executed once: after all tests. This method needs to be static unless the entire test class is annotated with `@TestInstance(Lifecycle.PER_CLASS)`.
- To run this test class, you can execute the following from the command line:
`mvn -Dtest=SUTTest.java clean install`.

2.1.1 The `@DisplayName` annotation

The `@DisplayName` annotation can be used over classes and test methods. It helps the engineers at Tested Data Systems declare their own display name for an annotated test class or test method. Typically, this annotation is used for test reporting in IDEs and build tools. The string argument of the `@DisplayName` annotation may contain spaces, special characters, and even emojis.

The following listing demonstrates the use of the `@DisplayName` annotation through the class `displayname.DisplayNameTest`. The name that's displayed is usually a full phrase that provides significant information about the purpose of the test.

Listing 2.4 `@DisplayName` annotation

```

@DisplayName("Test class showing the @DisplayName annotation.")
class DisplayNameTest {
    private SUT systemUnderTest = new SUT();

    @Test
    @DisplayName("Our system under test says hello.")
    void testHello() {
        assertEquals("Hello", systemUnderTest.hello());
    }

    @Test
    @DisplayName("👋")
    void testTalking() {
        assertEquals("How are you?", systemUnderTest.talk());
    }

    @Test
    void testBye() {
        assertEquals("Bye", systemUnderTest.bye());
    }
}

```

This example does the following:

- Shows the display name applied to the entire class ①
- Applies a normal text display name ②
- Uses a display name that includes an emoji ③

A test that does not have an associated display name simply shows the method name. From IntelliJ, you can run a test by right-clicking on it and then executing the run command. The results of these tests in the IntelliJ IDE are shown in figure 2.1.

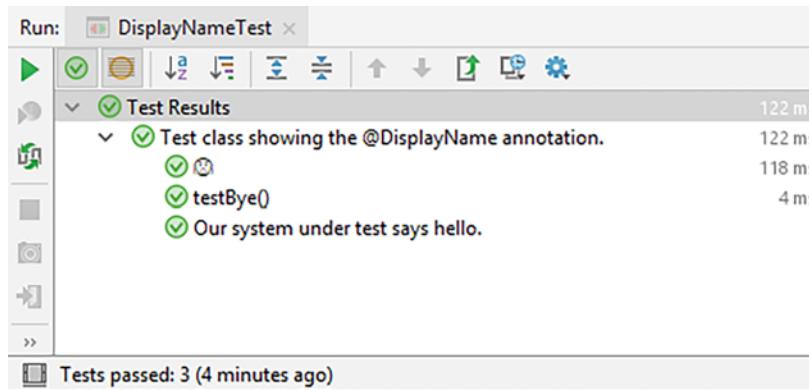


Figure 2.1 Running DisplayNameTest in IntelliJ

2.1.2 The `@Disabled` annotation

The `@Disabled` annotation can be used over classes and test methods. It signals that the annotated test class or test method is disabled and should not be executed. The programmers at Tested Data Systems use it to give their reasons for disabling a test so the rest of the team knows exactly why that was done. If this annotation is applied to a class, it disables all the methods of the test. Also, the disabled tests and the reasons for their being disabled are displayed differently on each programmer's console when the programmer runs them from the IDE.

The use of the annotation is demonstrated by the classes `disabled.DisabledClassTest` and `disabled.DisabledMethodsTest`. Listings 2.5 and 2.6 show the code for these classes.

Listing 2.5 `@Disabled` annotation used on a test class

```
@Disabled("Feature is still under construction.")
class DisabledClassTest {
    private SUT systemUnderTest= new SUT("Our system under test");

    @Test
    void testRegularWork() {
        boolean canReceiveRegularWork = systemUnderTest.
        canReceiveRegularWork();

        assertTrue(canReceiveRegularWork);
    }
}
```

```

    @Test
    void testAdditionalWork() {
        boolean canReceiveAdditionalWork =
            systemUnderTest.canReceiveAdditionalWork();

        assertFalse(canReceiveAdditionalWork);
    }

```

The entire testing class is disabled, and the reason is provided ①. This technique is recommended so that your colleagues (or even you) immediately understand why the test is not enabled.

Listing 2.6 @Disabled annotation used on methods

```

class DisabledMethodsTest {
    private SUT systemUnderTest= new SUT("Our system under test");

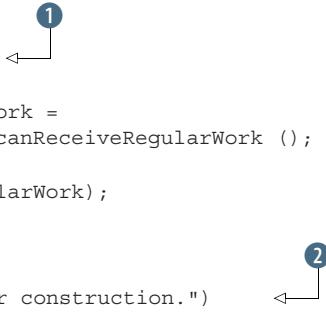
    @Test
    @Disabled
    void testRegularWork() {
        boolean canReceiveRegularWork =
            systemUnderTest.canReceiveRegularWork();

        assertTrue(canReceiveRegularWork);
    }

    @Test
    @Disabled("Feature still under construction.")
    void testAdditionalWork() {
        boolean canReceiveAdditionalWork =
            systemUnderTest.canReceiveAdditionalWork();

        assertFalse(canReceiveAdditionalWork);
    }
}

```



You see that

- The code provides two tests, both disabled.
- One of the tests is disabled without a reason given ①.
- The other test is disabled with a reason that other programmers will understand ②—the recommended approach.

2.2 Nested tests

An *inner class* is a class that is a member of another class. It can access any private instance variable of the outer class, as it is effectively part of that outer class. The typical use case is when two classes are tightly coupled, and it's logical to provide direct access from the inner class to all instance variables of the outer class. For example, we may test a flight that has two types of passengers trying to board. The behavior of the flight will be described in the outer test class, while the behavior of each type of

passenger will be described in its own nested class. Each passenger is able to interact with the flight. The nested tests will follow the business logic and lead to writing clearer code, as you will be able to follow the testing process more easily.

Following this tight-coupling idea, nested tests give the test writer more capabilities to express the relationships among several groups of tests. Inner classes may be package-private.

The Tested Data Systems company works with customers. Each customer has a gender, a first name, a last name, sometimes a middle name, and the date when they became a customer (if known). Some parameters may not be present, so the engineers are using the builder pattern to create and test a customer.

The following listing demonstrates the use of the `@Nested` annotation on the class `NestedTestsTest`. The customer being tested is John Michael Smith, and the date when he became a customer is known.

Listing 2.7 Nested tests

```
public class NestedTestsTest {
    private static final String FIRST_NAME = "John";
    private static final String LAST_NAME = "Smith"; ①

    @Nested
    class BuilderTest { ②
        private String MIDDLE_NAME = "Michael"; ③

        @Test
        void customerBuilder() throws ParseException { ④
            SimpleDateFormat simpleDateFormat =
                new SimpleDateFormat("MM-dd-yyyy");
            Date customerDate = simpleDateFormat.parse("04-21-2019");

            Customer customer = new Customer.Builder(
                Gender.MALE, FIRST_NAME, LAST_NAME)
                .withMiddleName(MIDDLE_NAME)
                .withBecomeCustomer(customerDate)
                .build(); ⑤

            assertAll(() -> {
                assertEquals(Gender.MALE, customer.getGender());
                assertEquals(FIRST_NAME, customer.getFirstName());
                assertEquals(LAST_NAME, customer.getLastName());
                assertEquals(MIDDLE_NAME, customer.getMiddleName());
                assertEquals(customerDate,
                    customer.getBecomeCustomer());
            });
        }
    }
}
```

The main test is `NestedTestsTest` ①, and it makes sense here that it is tightly coupled with the nested test `BuilderTest` ②. First, `NestedTestsTest` defines the first

name and last name of a customer that will be used for all nested tests ②. The nested test, `BuilderTest`, verifies the construction of a `Customer` object ④ with the help of the builder pattern ⑤. The equality of the fields is verified at the end of the `customerBuilder` test ⑥.

The source code file has another nested class, `CustomerHashCodeTest`, containing two more tests. You can follow along with it.

2.3 Tagged tests

If you are familiar with JUnit 4, *tagged tests* are replacements for JUnit 4 categories. You can use the `@Tag` annotation over classes and test methods. Later, you can use tags to filter test discovery and execution.

Listing 2.8 presents the `CustomerTest` tagged class, which tests the correct creation of Tested Data Systems customers. Listing 2.9 presents the `CustomersRepositoryTest` tagged class, which tests the existence and nonexistence of customers inside a repository. One use case may be to group your tests into a few categories based on the business logic and the things you are effectively testing. (Each test category has its own tag.) Then you may decide to run only some tests or alternate among categories, depending on your current needs.

Listing 2.8 CustomerTest tagged class

```
@Tag("individual")
public class CustomerTest {
    private String CUSTOMER_NAME = "John Smith";
```



①

```
    @Test
    void testCustomer() {
        Customer customer = new Customer(CUSTOMER_NAME);

        assertEquals("John Smith", customer.getName());
    }
}
```

The `@Tag` annotation is added to the entire `CustomerTest` class ①.

Listing 2.9 CustomersRepositoryTest tagged class

```
@Tag("repository")
public class CustomersRepositoryTest {
    private String CUSTOMER_NAME = "John Smith";
    private CustomersRepository repository = new CustomersRepository();

    @Test
    void testNonExistence() {
        boolean exists = repository.contains("John Smith");

        assertFalse(exists);
    }
}
```



①

```

    @Test
    void testCustomerPersistence() {
        repository.persist(new Customer(CUSTOMER_NAME));

        assertTrue(repository.contains("John Smith"));
    }
}

```

Similarly, the `@Tag` annotation is added to the entire `CustomersRepositoryTest` class ①. Here is the Maven configuration file for these tests.

Listing 2.10 pom.xml configuration file

```

<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
    <!--
    <configuration>
        <groups>individual</groups>
        <excludedGroups>repository</excludedGroups>
    </configuration>
    -->
</plugin>

```

To activate the tags, you have a few alternatives. One is to work at the level of the `pom.xml` configuration file. In this example, it's enough to uncomment the configuration node of the Surefire plugin ① and run `mvn clean install`.

Another alternative in the IntelliJ IDEA is to activate the tags by creating a configuration by choosing `Run > Run > Edit Configurations > Tags (JUnit 5)` as the test kind (figure 2.2). This is fine when you would like to quickly make some changes about which tests to run locally. However, it is strongly recommended that you make the changes at the level of the `pom.xml` file—otherwise, any automated build of the project will fail.

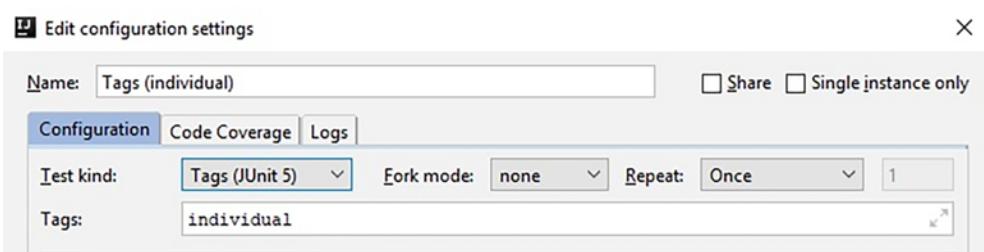


Figure 2.2 Configuring the tagged tests from the IntelliJ IDEA

2.4 Assertions

To perform test validation, you use the `assert` methods provided by the JUnit `Assertions` class. As you can see from the previous examples, we have statically imported these methods in our test class. Alternatively, you can import the JUnit `Assertions` class itself, depending on your taste for static imports. Table 2.1 lists some of the most popular `assert` methods.

Table 2.1 Sample JUnit 5 `assert` methods

assert method	What it is used for
<code>assertAll</code>	Overloaded method. It asserts that none of the supplied executables throw exceptions. An <code>Executable</code> is an object of type <code>org.junit.jupiter.api.function.Executable</code> .
<code>assertArrayEquals</code>	Overloaded method. It asserts that the expected array and the actual array are equal.
<code>assertEquals</code>	Overloaded method. It asserts that the expected values and the actual values are equal.
<code>assertX(..., String message)</code>	Assertion that delivers the supplied message to the test framework if the assertion fails.
<code>assertX(..., Supplier<String> messageSupplier)</code>	Assertion that delivers the supplied message to the test framework if the assertion fails. The failure message is retrieved lazily from the supplied <code>messageSupplier</code> .

JUnit 5 provides a lot of overloaded assertion methods. It includes many assertion methods from JUnit 4 and adds a few that can use Java 8 lambdas. All JUnit Jupiter assertions belong to the `org.junit.jupiter.api.Assertions` class and are static methods. The `assertThat()` method that works with Hamcrest matchers has been removed. The recommended approach in such a case is to use the Hamcrest `MatcherAssert.assertThat()` overloaded methods, which are more flexible and in the spirit of the Java 8 capabilities.

DEFINITION *Hamcrest* is a framework that assists with the writing of software tests in JUnit. It supports the creation of customized assertion matchers (*Hamcrest* is an anagram of *matchers*), letting us define match rules declaratively. Later in this chapter, we discuss the capabilities of Hamcrest.

As stated previously, one of the projects at our previously introduced Tested Data Systems company is testing a system that starts up, receives regular and additional work, and closes itself. After we run some operations, we need to verify more than a single condition. In this case, we'll also use the lambda expressions introduced in Java 8.

Lambda expressions treat functionality as a method argument and code as data. We can pass around a lambda expression as if it were an object and execute it on demand.

This section presents a few examples provided by the `assertions` package. Listing 2.11 shows some of the overloaded `assertAll` methods. The `heading` parameter allows us to recognize the group of assertions within the `assertAll()` methods. The failure message of the `assertAll()` method can provide detailed information about every particular assertion within a group. Also, we're using the `@DisplayName` annotation to provide easy-to-understand information about what the test is looking for. Our purpose is the verification of the same system under test (SUT) class that we introduced earlier.

After the `heading` parameter from the `assertAll` method, we provide the rest of the arguments as a collection of executables—a shorter, more convenient way to assert that supplied executables do not throw exceptions.

Listing 2.11 `assertAll` method

```
class AssertAllTest {
    @Test
    @DisplayName(
        "SUT should default to not being under current verification")
    void testSystemNotVerified() {
        SUT systemUnderTest = new SUT("Our system under test");

        assertAll("By default,
                  SUT is not under current verification",
                  () -> assertEquals("Our system under test",
                                      systemUnderTest.getSystemName()),
                  () -> assertFalse(systemUnderTest.isVerified()));
    }

    @Test
    @DisplayName("SUT should be under current verification")
    void testSystemUnderVerification() {
        SUT systemUnderTest = new SUT("Our system under test");

        systemUnderTest.verify();

        assertAll("SUT under current verification",
                  () -> assertEquals("Our system under test",
                                      systemUnderTest.getSystemName()),
                  () -> assertTrue(systemUnderTest.isVerified()));
    }
}
```

The diagram illustrates the execution flow of the `assertAll` method. It shows two main sections of code, each with its own `assertAll` call. Each `assertAll` call is annotated with a heading and a list of assertions. The assertions are grouped by a brace on the right side of the heading. The first section's heading is "By default, SUT is not under current verification". It contains two assertions: one for the system name and one for the verification status. The second section's heading is "SUT under current verification". It also contains two assertions for the system name and verification status. Arrows labeled 1 through 6 indicate the execution flow: 1 points to the first assertion in the first group; 2 points to the second assertion in the first group; 3 points to the brace of the first group; 4 points to the first assertion in the second group; 5 points to the second assertion in the second group; and 6 points to the brace of the second group.

The `assertAll` method will always check all the assertions that are provided to it, even if some of them fail—if any of the executables fail, the remaining ones will still be run. That is not true for the JUnit 4 approach: if you have a few `assert` methods, one under the other, and one of them fails, that failure will stop the execution of the others.

In the first test, the `assertAll` method receives as a parameter the message to be displayed if one of the supplied executables throws an exception ①. Then the method receives one executable to be verified with `assertEquals` ② and one to be verified with `assertFalse` ③. The assertion conditions are brief so that they can be read at a glance.

In the second test, the `assertAll` method receives as a parameter the message to be displayed if one of the supplied executables throws an exception ④. Then it receives one executable to be verified with `assertEquals` ⑤ and one to be verified with `assertTrue` ⑥. Just like in the first test, the assertion conditions are easy to read.

The next listing shows the use of some assertion methods with messages. Thanks to `Supplier<String>`, the instructions required to create a complex message aren't provided in the case of success. We can use lambda or method references to verify our SUT; they improve performance.

Listing 2.12 Some assertion methods with messages

```
...
@Test
@DisplayName("SUT should be under current verification")
void testSystemUnderVerification() {
    systemUnderTest.verify();
    assertTrue(systemUnderTest.isVerified(),
        () -> "System should be under verification");
}

@Test
@DisplayName("SUT should not be under current verification")
void testSystemNotUnderVerification() {
    assertFalse(systemUnderTest.isVerified(),
        () -> "System should not be under verification.");
}

@Test
@DisplayName("SUT should have no current job")
void testNoJob() {
    assertNull(systemUnderTest.getCurrentJob(),
        () -> "There should be no current job");
}
...
```

In this example:

- A condition is verified with the help of the `assertTrue` method ①. In case of failure, a message is lazily created ②.
- A condition is verified with the help of the `assertFalse` method ③. In case of failure, a message is lazily created ④.
- The existence of an object is verified with the help of the `assertNull` method ⑤. In case of failure, a message is lazily created ⑥.

The advantage of using lambda expressions as arguments for assertion methods is that all of them are lazily created, resulting in improved performance. If the condition at ① is fulfilled, meaning the test succeeded, the invocation of the lambda expression at ② does not take place, which would be impossible if the test were written in the old style.

There may be situations in which you expect a test to be executed within a given interval. In our example, it is natural for the user to expect that the system under test will run the given jobs quickly. JUnit 5 offers an elegant solution for this kind of use case.

The following listing shows the use of some `assertTimeout` and `assertTimeoutPreemptively` methods, which replace the JUnit 4 `Timeout` rule. *The methods* need to check whether the SUT is performant enough, meaning it is executing its jobs within a given timeout.

Listing 2.13 Some `assertTimeout` methods

```
class AssertTimeoutTest {
    private SUT systemUnderTest = new SUT("Our system under test");

    @Test
    @DisplayName("A job is executed within a timeout")
    void testTimeout() throws InterruptedException {
        systemUnderTest.addJob(new Job("Job 1"));
        assertTimeout(ofMillis(500), () -> systemUnderTest.run(200));
    }

    @Test
    @DisplayName("A job is executed preemptively within a timeout")
    void testTimeoutPreemptively() throws InterruptedException {
        systemUnderTest.addJob(new Job("Job 1"));
        assertTimeoutPreemptively(ofMillis(500),
            () -> systemUnderTest.run(200));
    }
}
```



`assertTimeout` waits until the executable finishes ①. The failure message looks something like this: `execution exceeded timeout of 500 ms by 193 ms`.

`assertTimeoutPreemptively` stops the executable when the time has expired ②. The failure message looks like this: `execution timed out after 500 ms`.

In some situations, you expect a test to be executed and to throw an exception, so you may force the rest to run under inappropriate conditions or to receive inappropriate input. In our example, it is natural that the SUT that tries to run without a job assigned to it will throw an exception. Again, JUnit 5 offers an elegant solution.

Listing 2.14 shows the use of some `assertThrows` methods, which replace the JUnit 4 `ExpectedException` rule and the `expected` attribute of the `@Test` annotation. All assertions can be made against the returned instance of `Throwable`. This makes the tests more readable, as we are verifying that the SUT is throwing exceptions: a current job is expected but not found.

Listing 2.14 Some `assertThrows` methods

```

class AssertThrowsTest {
    private SUT systemUnderTest = new SUT("Our system under test");

    @Test
    @DisplayName("An exception is expected")
    void testExpectedException() {
        assertThrows(NoJobException.class, systemUnderTest::run);    1
    }

    @Test
    @DisplayName("An exception is caught")
    void testCatchException() {
        Throwable throwable = assertThrows(NoJobException.class,
            () -> systemUnderTest.run(1000));    2
        assertEquals("No jobs on the execution list!",
            throwable.getMessage());
    }
}

```



In this example:

- We verify that the `systemUnderTest` object's call of the `run` method throws a `NoJobException` ①.
- We verify that a call to `systemUnderTest.run(1000)` throws a `NoJobException`, and we keep a reference to the thrown exception in the `throwable` variable ②.
- We check the message kept in the `throwable` exception variable ③.

2.5 Assumptions

Sometimes tests fail due to an external environment configuration or a date or time zone issue that we cannot control. We can prevent our tests from being executed under inappropriate conditions.

Assumptions verify the fulfillment of preconditions that are essential for running the tests. You can use assumptions when it does not make sense to continue the execution of a given test method. In the test report, these tests are marked as aborted.

JUnit 5 comes with a set of assumption methods suitable for use with Java 8 lambdas. The JUnit 5 assumptions are static methods belonging to the `org.junit.jupiter.api.Assumptions` class. The `message` parameter is in the last position.

JUnit 4 users should be aware that not all previously existing assumptions are provided in JUnit 5. There is no `assumeThat()` method, which we may regard as confirmation that matchers are no longer part of JUnit. The new `assumingThat()` method executes an assertion only if the assumption is fulfilled.

Suppose we have a test that needs to run only in the Windows OS and in the Java 8 version. These preconditions are turned into JUnit 5 assumptions. A test is executed only if the assumptions are true. The following listing shows the use of some assumption

methods and verifies our SUT only under the environmental conditions we imposed: the OS needs to be Windows, and the Java version needs to be 8. If these conditions (assumptions) are not fulfilled, the check is not made.

Listing 2.15 Some assumption methods

```
class AssumptionsTest {
    private static String EXPECTED_JAVA_VERSION = "1.8";
    private TestsEnvironment environment = new TestsEnvironment(
        new JavaSpecification(
            System.getProperty("java.vm.specification.version")),
        new OperationSystem(
            System.getProperty("os.name"),
            System.getProperty("os.arch")));
    }

    private SUT systemUnderTest = new SUT();

    @BeforeEach
    void setUp() {
        assumeTrue(environment.isWindows());
    }

    @Test
    void testNoJobToRun() {
        assumingThat(
            () -> environment.getJavaVersion()
                .equals(EXPECTED_JAVA_VERSION),
            () -> assertFalse(systemUnderTest.hasJobToRun()));
    }

    @Test
    void testJobToRun() {
        assumeTrue(environment.isAmd64Architecture());
        systemUnderTest.run(new Job());
        assertTrue(systemUnderTest.hasJobToRun());
    }
}
```

In this example:

- The `@BeforeEach` annotated method is executed before each test. The test will not run unless the assumption that the current environment is Windows is true ①.
- The first test checks that the current Java version is the expected one ②. Only if this assumption is true does it verify that no job is currently being run by the SUT ③.
- The second test checks the current environment architecture ④. Only if this architecture is the expected one does it run a new job on the SUT ⑤ and verify that the system has a job to run ⑥.

2.6 Dependency injection in JUnit 5

The previous JUnit versions did not permit test constructors or methods to have parameters. JUnit 5 allows test constructors and methods to have parameters, but they need to be resolved through dependency injection.

The `ParameterResolver` interface dynamically resolves parameters at runtime. A parameter of a constructor or method must be resolved at runtime by a registered `ParameterResolver`. You can inject as many parameters as you want, in any order.

JUnit 5 now has three built-in resolvers. You must explicitly enable other parameter resolvers by registering appropriate extensions via `@ExtendWith`. The parameter resolvers that are automatically registered are discussed in the following sections.

2.6.1 TestInfoParameterResolver

If a constructor or method parameter is of type `TestInfo`, `TestInfoParameterResolver` supplies an instance of this type. `TestInfo` is a class whose objects are used to inject information about the currently executed test or container into the `@Test`, `@BeforeEach`, `@AfterEach`, `@BeforeAll`, and `@AfterAll` methods. Then `TestInfo` gets information about the current test: the display name, test class or method, and associated tags. The display name can be the name of the test class or test method or a custom name provided with the help of `@DisplayName`. Here's how to use a `TestInfo` parameter as an argument of a constructor and annotated methods.

Listing 2.16 TestInfo parameters

```
class TestInfoTest {
    TestInfoTest(TestInfo testInfo) {
        assertEquals("TestInfoTest", testInfo.getDisplayName()); ← 1
    }
}

@BeforeEach
void setUp(TestInfo testInfo) {
    String displayName = testInfo.getDisplayName();
    assertTrue(displayName.equals("display name of the method") ||
               displayName.equals(
                   "testGetNameOfTheMethod(TestInfo)"));
}

@Test
void testGetNameOfTheMethod(TestInfo testInfo) {
    assertEquals("testGetNameOfTheMethod(TestInfo)",
                testInfo.getDisplayName()); ← 3
}

@Test
@DisplayName("display name of the method")
void testGetNameOfTheMethodWithDisplayNameAnnotation(TestInfo testInfo) {
    assertEquals("display name of the method",
                testInfo.getDisplayName()); ← 4
}
```

In this example:

- A `TestInfo` parameter is injected into the constructor and into three methods. The constructor verifies that the display name is `TestInfoTest`: its own name ①. This is the default behavior, which we can vary using `@DisplayName` annotations.
- The `@BeforeEach` annotated method is executed before each test. It has an injected `TestInfo` parameter, and it verifies that the displayed name is the expected one: the name of the method or the name specified by the `@DisplayName` annotation ②.
- Both tests have an injected `TestInfo` parameter. Each parameter verifies that the displayed name is the expected one: the name of the method in the first test ③ or the name specified by the `@DisplayName` annotation in the second test ④.
- The built-in `TestInfoParameterResolver` supplies an instance of `TestInfo` that corresponds to the current container or test as the value of the expected parameters of the constructor and of the methods.

2.6.2 `TestReporterParameterResolver`

If a constructor or method parameter is of type `TestReporter`, `TestReporterParameterResolver` supplies an instance of this type. `TestReporter` is a functional interface and therefore can be used as the assignment target for a lambda expression or method reference. `TestReporter` has a single `publishEntry` abstract method and several overloaded `publishEntry` default methods. Parameters of type `TestReporter` can be injected into methods of test classes annotated with `@BeforeEach`, `@AfterEach`, and `@Test`. `TestReporter` can also be used to provide additional information about the test that is run. Here's how to use a `TestReporter` parameter as an argument of `@Test` annotated methods.

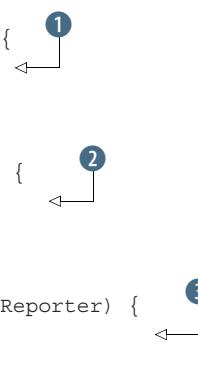
Listing 2.17 `TestReporter` parameters

```
class TestReporterTest {

    @Test
    void testReportSingleValue(TestReporter testReporter) {
        testReporter.publishEntry("Single value");
    } 1

    @Test
    void testReportKeyValuePair(TestReporter testReporter) {
        testReporter.publishEntry("Key", "Value");
    } 2

    @Test
    void testReportMultipleKeyValuePairs(TestReporter testReporter) {
        Map<String, String> values = new HashMap<>();
        values.put("user", "John");
        values.put("password", "secret");
    } 3 4
}
```



```

        testReporter.publishEntry(values);
    }
}

}

```



In this example, a `TestReporter` parameter is injected into three methods:

- In the first method, it is used to publish a single value entry ①.
- In the second method, it is used to publish a key-value pair ②.
- In the third method, we construct a map ③, populate it with two key-value pairs ④, and then use it to publish the constructed map ⑤.
- The built-in `TestReporterParameterResolver` supplies the instance of `TestReporter` needed to publish the entries.

The result of the execution of this test is shown in figure 2.3.

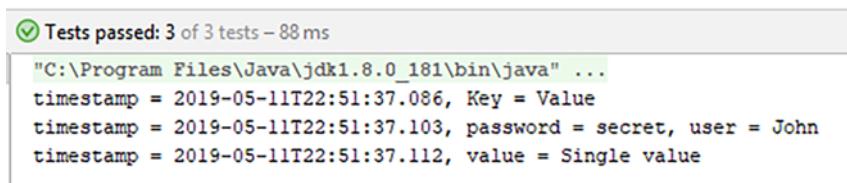


Figure 2.3 The result of executing `TestReporterTest`

2.6.3 `RepetitionInfoParameterResolver`

If a parameter in a method annotated with `@RepeatedTest`, `@BeforeEach`, or `@AfterEach` is of type `RepetitionInfo`, `RepetitionInfoParameterResolver` supplies an instance of this type. Then `RepetitionInfo` gets information about the current repetition and the total number of repetitions for a test annotated with `@RepeatedTest`. Repeated tests and examples are discussed in the following section.

2.7 `Repeated tests`

JUnit 5 allows us to repeat a test a specified number of times using the `@RepeatedTest` annotation, which has as a parameter the required number of repetitions. This feature can be particularly useful when conditions may change from one execution of a test to another. For example, some data that affects success may have changed between two executions of the same test, and an unexpected change in this data would be a bug that needs to be fixed.

A custom display name can be configured for each repetition using the `name` attribute of the `@RepeatedTest` annotation. The following placeholders are now supported:

- `{displayName}`—Display name of the method annotated with `@RepeatedTest`

- `{currentRepetition}`—Current repetition number
- `{totalRepetitions}`—Total number of repetitions

Listing 2.18 shows the use of repeated tests, display name placeholders, and `RepetitionInfo` parameters. The first repeated test verifies that the execution of the `add` method from the `Calculator` class is stable and always provides the same result. The second repeated test verifies that collections follow the appropriate behavior: a list receives a new element at each iteration, and a set does not get duplicate elements even if we try to insert such an element multiple times.

Listing 2.18 Repeated tests

```
public class RepeatedTestsTest {

    private static Set<Integer> integerSet = new HashSet<>();
    private static List<Integer> integerList = new ArrayList<>();

    @RepeatedTest(value = 5, name =
    "{displayName} - repetition {currentRepetition}/{totalRepetitions}")
    @DisplayName("Test add operation")
    void addNumber() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.add(1, 1),
                    "1 + 1 should equal 2");
    }

    @RepeatedTest(value = 5, name = "the list contains
    {currentRepetition} elements(s), the set contains 1 element")
    void testAddingToCollections(TestReporter testReporter,
                                 RepetitionInfo repetitionInfo) {
        integerSet.add(1);
        integerList.add(repetitionInfo.getCurrentRepetition());

        testReporter.publishEntry("Repetition number",
            String.valueOf(repetitionInfo.getCurrentRepetition()));
        assertEquals(1, integerSet.size());
        assertEquals(repetitionInfo.getCurrentRepetition(),
                    integerList.size());
    }
}
```

In this example:

- The first test is repeated five times. Each repetition shows the display name, the current repetition number, and the total number of repetitions ①.
- The second test is repeated five times. Each repetition shows the number of elements in the list (the current repetition number) and checks whether the set always has only one element ②.
- Each time the second test is repeated, the repetition number is displayed as it is injected into the `RepetitionInfo` parameter ③.

The results of executing these tests are shown in figures 2.4 and 2.5. Each invocation of a repeated test behaves like the execution of a regular @Test method with full support for life cycle callbacks and extensions. That is why the list and the set in the example are declared as static.

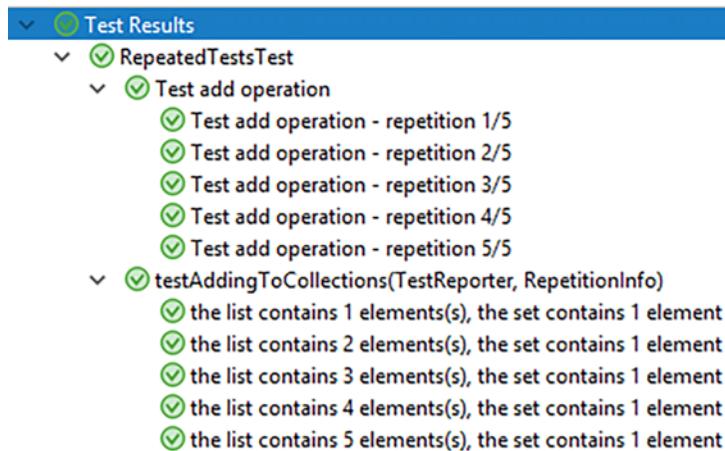


Figure 2.4 The names of the repeated tests at the time of execution

```
timestamp = 2019-05-12T17:26:34.783, Repetition number = 1
timestamp = 2019-05-12T17:26:34.798, Repetition number = 2
timestamp = 2019-05-12T17:26:34.803, Repetition number = 3
timestamp = 2019-05-12T17:26:34.813, Repetition number = 4
timestamp = 2019-05-12T17:26:34.817, Repetition number = 5
```

Figure 2.5 The messages shown on the console by the second repeated test

2.8 Parameterized tests

Parameterized tests allow a test to run multiple times with different arguments. The great benefit is that we can write a single test to be performed using arguments that check various input data. The methods are annotated with @ParameterizedTest. We must declare at least one source providing the arguments for each invocation. The arguments are then passed to the test method.

@ValueSource lets us specify a single array of literal values. At execution, this array provides a single argument for each invocation of the parameterized test. The following test checks the number of words in some phrases that are provided as parameters.

Listing 2.19 `@ValueSource` annotation

```
class ParameterizedWithValueSourceTest {
    private WordCounter wordCounter = new WordCounter();

    @ParameterizedTest
    @ValueSource(strings = { "Check three parameters",
                            "JUnit in Action" })
    void testWordsInSentence(String sentence) {
        assertEquals(3, wordCounter.countWords(sentence));
    }
}
```

1

2

In this example:

- We mark the test as being parameterized by using the corresponding annotation 1.
- We specify the values to be passed as an argument of the testing method 2. The testing method is executed twice: once for each argument provided by the `@ValueSource` annotation.

`@EnumSource` enables us to use enum instances. The annotation provides an optional `names` parameter that lets us specify which instances must be used or excluded. By default, all instances of an enum are used.

The following listing shows the use of the `@EnumSource` annotation to check the number of words in some phrases that are provided as enum instances.

Listing 2.20 `@EnumSource` annotation

```
class ParameterizedWithEnumSourceTest {
    private WordCounter wordCounter = new WordCounter();

    @ParameterizedTest
    @EnumSource(Sentences.class)
    void testWordsInSentence(Sentences sentence) {
        assertEquals(3, wordCounter.countWords(sentence.value()));
    }

    @ParameterizedTest
    @EnumSource(value=Sentences.class,
               names = { "JUNIT_IN_ACTION", "THREE_PARAMETERS" })
    void testSelectedWordsInSentence(Sentences sentence) {
        assertEquals(3, wordCounter.countWords(sentence.value()));
    }

    @ParameterizedTest #3
    @EnumSource(value=Sentences.class, mode = EXCLUDE, names =
               { "THREE_PARAMETERS" })
    void testExcludedWordsInSentence(Sentences sentence) {
        assertEquals(3, wordCounter.countWords(sentence.value()));
    }
}
```

1

2

3

```

enum Sentences {
    JUNIT_IN_ACTION("JUnit in Action"),
    SOME_PARAMETERS("Check some parameters"),
    THREE_PARAMETERS("Check three parameters");

    private final String sentence;

    Sentences(String sentence) {
        this.sentence = sentence;
    }

    public String value() {
        return sentence;
    }
}

```

This example has three tests, which work as follows:

- The first test is annotated as being parameterized. Then we specify the enum source as the entire `Sentences.class` ①. So this test is executed three times, once for each instance of the `Sentences` enum: `JUNIT_IN_ACTION`, `SOME_PARAMETERS`, and `THREE_PARAMETERS`.
- The second test is annotated as being parameterized. Then we specify the enum source as `Sentences.class`, but we restrict the instances to be passed to the test to `JUNIT_IN_ACTION` and `THREE_PARAMETERS` ②. So, this test will be executed twice.
- The third test is annotated as being parameterized. Then we specify the enum source as `Sentences.class`, but we exclude the `THREE_PARAMETERS` instance ③. So, this test is executed twice: for `JUNIT_IN_ACTION` and `SOME_PARAMETERS`.

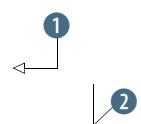
We can use `@CsvSource` to express argument lists as comma-separated values (CSV), such as String literals. The following test uses the `@CsvSource` annotation to check the number of words in some phrases that are provided as parameters—this time, in CSV format.

Listing 2.21 `@CsvSource` annotation

```

class ParameterizedWithCsvSourceTest {
    private WordCounter wordCounter = new WordCounter();
    @ParameterizedTest
    @CsvSource({"2, Unit testing", "3, JUnit in Action",
                "4, Write solid Java code"})
    void testWordsInSentence(int expected, String sentence) {
        assertEquals(expected, wordCounter.countWords(sentence));
    }
}

```



This example has one parameterized test, which functions as follows:

- The test is parameterized, as indicated by the appropriate annotation ①.
- The parameters passed to the test are from the parsed CSV strings listed in the `@CsvSource` annotation ②. So, this test is executed three times: once for each CSV line.
- Each CSV line is parsed. The first value is assigned to the `expected` parameter, and the second value is assigned to the `sentence` parameter.

`@CsvFileSource` allows us to use CSV files from the classpath. The parameterized test is executed once for each line of a CSV file. Listing 2.22 shows the use of the `@CsvFileSource` annotation, and listing 2.23 displays the contents of the `word_counter.csv` file on the classpath. The Maven build tool automatically adds the `src/test/resources` folder to the classpath. The test checks the number of words in some phrases that are provided as parameters—this time, in CSV format with a CSV file as resource input.

Listing 2.22 `@CsvFileSource` annotation

```
class ParameterizedWithCsvFileSourceTest {
    private WordCounter wordCounter = new WordCounter();

    @ParameterizedTest
    @CsvFileSource(resources = "/word_counter.csv")
    void testWordsInSentence(int expected, String sentence) {
        assertEquals(expected, wordCounter.countWords(sentence));
    }
}
```

1

Listing 2.23 Contents of the `word_counter.csv` file

```
2, Unit testing
3, JUnit in Action
4, Write solid Java code
```

This example has one parameterized test that receives as parameters the lines indicated in the `@CsvFileSource` annotation ①. So, this test is executed three times: once for each CSV file line. The CSV file line is parsed, the first value is assigned to the `expected` parameter, and then the second value is assigned to the `sentence` parameter.

2.9 Dynamic tests

JUnit 5 introduces a dynamic new programming model that can generate tests at runtime. We write a factory method, and at runtime, it creates a series of tests to be executed. Such a factory method must be annotated with `@TestFactory`.

A `@TestFactory` method is not a regular test but a factory that generates tests. A method annotated as `@TestFactory` must return one of the following:

- A `DynamicNode` (an abstract class; `DynamicContainer` and `DynamicTest` are the instantiable concrete classes)

- An array of `DynamicNode` objects
- A `Stream` of `DynamicNode` objects
- A `Collection` of `DynamicNode` objects
- An `Iterable` of `DynamicNode` objects
- An `Iterator` of `DynamicNode` objects

As with the requirements for `@Test`-annotated methods, `@TestFactory`-annotated methods are allowed to be package-private as a minimum requirement for visibility, but they cannot be private or static. They may also declare parameters to be resolved by a `ParameterResolver`.

A `DynamicTest` is a test case generated at runtime, composed of a display name and an `Executable`. Because the `Executable` is a Java 8 functional interface, the implementation of a dynamic test can be provided as a lambda expression or as a method reference.

A dynamic test has a different life cycle than a standard test annotated with `@Test`. The methods annotated with `@BeforeEach` and `@AfterEach` are executed for the `@TestFactory` method but not for each dynamic test; other than these methods, there are no life cycle callbacks for individual dynamic tests. The behavior of `@BeforeAll` and `@AfterAll` remains the same; they are executed before all tests and at the end of all tests.

Listing 2.24 demonstrates dynamic tests. We want to check a predicate against a numerical value. To do so, we use a single factory to generate three tests to be created at runtime: one for a negative value, one for zero, and one for a positive value. We write one method but get three tests dynamically.

Listing 2.24 Dynamic tests

```
class DynamicTestsTest {

    private PositiveNumberPredicate predicate = new
        PositiveNumberPredicate();

    @BeforeAll
    static void setUpClass() { ①
        System.out.println("@BeforeAll method");
    }

    @AfterAll
    static void tearDownClass() { ②
        System.out.println("@AfterAll method");
    }

    @BeforeEach
    void setUp() { ③
        System.out.println("@BeforeEach method");
    }

    @AfterEach
    void tearDown() { ④
    }
}
```

```

        System.out.println("@AfterEach method");
    }

    @TestFactory
    Iterator<DynamicTest> positiveNumberPredicateTestCases() {
        return asList(
            dynamicTest("negative number",
                () -> assertFalse(predicate.check(-1))),
            dynamicTest("zero",
                () -> assertFalse(predicate.check(0))),
            dynamicTest("positive number",
                () -> assertTrue(predicate.check(1)))
        ).iterator();
    }
}

```

The code is annotated with numbers 1 through 8. Number 1 is at the start of the first block of code. Number 2 is at the start of the last block of code. Number 3 is at the start of the first dynamic test. Number 4 is at the start of the last dynamic test. Number 5 is at the start of the `@TestFactory` annotated method. Number 6 is at the start of the first test case in the list. Number 7 is at the start of the second test case. Number 8 is at the start of the third test case.

In this example:

- The methods annotated with `@BeforeAll` 1 and `@AfterAll` 2 are executed once, as expected: at the beginning and at the end of the entire tests list, respectively.
- The methods annotated with `@BeforeEach` 3 and `@AfterEach` 4 are executed before and after the execution of the `@TestFactory`-annotated method, respectively 5.
- This factory method generates three test methods labeled "negative number" 6, "zero" 7, and "positive number" 8.
- The effective behavior of each test is given by the Executable provided as the second parameter of the `dynamicTest` method.

The result of executing these tests is shown in figure 2.6.

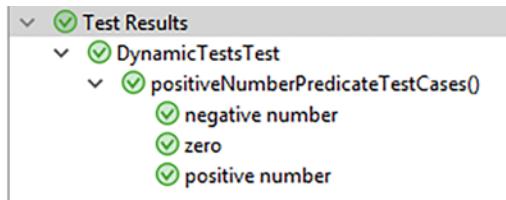


Figure 2.6 The result of executing dynamic tests

2.10 Using Hamcrest matchers

Statistics show that people easily become hooked on the unit-testing philosophy. When we get accustomed to writing tests and see how good it feels to be protected from possible mistakes, we wonder how it was possible to live without unit testing.

As we write more unit tests and assertions, we inevitably find that some assertions are big and hard to read. Our example company, Tested Data Systems, is working with customers whose data may be kept in lists. Engineers may populate a list with values like "Michael", "John", and "Edwin"; then they may search for customers like

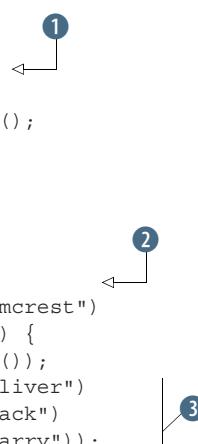
"Oliver", "Jack", and "Harry", as in the following listing. This test is intended to fail and show the description of the assertion failure.

Listing 2.25 Cumbersome JUnit assert method

```
[...]
public class HamcrestListTest {
    private List<String> values;

    @BeforeEach
    public void setUp () {
        values = new ArrayList<>();
        values.add("Michael");
        values.add("John");
        values.add("Edwin");
    }

    @Test
    @DisplayName("List without Hamcrest")
    public void testWithoutHamcrest() {
        assertEquals(3, values.size());
        assertTrue(values.contains("Oliver")
            || values.contains("Jack")
            || values.contains("Harry"));
    }
}
```



This example constructs a simple JUnit test like those described earlier in this chapter:

- A `@BeforeEach` fixture ① initializes some data for the test. A single test method is used ②.
- This test method makes a long, hard-to-read assertion ③. (Maybe the assertion itself is not too hard to read, but what it does definitely is not obvious at first glance.)
- The goal is to simplify the assertion made in the test method.

To solve this problem, Tested Data Systems uses a library of matchers for building test expressions: Hamcrest. Hamcrest (<https://code.google.com/archive/p/hamcrest>) is a library that contains a lot of helpful `Matcher` objects (also known as *constraints* or *predicates*) ported in several languages, such as Java, C++, Objective-C, Python, and PHP.

The Hamcrest library

Hamcrest is not a testing framework itself, but it helps us declaratively specify simple matching rules. These matching rules can be used in many situations, but they are particularly helpful for unit testing.

The next listing provides the same test method as listing 2.25, this time written with the Hamcrest library.

Listing 2.26 Using the Hamcrest library

```
[...]
import static org.hamcrest.CoreMatchers.anyOf;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;
[...]

@DisplayName("List with Hamcrest")
public void testListWithHamcrest() {
    assertThat(values, hasSize(3));
    assertThat(values, hasItem(anyOf(equalTo("Oliver"),
        equalTo("Jack"), equalTo("Harry"))));
}
[...]
```

This example adds a test method that imports the needed matchers and the `assertThat` method ① and then constructs a test method. The test method uses one of the most powerful features of matchers: they can nest ②. What Hamcrest gives us, and standard assertions do not, is a human-readable description of an assertion failure. Using assertion code with or without Hamcrest matchers is a personal preference.

The examples in the previous two listings construct a `List` with the customers "Michael", "John", and "Edwin" as elements. After that, the code asserts the presence of "Oliver", "Jack", or "Harry", so the tests will fail on purpose. The result from the execution without Hamcrest is shown in figure 2.7, and the result from the execution with Hamcrest is shown in figure 2.8. As the figures show, the test that uses Hamcrest provides more details.

```
org.opentest4j.AssertionFailedError:
Expected :<true>
Actual   :<false>
<Click to see difference>

<4 internal calls>
  at com.manning.junitbook.ch02.core.hamcrest.HamcrestListTest.testListWithoutHamcrest(HamcrestListTest.java:53) <19 internal calls>
  at java.util.ArrayList.forEach(ArrayList.java:1257) <9 internal calls>
  at java.util.ArrayList.forEach(ArrayList.java:1257) <21 internal calls>
```

Figure 2.7 The result of the test execution without Hamcrest

```
java.lang.AssertionError:
Expected: a collection containing ("Oliver" or "Jack" or "Harry")
but: mismatches were: [was "John", was "Michael", was "Edwin"]

  at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:18)
  at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:6)
  at com.manning.junitbook.ch02.core.hamcrest.HamcrestListTest.testListWithHamcrest(HamcrestListTest.java:60) <19 internal calls>
  at java.util.ArrayList.forEach(ArrayList.java:1257) <9 internal calls>
  at java.util.ArrayList.forEach(ArrayList.java:1257) <21 internal calls>
```

Figure 2.8 The result of the test execution with Hamcrest

To use Hamcrest in our projects, we need to add the required dependency to the pom.xml file.

Listing 2.27 pom.xml Hamcrest dependency

```
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-library</artifactId>
    <version>2.1</version>
    <scope>test</scope>
</dependency>
```

To use Hamcrest in JUnit 4, you have to use the `assertThat` method from the `org.junit.Assert` class. But as explained earlier in this chapter, JUnit 5 removes the `assertThat()` method. The user guide justifies the decision this way:

[...] org.junit.jupiter.api.Assertions class does not provide an assertThat() method like the one found in JUnit 4's org.junit.Assert class, which accepts a Hamcrest Matcher. Instead, developers are encouraged to use the built-in support for matchers provided by third-party assertion libraries.

This text means that if we want to use Hamcrest matchers, we have to use the `assertThat()` methods of the `org.hamcrest.MatcherAssert` class. As the previous examples illustrated, the overloaded methods take two or three method parameters:

- An error message shown when the assertion fails (optional)
- The actual value or object
- A `Matcher` object for the expected value

To create the `Matcher` object, we need to use one of the static factory methods provided by the `org.hamcrest.Matchers` class, as shown in table 2.2.

Table 2.2 Sample of common Hamcrest static factory methods

Factory method	Logical
<code>anything</code>	Matches absolutely anything; useful when we want to make the <code>assert</code> statement more readable
<code>is</code>	Used only to improve the readability of statements
<code>allOf</code>	Checks whether all contained matchers match (like the <code>&&</code> operator)
<code>anyOf</code>	Checks whether any of the contained matchers match (like the <code> </code> operator)
<code>not</code>	Inverts the meaning of the contained matchers (like the <code>!</code> operator in Java)
<code>instanceOf</code>	Check whether objects are instances of one another

Table 2.2 Sample of common Hamcrest static factory methods (continued)

Factory method	Logical
sameInstance	Tests object identity
nullValue, notNullValue	Tests for null or non-null values
hasProperty	Tests whether a Java Bean has a certain property
hasEntry, hasKey, hasValue	Tests whether a given Map has a given entry, key, or value
hasItem, hasItems	Tests a given collection for the presence of an item or items
closeTo, greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo	Tests whether given numbers are close to, greater than, greater than or equal, less than, or less than or equal to a given value
equalToIgnoringCase	Tests whether a given string equals another one, ignoring the case
equalToIgnoringWhiteSpace	Tests whether a given string equals another one, ignoring white space
containsString, endsWith, startsWith	Tests whether a given string contains, starts with, or ends with a certain string

All of these methods are pretty straightforward to read and use. Also, remember that we can compose them into one another.

For each service provided to customers, Tested Data Systems charges a particular price. The following code tests the properties of a customer and some service prices using a few Hamcrest methods.

Listing 2.28 A few Hamcrest static factory methods

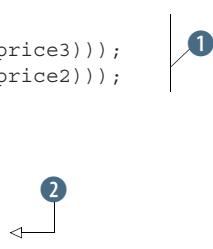
```
public class HamcrestMatchersTest {

    private static String FIRST_NAME = "John";
    private static String LAST_NAME = "Smith";
    private static Customer customer = new Customer(FIRST_NAME, LAST_NAME);

    @Test
    @DisplayName("Hamcrest is, anyOf, allOf")
    public void testHamcrestIs() {
        int price1 = 1, price2 = 1, price3 = 2;

        assertThat(1, is(price1));
        assertThat(1, anyOf(is(price2), is(price3)));
        assertThat(1, allOf(is(price1), is(price2)));
    }

    @Test
    @DisplayName("Null expected")
    void testNull() {
        assertThat(null, nullValue());
    }
}
```



```

@Test
@DisplayName("Object expected")
void testNotNull() {
    assertThat(customer, notNullValue());    ← 3
}

@Test
@DisplayName("Check correct customer properties")
void checkCorrectCustomerProperties() {
    assertThat(customer, allOf(
        hasProperty("firstName", is(FIRST_NAME)),
        hasProperty("lastName", is(LAST_NAME))
    ));    ← 4
}
}

```

This example shows

- The use of the `is`, `anyOf`, and `allOf` methods ①
- The use of the `nullValue` method ②
- The use of the `notNullValue` method ③
- The use of the `assertThat` method ①, ②, ③, and ④, as described in table 2.2

We have also constructed a `Customer` object and check its properties with the help of the `hasProperty` method ④.

Last but not least, Hamcrest is extremely extensible. Writing matchers that check a certain condition is easy: we implement the `Matcher` interface and an appropriately named factory method.

Chapter 3 analyzes the architectures of JUnit 4 and JUnit 5 and explains the move to the new architecture.

Summary

This chapter has covered the following:

- The core JUnit 5 classes related to assertions and assumptions.
- Using JUnit 5 methods and annotations: the methods from the assertions and assumptions classes, and annotations like `@Test`, `@DisplayName`, and `@Disabled`
- The life cycle of a JUnit 5 test, and controlling it through the `@BeforeEach`, `@AfterEach`, `@BeforeAll`, and `@AfterAll` annotations
- Applying JUnit 5 capabilities to create nested tests and tagged tests (annotation: `@Tag`)
- Implementing dependency injection with the help of test constructors and methods with parameters
- Applying dependency injection by using different parameter resolvers (`TestInfoParameterResolver`, `TestReporterParameterResolver`)

- Implementing repeated tests (annotation: `@RepeatedTest`) as another application of dependency injection
- Parameterized tests, a very flexible tool for testing that consumes different data sets and dynamic tests created at runtime (annotations: `@ParameterizedTest`, `@TestFactory`)
- Using Hamcrest matchers to simplify assertions

3

JUnit architecture

This chapter covers

- Demonstrating the concept and importance of software architecture
- Comparing the JUnit 4 and JUnit 5 architectures

Architecture is the stuff that's hard to change later. And there should be as little of that stuff as possible.

—Martin Fowler

So far, we have broadly surveyed JUnit and its capabilities (chapter 1). We've also looked at JUnit core classes and methods and how they interact with each other, as well as how to use the many features of JUnit 5 (chapter 2).

This chapter looks at the architecture of the two most recent JUnit versions. It discusses the architecture of JUnit 4 to show where JUnit 5 started, where the big changes are between versions, and which shortcomings had to be addressed.

3.1 The concept and importance of software architecture

Software architecture refers to the fundamental structures of a software system. Such a system must be created in an organized manner. A software system structure comprises software elements, relationships among those elements, and properties of both elements and relationships.

Software architecture is like the architecture of a building. The architecture of a software system characterizes the foundation on which everything else sits, represented by the bottom boxes in figure 3.1. Foundational software architectural elements are as hard to move around and replace as are the foundational parts of physical architecture, because you have to move all the things on top to reach them.

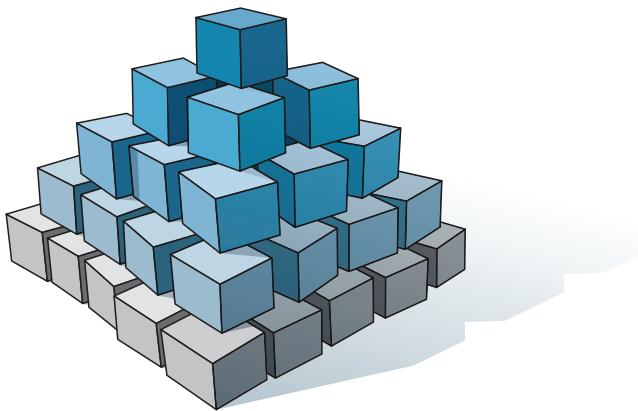


Figure 3.1 The architecture represents the foundation of the system. At the bottom level, the architecture is hard to move around and replace. The intermediary boxes represent the design; the top ones represent the idioms.

The corollary to Fowler's definition of architecture is that you should construct the architectural elements so that they're easier to replace. The architecture of JUnit 5 emerged from the shortcomings of JUnit 4. To understand the significant impact of the architecture on the entire system, read the stories in the next two sections.

3.1.1 **Story 1: The telephone directories**

Once upon a time, two companies published telephone directories. The companies' books were identical in shape, size, and cost.

Neither company could gain a significant advantage. Clerks were buying both products for \$1; they couldn't tell which book was better because the books contained similar information. Finally, company A hired a troubleshooter. The troubleshooter thought for a while and found a solution: "Let's make the format of our own book smaller than our competitor's book while keeping the same information."

When books are the same size, people often put the books near each other on their office desks. But when one book is large and flat, and the second is small but plump, people tend to put the small book on top of the large book (figure 3.2). At the end of the month, company A's customers realized that they had used only the top (small) directory; they never opened the large directory. So why spend a dollar on the larger one? This is one of the architectural changes from JUnit 4 to JUnit 5: smaller works better.

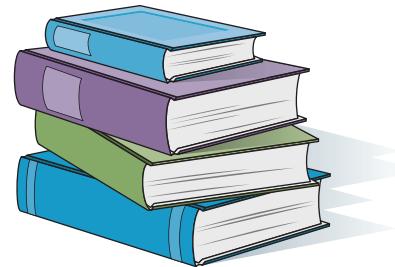


Figure 3.2 Changing the size of books is an architectural change that has a big impact. A small item is used and moved more easily and frequently than a big one.

3.1.2 Story 2: The sneakers manufacturer

A company began manufacturing sneakers in low-cost locations, expecting the net cost to be low. But losses from sneaker thefts were unexpectedly huge. The company tried to get more guards, but hiring guards increased the final price. The company needed a way to decrease the number of stolen sneakers without additional costs.

The company's analysts arrived at this solution: producing the left and right shoes in different locations (figure 3.3). Consequently, sneaker thefts decreased dramatically.

This scenario is another architectural change from JUnit 4 to JUnit 5: modularity improves the work. When we need some functionality, we can address the module that implements it. We depend on and load only a specific module instead of the entire testing framework, which saves time and memory.



Figure 3.3 Separating the production locations for left and right sneakers represents a major architectural change.

3.2 The JUnit 4 architecture

This book focuses on JUnit 5, but it also discusses JUnit 4 for some important reasons. One reason is that a lot of legacy code uses JUnit 4. Also, migration from JUnit 4 to JUnit 5 does not happen instantly (if ever), and projects may work with a hybrid

JUnit 4/JUnit 5 approach. Moreover, JUnit 5 was designed to work with old JUnit 4 code in existing legacy projects through JUnit Vintage. (Chapter 4 clarifies the best times to postpone or cancel migration from JUnit 4.)

The shortcomings of JUnit 4 helped JUnit 5 emerge, however. This section emphasizes some JUnit 4 characteristics that clearly show the need for the JUnit 5 approach: modularity, runners, and rules.

3.2.1 *JUnit 4 modularity*

JUnit 4, released in 2006, has a simple, monolithic architecture. All of its functionality is concentrated inside a single jar file (figure 3.4). If a programmer wants to use JUnit 4 in a project, all they need to do is add that jar file on the classpath.

3.2.2 *JUnit 4 runners*

A JUnit 4 *runner* is a class that extends the JUnit 4 abstract Runner class. A JUnit 4 runner is responsible for running JUnit tests. JUnit 4 remains a single jar file, but it's generally necessary to extend the functionality of this file. In other words, developers have the opportunity to add custom features to the functionality, such as executing additional things before and after running a test.

Runners may extend a test behavior by using reflection. Reflection breaks encapsulation, of course, but this technique was the only way to provide extensibility in JUnit 4 and earlier versions—one good reason for the JUnit 5 approach. You may need to keep existing JUnit 4 runners in your code for some time. (Extensions are the JUnit 5 equivalent of JUnit 4 runners and are introduced in chapter 4.)

In practice, you can work with existing runners, such as the runners for the Spring framework or for mocking objects with Mockito (chapter 8). We consider it very useful to demonstrate the creation and usage of custom runners, as they reveal the principles of runners in general. We can extend the JUnit 4 abstract Runner class, overriding its methods and working with reflection. This approach breaks encapsulation but was the only way to add custom functionality to JUnit 4. Revealing the shortcomings of working with custom runners in JUnit 4 opens the gate to understanding the capabilities and advantages of JUnit 5 extensions.

To demonstrate the use of JUnit 4 runners, we come back to our Calculator class.

Listing 3.1 Test Calculator class

```
public class Calculator {
    public double add(double number1, double number2) {
        return number1 + number2;
    }
}
```



Figure 3.4 The monolithic architecture of JUnit 4: a single jar file

We would like to enrich the behavior of tests that use this class by introducing an additional action before executing the test suite. We'll create a custom runner and use it as an argument of the `@RunWith` annotation to add custom features to the original JUnit functionality. The following listing demonstrates how to build a `CustomTestRunner`.

Listing 3.2 CustomTestRunner class

```
public class CustomTestRunner extends Runner {

    private Class<?> testedClass;

    public CustomTestRunner(Class<?> testedClass) {
        this.testedClass = testedClass;
    }

    @Override
    public Description getDescription() {
        return Description
            .createTestDescription(testedClass,
                this.getClass().getSimpleName() + " description");
    }

    @Override
    public void run(RunNotifier notifier) {
        System.out.println("Running tests with " +
            this.getClass().getSimpleName() + ": " + testedClass);
        try {
            Object testObject = testedClass.newInstance();
            for (Method method : testedClass.getMethods()) {
                if (method.isAnnotationPresent(Test.class)) { ④
                    notifier.fireTestStarted(Description
                        .createTestDescription(testedClass,
                            method.getName())); ⑤
                    method.invoke(testObject);
                    notifier.fireTestFinished(Description
                        .createTestDescription(testedClass,
                            method.getName())); ⑦
                }
            }
        } catch (InstantiationException | IllegalAccessException |
            InvocationTargetException e) {
            throw new RuntimeException(e);
        }
    }
}
```

The diagram illustrates the structure of the `CustomTestRunner` class with the following annotations:

- 1**: A callout points to the constructor `CustomTestRunner(Class<?> testedClass)`.
- 2**: A callout points to the `getDescription()` method, which returns a `Description` object.
- 3**: A callout points to the `run(RunNotifier notifier)` method.
- 4**: A callout points to the `if (method.isAnnotationPresent(Test.class))` condition within the `run` method's loop.
- 5**: A callout points to the first `createTestDescription` call within the `run` method's loop.
- 6**: A callout points to the `method.invoke(testObject);` line within the `run` method's loop.
- 7**: A callout points to the second `createTestDescription` call within the `run` method's loop.

In this listing:

- We keep a reference to the tested class, which is initialized inside the constructor **1**.
- We override the abstract `getDescription` method inherited from the abstract `Runner` class. This method contains information that is later exported and may be used by various tools **2**.

- We override the abstract `run` method inherited from the abstract `Runner` class. Inside it, we create an instance of the tested class ③.
- We browse all public methods from the tested class and filter the `@Test`-annotated ones ④.
- We invoke `fireTestStarted` to tell listeners that an atomic test is about to start ⑤.
- We reflectively invoke the original `@Test`-annotated method ⑥.
- We invoke `fireTestFinished` to tell listeners that an atomic test has finished ⑦.

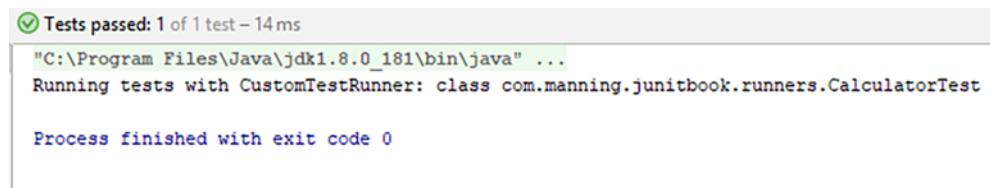
We'll use the `CustomTestRunner` class as an argument of the `@RunWith` annotation, to be applied to the `CalculatorTest` class.

Listing 3.3 `CalculatorTest` class

```
@RunWith(CustomTestRunner.class)
public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        double result = calculator.add(10, 50);
        assertEquals(60, result, 0);
    }
}
```

Figure 3.5 shows the result of executing this test. This example demonstrates how to add custom functionality to JUnit 4; using the recognized terminology, we are extending JUnit 4's functionality.



```
Tests passed: 1 of 1 test - 14 ms
"C:\Program Files\Java\jdk1.8.0_181\bin\java" ...
Running tests with CustomTestRunner: class com.manning.junitbook.runners.CalculatorTest
Process finished with exit code 0
```

Figure 3.5 The result of executing `CalculatorTest` with a custom runner

3.2.3 *JUnit 4 rules*

A JUnit 4 *rule* is a component that intercepts test method calls; it allows you to do something before a test method is run and something else after a test method has run. Rules are specific to JUnit 4.

To add behavior to the executed tests, you must use the `@Rule` annotation on `TestRule` fields. This technique increases the flexibility of tests by creating objects that can be used and configured in the test methods.

As with runners, you may need to keep existing rules in your code for some time, as migration to JUnit 5 mechanisms is not straightforward. The equivalent approach in JUnit 5 forces programmers to implement extensions (covered in part 4).

We would like to add two more methods to the `Calculator` class, as shown next.

Listing 3.4 Calculator class with additional functionality

```
public class Calculator {
    ...
    public double sqrt(double x) {
        if (x < 0) {
            throw new
                IllegalArgumentException("Cannot extract the square
                    root of a negative value");
        }
        return Math.sqrt(x);
    }

    public double divide(double x, double y) {
        if (y == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }
        return x/y;
    }
}
```

The new logic of the `Calculator` class does the following:

- Declares a method to calculate the square root of a number ①. If the number is negative, an exception containing a message is created and thrown ②.
- Declares a method to divide two numbers ③. If the second number is zero, an exception containing a message is created and thrown ④.

We would like to test the newly introduced methods and see whether the appropriate exceptions are thrown for particular inputs. The following listing specifies which exception message is expected while executing the test code, using the new functionality of the `Calculator` class.

Listing 3.5 RuleExceptionTester class

```
public class RuleExceptionTester {
    @Rule
    public ExpectedException expectedException =
        ExpectedException.none();

    private Calculator calculator = new Calculator();
    ...
    @Test
    public void expectIllegalArgumentException() {
        expectedException.expect(IllegalArgumentException.class);
    }
}
```

```

    expectedException.expectMessage("Cannot extract the square root
                                    of a negative value");
    calculator.sqrt(-1);
}

@Test
public void expectArithmaticException() {
    expectedException.expect(ArithmaticException.class);
    expectedException.expectMessage("Cannot divide by zero");
    calculator.divide(1, 0);
}
}

```

In this example:

- We declare an `ExpectedException` field annotated with `@Rule`. The `@Rule` annotation must be applied to a public nonstatic field or a public nonstatic method ①. The `ExpectedException.none()` factory method simply creates an unconfigured `ExpectedException`.
- We initialize an instance of the `Calculator` class whose functionality we are testing ②.
- `ExpectedException` is configured to keep the type of exception ③ and message ④ before it is thrown by invoking the `sqrt` method ⑤.
- `ExpectedException` is configured to keep the type of exception ⑥ and message ⑦ before it is thrown by invoking the `divide` method ⑧.

In some situations, you need to work with temporary resources, such as creating files and folders to store only test-specific information. The `TemporaryFolder` rule allows you to create files and folders that should be deleted when the test method finishes (whether the test passes or fails).

The next listing shows a `TemporaryFolder` field annotated with `@Rule`. We are testing the existence of these temporary resources.

Listing 3.6 RuleTester class

```

public class RuleTester {
    @Rule
    public TemporaryFolder folder = new TemporaryFolder();
    private static File createdFolder;
    private static File createdFile;

    @Test
    public void testTemporaryFolder() throws IOException {
        createdFolder = folder.newFolder("createdFolder");
        createdFile = folder.newFile("createdFile.txt");
        assertTrue(createdFolder.exists());
        assertTrue(createdFile.exists());
    }

    @AfterClass
    public static void cleanUpAfterAllTestsRan() {
}

```

```

    assertFalse(createdFolder.exists());
    assertFalse(createdFile.exists());
}
}

```

In this example:

- We declare a `TemporaryFolder` field annotated with `@Rule` and initialize it. The `@Rule` annotation must be applied to a public field or a public method **1**.
- We declare the static fields `createdFolder` and `createdFile` **2**.
- We use the `TemporaryFolder` field to create a folder and a file **3**, which are located in the Temp folder of our user profile in the operating system.
- We check the existence of the temporary folder and of the temporary file **4**.
- At the end of executing the tests, we check that the temporary resources do not exist any longer **5**.

We have demonstrated that two existing JUnit 4 rules—`ExpectedException` and `TemporaryFolder`—are working.

Now we would like to write a custom rule, which is useful for providing our own behavior before and after a test is run. We might like to start a process before executing a test and stop it after that, or connect to a database before executing a test and tear it down afterward.

To write a rule, we'll have to create a class that implements the `TestRule` interface (listing 3.7). Consequently, we'll override the `apply(Statement, Description)` method, which must return an instance of `Statement` (listing 3.8). Such an object represents the tests within the JUnit runtime, and `Statement#evaluate()` runs them. The `Description` object describes the individual test (listing 3.9); we can use this object to read information about the test through reflection.

Listing 3.7 CustomRule class

```

public class CustomRule implements TestRule {
    private Statement base;
    private Description description;
}

@Override
public Statement apply(Statement base, Description description) {
    this.base = base;
    this.description = description;
    return new CustomStatement(base, description);
}

```

In this example:

- We declare a `CustomRule` class that implements the `TestRule` interface **1**.
- We keep references to a `Statement` field and a `Description` field **2**, and we use them in the `apply` method that returns a `CustomStatement` **3**.

Listing 3.8 CustomStatement class

```

public class CustomStatement extends Statement {
    private Statement base;
    private Description description;

    public CustomStatement(Statement base, Description description) {
        this.base = base;
        this.description = description;
    }

    @Override
    public void evaluate() throws Throwable {
        System.out.println(this.getClass().getSimpleName() + " " +
                           description.getMethodName() + " has started");
        try {
            base.evaluate();
        } finally {
            System.out.println(this.getClass().getSimpleName() + " " +
                           description.getMethodName() + " has finished");
        }
    }
}

```

In this example:

- We declare our CustomStatement class that extends the Statement class ①.
- We keep references to a Statement field and a Description field ②, and we use them as arguments of the constructor ③.
- We override the inherited evaluate method and call base.evaluate() inside it ④.

Listing 3.9 CustomRuleTester class

```

public class CustomRuleTester {

    @Rule
    public CustomRule myRule = new CustomRule(); ①

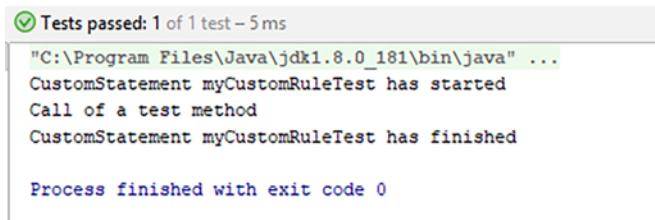
    @Test
    public void myCustomRuleTest() {
        System.out.println("Call of a test method");
    }
}

```

In this example, we use the previously defined CustomRule as follows:

- We declare a public CustomRule field and annotate it with @Rule ①.
- We create the myCustomRuleTest method and annotate it with @Test ②.

Figure 3.6 shows the result of executing this test. The effective execution of the test is surrounded by additional messages provided to the evaluate method of the CustomStatement class.



```

Tests passed: 1 of 1 test - 5 ms
"C:\Program Files\Java\jdk1.8.0_181\bin\java" ...
CustomStatement myCustomRuleTest has started
Call of a test method
CustomStatement myCustomRuleTest has finished

Process finished with exit code 0

```

Figure 3.6
The result of executing
CustomRuleTester

As an alternative, we provide the `CustomRuleTester2` class, which keeps the `CustomRule` field private and exposes it through a public getter annotated with `@Rule`. This annotation works only on public and nonstatic fields and methods.

Listing 3.10 The CustomRuleTester2 class

```

public class CustomRuleTester2 {

    private CustomRule myRule = new CustomRule();

    @Rule
    public CustomRule getMyRule() {
        return myRule;
    }

    @Test
    public void myCustomRuleTest() {
        System.out.println("Call of a test method");
    }
}

```

The result of executing this test is the same (figure 3.6). The effective execution of the test is surrounded by the additional messages provided to the `evaluate` method of the `CustomStatement` class.

Writing your own rules is very useful when you need to add custom behavior to tests. Typical use cases include allocating a resource before a test is run and freeing it afterward, starting a process before executing a test and stopping it after that, and connecting to a database before executing a test and tearing it down after that.

By using runners and rules, you can extend the monolithic architecture of JUnit 4. You may still encounter a lot of JUnit 4 code that uses runners and rules; plus, migrating to the equivalent JUnit 5 mechanisms (extensions) is not straightforward. So you may keep runners and rules in your code for a while, even if you move your work to JUnit 5.

One more thing to look for in these examples of JUnit 4 runners and rules is the Maven `pom.xml` configuration file.

Listing 3.11 pom.xml configuration file

```

<dependencies>
    <dependency>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>

```

```

<version>5.6.0</version>
<scope>provided</scope>
</dependency>
</dependencies>

```

The only required dependency is `junit-vintage-engine`, which belongs to JUnit 5. JUnit Vintage (a component of the JUnit 5 architecture, discussed in section 3.3.4) ensures backward compatibility with previous versions of JUnit. Working with JUnit Vintage, Maven transitively accesses the dependency to JUnit 4. As it may be tedious and time consuming to move the existing tests to JUnit 5, introducing this dependency can ensure the coexistence of JUnit 4 and JUnit 5 tests within the same project.

3.2.4 **Shortcomings of the JUnit 4 architecture**

Despite its apparent simplicity, the JUnit 4 architecture generated a series of problems that grew as time passed. JUnit was used not only by programmers, but also by many software programs, such as IDEs (Eclipse, NetBeans, and IntelliJ) and build tools (Ant and Maven). JUnit 4 is monolithic and was not designed to interact with these kinds of tools, but everyone wanted to work with such a popular, simple, useful framework.

The API provided by JUnit 4 was not flexible enough. Consequently, the IDEs and tools that were using JUnit 4 were tightly coupled to the unit testing framework—the API was not designed to provide classes and methods that were appropriate for the interaction with them. These tools needed to go into the JUnit classes and even use reflection to get the necessary information. If designers or JUnit decided to change the name of a private variable, this change could affect the tools that were accessing it reflectively. Maintaining the interaction with JUnit 4 was hard work, so the popularity and simplicity of the framework became obstacles.

Consequently, because everyone was using the same single jar and because all tools and IDEs were so tightly coupled to JUnit 4, its possibilities for evolution were seriously reduced. A new API designed for such tools resulted, along with a new architecture. As described in the stories in section 3.1, a need arose for smaller, modular things. This need was met with JUnit 5.

3.3 **The JUnit 5 architecture**

The new approach didn't come about instantly; it required time and analysis. The shortcomings of JUnit 4 were very good indicators of the needed improvements. Architects knew the problems, and they decided to take the path of breaking the single JUnit 4 jar file into several smaller files.

3.3.1 **JUnit 5 modularity**

A modular approach was needed to allow the JUnit framework to evolve. The architecture had to allow JUnit to interact with different programmatic clients that used different tools and IDEs. The logical separation of concerns required

- An API to write tests, mainly for use by developers
- A mechanism for discovering and running tests
- An API to allow easy interaction with IDEs and tools and to run tests from them

As a result, the JUnit 5 architecture contains three modules (figure 3.7):

- *JUnit Platform* serves as a foundation for launching testing frameworks on the Java virtual machine (JVM). It also provides an API to launch tests from the console, IDEs, and build tools.
- *JUnit Jupiter* combines the new programming and extension model for writing tests and extensions in JUnit 5. The name comes from the fifth planet of our solar system, which is also the largest.
- *JUnit Vintage* is a test engine for running JUnit 3– and JUnit 4–based tests on the platform, ensuring backward compatibility.



Figure 3.7 The modular architecture of JUnit 5

3.3.2 *JUnit Platform*

Going further with the modularity idea, let's take a brief look at the artifacts contained in the JUnit Platform:

- `junit-platform-commons`—A common internal library of JUnit intended solely for use within the JUnit framework. No use by external parties is supported.
- `junit-platform-console`—Provides support for discovering and executing tests on the JUnit Platform from the console.
- `junit-platform-console-standalone`—An executable jar with all dependencies included. This artifact is used by Console Launcher, a command-line Java application that lets us launch the JUnit Platform from the console. It can be used to run JUnit Vintage and JUnit Jupiter tests, for example, and to print test execution results to the console.
- `junit-platform-engine`—A public API for test engines.
- `junit-platform-launcher`—A public API for configuring and launching test plans, typically used by IDEs and build tools.
- `junit-platform-runner`—A runner for executing tests and test suites on the JUnit Platform in a JUnit 4 environment.
- `junit-platform-suite-api`—Contains the annotations for configuring test suites on the JUnit Platform.

- `junit-platform-surefire-provider`—Provides support for discovering and executing tests on the JUnit Platform by using Maven Surefire.
- `junit-platform-gradle-plugin`—Provides support for discovering and executing tests on the JUnit Platform by using Gradle.

3.3.3 *JUnit Jupiter*

JUnit Jupiter is a combination of the new programming model (annotations, classes, and methods) and an extension model for writing tests and extensions in JUnit 5. The Jupiter subproject provides a `TestEngine` for running Jupiter-based tests on the platform. Unlike the existing runner and rule extension points in JUnit 4, the JUnit Jupiter extension model consists of a single coherent concept: the `Extension API`, discussed in chapter 14.

The artifacts contained in JUnit Jupiter are as follows:

- `junit-jupiter-api`—JUnit Jupiter API for writing tests and extensions
- `junit-jupiter-engine`—JUnit Jupiter test engine implementation, required only at runtime
- `junit-jupiter-params`—Provides support for parameterized tests in JUnit Jupiter
- `junit-jupiter-migrationsupport`—Provides migration support from JUnit 4 to JUnit Jupiter and is required only for running selected JUnit 4 rules

3.3.4 *JUnit Vintage*

JUnit Vintage provides a `TestEngine` for running JUnit 3- and JUnit 4-based tests on the platform. It contains only `junit-vintage-engine`, the engine implementation to execute tests written in JUnit 3 or 4. For this purpose, of course, you also need the JUnit 3 or 4 jar.

This engine is very useful for interacting with old tests through JUnit 5. You may need to work on projects with JUnit 5 but still support many old tests, and JUnit Vintage is the solution in this situation.

3.3.5 *The big picture of the JUnit 5 architecture*

The JUnit Platform provides the facilities for running different kinds of tests: JUnit 3, 4, and 5 tests, as well as third-party tests (figure 3.8). Here are the details at the level of the jar files (figure 3.9):

- The test APIs provide the facilities for different test engines: `junit-jupiter-api` for JUnit 5 tests, `junit-4.12` for legacy tests, and custom engines for third-party tests.
- The test engines mentioned earlier are created by extending the `junit-platform-engine` public API, which is part of the JUnit Platform.
- The `junit-platform-launcher` public API provides the facilities to discover tests inside the JUnit Platform for build tools such as Maven and Gradle and for IDEs.

In addition to the modular architecture, JUnit 5 provides the extensions mechanism, which is discussed in chapter 4.

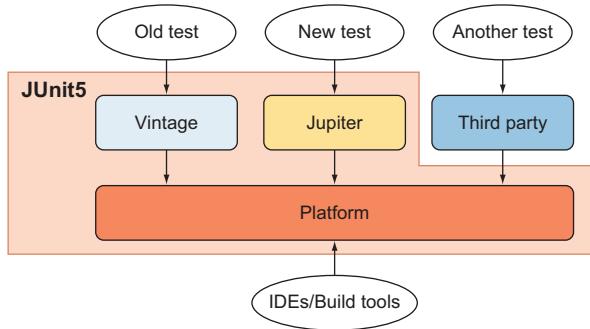


Figure 3.8 The big picture of the JUnit 5 architecture

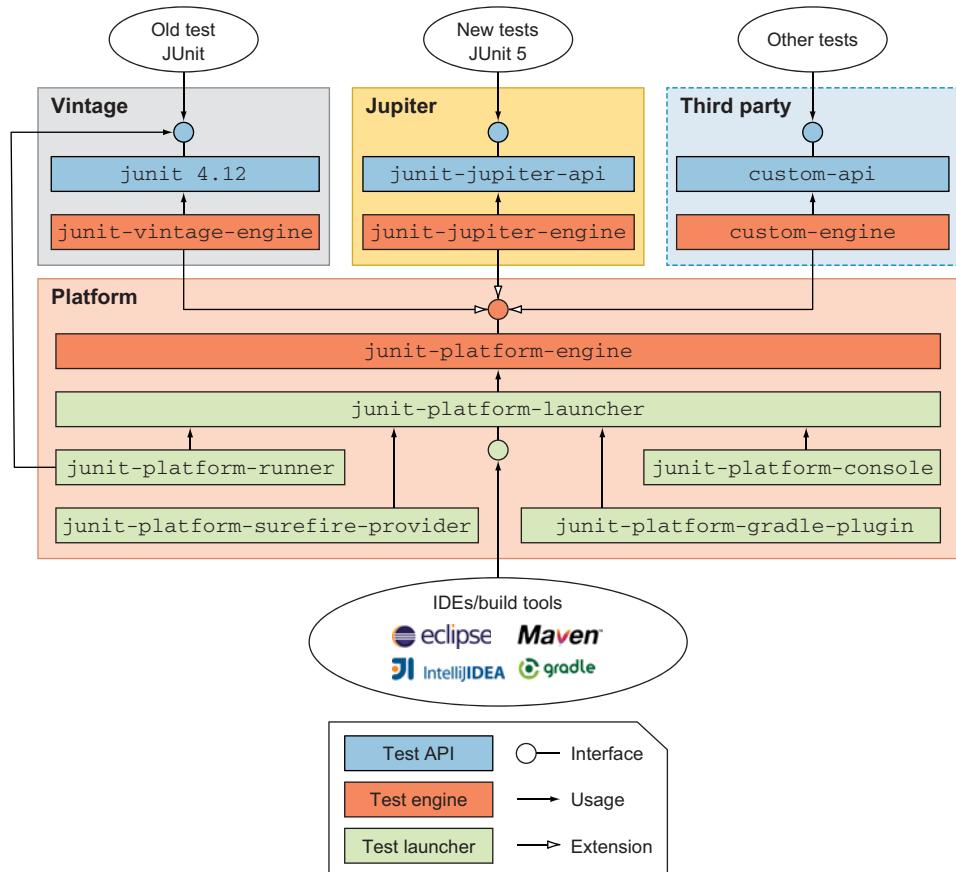


Figure 3.9 A detailed picture of the JUnit 5 architecture

The architecture of a system strongly determines its capabilities and behavior. Consider the runners and rules analyzed earlier in this chapter. Their way of working is generated by the JUnit 4 architecture that represents their foundation. Understanding the architectures of both JUnit 4 and JUnit 5 helps us apply their capabilities in practice, write efficient tests, and analyze alternative implementations, thereby quickening the pace at which we can master unit testing. In chapter 4, we analyze the process of migrating projects from JUnit 4 to JUnit 5 and the dependencies that are required.

Summary

This chapter has covered the following:

- The idea of software architecture as the fundamental structure of a software system. A software system structure comprises software elements, relationships among them, and properties of both elements and relationships.
- An analysis of the monolithic JUnit 4 architecture. It is simple, but real-life challenges like interactions with IDEs and build tools emphasize its shortcomings.
- Using JUnit 4 runners and rules, which represent possibilities to extend the monolithic architecture. They are and will continue to be in use, because there are many existing tests and because migrating them to JUnit 5 extensions is not straightforward.
- Contrasting the JUnit 4 architecture with the JUnit 5 architecture, which is modular and contains the following components: JUnit Platform, JUnit Jupiter, and JUnit Vintage.
- Details concerning the big picture of the JUnit 5 architecture and the interaction between the components.
- The JUnit Platform, which serves as a foundation for launching testing frameworks on the JVM and provides an API to launch tests from the console, IDEs, or build tools.
- JUnit Jupiter, which represents the combination of the new programming model and an extension model for writing tests and extensions in JUnit 5.
- JUnit Vintage, the test engine for running JUnit 3– and JUnit 4–based tests on the platform, ensuring the necessary backward compatibility.

Migrating from JUnit 4 to JUnit 5

This chapter covers

- Implementing the migration from JUnit 4 to JUnit 5
- Working with a hybrid approach for mature projects
- Comparing the needed JUnit 4 with JUnit 5 dependencies
- Comparing the equivalent JUnit 4 with JUnit 5 annotations
- Comparing JUnit 4 rules with JUnit 5 extensions

Nothing in this world can survive and remain useful without an update.

—Charles M. Tadros

So far, this book has introduced JUnit and its latest version, 5. We've discussed the core classes and methods and presented them in action so that you have a good understanding of how to build your tests efficiently. We've emphasized the importance of software architecture in general and shown the significant architectural changes between JUnit 4 and JUnit 5.

This chapter demonstrates how to make the step from JUnit 4 to JUnit 5 inside a project managed by our example company, Tested Data Systems Inc. The company keeps its customer information in a repository and addresses this repository to get the data. In addition, the company needs to track payments and other business rules.

JUnit 4 and JUnit 5 can work together within the same application. This fact is of particular benefit for implementing a migration in phases rather than all at once.

4.1 **Migration steps between JUnit 4 and JUnit 5**

JUnit 5 is a new paradigm introducing a new architecture. It also introduces new packages, annotations, methods, and classes. Some JUnit 5 features are similar to JUnit 4 features; others are new, providing new capabilities. The JUnit Jupiter programming and extension model does not natively support JUnit 4 features such as rules and runners. We do not need to update all existing tests, test extensions, and custom-built test infrastructure to migrate projects to JUnit Jupiter—at least, not instantly.

JUnit provides a migration path with the help of the JUnit Vintage test engine; table 4.1 summarizes the most important steps. This offers the possibility to execute tests based on old JUnit versions using the JUnit Platform infrastructure. All classes and annotations specific to JUnit Jupiter are located in the new `org.junit.jupiter` base package. All classes and annotations specific to JUnit 4 are located in the old `org.junit` base package. So, having both JUnit 4 and JUnit 5 Jupiter in the classpath does not result in a conflict. Consequently, your projects can keep previously implemented JUnit 4 tests together with JUnit Jupiter tests. JUnit 5 and JUnit 4 can coexist until you finalize your migration, whenever that may be, and the migration can be planned and executed slowly based on the priority of the tasks and the challenges of the various steps.

Table 4.1 Migrating from JUnit 4 and JUnit 5

Main step	Comments
Replace the needed dependencies.	JUnit 4 needs a single dependency. JUnit 5 requires more dependencies related to the features that are used. JUnit 5 uses JUnit Vintage to work with old JUnit 4 tests.
Replace the annotations, and introduce the new ones.	Some JUnit 5 annotations mirror those in JUnit 4. Some new annotations introduce new facilities and help us write better tests.
Replace the testing classes and methods.	JUnit 5 assertions and assumptions have been moved to different classes in different packages.
Replace the JUnit 4 rules and runners with the JUnit 5 extension model.	This step generally requires more effort than the other steps in this table. However, because JUnit 4 and JUnit 5 can coexist for an extended period, the rules and runners can remain in the code or be replaced much later.

Before developing and running JUnit tests,

- JUnit 4 requires Java 5 or later.
- JUnit 5 requires Java 8 or later.

Consequently, migrating from JUnit 4 to JUnit 5 can require an update of the Java version used in the project.

4.2 Needed dependencies

This section discusses the Tested Data Systems migration process from JUnit 4 to JUnit 5. The company has decided to migrate because it needs to create more testing code for its products and wants more flexibility and more clarity for writing these tests. JUnit 5 lets us label tests with display names using its nested tests and dynamic tests. Before Tested Data Systems can work effectively with these new JUnit 5 capabilities, the company needs to take the first step of creating JUnit 5 tests for its projects.

As we have explained, JUnit 4 has a monolithic architecture, so there is a single dependency in the Maven configuration that supports running JUnit 4 tests.

Listing 4.1 JUnit 4 Maven dependency

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

One JUnit 5 dependency, JUnit Vintage, can replace the dependency from listing 4.1 during migration. The first things Tested Data Systems needs to do in the migration process are at the level of the dependencies that are used.

The first dependency is `junit-vintage-engine` (listing 4.2). It belongs to JUnit 5 but ensures backward compatibility with previous versions of JUnit. Working with JUnit 5 Vintage, Maven transitively accesses the dependency to JUnit 4. Introducing this dependency is a first step in the migration Tested Data Systems has decided to implement for its projects. JUnit 4 and JUnit 5 tests can coexist within the same project until the migration process is finalized.

Listing 4.2 JUnit Vintage Maven dependency

```
<dependencies>
  <dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Running the JUnit 4 tests now, we can see that they are successfully executed, as shown in figure 4.1. Working with the JUnit 5 Vintage dependency instead of the old JUnit 4 dependency will not make any difference.

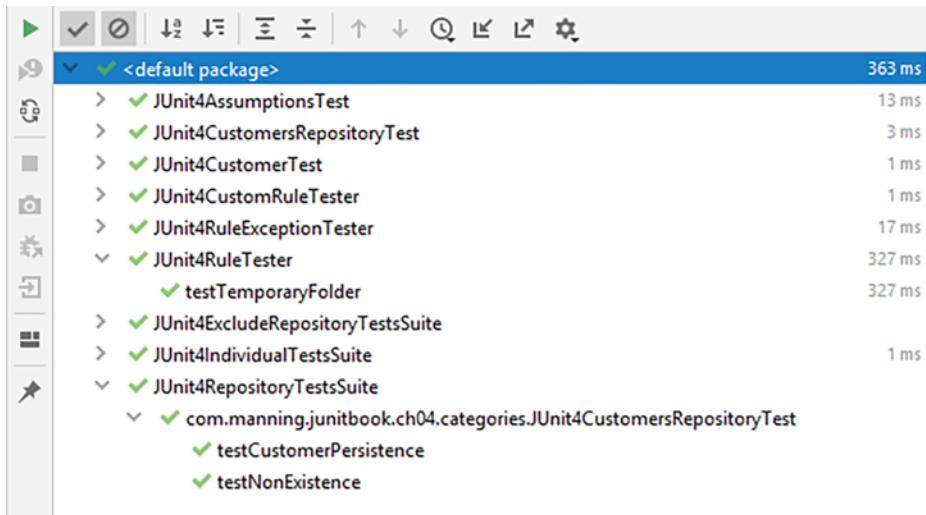


Figure 4.1 Running the JUnit 4 tests after replacing the old JUnit 4 dependency with JUnit 5 Vintage

After the company's programmers introduce the JUnit Vintage dependency, the migration path of Tested Data Systems' projects can continue with the introduction of JUnit 5 Jupiter annotations and features. The required dependencies are shown in the following listing.

Listing 4.3 The most useful JUnit Jupiter Maven dependencies

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

To write tests using JUnit 5, we will always need the `junit-jupiter-api` and `junit-jupiter-engine` dependencies. The first represents the API for writing tests with

JUnit Jupiter (including the annotations, classes, and methods to be migrated to). The second represents the core JUnit Jupiter package for the execution test engine.

An additional dependency that we may need is `junit-jupiter-params` (for running parameterized tests). At the end of the migration process (when no more JUnit 4 tests are left), we can remove the first `junit-vintage-engine` dependency, presented in listing 4.2.

4.3 Annotations, classes, and methods

After making the move at the level of project dependencies, Tested Data Systems continues implementing the migration process. As explained earlier in this book, JUnit 5 provides features similar to JUnit 4 as well as a lot of new ones. Tested Data Systems hopes to benefit from the flexibility and clarity of these new features for the many tests that need to be developed for its projects. First, however, the company needs to migrate the equivalent features.

4.3.1 Equivalent annotations, classes, and methods

Tables 4.2, 4.3, and 4.4 summarize the equivalent annotations, assertions, and assumptions between JUnit 4 and JUnit 5.

Table 4.2 Annotations

JUnit 4	JUnit 5
<code>@BeforeClass</code> , <code>@AfterClass</code>	<code>@BeforeAll</code> , <code>@AfterAll</code>
<code>@Before</code> , <code>@After</code>	<code>@BeforeEach</code> , <code>@AfterEach</code>
<code>@Ignore</code>	<code>@Disabled</code>
<code>@Category</code>	<code>@Tag</code>

Table 4.3 Assertions

JUnit 4	JUnit 5
<code>Assert class.</code>	<code>Assertions class.</code>
Optional assertion message is the first parameter.	Optional assertion message is the last parameter.
<code>assertThat</code> method.	<code>assertThat</code> method has been removed. New methods are <code>assertAll</code> and <code>assertThrows</code> .

Table 4.4 Assumptions

JUnit 4	JUnit 5
<code>Assume class.</code> <code>assumeNotNull</code> and <code>assumeNoException</code> .	<code>Assumptions class.</code> <code>assumeNotNull</code> and <code>assumeNoException</code> have been removed.

Now the Tested Data Systems engineers decide to take the next step by applying the needed changes to the code from their projects. This section uses tests similar to the Tested Data Systems examples, which are built with both JUnit 4 and JUnit 5, to clarify the changes that have been introduced in JUnit 5.

We start with a class that simulates a system under test (SUT). This class can be initialized, can receive regular work but cannot receive additional work to execute, and can close itself.

Listing 4.4 Tested SUT class

```
public class SUT {
    private String systemName;

    public SUT(String systemName) {
        this.systemName = systemName;
        System.out.println(systemName + " from class " +
            getClass().getSimpleName() + " is initializing.");
    }

    public boolean canReceiveRegularWork() {
        System.out.println(systemName + " from class " +
            getClass().getSimpleName() + " can receive regular work.");
        return true;
    }

    public boolean canReceiveAdditionalWork() {
        System.out.println(systemName + " from class " +
            getClass().getSimpleName() + " cannot receive additional
            work.");
        return false;
    }

    public void close() {
        System.out.println(systemName + " from class " +
            getClass().getSimpleName() + " is closing.");
    }
}
```

Listing 4.5 verifies the functionality of the SUT by using the JUnit 4 facilities, and listing 4.6 verifies the functionality of the SUT by using the JUnit 5 facilities. These examples also demonstrate the life cycle methods. As previously stated, the system can start up, can receive regular work but cannot receive additional work, and can close itself. The JUnit 4 and JUnit 5 life cycle and testing methods ensure that the system is initializing and then shutting down before and after each effective test. The test methods check whether the system can receive regular work and additional work. The annotations in the following listing will be discussed after listing 4.6.

Listing 4.5 JUnit4SUTTest class

```

public class JUnit4SUTTest {

    private static ResourceForAllTests resourceForAllTests;
    private SUT systemUnderTest;

    @BeforeClass
    public static void setUpClass() {
        resourceForAllTests =
            new ResourceForAllTests("Our resource for all tests");
    } 1

    @AfterClass
    public static void tearDownClass() {
        resourceForAllTests.close();
    } 2

    @Before
    public void setUp() {
        systemUnderTest = new SUT("Our system under test");
    } 3

    @After
    public void tearDown() {
        systemUnderTest.close();
    } 4

    @Test
    public void testRegularWork() {
        boolean canReceiveRegularWork =
            systemUnderTest.canReceiveRegularWork();

        assertTrue(canReceiveRegularWork);
    } 5

    @Test
    public void testAdditionalWork() {
        boolean canReceiveAdditionalWork =
            systemUnderTest.canReceiveAdditionalWork();
        assertFalse(canReceiveAdditionalWork);
    } 6

    @Test
    @Ignore
    public void myThirdTest() {
        assertEquals("2 is not equal to 1", 2, 1);
    }
}

```

We previously replaced the JUnit 4 dependency with the JUnit Vintage dependency. The result of running the `JUnit4SUTTest` class is the same in both cases (figure 4.2), and `myThirdTest` is marked with the `@Ignore` annotation. Now we can proceed to the effective migration of annotations, classes, and methods.

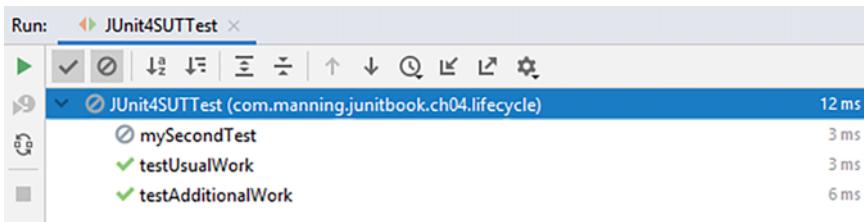


Figure 4.2 Running the JUnit4SUTTestSuite in IntelliJ using both the JUnit 4 dependency and the JUnit Vintage dependency

Listing 4.6 JUnit5SUTTest class

```
class JUnit5SUTTest {
    private static ResourceForAllTests resourceForAllTests;
    private SUT systemUnderTest;

    @BeforeAll
    static void setUpClass() {
        resourceForAllTests =
            new ResourceForAllTests("Our resource for all tests");
    }

    @AfterAll
    static void tearDownClass() {
        resourceForAllTests.close();
    }

    @BeforeEach
    void setUp() {
        systemUnderTest = new SUT("Our system under test");
    }

    @AfterEach
    void tearDown() {
        systemUnderTest.close();
    }

    @Test
    void testRegularWork() {
        boolean canReceiveRegularWork =
            systemUnderTest.canReceiveRegularWork();

        assertTrue(canReceiveRegularWork);
    }

    @Test
    void testAdditionalWork() {
        boolean canReceiveAdditionalWork =
            systemUnderTest.canReceiveAdditionalWork();

        assertFalse(canReceiveAdditionalWork);
    }
}
```

The diagram illustrates the execution flow of the JUnit5SUTTest class. Arrows point from numbered circles to specific code blocks:

- Circle 1: Points to the static `setUpClass()` method.
- Circle 2: Points to the static `tearDownClass()` method.
- Circle 3: Points to the `@BeforeEach` `setUp()` method.
- Circle 4: Points to the `@AfterEach` `tearDown()` method.
- Circle 5: Points to the `@Test` `testRegularWork()` method.

```

    @Test
    @Disabled
    void myThirdTest() {
        assertEquals(2, 1, "2 is not equal to 1");
    }
}

```



Comparing the JUnit 4 and JUnit 5 methods, we see that

- The methods annotated with `@BeforeClass` (1 in listing 4.5) and `@BeforeAll` (1' in listing 4.6), respectively, are executed once, before all tests. These methods need to be static. In the JUnit 4 version, the methods also need to be public. In the JUnit 5 version, we can make the methods nonstatic and annotate the entire test class with `@TestInstance(Life_cycle.PER_CLASS)`.
- The methods annotated with `@AfterClass` (2 in listing 4.5) and `@AfterAll` (2' in listing 4.6), respectively, are executed once, after all tests. These methods need to be static. In the JUnit 4 version, the methods also need to be public. In the JUnit 5 version, we can make the methods nonstatic and annotate the whole test class with `@TestInstance(Life_cycle.PER_CLASS)`.
- The methods annotated with `@Before` (3 in listing 4.5) and `@BeforeEach` (3' in listing 4.6), respectively, are executed before each test. In the JUnit 4 version, the methods need to be public.
- The methods annotated with `@After` (4 in listing 4.5) and `@AfterEach` (4' in listing 4.6), respectively, are executed after each test. In the JUnit 4 version, the methods need to be public.
- The methods annotated with `@Test` (5 in listing 4.5) and `@Test` (5' in listing 4.6) are executed independently. In the JUnit 4 version, the methods need to be public. The two annotations belong to different packages: `org.junit.Test` and `org.junit.jupiter.api.Test`, respectively.
- To skip the execution of a test method, JUnit 4 uses the annotation `@Ignore` (6 in listing 4.5), whereas JUnit 5 uses the annotation `@Disabled` (6' in listing 4.6).

The access level has been relaxed for the test methods, from public to package-private. These methods are accessed only from within the package to which the test class belongs, so they didn't need to be made public.

4.3.2 Categories vs. tags

The Tested Data Systems engineers applied the equivalences presented in tables 4.2, 4.3, and 4.4 to the migration of their code. Now the company needs to verify its customers' information, as well as customers' existence or nonexistence. It wants to classify the verification tests in two groups: those that work with individual customers and those that check inside a repository. The company has used categories (in JUnit 4) and needs to switch to tags (in JUnit 5).

The following listing shows the two interfaces that need to be declared for the definition of JUnit 4 categories. These interfaces will be used as arguments of the JUnit 4 `@Category` annotation.

Listing 4.7 Interfaces created to define categories with JUnit 4

```
public interface IndividualTests {
}

public interface RepositoryTests {
}
```

The next listing defines a JUnit 4 test that contains a method annotated as `@Category(IndividualTests.class)`. This annotation assigns that test method as belonging to this category.

Listing 4.8 JUnit4CustomerTest class; one test method annotated with @Category

```
public class JUnit4CustomerTest {
    private String CUSTOMER_NAME = "John Smith";

    @Category(IndividualTests.class)
    @Test
    public void testCustomer() {
        Customer customer = new Customer(CUSTOMER_NAME);

        assertEquals("John Smith", customer.getName());
    }
}
```

The following listing defines a JUnit 4 test class annotated as `@Category(IndividualTests.class, RepositoryTests.class)`. This annotation assigns the two included test methods as belonging to these two categories.

Listing 4.9 JUnit4CustomersRepositoryTest class annotated with @Category

```
@Category({IndividualTests.class, RepositoryTests.class})
public class JUnit4CustomersRepositoryTest {
    private String CUSTOMER_NAME = "John Smith";
    private CustomersRepository repository = new CustomersRepository();

    @Test
    public void testNonExistence() {
        boolean exists = repository.contains(CUSTOMER_NAME);

        assertFalse(exists);
    }

    @Test
    public void testCustomerPersistence() {
        repository.persist(new Customer(CUSTOMER_NAME));
    }
}
```

```

        assertTrue(repository.contains("John Smith"));
    }
}

```

Listings 4.10, 4.11, and 4.12 describe three suites that look for particular test categories in the given classes.

Listing 4.10 JUnit4IndividualTestsSuite class

```

@RunWith(Categories.class)
@Categories.IncludeCategory(IndividualTests.class)
@Suite.SuiteClasses({JUnit4CustomerTest.class,
    JUnit4CustomersRepositoryTest.class})
public class JUnit4IndividualTestsSuite {
}

```

In this example, `JUnit4IndividualTestsSuite`

- Is annotated with `@RunWith(Categories.class)` ①, informing JUnit that it has to execute the tests with this particular runner
- Includes the category of tests annotated with `IndividualTests` ②
- Looks for these annotated tests in the `JUnit4CustomerTest` and `JUnit4CustomersRepositoryTest` classes ③

The result of running this suite is shown in figure 4.3. All tests from the `JUnit4CustomerTest` and `JUnit4CustomersRepositoryTest` classes will be executed, as all of them are annotated with `IndividualTests`.

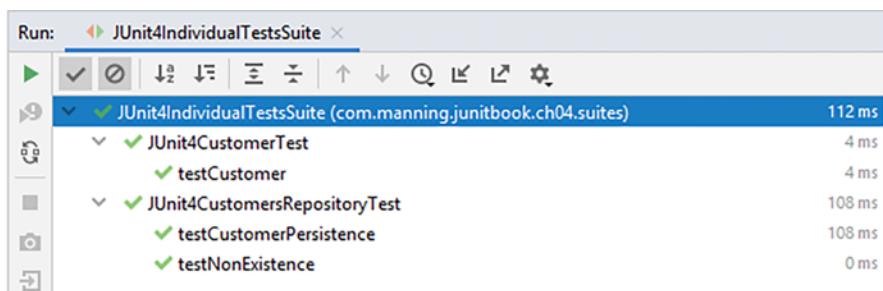


Figure 4.3 Running `JUnit4IndividualTestsSuite` in IntelliJ

Listing 4.11 JUnit4RepositoryTestsSuite class

```

@.RunWith(Categories.class)
@Categories.IncludeCategory(RepositoryTests.class)
@Suite.SuiteClasses({JUnit4CustomerTest.class,
    JUnit4CustomersRepositoryTest.class})
public class JUnit4RepositoryTestsSuite {
}

```

In this example, `JUnit4RepositoryTestsSuite`

- Is annotated with `@RunWith(Categories.class)` ①, informing JUnit that it has to execute the tests with this particular runner
- Includes the category of tests annotated with `RepositoryTests` ②
- Looks for these annotated tests in the `JUnit4CustomerTest` and `JUnit4CustomersRepositoryTest` classes ③

The result of running this suite is shown in figure 4.4. Two tests from the `JUnit4CustomersRepositoryTest` class will be executed, as they are annotated with `RepositoryTests`.

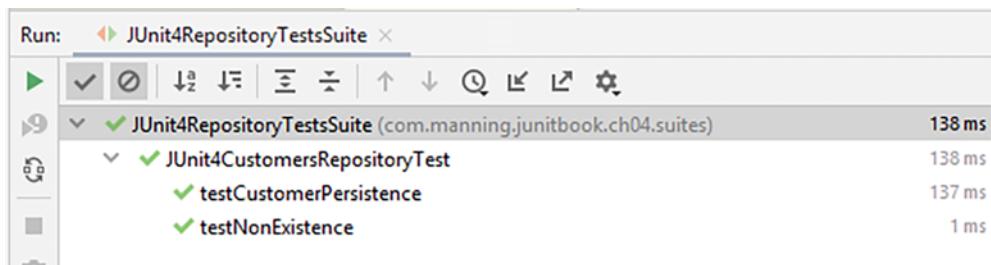


Figure 4.4 Running the `JUnit4RepositoryTestsSuite` in IntelliJ

Listing 4.12 `JUnit4ExcludeRepositoryTestsSuite` class

```

1
@RunWith(Categories.class)
2
@Categories.ExcludeCategory(RepositoryTests.class)
3
@Suite.SuiteClasses({JUnit4CustomerTest.class,
    JUnit4CustomersRepositoryTest.class})
public class JUnit4ExcludeRepositoryTestsSuite {
}

```

In this example, `JUnit4ExcludeRepositoryTestsSuite`

- Is annotated with `@RunWith(Categories.class)` ①, informing JUnit that it has to execute the tests with this particular runner
- Excludes the category of tests annotated with `RepositoryTests` ②
- Looks for these annotated tests in the `JUnit4CustomerTest` and `JUnit4CustomersRepositoryTest` classes ③

The result of running this suite is shown in figure 4.5. One test from the `JUnit4CustomerTest` class will be executed, as it is not annotated with `RepositoryTests`.

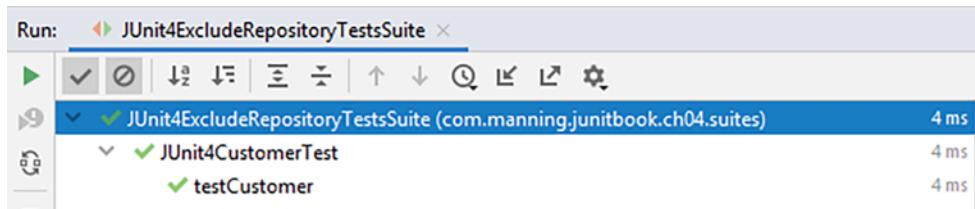


Figure 4.5 Running JUnit4ExcludeRepositoryTestsSuite in IntelliJ

The approach using JUnit 4 categories works, but it also has some shortcomings: it is a lot of code to write, and the team needs to define special test suites and special interfaces to be used solely as marker interfaces for the tests. For that reason, the engineers at Tested Data Systems decide to switch to the JUnit 5 tags approach. Listing 4.13 introduces the `JUnit5CustomerTest` tagged class, and listing 4.14 shows the `CustomersRepositoryTest` tagged class.

Listing 4.13 JUnit5CustomerTest tagged class

```
@Tag("individual")
public class JUnit5CustomerTest {
    private String CUSTOMER_NAME = "John Smith"; ①

    @Test
    void testCustomer() {
        Customer customer = new Customer(CUSTOMER_NAME);

        assertEquals("John Smith", customer.getName());
    }
}
```

The `@Tag` annotation is added to the whole `JUnit5CustomerTest` class ①.

Listing 4.14 JUnit5CustomersRepositoryTest tagged class

```
@Tag("repository")
public class JUnit5CustomersRepositoryTest {
    private String CUSTOMER_NAME = "John Smith";
    private CustomersRepository repository = new CustomersRepository();

    @Test
    void testNonExistence() {
        boolean exists = repository.contains("John Smith");

        assertFalse(exists);
    }

    @Test
    void testCustomerPersistence() {
        repository.persist(new Customer(CUSTOMER_NAME));
    }
}
```

```

        assertTrue(repository.contains("John Smith"));
    }
}

```

Similarly, the `@Tag` annotation is added to the whole `JUnit5CustomersRepositoryTest` class ①.

To activate the JUnit 5 tags that replace the JUnit 4 categories, we have a few alternatives. For one, we can work at the level of the `pom.xml` configuration file.

Listing 4.15 `pom.xml` configuration file

```

<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
    <!--
    <configuration>
        <groups>individual</groups>
        <excludedGroups>repository</excludedGroups>
    </configuration>
    -->
</plugin>

```

Here it is enough to uncomment the configuration node of the Surefire plugin ① and run `mvn clean install`. Or, from the IntelliJ IDEA, we can activate tag functionality by choosing `Run > Edit Configurations` and choosing `Tags (JUnit 5)` as the Test Kind (figure 4.6). However, the recommended way to go is to change `pom.xml` so the tests can be correctly executed from the command line.

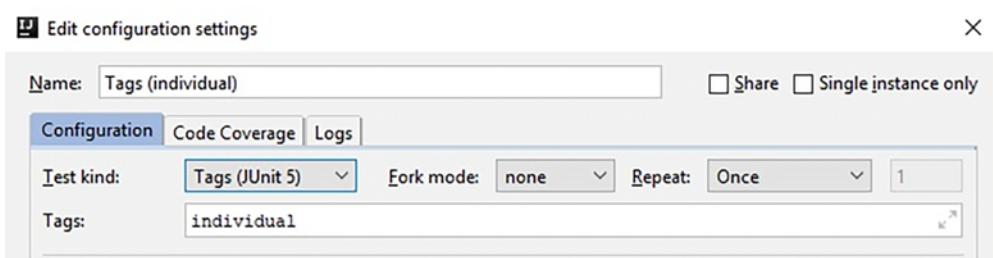


Figure 4.6 Configuring the tagged tests from the IntelliJ IDEA

Notice that we no longer define special interfaces for this goal; we no longer create many tedious suites at the level of the code. We simply annotate the classes with our own tags and then choose what to run by making the selections through the Maven configuration file or the IDE. We have less code to write and fewer changes to make at the level of the code itself. The Tested Data Systems engineers decide to take advantage of the JUnit 5 facilities.

4.3.3 Migrating Hamcrest matcher functionality

To continue our comparison of JUnit 4 and JUnit 5, we will look at the Hamcrest matcher functionality introduced in chapter 2. In this chapter, we use our collections example to put the two versions face to face. We will populate a list with values from Tested Data Systems' internal repository and then investigate whether its elements match some patterns using JUnit 4 (listing 4.16) and JUnit 5 (listing 4.17).

Listing 4.16 JUnit4HamcrestListTest class

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

public class JUnit4HamcrestListTest {
    private List<String> values;

    @Before
    public void setUp() {
        values = new ArrayList<>();
        values.add("Oliver");
        values.add("Jack");
        values.add("Harry");
    }

    @Test
    public void testListWithHamcrest() {
        assertThat(values, hasSize(3));
        assertThat(values, hasItem(anyOf(equalTo("Oliver"), equalTo("Jack"),
            equalTo("Harry"))));
        assertThat("The list doesn't contain all the expected objects, in
            order", values, contains("Oliver", "Jack", "Harry"));
        assertThat("The list doesn't contain all the expected objects",
            values, containsInAnyOrder("Jack", "Harry", "Oliver"));
    }
}
```

Listing 4.17 JUnit5HamcrestListTest class

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

public class JUnit5HamcrestListTest {
    private List<String> values;

    @BeforeEach
    public void setUp() {
        values = new ArrayList<>();
        values.add("Oliver");
        values.add("Jack");
        values.add("Harry");
    }
}
```

```

 2' @Test
 2' @DisplayName("List with Hamcrest")
 2' public void testListWithHamcrest() {
 2'     assertThat(values, hasSize(3));
 2'     assertThat(values, hasItem(anyOf(equalTo("Oliver"), equalTo("Jack"),
 2'         equalTo("Harry"))));
 2'     assertThat("The list doesn't contain all the expected objects, in #
 2'         order", values, contains("Oliver", "Jack", "Harry"));
 2'     assertThat("The list doesn't contain all the expected objects",
 2'         values, containsInAnyOrder("Jack", "Harry", "Oliver"));
 2' }
 5' }
 3' 4'
 5' 6'
 6'

```

These examples are very similar (except for the `@Before`/`@BeforeEach` and `@DisplayName` annotations). The old imports are gradually replaced by the new ones, which is why the annotations `org.junit.Test` and `org.junit.jupiter.api.Test` belong to different packages.

What do these programs do with the internal information managed at Tested Data Systems?

- We initialize the list to work with. The code is the same, but the annotations are different: `@Before` ① and `@BeforeEach` ①'.
- The test methods are annotated with `org.junit.Test` ② and `org.junit.jupiter.api.Test` ②', respectively.
- Verification ③ uses the `org.junit.Assert.assertThat` method. JUnit 5 removes this method, so we use `org.hamcrest.MatcherAssert.assertThat` ③'.
- We use the `anyOf` and `equalTo` methods from the `org.hamcrest.Matchers` class ④ and the `anyOf` and `equalTo` methods from the `org.hamcrest.CoreMatchers` class ④'.
- We use the same `org.hamcrest.Matchers.contains` method (⑤ and ⑤').
- We use the same `org.hamcrest.Matchers.containsInAnyOrder` method (⑥ and ⑥').

4.3.4 Rules vs. the extension model

A JUnit 4 rule is a component that lets us introduce additional actions when a method is executed, by intercepting its call and doing something before and after the execution of the method. To compare the JUnit 4 rules model and the JUnit 5 extension model, we'll revisit our extended `Calculator` class (listing 4.18). The developers at Tested Data Systems use this class to execute mathematical operations for verifying their SUTs. They are interested in testing the methods that can throw exceptions. One rule the Tested Data Systems test code uses extensively is `ExpectedException`, which can easily be replaced by the JUnit 5 `assertThrows` method.

Listing 4.18 Extended Calculator class

```
public class Calculator {
    ...

    public double sqrt(double x) {
        if (x < 0) {
            throw new
                IllegalArgumentException("Cannot extract the square
                                         root of a negative value");
        }
        return Math.sqrt(x);
    }

    public double divide(double x, double y) {
        if (y == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }
        return x/y;
    }
}
```

The logic that can throw exceptions in the Calculator class does the following:

- Declares a method to calculate the square root of a number ①. If the number is negative, an exception containing a message is created and thrown ②.
- Declares a method to divide two numbers ③. If the second number is zero, an exception containing a message is created and thrown ④.

The following listing provides an example that specifies which exception message is expected during the execution of the test code, using the new functionality of the Calculator class (see listing 4.18).

Listing 4.19 JUnit4RuleExceptionTester class

```
public class JUnit4RuleExceptionTester {
    @Rule
    public ExpectedException expectedException =
        ExpectedException.none();

    private Calculator calculator = new Calculator();
    @Test
    public void expectIllegalArgumentException() {
        expectedException.expect(IllegalArgumentException.class);
        expectedException.expectMessage("Cannot extract the square root
                                         of a negative value");
        calculator.sqrt(-1);
    }
}
```

```

    @Test
    public void expectArithmeticException() {
        expectedException.expect(ArithmeticException.class);
        expectedException.expectMessage("Cannot divide by zero");
        calculator.divide(1, 0);
    }
}

```

In this example:

- We declare an `ExpectedException` field annotated with `@Rule`. The `@Rule` annotation must be applied on a public nonstatic field or a public nonstatic method ①. The `ExpectedException.none()` factory method simply creates an unconfigured `ExpectedException`.
- We initialize an instance of the `Calculator` class whose functionality we are testing ②.
- `ExpectedException` is configured to expect the type of exception ③ and the message ④ before being thrown by invoking the `sqrt` method ⑤.
- `ExpectedException` is configured to expect the type of exception ⑥ and message ⑦ before being thrown by invoking the `divide` method ⑧.

The next listing shows the JUnit 5 approach.

Listing 4.20 JUnit5ExceptionTester class

```

public class JUnit5ExceptionTester {
    private Calculator calculator = new Calculator();
}

@Test
public void expectIllegalArgumentException() {
    Throwable throwable = assertThrows(
        IllegalArgumentException.class,
        () -> calculator.sqrt(-1));
    assertEquals("Cannot extract the square root of a
                negative value", throwable.getMessage());
}

@Test
public void expectArithmeticException() {
    Throwable throwable = assertThrows(ArithmeticException.class,
        () -> calculator.divide(1, 0));
    assertEquals("Cannot divide by zero", throwable.getMessage());
}

```

In this example:

- We initialize an instance of the `Calculator` class whose functionality we are testing ①'.

- We assert that the execution of the supplied `calculator.sqrt(-1)` executable throws an `IllegalArgumentException` ②', and we check the message from the exception ③'.
- We assert that the execution of the supplied `calculator.divide(1, 0)` executable throws an `ArithmaticException` ④', and we check the message from the exception ⑤'.

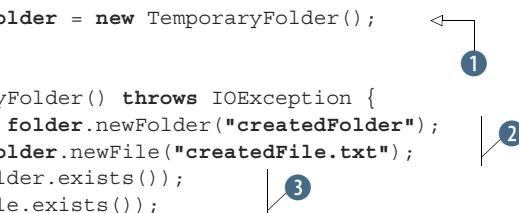
A clear difference exists between JUnit 4 and JUnit 5 in the clarity and length of the code. The effective JUnit 5 testing code is 13 lines, whereas the effective JUnit 4 code is 20 lines. We do not need to initialize and manage any additional rules. The JUnit 5 testing methods contain one line each.

Another rule that Tested Data Systems would like to migrate is `TemporaryFolder`. The `TemporaryFolder` rule lets us create files and folders that should be deleted when the test method finishes (whether it passes or fails). Because the tests in Tested Data Systems' projects work intensively with temporary resources, this step is also a must. The JUnit 4 rule has been replaced by the `@TempDir` annotation in JUnit 5. The following listing presents the JUnit 4 approach.

Listing 4.21 JUnit4RuleTester class

```
public class JUnit4RuleTester {
    @Rule
    public TemporaryFolder folder = new TemporaryFolder(); ①

    @Test
    public void testTemporaryFolder() throws IOException {
        File createdFolder = folder.newFolder("createdFolder");
        File createdFile = folder.newFile("createdFile.txt");
        assertTrue(createdFolder.exists()); ②
        assertTrue(createdFile.exists()); ③
    }
}
```



In this example:

- We declare a `TemporaryFolder` field annotated with `@Rule` and initialize it. The `@Rule` annotation must be applied on a public field or a public method ①.
- We use the `TemporaryFolder` field to create a folder and a file ②, located in the Temp folder of our user profile in the OS.
- We check the existence of the temporary folder and the temporary file ③.

And here is the new JUnit 5 approach.

Listing 4.22 JUnit5TempDirTester class

```
public class JUnit5TempDirTester {
    @TempDir
    Path tempDir; ①'
```

```

    2'    private static Path createdFile;

    @Test
    public void testTemporaryFolder() throws IOException {
        assertTrue(Files.isDirectory(tempDir));
        createdFile = Files.createFile(
            tempDir.resolve("createdFile.txt")
        );
        assertTrue(createdFile.toFile().exists());
    }

    @AfterAll
    public static void afterAll() {
        assertFalse(createdFile.toFile().exists());
    }
}

```

In this example:

- We declare a `@TempDir` annotated field 1'.
- We declare the `createdFile` variable 2'.
- We check the creation of this temporary directory before the execution of the test 3'.
- We create a file within this directory and check its existence 4'.
- After the execution of the tests, we check that the temporary resource has been removed 5'. The temporary folder is removed after the execution of the `afterAll` method ends.

The advantage of the JUnit 5 extension approach is that we do not have to create the folder ourselves through a constructor; the folder is created automatically when we annotate a field with `@TempDir`.

4.3.5 **Custom rules**

Tested Data Systems has defined custom rules for its tests. Custom rules are particularly useful when some tests need similar additional actions before and after execution.

In JUnit 4, the Tested Data Systems engineers needed their additional actions to be executed before and after the execution of a test. So, they created their own classes to implement the `TestRule` interface (see listings 4.23–4.25). To do this, they had to override the `apply(Statement, Description)` method, which returns an instance of `Statement`. Such an object represents the tests within the JUnit runtime, and `Statement#evaluate()` runs them. The `Description` object describes the individual test. This object can be used to read information about the test through reflection.

Listing 4.23 CustomRule class

```
public class CustomRule implements TestRule {
    private Statement base;
    private Description description;

    @Override
    public Statement apply(Statement base, Description description) {
        this.base = base;
        this.description = description;
        return new CustomStatement(base, description);
    }
}
```

In this example:

- We declare our `CustomRule` class that implements the `TestRule` interface ①.
- We keep references to a `Statement` field and a `Description` field ②, and we use them in the `apply` method that returns a `CustomStatement` ③.

Listing 4.24 CustomStatement class

```
public class CustomStatement extends Statement {
    private Statement base;
    private Description description;

    public CustomStatement(Statement base, Description description) {
        this.base = base;
        this.description = description;
    }

    @Override
    public void evaluate() throws Throwable {
        System.out.println(this.getClass().getSimpleName() + " " +
                           description.getMethodName() + " has started");
        try {
            base.evaluate();
        } finally {
            System.out.println(this.getClass().getSimpleName() + " " +
                           description.getMethodName() + " has finished");
        }
    }
}
```

In this example:

- We declare our `CustomStatement` class that extends the `Statement` class ①.
- We keep references to a `Statement` field and to a `Description` field ②, and we use them as arguments of the constructor ③.
- We override the inherited `evaluate` method and call `base.evaluate()` inside it ④.

Listing 4.25 JUnit4CustomRuleTester class

```
public class JUnit4CustomRuleTester {

    @Rule
    public CustomRule myRule = new CustomRule(); ①

    @Test
    public void myCustomRuleTest() {
        System.out.println("Call of a test method");
    }
}
```

In this example, we use the previously defined `CustomRule` by doing the following:

- We declare a public `CustomRule` field and annotate it with `@Rule` ①.
- We create the `myCustomRuleTest` method and annotate it with `@Test` ②.

The result of executing this test is shown in figure 4.7. As the engineers from Tested Data Systems required, the effective execution of the test is surrounded by the additional messages provided to the `evaluate` method of the `CustomStatement` class.

```
Tests passed: 1 of 1 test - 5 ms
"C:\Program Files\Java\jdk1.8.0_181\bin\java" ...
CustomStatement myCustomRuleTest has started
Call of a test method
CustomStatement myCustomRuleTest has finished

Process finished with exit code 0
```

Figure 4.7 The result of executing `JUnit4CustomRuleTester`

The engineers from Tested Data Systems would like to migrate their own rules as well. JUnit 5 allows effects similar to those of the JUnit 4 rules by introducing custom extensions that extend the behavior of test classes and methods. The code is shorter and relies on the declarative annotations style. First, we define the `CustomExtension` class, which is used as an argument of the `@ExtendWith` annotation on the tested class.

Listing 4.26 CustomExtension class

```
public class CustomExtension implements AfterEachCallback, ①
BeforeEachCallback {
    @Override
    public void beforeEach(ExtensionContext extensionContext)
        throws Exception {
        System.out.println(this.getClass().getSimpleName() + " " +
            extensionContext.getDisplayName() + " has started");
    }
}
```

```

@Override
public void afterEach(ExtensionContext extensionContext)
    throws Exception {
    System.out.println(this.getClass().getSimpleName() + " " +
        extensionContext.getDisplayName() + " has finished");
}
}

```

In this example:

- We declare `CustomExtension` as implementing the `AfterEachCallback` and `BeforeEachCallback` interfaces ①'.
- We override the `beforeEach` method, to be executed before each test method from the testing class that will be extended with `CustomExtension` ②'.
- We override the `afterEach` method, to be executed after each test method from the testing class that will be extended with `CustomExtension` ③'.

Listing 4.27 The JUnit5CustomExtensionTester class

```

@ExtendWith(CustomExtension.class)
public class JUnit5CustomExtensionTester {
    {
        @Test
        public void myCustomRuleTest() {
            System.out.println("Call of a test method");
        }
    }
}

```

In this example:

- We extend `JUnit5CustomExtensionTester` with the `CustomExtension` class ①'.
- We create the `myCustomRuleTest` method and annotate it with `@Test` ②'.

The result of executing this test is shown in figure 4.8. Because the test class is extended with the `CustomExtension` class, the previously defined `beforeEach` and `afterEach` methods are executed before and after each test method, respectively.

Notice the clear difference between JUnit 4 and JUnit 5 in the clarity and length of the code. The JUnit 4 approach needs to work with three classes; the JUnit 5 approach

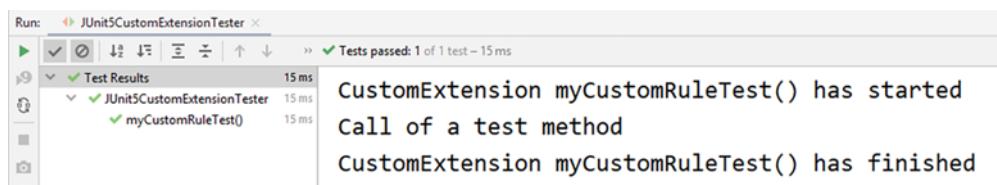


Figure 4.8 The result of executing `JUnit5CustomExtensionTester`

needs to work with only two. The code to be executed before and after each test method is isolated in a dedicated method with a clear name. We only need to annotate the testing class with `@ExtendWith`.

The JUnit 5 extension model can also be used to replace the runners from JUnit 4 gradually. For the extensions that have already been created, the migration process is simple:

- To migrate the Mockito tests, we need to replace the annotation `@RunWith(MockitoJUnitRunner.class)` in the tested class with the annotation `@ExtendWith(MockitoExtension.class)`.
- To migrate the Spring tests, we need to replace the annotation `@RunWith(SpringJUnit4ClassRunner.class)` in the tested class with the annotation `@ExtendWith(SpringExtension.class)`.

When this chapter was written, there was no extension for the Arquillian tests. The runners will be discussed in more detail in later chapters.

In chapter 5, we find out more about software testing principles and test types, the need to start from unit tests, and ways to advance to different testing levels.

Summary

This chapter has covered the following:

- Applying the steps needed by the migration from JUnit 4 to JUnit 5, and summarizing them into a guiding table: replace the dependencies, replace the annotations, replace the testing classes and methods, and replace the JUnit 4 rules and the runners with the JUnit 5 extension model.
- Using the needed dependencies in both cases, making the change from a single dependency in JUnit 4 to multiple dependencies in JUnit 5.
- Comparing and using the equivalent annotations, classes, and methods of JUnit 4 and JUnit 5, which drive similar effects: defining tests, controlling the life cycle of a test, and checking the results.
- Using examples to implement the effective changes in code when migrating from JUnit 4 to JUnit 5.



Software testing principles

This chapter covers

- Examining the need for unit tests
- Differentiating between types of software tests
- Comparing black-box and white-box testing

A crash is when your competitor's program dies. When your program dies, it is an "idiosyncrasy." Frequently, crashes are followed with a message like "ID 02." "ID" is an abbreviation for idiosyncrasy, and the number that follows indicates how many more months of testing the product should have had.

—Guy Kawasaki

Earlier chapters in this book took a very pragmatic approach to designing and deploying unit tests. We took a deep look at JUnit 5's capabilities and the architectures of both JUnit 4 and JUnit 5 and demonstrated how to migrate between the two versions. This chapter steps back to look at the various types of software tests and the roles they play in the application life cycle.

Why do you need to know all this information? Unit testing is not something you do without planning and preparation. To become a top-level developer, you

need to contrast unit tests with functional and other types of tests. *Functional testing* simply means evaluating the compliance of a system or component with the requirements. When you understand why unit tests are necessary, you also need to know how far to take your tests. Testing in and of itself is not the goal.

5.1 **The need for unit tests**

The main goals of unit testing are to verify that your application works as expected and to catch bugs early. Although functional testing helps you accomplish the same goals, unit tests are extremely powerful and versatile and offer much more. Unit tests

- Allow greater test coverage than functional tests
- Increase team productivity
- Detect regressions and limit the need for debugging
- Give us the confidence to refactor and, in general, make changes
- Improve implementation
- Document expected behavior
- Enable code coverage and other metrics

5.1.1 **Allowing greater test coverage**

Unit tests are the first type of tests any application should have. If you have to choose between writing unit tests and functional tests, you should choose to write the second kind. In our experience, functional tests cover about 70% of the application code. If you want to go further and provide more test coverage, you need to write unit tests.

Unit tests can easily simulate error conditions, which is extremely difficult for functional tests to do (and impossible in some instances). Unit tests also provide much more than just testing, as explained in the following sections.

5.1.2 **Increasing team productivity**

Imagine that you are on a team working on a large application. Unit tests allow you to deliver quality code (tested code) without waiting for all the other components to be ready. On the other hand, functional tests are more coarse grained and need the full application, or a good part of it, to be ready before you can test it.

5.1.3 **Detecting regressions and limiting debugging**

A passing unit test suite confirms that your code works and gives you the confidence to modify your existing code, either for refactoring or to add and modify new features. As a developer, you can have no better feeling than knowing someone is watching your back and will warn you if you break something.

A suite of unit tests reduces the need to debug an application to find out why something is failing. Whereas a functional test tells you that a bug exists somewhere in the implementation of a use case, a unit test tells you that a specific method is failing for a specific reason. You no longer need to spend hours trying to find the problem.

5.1.4 Refactoring with confidence

Without unit tests, it is difficult to justify refactoring because there is always a relatively high chance that you will break something. Why would you risk spending hours of debugging time (and put delivery at risk) only to improve the implementation or change a method name? As shown in figure 5.1, unit tests provide the safety net that gives you the confidence to refactor.

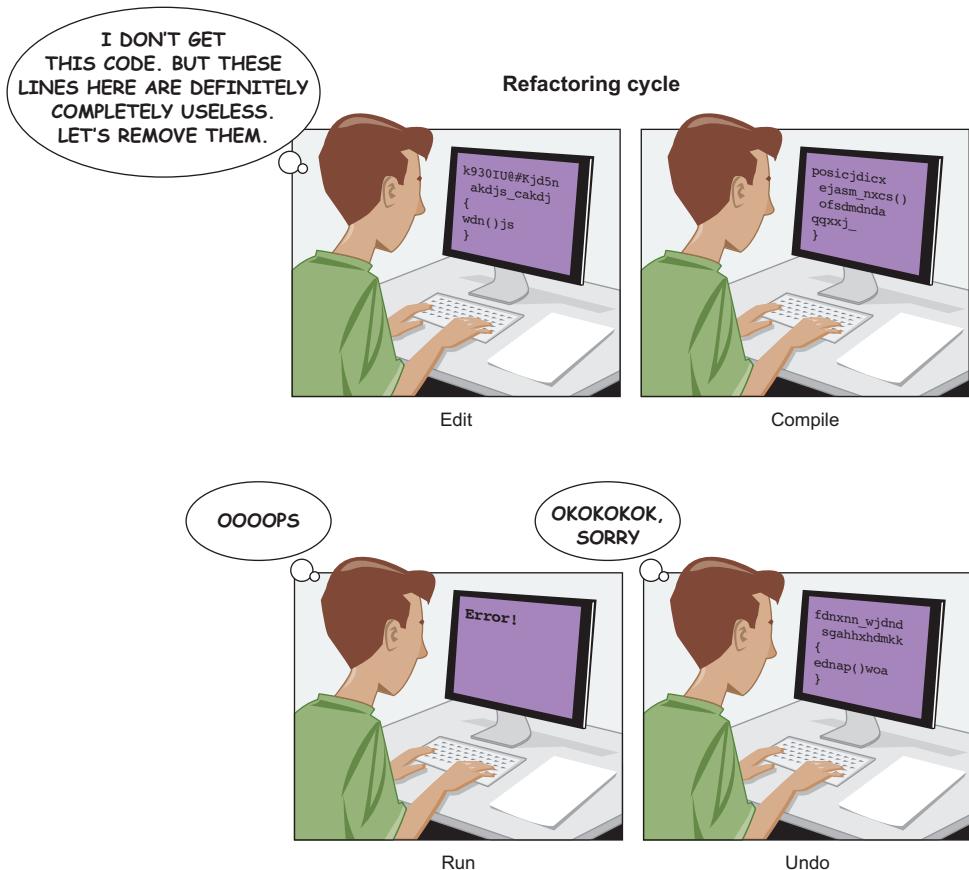


Figure 5.1 Unit tests provide a safety net that gives you the confidence to refactor.

JUnit best practice: Refactoring

Throughout the history of computer science, many great teachers have advocated iterative development. Niklaus Wirth, for example, who gave us the now-ancient languages Algol and Pascal, championed techniques such as stepwise refinement.

(continued)

For a time, these techniques seemed to be difficult to apply to larger, layered applications. Small changes can reverberate throughout a system. Project managers looked to up-front planning as a way to minimize change, but productivity remained low.

The rise of the xUnit framework fueled the popularity of agile methodologies, which (once again) advocate iterative development. Agile methodologists favor writing code in vertical slices to produce a working use case, as opposed to writing code in horizontal slices to provide services layer by layer.

When you design and write code for a single use case or functional chain, your design may be adequate for this feature, but it may not be adequate for the next feature. To retain a design across features, agile methodologies encourage refactoring to adapt the code base as needed.

How do you ensure that *refactoring* (improving the design of existing code) does not break the existing code? The answer is that unit tests tell you when and where code breaks. In short, unit tests give you the confidence to refactor.

Agile methodologies try to lower project risks by enabling you to cope with change. They allow and embrace change by standardizing quick iterations and applying principles such as YAGNI (you ain't gonna need it) and “The Simplest Thing That Could Possibly Work.” The foundation upon which all these principles rest, however, is a solid bed of unit tests.

5.1.5 **Improving implementation**

Unit tests are first-rate clients of the code they test. They force the API under test to be flexible and to be unit testable in isolation. Sometimes you have to refactor your code under test to make it unit testable. You may eventually use the test-driven development (TDD) approach, which by its own conception generates code that can be unit tested. We’ll experiment with TDD in detail in chapter 20.

It is important to monitor your unit tests as you create and modify them. If a unit test is too long and unwieldy, the code under test usually has a design smell, and you should refactor it. You may also be testing too many features in one test method. If a test cannot verify a feature in isolation, the code probably is not flexible enough, and you should refactor it. Modifying code to test it is normal.

5.1.6 **Documenting expected behavior**

Imagine that you need to learn a new API. On the one hand, you have a 300-page document describing the API, and on the other hand, you have some examples of how to use the API. Which would you choose?

The power of examples is well known. Unit tests are examples that show how to use the API. As such, they make excellent developer documentation. Because unit tests match the production code, they *must* be up to date, unlike other forms of documentation.

To examine how tests effectively document the expected behavior, let's go back to the `Calculator` class that was introduced in previous chapters. Listing 5.1 illustrates how unit tests help provide documentation. The `expectIllegalArgument-Exception()` method shows that an `IllegalArgumentException` is thrown when we try to extract the square root from a negative number. The `expectArithmeti-cException()` method shows that an `ArithmetiException` is thrown when we try to divide by zero.

Listing 5.1 Unit tests as automatic documentation

```
public class JUnit5ExceptionTester {
    private Calculator calculator = new Calculator(); ①

    @Test
    public void expectIllegalArgumentException() {
        assertThrows(IllegalArgumentException.class,
                     () -> calculator.sqrt(-1)); ②
    }

    @Test
    public void expectArithmetiException() {
        assertThrows(ArithmetiException.class,
                     () -> calculator.divide(1, 0)); ③
    }
}
```

In this example:

- We initialize an instance of the `Calculator` class whose functionality we are testing ①.
- We assert that the execution of the supplied `calculator.sqrt(-1)` executable throws an `IllegalArgumentException` ②.
- We assert that the execution of the supplied `calculator.divide(1, 0)` executable throws an `ArithmetiException` ③.

The execution of the tests clearly shows the use cases. You can follow these use cases and examine what a particular execution will trigger, so the tests are an effective part of the project documentation.

5.1.7 Enabling code coverage and other metrics

Unit tests tell you, at the push of a button, whether everything still works. Furthermore, unit tests enable you to gather code-coverage metrics (see chapter 6) that show, statement by statement, what code execution the tests triggered and what code the tests did not touch. You can also use tools to track the progress of passing versus failing tests from one build to the next. Further, you can monitor performance and cause a test to fail if its performance has degraded from a previous build.

5.2 Test types

Figure 5.2 outlines four categories of software tests. There are other ways of categorizing software tests—other authors use different names—but these categories are the most useful for the purposes of this book. Please note that this section examines *software tests in general*, not just the automated unit tests covered elsewhere in the book.

In figure 5.2, the outermost tests are broadest in scope. The innermost tests are narrowest in scope. As you move from the inner boxes to the outer boxes, the software tests get more functional, requiring that more of the application be present.

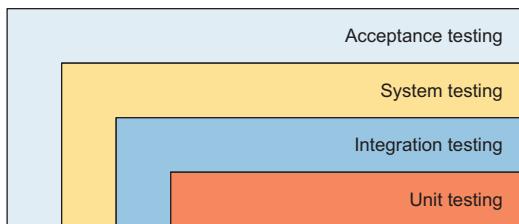


Figure 5.2 The four types of tests, from the innermost (narrowest) to the outermost (broadest)

Earlier, we mentioned that each unit test focuses on a distinct unit of work. What about testing different units of work combined into a workflow? Will the result of the workflow do what you expect?

Different kinds of tests answer these questions. We categorize them in these varieties:

- Unit tests
- Integration tests
- System tests
- Acceptance tests

The following sections look at each test type, starting with the innermost (the narrowest in scope) and working out to the broadest in scope.

5.2.1 Unit testing

Unit testing is a software testing method in which individual units of source code (methods or classes) are tested to determine whether they are fit for use. Unit testing increases developer confidence in changing the code because, from the beginning, it serves as a safety net. If we have good unit tests and run them every time we change the code, we will be certain that our changes are not affecting the existing functionality.

We're focusing here on testing classes and methods in isolation. Suppose that the engineers at our example company, Tested Data Systems Inc., want to manage the company's customers. They will first test the creation of the customer. Then they will test the operations of searching for a customer and adding them to and removing them from the company database. Because a unit test is narrowest in scope, it may only require that some isolated classes be present.

5.2.2 Integration software testing

Individual unit tests are essential quality controls, but what happens when different units of work are combined into a workflow? When you have the tests for a class up and running, the next step is hooking up the class with other methods and services. Examining the interaction among components, possibly running in their target environment, is the job of integration testing. Table 5.1 differentiates the various cases under which components interact.

Table 5.1 Testing how objects, services, and subsystems interact

Interaction	Test description
Objects	The test instantiates objects and calls methods on these objects. You can use this test type to see how objects belonging to different classes cooperate to solve the problem.
Services	The test runs while a container hosts the application, which may connect to a database or attach to any other external resource or device. You can use this test type when you are developing an application that is deployed into a software container.
Subsystems	A layered application may have a frontend to handle the presentation and a backend to execute the business logic. Tests can verify that a request passes through the frontend and returns an appropriate response from the backend. You can use this test type in the case of an application with an architecture made of a presentation layer (web interface, for example) and a business service layer executing the logic.

Just as more traffic collisions occur at intersections, the points where objects interact are major contributors to bugs. Ideally, you should define integration tests before you write the application code. Being able to code to the test strongly increases a programmer's ability to write well-behaved objects. The engineers at Tested Data Systems will use integration testing to check whether the objects representing customers and offers cooperate well, such as when a customer is assigned only once to an offer. When the offer expires, the customer is automatically removed from the offer if they haven't accepted it; if we have added an offer on the customer side, the customer is added on the offer side.

5.2.3 System software testing

System testing of software is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. The objective is to detect inconsistencies among units that are already integrated.

Test doubles or mock objects can simulate the behavior of complex, real objects and therefore are useful when a real object (such as some depended-on component) is impractical to incorporate into a test or when testing is impossible—at least for the moment—because the dependent component is not yet available. *Mock objects* may appear at the level of a unit test: their role is to replace a part of a system that is not available or impractical to incorporate into a test. *Test doubles* are simulated objects that mimic the behavior of real objects in a controlled way. They are created to test the behavior of some other objects that use them.

The engineers at Tested Data Systems will use test doubles when they are communicating with an external service; with an internal service that is not yet available or is slow, hard to configure, or difficult to access; or with a service that is not yet fully available. A test double is handy in that it relieves tests from waiting for the availability of that service.

The customers and offers that the Tested Data Systems engineers are managing, for example, need to be persisted to a database. This database is hard to set up and configure; the engineers must install the software and then create and populate the tables. These processes require time and people. The team will use a mock database for their programs.

5.2.4 Acceptance software testing

It is important that an application perform well, but the application must also meet the customer's needs. Acceptance tests are our final level of testing. The customer or a proxy usually specifies acceptance tests to ensure that the application meets whatever goals the customer or stakeholder has defined.

Acceptance tests are supersets of all other tests. They try to answer essential questions, such as whether the application addresses the business goals and is doing the right thing.

Acceptance tests may be expressed using the keywords Given, When, and Then. When you use these keywords, you are essentially following a scenario: the interaction of the user with the system. The verification in the Then step may look like a unit test, but it checks the end of the scenario and answers the question “Are we addressing the business goals?”

Tested Data Systems may implement acceptance tests such as these:

Given that there is an economy offer,

When we have a regular customer,

Then we can add them to and remove them from the offer.

Given that there is an economy offer,

When we have a VIP customer,

Then we can add them to the offer but not remove them from the offer.

Because acceptance tests are the broadest in scope, they are functional and require a larger part of the application to be present. The acceptance tests we just listed are anticipating the behavior-driven development (BDD) way of working, to be detailed in chapter 21.

5.3 Black-box vs. white-box testing

Before we close this chapter, we'll focus on one other category of software tests: black-box and white-box testing. This category is intuitive and easy to grasp, but developers often forget about it.

5.3.1 Black-box testing

A *black-box test* has no knowledge of the internal state or behavior of the system. The test relies solely on the external system interface to verify its correctness.

As the name of this methodology suggests, you treat the system as a black box. Imagine that the box is covered with buttons and LEDs. You do not know what is inside or how the system operates; you only know that when the correct input is provided, the system produces the desired output. All you need to know to test the system properly is the system's functional specification. The early stages of a project typically produce this kind of specification, which means we can start testing early. Anyone can take part in testing the system, such as a QA engineer, a developer, or even a customer.

The simplest form of black-box testing tries to mimic actions in the user interface manually. A more sophisticated approach is to use a tool for this task, such as HttpUnit, HtmlUnit, or Selenium. We will apply some of these tools in another part of the book (chapter 15).

At Tested Data Systems, black-box testing is used for applications that provide a web interface. The testing tool knows only that it has to interact with the frontend (selections, push-buttons, and so on) and verify the results (the result of the action, the content of the destination page, and so on).

5.3.2 White-box testing

At the other end of the spectrum is *white-box testing*, sometimes called *glass-box testing*. In this type of testing, we use detailed knowledge of the implementation to create tests and drive the testing process. In addition to understanding the component implementation, we need to know how this testing process interacts with other components. For these reasons, the implementers are the best candidates to create white-box tests.

White-box testing can be implemented at an earlier stage; there is no need to wait for the GUI to be available. You may cover many execution paths. Consider the scenarios from section 5.2.4:

Given that there is an economy offer,
When we have a regular customer,
Then we can add them to and remove them from the offer.

Given that there is an economy offer,
When we have a VIP customer,
Then we can add them to the offer but not remove them from the offer.

These scenarios are good examples of white-box testing. They require knowledge of the application internals (at least of the API) and cover different execution paths of which the external user is not aware. Each step corresponds to writing pieces of code that work with the existing API. Developers can apply the code from the early stages without needing a GUI.

5.3.3 Pros and cons

Which of the two approaches should you use? There is no absolute answer, so we suggest that you use both approaches. In some situations, you need user-centric tests (no need for details), and in others, you need to test the implementation details of the system. Tables 5.2 and 5.3 contrast black-box and white-box testing; let's consider the pros and cons of both approaches.

Table 5.2 The pros of black-box and white-box tests

Black-box tests	White-box tests
Tests are user-centric and expose specifications to discrepancies.	Testing can be implemented from the early stages of the project.
The tester may be a nontechnical person.	There is no need for an existing GUI.
Tests can be conducted independently of the developers.	Testing is controlled by the developers and can cover many execution paths.

Table 5.3 The cons of black-box and white-box tests

Black-box tests	White-box tests
A limited number of inputs may be tested.	Testing can be implemented only by skilled people with programming knowledge.
Many program paths may be left uncovered.	Tests will need to be rewritten if the implementation changes.
Tests may become redundant; the lack of details may mean covering the same execution paths.	Tests are tightly coupled to the implementation.

USER-CENTRIC APPROACH

Black-box testing first addresses user needs. We know that there is tremendous value in customer feedback, and one of our goals in extreme programming is to release early and release often. We are unlikely to get useful feedback, however, if we just tell the customer, “Here it is. Let us know what you think.” It is far better to get the customer involved by providing a manual test script to run through. By making the customer think about the application, they can also clarify what the system should do. Interacting with the constructed GUI and comparing the results that are obtained with the expected ones is an example of testing that customers can run by themselves.

TESTING DIFFICULTIES

Black-box tests are more difficult to write and run¹ because they usually deal with a graphical frontend, whether it's a web browser or desktop application. Another issue is that a valid result on the screen does not always mean the application is correct.

¹ Black-box testing is getting easier with tools such as Selenium and HtmlUnit, which we examine in chapter 15.

White-box tests usually are easier to write and run than black-box tests, but the developers must implement them.

TEST COVERAGE

White-box testing provides better test coverage than black-box testing. On the other hand, black-box tests can bring more value than white-box tests. We focus on test coverage in chapter 6.

Although these test distinctions may seem to be academic, recall that “divide and conquer” does not have to apply only to writing production software; it can also apply to testing. We encourage you to use these different types of tests to provide the best code coverage possible, which will give you the confidence to refactor and evolve your applications.

Chapter 6, which starts part 2 of this book, looks at test quality. In it, we present best practices such as measuring test coverage and writing testable code, and we introduce test-driven development, behavior-driven development, and mutation testing.

Summary

This chapter has covered the following:

- Examining the need for unit tests. They offer greater test coverage than functional tests, increase team productivity, detect regressions, give us the confidence to refactor, improve implementation, document expected behavior, and enable code coverage.
- Comparing different types of software testing. From the narrowest to the broadest scope, they are as follows: unit testing, integration testing, system testing, and acceptance testing.
- Contrasting black-box testing and white-box testing. Black-box testing has no knowledge of the internal state or behavior of the system and relies solely on the external system interface to verify its correctness. White-box testing is controlled by the developer, can cover many execution paths, and provides better test coverage.

Part 2

Different testing strategies

T

his part of the book reveals different strategies and techniques in testing. Here, we take a more scientific and theoretical approach to explain the differences. We talk about test quality, describe incorporating mock objects and stubs, and dive into the details of in-container testing.

Chapter 6 is dedicated to the analysis of test quality. It introduces concepts such as code coverage, test-driven development, behavior-driven development, and mutating testing. Chapter 7 is dedicated to stubs, taking a look at a solution to isolate the environment and make tests seamless. Chapter 8 explains mock objects, providing an overview of how to construct and use them. The chapter also provides a real-world example showing not only where mock objects fit best, but also how to benefit by integrating them with JUnit tests.

Chapter 9 describes a different technique: executing tests inside a container. This solution is very different from the preceding ones, and like them, it has pros and cons. The chapter starts by providing an overview of in-container testing. The end of the chapter compares the mocks-stubs approach with the in-container approach.



Test quality

This chapter covers

- Measuring test coverage
- Writing testable code
- Investigating test-driven and behavior-driven development
- Introducing mutation testing
- Testing in the development cycle

I don't think anybody tests enough of anything.

—James Gosling

In the previous chapters, we introduced testing software, began to explore testing with JUnit, and presented different test methodologies. Now that you are writing test cases, it is time to measure how good these tests are by using a test-coverage tool to report what code is executed by the tests and what code is not. This chapter will also discuss how to write code that is easy to test and finish by taking a first look at test-driven development (TDD).

6.1 Measuring test coverage

Writing unit tests gives you the confidence to change and refactor an application. As you make changes, you run tests, which give you immediate feedback about new features under test and whether changes break the existing tests. The issue is that these changes may still break untested functionality.

To resolve this issue, you need to know precisely what code runs when you or the build invoke tests. Ideally, your tests should cover 100% of your application code. This section examines test coverage in detail.

Test coverage can ensure some of the quality of your programming, but it is also a controversial metric. High code coverage does not tell you anything about the quality of the tests. A good programmer should be able to see beyond the pure percentage obtained by running tests.

6.1.1 Introduction to test coverage

Using black-box testing, we can create tests that cover the public API of an application. Because we are using documentation—not knowledge of the implementation—as our guide, we do not create tests that use special parameter values to exercise special conditions in the code, for example.

Many metrics can be used to calculate test coverage. Some of the most basic are the percentage of program methods and the percentage of program lines called during execution of the test suite. Another metric of test coverage is tracking which methods the tests call. The result does not tell you whether the tests are complete, but it does tell you whether you have a test for a method. Figure 6.1 shows the partial test coverage typically achieved by black-box testing alone.

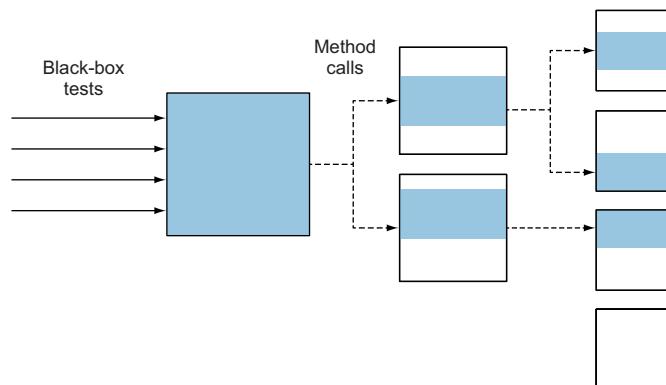


Figure 6.1 Partial test coverage with black-box tests. The boxes represent components or modules. The shaded areas represent the portions of the system effectively covered by tests, and the white areas represent the untested portions.

You *can* write a unit test with intimate knowledge of a method's implementation. If a method contains a conditional branch, you can write two unit tests, one for each branch. Because you need to see into the method to create such a test, this type of test falls in the category of white-box testing. Figure 6.2 shows 100% test coverage with white-box testing.

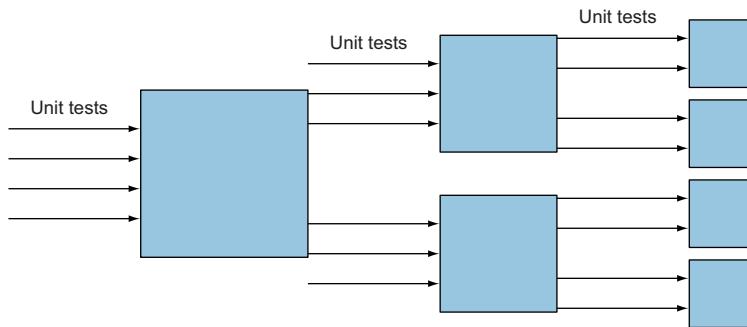


Figure 6.2 Complete test coverage with white-box tests. Each box represents a component or a module. The boxes are fully shaded, as they are fully covered by tests.

You can achieve higher test coverage by using white-box unit tests, because you have access to more methods and can control the inputs to each method and the behavior of secondary objects (using stubs or mock objects, as you'll see in later chapters). Because you can write white-box unit tests against protected, package-private, and public methods, you get more code coverage.

If you haven't been able to achieve high code coverage with black-box testing, you need more tests (you may have uncovered ways to use the application), or you may have superfluous code that does not contribute to the business goals. In either case, an analysis may be required to reveal the real causes.

A program with high test coverage, measured as a percentage, has more of its source code executed during testing, which suggests that it has a lower chance of containing undetected software bugs compared with a program with low test coverage. The metric is built by a tool that runs the test suite and analyzes the code that is effectively executed as a result.

6.1.2 Tools for measuring code coverage

This section demonstrates the use of tools to measure code coverage with the help of the source code provided for this chapter. We'll use the `com.manning.junit-book.ch06` package, which uses the `Calculator` and `CalculatorTest` classes.

A few code-coverage tools are well integrated with JUnit. A very convenient way to determine the code coverage is to work directly from IntelliJ IDEA. The IDE provides an option for executing the tests with coverage, as shown in figure 6.3.

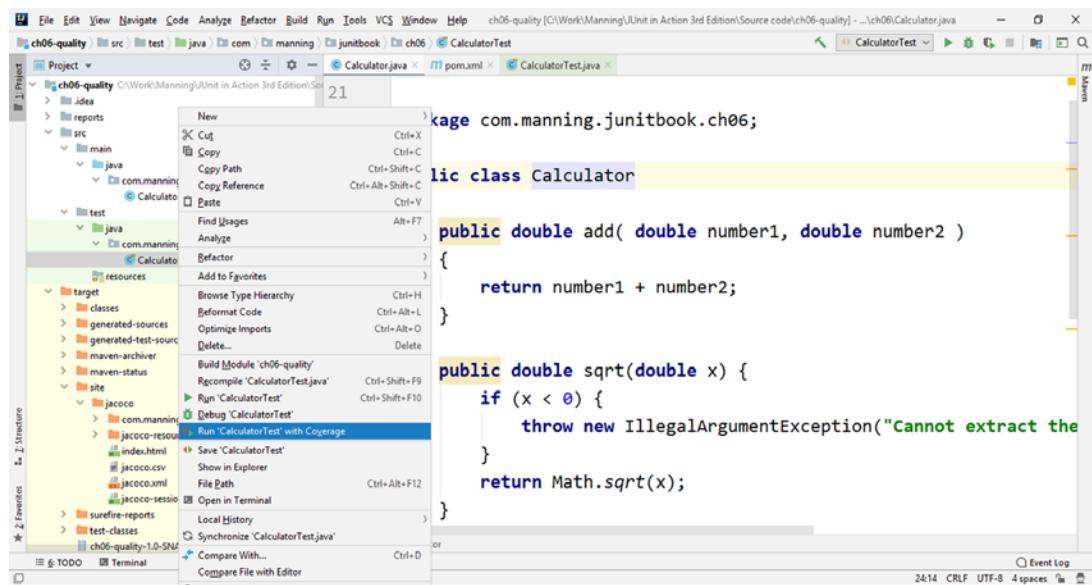


Figure 6.3 Running tests with coverage in IntelliJ

When you run the code with coverage, you get the report shown in figure 6.4. You can click Generate Coverage Report to create reports in HTML format.

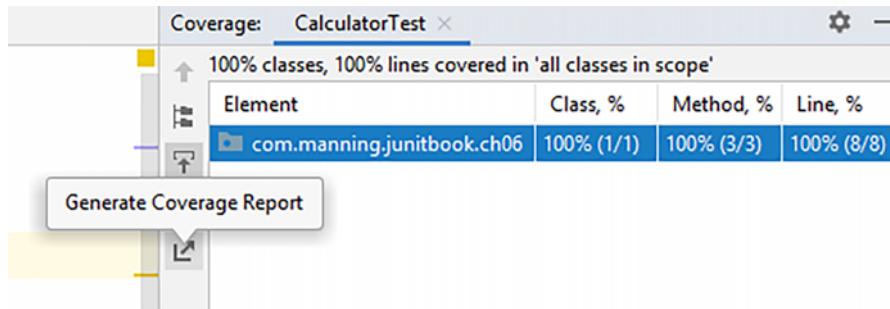


Figure 6.4 The result of running the tests with coverage

The HTML reports can be at the level of the `com.manning.junitbook.ch06` package (figure 6.5), the `Calculator` class (figure 6.6), or individual lines of code (figure 6.7).

[all classes]	
Overall Coverage Summary	
Package	Class, %
all classes	100% (1/ 1)
Coverage Breakdown	
Package ▲	Class, %
com.manning.junitbook.ch06	100% (1/ 1)

Figure 6.5 The code-coverage report at the level of a package

[all classes]	[com.manning.junitbook.ch06]
Coverage Summary for Package: com.manning.junitbook.ch06	
Package	Class, %
com.manning.junitbook.ch06	100% (1/ 1)
Class ▲	
Calculator	Class, %
	100% (1/ 1)

Figure 6.6 The code-coverage report at the level of an individual class

```

44
22 package com.manning.junitbook.ch06;
23
24 public class Calculator
25 {
26     public double add( double number1, double number2 )
27     {
28         return number1 + number2;
29     }
30
31     public double sqrt(double x) {
32         if (x < 0) {
33             throw new IllegalArgumentException("Cannot extract the square root of a negative value");
34         }
35         return Math.sqrt(x);
36     }
37
38     public double divide(double x, double y) {
39         if (y == 0) {
40             throw new ArithmeticException("Cannot divide by zero");
41         }
42         return x/y;
43     }

```

Figure 6.7 The code-coverage report at the level of individual lines

The recommended way to work is to execute the tests from the command line with code coverage; this works fine in conjunction with the continuous integration/continuous development (CI/CD) pipeline. To do so, you can use the JaCoCo Java code coverage



Figure 6.8 The JaCoCo plugin configuration in the Maven pom.xml file

tool. JaCoCo is an open source toolkit that is updated frequently and has very good integration with Maven. To make Maven and JaCoCo work together, you need to insert the JaCoCo plugin information in the pom.xml file (see figure 6.8).

Go to the command prompt, and run mvn test (figure 6.9). This command also generates—for the com.manning.junitbook package and the Calculator class—the code-coverage report, which you can access from the project folder, target\site\jacoco (figure 6.10). We’ve chosen to show an example that does not have 100% code coverage. You can also do this by removing some of the existing tests from the CalculatorTest class in the book’s ch06-quality folder.

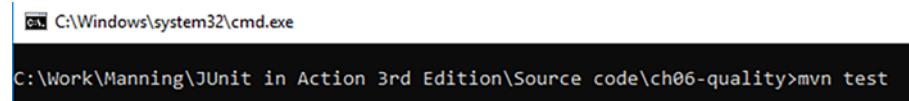


Figure 6.9 Running mvn test from the command prompt

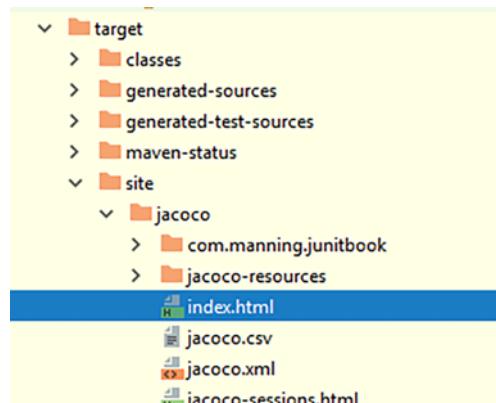


Figure 6.10 The code-coverage report, accessible in the project folder target\site\jacoco

From here, you can access the reports at the level of the `com.manning.junitbook` package (figure 6.11), `Calculator` class (figure 6.12), methods (figure 6.13), and individual lines (figure 6.14).

ch06-quality

Element	Missed Instructions	Cov.	Missed Branches	Cov.
<code>com.manning.junitbook.ch06</code>	84%	75%		
Total	5 of 32	84%	1 of 4	75%

Figure 6.11 JaCoCo report at the package level

[ch06-quality](#) > `com.manning.junitbook.ch06`

com.manning.junitbook.ch06

Element	Missed Instructions	Cov.	Missed Branches	Cov.
<code>Calculator</code>	84%	75%		
Total	5 of 32	84%	1 of 4	75%

Figure 6.12 JaCoCo report at the class level

[ch06-quality](#) > `com.manning.junitbook.ch06` > `Calculator`

Calculator

Element	Missed Instructions	Cov.	Missed Branches	Cov.
<code>sqrt(double)</code>		58%		50%
<code>divide(double, double)</code>		100%		100%
<code>add(double, double)</code>		100%		n/a
<code>Calculator()</code>		100%		n/a
Total	5 of 32	84%	1 of 4	75%

Figure 6.13 JaCoCo report at the method level

```

21.
22. package com.manning.junitbook.ch06;
23.
24. public class Calculator {
25.
26.     public double add( double number1, double number2 ) {
27.         return number1 + number2;
28.     }
29.
30.
31.     public double sqrt(double x) {
32.         if (x < 0) {
33.             throw new IllegalArgumentException("Cannot extract the square root of a negative value");
34.         }
35.         return Math.sqrt(x);
36.     }
37.
38.     public double divide(double x, double y) {
39.         if (y == 0) {
40.             throw new ArithmeticException("Cannot divide by zero");
41.         }
42.         return x/y;
43.     }
44.
45.

```

Figure 6.14 JaCoCo report at the level of individual lines: covered lines are green, uncovered lines are red, and partially covered conditions are yellow.

6.2 Writing testable code

This chapter is dedicated to best practices in software testing. You’re ready to move to the next level: writing code that is easy to test. Sometimes writing a single test case is easy; sometimes it isn’t. Everything depends on the complexity of the application. A best practice is to avoid complexity as much as possible; code should be readable and testable.

In this section, we discuss best practices that can improve your design and code. Remember that it is always easier to write easily testable code than it is to refactor existing code to make it easily testable.

6.2.1 **Understanding that public APIs are contracts**

One principle in providing backward-compatible software states, “Never change the signature of a public method.” An application code review shows that most calls are made from external applications, which play the role of clients of your API. If you change the signature of a public method, you need to change every call site in the application and unit tests. Even with refactoring wizards in tools such as IntelliJ and Eclipse, you must always perform this task with care. During initial development, especially if you’re working with TDD, it is a great time to refactor the API as needed, because you do not yet have any public users. Things will change later!

In the open source world, and for any API made public by a commercial product, life can get even more complicated. Many people use your code, and you should be very careful about the changes you make to stay backward compatible. Public methods become the articulation points of an application between components, open source projects, and commercial products that usually do not know about one another’s existence.

Imagine a public method that takes a distance as a double parameter and uses a black-box test to verify a computation. At some point, the meaning of the parameter changes from miles to kilometers. Your code still compiles, but the runtime breaks. Without a unit test to fail and tell you what is wrong, you may spend a lot of time debugging and talking to angry customers.

Mismatched units are the reason the NASA Mars Climate Observer was lost in 1999; one part of the system calculated the amount of thrust needed to adjust course and reduce speed, and another part used that information to determine how long to fire the thrusters. The problem was that one system used imperial measurements (foot-pounds), and the other used metric measurements (Newtons), and there was no test to catch the mistake! This example illustrates why you must test all public methods. For nonpublic methods, you need to go to a deeper level and use white-box tests.

6.2.2 **Reducing dependencies**

Remember that unit tests verify your code in isolation. Your unit tests should instantiate the class you want to test, use it, and assert its correctness. Your test cases should be simple. What happens when your class instantiates a new set of objects, directly or

indirectly? Now your class depends on those classes. To write testable code, you should reduce dependencies as much as possible. If your classes depend on many other classes that need to be instantiated and set up with some state, your tests will be very complicated; you may need to use a complicated mock-objects solution (see chapter 8).

A solution that reduces dependencies separates methods that instantiate new objects (factories) from methods that provide your application logic. At Tested Data Systems Inc., one of the projects under development is in the automotive field. The project has to manage vehicles and drivers, and there are classes that shape them. Consider the following listing.

Listing 6.1 Reducing dependencies

```
class Vehicle {  
  
    Driver d = new Driver();  
    boolean hasDriver = true;  
  
    private void setHasDriver(boolean hasDriver) {  
        this.hasDriver = hasDriver;  
    }  
}
```

Every time the `Vehicle` class is instantiated with an object, the `Driver` class is also instantiated with an object. The concepts are mixed. The solution is to have the `Driver` instance passed to the `Vehicle` class.

Listing 6.2 Passing the Driver to the Vehicle

```
class Vehicle {  
  
    Driver d;  
    boolean hasDriver = true;  
  
    Vehicle(Driver d) {  
        this.d = d;  
    }  
  
    private void setHasDriver(boolean hasDriver) {  
        this.hasDriver = hasDriver;  
    }  
}
```

This code allows us to produce a mock `Driver` object (see chapter 8) and pass it to the `Vehicle` class on instantiation. This process is called *dependency injection*—a technique in which a dependency is supplied to another object. We can mock any other type of `Driver` implementation. The requirements for the engineers at Tested Data Systems may introduce specific classes such as `JuniorDriver` and `SeniorDriver`, which are passed to the `Vehicle` class. Through dependency injection, the code supplies

the `Driver` object to the `Vehicle` object by invoking the `Vehicle(Driver d)` constructor.

6.2.3 ***Creating simple constructors***

By striving for better test coverage, you add more test cases. In each of these test cases, you

- 1 Instantiate the class to test.
- 2 Set the class to a particular state.
- 3 Do some action.
- 4 Assert the final state of the class.

By doing work in the constructor (other than populating instance variables), listing 6.3 mixes steps 1 and 2. It is a bad practice not only from the design point of view (we do the same work every time we instantiate the class) but also because we get the class in a predefined state. This code is hard to maintain and test. Setting the class to a particular state should be a separate operation, as shown in listings 6.3 and 6.4.

Listing 6.3 Setting the class to a particular state inside the constructor

```
class Car {
    private int maxSpeed;

    Car() {
        this.maxSpeed = 180;
    }
}
```

Listing 6.4 Setting the class to a particular state by using a setter

```
class Car {
    private int maxSpeed;

    public void setMaxSpeed(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }
}
```

6.2.4 ***Following the Law of Demeter (Principle of Least Knowledge)***

The Law of Demeter, or Principle of Least Knowledge, states that one class should know only as much as it needs to know. The Law of Demeter can also be described this way:

Talk to your immediate friends.

Or

Don't talk to strangers.

Consider the following example violation.

Listing 6.5 Law of Demeter violation

```
class Car {  
    private Driver driver;  
  
    Car(Context context) {  
        this.driver = context.getDriver();  
    }  
}
```

In this example, you pass a `Context` object to the `Car` constructor. This object shapes some environmental attributes, such as whether the drive is long-distance, daytime, and/or nighttime. The code violates the Law of Demeter because the `Car` class needs to know that the `Context` object has a `getDriver` method. If you want to test this constructor, you need to get a valid `Context` object before calling the constructor. If the `Context` object has a lot of variables and methods, you could be forced to use mock objects (see chapter 8) to simulate the context.

The proper solution is to apply the Law of Demeter and pass references to methods and constructors only when you need to do so. In this example, you should pass the `Driver` to the `Car` constructor:

```
Car(Driver driver) {  
    this.driver = driver;  
}
```

These concepts are key: require objects, do not search for objects, and ask only for objects that your application requires.

Miško Hevery's society analogy

You can live in a society in which everyone (every class) declares who their friends (collaborators) are. If we know that Joe knows Mary, but neither Mary nor Joe knows Tim, it is safe for us to assume that if we give some information to Joe, he may give it to Mary, but under no circumstances will Tim get hold of it. Now imagine that everyone (every class) declares some of their friends (collaborators), but other friends are kept secret. We are left wondering how in the world Tim got the information we gave to Joe. This is the analogy that Miško Hevery provides on his blog.

Here is the interesting part. If you are the person who built the relationships (code), you know the true dependencies, but anyone who comes after you will be baffled, because the friends that are declared are not the sole friends of objects, and information flows into secret paths that are not clear. The ones coming after you live in a society full of liars.

6.2.5 **Avoiding hidden dependencies and global state**

Be very careful with the global state, because it allows many clients to share the global object. This sharing can have unintended consequences if the global object is not coded for shared access or if clients expect exclusive access to the global object.

At Tested Data Systems, the internal organization needs to install a database that is under the control of the database manager. Some reservations must be created for internal meetings and appointments. Consider the next example.

Listing 6.6 Global state in action

```
public void makeReservation() {
    Reservation reservation = new Reservation();
    reservation.makeReservation();
}

public class Reservation {
    public void makeReservation() {
        manager.initDatabase(); //manager is a reference to a global
                               //DBManager, already initialized
        //require the global DBManager to do more action
    }
}
```

DBManager implies a global state. Without instantiating the database, we will not be able to make a reservation. Internally, Reservation uses DBManager to access the database. Unless it's documented, the Reservation class hides its dependency on the database manager from the programmer, because the API does not provide any clues.

The following listing provides a better implementation. The DBManager dependency is injected into the Reservation object by calling the constructor with an argument.

Listing 6.7 Avoiding global state

```
public void makeReservation() {
    DBManager manager = new DBManager();
    manager.initDatabase();
    Reservation reservation = new Reservation(manager);
    reservation.makeReservation();
}
```

In this example, the Reservation object is constructed with a given database manager. Strictly speaking, the Reservation object should be able to function only if it has been configured with a database manager.

Avoid global state. When you provide access to a global object, you share not only that object, but also any object to which it refers.

6.2.6 Favoring generic methods

Static methods, like factory methods, are very useful, but large groups of static utility methods can introduce issues of their own. Recall that unit testing is testing in isolation. To achieve isolation, you need some articulation points where you can easily substitute your code for the test code. These points use polymorphism. With *polymorphism* (the ability of one object to pass more than one IS-A test), the method you are calling is not determined at compile time. You can easily use polymorphism to substitute application code for test code to force certain code patterns to be tested.

The opposite situation occurs when you use nothing but static methods. Then you practice procedural programming, and all your method calls are determined at compile time. You no longer have articulation points that you can substitute.

Sometimes the harm of static methods to your test is not great, especially when you choose a method that ends execution, such as `Math.sqrt()`. On the other hand, if you choose a method that lies in the heart of your application logic, every method that gets executed inside that static method becomes difficult to test.

Static code and the inability to use polymorphism in your application affect the application and tests equally. No polymorphism means no code reuse for both your application and your tests. This situation can lead to code duplication in the application and tests, which you should try to avoid.

Consequently, static utility methods that operate on parameterized types should be generic. Consider this method, which returns the union of two sets:

```
public static Set union(Set s1, Set s2) {  
    Set result = new HashSet(s1);  
    result.addAll(s2);  
    return result;  
}
```

This method compiles, but with two warnings:

```
Union.java:5: warning: [unchecked] unchecked call to  
HashSet(Collection<? extends E>) as a member of raw type HashSet  
Set result = new HashSet(s1);  
^  
Union.java:6: warning: [unchecked] unchecked call to  
addAll(Collection<? extends E>) as a member of raw type Set  
result.addAll(s2);  
^
```

To make the method typesafe and eliminate the warnings, you can modify it to declare a type parameter representing the element type for the three sets (the two arguments and the return value) and use this type parameter throughout the method. The type parameter list is `<E>`, and the return type is `Set<E>`:

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {  
    Set<E> result = new HashSet<>(s1);  
    result.addAll(s2);  
    return result;  
}
```

This method not only compiles without generating any warnings but also provides type safety, ease of use, and ease of testing.

6.2.7 **Favoring composition over inheritance**

Many people choose inheritance as a code-reuse mechanism. We think composition can be easier to test. At runtime, code cannot change an inheritance hierarchy, but you can compose objects differently. Strive to make your code as flexible as possible at runtime. This way, you can be sure it is easy to switch from one state of your objects to another, which makes the code easy to test.

It is safe to use inheritance within a package, where the subclass and the superclass are under the control of the same programmers. Inheriting from ordinary concrete classes across package boundaries may be risky.

Inheritance is appropriate only when the subclass is a subtype of the superclass. When you need classes A and B, you may ask yourself if you can relate them. Class B should extend class A only if an IS-A relationship exists between the two classes.

You'll also encounter violations of this principle in the Java platform libraries: a stack is not a vector, so `Stack` should not extend `Vector`. Similarly, a property list is not a hash table, so `Properties` should not extend `Hashtable`. In both cases, composition would be preferable.

6.2.8 **Favoring polymorphism over conditionals**

As we mentioned previously, all you do in your tests is

- 1 Instantiate the class to test.
- 2 Set the class to a particular state.
- 3 Assert the final state of the class.

Difficulties may arise at any of these points. Instantiating your class may be difficult if the class is too complex, for example.

One of the main ways to decrease complexity is to try to avoid long multiple-choice `switch` statements or `if` statements. Consider this listing.

Listing 6.8 Example of bad design with conditionals

```
public class DocumentPrinter {  
    [...]  
    public void printDocument() {  
        switch (document.getDocumentType()) {  
            case WORD_DOCUMENT:  
                printWORDDocument();  
                break;  
            case PDF_DOCUMENT:  
                printPDFDocument();  
                break;  
            case TEXT_DOCUMENT:  
                printTextDocument();  
                break;
```

```
    default:
        printBinaryDocument();
        break;
    }
}
[...]
}
```

This implementation is awful for several reasons, and the code is hard to test and maintain. Every time you want to add a new document type, you need additional `case` clauses. If such a situation happens often in your code, you will have to change it in every place that it occurs.

TIP Every time you use a long conditional statement, think of polymorphism. Polymorphism is a natural object-oriented way to avoid long conditionals by breaking a class into several smaller classes. Several smaller components are easier to test than one large, complex component.

In this example, you can avoid the conditional by creating different document types, such as `WordDocument`, `PDFDocument`, and `TextDocument`, each of which implements a `printDocument()` method (listing 6.9). This solution decreases the complexity of the code and makes it easier to read. When the `printDocument(Document)` method of the `DocumentPrinter` class is called, it delegates to `printDocument()` from `Document`. The effective method to be executed is determined at runtime through polymorphism, and the code is easier to understand and test because it is not cluttered with conditionals.

Listing 6.9 Replacing conditionals with polymorphism

```
public class DocumentPrinter {
    [...]
    public void printDocument(Document document) {
        document.printDocument();
    }
}

public abstract class Document {
    [...]
    public abstract void printDocument();
}

public class WordDocument extends Document{
    [...]
    public void printDocument() {
        printWORDDocument();
    }
}

public class PDFDocument extends Document {
    [...]
```

```

    public void printDocument() {
        printPDFDocument();
    }
}

public class TextDocument extends Document {
    [...]
    public void printDocument() {
        printTextDocument();
    }
}

```

6.3 **Test-driven development**

As you design an application, tests may help you improve the initial design. As you write more unit tests, positive reinforcement encourages you to write them earlier. As you design and implement, it becomes natural to wonder how you will test a class. Following this methodology, more developers are making the leap from test-friendly designs to TDD.

DEFINITION *Test-driven development* (TDD)—A programming practice that instructs developers to write tests first and then write the code that makes the software pass those tests. Then the developer examines the code and refactors it to clean up any mess or improve things. The goal of TDD is clean code that works.

6.3.1 **Adapting the development cycle**

When you develop code, you design an application programming interface (API) and then implement the behavior promised by the interface. When you unit test code, you verify the promised behavior through an API. The test is a client of the API, just as your domain code is a client of the API.

The conventional development cycle goes something like this:

`[code, test, (repeat)]`

Developers who practice TDD make a seemingly slight but surprisingly effective adjustment:

`[test, code, (repeat)]`

The test drives the design and becomes the first client of the method, as opposed to software development that allows the addition of software that does not prove to meet requirements.

This approach has a few advantages:

- You write code that is driven by clear goals and can be sure that you address exactly what your application needs to do. Tests represent a means to design the code.

- You can introduce new functionality much faster. Tests drive you to implement code that does what it is supposed to do.
- Tests prevent you from introducing bugs into existing code that is working well.
- Tests serve as documentation, so you can follow them and understand what problems the code is supposed to solve.

6.3.2 Doing the TDD two-step

Earlier, we said that TDD tweaks the development cycle to go something like

```
[test, code, (repeat)]
```

The problem with this process is that it leaves out a key step. Development should go more like this:

```
[test, code, refactor, (repeat)]
```

Refactoring is the process of changing a software system in such a way that it improves the code's internal structure without altering its external behavior. To make sure external behavior is not affected, you need to rely on tests.

The core tenets of TDD are as follows:

- Write a failing test before writing new code.
- Write the smallest piece of code that will make the failing test pass.

Developers who follow this practice have found that test-backed, well-factored code is, by its very nature, easy and safe to change. TDD gives you the confidence to solve today's problems today and tomorrow's problems tomorrow. *Carpe diem!* Chapter 20 is dedicated to using TDD with JUnit 5.

JUnit best practice: Write failing tests first

If you take the TDD development pattern to heart, an interesting thing happens: before you can write any code, *you must write a test that fails*. Why does it fail? *Because you have not written the code to make it succeed.*

Faced with this situation, most of us begin by writing a simple implementation to let the test pass. When the test succeeds, you could stop and move on to the next problem. Being a professional, you would take a few minutes to refactor the implementation to remove redundancy, clarify intent, and optimize the investment in the new code. But as long as the test succeeds, technically, you're done.

If you always test first, you will never write a line of new code without a failing test.

6.4 Behavior-driven development

Behavior-driven development (BDD), which was originated by Dan North in the mid-2000s, is a methodology for developing IT solutions that satisfy business requirements directly. Its philosophy is driven by business strategy, requirements, and goals, which

are refined and transformed into an IT solution. Whereas TDD helps you build good-quality software, BDD helps you build software that's worth building to solve the problems of the users.

BDD helps you write software in a way that lets you discover and focus your efforts on what really matters. You find out what features will benefit the organization and effective ways to implement them. You see beyond what the user asks for and build what the user actually needs.

A natural question that arises is, “What gives business value to the software?” The answer is that working features provide business value to the software. A *feature* is a tangible, deliverable piece of functionality that helps the business achieve its business goals. Also, software that can be easily maintained and enhanced is more valuable than software that is hard to change without breaking, so following a practice like BDD or TDD adds value.

To address business goals, a business analyst works with the customer to decide what software features achieve these goals. These features are high-level requirements, such as “Provide a way for a customer to choose among a few alternative routes to the destination” and “Provide a way for a customer to choose the optimal route to the destination.” These features then need to be broken into *stories*. The stories might be, “Find the route between source and destination with the smallest number of flight changes” and “Find the quickest route between source and destination.” Stories need to be described in terms of concrete examples, which become the acceptance criteria for the story.

Acceptance tests (introduced in chapter 5) are expressed BDD style in a way that can be automated later. The keywords are Given, When, and Then. Following is an example of acceptance criteria:

Given the flights operated by company X,

When we want to find the quickest route from Bucharest to New York from May 15 to 20,

Then we will be provided the route Bucharest–Frankfurt–New York.

Chapter 21 discusses BDD with JUnit 5.

6.5 **Mutation testing**

If you strive for testing at different levels (from unit testing to acceptance testing) and obtain high code coverage (as close to 100% as possible), how can you be sure that your code is close to perfection? The bad news is that perfection may not be enough! Even 100% code coverage does not mean your code works perfectly—your tests may not be good enough. The simplest way for this to happen is to omit assertions in the tests; for example, the output may be too complex for the test to verify, so we display it or log it and let a person decide what to do next.

How can you check the quality of the tests and make sure they do what they are supposed to do? Mutation testing comes into play in this situation. *Mutation testing* (or *mutation analysis* or *program mutation*) is used to design new software tests and evaluate the quality of existing software tests.

The basic idea of mutation testing involves modifying a program in small ways. Each mutated version is called a *mutant*. The behavior of the original version differs from that of the mutant. Tests detect and reject mutants, which is called *killing the mutant*. Test suites are measured by the percentage of mutants that they kill. New tests can be designed to kill additional mutants.

Mutants are generated by well-defined mutation operators that exchange an existing operator for another one or invert some conditions. The goal is to support the creation of effective tests or to locate weaknesses in the test data used for the program or sections of the code that are seldom or never accessed during execution. Consequently, mutation testing is a form of white-box testing.

Given the following piece of code,

```
if (a) {  
    b = 1;  
} else {  
    b = 2;  
}
```

the mutation operation may reverse the condition so that the newly tested piece of code is

```
if (!a) {  
    b = 1;  
} else {  
    b = 2;  
}
```

A strong mutation test fulfills the following conditions:

- 1 The test reaches the `if` condition that has been mutated.
- 2 The test continues on a different branch from the initial correct one.
- 3 The changed value of `b` propagates to the output of the program and is checked by the test.
- 4 The test will fail because the method returned the wrong value for `b`.

Well-written tests must determine the failure of mutated tests to demonstrate that they initially covered the necessary logical conditions.

The best-known Java mutation testing framework is Pitest (<https://pitest.org>), which changes the Java bytecode to generate mutants. Mutation testing exceeds the scope of this book, but we wanted to introduce its primary ideas in the context of discussing test quality.

6.6 Testing in the development cycle

Testing occurs at different places and times during the development cycle. This section first introduces a development life cycle and then uses it as a basis for deciding what types of tests are executed when. Figure 6.15 shows a typical development cycle.

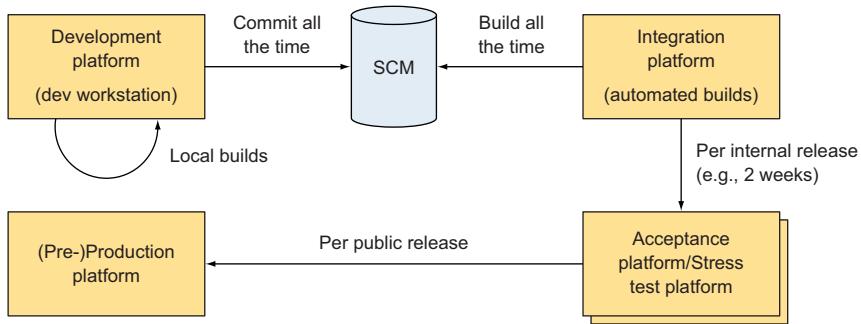


Figure 6.15 A typical application development life cycle, using the continuous integration principle

The life cycle is divided into the following platforms:

- *Development*—This coding happens on developers' workstations. One important rule is to *commit* (check in) to your source control management (SCM) system (Git, SVN, CVS, ClearCase, and so on) several times per day. After you commit, others can begin using your work. It is important to commit only something that works. To ensure this, you should first merge your changes with the current code in the repo. Then you should run a local build with Maven or Gradle. You can also watch the results of an automated build based on the latest changes to the SCM repository.
- *Integration*—This platform builds the application from its components (which may have been developed by different teams) and ensures that the components work together. This step is extremely valuable because problems are often discovered here. The step is so valuable, in fact, that you should automate it. After automation, the process is called *continuous integration* (see www.martinfowler.com/articles/continuousIntegration.html). You can achieve continuous integration by building the application automatically as part of the build process (see chapter 13).
- *Acceptance/stress test*—Depending on the resources available to your project, this can be one or two platforms. The stress-test platform exercises the application under load and verifies that it scales correctly (with respect to size and response time). The acceptance platform is where the project's customers accept (sign off on) the system. It is highly recommended that the system be deployed on the acceptance platform as often as possible to get user feedback.
- *(Pre)production*—The preproduction platform is the last staging area before production. This platform is optional, and small or noncritical projects can do without it.

Next, we discuss how testing fits into the development cycle. Figure 6.16 highlights the types of tests you can perform on each platform.

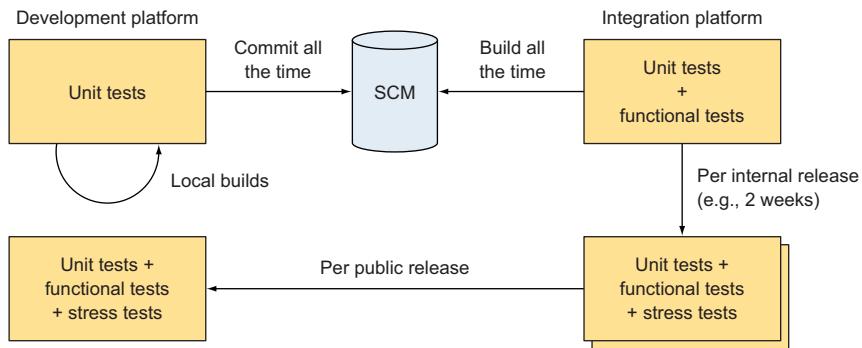


Figure 6.16 The different types of tests performed on each platform of the development cycle

On the *development platform*, you execute *logic unit tests* (tests that can be executed in isolation from the environment). These tests execute very quickly, and you usually execute them from your IDE to verify that any change you have made in the code has not broken anything. These tests are also executed by your automated build before you commit the code to your SCM. You could also execute integration tests, but they often take much longer because they need some part of the environment to be set up (database, application server, and so on). In practice, you execute only a subset of all integration tests, including any new integration tests that you have written.

The *integration platform* usually runs the build process automatically to package and deploy the application; then it executes unit and functional tests. *Functional testing* means evaluating the compliance of a system or component with the requirements. This black-box testing type usually describes what the system does. Generally, only a subset of all functional tests is run on the integration platform, because compared with the target production platform, integration is a simple platform that lacks elements. (It may be missing a connection to an external system being accessed, for example.) All tests are executed on the integration platform. Time is less important, and the whole build can take several hours with no effect on development.

On the *acceptance platform/stress test platform*, you execute the same tests executed by the integration platform; in addition, you run stress tests (to verify the robustness and performance of the software). The acceptance platform is extremely close to the production platform, and more functional tests can be executed here.

A good habit is to also run on the *(pre)production platform* the tests you ran on the acceptance platform. This technique acts as a sanity check, verifying that everything is set up correctly.

Human beings are strange creatures that tend to neglect details. In a perfect world, we would have all four platforms to run our tests on. In the real world, however, most software companies try to skip some of the platforms listed here, or the concept of testing as a whole. As a developer who bought this book, you've already made the right decision: more tests and less debugging.

Now, again, it is up to you. Are you going to strive for perfection, stick to everything that you've learned so far, and let your code benefit from that knowledge?

JUnit best practice: Continuous regression testing

Most tests are written for the here and now. You write a new feature, and you write a new test. You see whether the feature plays well with others and whether the users like it. If everyone is happy, you can lock the feature and move to the next item on your list. Most software is written in a progressive fashion: you add first one feature and then another.

Most often, each new feature is built on a path paved by existing features. If an existing method can service a new feature, you reuse the method and save the cost of writing a new one. The process is never quite that easy, of course. Sometimes you need to change an existing method to make it work with a new feature. In this case, you need to confirm that the old features still work with the amended method.

A strong benefit of JUnit is that it makes test cases easy to automate. When a change is made to a method, you can run the test for that method. If that test passes, you can run the rest. If any test fails, you can change the code (or the tests) until all tests pass again.

Using old tests to guard against new changes is a form of regression testing. Any kind of test can be used as a regression test, but running unit tests after every change is your first, best line of defense.

The best way to ensure that regression testing takes place is to automate your test suites.

Chapter 7 discusses test granularity and introduces *stubs*: programs that simulate the behavior of software components that a module under test depends on.

Summary

This chapter has covered the following:

- Techniques in unit testing that check the quality of the tests
- The concept of code coverage and tools that inspect it: you can do this from IntelliJ IDEA or with JaCoCo
- Designing improvements of the code to make it easily testable
- Investigating test-driven development (TDD) and behavior-driven development (BDD) as software development processes focused on quality
- Introduction to mutation testing, a technique to evaluate the quality of existing software tests by modifying programs in small ways
- Examining how testing in the development cycle using the continuous integration principle can start with coding (conventional development cycle) or with testing (TDD)



Coarse-grained testing with stubs

This chapter covers

- Testing with stubs
- Using an embedded server in place of a real web server
- Implementing unit tests of an HTTP connection with stubs

And yet it moves.

—Galileo Galilei

As you develop your applications, you will find that the code you want to test depends on other classes, which themselves depend on other classes, which then depend on the environment (figure 7.1). You might be developing an application that uses Hibernate to access a database, a Java EE application (one that relies on a Java EE container for security, persistence, and other services), an application that accesses a filesystem, or an application that connects to some resource by using HTTP or another protocol.



Figure 7.1 The application depends on other classes, which in turn depend on other classes, which then depend on the environment.

Starting in this chapter, we will look at using JUnit 5 to test an application that depends on external resources. We will include HTTP servers, database servers, and physical devices.

For applications that depend on a specific runtime environment, writing unit tests is a challenge. Your tests need to be stable, and when they run repeatedly, they need to yield the same results. You need a way to control the environment in which the tests run. One solution is to set up the real required environment as part of the tests and run the tests from within that environment. In some cases, this approach is practical and brings real benefits. (See chapter 9, which discusses in-container testing.) It works well only if you can set up the real environment on your development and build platforms, however, which is not always feasible.

If your application uses HTTP to connect to a web server provided by another company, for example, you usually will not have that server application available in your development environment. Therefore, you need a way to simulate that server so that you can still write and run tests for your code.

Alternatively, suppose that you are working with other developers on a project. What if you want to test your part of the application, but the other part is not ready? One solution is to simulate the missing part by replacing it with a fake that behaves similarly. There are two strategies for providing these fake objects: stubbing and using mock objects.

When you write stubs, you provide a predetermined behavior right from the beginning. The stubbed code is written outside the test, and it will always have a fixed behavior, no matter how many times or where you use the stub; its methods will usually return hardcoded values. The pattern of testing with a stub this: *initialize stub > execute test > verify assertions*.

A mock object does not have a predetermined behavior. When running a test, you are setting the expectations on the mock before effectively using it. You can run different tests, and you can reinitialize a mock and set different expectations on it. The pattern of testing with a mock object this: *initialize mock > set expectations > execute test > verify assertions*. This chapter is dedicated to stubbing; chapter 8 covers mock objects.

7.1 Introducing stubs

Stubs are mechanisms for faking the behavior of real code or code that is not ready yet. In other words, stubs allow you to test a portion of a system when the other part is not available. Stubs usually do not change the code you are testing; instead, they adapt to provide seamless integration.

DEFINITION *Stub*—A piece of code that is inserted at runtime in place of the real code to isolate the caller from the real implementation. The intent is to replace a complex behavior with a simpler one that allows independent testing of some part of the real code.

Here are some examples of when you might use stubs:

- You cannot modify an existing system because it is too complicated and fragile.
- You depend on an environment that you cannot control.
- You are replacing a full-blown external system such as a filesystem, a connection to a server, or a database.
- You are performing coarse-grained testing, such as integration testing between different subsystems.

Using stubs is not recommended in situations such as these:

- You need fine-grained tests to provide precise messages that underline the cause of the failure.
- You would like to test a small portion of the code in isolation.

In these situations, you should use mock objects (discussed in chapter 8).

Stubs usually provide very good confidence in the tested system. With stubs, you are not modifying the objects under test, and what you are testing is the same as what will execute in production. An automated build or a developer usually executes tests involving stubs in their running environment, providing additional confidence.

On the downside, stubs are usually hard to write, especially when the system to fake is complex. The stub needs to implement, in a brief, simplified way, the same logic as the code it is replacing, which is difficult to get right for complex logic. Here are some cons of stubbing:

- Stubs are often complex to write and need debugging themselves.
- Stubs can be difficult to maintain because they are complex.
- A stub does not lend itself well to fine-grained unit testing.
- Each situation requires a different stubbing strategy.

In general, stubs are better adapted than mocks for replacing coarse-grained portions of code.

7.2 **Stubbing an HTTP connection**

To demonstrate what stubs can do, we'll explain how the engineers at Tested Data Systems Inc. build stubs for an application that opens an HTTP connection to a URL and reads its content. The sample application (limited to a `WebClient.getContent` method) opens an HTTP connection to a remote web resource. The remote web resource is a servlet, which generates an HTML response. The web resource in figure 7.2 is what we call the *real code* in the stub definition.

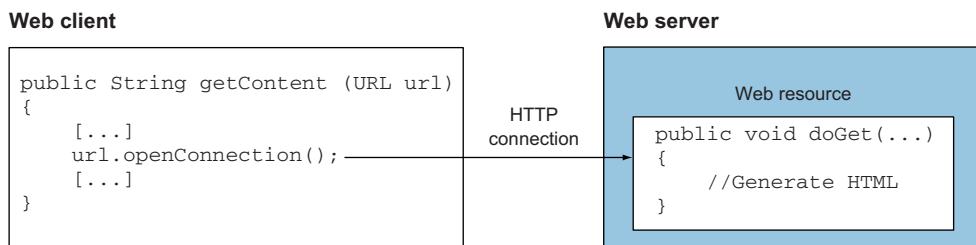


Figure 7.2 The sample application opens an HTTP connection to a remote web resource. The web resource is the real code in the stub definition.

The goal of the engineers at Tested Data Systems is to unit test the `getContent` method by stubbing the remote web resource, as demonstrated in figure 7.3. The remote web resource is not yet available, and the engineers need to progress with their part of the work in the absence of that resource. They replace the servlet web resource with the stub—a simple HTML page that returns whatever they need for the `TestWebClient` test case. This approach allows them to test the `getContent` method independently of the implementation of the web resource (which in turn could call several other objects down the execution chain, possibly down to a database).

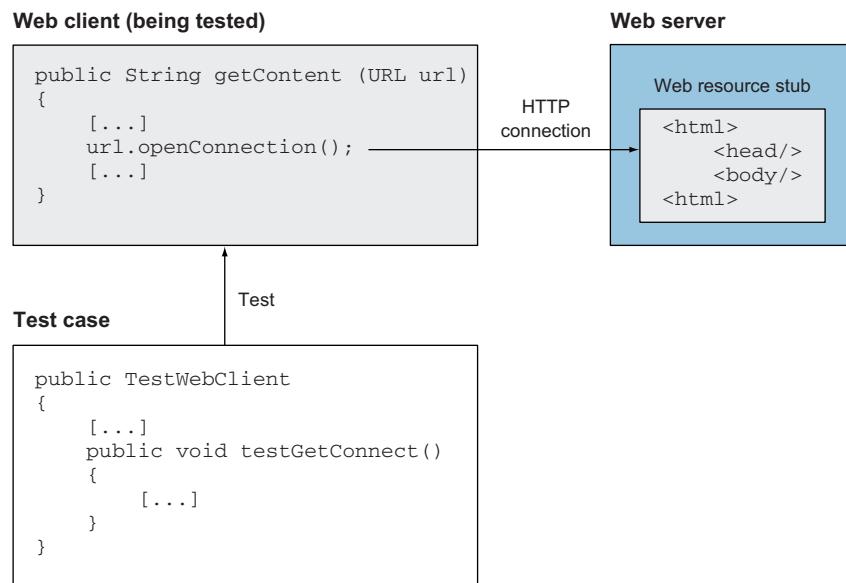


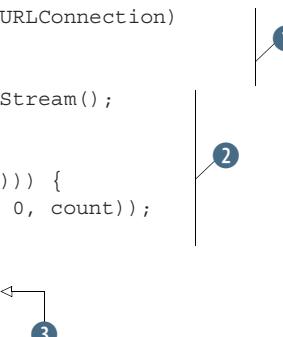
Figure 7.3 Adding a test case and replacing the real web resource with a stub

The important point to notice with stubbing is that `getContent` is not modified to accept the stub. The change is transparent to the application under test. For stubbing to be possible, the target code needs to have a well-defined interface and allow different implementations (a stub, in this case) to be plugged in. The interface is actually the public abstract class `java.net.URLConnection`, which cleanly isolates the implementation of the page from its caller.

Following is an example of a stub in action, using the simple HTTP connection example. The listing demonstrates a code snippet that opens an HTTP connection to a given URL and reads the content at that URL. The method is one part of a bigger application to be unit tested.

Listing 7.1 Sample method opening an HTTP connection

```
[...]  
import java.net.URL;  
import java.net.HttpURLConnection;  
import java.io.InputStream;  
import java.io.IOException;  
  
public class WebClient {  
    public String getContent(URL url) {  
        StringBuffer content = new StringBuffer();  
        try {  
            HttpURLConnection connection = (HttpURLConnection)  
                url.openConnection();  
            connection.setDoInput(true);  
            InputStream is = connection.getInputStream();  
            byte[] buffer = new byte[2048];  
            int count;  
            while (-1 != (count = is.read(buffer))) {  
                content.append(new String(buffer, 0, count));  
            }  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
        return content.toString();  
    }  
}
```



In this listing:

- We start by opening an HTTP connection using the `HttpURLConnection` class 1.
- We read the stream content until there is nothing more to read 2.
- If an error occurs, we pack it into a `RuntimeException` and rethrow it 3.

7.2.1 Choosing a stubbing solution

The example application has two possible scenarios: the remote web server (see figure 7.1) could be located outside the development platform (such as on a partner site), or it could be part of the platform where the application is to be deployed. In both cases, we need to introduce a server into the development platform to be able to unit test the `WebClient` class. One relatively easy solution would be to install an Apache test server and drop some test web pages in its document root. This stubbing solution is typical and widely used, but it has several drawbacks, as shown in table 7.1.

Table 7.1 Drawbacks of the chosen stubbing solution

Drawback	Explanation
Reliance on the environment	We need to be sure that the full environment is up and running before the test starts. If the web server is down, and we execute the test, it will fail, and we will spend time determining why it is failing. We will discover that the code is working fine and the problem was only a setup issue generating a false failure. When we are unit testing, it is important to be able to control as much as possible of the environment in which the tests execute, such that test results are reproducible.
Separated test logic	The test logic is scattered in two locations: in the JUnit 5 test case and on the test web page. We need to keep both types of resources in sync for the tests to succeed.
Difficult tests to automate	Automating the execution of the tests is difficult because it involves deploying the web pages on the web server, starting the web server, and running the unit tests.

Fortunately, an easier solution exists: using an embedded web server. Because the testing is in Java, the easiest solution is to use a Java web server that can be embedded in the test case. You can use the free, open source Jetty server for this purpose. The developers at Tested Data Systems will use Jetty to set up their stubs. (For more information about Jetty, visit www.eclipse.org/jetty.)

7.2.2 Using Jetty as an embedded server

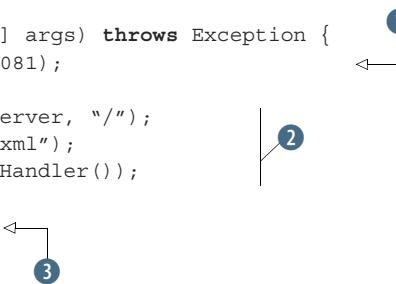
We use Jetty because it is fast (important when running tests), it is lightweight, and the test cases can control it programmatically. Additionally, Jetty is a very good web, servlet, and JSP container that you can use in production.

Using Jetty allows us to eliminate the drawbacks outlined previously: the JUnit 5 test case starts the server, we write the tests in Java in one location, and automating the test suite becomes a nonissue. Thanks to Jetty modularity, the real task facing the developers at Tested Data Systems is stubbing the Jetty handlers, not the whole server from the ground up.

To understand how Jetty works, let's implement and examine an example of setting up and controlling it from the tests. The following listing demonstrates how to start Jetty from Java and how to define a document root (/) from which to start serving files.

Listing 7.2 Starting Jetty in embedded mode: JettySample class

```
[...]  
import org.mortbay.jetty.Server;  
import org.mortbay.jetty.handler.ResourceHandler;  
import org.mortbay.jetty.servlet.Context;  
  
public class JettySample {  
    public static void main(String[] args) throws Exception {  
        Server server = new Server(8081);  
  
        Context root = new Context(server, "/");  
        root.setResourceBase("./pom.xml");  
        root.setHandler(new ResourceHandler());  
  
        server.start();  
    }  
}
```



In this listing:

- We start by creating the Jetty Server object ① and specifying in the constructor which port to listen to for HTTP requests (port 8081). Be sure to check that port 8081 is not already in use—you may change it in the source files, if necessary.
- Next, we create a Context object ② that processes the HTTP requests and passes them to various handlers. We map the context to the already created server instance, and to the root (/) URL. The setResourceBase method sets the document root from which to serve resources. On the next line, we attach a ResourceHandler to the root to serve files from the filesystem.
- Finally, we start the server ③.

If you start the program from listing 7.2 and navigate your browser to <http://localhost:8081>, you should be able to see the content of the pom.xml file (figure 7.4). You get a similar effect by changing the code to set the base as `root.setResourceBase(" . ")`, restarting the server, and then navigating to <http://localhost:8081/pom.xml>.

Figure 7.4 demonstrates the results of running the code in listing 7.2 after opening a browser on <http://localhost:8081>. Now that you have seen how to run Jetty as an embedded server, the next section shows you how to stub the server resources.

```

- <!--

Licensed to the Apache Software Foundation (ASF) under one or more
contributor license agreements. See the NOTICE file distributed with
this work for additional information regarding copyright ownership.
The ASF licenses this file to you under the Apache License, Version
2.0 (the "License"); you may not use this file except in compliance
with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0 Unless required by
applicable law or agreed to in writing, software distributed under the
License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for
the specific language governing permissions and limitations under the
License.

-->
-<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.manning.junitbook</groupId>
  <artifactId>ch07-stubs</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>ch07-stub</name>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>

```

Figure 7.4 Testing the JettySample in a browser. It displays the content of the pom.xml file to demonstrate how the Jetty web server works.

7.3 Stubbing the web server resources

This section focuses on the HTTP connection unit test. The developers at Tested Data Systems will write tests to verify that they can call a valid URL and get its content. These tests are the first steps in checking the functionality of web applications that are interacting with external customers.

7.3.1 Setting up the first stub test

To verify that the WebClient works with a valid URL, we need to start the Jetty server before the test, which we can implement in a test case `setUp` method. We can also stop the server in a `tearDown` method.

Listing 7.3 Skeleton of the first test to verify that WebClient works with a valid URL

```

[...]
import java.net.URL;
import org.junit.jupiter.api.*;

public class TestWebClientSkeleton {

    @BeforeAll
    public static void setUp() {
        // Start Jetty and configure it to return "It works" when
        // the http://localhost:8081/testGetContentOk URL is
        // called.
    }
}

```

```

@AfterAll
public static void tearDown() {
    // Stop Jetty.
}

@Test
@Disabled(value = "This is just the initial skeleton of a test.
    Therefore, if we run it now, it will fail.")
public void testGetContentOk() throws MalformedURLException {
    WebClient client = new WebClient();
    String workingContent = client.getContent(new URL(
        "http://localhost:8081/testGetContentOk"));
    assertEquals ("It works", workingContent);
}
}

```

To implement the `@BeforeAll` and `@AfterAll` methods, we have two options. We can prepare a static page containing the text "It works", which we put in the document root (controlled by the call to `root.setResourceBase(String)` in listing 7.2). Alternatively, we can configure Jetty to use a custom handler that returns the string "It works" instead of getting it from a file. This technique is much more powerful because it lets us unit test the case when the remote HTTP server returns an error code to the `WebClient` client application.

CREATING A JETTY HANDLER

The next listing creates a Jetty Handler that returns the string "It works".

Listing 7.4 Creating a Jetty Handler that returns "It works"

```

private static class TestGetContentOkHandler extends AbstractHandler {
    @Override
    public void handle(String target, HttpServletRequest request,
        HttpServletResponse response, int dispatch) throws IOException {
        OutputStream out = response.getOutputStream();
        ByteArrayISO8859Writer writer = new ByteArrayISO8859Writer();
        writer.write("It works");
        writer.flush();
        response.setIntHeader(HttpHeaders.CONTENT_LENGTH, writer.size());
        writer.writeTo(out);
        out.flush();
    }
}

```



In this listing:

- The class creates a handler ① by extending the Jetty `AbstractHandler` class and implementing a single method: `handle`.
- Jetty calls the `handle` method to forward an incoming request to our handler. After that, we use the Jetty `ByteArrayISO8859Writer` class ② to send back the string "It works", which we write in the HTTP response ③.

- The last step is setting the response content length to the length of the string written to the output stream (required by Jetty) and then sending the response [4](#).

Now that this handler is written, we can tell Jetty to use it by calling `context.setHandler(new TestGetContentOkHandler())`.

WRITING THE TEST CLASS

We are almost ready to run the test that is the basis of verifying the functionality of the web applications interacting with the external customers of Tested Data Systems, as shown in the following listing.

Listing 7.5 Putting it all together

```
[...]
import java.net.URL;
[...]

public class TestWebClient {
    private WebClient client = new WebClient();

    @BeforeAll
    public static void setUp() throws Exception {
        Server server = new Server(8081);

        Context contentOkContext = new Context(server, "/testGetContentOk");
        contentOkContext.setHandler(new TestGetContentOkHandler());

        server.setStopAtShutDown(true);
        server.start();
    }

    @AfterAll
    public static void tearDown() {
        // Empty
    }

    @Test
    public void testGetContentOk() throws Exception {
        String workingContent = client.getContent(new URL(
            "http://localhost:8081/testGetContentOk"));
        assertEquals("It works", workingContent);
    }

    private static class TestGetContentOkHandler extends AbstractHandler {
        //Listing 7.4 here.
    }
}
```

The test class has become quite simple. The `@BeforeAll` `setUp` method constructs the `Server` object the same way as in listing 7.2. Then we have our `@Test` method. We leave our `@AfterAll` method empty intentionally because we programmed the server to stop at shutdown; the server instance is explicitly stopped when the JVM is shut down. If you run the test, you see the result in figure 7.5. The test passes.

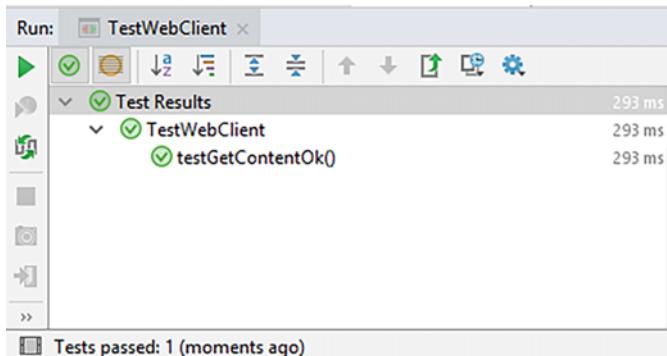


Figure 7.5 Result of the first working test using a Jetty stub. JUnit 5 starts the server before the first test, and the server shuts itself down after the last test.

7.3.2 Reviewing the first stub test

You have been able to fully unit test the `getContent` method in isolation by stubbing the web resource. What have you really tested? What kind of test have you achieved? You have done something quite powerful: unit tested the method and, at the same time, executed an integration test. In addition, you have tested not only the code logic but also the connection part that is outside the code (through the Java `HttpURLConnection` class).

The drawback of this approach is that it is complicated. At Tested Data Systems it can take a Jetty novice half a day to learn enough about Jetty to set it up correctly. In some instances, the novice will have to debug stubs to get them to work correctly. Keep in mind that the stub must stay simple, not become a full-fledged application that requires tests and maintenance. If you spend too much time debugging the stubs, a different solution may be called for.

In these examples, we need a web server—but another example and stub will be different and need a different setup. Experience helps, but different cases usually require different stubbing solutions.

The example tests the web application developed at Tested Data Systems, which is nice because it allows us to unit test the code and to perform some integration tests at the same time. This functionality, however, comes at the cost of complexity. More solutions that are lightweight focus on unit testing the code without performing integration tests. The rationale is that although we need integration tests, they could run in a separate test suite or as part of functional tests.

The next section looks at another solution that still qualifies as stubbing and is simpler in the sense that it does not require stubbing a whole web server. This solution brings us one step closer to the mock-object strategy, which is described in chapter 8.

7.4 Stubbing the connection

So far, we have stubbed the web server resources. Next, we will stub the HTTP connection instead. Doing so will prevent us from effectively testing the connection, which is fine because that is not the real goal at this point. We want to test the code in isolation. Functional or integration tests will test the connection at a later stage.

When it comes to stubbing the connection without changing the code, we benefit from the Java `URL` and `HttpURLConnection` classes, which let us plug in custom protocol handlers to process any kind of communication protocol. We can have any call to the `HttpURLConnection` class redirected to our own class, which will return whatever we need for the test.

7.4.1 Producing a custom URL protocol handler

To implement a custom URL protocol handler, we need to call the `URL` static method `setURLStreamHandlerFactory` and pass it a custom `URLStreamHandlerFactory`. The engineers at Tested Data Systems are using this approach to create their stub implementation of the URL stream handler. Whenever the `URL` `openConnection` method is called, the `URLStreamHandlerFactory` class is called to return a `URLStreamHandler`. The next listing shows the code that performs this action. The idea is to call `setURLStreamHandlerFactory` in the JUnit 5 `setUp` method.

Listing 7.6 Providing custom stream handler classes for testing

```
[...]
import java.net.URL;
import java.net.URLStreamHandlerFactory;
import java.net.URLStreamHandler;
import java.net.URLConnection;
import java.net.MalformedURLException;

public class TestWebClient1 {

    @BeforeAll
    public static void setUp() {
        URL.setURLStreamHandlerFactory(new StubStreamHandlerFactory()); 1
    }

    private static class StubStreamHandlerFactory implements
        URLStreamHandlerFactory {
        @Override
        public URLStreamHandler createURLStreamHandler(String protocol) {
            return new StubHttpURLStreamHandler(); 2
        }
    }
}
```



```

private static class StubHttpURLStreamHandler
    extends URLStreamHandler {
    @Override
    protected URLConnection openConnection(URL url)
        throws IOException {
        return new StubHttpURLConnection(url);
    }
}
@Test
public void testGetContentOk() throws MalformedURLException {
    WebClient client = new WebClient();
    String workingContent = client.getContent(
        new URL("http://localhost"));
    assertEquals("It works", workingContent);
}
}

```

3

In this listing:

- We start by calling `setURLStreamHandlerFactory` ① with our first stub class, `StubStreamHandlerFactory`.
- We use several (inner) classes ② and ③ to use the `StubHttpURLConnection` class.
- In `StubStreamHandlerFactory`, we override the `createURLStreamHandler` method ②, in which we return a new instance of our second private stub class, `StubHttpURLStreamHandler`.
- In `StubHttpURLStreamHandler`, we override one method, `openConnection`, to open a connection to the given URL ③.

Note that we have not written the `StubHttpURLConnection` class yet. That class is the topic of the next section.

7.4.2 Creating a JDK `HttpURLConnection` stub

The last step is creating a stub implementation of the `HttpURLConnection` class so that we can return any value we want for the test. This simple implementation returns the string "It works" as a stream to the caller.

Listing 7.7 Stubbed `HttpURLConnection` class

```

[...]
import java.net.HttpURLConnection;
import java.net.ProtocolException;
import java.net.URL;
import java.io.InputStream;
import java.io.IOException;
import java.io.ByteArrayInputStream;

public class StubHttpURLConnection extends HttpURLConnection {
    private boolean isInput = true;
    protected StubHttpURLConnection(URL url) {

```

```

        super(url);
    }

    @Override
    public InputStream getInputStream() throws IOException {
        if (!isInput) {
            throw new ProtocolException(
                "Cannot read from URLConnection"
                + " if doInput=false (call setDoInput(true))");
        }
        ByteArrayInputStream readStream = new ByteArrayInputStream(
            new String("It works").getBytes());
        return readStream;
    }

    @Override
    public void connect() throws IOException {}

    @Override
    public void disconnect() {}

    @Override
    public boolean usingProxy() {
        return false;
    }
}

```

In this listing:

- `HttpURLConnection` is an abstract public class that does not implement an interface, so we extend it and override the methods wanted by the stub.
- In this stub, we provide an implementation for the `getInputStream` method, as it is the only method used by our code under test.
- If the code to be tested used more APIs from `HttpURLConnection`, we would need to stub these additional methods. This part is where the code would become more complex; we would need to reproduce the same behavior as the real `HttpURLConnection`.
- We test whether `setDoInput(false)` has been called in the code under test
 - ➊ The `isInput` flag will tell if we use the URL connection for input. Then, a call to the `getInputStream` method returns a `ProtocolException` (the behavior of `HttpURLConnection`). Fortunately, in most cases, we need to stub only a few methods, not the whole API.

7.4.3 Running the test

We will test the `getContent` method by stubbing the connection to the remote resource using the `TestWebClient1` test, which uses the `StubHttpURLConnection`. Figure 7.6 shows the result of the test.

As you can see, it is much easier to stub the connection than to stub the web resource. This approach does not provide the same level of testing (does not perform

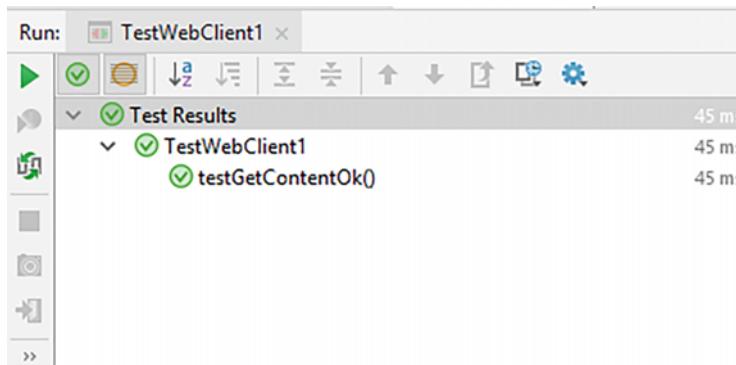


Figure 7.6 Result of executing `TestWebClient1` (which uses `StubHttpURLConnection`)

integration tests), but it enables you to write a focused unit test for the `WebClient` logic more easily.

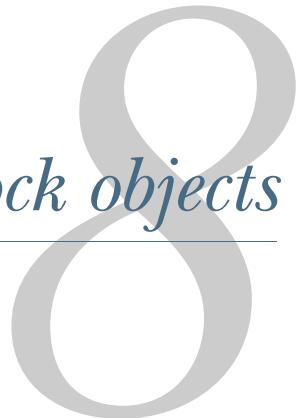
Chapter 8 demonstrates using mock objects, which allows fine-grained unit testing that is completely generic and (best of all) forces you to write good code. Although stubs are very useful in some cases, some people consider them to be vestiges of the past, when the consensus was that tests should be separate activities and should not modify existing code. The new mock-objects strategy not only allows modification of code but also favors it. Using mock objects is more than a unit testing strategy: it is a completely new way of writing code.

Summary

This chapter has covered the following:

- Analyzing when to use stubs: when you cannot modify an existing complex or fragile system, when you depend on an uncontrollable environment, to replace a full-blown external system, or for coarse-grained testing.
- Analyzing when not to use stubs: when you need fine-grained tests to provide precise messages that underline the exact cause of a failure, or when you would like to test a small portion of the code in isolation.
- Demonstrating how using a stub helps us unit test code accessing a remote web server using the Java `HttpURLConnection` API.
- In particular, implementing a stub for a remote web server by using the open source Jetty server. The embeddable nature of Jetty lets us concentrate on stubbing only the Jetty HTTP request handler, instead of having to stub the whole container.
- Implementing a more lightweight solution by stubbing the Java `HttpURLConnection` class.

Testing with mock objects



This chapter covers

- Introducing and demonstrating mock objects
- Executing different refactorings with the help of mock objects
- Practicing on an HTTP connection sample application
- Working with and comparing the EasyMock, JMock, and Mockito frameworks

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

—Rich Cook

Unit testing each method in isolation from other methods or the environment is certainly a nice goal. How do you perform this task? You saw in chapter 7 that the stubbing technique lets you unit test portions of code by isolating them from the environment (such as by stubbing a web server, the filesystem, a database, and so on). What about fine-grained isolation, such as being able to isolate a method call

to another class? Can you achieve this without deploying huge amounts of energy that would negate the benefits of having tests?

The answer is yes. The technique is called *mock objects*. Tim Mackinnon, Steve Freeman, and Philip Craig first presented the mock-objects concept at XP2000. The strategy allows you to unit test at the finest possible level. You develop method by method after having provided unit tests for each method.

8.1 Introducing mock objects

Testing in isolation offers strong benefits, such as the ability to test code that has not yet been written (as long as you at least have an interface to work with). In addition, testing in isolation helps teams unit test one part of the code without waiting for all the other parts.

The biggest advantage is the ability to write focused tests that test a single method, with no side effects resulting from other objects being called from the method under test. Small is beautiful. Writing small, focused tests is a tremendous help; small tests are easy to understand and do not break when other parts of the code are changed. Remember that one of the benefits of having a suite of unit tests is the courage it gives you to refactor mercilessly; the unit tests act as a safeguard against regression. If you have large tests, and your refactoring introduces a bug, several tests will fail. That result will tell you that there is a bug somewhere, but you will not know where. With fine-grained tests, potentially fewer tests are affected, and they provide precise messages that pinpoint the cause of the failure.

Mock objects (*mocks*, for short) are perfectly suited for testing a portion of code logic in isolation from the rest of the code. Mocks replace the objects with which your methods under test collaborate, thus offering a layer of isolation. In that sense, they are similar to stubs. The similarity ends there, however, because mocks do not implement any logic: they are empty shells that provide methods that let the tests control the behavior of all the business methods of the faked class.

A stub is created with a predetermined behavior, even a very simple one, and this behavior cannot be changed when running different tests. A mock does not have a predetermined behavior. When running a test, you are setting the expectations on the mock before effectively using it. You may run different tests, and you may reinitialize a mock and set different expectations on it. The pattern for testing with a mock is this: *initialize mock > set expectations > execute test > verify assertions*. We discuss when to use mock objects in section 8.6 at the end of this chapter, after we show them in action on some examples.

8.2 Unit testing with mock objects

In this section, we present an application and tests by using mock objects. Imagine a very simple use case in which we want to be able to make a bank transfer from one account to another (figure 8.1 and listings 8.1 and 8.2).

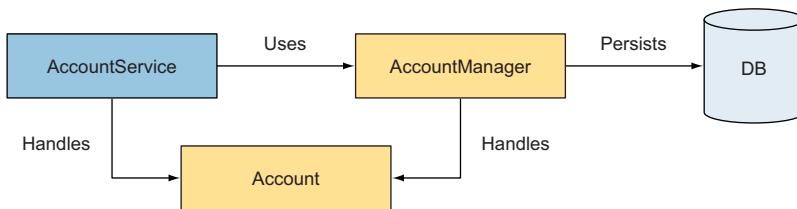


Figure 8.1 This simple bank account example uses a mock object to test an account transfer method.

The `AccountService` class offers services related to `Account` objects and uses `AccountManager` to persist data to the database (by using JDBC, for example). The service of interest to us materializes itself via the `AccountService.transfer` method, which makes the transfer. Without mocks, testing the `AccountService.transfer` behavior would imply setting up a database, presetting it with test data, deploying the code inside the container (Java EE application server, for example), and so forth. Although this process is required to ensure that the application works end to end, it is too much work when you want to unit test only your code logic.

Tested Data Systems Inc. creates software projects for other companies. One of the projects under development requires managing accounts and money transfers. The engineers need to design some solutions for it. The following listing presents a very simple `Account` object with two properties: an account ID and a balance.

Listing 8.1 The Account class

```

[...]
public class Account {
    private String accountId;
    private long balance;

    public Account(String accountId, long initialBalance) {
        this.accountId = accountId;
        this.balance = initialBalance;
    }

    public void debit(long amount) {
        this.balance -= amount;
    }

    public void credit(long amount) {
        this.balance += amount;
    }

    public long getBalance() {
        return this.balance;
    }
}
  
```

As part of the solution, the following `AccountManager` interface manages the life cycle and persistence of `Account` objects (limited to finding accounts by ID and updating accounts):

```
[...]
public interface AccountManager {
    Account findAccountForUser(String userId);
    void updateAccount(Account account);
}
```

The next listing shows the `transfer` method, designed for transferring money between two accounts. It uses the previously defined `AccountManager` interface to find the debit and credit accounts by ID and to update them.

Listing 8.2 The `AccountService` class

```
[...]
public class AccountService {
    private AccountManager accountManager;

    public void setAccountManager(AccountManager manager) {
        this.accountManager = manager;
    }

    public void transfer(String senderId, String beneficiaryId, long amount) {
        Account sender = accountManager.findAccountForUser(senderId);
        Account beneficiary =
            accountManager.findAccountForUser(beneficiaryId);

        sender.debit(amount);
        beneficiary.credit(amount);
        this.accountManager.updateAccount(sender);
        this.accountManager.updateAccount(beneficiary);
    }
}
```

We want to be able to unit test the `AccountService.transfer` behavior. For that purpose, until the implementation of the `AccountManager` interface is ready, we use a mock implementation of the `AccountManager` interface because the `transfer` method is using this interface, and we need to test it in isolation.

Listing 8.3 The `MockAccountManager` class

```
[...]
import java.util.HashMap;

public class MockAccountManager implements AccountManager {
    private Map<String, Account> accounts = new HashMap<String, Account>();

    public void addAccount(String userId, Account account) {
        this.accounts.put(userId, account);
    }
}
```



```

public Account findAccountForUser(String userId) {
    return this.accounts.get(userId);
}

public void updateAccount(Account account) {
    // do nothing
}
}

```

In this listing:

- The `addAccount` method uses an instance variable to hold the values to return ①. Because we have several `Account` objects that we want to be able to return, we store them in a `HashMap`. This step makes the mock generic and able to support different test cases. One test could set up the mock with one account, another test could set it up with two accounts or more, and so forth.
- We implement a method to retrieve the account from the `accounts` map ②. We can retrieve only accounts that were added earlier.
- The `updateAccount` method does nothing for now, and it does not return any value ③. Thus, we do nothing.

JUnit best practices: Do not write business logic in mock objects

The single most important rule to follow when writing a mock is that it should not have any business logic: a mock must be a dumb object that does only what the test tells it to do. In other words, it is purely driven by the tests. This characteristic is precisely the opposite of stubs, which contain all the logic (see chapter 7).

This point has two nice corollaries. First, mock objects can be generated easily. Second, because mock objects are empty shells, they are too simple to break and do not need testing themselves.

Now we are ready to write a unit test for `AccountService.transfer`. The following listing shows a typical test that uses a mock.

Listing 8.4 Testing transfer with MockAccountManager

```

[...]
public class TestAccountService {

    @Test
    public void testTransferOk() {
        Account senderAccount = new Account("1", 200);
        Account beneficiaryAccount = new Account("2", 100);

        MockAccountManager mockAccountManager = new MockAccountManager();
        mockAccountManager.addAccount("1", senderAccount);
        mockAccountManager.addAccount("2", beneficiaryAccount);

        AccountService accountService = new AccountService();
        accountService.setAccountManager(mockAccountManager);
    }
}

```

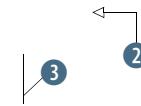
```

accountService.transfer("1", "2", 50);

assertEquals(150, senderAccount.getBalance());
assertEquals(150, beneficiaryAccount.getBalance());
}

}

```



As usual, a test has three steps: setup ①, execution ②, and the verification of the result ③. During the test setup, we create the `MockAccountManager` object and define what it should return when it's called for the two accounts we manipulate (the sender and beneficiary accounts). Practically, setting the expectations of the `MockAccountManager` object by adding two accounts to it transforms it into our own defined mock. As stated earlier, one of the characteristics of a mock is that when you run a test, you are setting the expectations on the mock before effectively using it.

We have succeeded in testing the `AccountService` code in isolation from the other domain object, `AccountManager`, which in this case did not exist but could have been implemented with JDBC in real life.

JUnit best practices: Test only what could possibly break

You may have noticed that we did not mock the `Account` class. The reason is that this data-access-object class does not need to be mocked; it does not depend on the environment, and it is very simple. The other tests use the `Account` object, so they test it indirectly. If this class failed to operate correctly, the tests that rely on `Account` would fail and alert us to the problem.

At this point in the chapter, you should have a reasonably good understanding of mocks. The next section demonstrates that writing unit tests with mocks leads to refactoring the code under test—and that this process is a good thing.

8.3 Refactoring with mock objects

Some people used to say that unit tests should be fully transparent to the code under test and that runtime code should not be changed to simplify testing. *This is wrong!* Unit tests are first-class users of the runtime code and deserve the same consideration as any other user. If the code is too inflexible for the tests to use, we should correct the code.

Consider the following piece of code created by one of the engineers at Tested Data Systems. This engineer is taking care of the implementation of the `AccountManager` class that we previously mocked until it is fully available.

Listing 8.5 The `DefaultAccountManager` class

```

[...]
import java.util.PropertyResourceBundle;
import java.util.ResourceBundle;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

```

```
[...]
public class DefaultAccountManager1 implements AccountManager {
    private static final Log logger =
        LogFactory.getLog(DefaultAccountManager1.class); ①

    public Account findAccountForUser(String userId) {
        logger.debug("Getting account for user [" + userId + "]");
        ResourceBundle bundle =
            PropertyResourceBundle.getBundle("technical"); ②
        String sql = bundle.getString("FIND_ACCOUNT_FOR_USER");
        // Some code logic to load a user account using JDBC
        [...]
    }
    [...]
}
```

In this listing:

- We create a Log object ①.
- We retrieve an SQL command ②.

Does the code look fine to you? We can see two issues, both of which relate to code flexibility and the ability to resist change. The first problem is that it is not possible to decide to use a different Log object, as it is created inside the class. For testing, for example, you probably would like to use a Log that does nothing, but you cannot do so.

As a rule, a class like this one should be able to use whatever Log it is given. The goal of this class is not to create loggers, but to perform some JDBC logic. The same goal applies to `PropertyResourceBundle`. It may sound OK right now, but what happens if we decide to use XML to store the configuration? Again, it should not be the goal of this class to decide what implementation to use.

An effective design strategy is to pass to an object any other object that is outside its immediate business logic. Ultimately, as you move up in the calling layers, the decision to use a given logger or configuration should be pushed to the top level. This strategy provides the best possible code flexibility and ability to cope with changes. And as we all know, change is the only constant. Taking these issues into consideration, the Tested Data Systems engineer who created the code will need to refactor it.

8.3.1 **Refactoring example**

Refactoring all code so that domain objects are passed around can be time consuming. You may not be ready to refactor the entire application just to be able to write a unit test. Fortunately, an easy refactoring technique lets you keep the same interface for the code but allows it to be passed domain objects that it should not create. As proof, the following listing shows what the refactored `DefaultAccountManager1` class could look like.

Listing 8.6 Refactoring DefaultAccountManager2 for testing

```
[...]
public class DefaultAccountManager2 implements AccountManager {
    private Log logger;
    private Configuration configuration; 1

    public DefaultAccountManager2() {
        this(LogFactory.getLog(DefaultAccountManager2.class),
            new DefaultConfiguration("technical"));
    }

    public DefaultAccountManager2(Log logger, Configuration configuration) {
        this.logger = logger;
        this.configuration = configuration;
    }

    public Account findAccountForUser(String userId) {
        this.logger.debug("Getting account for user [" + userId + "]");
        this.configuration.getSQL("FIND_ACCOUNT_FOR_USER");
        // Some code logic to load a user account using JDBC
        [...]
    }
}
[...]
```

At 1, we swap the `PropertyResourceBundle` class from the previous listing in favor of a new `Configuration` field. This swap makes the code more flexible because it introduces an interface (which will be easy to mock), and the implementation of the `Configuration` interface can be anything we want (including using resource bundles). The design is better now because we can use and reuse the `DefaultAccountManager2` class with any implementation of the `Log` and `Configuration` interfaces (if we use the constructor that takes two parameters). The class can be controlled from the outside (by its caller). We have a no-args constructor and a constructor with arguments. The no-args constructor initializes the `logger` and `configuration` field members with default implementations.

8.3.2 Refactoring considerations

With this refactoring, we have provided a trap door for controlling the domain objects from the tests. We retain backward compatibility and pave an easy refactoring path for the future. Calling classes can start using the new constructor at their own pace.

Should you worry about introducing trap doors to make your code easier to test? Here's how Extreme Programming guru Ron Jeffries explains it:

My car has a diagnostic port and an oil dipstick. There is an inspection port on the side of my furnace and on the front of my oven. My pen cartridges are transparent so I can see if there is ink left.

And if I find it useful to add a method to a class to enable me to test it, I do so. It happens once in a while, for example in classes with easy interfaces and complex inner function (probably starting to want an Extract Class).

I just give the class what I understand of what it wants, and keep an eye on it to see what it wants next.

Design patterns in action: Inversion of Control (IoC)

Applying the IoC pattern to a class means removing the creation of all object instances for which this class is not directly responsible and passing any needed instances instead. The instances may be passed by a specific constructor or a setter, or as parameters of the methods that need them. It becomes the responsibility of the calling code to set these domain objects correctly on the class that is called.^a

^a See Spring for a framework that implements the IoC pattern (<https://spring.io>).

IoC makes unit testing a breeze. To prove the point, here's how easily we can write a test for the `findAccountByUser` method.

Listing 8.7 The `testFindAccountByUser` method

```
public void testFindAccountByUser() {
    MockLog logger = new MockLog();
    MockConfiguration configuration = new MockConfiguration();
    configuration.setSQL("SELECT * [...]");
    DefaultAccountManager2 am = new DefaultAccountManager2(logger,
                                                          configuration);
    Account account = am.findAccountForUser("1234");
    // Perform asserts here
    [...]
}
```

In this listing:

- We use a mock logger that implements the `Log` interface but does nothing ①.
- Next, we create a `MockConfiguration` instance ② and set it up to return a given SQL query when `configuration.getSQL` is called.
- Finally, we create the instance of `DefaultAccountManager2` ③ that we will test, passing to it the `Log` and `Configuration` instances.

We have been able to completely control the logging and configuration behavior from outside the code to test, in the test code. As a result, the code is more flexible and allows for any logging and configuration implementation to be used. We will implement more of these code refactorings in this chapter and later ones. For now, the developer from Tested Data Systems has solved the issues we previously examined by taking this approach.

One last point to note is that if we write the test first, we will automatically design the code to be flexible. Flexibility is a key point in writing a unit test. If we test first, we will not incur the cost of refactoring the code for flexibility later.

8.4 Mocking an HTTP connection

To see how mock objects work in a practical example, let's go back to the application developed at Tested Data Systems that opens an HTTP connection to a remote server and reads the content of a page. In chapter 7, we tested that application using stubs, but the developers from Tested Data Systems decided to move to a mock-object approach to simulate the HTTP connection. In addition, they will create mocks for classes that do not implement a Java interface (namely, the `HttpURLConnection` class).

In the full implementation scenario, we start with an initial testing implementation, improve the implementation as we go, and modify the original code to make it more flexible. We also use mocks to test for error conditions.

As we dive in, we will keep improving both the test code and the sample application, exactly as we might if we were writing the unit tests for the same application. In the process, you will learn how to reach a simple and elegant testing solution while making the application code more flexible and capable of handling change.

Figure 8.2 shows the sample HTTP application, which consists of a simple `Web-Client.getContent` method performing an HTTP connection to a web resource executing on a web server. We want to be able to unit test the `getContent` method in isolation from the web resource.

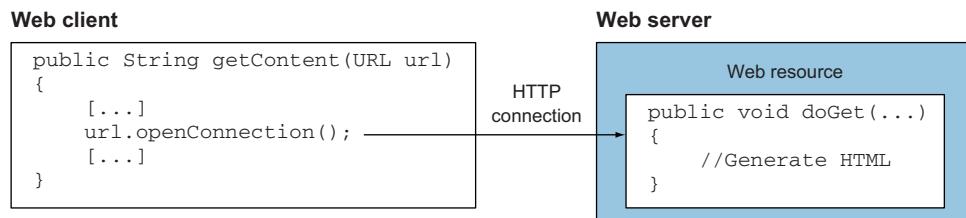


Figure 8.2 The sample HTTP application before the test is introduced

8.4.1 Defining the mock objects

Figure 8.3 illustrates a mock object. The `MockURL` class stands in for the real `URL` class, and all calls to the `URL` class in `getContent` are directed to the `MockURL` class. As you can see, the test is the controller: it creates and configures the behavior that the mock must have for this test, it (somehow) replaces the real `URL` class with the `MockURL` class, and it runs the test.

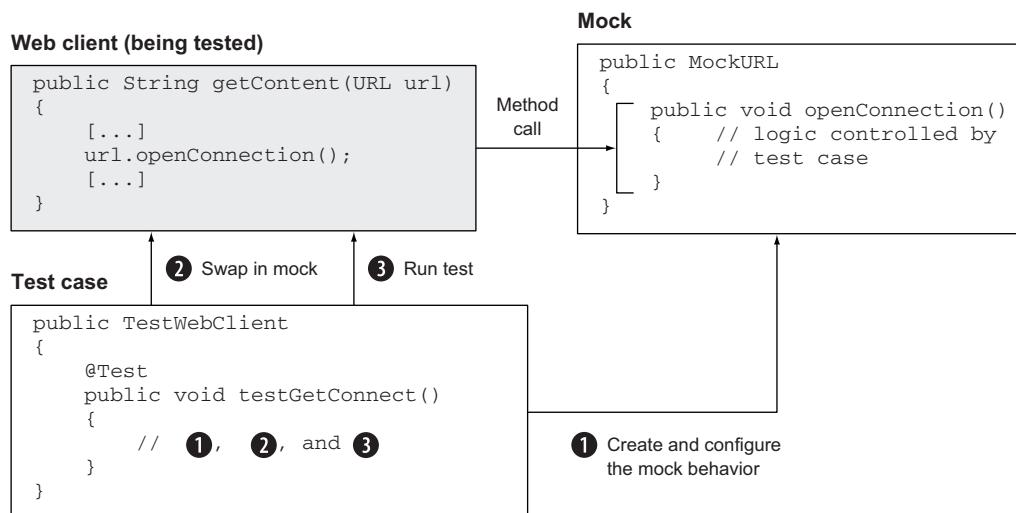


Figure 8.3 The steps involved in a test using mock objects: create and configure the mock object behavior, swap it in, and run the test.

Figure 8.3 shows an interesting aspect of the mock-objects strategy: the ability to swap in the mock in the production code. The perceptive reader will have noticed that because the `URL` class is final, it is not possible to create a `MockURL` class that extends it.

In the coming sections, we demonstrate how this feat can be performed in a different way (by mocking at another level). In any case, when using the mock-objects strategy, swapping in the mock instead of the real class is the hard part unless we are using dependency injection. This point may be viewed as negative for mock objects because you usually need to modify your code to provide a trap door. Ironically, modifying code to encourage flexibility is one of the strongest advantages of using mocks, as explained in sections 8.3.1 and 8.3.2.

8.4.2 Testing a sample method

The following code snippet opens an HTTP connection to a given URL and reads the content at that URL. Suppose that this code is one method of a bigger application that we want to unit test (see the `WebClient` class in chapter 7).

Listing 8.8 Sample method that opens an HTTP connection

```
[...]
import java.net.URL;
import java.net.HttpURLConnection;
import java.io.InputStream;
import java.io.IOException;
```

```

public class WebClient {
    public String getContent(URL url) {
        StringBuffer content = new StringBuffer();
        try {
            HttpURLConnection connection =
                (HttpURLConnection) url.openConnection();
            connection.setDoInput(true);
            InputStream is = connection.getInputStream();
            int count;
            while (-1 != (count = is.read())) {
                content.append( new String( Character.toChars( count ) ) );
            }
        } catch (IOException e) {
            return null;
        }
        return content.toString();
    }
}

```

In this listing:

- We open an HTTP connection ①.
- We read all the content that is received ②.
- If an error occurs, we return null ③. Admittedly, this is not the best possible error-handling solution, but it is good enough for the moment (and our tests will give us the courage to refactor later).

8.4.3 Try #1: Easy refactoring technique

The idea of this first refactoring technique applied at Tested Data Systems is to test the `getContent` method independently of a real HTTP connection to a web server. If you map the knowledge you acquired in section 8.2, it means writing a mock URL in which the `url.openConnection` method returns a mock `HttpURLConnection`. The `MockHttpURLConnection` class would provide an implementation that lets the test decide what the `getInputStream` method returns. Ideally, we would be able to write the following test.

Listing 8.9 testgetContentOk method

```

@Test
public void testgetContentOk() throws Exception {
    MockHttpURLConnection mockConnection = new MockHttpURLConnection();
    mockConnection.setupGetInputStream(
        new ByteArrayInputStream("It works".getBytes()));
    MockURL mockURL = new MockURL();
    mockURL.setupOpenConnection(mockConnection);
    WebClient client = new WebClient();
    String workingContent = client.getContent(mockURL);
    assertEquals("It works", workingContent);
}

```

In this listing:

- We create a mock HttpURLConnection ①.
- We create a mock URL ②.
- We test the `getContent` method ③.
- We assert the result ④.

Unfortunately, this approach does not work! The JDK `URL` class is a final class, and no `URL` interface is available. So much for extensibility.

We need to find another solution and, potentially, another object to mock. One solution is to stub the `URLStreamHandlerFactory` class. We explored this solution in chapter 7, so we must find a technique that uses mock objects: refactoring the `getContent` method. If you think about it, this method does two things: it gets an `HttpURLConnection` object and then reads the content from it. Refactoring leads to the following class. (Changes from listing 8.8 are shown in bold.) We have extracted the part that retrieved the `HttpURLConnection` object.

Listing 8.10 Extracting retrieval of the connection object from `getContent`

```
public class WebClient1 {
    public String getContent(URL url) {
        StringBuffer content = new StringBuffer();
        try {
            HttpURLConnection connection = createHttpURLConnection(url); ①
            InputStream is = connection.getInputStream();
            int count;
            while (-1 != (count = is.read())) {
                content.append( new String( Character.toChars( count ) ) );
            }
        }
        catch (IOException e) {
            return null;
        }
        return content.toString();
    }

    protected HttpURLConnection createHttpURLConnection(URL url)
        throws IOException { ②
        return (HttpURLConnection) url.openConnection();
    }
}
```

In this listing, we call `createHttpURLConnection` ① to create the HTTP connection. How does this solution test `getContent` more effectively? It allows us to apply a useful trick, which writes a test helper class that extends the `WebClient1` class and overrides its `createHttpURLConnection` method, as follows:

```
private class TestableWebClient extends WebClient1 {
    private HttpURLConnection connection;
    public void setHttpURLConnection(HttpURLConnection connection) {
        this.connection = connection;
```

```

    }
    public HttpURLConnection createHttpURLConnection(URL url)
        throws IOException {
        return this.connection;
    }
}

```

A common refactoring approach called *Method Factory* is especially useful when the class to mock has no interface. The strategy is to extend that class, add some setter methods to control it, and override some of its getter methods to return what we want for the test.

In the test, we can call the `setHttpURLConnection` method, passing it the mock `HttpURLConnection` object. Now the test becomes the following. (Differences are shown in bold.)

Listing 8.11 Modified `testGetContentOk` method

```

@Test
public void testGetContentOk() throws Exception {
    MockHttpURLConnection mockConnection = new MockHttpURLConnection();
    mockConnection.setupInputStream(
        new ByteArrayInputStream("It works".getBytes()));
    TestableWebClient client = new TestableWebClient();
    client.setHttpURLConnection(mockConnection);
    String workingContent =
        client.getContent(new URL("http://localhost"));
    assertEquals("It works", workingContent);
}

```



In this listing:

- We configure `TestableWebClient` ① so that the `createHttpURLConnection` method returns a mock object.
- Next, the `getContent` method is called ②.
- In the case at hand, the Method Factory approach is OK, but it is not perfect. It is a bit like the Heisenberg Uncertainty Principle: the act of subclassing the class under test changes its behavior, so when you test the subclass, what are you truly testing?

This technique is useful as a means of opening an object to be more testable, but stopping here means testing something that is similar (but not identical) to the class you want to test. You are not writing tests for a third-party library and cannot change the code; you have complete control of the code to test. You can enhance the code and make it more test-friendly in the process.

8.4.4 Try #2: Refactoring by using a class factory

The developers at Tested Data Systems want to give another refactoring approach a try by applying the IoC pattern, which says that any resource we use needs to be passed to the `getContent` method or `WebClient` class. The only resource we use is the

HttpURLConnection object. We could change the WebClient.getContent signature to

```
public String getContent(URL url, HttpURLConnection connection)
```

This change means we are pushing the creation of the HttpURLConnection object to the caller of WebClient. The URL is retrieved from the HttpURLConnection class, however, and the signature does not look very nice. Fortunately, a better solution involves creating a ConnectionFactory interface, as shown in listings 8.12 and 8.13. The role of classes implementing the ConnectionFactory interface is to return an InputStream from a connection, whatever the connection may be (HTTP, TCP/IP, and so on). This refactoring technique is sometimes called a *Class Factory* refactoring.

Listing 8.12 The ConnectionFactory class

```
[...]
import java.io.InputStream;
public interface ConnectionFactory {
    InputStream getData() throws Exception;
}
```

The WebClient code becomes as shown in the next listing.

Listing 8.13 Refactored WebClient2 using ConnectionFactory

```
[...]
import java.io.InputStream;

public class WebClient2 {
    public String getContent(ConnectionFactory connectionFactory) {
        String workingContent;
        StringBuffer content = new StringBuffer();
        try(InputStream is = connectionFactory.getData()) {
            int count;
            while (-1 != (count = is.read())) {
                content.append( new String( Character.toChars( count ) ) );
            }
            workingContent = content.toString();
        }
        catch (Exception e) {
            workingContent = null;
        }
        return workingContent;
    }
}
```

This solution is better because we have made the retrieval of the data content independent of the way we get the connection. The first implementation worked only with URLs using the HTTP protocol. The new implementation can work with any standard

protocol (file://, http://, ftp://, jar://, and so forth) or even a custom protocol. The next listing shows the `ConnectionFactory` implementation for the HTTP protocol.

Listing 8.14 The `HttpURLConnectionFactory` class

```
[...]
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class HttpURLConnectionFactory implements ConnectionFactory {
    private URL url;
    public HttpURLConnectionFactory(URL url) {
        this.url = url;
    }
    public InputStream getData() throws Exception {
        HttpURLConnection connection =
            (HttpURLConnection) this.url.openConnection();
        return connection.getInputStream();
    }
}
```

Now we can easily test the `getContent` method by writing a mock for `ConnectionFactory`.

Listing 8.15 The `MockConnectionFactory` class

```
[...]
import java.io.InputStream;

public class MockConnectionFactory implements ConnectionFactory {
    private InputStream inputStream;

    public void setData(InputStream stream) {
        this.inputStream = stream;
    }
    public InputStream getData() throws Exception {
        return inputStream;
    }
}
```

As usual, the mock does not contain any logic and is completely controllable from the outside (by calling the `setData` method). Now we can easily rewrite the test to use `MockConnectionFactory`.

Listing 8.16 Refactored test using `MockConnectionFactory`

```
[...]
import java.io.ByteArrayInputStream;

public class TestWebClient {
```

```

@Test
public void testGetContentOk() throws Exception {
    MockConnectionFactory mockConnectionFactory =
        new MockConnectionFactory();
    mockConnectionFactory.setData(
        new ByteArrayInputStream("It works".getBytes()));

    WebClient2 client = new WebClient2();
    String workingContent = client.getContent(mockConnectionFactory);
    assertEquals("It works", workingContent);
}
}

```

We have achieved our initial goal: to unit test the code logic of the `WebClient.getContent` method, which returns the content of a given URL. During this process, we had to refactor the method for the test, which led to a more extensible implementation that is better able to cope with change.

8.5 Using mocks as Trojan horses

Mock objects are Trojan horses, but they are not malicious. Mocks replace real objects from the inside without the calling classes being aware of it. Mocks have access to internal information about the class, making them quite powerful. In the examples so far, we have used them to emulate real behaviors, but we haven't mined all the information they can provide.

It is possible to use mocks as probes by letting them monitor the method calls that the object under test makes. Consider the HTTP connection example. One interesting call we could monitor is the `close` method on the `InputStream`, as we would like to make sure that the programmer will always close the stream; otherwise, we may experience a resource leak. A *resource leak* occurs when we do not close a reader, a scanner, a buffer, or another process that uses resources and needs to clean them out of memory. We have not used a mock object for `InputStream` so far, but we can easily create one and provide a `verify` method to ensure that `close` has been called. Then we can call the `verify` method at the end of the test to verify that all methods that should have been called were called (see listing 8.17). We may also want to verify that `close` has been called exactly once and raise an exception if it was called more than once or not at all. These kinds of verifications are often called *expectations*.

DEFINITION *Expectation*—When we're talking about mock objects, an expectation is a feature built into the mock that verifies whether the external class calling this mock has the correct behavior. A database connection mock, for example, could verify that the `close` method on the connection is called exactly once during any test that involves code using this mock.

To demonstrate the expectations that the resource has been closed and we avoid a resource leak, take a look at the following listing.

Listing 8.17 Mock InputStream with an expectation on close

```
[...]
import java.io.IOException;
import java.io.InputStream;

public class MockInputStream extends InputStream {
    private String buffer;
    private int position = 0;
    private int closeCount = 0;
    public void setBuffer(String buffer) {
        this.buffer = buffer;
    }
    public int read() throws IOException {
        if (position == this.buffer.length()) {
            return -1;
        }
        return this.buffer.charAt(this.position++);
    }
    public void close() throws IOException {
        closeCount++;
        super.close();
    }
    public void verify() throws java.lang.AssertionError {
        if (closeCount != 1) {
            throw new AssertionError ("close() should "
                + "have been called once and once only");
        }
    }
}
```

In this listing:

- We tell the mock what the `read` method should return ①.
- We count the number of times `close` is called ②.
- We verify that the expectations are met ③.

In the case of the `MockInputStream` class, the expectation for `close` is simple: we always want it to be called once. Most of the time, however, the expectation for `closeCount` depends on the code under test. A mock usually has a method such as `setExpectedCloseCalls` so that the test can tell the mock what to expect.

Modify the `testGetContentOk` test method as follows to use the new `MockInputStream`:

```
[...]

public class TestWebClientFail {

    @Test
    public void testGetContentOk() throws Exception {
        MockConnectionFactory mockConnectionFactory =
            new MockConnectionFactory();
```

```

MockInputStream mockStream = new MockInputStream();
mockStream.setBuffer("It works");
mockConnectionFactory.setData(mockStream);
WebClient2 client = new WebClient2();
String workingContent = client.getContent(mockConnectionFactory);

assertEquals("It works", workingContent);
mockStream.verify();
}
}

```

Instead of using a real `ByteArrayInputStream`, as in previous tests, we use `MockInputStream`. Note that we call the `verify` method of `MockInputStream` at the end of the test to ensure that all expectations are met. The result of the test is shown in figure 8.4.

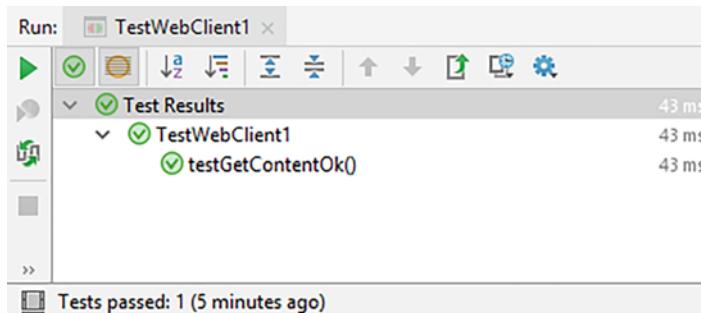


Figure 8.4 The result of running `TestWebClient1`

There are other handy uses for expectations. If you have a component manager calling different methods of your component life cycle, for example, you might expect them to be called in a given order. Or you might expect a given value to be passed as a parameter to the mock. The general idea is that in addition to behaving the way we want during a test, our mock can provide useful feedback on its use. The test can provide information regarding the number of method invocations, parameters that are passed to the methods, and the order in which methods are called.

The next section demonstrates the use of some of the most popular open source mocking frameworks. They are powerful enough for our needs, and we don't need to implement our mocks from the beginning.

8.6 Introducing mock frameworks

So far, the engineers at Tested Data Systems have implemented the mock objects from scratch. The task is not tedious, but it recurs. You might guess that we don't need to reinvent the wheel every time we need a mock, and you would be right: a lot of good

existing frameworks can facilitate the use of mocks in our projects. In this section, we will take a closer look at three of the most widely used mock frameworks: EasyMock, JMock, and Mockito.

The developers at Tested Data Systems will try to rework the example HTTP connection application to demonstrate how to use the three frameworks and to come up with a basis for choosing one of the alternatives. People have their own experiences, preferences, and habits, and because the engineers have these three framework alternatives, they would like to compare them and make some conclusions.

8.6.1 Using EasyMock

EasyMock (<https://easymock.org>) is an open source framework that provides useful classes for mocking objects. To work with it, we need to add the following dependencies to the pom.xml file.

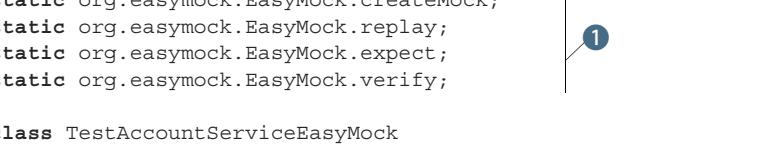
Listing 8.18 EasyMock dependencies from the pom.xml configuration file

```
<dependency>
  <groupId>org.easymock</groupId>
  <artifactId>easymock</artifactId>
  <version>4.2</version>
</dependency>
<dependency>
  <groupId>org.easymock</groupId>
  <artifactId>easymockclassextension</artifactId>
  <version>3.2</version>
</dependency>
```

While trying to introduce EasyMock, the developers at Tested Data Systems revise some of the mocks constructed in the previous sections. The start is simple: reworking the `AccountService` test from listing 8.2.

Listing 8.19 Reworking the TestAccountService test using EasyMock

```
[...]  
import static org.easymock.EasyMock.createMock;  
import static org.easymock.EasyMock.replay;  
import static org.easymock.EasyMock.expect;  
import static org.easymock.EasyMock.verify;  
  
public class TestAccountServiceEasyMock  
{  
    private AccountManager mockAccountManager;  
  
    @BeforeEach  
    public void setUp()  
    {  
        mockAccountManager = createMock( "mockAccountManager",  
                                         AccountManager.class );  
    }  
}
```



```

@Test
public void testTransferOk()
{
    Account senderAccount = new Account( "1", 200 );
    Account beneficiaryAccount = new Account( "2", 100 ); | 4

    // Start defining the expectations
    mockAccountManager.updateAccount( senderAccount );
    mockAccountManager.updateAccount( beneficiaryAccount ); | 5

    expect( mockAccountManager.findAccountForUser( "1" ) )
        .andReturn( senderAccount );
    expect( mockAccountManager.findAccountForUser( "2" ) )
        .andReturn( beneficiaryAccount ); | 6

    // we're done defining the expectations
    replay( mockAccountManager ); | 7

    AccountService accountService = new AccountService();
    accountService.setAccountManager( mockAccountManager );
    accountService.transfer( "1", "2", 50 ); | 8

    assertEquals( 150, senderAccount.getBalance() );
    assertEquals( 150, beneficiaryAccount.getBalance() ); | 9

}

@AfterEach
public void tearDown()
{
    verify( mockAccountManager ); | 10
}
}

```

As you can see, the listing is pretty much the same length as listing 8.4, but we do not write additional mock classes. In this listing:

- We start by defining the imports that we need from the EasyMock library ①. We rely heavily on static imports.
- We declare the object that we would like to mock ②. Notice that our AccountManager is an interface. The reason is simple: the core EasyMock framework can mock only interface objects.
- We call the `createMock` method to create a mock of the class we want ③.
- As in listing 8.4, we create two `Account` objects that we are going to use in our tests ④. After that, we start declaring our expectations.
- With EasyMock, we declare the expectations in two ways. When the method return type is `void`, we call it on the mock object ⑤. When the method returns any kind of object, we use the `expect` and `andReturn` methods from the EasyMock API ⑥.
- When we finish defining the expectations, we call the `replay` method. The `replay` method passes the mock from the phase where we record the method

we expect to be called to where we test. Before, we simply recorded the behavior, but the object was not working as a mock. After calling `replay`, the object works as expected 7.

- We call the `transfer` method to transfer some money between the two accounts 8.
- We assert the expected result 9.
- The `@AfterEach` method that is executed after every `@Test` method holds the verification of the expectations. With EasyMock, we can call the `verify` method with any mock object 10 to verify that the method-call expectations we declared were triggered. Including the verification in the `@AfterEach` method allows us to introduce new tests easily, and we'll rely on the execution of the `verify` method from now on.

JUnit best practices: EasyMock object creation

Here is a nice-to-know tip on the `createMock` method. If you check the API of EasyMock, you see that the `createMock` method comes with numerous signatures. The signature that we use is

```
createMock(String name, Class clazz);
```

but there's also

```
createMock(Class clazz);
```

Which one should we use? `createMock(String name, Class clazz)` is better. If we use `createMock(Class clazz)` and our expectations are not met, we get an error message like the following:

```
java.lang.AssertionError:  
Expectation failure on verify:  
    read(): expected: 7, actual: 0
```

As you see, this message is not as descriptive as it could be. If we use `createMock(String name, Class clazz)` instead and map the class to a given name, we will get something like the following:

```
java.lang.AssertionError:  
Expectation failure on verify:  
    name.read(): expected: 7, actual: 0
```

That was pretty easy, right? So how about moving a step forward and revising a more complicated example? Listing 8.20 demonstrates the reworked `WebClient` test from listing 8.16: verifying the correct value returned by the `getContent` method.

We want to test the `getContent` method of the `WebClient`. For this purpose, we need to mock all the dependencies to that method. In this example, we have two dependencies: `ConnectionFactory` and `InputStream`. It appears to be a problem because EasyMock can only mock interfaces, and `InputStream` is a class.

To mock the `InputStream` class, we are going to use the class extensions of EasyMock, which represent an extension project of EasyMock that lets you generate mock objects¹ for classes and interfaces. These class extensions are addressed by the second Maven dependency in the following listing..

Listing 8.20 Reworking the `WebClient` test using EasyMock

```
[...]
import static org.easymock.classextension.EasyMock.createMock;
import static org.easymock.classextension.EasyMock.replay;
import static org.easymock.classextension.EasyMock.verify;

public class TestWebClientEasyMock {
    private ConnectionFactory factory; 1
    private InputStream stream; 2

    @BeforeEach
    public void setUp() {
        factory = createMock( "factory", ConnectionFactory.class ); 3
        stream = createMock( "stream", InputStream.class );
    }

    @Test
    public void testGetContentOk() throws Exception {
        expect( factory.getData() ).andReturn( stream );
        expect( stream.read() ).andReturn( new Integer( (byte) 'W' ) );
        expect( stream.read() ).andReturn( new Integer( (byte) 'o' ) );
        expect( stream.read() ).andReturn( new Integer( (byte) 'r' ) );
        expect( stream.read() ).andReturn( new Integer( (byte) 'k' ) );
        expect( stream.read() ).andReturn( new Integer( (byte) 's' ) );
        expect( stream.read() ).andReturn( new Integer( (byte) '!' ) );
        expect( stream.read() ).andReturn( -1 );
        stream.close(); 4
        replay( factory ); 5
        replay( stream );
    }

    WebClient2 client = new WebClient2();
    String workingContent = client.getContent( factory ); 6
    assertEquals( "Works!", workingContent ); 7
}

[...]
@Test
public void testGetContentCannotCloseInputStream() throws Exception {
    expect( factory.getData() ).andReturn( stream );
    expect( stream.read() ).andReturn( -1 );
}
```

¹ Final and private methods cannot be mocked.

```

    stream.close();
    expectLastCall().andThrow(new IOException("cannot close"));
    replay(factory);
    replay(stream);
    WebClient2 client = new WebClient2();
    String workingContent = client.getContent(factory);

    assertNull(workingContent);
}

@AfterEach
public void tearDown() {
    verify(factory);
    verify(stream);
}
}

```

In this listing:

- We start by importing the objects that we need ①. Notice that because we use the class extensions of EasyMock, we need to import the `org.easymock.classextension.EasyMock` object instead of `org.easymock.EasyMock`. Now we are ready to create mock objects of classes and interfaces by using the statically imported methods of the class extensions.
- As in the previous listings, we declare the objects that we want to mock ② and call the `createMock` method to initialize them ③.
- We define the expectation of the stream when the `read` method is invoked ④. (Notice that to stop reading from the `stream`, the last thing to return is `-1`.) Working with a low-level stream, we define how to read one byte at a time, as `InputStream` is reading byte by byte. We expect the `close` method to be called on the `stream` ⑤.
- To denote that we are done declaring our expectations, we call the `replay` method ⑥. The `replay` method passes the mock from the phase where we record the method that we expect to be called to where we test. Initially, the mock just records what it is supposed to do and will not react if called. But after calling `replay`, it will use the recorded behavior when called.
- The rest is invoking the method under test ⑦ and asserting the expected result ⑧.
- We also add another test to simulate a condition when we cannot close the `InputStream`. We define an expectation in which we expect the `close` method of the stream to be invoked ⑨.
- On the next line, we declare that an `IOException` should be raised if this call occurs ⑩.

As the name of the framework suggests, using EasyMock is very easy, and it is an option you may consider for your projects.

8.6.2 Using JMock

So far, you've seen how to implement your own mock objects and use the EasyMock framework. In this section, we introduce the JMock framework (<http://jmock.org>). We'll follow the same scenario that the engineers from Tested Data Systems are following to evaluate the capabilities of a mock framework and compare them with those of other frameworks: testing money transfers with the help of a mock AccountManager, this time using JMock. To work with JMock, we need to add the following dependencies to the pom.xml file.

Listing 8.21 JMock dependencies from the pom.xml configuration file

```
<dependency>
  <groupId>org.jmock</groupId>
  <artifactId>jmock-junit5</artifactId>
  <version>2.12.0</version>
</dependency>
<dependency>
  <groupId>org.jmock</groupId>
  <artifactId>jmock-legacy</artifactId>
  <version>2.5.1</version>
</dependency>
```

As in section 8.6.1, we start with a simple example: reworking listing 8.4 by means of JMock, as shown in the following listing.

Listing 8.22 Reworking the TestAccountService test using JMock

```
[...]
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.junit5.JUnit5Mockery;


- 1


public class TestAccountServiceJMock {
    @RegisterExtension
    Mockery context = new JUnit5Mockery();


- 2


    private AccountManager mockAccountManager;


- 3


    @BeforeEach
    public void setUp() {
        mockAccountManager = context.mock( AccountManager.class );
    }


- 4


    @Test
    public void testTransferOk() {
        Account senderAccount = new Account( "1", 200 );
        Account beneficiaryAccount = new Account( "2", 100 );


- 5


        context.checking( new Expectations()
    {


- 6


        }
```

```

    oneOf( mockAccountManager ).findAccountForUser( "1" );
    will( returnValue( senderAccount ) );
    oneOf( mockAccountManager ).findAccountForUser( "2" );
    will( returnValue( beneficiaryAccount ) );

    oneOf( mockAccountManager ).updateAccount( senderAccount );
    oneOf( mockAccountManager )
        .updateAccount( beneficiaryAccount );
    }
}

AccountService accountService = new AccountService();
accountService.setAccountManager( mockAccountManager );
accountService.transfer( "1", "2", 50 );
}

assertEquals( 150, senderAccount.getBalance() );
assertEquals( 150, beneficiaryAccount.getBalance() );
}
}

```



In this listing:

- As always, we start by importing all the objects we need ①. Unlike EasyMock, the JMock framework does not rely on any static import features.
- JUnit 5 provides a programmatic way to register extensions. For JMock, this way is annotating a JUnit5Mockery nonprivate instance field with `@RegisterExtension`. (Part 4 of this book discusses the JUnit 5 extension model in detail.) The `context` object serves us to create mocks and define expectations ②.
- We declare the `AccountManager` that we would like to mock ③.
- In the `@BeforeEach` method that is executed before each of the `@Test` methods, we create the mock programmatically by means of the `context` object ④.
- As in previous listings, we declare two accounts between which we are going to transfer money ⑤.
- We start declaring the expectations by constructing a new `Expectations` object ⑥.
- We declare the first expectation ⑦, with each expectation having the form

```

invocation-count (mock-object).method(argument-constraints);
inSequence(sequence-name);
when(state-machine.is(state-name));
will(action);
then(state-machine.is(new-state-name));

```

All the clauses are optional except the bold ones: `invocation-count` and `mock-object`. We need to specify how many invocations will occur and on which object. After that, if the method returns an object, we can declare the return object by using the `will(returnValue())` construction.

- We start the transfer from one account to the other one ⑧ and then assert the expected results ⑨. It's as simple as that!

But wait—what happened with the verification of the invocation count? In all the previous examples, we needed to verify that invocations of expectations happened the expected number of times. Well, with JMock, you don't have to do that. The JMock extension takes care of this task, and if the expected calls were not made, the test will fail.

Following the EasyMock pattern, we rework listing 8.20 and show the `WebClient` test, this time using JMock.

Listing 8.23 Reworking the `TestWebClient` test using JMock

```
[...]

public class TestWebClientJMock {
    @RegisterExtension
    Mockery context = new JUnit5Mockery() {
        {
            setImposteriser( ClassImposteriser.INSTANCE );
        }
    };

    @Test
    public void testGetContentOk() throws Exception {
        ConnectionFactory factory =
            context.mock( ConnectionFactory.class );
        InputStream mockStream =
            context.mock( InputStream.class );

        context.checking( new Expectations() {
            {
                oneOf( factory ).getData();
                will( returnValue( mockStream ) );
                will( onConsecutiveCalls(
                    returnValue( Integer.valueOf( (byte) 'W' ) ),
                    returnValue( Integer.valueOf( (byte) 'o' ) ),
                    returnValue( Integer.valueOf( (byte) 'r' ) ),
                    returnValue( Integer.valueOf( (byte) 'k' ) ),
                    returnValue( Integer.valueOf( (byte) 's' ) ),
                    returnValue( Integer.valueOf( (byte) '!' ) ),
                    returnValue( -1 ) ) );
                oneOf( mockStream ).close();
            }
        } );
        WebClient2 client = new WebClient2();
        String workingContent = client.getContent( factory );
        assertEquals( "Works!", workingContent );
    }
}
```

```

@Test
public void testGetContentCannotCloseInputStream() throws Exception {

    ConnectionFactory factory =
        context.mock( ConnectionFactory.class );
    InputStream mockStream = context.mock( InputStream.class );

    context.checking( new Expectations() {
        {
            oneOf( factory ).getData();
            will( returnValue( mockStream ) );
            oneOf( mockStream ).read();
            will( returnValue( -1 ) );
            oneOf( mockStream ).close();
            will( throwException
                new IOException( "cannot close" ) );
        }
    } );
}

WebClient2 client = new WebClient2();

String workingContent = client.getContent( factory );
assertNull( workingContent );
}
}

```



In this listing:

- We start the test case by registering the JMock extension. The `JUnit5Mockery` nonprivate instance field `context` is annotated with `@RegisterExtension` ①. The JUnit 5 extension model will be analyzed in part 4.
- To tell JMock to create mock objects not only for interfaces but also for classes, we need to set the `impostoriser` property of the `context` ②. Now we can continue creating mocks the normal way.
- We declare and programmatically initialize the two mock objects ③.
- We start declaring the expectations ④. Notice how we declare the consecutive execution of the `read()` method of the stream ⑤ and the returned values.
- We call the method under test ⑥.
- We assert the expected result ⑦.
- For a full view of how to use the JMock mocking library, we also provide another `@Test` method that tests our `WebClient` under exceptional conditions. We declare the expectation of the `close()` method being triggered ⑧, and we instruct JMock to raise an `IOException` when this trigger happens ⑨.

As you can see, the JMock library is as easy to use as the EasyMock one but provides better integration with JUnit 5. We can register the `Mockery` `context` field programmatically. But we still need to look at the third proposed framework, Mockito, which is closer to the JUnit 5 paradigm.

8.6.3 Using Mockito

This section introduces Mockito (<https://site.mockito.org>), another popular mocking framework. The engineers at Tested Data Systems want to evaluate it and eventually introduce it into their projects. To work with Mockito, we need to add the following dependency to the pom.xml file.

Listing 8.24 Mockito dependency from the pom.xml configuration file

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.2.4</version>
    <scope>test</scope>
</dependency>
```

As with EasyMock and JMock, we rework the example from listing 8.4 (testing money transfers with the help of a mock AccountManager), this time by means of Mockito.

Listing 8.25 Reworking the TestAccountService test using Mockito

```
[...]

import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;
import org.mockito.MockitoExtension.class
public class TestAccountServiceMockito {

    @Mock
    private AccountManager mockAccountManager;

    @Test
    public void testTransferOk() {
        Account senderAccount = new Account( "1", 200 );
        Account beneficiaryAccount = new Account( "2", 100 );

        Mockito.lenient()
            .when(mockAccountManager.findAccountForUser("1"))
            .thenReturn(senderAccount);
        Mockito.lenient()
            .when(mockAccountManager.findAccountForUser("2"))
            .thenReturn(beneficiaryAccount);

        AccountService accountService = new AccountService();
        accountService.setAccountManager( mockAccountManager );
        accountService.transfer( "1", "2", 50 );
    }
}
```

```

        assertEquals( 150, senderAccount.getBalance() );
        assertEquals( 150, beneficiaryAccount.getBalance() );
    }
}

```



In this listing:

- As usual, we start by importing all the objects we need ①. This example does not rely on static import features.
- We extend this test by using MockitoExtension ②. @ExtendWith is a repeatable annotation that is used to register extensions for the annotated test class or test method. (We discuss the JUnit 5 extension model in detail in part 4.) For this Mockito example, we'll only note that this extension is needed to create mock objects through annotations ③. This code tells Mockito to create a mock object of type AccountManager.
- As in the previous listings, we declare two accounts between which we are going to transfer money ④.
- We start declaring the expectations by using the when method ⑤. Additionally, we use the lenient method to modify the strictness of object mocking. Without this method, only one expectation declaration is allowed for the same findAccountForUser method, whereas we need two (one for the "1" argument and one for the "2" argument).
- We start the transfer from one account to the other ⑥.
- We assert the expected results ⑦.

Following the pattern in the previous sections, we rework listing 8.20, which shows the WebClient test, with Mockito.

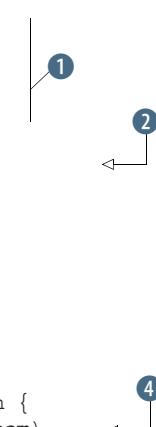
Listing 8.26 Reworking the TestWebClient test using Mockito

```

[...]
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import static org.mockito.Mockito.doThrow;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
public class TestWebClientMockito {
    @Mock
    private ConnectionFactory factory;
    @Mock
    private InputStream mockStream;
    @Test
    public void testGetContentOk() throws Exception {
        when(factory.getData()).thenReturn(mockStream);
    }
}

```



```

when(mockStream.read()).thenReturn((int) 'W')
    .thenReturn((int) 'o')
    .thenReturn((int) 'r')
    .thenReturn((int) 'k')
    .thenReturn((int) 's')
    .thenReturn((int) '!')
    .thenReturn(-1);

WebClient2 client = new WebClient2();

String workingContent = client.getContent( factory );
    ↪ 6

assertEquals( "Works!", workingContent );
    ↪ 7
}

@Test
public void testGetContentCannotCloseInputStream()
    throws Exception {
    when(factory.getData()).thenReturn(mockStream);
    when(mockStream.read()).thenReturn(-1);
    doThrow(new IOException( "cannot close" ))
        .when(mockStream).close();
    ↪ 8
    ↪ 9
    ↪ 10
}

WebClient2 client = new WebClient2();

String workingContent = client.getContent( factory );
    ↪ 6

assertNull( workingContent );
}
}

```

In this listing:

- We import the needed dependencies, static and nonstatic ①.
- We extend this test by using MockitoExtension ②.
- In this example, the extension is needed to create mock objects through annotations ③. It tells Mockito to create one mock object of type ConnectionFactory and one mock object of type InputStream.
- We start the declaration of the expectations ④. Notice how we declare the consecutive execution of the read() method of the stream ⑤ and the returned values.
- We call the method under test ⑥.
- We assert the expected result ⑦.
- We provide another @Test method that tests our WebClient under exceptional conditions. We declare the expectation of the factory.getData() method ⑧, and we declare the expectation of the mockStream.read() method ⑨. Then we instruct Mockito to raise an IOException when we close the stream ⑩.

As you can see, the Mockito framework can be used with the new JUnit 5 extension model—not programmatically, like JMock, but by using the JUnit 5 @ExtendWith

and the Mockito `@Mock` annotation. We use it to a greater extent in other chapters, as it is better integrated with JUnit 5.

Chapter 9 addresses another approach to unit testing components: in-container unit testing or integration unit testing.

Summary

This chapter has covered the following:

- Demonstrating the mock-objects technique. This approach lets you unit test code in isolation from other domain objects and the environment. When it comes to writing fine-grained unit tests, it helps you abstract yourself from the executing environment.
- Showing a nice side effect of writing mock-object tests: it forces you to rewrite some of the code under test. In practice, code is often not well written. With mock objects, you must think differently about the code and apply better design patterns, like interfaces and IoC.
- Implementing and comparing the Method Factory and Class Factory techniques for refactoring code that uses a mock HTTP connection.
- Using three mocking frameworks: EasyMock, JMock, and Mockito. Of the three frameworks, we demonstrated that at this time, Mockito has the best integration with JUnit 5, in particular with the JUnit 5 extension model.

In-container testing



This chapter covers

- Analyzing the limitations of mock objects
- Using in-container testing
- Evaluating and comparing stubs, mock objects, and in-container testing
- Working with Arquillian

The secret of success is sincerity. Once you can fake that you've got it made.

—Jean Giraudoux

This chapter examines one approach to unit testing components in an application container: in-container unit testing, or integration testing. These components are modules that may be developed by different programmers or teams and that need to be tested together or integrated. We will analyze in-container testing pros and cons and show what you can achieve by using the mock-objects approach introduced in chapter 8, where mock objects fall short, and how in-container testing enables you to write integration unit tests. We will also work with Arquillian, a Java EE

container-agnostic framework for integration testing, and show you how to use it to conduct integration testing. Finally, we will compare the stub, mock-object, and in-container approaches covered in this part of this book.

9.1 Limitations of standard unit testing

Let's start with the example servlet in listing 9.1, which implements the `HttpServlet` method `isAuthenticated`, the method we want to unit test. A *servlet* is a Java software application that extends the capabilities of a server. Our example company, Tested Data Systems, uses servlets to develop web applications, one of which is an online shop that serves new customers. To access the online shop, users need to connect to the frontend interface. The online shop needs authentication mechanisms so that the client knows who is performing the operations, and the Tested Data Systems engineers would like to test a method that verifies whether a user is authenticated.

Listing 9.1 Servlet implementing isAuthenticated

```
[...]
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class SampleServlet extends HttpServlet {
    public boolean isAuthenticated(HttpServletRequest request) {
        HttpSession session = request.getSession(false);
        if (session == null) {
            return false;
        }
        String authenticationAttribute =
            (String) session.getAttribute("authenticated");
        return Boolean.valueOf(authenticationAttribute).booleanValue();
    }
}
```

This servlet, although it is simple enough, allows us to show the limitation of standard unit testing. To test the method `isAuthenticated`, we need a valid `HttpServletRequest`. Because `HttpServletRequest` is an interface, we cannot just call new `HttpServletRequest`. The `HttpServletRequest` life cycle and implementation are provided by the container (in this case, a servlet container). The same is true of other server-side objects, such as `HttpSession`. JUnit alone is not enough to write a test for the `isAuthenticated` method and for servlets in general.

DEFINITION *Component* and *container*—A *component* is an application or a part of an application. A *container* is an isolated space where a component executes. A container offers services for the components it is hosting, such as life cycle, security, transactions, and so forth.

In the case of servlets and Java Server Pages (JSP), the container is a servlet container like Jetty or Tomcat. Other types of containers include JBoss (renamed WildFly), which is an Enterprise Java Beans (EJB) container. Java code can run in all these containers. As long as a container creates and manages objects at runtime, we should not use the standard JUnit techniques (the features of JUnit 5, stubs, and mock objects) to test those objects.

9.2 The mock-objects solution

The engineers at Tested Data Systems have to test the authentication mechanisms of the online shop. The first solution they consider for unit testing the `isAuthenticated` method (listing 9.1) is to mock the `HttpServletRequest` class using the approach described in chapter 8. Although mocking works, we need to write a lot of code to create a test. We can achieve the same result more easily by using the open source EasyMock framework (see chapter 8).

Listing 9.2 Testing a servlet with EasyMock

```
[...]
import javax.servlet.http.HttpServletRequest;
import static org.easymock.EasyMock.createStrictMock;
import static org.easymock.EasyMock.expect;
import static org.easymock.EasyMock.replay;
import static org.easymock.EasyMock.verify;
import static org.easymock.EasyMock.eq;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
[...]
public class TestSampleServletWithEasyMock {

    private SampleServlet servlet;
    private HttpServletRequest mockHttpServletRequest;
    private HttpSession mockHttpSession;

    @BeforeEach
    public void setUp() {
        servlet = new SampleServlet();
        mockHttpServletRequest =
            createStrictMock(HttpServletRequest.class);
        mockHttpSession = createStrictMock(HttpSession.class);
    }

    @Test
    public void testIsAuthenticatedAuthenticated() {
        expect(mockHttpServletRequest.getSession(eq(false)))
            .andReturn(mockHttpSession);
        expect(mockHttpSession.getAttribute(eq("authenticated")))
            .andReturn("true");
        replay(mockHttpServletRequest);
        replay(mockHttpSession);
        assertTrue(servlet.isAuthenticated(mockHttpServletRequest));
    }
}
```

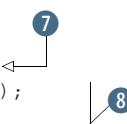
```

    @Test
    public void testIsAuthenticatedNotAuthenticated() {
        expect(mockHttpSession.getAttribute(eq("authenticated")))
            .andReturn("false");
        replay(mockHttpSession);
        expect(mockHttpServletRequest.getSession(eq(false)))
            .andReturn(mockHttpSession);
        replay(mockHttpServletRequest);
        assertFalse(servlet.isAuthenticated(mockHttpServletRequest));
    }

    @Test
    public void testIsAuthenticatedNoSession() {
        expect(mockHttpServletRequest.getSession(eq(false))).andReturn(null);
        replay(mockHttpServletRequest);
        replay(mockHttpSession);
        assertFalse(servlet.isAuthenticated(mockHttpServletRequest));
    }

    @AfterEach
    public void tearDown() {
        verify(mockHttpServletRequest);
        verify(mockHttpSession);
    }
}

```



In this listing:

- We start by importing the necessary classes and methods using static imports. We use the `EasyMock` class extensively ①.
- Next, we declare instance variables for the objects ② we want to mock: `HttpServletRequest` and `HttpSession`.
- The `setUp` method annotated with `@BeforeEach` ③ runs before each call to `@Test` methods; this is where we instantiate all the mock objects.
- Next, we implement our tests, following this pattern:
 - Set our expectations by using the `EasyMock` API ④.
 - Invoke the `replay` method to finish declaring our expectations ⑤. The `replay` method passes the mock from the phase where we record the method that we expect to be called to where we test. Initially, the mock just records what it is supposed to do and will not react if called. But after calling `replay`, it will use the recorded behavior when called.
 - Assert test conditions on the servlet ⑥.
- The `@AfterEach` method executed after each `@Test` method ⑦ calls the `EasyMock` `verify` API method ⑧ to check whether the mocked objects met all of our programmed expectations.

Mocking a minimal portion of a container is a valid approach for testing components. But mocking can be complicated and require a lot of code. The source code provided with this book also includes the servlet testing versions for the JMock and Mockito

frameworks. As with other kinds of tests, when the servlet changes, the test expectations must change to match. In the next section, the engineers at Tested Data Systems attempt to ease the task of testing the online shop’s authentication mechanism.

9.3 The step to *in-container* testing

The next approach in testing the `SampleServlet` is running the test cases where the `HttpServletRequest` and `HttpSession` objects live: in the container itself. This approach eliminates the need to mock any objects; we simply access the objects and methods we need in the real container.

For our example of testing the online shop’s authentication mechanism, we need the web request and session to be real `HttpServletRequest` and `HttpSession` objects managed by the container. Using a mechanism to deploy and execute our tests in a container, we have *in-container* testing. The next section covers options for *in-container* tests.

9.3.1 Implementation strategies

Two architectural choices drive *in-container* tests: server side and client side. We can drive the tests directly by controlling the server-side container and the unit tests. Alternatively, we can drive the tests from the client side, as shown in figure 9.1.

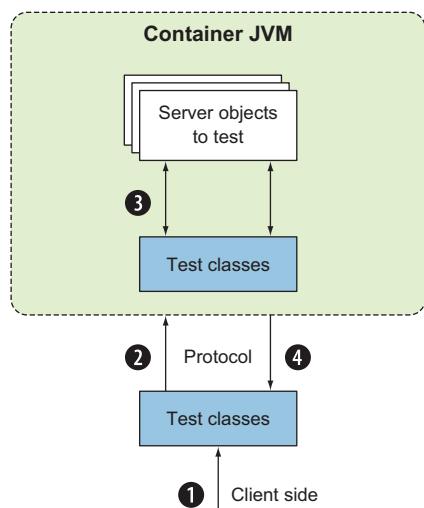


Figure 9.1 Life cycle of a typical *in-container* test: ① executing the client test classes, ② calling the same test case on the server side, ③ testing the domain objects, and ④ returning the results to the client

When the tests are packaged and deployed in the container and to the client, the JUnit test runner executes the test classes on the client ①. A test class opens a connection via a protocol such as HTTP(S) and calls the same test case on the server side ②. The server-side test case operates on server-side objects that are normally available (`HttpServletRequest`, `HttpServletResponse`, `HttpSession`, `ServletContext`,

and so on) and tests our domain objects ③. The server returns the result of the tests to the client ④, which an IDE or Maven can gather.

9.3.2 In-container testing frameworks

As we have just seen, in-container testing is applicable when code interacts with a container and tests cannot create valid container objects (`HttpServletRequest` in the preceding section). Our example uses a servlet container, but many other types of containers are available, including Java EE, web server, applets, and EJB. In all these cases, the in-container testing strategy can be applied.

9.4 Comparing stubs, mock objects, and in-container testing

This section compares¹ the approaches we presented to test components: stubs, mocks, and in-container testing. It draws on many questions from forums and mailing lists.

9.4.1 Stubs evaluation

Chapter 7 introduces using stubs as an out-of-container testing technique. Stubs work well to isolate a given class for testing and assert the state of its instances. Stubbing a servlet container allows us to track the number of requests made, the state of the server, and which URLs were requested, for example. But stubs have some predefined behaviors from the beginning of their existence.

When using mock objects, we code and verify expectations. The engineers at Tested Data Systems were able to check the business logic and how many times tests call some particular methods.

One of the biggest advantages of stubs compared to mocks is that stubs are easier to understand. Stubs isolate a class with little extra code compared with mock objects, which require an entire framework to function. The drawbacks of stubs are that they rely on external tools and hacks and do not track the state objects they fake.

In chapter 7, the engineers at Tested Data Systems easily faked a servlet container with stubs. Doing so with mock objects would be much harder, however, because the engineers would need to fake container objects with state and behavior.

Here is a summary of stub pros and cons:

- *Pros:*
 - Fast and lightweight
 - Easy to write and understand
 - Powerful
 - More coarse-grained tests
- *Cons:*
 - Specialized methods required to verify the state
 - Do not test the behavior of faked objects

¹ For an in-depth comparison of stubs and mocks technology, see “Mocks Aren’t Stubs,” by Martin Fowler, at <http://martinfowler.com/articles/mocksArentStubs.html>.

- Time consuming for complicated interactions
- Require more maintenance when the code changes

9.4.2 **Mock-objects evaluation**

The greatest advantage of mock objects over in-container testing is that mocks do not require a running container to execute tests. Tests can be set up and run quickly. The main drawback is that the tested components do not run in the container in which we will deploy them. The tests cannot verify the interaction between components and the container. Also, the tests do not test the interactions among the components themselves as they run in the container.

We still need a way to perform integration tests—to check that modules developed by different developers or teams work together. Writing and running functional tests could achieve this goal. The problem with functional tests is that they are coarse grained and test only a full use case; we lose the benefits of fine-grained unit testing. We will not be able to test as many cases with functional tests as we can with unit tests.

Mock objects have other disadvantages. There are many mock objects to set up, for example, which may prove to be non-negligible overhead—the operating costs of managing all these mocks may be significant. Obviously, the cleaner the code (small and focused methods) is, the easier tests are to set up.

Another important drawback of mock objects is that to set up a test, we usually must know exactly how the mocked API behaves, which may require some knowledge of a domain outside our own. We know the behavior of our own API, but that may not be the case with other APIs, such as the Servlet API that the engineers at Tested Data Systems need to use for the online shop. Even though all containers of a given type implement the same API, not all containers behave the same way, so we may have to deal with bugs, tricks, and hacks for various third-party libraries in any project.

To wrap up this section, here are the pros and cons of unit testing with mock objects:

- *Pros:*
 - Do not require a running container to execute tests
 - Are quick to set up and run
 - Allow fine-grained unit testing
- *Cons:*
 - Do not test interactions with the container or between the components
 - Do not test the deployment of components
 - Require good knowledge of the API to mock, which can be difficult (especially for external libraries)
 - Do not provide confidence that the code will run in the target container
 - Offer more fine-grained testing, which may lead to testing code being swamped with interfaces
 - Like stubs, require maintenance when the code changes

9.4.3 In-container testing evaluation

So far, we have examined the advantages of in-container unit testing. This approach also has the disadvantages discussed in the following sections.

SPECIFIC TOOLS REQUIRED

A major drawback is that although the concept is generic, the tools that implement in-container unit testing are specific to the tested API, such as Jetty or Tomcat for servlets and WildFly for EJBs. With mock objects, because the concept is generic, you can test almost any API.

POOR IDE SUPPORT

A significant drawback of most in-container testing frameworks is the lack of good IDE integration. In most cases, you can use Maven or Gradle to execute tests in an embedded container, which also allows you to run a build in a continuous integration server (CIS; see chapter 13). Alternatively, IDEs can execute tests that use mock objects as normal JUnit tests.

In-container testing falls in the category of integration testing, which means you do not need to execute your in-container tests as often as normal unit tests and will most likely run them in a CIS, alleviating the need for IDE integration.

LONGER EXECUTION TIME

Another issue is performance. For a test to run in a container, you need to start and manage the container, which can be time consuming. The overhead in time and memory depends on the container. Startup overhead is not limited to the container. If a unit test hits a database, for example, the database must be in an expected state before the test starts. In terms of execution time, integration unit tests cost more than mock objects; consequently, you may not run them as often as unit tests.

COMPLEX CONFIGURATION

The biggest drawback of in-container testing is that the tests are complex to configure. Because the application and its tests run in a container, your application must be packaged (usually as a war or ear file) and deployed to the container. Then you must start the container and run the tests.

On the other hand, because you must perform the same tasks for production, it is a best practice to automate this process as part of the build and reuse it for testing purposes. As one of the most complex tasks of a Java EE project, providing automation for packaging and deployment becomes a win-win situation. The need to provide in-container testing drives the creation of this automated process at the beginning of the project, which facilitates continuous integration.

To further this goal, most in-container testing frameworks include support for build tools such as Maven and Gradle and allow the embedded startup of the container. This support helps hide the complexity involved in building various runtime artifacts, running tests, and gathering reports.

Taking into consideration the evaluation we have provided for in-container testing, the next section introduces Arquillian as a container-agnostic integration testing framework for Java EE.

9.5 Testing with Arquillian

Arquillian (<http://arquillian.org>) is a testing framework for Java that uses JUnit to execute test cases against a Java container. The Arquillian framework is broken into three major sections:

- Test runners (JUnit, in our case)
- Containers (WildFly, Tomcat, GlassFish, Jetty, and so on)
- Test enrichers (the injection of container resources and beans directly into the test class)

Despite the lack of integration with JUnit 5 (Arquillian does not have a JUnit 5 extension, at least when this chapter was written), it is very popular and has been largely adopted in projects up to JUnit 4. It greatly eases the task of managing containers, deployments, and framework initializations. On the other hand, as Arquillian is testing Java EE applications, its use in our examples requires some basic knowledge, in particular about Contexts and Dependency Injection (CDI)—a Java EE standard for the inversion of control design pattern.

ShrinkWrap is an external dependency used with Arquillian and a simple way to create archives in Java. Using the fluent ShrinkWrap API, developers at Tested Data Systems can assemble jar, war, and ear files to be deployed directly by Arquillian during testing. Such files are archives that may contain all the classes needed to run an application. ShrinkWrap helps define the deployments and the descriptors to be loaded to the Java container being tested against.

One project under development at Tested Data Systems is a flight-management application. The application creates and sets up flights; it also adds and removes passengers. The engineers want to test the integration between two classes: `Passenger` and `Flight`. The test will check whether each passenger can be added to or removed from a flight correctly, as well as whether the number of passengers exceeds the number of seats. (Nobody wants to get on a flight without having a seat!) The following two listings show the `Passenger` and `Flight` classes, as well as their logic.

Listing 9.3 Passenger class

```
public class Passenger {
    private String identifier;    1
    private String name;
    public Passenger(String identifier, String name) {
        this.identifier = identifier;    2
        this.name = name;
    }
    public String getIdentifier() {
        return identifier;
    }
    public String getName() {    3
    }
}
```

```

        return name;
    }

    @Override
    public String toString() {
        return "Passenger " + getName() +
            " with identifier: " + getIdentifier();
    }
}

```

In this listing:

- We provide two fields—`identifier` and `name`—to describe a Passenger ①.
- We provide a constructor with these two parameters ②.
- We provide two getters ③ and the overridden `toString` method ④.

Listing 9.4 Flight class

```

public class Flight {

    private String flightNumber;
    private int seats;
    Set<Passenger> passengers = new HashSet<>();

    public Flight(String flightNumber, int seats) {
        this.flightNumber = flightNumber;
        this.seats = seats;
    }

    public String getFlightNumber() {
        return flightNumber;
    }

    public int getSeats() {
        return seats;
    }

    public int getNumberOfPassengers () {
        return passengers.size();
    }

    public void setSeats(int seats) {
        if(passengers.size() > seats) {
            throw new RuntimeException(
                "Cannot reduce seats under the number of existing passengers!");
        }
        this.seats = seats;
    }

    public boolean addPassenger(Passenger passenger) {
        if(passengers.size() >= seats) {
            throw new RuntimeException(
                "Cannot add more passengers than the capacity of the flight!");
        }
        return passengers.add(passenger);
    }
}

```

```

public boolean removePassenger(Passenger passenger) {
    return passengers.remove(passenger);
}

@Override
public String toString() {
    return "Flight " + getFlightNumber();
}

}

```

In this listing:

- We provide three fields—flightNumber, seats, and passengers—to describe a Flight ①.
- We provide a constructor with the first two mentioned parameters ②.
- We provide three getters ③ and one setter ④ to address the fields we have defined. The setter on the seats field checks that the value of the field is not less than the number of existing passengers.
- The addPassenger method adds a passenger to the flight. It also compares the number of passengers with the number of seats so the flight will not be overbooked ⑤.
- We provide the capability to remove a passenger from a flight ⑥.
- We override the `toString` method ⑦.

The next listing describes the 20 passengers on the flight by identifier and name. The list is stored in a CSV file.

Listing 9.5 flights_information.csv file

```

1236789; John Smith
9006789; Jane Underwood
1236790; James Perkins
9006790; Mary Calderon
1236791; Noah Graves
9006791; Jake Chavez
1236792; Oliver Aguilar
9006792; Emma McCann
1236793; Margaret Knight
9006793; Amelia Curry
1236794; Jack Vaughn
9006794; Liam Lewis
1236795; Olivia Reyes
9006795; Samantha Poole
1236796; Patricia Jordan
9006796; Robert Sherman
1236797; Mason Burton
9006797; Harry Christensen
1236798; Jennifer Mills
9006798; Sophia Graham

```

The next listing implements the `FlightBuilderUtil` class, which parses the CSV file and populates the flight with the corresponding passengers. Thus, the code brings the information from an external file to the memory of the application.

Listing 9.6 FlightBuilderUtil class

```
[...]
public class FlightBuilderUtil {

    public static Flight buildFlightFromCsv() throws IOException {
        Flight flight = new Flight("AA1234", 20);
        try(BufferedReader reader =
            new BufferedReader(new FileReader(
                "src/test/resources/flights_information.csv"))){
            String line = null;
            do {
                line = reader.readLine();
                if (line != null) {
                    String[] passengerString = line.toString().split(";");
                    Passenger passenger =
                        new Passenger(passengerString[0].trim(),
                                      passengerString[1].trim());
                    flight.addPassenger(passenger);
                }
            } while (line != null);
        }
        return flight;
    }
}
```

The code is annotated with numbered callouts: 1 points to the creation of a `Flight` object; 2 points to the opening of a CSV file using a `BufferedReader`; 3 points to the reading of a line from the file; 4 points to the splitting of the line into an array; 5 points to the creation of a `Passenger` object from the array elements; 6 points to the addition of the passenger to the flight; 7 points to the final return statement of the method.

In this listing:

- We create a flight ①.
- We open the CSV file to parse ②.
- We read line by line ③, split each line ④, create a passenger based on the information that has been read ⑤, and add the passenger to the flight ⑥.
- We return the fully populated flight from the method ⑦.

So far, all the classes that have been implemented for the development of the flight and passengers are pure Java classes; no particular framework and technology have been used. Arquillian is a testing framework that executes test cases against a Java container, so it requires some understanding of notions related to Java EE and CDI. Because it is a widely used framework for integration testing, we have decided to introduce it here; we'll explain the most important ideas so that you can quickly adopt it within your projects.

Arquillian abstracts the container or application startup logic from the unit tests; it drives a deployment runtime paradigm with the application, allowing the deployment

of the program to a Java EE application server. Therefore, the framework is ideal for the in-container testing implemented in this chapter. In addition, it eliminates the need for specific tools that implement in-container testing.

Arquillian deploys the application to the targeted runtime to execute test cases. The targeted runtime can be an application server, embedded or managed. We need to add the following dependencies to the Maven pom.xml configuration file to work with Arquillian.

Listing 9.7 Required pom.xml dependencies

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.arquillian</groupId> 1
            <artifactId>arquillian-bom</artifactId>
            <version>1.4.0.Final</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.jboss.spec</groupId> 2
        <artifactId>jboss-javaee-7.0</artifactId>
        <version>1.0.3.Final</version>
        <type>pom</type>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.vintage</groupId> 3
        <artifactId>junit-vintage-engine</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.arquillian.junit</groupId> 4
        <artifactId>arquillian-junit-container</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.arquillian.container</groupId>
        <artifactId>arquillian-weld-ee-embedded-1.1</artifactId> 5
        <version>1.0.0.CR9</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.weld</groupId>
        <artifactId>weld-core</artifactId>
        <version>2.3.5.Final</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

This listing adds

- The Arquillian API dependency ①.
- The Java EE 7 API dependency ②.
- The JUnit Vintage Engine dependency ③. As mentioned earlier, at least for the moment, Arquillian is not integrated with JUnit 5. Because Arquillian lacks a JUnit 5 extension, we'll have to use the JUnit 4 dependencies and annotations to run our tests.
- The Arquillian JUnit integration dependency ④.
- The container adapter dependencies ⑤ ⑥. To execute our tests against a container, we must include the dependencies that correspond to that container. This requirement demonstrates one of the strengths of Arquillian: it abstracts the container from the unit tests and is not tightly coupled to specific tools that implement in-container unit testing.

An Arquillian test, as implemented in the following listing, looks just like a unit test, with some additions. The test is named `FlightWithPassengersTest` to show the goal of the integration testing between the two classes.

Listing 9.8 `FlightWithPassengersTest` class

```
[...]
@RunWith(Arquillian.class)
public class FlightWithPassengersTest {
    
    @Deployment
    public static JavaArchive createDeployment() {
        return ShrinkWrap.create(JavaArchive.class)
            .addClasses(Passenger.class, Flight.class)
            .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }

    
    @Inject
    Flight flight;
    
    @Test(expected = RuntimeException.class)
    public void testNumberOfSeatsCannotBeExceeded() throws IOException {
        
        assertEquals(20, flight.getNumberOfPassengers());
        flight.addPassenger(new Passenger("1247890", "Michael Johnson"));
    }

    
    @Test
    public void testAddRemovePassengers() throws IOException {
        flight.setSeats(21);
        Passenger additionalPassenger =
            new Passenger("1247890", "Michael Johnson");
        flight.addPassenger(additionalPassenger);
        assertEquals(21, flight.getNumberOfPassengers());
        flight.removePassenger(additionalPassenger);
        assertEquals(20, flight.getNumberOfPassengers());
        assertEquals(21, flight.getSeats());
    }
}
```

As this listing shows, an Arquillian test case must have three things:

- A `@RunWith(Arquillian.class)` annotation on the class ①. The `@RunWith` annotation tells JUnit to use Arquillian as the test controller.
- A public static method annotated with `@Deployment` that returns a Shrink-Wrap archive ②. The purpose of the test archive is to isolate the classes and resources that the test needs. The archive is defined with ShrinkWrap. The microdeployment strategy lets us focus on precisely the classes we want to test. As a result, the test remains very lean and manageable. For the moment, we have included only the `Passenger` and the `Flight` classes. We try to inject a `Flight` object as a class member using the CDI `@Inject` annotation ③. The `@Inject` annotation allows us to define injection points inside classes. In this case, `@Inject` instructs CDI to inject into the test a reference to an object of type `Flight`.
- At least one method annotated with `@Test` ④ and ⑤. Arquillian looks for a public static method with the `@Deployment` annotation to retrieve the test archive. Then each `@Test`-annotated method is run inside the container environment.

When the ShrinkWrap archive is deployed to the server, it becomes a real archive. The container has no knowledge that the archive was packaged by ShrinkWrap.

We have provided the infrastructure for using Arquillian in our project, and now we'll run our integration tests with its help! But we get an error (figure 9.2).

```
org.jboss.weld.exceptions.DeploymentException: WELD-001408: Unsatisfied dependencies for type Flight with qualifiers @Default
  at injection point [BackedAnnotatedField] @Inject com.manning.junitbook.ch09.airport.FlightWithPassengersTest.flight
  at com.manning.junitbook.ch09.airport.FlightWithPassengersTest.flight(FlightWithPassengersTest.java:0)
```

Figure 9.2 The result of running `FlightWithPassengersTest`

The error says `Unsatisfied dependencies for type Flight with qualifiers @Default`. It means the container is trying to inject the dependency, as it has been instructed to do through the CDI `@Inject` annotation, but it is unsatisfied. Why? What have the developers from Tested Data Systems missed? The `Flight` class provides only a constructor with arguments, and it has no default constructor to be used by the container for the creation of the object. The container does not know how to invoke the constructor with parameters and which parameters to pass to it to create the `Flight` object that must be injected.

What is the solution in this case? Java EE offers producer methods that are designed to inject objects that require custom initialization. The solution fixes the issue and is easy to put into practice, even by a junior developer.

Listing 9.9 FlightProducer class

```
[...]
public class FlightProducer {

    @Produces
    public Flight createFlight() throws IOException {
        return FlightBuilderUtil.buildFlightFromCsv();
    }
}
```

In this listing, we create the `FlightProducer` class with the `createFlight` method, which invokes `FlightBuilderUtil.buildFlightFromCsv()`. We can use such a method to inject objects that require custom initialization; in this case, we are injecting a flight that has been configured based on the CSV file. We annotated the `createFlight` method with `@Produces`, which is also a Java EE annotation. The container will automatically invoke this method to create the configured flight; then it injects the method into the `Flight` field, annotated with `@Inject` from the `FlightWithPassengersTest` class.

Next we add the `FlightProducer` class to the `ShrinkWrap` archive.

Listing 9.10 Modified deployment method from FlightWithPassengersTest

```
[...]
@Deployment
public static JavaArchive createDeployment() {
    return ShrinkWrap.create(JavaArchive.class)
        .addClasses(Passenger.class, Flight.class,
                    FlightProducer.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
}
```

If we run the tests now, they are green. The container injected the correctly configured flight (figure 9.3).

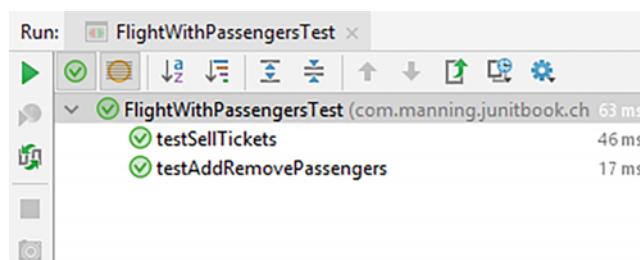


Figure 9.3 The result of running `FlightWithPassengersTest` after introducing the producer method

Considering the advantages of in-container testing, which eliminates tight coupling with specific tools, it is likely to receive more attention and attain greater applicability inside Tested Data Systems. The developers are waiting for one more thing: the creation of a JUnit 5 extension. (We are looking forward to it as well, so fingers crossed!)

Chapter 10 starts part 3 of this book by taking a deeper look at the integration of JUnit into the build process with Maven.

Summary

This chapter has covered the following:

- Analyzing the limitations of unit testing, including limitations of using mock objects: they cannot use objects whose implementation is provided by a container, they require a lot of code, the test expectations must continuously change to match the code evolution, and they do not provide an isolated running environment to run.
- Examining the need for and the steps to in-container testing: running the test cases where the objects actually live, in the container itself.
- Evaluating testing using stubs, mock objects, and in-container testing, and comparing their advantages and drawbacks.
- Working with Arquillian, a container-agnostic integration testing framework for Java EE, and implementing integration tests with its help. Despite its current lack of integration with JUnit 5 (it is missing a JUnit 5 extension), it has been used for many years in Java EE projects, as it uses JUnit to execute test cases against a Java container.

Part 3

Working with JUnit 5 and other tools

This part of the book deals with very important aspects of the development of every project: the build and the IDEs. The importance of the build is reconsidered more and more these days, especially in large projects. Also, an IDE that is comfortable for the developer strongly contributes to the speed of development and local testing. For these reasons, we dedicate a whole part of the book to integration of JUnit 5, build tools, and IDEs.

Chapter 10 provides a very quick introduction to Maven and its terminology. It shows you how to include the execution of your tests in the Maven build life cycle and how to produce nice HTML reports by means of some of the Maven plugins. Chapter 11 guides you through the same concepts, this time by means of another popular tool called Gradle. In chapter 12, you investigate the way that a developer may work with JUnit 5 by using the most popular IDEs today: IntelliJ IDEA, Eclipse, and NetBeans. Chapter 13 is devoted to continuous integration tools. This practice, which is highly recommended by extreme programmers, helps you maintain a code repository and automate the build on it. CI is a must for building large projects that depend on many other projects that change often (like any open source projects).

10

Running JUnit tests from Maven 3

This chapter covers

- Creating a Maven project from scratch
- Testing the Maven project with JUnit 5
- Using Maven plugins
- Using the Maven Surefire plugin

The conventional view serves to protect us from the painful job of thinking.

—John Kenneth Galbraith

This chapter discusses a common build system tool called Maven. In the previous chapters, with provided Maven projects, you needed only to look at some external dependencies, run some simple commands, or run the tests from inside the IDE. This chapter gives you a brief introduction to the Maven build system, which will be very useful if you need a systematic way to start your tests.

Maven addresses two aspects of building software. First, it describes how software is built; then, it describes the needed dependencies. Unlike earlier tools, such as Apache Ant, it uses conventions for the build procedure, and only exceptions need to be written down. It relies on an XML file to describe its full configuration—most

important are the meta-information about the software project being built, the needed dependencies on other external components, and the required plugins.

By the end of this chapter, you will know how to build Java projects with Maven, including managing their dependencies, executing JUnit tests, and generating JUnit reports. For basic Maven concepts and how to set up Maven, see appendix A.

10.1 Setting up a Maven project

If you have installed Maven, you are ready to use it. The first time you execute a plugin, your internet connection must be on, because Maven automatically downloads from the web all the third-party .jar files that the plugin requires.

First, create the C:\junitbook\ folder. This is your work directory and where you will set up the Maven examples. Type the following on the command line:

```
mvn archetype:generate -DgroupId=com.manning.junitbook  
-DartifactId=maven-sampling  
-DarchetypeArtifactId=maven-artifact-mojo
```

Press Enter, wait for the appropriate artifacts to be downloaded, and accept the default options. You should see a folder named maven-sampling being created. If you open the new project in IntelliJ IDEA, its structure should look like figure 10.1 (just note that the .idea folder is created not by Maven, but by the IDE).

What happened here? We invoked `maven-archetype-plugin` from the command line and told it to generate a new project from scratch with the given parameters. As a result, this Maven plugin created a new project with a new folder structure, following the convention for that folder structure. Further, it created a sample `App.java` class with the `main` method and a corresponding `AppTest.java` file that is a

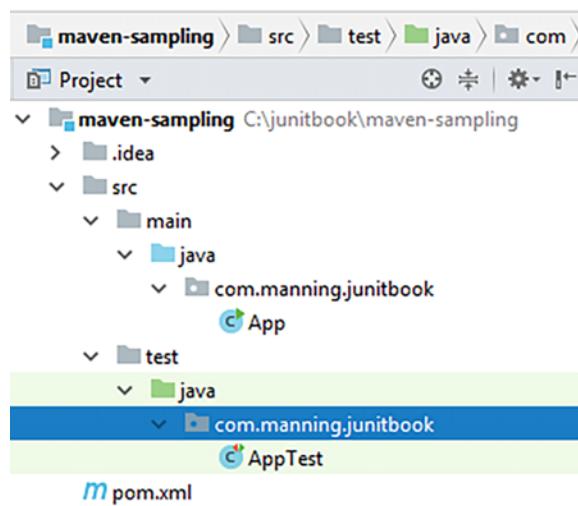


Figure 10.1 Folder structure after the project is created

unit test for our application. After looking at this folder structure, you should understand what files are in `src/main/java` and what files are in `src/test/java`.

The Maven plugin also generated a `pom.xml` file for us. Following are some parts of the descriptor.

Listing 10.1 `pom.xml` for the `maven-sampling` project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.manning.junitbook</groupId>
    <artifactId>maven-sampling</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>maven-sampling</name>
    <!-- FIXME change it to the project's website -->
    <url>http://www.example.com</url>
    [...]
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    [...]
</project>
```

This code is the build descriptor for our project. It starts with a global `<project>` tag with the appropriate namespaces. Inside it, we place all of our components:

- `modelVersion`—Represents the version of the model of the pom being used. Currently, the only supported version is 4.0.0.
- `groupId`—Acts as the Java packaging in the filesystem, grouping different projects from one organization, company, or group of people. We provide this value in the command line when invoking Maven.
- `artifactId`—Represents the name that the project is known by. Again, the value here is the one we specified in the command line.
- `version`—Identifies the current version of our project (or project artifact). The `SNAPSHOT` ending indicates that this artifact is still in development mode; we have not released it yet.
- `dependencies`—Lists our dependencies.

Now that we have our project descriptor, we can improve it a little (listing 10.2). First, we need to change the version of the JUnit dependency, because we are using JUnit Jupiter 5.4.2, and the version that the plugin generated is 4.11. After that, we can

insert some additional information to make the pom.xml more descriptive, such as a `developers` section. This information not only makes the pom.xml more descriptive but also will be included later when we build the website.

Listing 10.2 Changes and additions to pom.xml

```

<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<developers>
    <developer>
        <name>Catalin Tudose</name>
        <id>ctudose</id>
        <organization>Manning</organization>
        <roles>
            <role>Java Developer</role>
        </roles>
    </developer>
    <developer>
        <name>Petar Tahchiev</name>
        <id>ptahchiev</id>
        <organization>Apache Software Foundation</organization>
        <roles>
            <role>Java Developer</role>
        </roles>
    </developer>
</developers>

```

We also specify `organization`, `description`, and `inceptionYear`.

Listing 10.3 Description elements in pom.xml

```

<description>
    "JUnit in Action III" book, the sample project for the "Running JUnit
    tests from Maven" chapter.
</description>
<organization>
    <name>Manning Publications</name>
    <url>http://manning.com/</url>
</organization>
<inceptionYear>2019</inceptionYear>

```

Now we can start developing our software. What if you want to use a Java IDE other than IntelliJ IDEA, such as Eclipse? No problem. Maven offers additional plugins that let you import the project into your favorite IDE. To use Eclipse, open a terminal and navigate to the directory that contains the project descriptor (pom.xml). Then type the following and press Enter:

```
mvn eclipse:eclipse
```

This command invokes maven-eclipse-plugin, which, after downloading the necessary artifacts, produces the two files (.project and .classpath) that Eclipse needs to recognize the project as an Eclipse project. Next, you can import your project into Eclipse. All the dependencies listed in the pom.xml file are added to the project (figure 10.2).

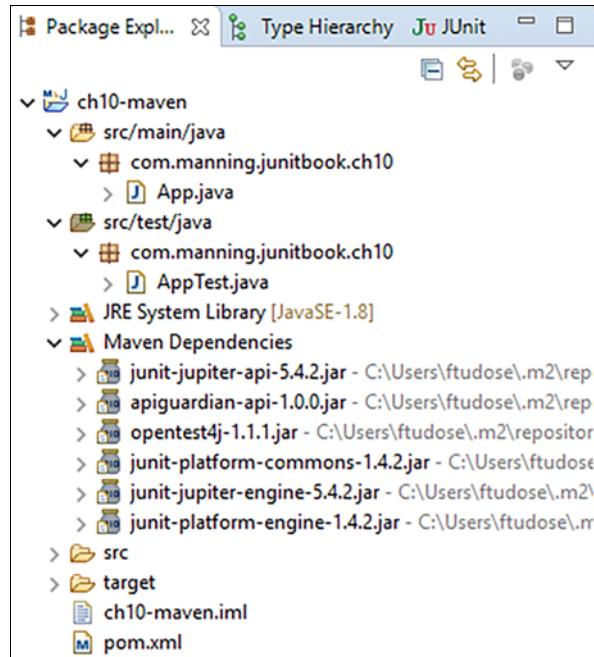


Figure 10.2 The imported project, with the Maven folders structure and needed dependencies (including the JUnit 5 ones)

Developers who use IntelliJ IDEA can import the project directly (refer to figure 10.1), as this IDE invokes the Maven plugin when such a project is open.

10.2 Using the Maven plugins

We have seen what Maven is and how to use it to start a project from scratch. We have also seen how to generate the project documentation and how to import a project into Eclipse and IntelliJ.

Whenever you would like to clean the project from previous activities, execute the following command:

```
mvn clean
```

This command causes Maven to go through the clean phase and invoke all the plugins that are attached to this phase—in particular, `maven-clean-plugin`, which deletes the `target/` folder where your generated site resides.

10.2.1 *Maven compiler plugin*

Like any other build system, Maven is supposed to build our projects (compile our software and package in an archive). Every task in Maven is performed by an appropriate plugin, the configuration of which is in the `<plugins>` section of the project descriptor. To compile the source code, all you need to do is invoke the `compile` phase on the command line

```
mvn compile
```

which causes Maven to execute all the plugins attached to the `compile` phase (in particular, it will invoke `maven-compiler-plugin`). But before invoking the `compile` phase, as already discussed, Maven goes through the `validate` phase, downloads all the dependencies listed in the `pom.xml` file, and includes them in the classpath of the project. When the compilation process is complete, you can go to the `target/classes/` folder and see the compiled classes there.

Next, we'll try to configure the compiler plugin, escaping from the convention-over-configuration principle. So far, the conventional compiler plugin has worked well. But what if we need to include the `-source` and `-target` attributes in the compiler invocation to generate class files for the specific version of the JVM? We can add the following code to the `<build>` section of our build file.

Listing 10.4 Configuring `maven-compiler-plugin`

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This code is a general way to configure each of our Maven plugins: enter a `<plugins>` section in our `<build>` section. There, we list each plugin that we want to configure—in this case, the `maven-compiler-plugin`. We need to enter the configuration parameters in the plugin configuration section. We can get a list of parameters for every plugin from the Maven website. Without the `<source>` and `<target>` parameters, the Java version to be used will be 5, which is quite old.

As we see in the declaration of the `maven-compiler-plugin` in listing 10.4, we have not set the `groupId` parameter. `maven-compiler-plugin` is one of the core Maven plugins that has an `org.apache.maven.plugins` `groupId`, and plugins with such a `groupId` can skip the `groupId` parameter.

10.2.2 Maven Surefire plugin

To process the unit tests from our project, Maven uses (of course) a plugin. The Maven plugin that executes the unit tests is called `maven-surefire-plugin`. The Surefire plugin executes the unit tests for our code, but these unit tests are not necessarily JUnit tests.

There are other frameworks for unit testing, and the Surefire plugin can execute their tests, too. The following listing shows the configuration of the Maven Surefire plugin.

Listing 10.5 The Maven Surefire plugin

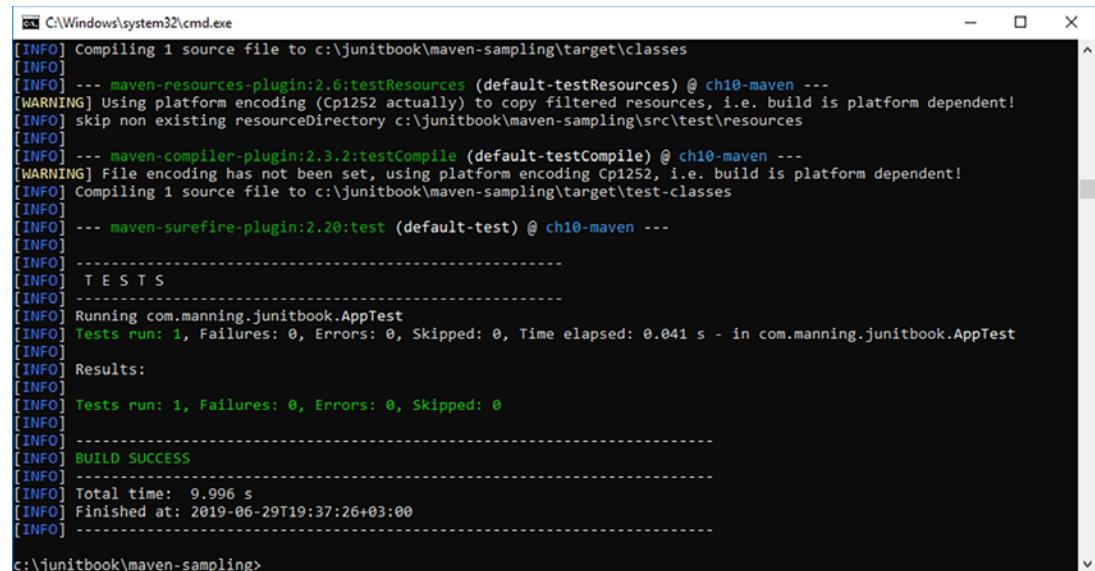
```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
    </plugin>
  </plugins>
</build>
```

The conventional way to start `maven-surefire-plugin` is very simple: invoke the Maven test phase. This way, Maven first invokes all the phases that are supposed to come before the test phase (validate and compile) and then invokes all the plugins that are attached to the test phase, thus invoking `maven-surefire-plugin`. So by calling

```
mvn clean test
```

Maven first cleans the `target/` directory, then compiles the source code and the tests, and finally lets JUnit 5 execute all the tests that are in the `src/test/java` directory (remember the convention). The output should be similar to figure 10.3.

That's great, but what if we want to execute only a single test case? This execution is unconventional, so we need to configure `maven-surefire-plugin` to do it.



```

C:\Windows\system32\cmd.exe
[INFO] Compiling 1 source file to c:\junitbook\maven-sampling\target\classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ ch10-maven ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory c:\junitbook\maven-sampling\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @ ch10-maven ---
[WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is platform dependent!
[INFO] Compiling 1 source file to c:\junitbook\maven-sampling\target\test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.20:test (default-test) @ ch10-maven ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.manning.junitbook.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.041 s - in com.manning.junitbook.AppTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 9.996 s
[INFO] Finished at: 2019-06-29T19:37:26+03:00
[INFO]
c:\junitbook\maven-sampling>

```

Figure 10.3 Execution of JUnit tests with Maven 3

Ideally, a parameter for the plugin allows us to specify the pattern of test cases that we want to execute. We configure the Surefire plugin in exactly the same way that we configure the compiler plugin.

Listing 10.6 Configuration of maven-surefire-plugin

```

<build>
  <plugins>
    [...]
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
      <configuration>
        <includes>**/*Test.java</includes>
      </configuration>
      [...]
    </plugin>
    [...]
  </plugins>
</build>

```

We have specified the `includes` parameter to denote that we want only the test cases matching the given pattern to be executed. But how do we know what parameters `maven-surefire-plugin` accepts? No one knows all the parameters by heart, of course, but we can always consult the `maven-surefire-plugin` documentation (and any other plugin documentation) on the Maven website (<http://maven.apache.org>).

The next step is generating documentation for the project. But wait a second—how are we supposed to do that with no files to generate the documentation from? This is another one of Maven's great benefits: with a little configuration and description, we can produce a fully functional website skeleton.

First, add the maven-site-plugin to the Maven pom.xml configuration file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-site-plugin</artifactId>
  <version>3.7.1</version>
</plugin>
```

Then, type

```
mvn site
```

on the command line where the pom.xml file is. Maven should start downloading its plugins; and after their successful installation, it produces the website shown in figure 10.4.

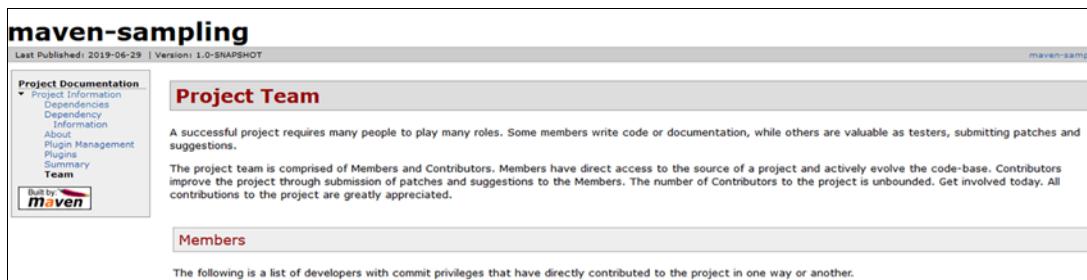


Figure 10.4 Maven produces a website documenting the project.

This website is generated in the Maven build directory—another convention. Maven uses the target/ folder for all the needs of the build itself. The source code is compiled in the target/classes/ folder, and the documentation is generated in target/site/.

Notice that the website is more like a skeleton of a website. Remember that we entered a small amount of data; we could enter more data and web pages in src/site, and Maven would include them on the website, thus generating full-blown documentation.

10.2.3 Generating HTML JUnit reports with Maven

Maven can generate reports from the JUnit XML output. Because, by default, Maven produces plain-text and XML-formatted output (by convention, they go in the target/Surefire-reports/ folder), we do not need any other configuration to produce HTML Surefire reports for the JUnit tests.

As you may already guess, the job of producing these reports is done by a Maven plugin. The name of the plugin is `maven-surefire-report-plugin`, and by default, it is not attached to any of the core phases that we already know. (Many people do not need HTML reports every time they build software.) We can't invoke the plugin by running a certain phase (as we did with both the compiler plugin and the Surefire plugin). Instead, we have to call it directly from the command line:

```
mvn surefire-report:report
```

Maven tries to compile the source files and the test cases, and then it invokes the Surefire plugin to produce the plain-text and XML-formatted output of the tests. After that, the `surefire-report` plugin tries to transform all the XML from the `target/surefire-reports/` directory into an HTML report that will be placed in the `target/site` directory. (Remember that this is the convention for the folder—to keep all the generated documentation of the project—and that the HTML reports are considered to be documentation.) If we open the generated HTML report, it looks something like figure 10.5.

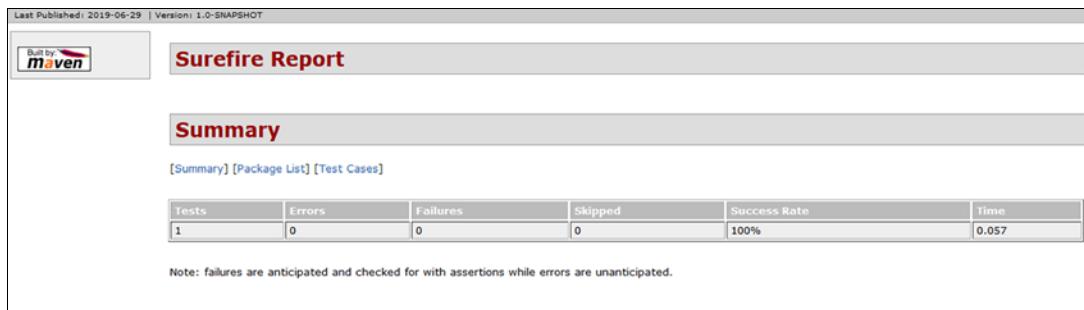


Figure 10.5 HTML report from the `maven-surefire-report` plugin

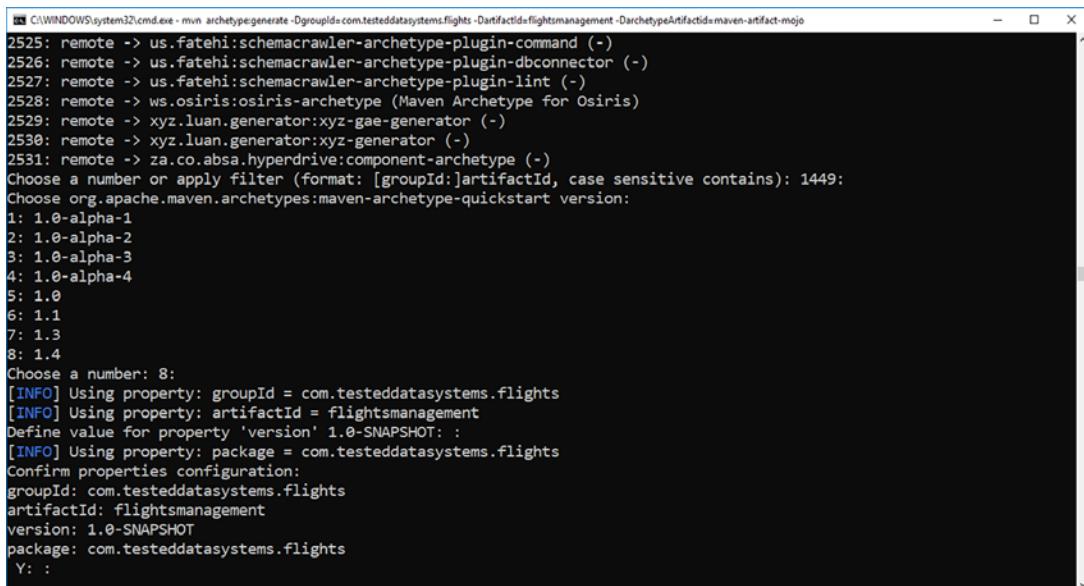
10.3 Putting it all together

This section demonstrates all the steps involved in creating a JUnit 5 project that is managed by Maven. As discussed in chapter 9, one of the projects under development at Tested Data Systems is a flight-management application. To start such an application, George, a project developer, creates the `C:\Work\` folder, which will become the working directory and the one where he will set up the Maven project. He types the following on the command line:

```
mvn archetype:generate -DgroupId=com.testeddatasystems.flights
-DartifactId=flightsmanagement -DarchetypeArtifactId=maven-artifact-mojo
```

After pressing Enter and waiting for the appropriate artifacts to be downloaded, George accepts the default options (figure 10.6). Figure 10.7 shows the structure of

the project: `src/main/java/` is the Maven folder in which the Java code for the project resides, and `src/test/java` contains the unit tests. The `.idea` folder and the `flightsmanagement.iml` file are created by the IDE.



```
2525: remote -> us.fatehi:schemacrawler-archetype-plugin-command (-)
2526: remote -> us.fatehi:schemacrawler-archetype-plugin-dbconnector (-)
2527: remote -> us.fatehi:schemacrawler-archetype-plugin-lint (-)
2528: remote -> ws.osiris:osiris-archetype (Maven Archetype for Osiris)
2529: remote -> xyz.luan.generator:xyz-gae-generator (-)
2530: remote -> xyz.luan.generator:xyz-gae-generator (-)
2531: remote -> za.co.absa.hyperdrive:component-archetype (-)
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 1449:
Choose org.apache.maven.archetypes:maven-archetype-quickstart version:
1: 1.0-alpha-1
2: 1.0-alpha-2
3: 1.0-alpha-3
4: 1.0-alpha-4
5: 1.0
6: 1.1
7: 1.3
8: 1.4
Choose a number: 8:
[INFO] Using property: groupId = com.testeddatasystems.flights
[INFO] Using property: artifactId = flightsmanagement
Define value for property 'version' 1.0-SNAPSHOT: :
[INFO] Using property: package = com.testeddatasystems.flights
Confirm properties configuration:
groupId: com.testeddatasystems.flights
artifactId: flightsmanagement
version: 1.0-SNAPSHOT
package: com.testeddatasystems.flights
Y: :
```

Figure 10.6 Creating a Maven 3 project and accepting the default options

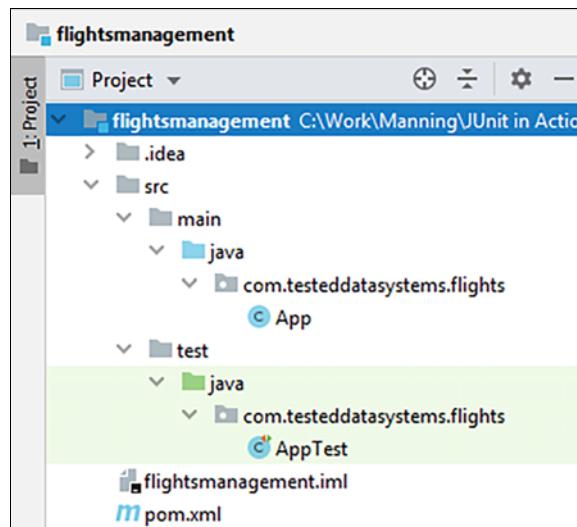


Figure 10.7 The newly created flight-management project

George invokes `maven-archetype-plugin` from the command line and tells it to generate a new project from scratch with the given parameters. As a result, the Maven plugin creates a new project with the default folder structure, as well as a sample `App.java` class with the `main` method and a corresponding `AppTest.java` file that is a unit test for the application. The Maven plugin also generates a `pom.xml` file, as shown in the next listing.

Listing 10.7 `pom.xml` for the flight-management project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.testeddatasystems.flights</groupId>
    <artifactId>flightsmanagement</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name> flightsmanagement </name>
    [...]
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    [...]
</project>
```

This code is the build descriptor for the project. It includes the `modelVersion` (the version of the model of the POM being used, which currently is 4.0.0), the `groupId` (`com.testeddatasystems.flights`), the `artifactId` (the name by which the project is known—`flightsmanagement` in this case), and the `version. 1.0-SNAPSHOT` indicates that the artifact is still in development mode. POM stands for *project object model*; the POM contains information about the project and its configuration, and is the fundamental unit of work in Maven.

George needs to change the version of the JUnit dependency because he is using JUnit Jupiter 5.6.0, and the one that the plugin generated is version 4.11.

Listing 10.8 Changes and additions to `pom.xml`

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
```

```

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.6.0</version>
  <scope>test</scope>
</dependency>
</dependencies>

```

George removes the existing autogenerated App and AppTest classes and instead introduces the two classes that will start his application: Passenger (listing 10.9) and PassengerTest (listing 10.10).

Listing 10.9 Passenger class

```

package com.testeddatasystems.flights;

public class Passenger {

    private String identifier;           1
    private String name;

    public Passenger(String identifier, String name) {
        this.identifier = identifier;
        this.name = name;
    }

    public String getIdentifier() {
        return identifier;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Passenger " + getName() +
               " with identifier: " + getIdentifier();
    }
}

```

The Passenger class contains the following:

- Two fields: identifier and name 1
- A constructor, receiving as arguments identifier and name 2
- Getters for the identifier and name fields 3
- The overridden `toString` method 4

Listing 10.10 PassengerTest class

```

package com.testeddatasystems.flights;
[...]
public class PassengerTest {

```

```

    @Test
    void testPassenger() {
        Passenger passenger =
            new Passenger("123-456-789", "John Smith");
        assertEquals("Passenger John Smith with identifier:
                     123-456-789",
                     passenger.toString());
    }
}

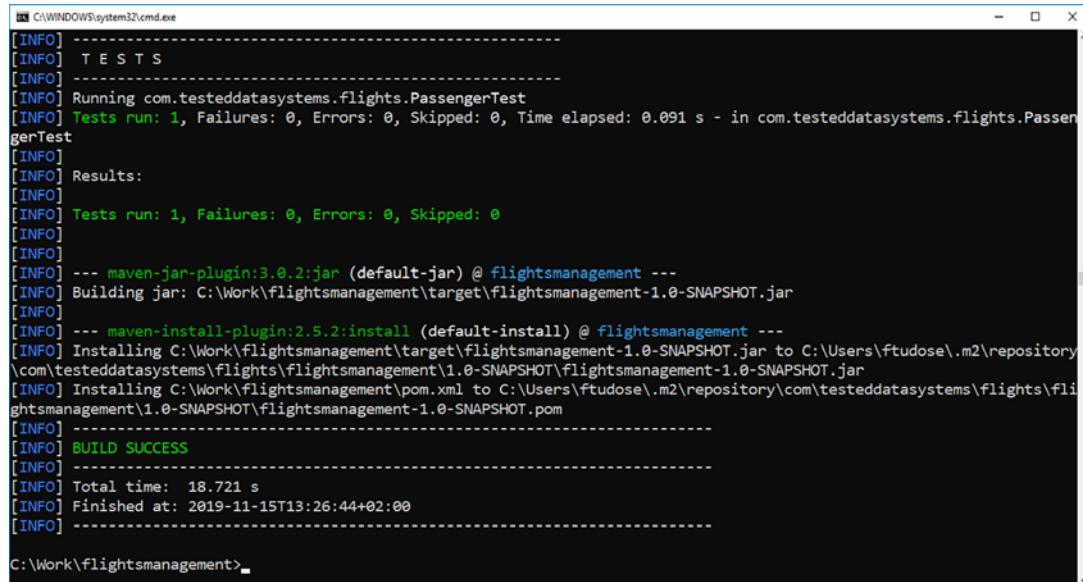
```

1

The PassengerTest class contains one test: `testPassenger`, which verifies the output of the overridden `toString` method ①.

Next, George executes the `mvn clean install` command at the level of the project folder (figure 10.8). This command first cleans the project, removing existing artifacts. Then it compiles the source code of the project, tests the compiled source code (using JUnit 5), packages the compiled code in JAR format (figure 10.9), and installs the package in the local Maven repository (figure 10.10). The local Maven repository is located at `~/.m2/repository/` in UNIX or `C:\Documents and Settings\<UserName>\.m2\repository\` in Windows.

George has created a fully functional Maven project to develop the flight-management application and test it with JUnit 5. From now on, he can continue to add classes and tests and execute the Maven commands to package the application and run the tests.



```

C:\WINDOWS\system32\cmd.exe
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.testeddatasystems.flights.PassengerTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.091 s - in com.testeddatasystems.flights.PassengerTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ flightsmanagement ---
[INFO] Building jar: C:\Work\flightsmanagement\target\flightsmanagement-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ flightsmanagement ---
[INFO] Installing C:\Work\flightsmanagement\target\flightsmanagement-1.0-SNAPSHOT.jar to C:\Users\ftudose\.m2\repository\com\testeddatasystems\flights\flightsmanagement\1.0-SNAPSHOT\flightsmanagement-1.0-SNAPSHOT.jar
[INFO] Installing C:\Work\flightsmanagement\pom.xml to C:\Users\ftudose\.m2\repository\com\testeddatasystems\flights\flightsmanagement\1.0-SNAPSHOT\flightsmanagement-1.0-SNAPSHOT.pom
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 18.721 s
[INFO] Finished at: 2019-11-15T13:26:44+02:00
[INFO] -----
C:\Work\flightsmanagement>_

```

Figure 10.8 Executing the `mvn clean install` command for the Maven project under development

sk (C:) > Work > flightsmanagement > target >			
Name	Date modified	Type	Size
classes	9/16/2019 7:14 PM	File folder	
generated-sources	9/16/2019 7:14 PM	File folder	
generated-test-sources	9/16/2019 7:14 PM	File folder	
maven-archiver	9/16/2019 7:14 PM	File folder	
maven-status	9/16/2019 7:14 PM	File folder	
surefire-reports	9/16/2019 7:14 PM	File folder	
test-classes	9/16/2019 7:14 PM	File folder	
 flightsmanagement-1.0-SNAPSHOT.jar	9/16/2019 7:14 PM	Executable Jar File	3 KB

Figure 10.9 The jar-packaged file in the Maven target folder of the flight-management project

sk (C:) > Users > ftudose > .m2 > repository > com > testit > flights > flightsmanagement > 1.0-SNAPSHOT			
Name	Date modified	Type	Size
 _remote.repositories	9/16/2019 7:14 PM	REPOSITORIES File	1 KB
 flightsmanagement-1.0-SNAPSHOT.jar	9/16/2019 7:14 PM	Executable Jar File	3 KB
 flightsmanagement-1.0-SNAPSHOT.pom	9/16/2019 6:45 PM	POM File	3 KB
 maven-metadata-local.xml	9/16/2019 7:14 PM	XML Document	1 KB

Figure 10.10 The local Maven repository containing the flight-management project's artifacts

10.4 Maven challenges

A lot of people who have used Maven agree that it is really easy to start using and that the idea behind the project is amazing. Things are more challenging when you need to do something unconventional, however.

What is great about Maven is that it sets up a frame for you and constrains you to think inside that frame—to think the Maven way and do things the Maven way. In most cases, Maven will not let you execute any nonsense. It restricts you and shows you the way things need to be done. But these restrictions may be a challenge if you are accustomed to doing things your own way and to having freedom of choice as a build engineer. Chapter 11 shows how to run JUnit tests with another build-automation tool inspired by Maven: Gradle.

Summary

This chapter has covered the following:

- A brief introduction to what Maven is and how to use it in a development environment to build your source code
- How to manage the project dependencies in the Maven style by adding them to the pom.xml file in a declarative way
- Creating a Maven project from scratch, opening it in IntelliJ IDEA, and testing it using JUnit 5
- Comparing how to prepare a Maven JUnit 5 project to be used with both Eclipse and IntelliJ IDEA
- Analyzing Maven plugins: using the compiler plugin, the Surefire plugin to process the unit tests from a project, the site plugin to build a site from a project, and the Surefire report plugin to build reports for the project

11

Running JUnit tests from Gradle 6

This chapter covers

- Introducing Gradle
- Setting up a Gradle project
- Using Gradle plugins
- Creating a Gradle project from scratch and testing it with JUnit 5
- Comparing Gradle and Maven

Mixing one's wines may be a mistake, but old and new wisdom mix admirably.

—Bertolt Brecht

In this chapter, we will analyze the last part of the world of build system tools. Gradle is an open source build-automation system that started with the concepts of Apache Ant and Apache Maven. Instead of the XML form that Apache Maven uses, as you saw in chapter 10, Gradle introduces a domain-specific language (DSL) based on Groovy for declaring the project configuration.

A DSL is a computer language dedicated to addressing a specific application domain. The idea is to have languages whose purpose is to solve problems belonging

to a specific domain. In the case of builds, one of the results of applying the DSL idea is Gradle. Groovy is a Java-syntax-compatible object-oriented programming language that runs on the Java Virtual Machine (JVM).

11.1 *Introducing Gradle*

We'll take a look at various aspects of using Gradle to manage building and testing Java applications, with a focus on testing. We have worked with Maven, and you know from chapter 10 that it uses convention over configuration. Gradle also has a series of building conventions that we can follow when we do the build. This allows other developers who are also using Gradle to easily follow our build's configuration.

The conventions can easily be overridden. As we have explained, the build language of Gradle is a DSL based on Groovy; this makes it easy for developers to configure builds. The DSL replaces the XML approach promoted by Maven. XML has been used for years to store information. It is not surprising that XML is used to configure build files in Apache Ant and Apache Maven, as discussed in chapter 10. Maven came up with the idea of introducing convention over configuration, which was lacking in Ant. And we have discussed the Maven folder structure, support for dependencies, and build life cycles. Gradle has a declarative build language that expresses the intent of the build—this means you tell it *what* you would like to happen, not *how* you would like it to happen.

The engineers from Tested Data Systems Inc. are considering using Gradle for some of their projects. Consequently, they are weighing its capabilities, applying it to some pilot projects, and trying to differentiate it from other possible alternatives: mainly, Maven.

To get a first impression, the engineers at Tested Data Systems consider a simple Maven pom.xml configuration file (as you have previously seen; listing 11.1) and a simple build.gradle file (the default name of the build descriptor in Gradle; listing 11.2).

Listing 11.1 A simple Maven pom.xml file

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.manning.junitbook</groupId>
  <artifactId>example-pom</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <version>5.6.0</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.6.0</version>
```

```
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

Listing 11.2 A simple Gradle build.gradle file

```
plugins {
    // Apply the java plugin to add support for Java
    id 'java'

    // Apply the application plugin to support building a CLI application
    id 'application'
}

repositories {
    // Use jcenter for resolving dependencies.
    // You can declare any Maven/Ivy/file repository here.
    jcenter()
}

dependencies {
    // Use JUnit Jupiter API for testing.
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'

    // Use JUnit Jupiter Engine for testing.
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.6.0'
}

application {
    // Define the main class for the application
    mainClassName = 'com.manning.junitbook.ch11.App'
}

test {
    // Use junit platform for unit tests
    useJUnitPlatform()
}
```

You may be looking at this build.gradle configuration file and wondering what is happening in it. Its purpose is to give you a first taste of the way Gradle works. We'll get into details in this chapter. Additionally, many comments explain what is happening. The DSL approach, one of Gradle's strengths, is largely self-explanatory and relatively easy to maintain and manage. For the Gradle installation procedure and a brief explanation of its central concepts, see appendix B.

11.2 Setting up a Gradle project

It is time now for the engineers at Tested Data Systems to move on in their consideration of using Gradle for some of their projects by creating JUnit 5 pilot projects with Gradle. Oliver is involved in developing such a pilot project. He creates a new folder called `junit5withgradle`, opens a command prompt, and executes the `gradle init` command.

Then, he chooses the following options:

- Select Type of Project to Generate: Application
- Select Implementation Language: Java
- Select Build Script DSL: Groovy
- Select Test Framework: JUnit Jupiter

The result is shown in figure 11.1.

```
cmd C:\Windows\system32\cmd.exe
C:\Work\Manning\junit5withgradle>gradle init

Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 2

Select implementation language:
  1: C++
  2: Groovy
  3: Java
  4: Kotlin
Enter selection (default: Java) [1..4] 3

Select build script DSL:
  1: Groovy
  2: Kotlin
Enter selection (default: Groovy) [1..2] 1

Select test framework:
  1: JUnit 4
  2: TestNG
  3: Spock
  4: JUnit Jupiter
Enter selection (default: JUnit 4) [1..4] 4

Project name (default: junit5withgradle):

Source package (default: junit5withgradle): com.manning.junitbook.ch11

> Task :init
Get more help with your project: https://docs.gradle.org/5.5.1/userguide/tutorial\_java\_projects.html

BUILD SUCCESSFUL in 31s
2 actionable tasks: 2 executed
```

Figure 11.1 The result of executing the `gradle init` command

Because Oliver has initialized a new Java project with Gradle and chosen a few options, Gradle creates the project's folder structure, which follows Maven's structure:

- `src/main/java` contains the Java source code.
- `src/test/java` contains the Java tests.

To start the build, Oliver types `gradle build` at the command-line prompt, in the folder that contains the `build.gradle` file and the project that has just been created.

We'll now take a look at the build.gradle file that is created, explain its content, and demonstrate how Oliver can extend it.

Listing 11.3 Defining repositories in the build.gradle file

```
repositories {  
    // Use jcenter for resolving dependencies.  
    // You can declare any Maven/Ivy/file repository here.  
    jcenter()  
}
```

By default, Gradle uses JCenter as the repository for the applications it manages. JCenter is the largest Java repository in the world. This means whatever is available on Maven Central is available on JCenter as well. Of course, we can specify that the Maven Central repository, our own repository, or multiple repositories should be used. The following listing shows that Oliver has chosen to use the Maven Central repository and the repository from Tested Data Systems, which allows access to proprietary dependencies.

Listing 11.4 Defining two repositories in the build.gradle file

```
repositories {  
    mavenCentral()  
  
    testit {  
        url "https://testeddatasystems.com/repository"  
    }  
}
```

Based on the options Oliver provides, Gradle adds some dependencies to the build.gradle file configuration.

Listing 11.5 Defining dependencies in the build.gradle file

```
dependencies {  
    // This dependency is used by the application.  
    implementation 'com.google.guava:guava:27.1-jre'  
  
    // Use JUnit Jupiter API for testing.  
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'  
  
    // Use JUnit Jupiter Engine for testing.  
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.6.0'  
}
```

The dependency configuration declares the external dependencies that we would like to be downloaded from the repository or repositories in use. It defines the standard configurations listed in table 11.1.

Table 11.1 Standard dependency configurations and their meaning

Standard configuration	Meaning
implementation	The dependencies are required to compile the production source of the project.
runtime	The production classes require the dependencies at runtime. By default, the configuration also includes the compile-time dependencies.
testImplementation	The dependencies are required to compile the test source of the project. By default, the configuration includes compiled production classes and the compile-time dependencies.
testRuntime	The dependencies are required to run the tests. By default, the configuration includes runtime and test compile time dependencies.
runtimeOnly	The dependencies are required only at runtime, not at compile time.
testRuntimeOnly	The dependencies are required only at test runtime, not at test compile time.

Based on the option to create an application that Oliver provides, Gradle also adds the following configuration.

Listing 11.6 Configuring the main class of the application in the build.gradle file

```
application {
    // Define the main class for the application
    mainClassName = 'com.manning.junitbook.ch11.App'
}
```

This defines `com.manning.junitbook.ch11.App` as the main class of the application—the execution entry point.

The code in the next listing enables JUnit Platform support in `build.gradle`. More exactly, it specifies that JUnit Platform should be used to execute the tests—remember that we were allowed to choose among a few platforms when Gradle created the project for us.

Listing 11.7 Enabling JUnit Platform support in the build.gradle file

```
test {
    // Use junit platform for unit tests
    useJUnitPlatform()
}
```

`useJUnitPlatform` can take additional options. For example, we can specify tags to be included or excluded when the tests are executed. The configuration looks something like this.

Listing 11.8 Including and excluding tags to be executed in build.gradle

```
test {
    // Use junit platform for unit tests
    useJUnitPlatform {
        includeTags 'individual'
        excludeTags 'repository'
    }
}
```

These options mean the tests tagged by JUnit 5 as '`'individual'`' will be executed by Gradle, whereas tests tagged as '`'repository'`' will be excluded.

Listing 11.9 Gradle selectively executing differently tagged tests

```
@Tag("individual")
public class CustomerTest {
    ...
}

@Tag("repository")
public class CustomersRepositoryTest {
    ...
}
```

When Gradle creates a project, it also creates a *wrapper*. This wrapper represents a script that invokes a declared Gradle version, first downloading it if it is not available. The script is contained in the `gradlew.bat` or `gradlew` file (depending on the operating system). If you build a project and distribute it to your customers, they will quickly be able to run that project without any manual installation and with the certainty that they will be using exactly the same Gradle version used to create the project.

We can create reports concerning the execution of the tests. When running the `gradle test` command (or `gradlew test`, if we use the wrapper), a report is created in the `build/reports/tests/test` folder. Accessing the `index.html` file, we get the report shown in figure 11.2. We have included in our project the tagged tests introduced in chapter 2 and executed them through Gradle, including the individual tests and excluding the repository tests, as described in the `build.gradle` file.

At the time of writing, there are some known limitations regarding using JUnit 5 with Gradle: classes and tests are still displayed using their names instead of `@DisplayName`. This should be fixed in future releases of Gradle.

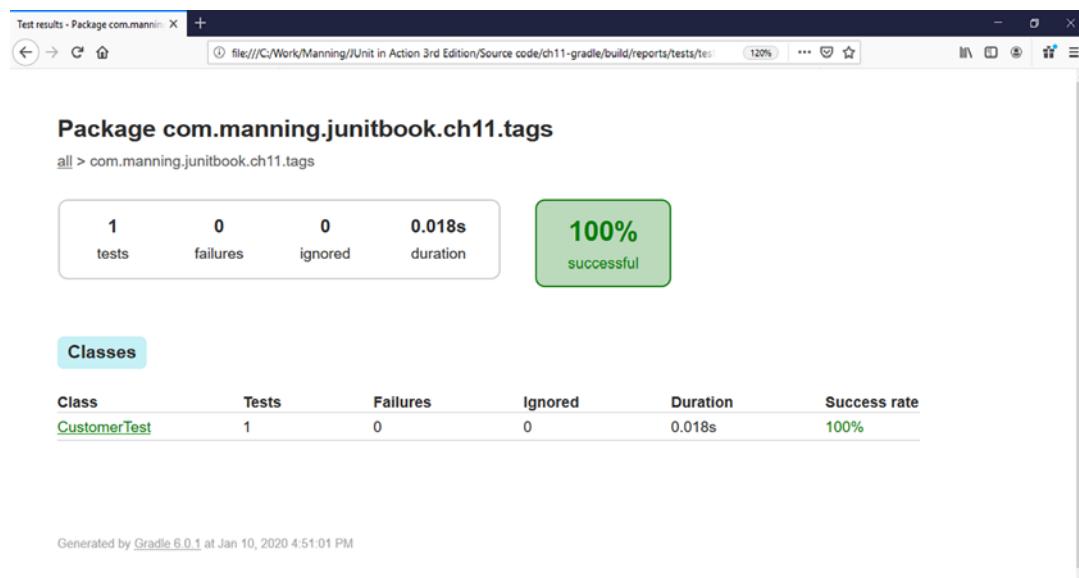


Figure 11.2 The report generated by Gradle includes the execution of `CustomerTest`, tagged as 'individual'.

11.3 Using Gradle plugins

So far, so good—we have seen what Gradle is and how Oliver has used it to start a pilot project at Tested Data Systems from scratch. We have also seen how Gradle is able to manage dependencies by using a DSL.

Oliver would like to add plugins to his project. A Gradle plugin is a set of tasks. A lot of common tasks, such as compiling and setting up source files, are handled by plugins. Applying a plugin to a project means you allow the plugin to extend the project's capabilities.

There are two types of plugins in Gradle: script plugins and binary plugins. Script plugins represent tasks that are defined and can be applied from a script on the local filesystem or at a remote location. Of particular interest for us as Java developers using JUnit 5 are the binary plugins. Each binary plugin is identified by an ID: the core plugins use short names to be applied.

In the following listing, Oliver applies two plugins to the `build.gradle` file.

Listing 11.10 Applying plugins to the `build.gradle` file

```
plugins {
    // Apply the java plugin to add support for Java
    id 'java'

    // Apply the application plugin to add support for building a CLI
    // application
    id 'application'
}
```

In this listing:

- Oliver applies the Java plugin to add support for the programming language—remember that during initialization with the `gradle init` command, Oliver chose Java as the language, and Gradle has added the needed support.
- He chooses “application” as the type of project to generate and adds the plugin that supports building a CLI application.

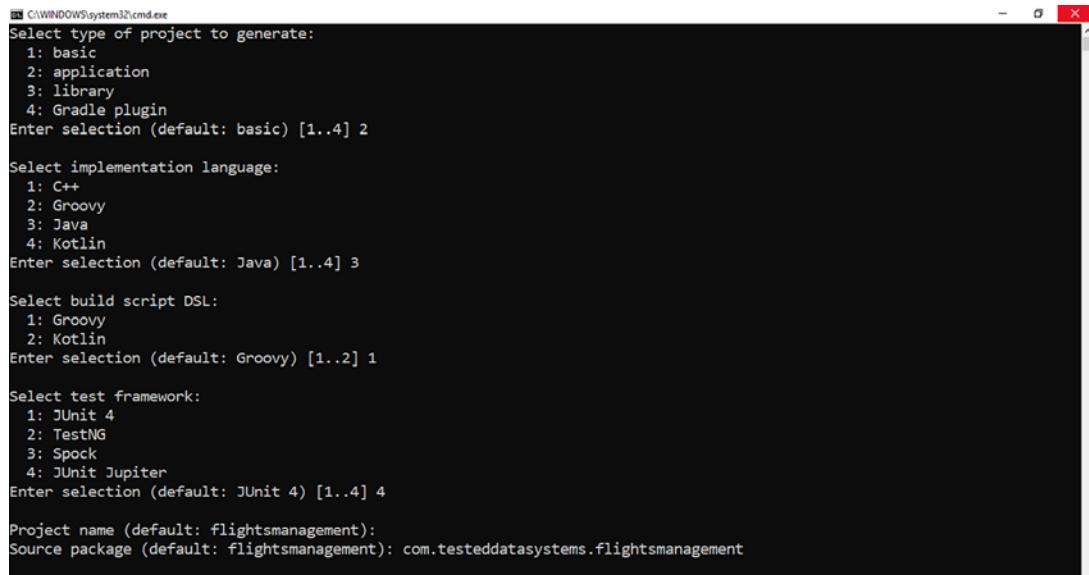
If we compare the code in listings 11.3–11.10, we get Oliver’s Gradle configuration file from listing 11.2.

11.4 ***Creating a Gradle project from scratch and testing it with JUnit 5***

We’ll now demonstrate all the steps necessary to create a JUnit 5 project that is managed by Gradle. As we have discussed previously, one of the projects under development at Tested Data Systems is a flight-management application. In order to start such an application, Oliver, the project developer, creates the `C:\Work\flightsmanagement` folder that will become the working directory and where he will set up the Gradle project. He types the following on the command line:

```
gradle init
```

The result of running this command is shown in figure 11.3.



The screenshot shows a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window displays the following interaction with the 'gradle init' command:

```
Select type of project to generate:
 1: basic
 2: application
 3: library
 4: Gradle plugin
Enter selection (default: basic) [1..4] 2

Select implementation language:
 1: C++
 2: Groovy
 3: Java
 4: Kotlin
Enter selection (default: Java) [1..4] 3

Select build script DSL:
 1: Groovy
 2: Kotlin
Enter selection (default: Groovy) [1..2] 1

Select test framework:
 1: JUnit 4
 2: TestNG
 3: Spock
 4: JUnit Jupiter
Enter selection (default: JUnit 4) [1..4] 4

Project name (default: flightsmanagement):
Source package (default: flightsmanagement): com.testeddatasystems.flightsmanagement
```

Figure 11.3 The result of executing the `gradle init` command to create the flight-management application

The structure of the project is shown in figure 11.4. `src/main/java/` is the Gradle folder where the Java code for the project resides, and `src/test/java` contains the unit tests. The `.idea` folder is created by the IDE.

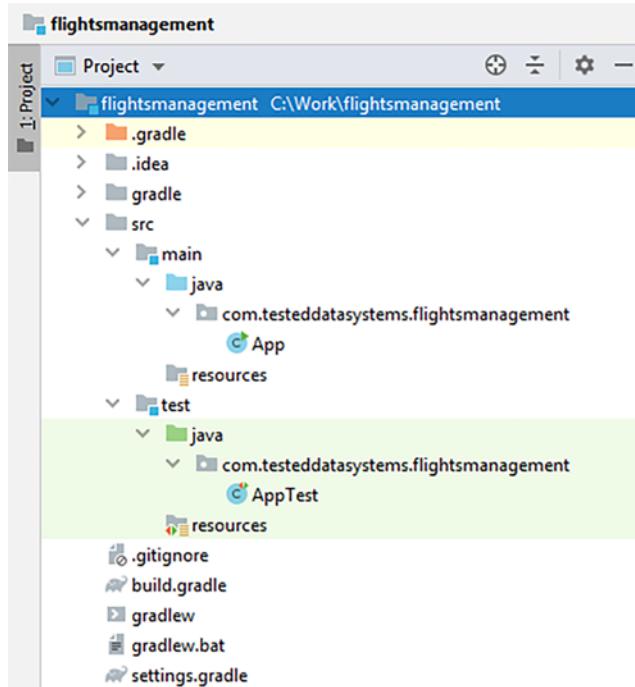


Figure 11.4 The newly created flight-management project

Executing the `gradle init` command also generates a `build.gradle` file.

Listing 11.11 build.gradle file for the flight-management application

```
plugins {
    // Apply the java plugin to add support for Java
    id 'java'

    // Apply the application plugin to add support for building a CLI application
    id 'application'
}

repositories {
    // Use jcenter for resolving dependencies.
    // You can declare any Maven/Ivy/file repository here.
    jcenter()
}
```

```

dependencies {
    // Use JUnit Jupiter API for testing.
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'

    // Use JUnit Jupiter Engine for testing.
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.6.0'
}

application {
    // Define the main class for the application
    mainClassName = 'com.testeddatasystems.flightsmanagement.App'
}

test {
    // Use junit platform for unit tests
    useJUnitPlatform()
}

```

Oliver removes the existing autogenerated App and AppTest classes. He adds the two classes that will start his application: Passenger (listing 11.12) and PassengerTest (listing 11.13).

Listing 11.12 Passenger class

```

package com.testeddatasystems.flights;

public class Passenger {
    private String identifier;           | 1
    private String name;               |

    public Passenger(String identifier, String name) { | 2
        this.identifier = identifier;
        this.name = name;
    }

    public String getIdentifier() { | 3
        return identifier;
    }

    public String getName() { | 4
        return name;
    }

    @Override
    public String toString() { | 5
        return "Passenger " + getName() +
            " with identifier: " + getIdentifier();
    }

    public static void main (String args[]) {
        Passenger passenger = new Passenger("123-456-789", "John Smith");
        System.out.println(passenger);
    }
}

```

The Passenger class contains the following:

- Two fields, identifier and name ①
- A constructor receiving as arguments identifier and name ②
- Getters for the identifier and name fields ③
- The overridden `toString` method ④
- The main method will create and display a passenger ⑤.

Listing 11.13 The PassengerTest class

```
package com.testeddatasystems.flights;
[...]

public class PassengerTest {

    @Test
    void testPassenger() {
        Passenger passenger = new Passenger("123-456-789",
                                             "John Smith");
        assertEquals("Passenger John Smith with identifier:
                     123-456-789",
                     passenger.toString());
    }
}
```

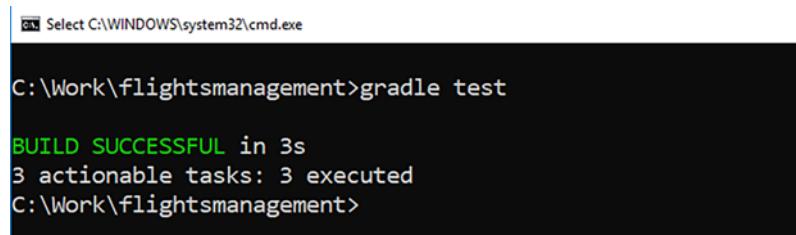
1

The PassengerTest class contains one test: `testPassenger`, which verifies the output of the overridden `toString` method ①.

In the `build.gradle` file, Oliver changes the main class to

```
application {
    // Define the main class for the application
    mainClassName = 'com.testeddatasystems.flightsmanagement.Passenger'
}
```

He then executes the `gradle test` command at the level of the project folder. This command simply runs the JUnit 5 test introduced in the application (figure 11.5).

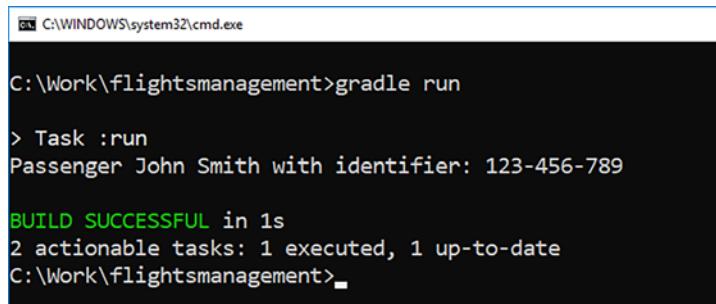


```
C:\Work\flightsmanagement>gradle test

BUILD SUCCESSFUL in 3s
3 actionable tasks: 3 executed
C:\Work\flightsmanagement>
```

Figure 11.5 Executing the `gradle test` command for the Gradle project under development

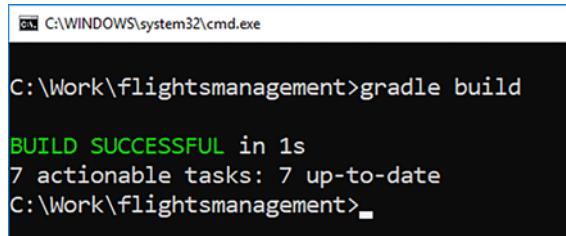
Next, Oliver executes the `gradle run` command at the level of the project folder. This command runs the main class of the application, as described in the `build.gradle` file, and displays the information about passenger John Smith (figure 11.6). Running the `gradle build` command at the level of the project folder (figure 11.7) also creates the jar artifact in the `build/libs` folder (figure 11.8).



```
C:\Work\flightsmanagement>gradle run
> Task :run
Passenger John Smith with identifier: 123-456-789

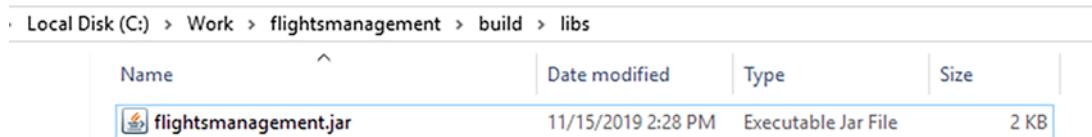
BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
C:\Work\flightsmanagement>
```

Figure 11.6 Executing the `gradle run` command



```
C:\Work\flightsmanagement>gradle build
BUILD SUCCESSFUL in 1s
7 actionable tasks: 7 up-to-date
C:\Work\flightsmanagement>
```

Figure 11.7 Executing the `gradle build` command



Local Disk (C:) > Work > flightsmanagement > build > libs			
Name	Date modified	Type	Size
flightsmanagement.jar	11/15/2019 2:28 PM	Executable Jar File	2 KB

Figure 11.8 The jar file in the Gradle build/libs folder

At this time, Oliver has created a fully functional Gradle project to develop the flight-management application and test it using JUnit 5. From now on, he can continue to add new classes and tests and execute the Gradle commands to package the application and run the tests.

11.5 Comparing Gradle and Maven

We have now used and compared two build tools that will help us manage our JUnit 5 projects. The engineers at Tested Data Systems have some freedom of choice: some projects may use Maven, as it is an older and reliable tool that most of them know well. For other projects, they may decide to adopt Gradle: if they need to write custom tasks (which is harder to do with Maven), if they consider XML tedious and prefer to use a DSL that is close to Java, or if they are challenged to adopt something new. The choice may be a matter of personal or project preference—and it may be the same for you. In general, once you master one of these two build tools, joining a project that uses the other is not difficult.

We have mentioned that, at least of the time of writing, there are some known issues when using JUnit 5 with Gradle: classes and tests are still displayed with their names instead of `@DisplayName`. This is expected to be solved in future releases.

We should mention that projects at the global level still predominantly use Maven—about three-quarters of them, according to our research. In addition, the numbers are quite stable: Gradle hasn't advanced much in the market over the years. This is why, in this book, we mostly use Maven.

Summary

This chapter has covered the following:

- Creating a Java project using JUnit 5 with the help of Gradle
- Introduction to Gradle plugins
- Creating a Gradle project from scratch, opening it in IntelliJ IDEA, and testing it using JUnit 5
- Comparing Gradle and Maven as build tool alternatives

12

JUnit 5 IDE support

This chapter covers

- Using JUnit 5 in IntelliJ IDEA
- Using JUnit 5 in Eclipse
- Using JUnit 5 in NetBeans

Enjoy the little things, for one day you may look back and realize they were the big things.

—Robert Brault

In this chapter, we will analyze and compare the leading IDEs that can be used to develop Java applications tested with the help of JUnit 5. An integrated development environment (IDE) is an application that provides software development facilities for computer programmers. Java IDEs are tools that let us write and debug Java programs more easily.

Many IDEs have been created to help Java developers. Of these, we have selected three of the most popular to analyze and compare: IntelliJ IDEA, Eclipse, and NetBeans. At the time of writing this chapter, IntelliJ IDEA and Eclipse each had about 40% of the market share. NetBeans had about 10%; we are including it

in our analysis because it is the third option overall and particularly popular in some regions of the world.

Which of the possible IDEs you choose may be a matter of personal preference or project tradition. All of them may be used in the same company or even the same project. In fact, this is the case for our example company, Tested Data Systems Inc. We'll point out the reasons that determined which options the developers chose.

Our analysis and demonstration will focus on the capabilities of these IDEs using JUnit 5—the just-in-time information that you need now. For more comprehensive guidance, there are many other available sources, starting with the official documentation of each of the IDEs. We'll demonstrate using JUnit 5 with IntelliJ IDEA, then Eclipse, and finally NetBeans, as this is the order in which they were integrated with JUnit 5.

12.1 **Using JUnit 5 with IntelliJ IDEA**

IntelliJ IDEA is an IDE developed by JetBrains. It is available as an Apache 2 licensed community edition and in a proprietary commercial edition. For brief IntelliJ IDEA installation instructions, see appendix C.

We will demonstrate JUnit 5 in different IDEs using a selection of tests from chapter 2. Those tests covered the capabilities of JUnit 5 comprehensively, and we have chosen the ones that are significant in the context of an IDE. You can review the new JUnit 5 features and their use cases in chapter 2. Here, we are most interested in JUnit 5 within each IDE, but we'll briefly remind you in each case why you would use that particular feature.

Launch IntelliJ IDEA, and then open a project by choosing File > Open. When you open the project from this chapter's source code, the IDE will look as shown in figure 12.1. The project contains the tests that demonstrate JUnit 5's new features: capabilities such as displaying names and nested, parameterized, repeated, and dynamic tests using tags.

From here, you can execute all tests by right-clicking the highlighted test/java folder and then choosing Run > All Tests. You will get a result like that shown in figure 12.2. You will probably want to run all of the tests periodically while you are working on the implementation of a particular feature, to make sure the full project is working correctly and that your implementation does not affect the previously existing functionality.

Let's have a look at how to run particular test types with the help of IntelliJ IDEA. In figure 12.2, disabled tests are shown in gray, and they are marked as “ignored” after executing the whole suite. You can, however, force the execution of a particular test, even if it is disabled, by right-clicking it and running it directly, as shown in figure 12.3. You may want to do that occasionally to check whether the conditions that prevented the correct execution of that test have ceased to exist. For example, your test may wait for the implementation of a feature by a different team or for the availability of a resource, and you can verify fulfillment.

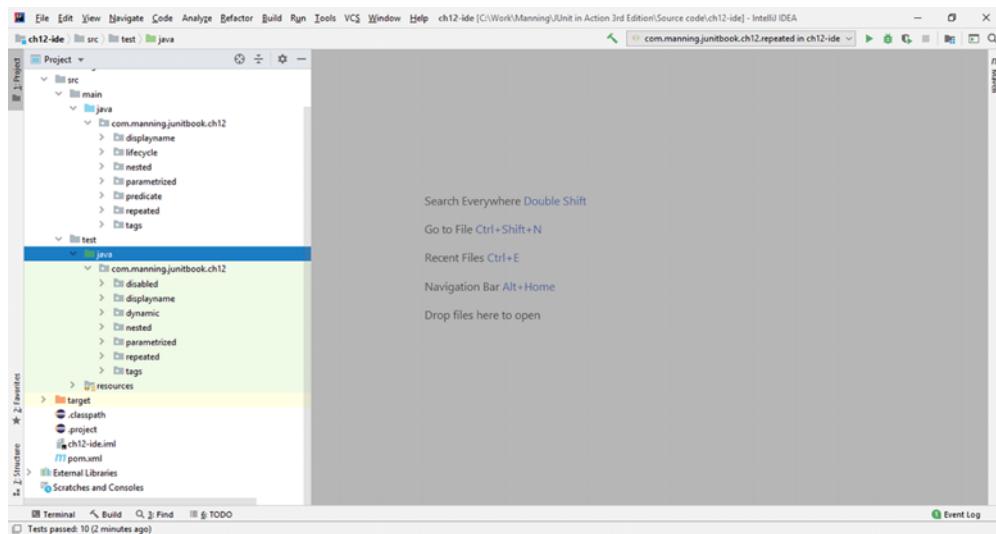


Figure 12.1 IntelliJ IDEA with an open JUnit 5 project

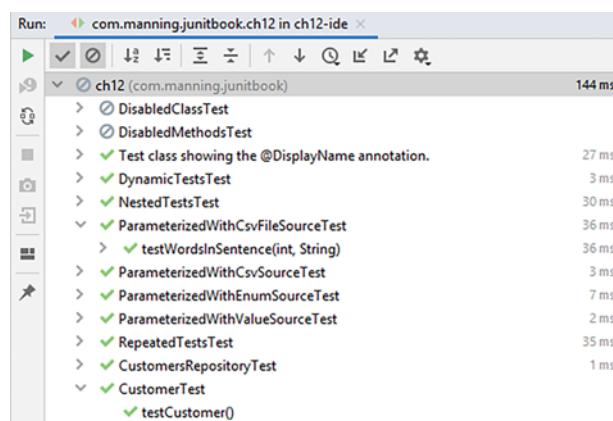


Figure 12.2 Executing all of the tests from inside IntelliJ IDEA

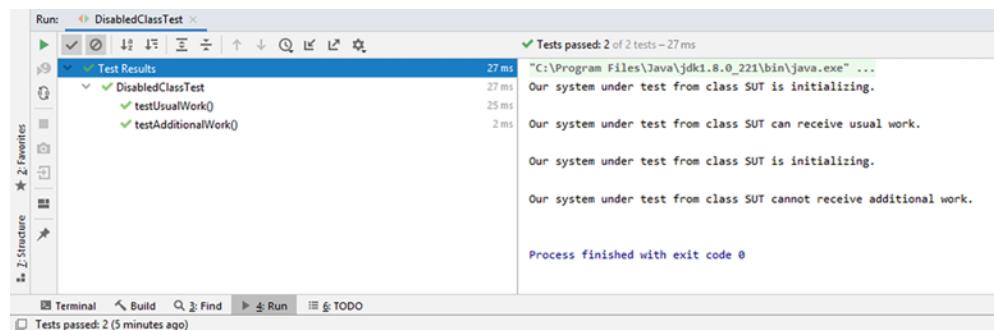


Figure 12.3 Forcing the execution of a disabled test from IntelliJ IDEA

When you run tests that are annotated with `@DisplayName`, they are shown as in figure 12.4. You will use this annotation and work with this kind of test from IntelliJ IDEA when you want to watch some significant information while you are developing from the IDE. We discussed this annotation in chapter 2.

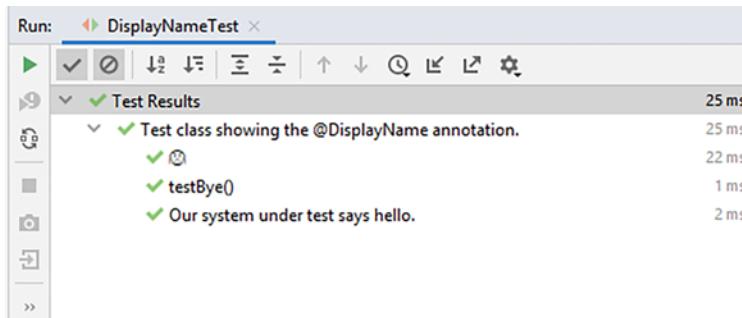


Figure 12.4 Running a test annotated with `@DisplayName` from IntelliJ IDEA

When you run dynamic tests that are annotated with `@TestFactory`, they are shown as in figure 12.5. You will use this annotation and will work with this kind of test from IntelliJ IDEA when you want to write a reasonable amount of code to generate tests at runtime. We discussed this annotation in chapter 2.

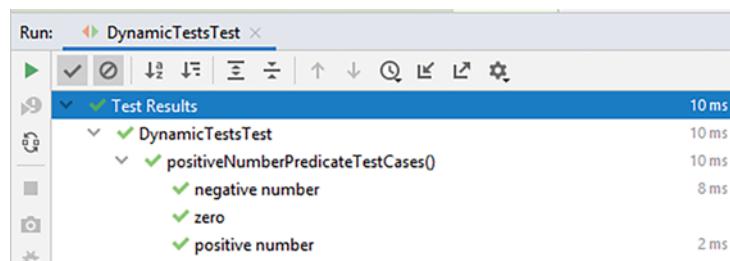


Figure 12.5 Running a dynamic test from IntelliJ IDEA

When you run nested tests, they are shown hierarchically, as in figure 12.6, where we have also taken advantage of the `@DisplayName` annotation's capabilities. You can use nested tests to express the relationships among several groups of tests that are tightly coupled; IntelliJ IDEA will display the hierarchy. We discussed the `@Nested` annotation in chapter 2.

When you run parameterized tests, they are displayed in detail, showing all of their parameters, as in figure 12.7. You can use parameterized tests to write a single test and then perform the testing using a series of different arguments. We discussed the annotation related to parameterized tests in chapter 2.

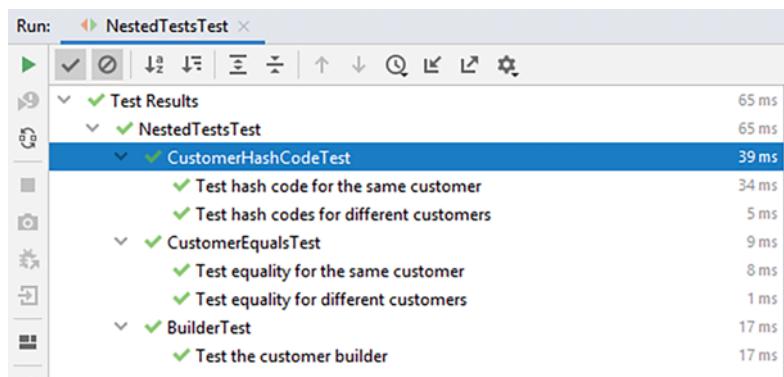


Figure 12.6 Running nested tests from IntelliJ IDEA

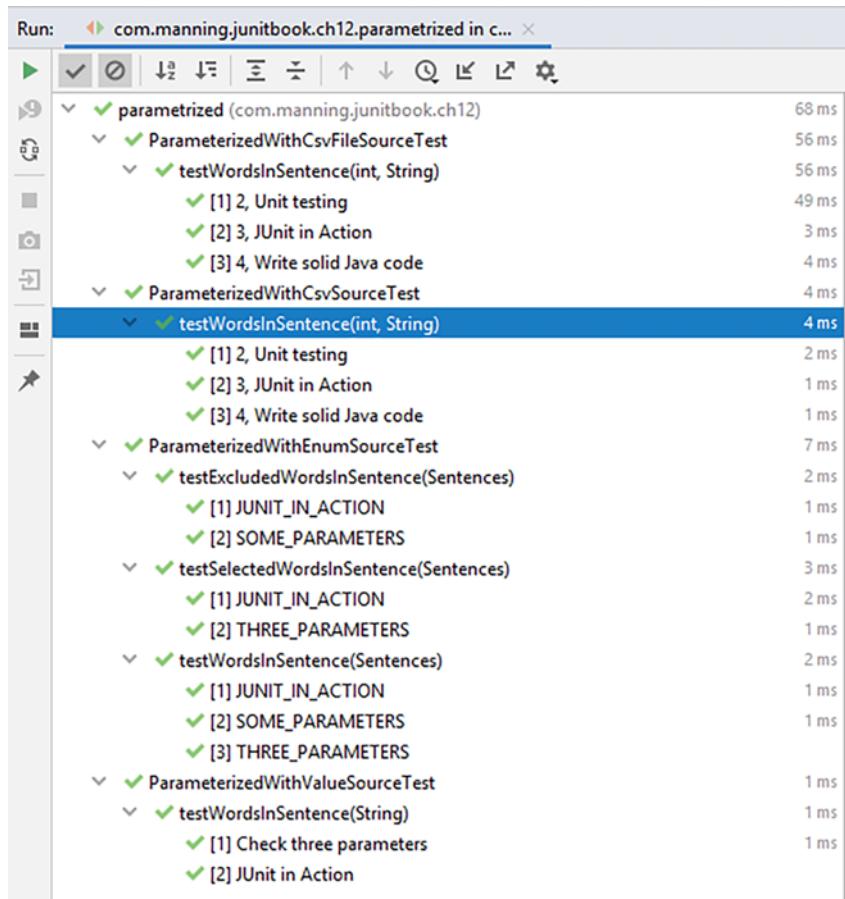


Figure 12.7 Running parameterized tests from IntelliJ IDEA

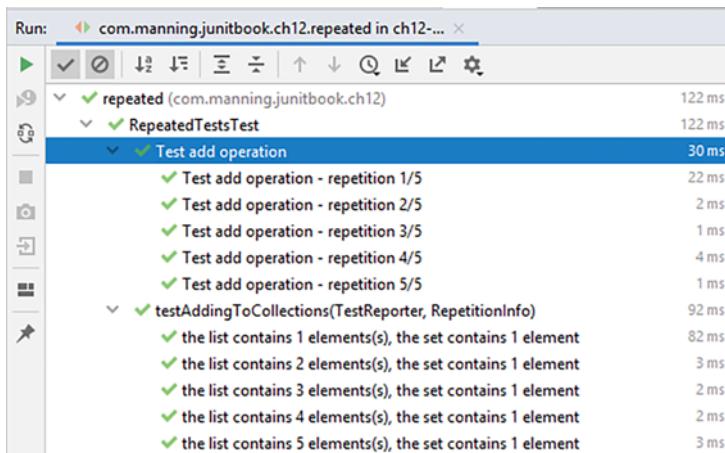


Figure 12.8 Running repeated tests from IntelliJ IDEA

When you run repeated tests, they are displayed in detail, showing all the required information about the currently repeated test, as in figure 12.8. You can use repeated tests if the running conditions may change from one test execution to the next. We discussed the `@RepeatedTest` annotation in chapter 2.

When you run the entire test suite from IntelliJ IDEA, the tagged tests are also run. If you want to run only particular tagged tests, choose `Run > Edit Configurations` and select from the `Test Kind > Tags` list, as shown in figure 12.9. You need to create the

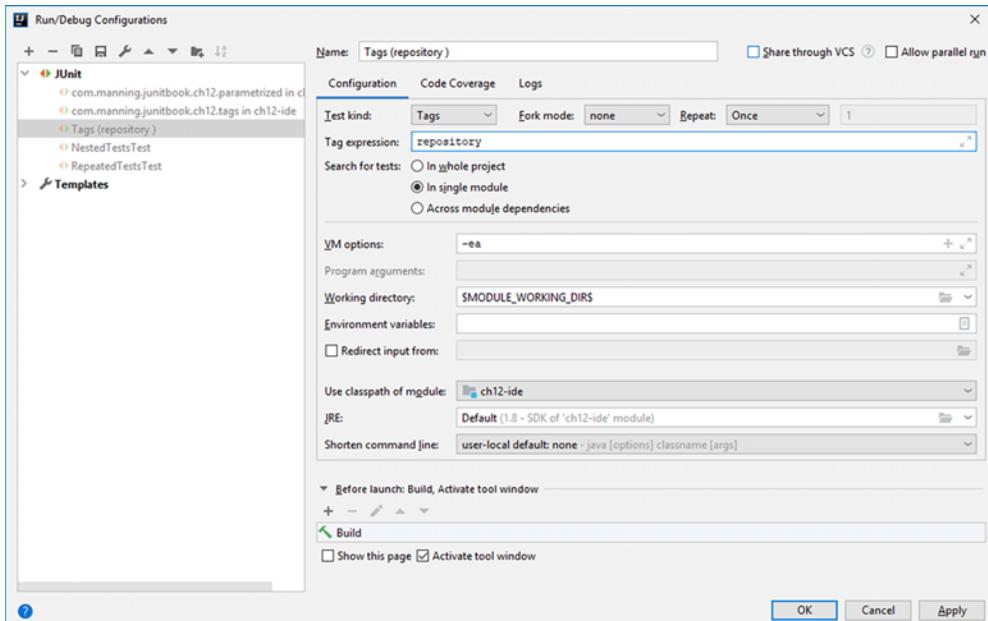


Figure 12.9 Configuring the option of running only tagged tests from IntelliJ IDEA

configuration by hand and execute it. Then you will be able to run only those particular tagged tests' configurations. You'll want to do this when you have a larger suite of tests and need to focus on particular tests rather than spend time with the entire suite. We discussed the `@Tag` annotation in chapter 2.

IntelliJ IDEA also makes it easy to run tests with code coverage. If you right-click the test or suite you want to run and choose Run with Coverage, you will see a table like that in figure 12.10. You can investigate at the class level and get information, as shown in figure 12.11. Or you can click the Generate Coverage Report button and obtain an HTML report, as in figure 12.12.

Element	Class, %	Method, %	Line, %
<code>com.manning.junitbook.ch12.li...</code>	100% (1/1)	100% (3/3)	100% (4/4)
<code>com.manning.junitbook.ch12.n...</code>	100% (3/3)	100% (16/16)	100% (35/35)
<code>com.manning.junitbook.ch12.p...</code>	100% (1/1)	100% (1/1)	100% (2/2)
<code>com.manning.junitbook.ch12.p...</code>	100% (1/1)	100% (1/1)	100% (2/2)
<code>com.manning.junitbook.ch12.r...</code>	100% (1/1)	100% (1/1)	100% (2/2)
<code>com.manning.junitbook.ch12.t...</code>	100% (2/2)	100% (4/4)	100% (9/9)

Figure 12.10 The table produced by IntelliJ IDEA after choosing Run with Coverage

```

public class SUT {
    private String systemName;

    public SUT(String systemName) {
        this.systemName = systemName;
        System.out.println(systemName + " from class " + getClass().getSimpleName() + " is initialized");
    }

    public boolean canReceiveUsualWork() {
        System.out.println(systemName + " from class " + getClass().getSimpleName() + " can receive usual work");
        return true;
    }

    public boolean canReceiveAdditionalWork() {
    }
}

```

Figure 12.11 Code coverage at the individual class level

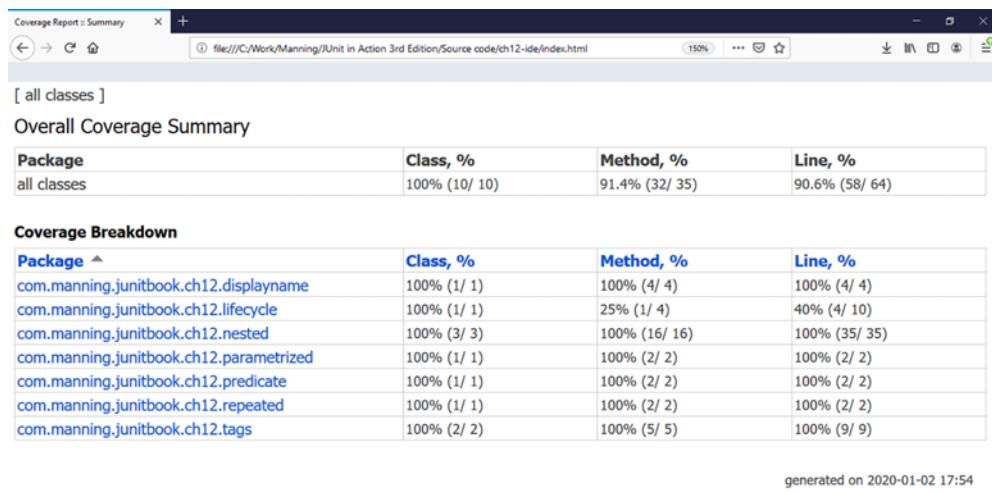


Figure 12.12 Code coverage HTML report obtained with the help of IntelliJ IDEA

We discussed code coverage in more detail in chapter 6. Remember that it is a metric of test quality, showing how much of the production code is covered by the tests we are executing. While developing code and tests, we want to get quick feedback not only about whether the tests run successfully but also about their coverage. IntelliJ IDEA strongly supports both goals.

At Tested Data Systems, the developers who have decided to use IntelliJ for their projects are not only those who were using it before moving to JUnit 5, but also those who introduced this version of the testing framework during its early days. At that time, IntelliJ IDEA was the only IDE providing support for JUnit 5—it was the only option.

12.2 Using JUnit 5 with Eclipse

Eclipse is an IDE written mostly in Java. It can be used to develop applications in many programming languages, but its primary use is developing Java applications. For brief Eclipse installation instructions, see appendix C.

We'll walk through a demonstration similar to the one for IntelliJ IDEA: we selected tests from chapter 2 that are significant in the context of usage from inside an IDE. Launch Eclipse, import a project by selecting File > Import > General > Existing Projects into Workspace, and then choose the folder where the project is located. The IDE will look as shown in figure 12.13.

From here, you can execute all of the tests by right-clicking the project and choosing Run As > JUnit Test. You will see a result like that in figure 12.14. You will probably want to run all of the tests periodically while you are working on the implementation of a particular feature, to make sure the full project is working correctly and that your implementation does not affect the previously existing functionality.

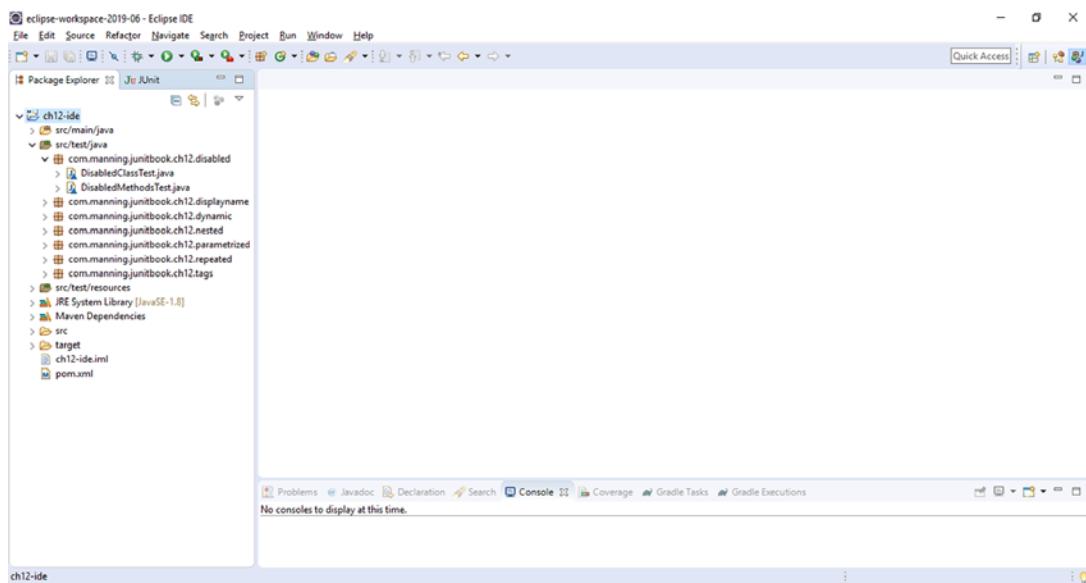


Figure 12.13 Eclipse with an open JUnit 5 project

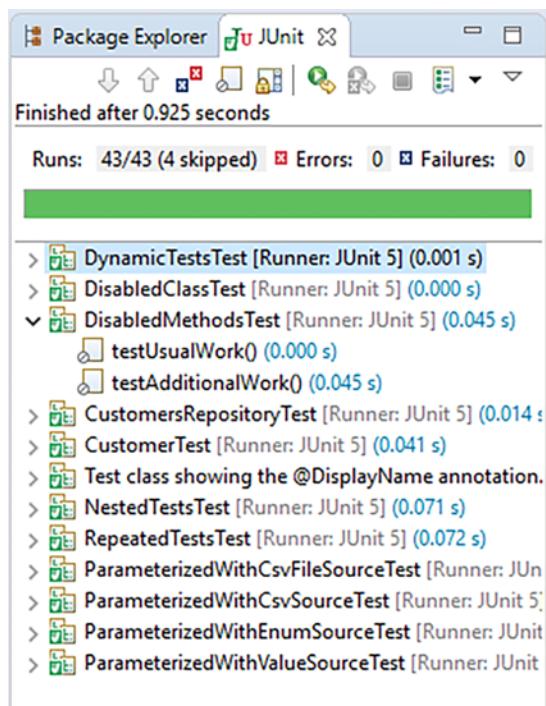


Figure 12.14 Executing all tests from inside Eclipse

Let's have a look at how to run particular test types with the help of Eclipse. In figure 12.14, disabled tests are shown in gray, and they are marked as "skipped" after executing the whole suite. In contrast to the capabilities of IntelliJ IDEA, you cannot force the execution of a particular disabled test.

When you run tests that are annotated with `@DisplayName`, they are shown as in figure 12.15. You will use this annotation and work with this kind of test from Eclipse when you want to watch some significant information while developing from the IDE. We discussed this annotation in chapter 2.

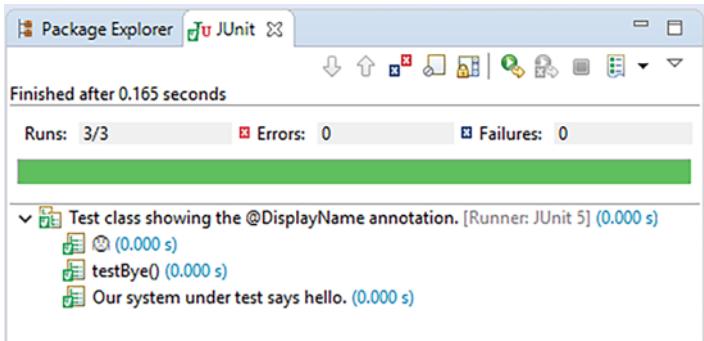


Figure 12.15 Running a test annotated with `@DisplayName` from Eclipse

When you run dynamic tests that are annotated with `@TestFactory`, they are shown as in figure 12.16. You can use this annotation and work with this kind of test from Eclipse when you want to write a reasonable amount of code to generate tests at runtime. We discussed this annotation in chapter 2.

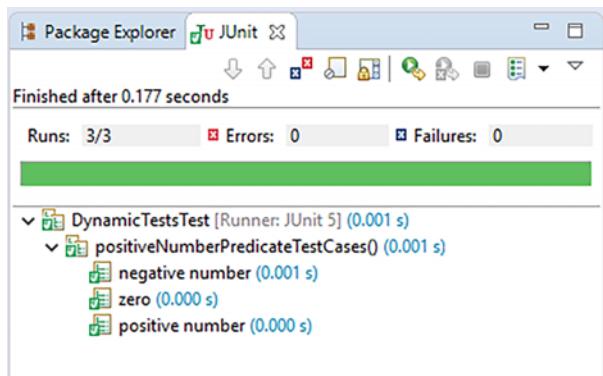


Figure 12.16 Running a dynamic test from Eclipse

When you run nested tests, they are shown hierarchically, as in figure 12.17, where we have also taken advantage of the `@DisplayName` annotation's capabilities. You can use nested tests to express the relationships among several groups of tests that are tightly coupled; Eclipse will display the hierarchy. We discussed `@Nested` in chapter 2.

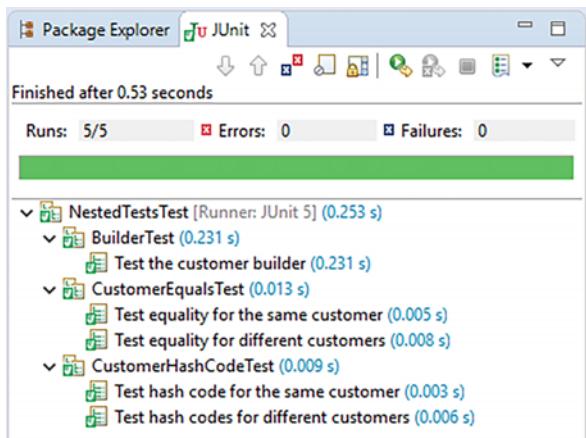


Figure 12.17 Running nested tests from Eclipse

When you run parameterized tests, they are displayed in detail, showing their parameters, as in figure 12.18. You can use parameterized tests to write a single test and then perform the testing using a series of different arguments. We discussed the annotation for parameterized tests in chapter 2.



Figure 12.18 Running parameterized tests from Eclipse

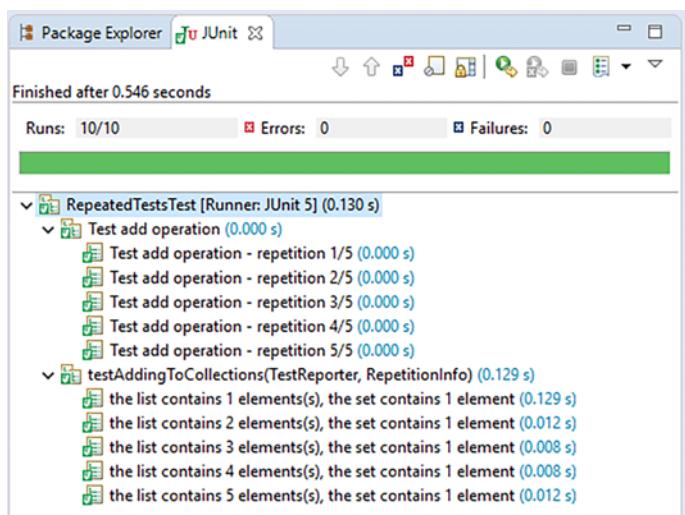


Figure 12.19 Running repeated tests from Eclipse

When you run repeated tests, they are displayed in detail, showing all the required information about the currently repeated test, as in figure 12.19. You can use repeated tests if running conditions may change from one test execution to the next. We discussed the `@RepeatedTest` annotation in chapter 2.

When you run the entire test suite from Eclipse, the tagged tests are also run. If you want to run only particular tagged tests, choose *Run* > *Run Configurations* > *JUnit* and select from *Include Tags* and *Exclude Tags*, as shown in figure 12.20.

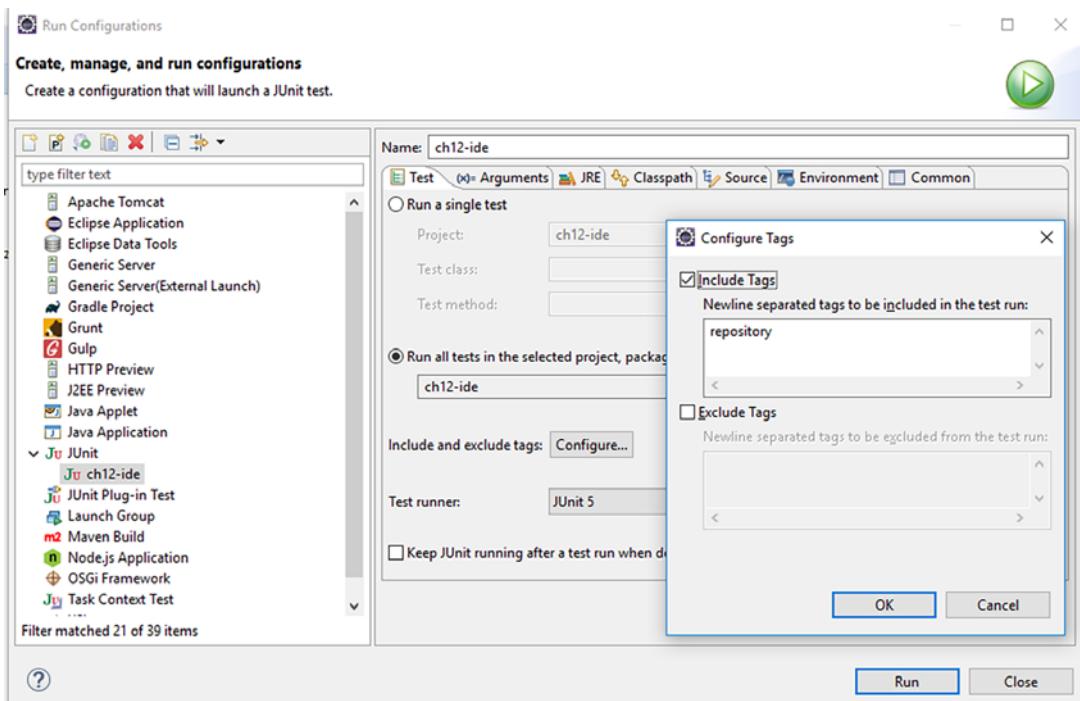


Figure 12.20 Configuring the option to run only tagged tests from Eclipse

Then you can run the configuration that only addresses tagged tests. You will do this when you have a larger suite of tests and need to focus on particular tests rather than spend time with the entire suite. We discussed the `@Tag` annotation in chapter 2.

Eclipse also makes it easy to run tests with code coverage. If you right-click the project, test, or suite to be run, and then choose `Coverage As > JUnit Test`, you will see a table like the one shown in figure 12.21. You can investigate at the individual class level and get information like that in figure 12.22. Or you can right-click the Coverage table, choose `Export Session`, and obtain an HTML report, as in figure 12.23.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
ch12-ide	88.9 %	776	97	873
src/main/java	81.0 %	255	60	315
com.manning.junitbook.ch12.lifecycle	28.2 %	22	56	78
SUT.java	28.2 %	22	56	78
com.manning.junitbook.ch12.nested	97.6 %	165	4	169
Customer.java	97.2 %	141	4	145
Gender.java	100.0 %	24	0	24
com.manning.junitbook.ch12.displayname	100.0 %	9	0	9
com.manning.junitbook.ch12.parameterized	100.0 %	8	0	8
com.manning.junitbook.ch12.predicate	100.0 %	9	0	9
com.manning.junitbook.ch12.repeated	100.0 %	7	0	7
com.manning.junitbook.ch12.tags	100.0 %	35	0	35
src/test/java	93.4 %	521	37	558

Figure 12.21 The table produced by Eclipse after running `Coverage As > JUnit Test`

```

1 SUT.java
2 * Licensed to the Apache Software Foundation (ASF) under one or more
3
4 package com.manning.junitbook.ch12.lifecycle;
5
6 public class SUT {
7     private String systemName;
8
9     public SUT(String systemName) {
10         this.systemName = systemName;
11         System.out.println(systemName + " from class " + getClass().getSimpleName() + " is initializing.");
12     }
13
14     public boolean canReceiveUsualWork() {
15         System.out.println(systemName + " from class " + getClass().getSimpleName() + " can receive usual work.");
16         return true;
17     }
18
19     public boolean canReceiveAdditionalWork() {
20         System.out.println(systemName + " from class " + getClass().getSimpleName() + " cannot receive additional work.");
21         return false;
22     }
23
24     public void close() {
25         System.out.println(systemName + " from class " + getClass().getSimpleName() + " is closing.");
26     }
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47 }

```

Figure 12.22 Code coverage at the individual class level

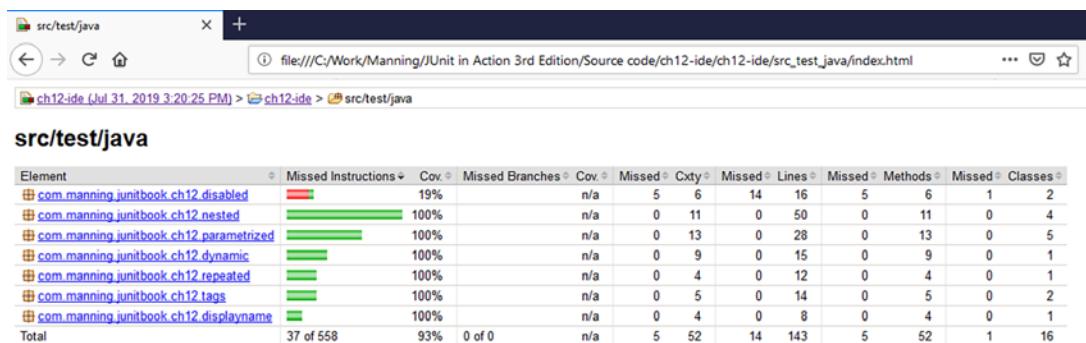


Figure 12.23 Code coverage HTML report obtained with the help of Eclipse

While developing code and tests, we want to get quick feedback not only about whether the tests run successfully but also about their coverage. Eclipse strongly supports both goals. Code coverage is a metric of test quality, showing how much of the production code is covered by the tests we are executing. At Tested Data Systems, the developers who have decided to use Eclipse for their projects are generally those who were already using it before moving to JUnit 5 and were not under pressure to adopt it. This is simply because Eclipse came with JUnit 5 support later than IntelliJ IDEA.

12.3 Using JUnit 5 with NetBeans

NetBeans is an IDE that we can use to write code in several programming languages, including Java. For brief NetBeans installation instructions, see appendix C.

We'll walk through a demonstration similar to that for IntelliJ IDEA and Eclipse. We have selected JUnit 5 tests from chapter 2 that are significant in the context of usage from inside an IDE: tests demonstrating capabilities such as displaying names and nested, parameterized, repeated, and dynamic tests, using tags.

Launch NetBeans using one of the executables from the netbeans/bin folder: netbeans or netbeans64, depending on the operating system. To open a project, select File > Open Project, and then choose the folder where the project is located. The IDE will look as shown in figure 12.24.

From here, you can execute all of the tests by right-clicking the project and choosing Run/Test project. You will get a result like that shown in figure 12.25. You will probably want to run all of the tests periodically while you are working on the implementation of a particular feature, to make sure the full project is working correctly and that your implementation does not affect the previously existing functionality.

In figure 12.25, disabled tests are shown in gray, and they are marked as “skipped” after executing the whole suite. In contrast to the capabilities of IntelliJ IDEA, you cannot force the execution of a particular disabled test.

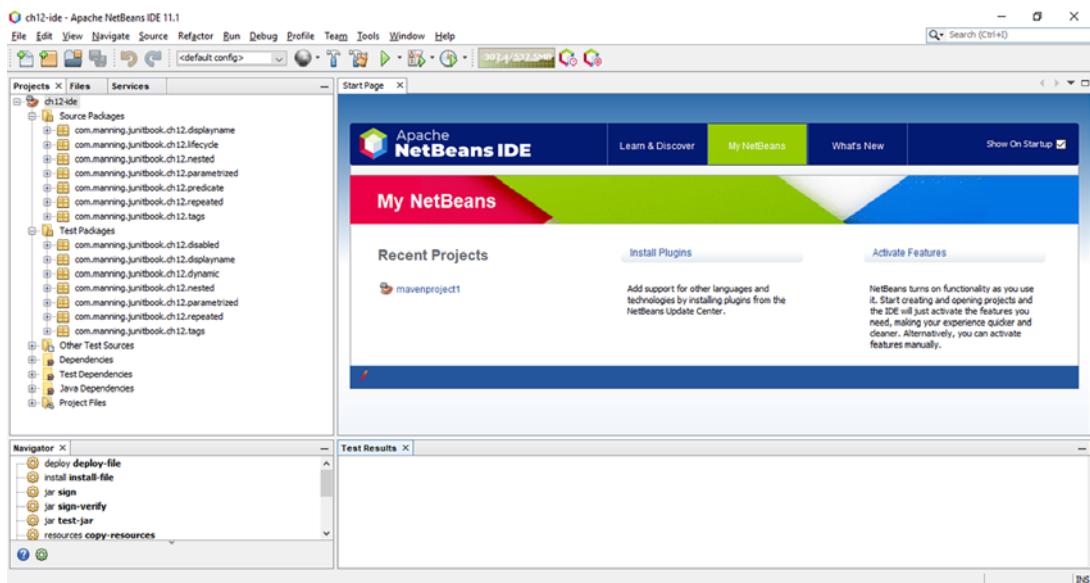


Figure 12.24 NetBeans with an open JUnit 5 project

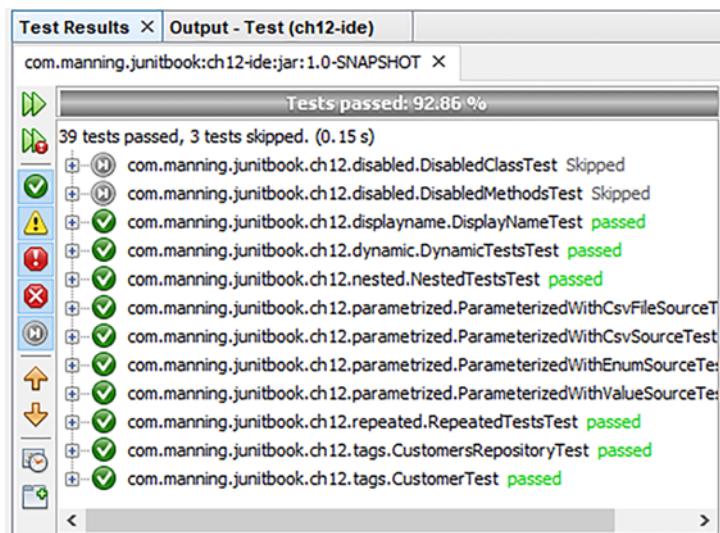


Figure 12.25 Executing all tests from inside NetBeans

When you run tests that are annotated with `@DisplayName`, they are shown as in figure 12.26. Unlike IntelliJ IDEA and Eclipse, NetBeans cannot use the information provided by this annotation and simply shows the names of tests that have been run. We discussed this annotation in chapter 2.

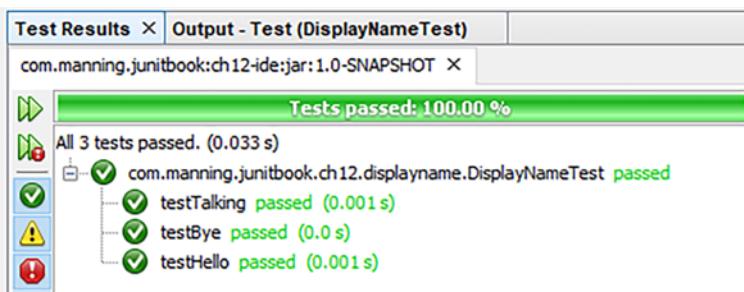


Figure 12.26 Running a test annotated with `@DisplayName` from IntelliJ

When you run dynamic tests that are annotated with `@TestFactory`, they are shown as in figure 12.27. Unlike IntelliJ IDEA and Eclipse, NetBeans cannot use the names of the dynamically generated tests and simply numbers them. We discussed this annotation in chapter 2.

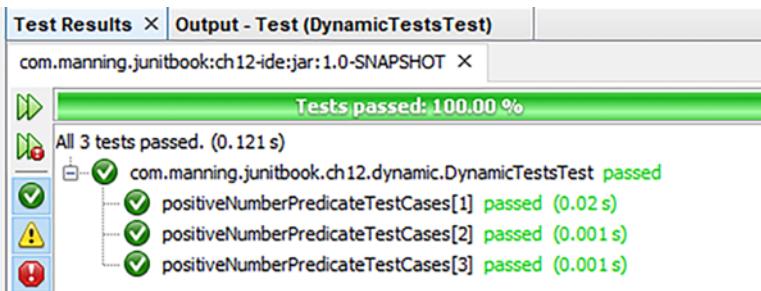


Figure 12.27 Running a dynamic test from NetBeans

When you run nested tests, they are not even recognized by NetBeans, as shown in figure 12.28. We discussed the `@Nested` annotation in chapter 2.

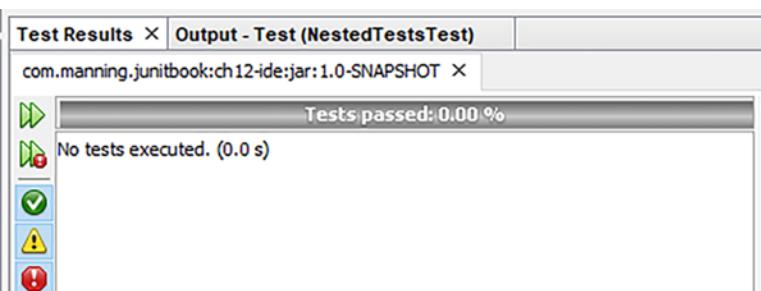


Figure 12.28 Running nested tests from NetBeans

When you run parameterized tests, they are displayed with numbers but without details of their parameters, unlike in IntelliJ IDEA and Eclipse. This is shown in figure 12.29. We discussed parameterized test annotations in chapter 2.

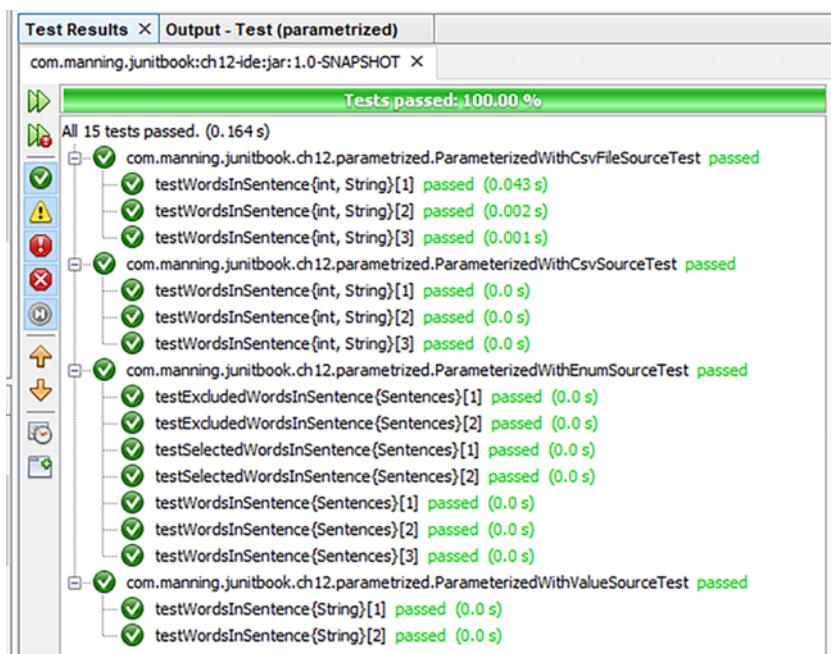


Figure 12.29 Running parameterized tests from NetBeans

When you run repeated tests, they are shown without any of the required information about the currently repeated test, as in IntelliJ IDEA and Eclipse. You can see the results in figure 12.30. We discussed the `@RepeatedTest` annotation in chapter 2.

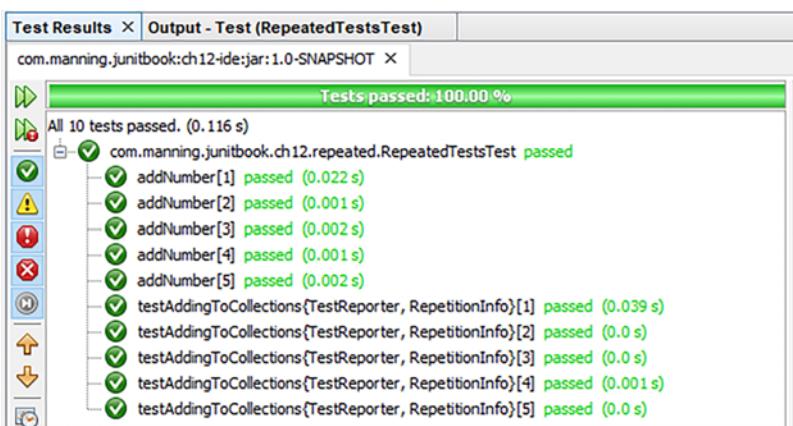


Figure 12.30
Running
repeated tests
from NetBeans

When running the whole test suite from NetBeans, tagged tests are also run. The IDE does not provide the option to run only particular tagged tests. If you are running a Maven project, you have to make changes at the level of the pom.xml file with the help of the Surefire plugin. This plugin is used during the test phase of a project to execute an application's unit tests; it may filter some of the executed tests or generate reports. A possible configuration is shown in the following listing, including tests tagged with `individual` and excluding tests tagged with `repository`. We discussed the `@Tag` annotation in chapter 2.

Listing 12.1 Possible filtering configuration for the Maven Surefire plugin

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
      <configuration>
        <groups>individual</groups>
        <excludedGroups>repository</excludedGroups>
      </configuration>
    </plugin>
  </plugins>
</build>
```

It is not possible to run tests directly with code coverage from NetBeans. We can do this with the help of JaCoCo, an open source toolkit for measuring and reporting Java code coverage. If you are running a Maven project, you have to make changes at the level of the pom.xml file, using JaCoCo. The configuration of the JaCoCo plugin is shown next.

Listing 12.2 Configuration of the JaCoCo plugin

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.7.7.201606060606</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

After you build the project (right-click and choose Build), the Code Coverage menu item appears when you right-click again on the project (figure 12.31). You can choose Show Report (figure 12.32) or investigate at the individual class level (figure 12.33).

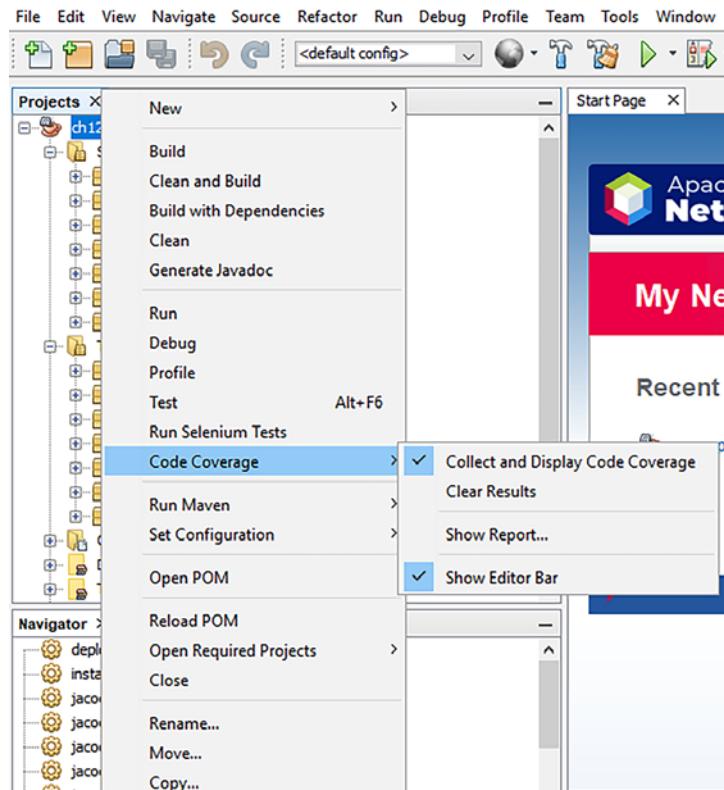


Figure 12.31 The newly added Code Coverage menu

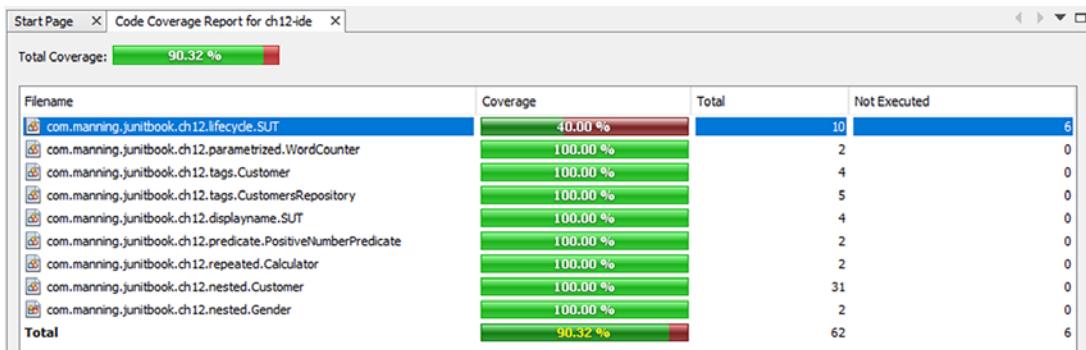


Figure 12.32 Code coverage as shown from NetBeans, with the help of the JaCoCo plugin

```

21
22 package com.manning.junitbook.ch12.lifecycle;
23
24 public class SUT {
25     private String systemName;
26
27     public SUT(String systemName) {
28         this.systemName = systemName;
29         System.out.println(systemName + " from class " + getClass().getSimpleName() + " is initializing.");
30     }
31
32     public boolean canReceiveUsualWork() {
33         System.out.println(systemName + " from class " + getClass().getSimpleName() + " can receive usual work.");
34         return true;
35     }
36
37     public boolean canReceiveAdditionalWork() {
38         System.out.println(systemName + " from class " + getClass().getSimpleName() + " cannot receive additional work.");
39         return false;
40     }
41
42     public void close() {
43         System.out.println(systemName + " from class " + getClass().getSimpleName() + " is closing.");
44     }

```

Code Coverage: 40.00 %

Test All Tests Clear Report... Disable

Figure 12.33 Code coverage at the individual class level

Using NetBeans, you can get feedback about code coverage, but you need to use separate plugins, as we have demonstrated with JaCoCo. At Tested Data Systems, the developers who have decided to use NetBeans for their projects are generally those who have been using it a long time and moved very late to JUnit 5 (this IDE was the last one to introduce JUnit 5 support). Overall, they are not very interested in the user-friendly behavior of the testing framework's new features.

12.4 Comparing JUnit 5 usage in IntelliJ, Eclipse, and NetBeans

Table 12.1 summarizes our comparison of the three IDEs that can help you develop JUnit 5 projects. Remember that we used the latest version of each IDE at the time of writing; the IDE developers may focus on more integration in the future. IntelliJ IDEA now has the best integration, followed closely by Eclipse. NetBeans neglects most of the capabilities and user-friendly behavior provided by JUnit 5.

Which IDE you decide to use will depend on personal preference or project requirements. To make a good decision, consider the other capabilities of the IDEs, not only their integration with JUnit 5. Your research should include sources that present each IDE in detail, starting with the official documentation. Our purpose was to provide just-in-time information to evaluate IntelliJ IDEA, Eclipse, and NetBeans from the perspective of their relationship with the framework that is the topic of our book: JUnit 5.

Table 12.1 Comparison of JUnit 5's integration with IDEs

Feature	IntelliJ IDEA	Eclipse	NetBeans
Force running disabled tests	Yes	No	No
User-friendly display of @DisplayName tests	Yes	Yes	No
User-friendly display of dynamic tests	Yes	Yes	No
User-friendly display of nested tests	Yes	Yes	Does not execute nested tests
User-friendly display of parameterized tests	Yes	Yes	No
User-friendly display of repeated tests	Yes	Yes	No
User-friendly execution of tagged tests	Yes	Yes	Only with the help of the Surefire plugin
Run with code coverage from inside the IDE	Yes	Yes	Only with the help of the JaCoCo plugin

In the next chapter, we'll investigate the capabilities of JUnit 5 related to another important type of software it is used in conjunction with: continuous integration tools.

Summary

This chapter has covered the following:

- IDEs and the importance of using them with Java and JUnit 5 to gain productivity
- Executing JUnit 5 tests from IntelliJ IDEA, and the capabilities of this IDE for addressing the features of the testing framework
- Executing JUnit 5 tests from Eclipse, and the capabilities of this IDE for addressing the features of the testing framework
- Executing JUnit 5 tests from NetBeans, and the capabilities of this IDE for addressing the features of the testing framework
- Comparing IntelliJ IDEA, Eclipse, and NetBeans as IDE alternatives for working on JUnit 5 projects, focusing on how each of them can work with the new capabilities of the testing framework

13

Continuous integration with JUnit 5

This chapter covers

- Customizing and configuring Jenkins
- Practicing continuous integration on a development team
- Working on tasks in a continuous integration environment

Life is a continuous exercise in creative problem solving.

—Michael J. Gelb

In chapters 10 and 11, we implemented ways to automatically execute tests by using tools such as Maven and Gradle. The build then triggered our tests. Now it is time to go to the next level: automatically executing the build and the tests at regular intervals using other popular tools. In this chapter, we will explore the paradigm of continuous integration and demonstrate how to schedule projects to be built automatically at a particular time.

13.1 Continuous integration testing

Integration tests are usually time consuming, and as a single developer, you may not have all the different modules built on your machine. Therefore, it makes no sense to run all of the integration tests during development time. That is because, at development time, we are focused on our module, and all we want to know is whether it works as a single unit. At development time, we are mostly concerned that if we provide the right input data, the module behaves as expected and produces the expected result.

Integrating the execution of JUnit tests as part of your development cycle—[code : run : test : code], or [test : code : run : test] if you are using test-driven development (TDD)—is a very important concept in the sense that JUnit tests are unit tests: that is, they test a single component of your project in isolation. But many projects have a modular architecture, where different developers on the team work on different modules of the project. Each developer takes care of their own module and their own unit tests to make sure their module is well tested. Tested Data Systems Inc. is no exception: for a project that is under development, tasks are assigned to a few programmers who work independently and test their own code through JUnit 5 tests.

Different modules interact with each other, so we need to assemble all the modules to see how they work together. In order for the application to be test proven, we need other sorts of tests: integration or functional tests. We saw in chapter 5 that these test the interaction between different modules. At Tested Data Systems, the engineers will also need to test the integration of the code and of the modules they have developed independently.

TDD teaches us to test early and to test often. Executing all of our unit, integration, and functional tests every time we make a small change will slow us down immensely. To avoid this, we execute the unit tests only at development time—as early and as often as reasonable. What happens with the integration tests?

Integration tests should be executed independently from the development process. It's best to execute them at regular intervals (say, 15 minutes). This way, if something gets broken, we will hear about it in the next 15 minutes and have a better chance to fix it.

DEFINITION *Continuous integration (CI)*—“A software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.”¹

¹ This definition is taken from the marvelous article “Continuous Integration” by Martin Fowler and Matthew Foemmel. It can be found at www.martinfowler.com/articles/continuousIntegration.html.

To execute integration tests at regular intervals, we also need to have the system modules prepared and built. After the modules are built and the integration tests are executed, we want to see the results of the execution as quickly as possible. We need a software tool to perform all of the following steps automatically:

- 1 Check out the project from the source control system.
- 2 Build each of the modules and execute all of the unit tests to verify that the different modules work as expected in isolation.
- 3 Execute the integration tests to verify that the different modules integrate with each other as expected.
- 4 Publish the results from the tests executed in step 3.

Several questions may arise at this point. First, what is the difference between a human executing all these steps and a tool doing so? The answer is that there is no difference, and there shouldn't be! Apart from the fact that no one can stand to do such a job, if you take a close look at step 1, you see that we simply check out the project from the source control system. We do that as if we were a new member of the team and we just started with the project—with a clean checkout in an empty folder. Then, before moving on, we want to make sure all of the modules work properly in isolation—because if they don't, it doesn't make much sense to test whether they integrate well with the other modules, does it? The last step in the proposed scenario is to notify the developers about the test results. The notification could be done with an email, or simply by publishing the reports from the tests on a web server.

This overall interaction is illustrated in figure 13.1. The CI tool interacts with the source control system to get the project (1). After that, it uses the build tool that the project is using to build the project and execute different kinds of tests (2 and 3). Finally (4), the CI tool publishes the results and blows the whistle so that everybody can see them.

These four steps are very general and could be improved considerably. For instance, it would be better to check whether any changes have been made in the source control system before we start building. Otherwise, we waste the CPU power of the machine, knowing for sure that we will get the same results.

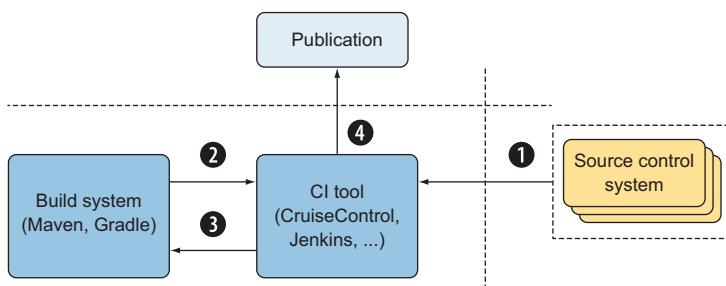


Figure 13.1 Continuous integration scheme: get the project; build it and execute tests; publish the results.

Now that we agree that we need a tool to integrate our projects continuously, let's look at one open source solution we might use: Jenkins (it makes no sense to reinvent the wheel when good tools are already available).

13.2 Introducing Jenkins

Jenkins (<https://jenkins.io>) is a CI tool alternative that is really worth considering. It has its roots in a similar CI project named Hudson. Initially developed at Sun Microsystems, Hudson was free software. Oracle bought Sun and intended to make Hudson a commercial version. The majority of the development community decided to continue the project under the name Jenkins in early 2011. Interest in Hudson strongly decreased after this, and Jenkins replaced it. Since February 2017, Hudson is no longer under maintenance.

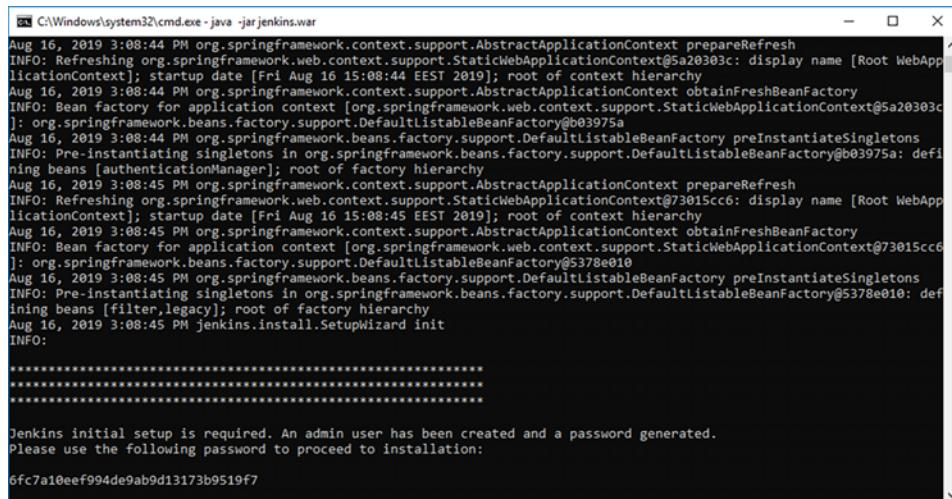
For the Jenkins installation procedure, see appendix D. After the installation is completed, a folder structure is created, in which the most important element is the `jenkins.war` file (figure 13.2).

Program Files (x86) > Jenkins >			
Name	Date modified	Type	Size
users	8/15/2019 5:22 PM	File folder	
war	8/15/2019 5:14 PM	File folder	
workflow-libs	8/15/2019 5:19 PM	File folder	
.lastStarted	8/15/2019 5:15 PM	LASTSTARTED File	0 KB
.owner	8/15/2019 9:26 PM	OWNER File	1 KB
config.xml	8/15/2019 5:23 PM	XML Document	2 KB
hudson.model.UpdateCenter.xml	8/15/2019 5:14 PM	XML Document	1 KB
hudson.plugins.git.GitTool.xml	8/15/2019 5:19 PM	XML Document	1 KB
identity.key.enc	8/15/2019 5:14 PM	Wireshark capture...	2 KB
jenkins.err.log	8/15/2019 8:49 PM	Text Document	157 KB
jenkins.exe	7/17/2019 6:08 AM	Application	363 KB
jenkins.exe.config	4/5/2015 10:05 AM	CONFIG File	1 KB
jenkins.install.InstallUtil.lastExecVersion	8/15/2019 5:23 PM	LASTEXECVERSIO...	1 KB
jenkins.install.UpgradeWizard.state	8/15/2019 5:23 PM	STATE File	1 KB
jenkins.model.JenkinsLocationConfigura...	8/15/2019 5:22 PM	XML Document	1 KB
jenkins.out.log	8/15/2019 5:14 PM	Text Document	1 KB
jenkins.pid	8/15/2019 5:13 PM	PID File	1 KB
jenkins.telemetry.Correlator.xml	8/15/2019 5:14 PM	XML Document	1 KB
jenkins.war	7/17/2019 6:08 AM	WAR File	75,566 KB
jenkins.wrapper.log	8/15/2019 9:57 PM	Text Document	3 KB
jenkins.xml	7/17/2019 6:08 AM	XML Document	3 KB
nodeMonitors.xml	8/15/2019 5:14 PM	XML Document	1 KB
secret.key	8/15/2019 5:14 PM	KEY File	1 KB
secret.key.not-so-secret	8/15/2019 5:14 PM	NOT-SO-SECRET ...	0 KB

Figure 13.2 The Jenkins installation folder with the `jenkins.war` file

Start the server from the Jenkins installation folder by executing the following command (figure 13.3):

```
java -jar jenkins.war
```



```
C:\Windows\system32\cmd.exe - java -jar jenkins.war
Aug 16, 2019 3:08:44 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.web.context.support.StaticWebApplicationContext@5a20303c: display name [Root WebApplicationContext]
Aug 16, 2019 3:08:44 PM org.springframework.context.support.AbstractApplicationContext obtainFreshBeanFactory
INFO: Bean factory for application context [org.springframework.web.context.support.StaticWebApplicationContext@5a20303c]
Aug 16, 2019 3:08:44 PM org.springframework.beans.factory.support.DefaultListableBeanFactory@0b3975a
Aug 16, 2019 3:08:44 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@0b3975a: defining beans [authenticationManager]; root of factory hierarchy
Aug 16, 2019 3:08:45 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.web.context.support.StaticWebApplicationContext@73015cc6: display name [Root WebApplicationContext]
Aug 16, 2019 3:08:45 PM org.springframework.context.support.AbstractApplicationContext obtainFreshBeanFactory
INFO: Bean factory for application context [org.springframework.web.context.support.StaticWebApplicationContext@73015cc6]
Aug 16, 2019 3:08:45 PM org.springframework.beans.factory.support.DefaultListableBeanFactory@5378e010
Aug 16, 2019 3:08:45 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@5378e010: defining beans [filter,legacy]; root of factory hierarchy
Aug 16, 2019 3:08:45 PM jenkins.install.SetupWizard init
INFO:
*****
*****
*****
Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:
6fc7a10eef994de9ab9d13173b9519f7
```

Figure 13.3 Launching the Jenkins server from the command line

To start using the server, navigate to the <http://localhost:8080/> URL. You should see something similar to figure 13.4. Unlock Jenkins using the password generated in the previous step (see figure 13.3).

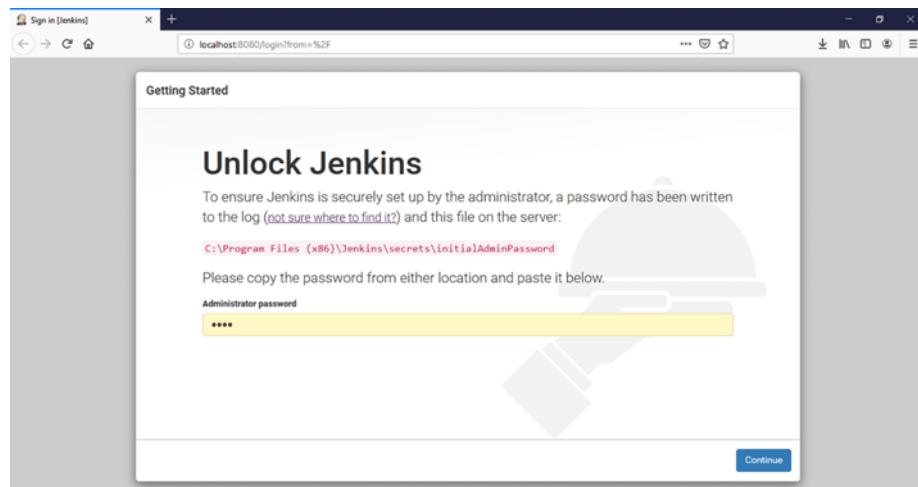


Figure 13.4 Accessing Jenkins from the web interface

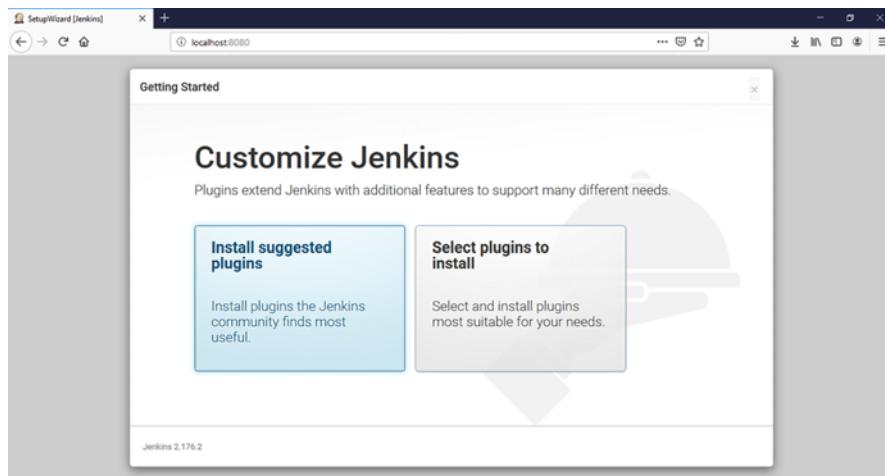


Figure 13.5 Customizing the Jenkins window: choosing the plugins to be installed

After entering the password, you will see the window shown in figure 13.5. Choose Install Suggested Plugins to install several useful plugins in a typical configuration: among these are the folders plugin (for grouping tasks), the monitoring plugin (for managing charts of CPU, HTTP response time, and memory), and the metrics plugin (provides health checks). For details about the plugins, see the Jenkins documentation at <https://jenkins.io/doc>.

After the plugins are installed, a window opens in which you are required to create the first admin user by providing new credentials (figure 13.6). Click Save and Continue, and you are directed to the window where you can start using Jenkins (figure 13.7).

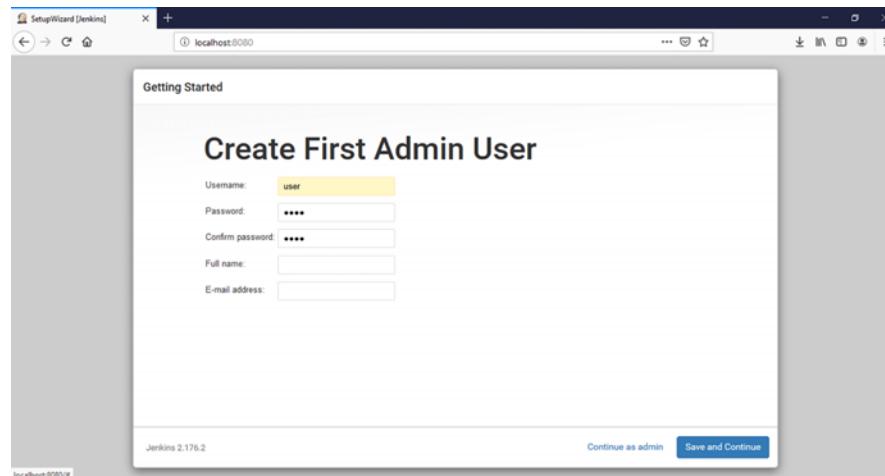


Figure 13.6 Creating the first admin user by providing new credentials

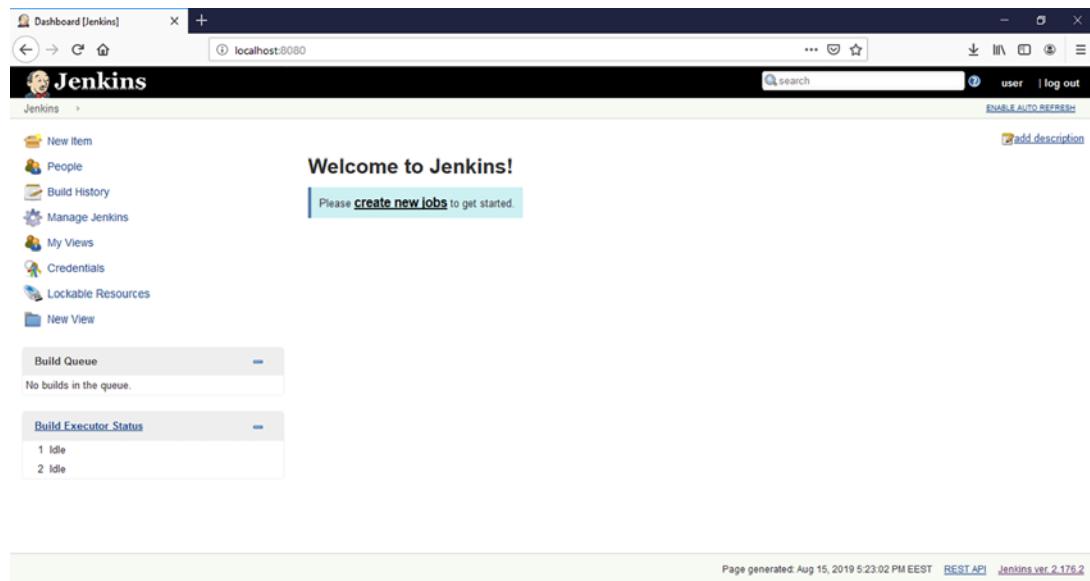


Figure 13.7 The Welcome to Jenkins page

13.3 Practicing CI on a team

The engineers at Tested Data Systems have decided to investigate using Jenkins for some of the projects developed inside the company. One of the projects under development is a flight-management application. John and Beth are developers working on this project, but have different tasks: John is responsible for developing the part concerning passengers, while Beth is responsible for developing the part concerning flights. They work independently and write their own code and tests.

John has implemented the `Passenger` and `PassengerTest` classes shown in listings 13.1 and 13.2, respectively.

Listing 13.1 Passenger class

```
[...]
public class Passenger {

    private String identifier;
    private String name;
    private String countryCode;
    [...]

    public Passenger(String identifier, String name,
                     String countryCode) {
        if (!Arrays.asList(Locale.getISOCountries())
                     .contains(countryCode)) {
            throw new RuntimeException("Invalid country code");
        }
    }
}
```

```

    }
    this.identifier = identifier;
    this.name = name;
    this.countryCode = countryCode;
}

public String getIdentifier() {
    return identifier;
}

public String getName() {
    return name;
}

public String getCountryCode() {
    return countryCode;
}

@Override
public String toString() {
    return "Passenger " + getName() + " with identifier: " +
        getIdentifier() + " from " + getCountryCode();
}
}

```

In this listing:

- A passenger is described through an identifier, a name, and a country code ①.
- The Passenger constructor checks the validity of the country code and, if this verification passes, sets the previously defined fields: identifier, name, and countryCode ②.
- The class defines getters for the three previously defined fields: identifier, name, and countryCode ③.
- The `toString` method is overridden to allow the information from a passenger to be displayed ④.

Listing 13.2 PassengerTest class

```

[...]
public class PassengerTest {

    @Test
    public void testPassengerCreation() {
        Passenger passenger = new Passenger("123-45-6789",
                                            "John Smith", "US");
        assertNotNull(passenger);
    }

    @Test
    public void testInvalidCountryCode() {
        assertThrows(RuntimeException.class,
                     () ->{
}

```

```

Passenger passenger = new Passenger("900-45-6789",
                                    "John Smith", "GJ");
    }) ;
}

@Test
public void testPassengerToString() {
    Passenger passenger = new Passenger("123-45-6789",
                                         "John Smith", "US");
    assertEquals("Passenger John Smith with identifier:
123-45-6789 from US", passenger.toString());
}
}

```

In this listing:

- A test checks the creation of a passenger with the correct parameters ①.
- A test checks that the `Passenger` constructor throws an exception when the `countryCode` parameter is invalid ②.
- A test checks the correct behavior of the `toString` method ③.

Beth has implemented the `Flight` and `FlightTest` classes, shown in listings 13.3 and 13.4.

Listing 13.3 Flight class

```

[...]
public class Flight {

    private String flightNumber;
    private int seats;
    private Set<Passenger> passengers = new HashSet<>();

    private static String flightNumberRegex = "^[A-Z]{2}\\d{3,4}$";
    private static Pattern pattern =
        Pattern.compile(flightNumberRegex);

    public Flight(String flightNumber, int seats) {
        Matcher matcher = pattern.matcher(flightNumber);
        if(!matcher.matches()) {
            throw new RuntimeException("Invalid flight number");
        }
        this.flightNumber = flightNumber;
        this.seats = seats;
    }

    public String getFlightNumber() {
        return flightNumber;
    }

    public int getNumberOfPassengers () {
        return passengers.size();
    }
}

```

```
public boolean addPassenger(Passenger passenger) {  
    if(getNumberOfPassengers() >= seats) {  
        throw new RuntimeException("Not enough seats for flight "  
            + getFlightNumber());  
    }  
    return passengers.add(passenger);  
}  
  
public boolean removePassenger(Passenger passenger) {  
    return passengers.remove(passenger);  
}  
}
```

In this listing:

- A flight is described using the flight number, the number of seats, and the set of passengers ①.
- `flightNumberRegex` provides the regular expression that describes a correctly formed flight number ②. A *regular expression* is a sequence of characters that defines a search pattern. In this case, the regular expression requires that a flight number start with two capital letters followed by three or four digits. This regular expression is the same for all flights, so the field is static.
- A `Pattern` instance is created from the string regular expression ③. The `Pattern` class belongs to the `java.util.regex` package and is the main access point of the Java regular expression API. Whenever you need to work with regular expressions in Java, you start by creating such an object. This field is also the same for all flights, so it is static.
- In the constructor of the `Flight` class, we create a `Matcher` object ④. A `Matcher` is used to search through text for occurrences of a regular expression defined through a `Pattern`.
- If `flightNumber` does not match the required regular expression, an exception is thrown ⑤. Otherwise, the instance fields are set with the values of the parameters ⑥.
- The class defines getters for the flight number and the passenger number ⑦.
- The `addPassenger` method tries to add a passenger to the current flight. If the number of passengers exceeds the number of seats, an exception is thrown. If the passenger is successfully added, it returns `true`. If the passenger already exists, it returns `false` ⑧.
- The `removePassenger` method removes a passenger from the current flight. If the passenger is successfully removed, it returns `true`. If the passenger does not exist, it returns `false` ⑨.

Listing 13.4 FlightTest class

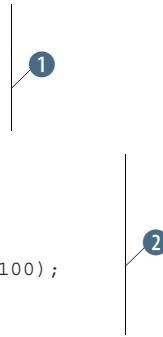
```
[...]  
public class FlightTest {
```

```

    @Test
    public void testFlightCreation() {
        Flight flight = new Flight("AA123", 100);
        assertNotNull(flight);
    }

    @Test
    public void testInvalidFlightNumber() {
        assertThrows(RuntimeException.class,
            () -> {
                Flight flight = new Flight("AA12", 100);
            });
    }
}

```



In this listing:

- A test checks the creation of a flight with the correct parameters 1.
- A test checks that the Flight constructor throws an exception when the flightNumber parameter is invalid 2.

Additionally, Beth has written an integration test between the Passenger and Flight classes, because she is handling the Flight class (which also works with Passenger objects).

Listing 13.5 FlightWithPassengersTest class

```

[...]
public class FlightWithPassengersTest {
    private Flight flight = new Flight("AA123", 1);

    @Test
    public void testAddRemovePassengers() throws IOException {
        Passenger passenger = new Passenger("124-56-7890",
            "Michael Johnson", "US");
        assertTrue(flight.addPassenger(passenger));
        assertEquals(1, flight.getNumberOfPassengers ());

        assertFalse(flight.removePassenger(passenger));
        assertEquals(0, flight.getNumberOfPassengers ());
    }

    @Test
    public void testNumberOfSeats() {
        Passenger passenger1 = new Passenger("124-56-7890",
            "Michael Johnson", "US");
        flight.addPassenger(passenger1);
        assertEquals(1, flight.getNumberOfPassengers ());

        Passenger passenger2 = new Passenger("127-23-7991",
            "John Smith", "GB");
        assertThrows(RuntimeException.class,
            () -> flight.addPassenger(passenger2));
    }
}

```



In this listing:

- We create a flight with a correct flight number and one seat—this is for convenient testing purposes ①.
- A test creates a passenger ② to be added to ③ and removed from ④ the flight. At each step, we check that the operation is successful and the correct number of passengers are on the flight.
- A test creates a passenger to be added to a flight ⑤, followed by another passenger that will exceed the number of seats on the flight ⑦. We first check the correct number of passengers ⑥ and then the fact that an exception is thrown when the number of passengers exceeds the number of seats ⑧.

The code developed for this project is managed by Git on a CI machine. Git is a distributed version control system that tracks changes in source code. It can be downloaded from <https://git-scm.com/downloads>; the folder Git/cmd containing the git executable is on the path of the OS installed on the CI machine. For our examples, we'll use some basic Git commands and explain what they do and why. For more comprehensive details about Git, check the documentation.

The folder where the project sources reside is a Git repository. To make the folder a Git repository, the machine administrator ran the following command in the folder:

```
git init
```

This command creates a new Git repository. The folder is shown in figure 13.8: it contains the pom.xml Maven file, the src folder containing the Java sources, and the .git folder with meta-information for the Git distributed version control system.

Name	Date modified	Type	Size
.git	8/26/2019 6:04 PM	File folder	
src	8/26/2019 3:18 PM	File folder	
pom.xml	8/26/2019 4:16 PM	XML Document	2 KB

Figure 13.8 The src folder on the CI machine managed by Git

13.4 Configuring Jenkins

Configuring Jenkins is all done through the web interface. We would like to set up a project under its management. As Jenkins is running on the CI machine, we go back to its web interface, accessible at <http://localhost:8080/> (figure 13.9). No item is defined so far, so we'll click the New Item option on the left side to create a new CI job.

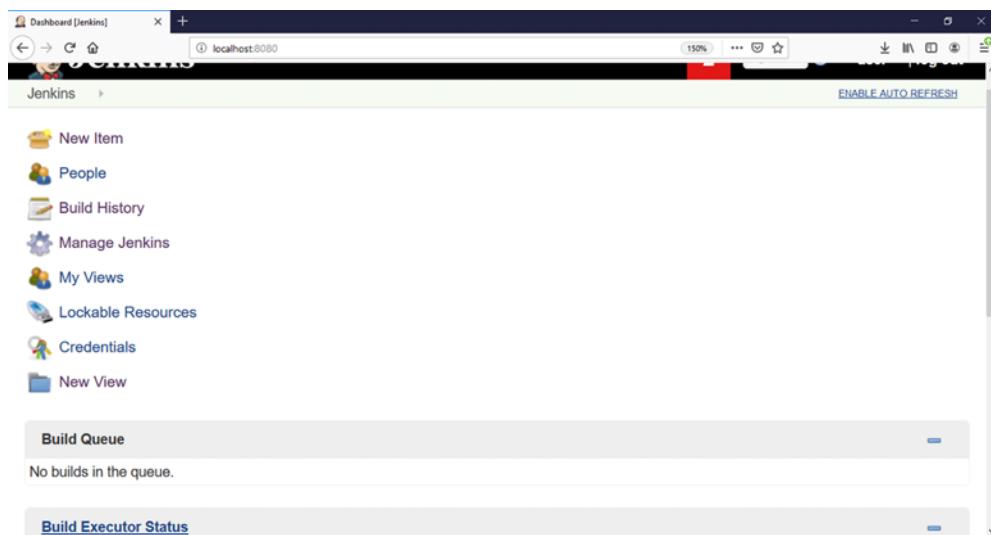


Figure 13.9 The Jenkins main page with no jobs created so far

We choose the Freestyle Project option, insert the item name ch13-continuous (figure 13.10), and then click the OK button. On the newly displayed page, we choose Git as the Source Code Management option; fill in the Repository URL (figure 13.11); go to the bottom of the page and choose Build > Add Build Step > Invoke Top Level Maven Targets; enter `clean install` (figure 13.12); and click the Save button.

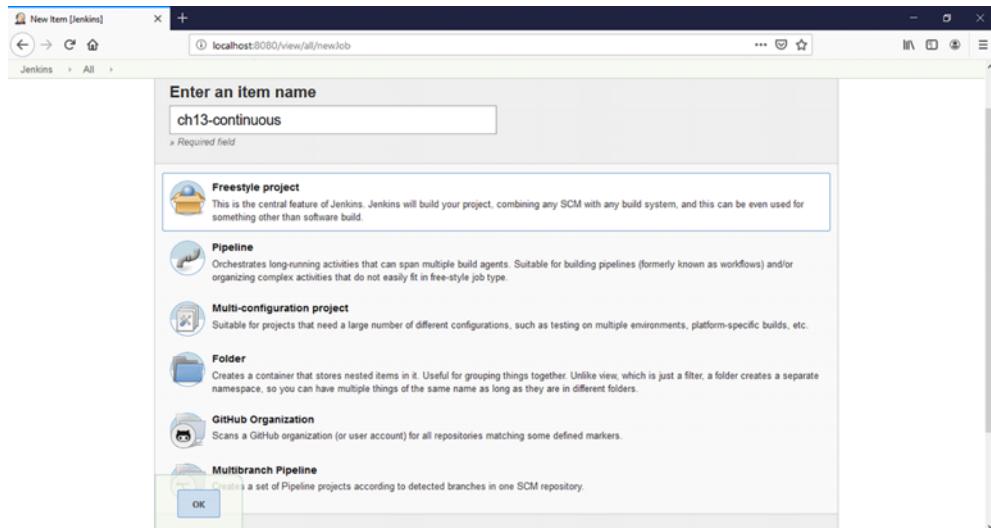


Figure 13.10 Creating a new CI job in Jenkins

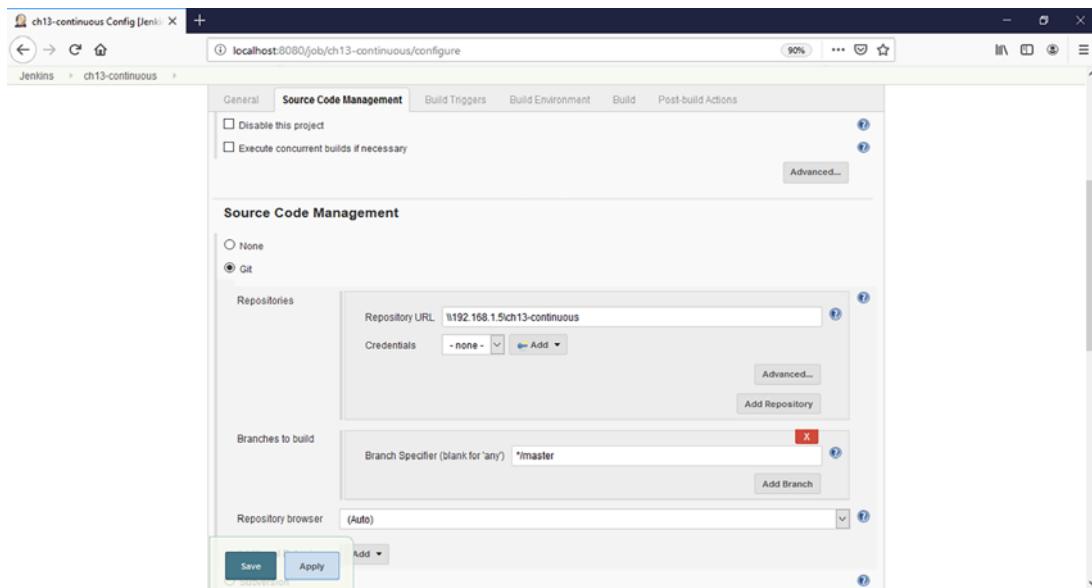


Figure 13.11 Defining the repository URL containing the source code

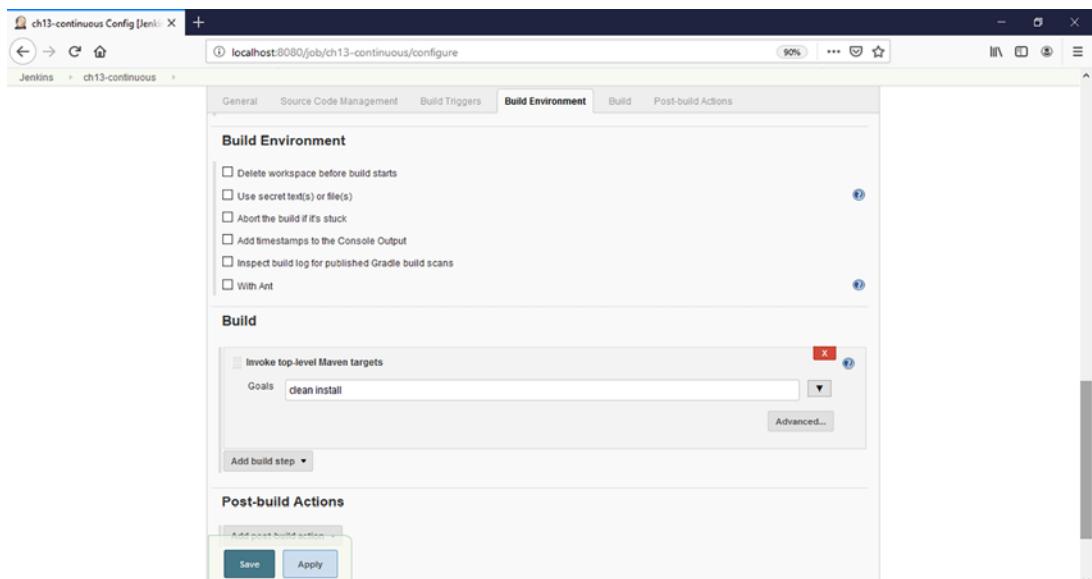
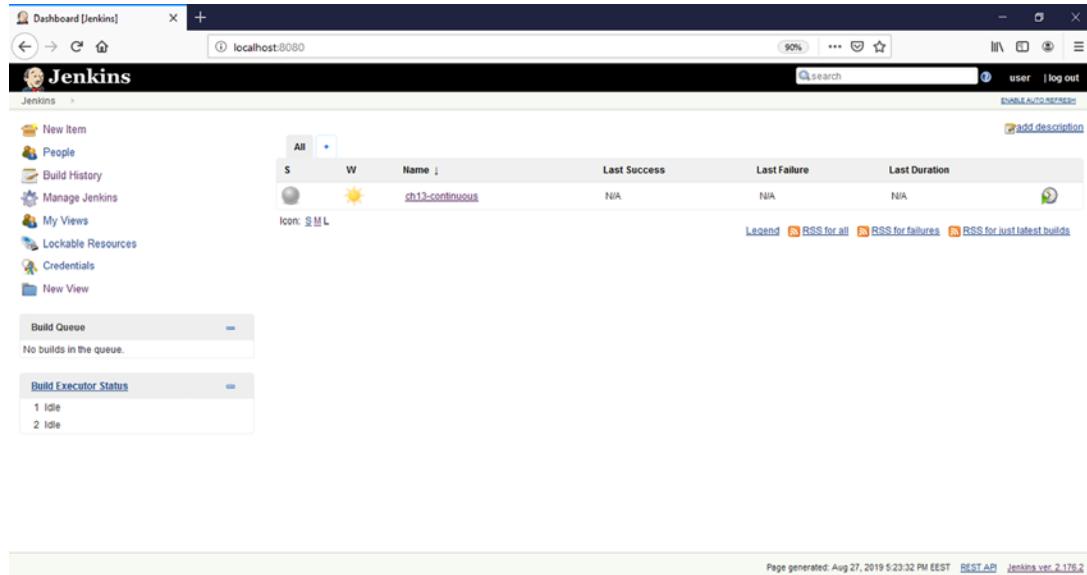


Figure 13.12 The build configuration for the newly created CI project

On the Jenkins main page, we now see the newly created CI project (figure 13.13). We click the Build button on the right side of the project and wait for the project to be executed (figure 13.14).

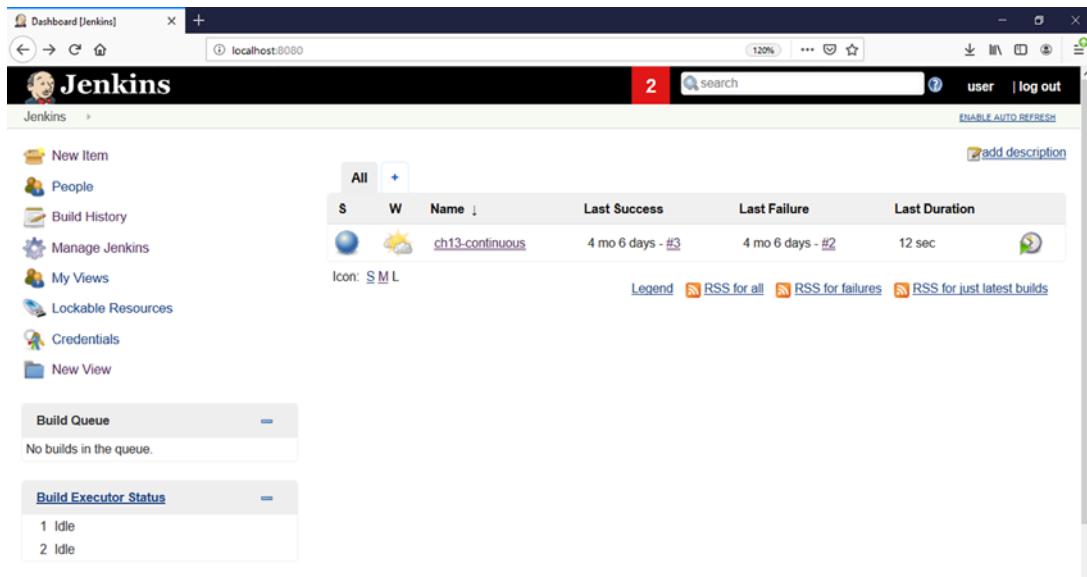


The screenshot shows the Jenkins dashboard at localhost:8080. The left sidebar contains links for New Item, People, Build History, Manage Jenkins, My Views, Lockable Resources, Credentials, and New View. The main content area displays a table of Jenkins projects. The 'ch13-continuous' project is listed with the following details:

S	W	Name	Last Success	Last Failure	Last Duration
●	☀	ch13-continuous	N/A	N/A	N/A

Below the table, a legend provides links for RSS feeds: RSS for all, RSS for failures, and RSS for just latest builds. The status bar at the bottom right indicates the page was generated on Aug 27, 2019, at 5:23:32 PM EEST, and shows Jenkins version 2.176.2.

Figure 13.13 The newly created CI project



The screenshot shows the Jenkins dashboard at localhost:8080. The left sidebar contains links for New Item, People, Build History, Manage Jenkins, My Views, Lockable Resources, Credentials, and New View. The main content area displays a table of Jenkins projects. The 'ch13-continuous' project is listed with the following details:

S	W	Name	Last Success	Last Failure	Last Duration
●	☀	ch13-continuous	4 mo 6 days - #3	4 mo 6 days - #2	12 sec

Below the table, a legend provides links for RSS feeds: RSS for all, RSS for failures, and RSS for just latest builds. The status bar at the bottom right indicates the page was generated on Aug 27, 2019, at 5:23:32 PM EEST, and shows Jenkins version 2.176.2.

Figure 13.14 The result of the first execution of a build on the CI machine using Jenkins

Now we are sure that everything is fine from the CI point of view. The work created by John and Beth runs fine and also integrates well.

At the beginning of this chapter, we said that at development time, a programmer is usually focused on their module and wants to know that it is working as a single unit. At development time, the programmer only executes unit tests. Integration tests are executed independently from the development process—and now we have a fully configured Jenkins project to take care of this. We'll demonstrate how CI will help John and Beth quickly try their work head to head and easily fix any integration problems.

13.5 Working on tasks in a CI environment

John has a new task for the flight-management project: passengers must be able to join a flight themselves, not just be added to the flight. This is necessary because the new interactive system will allow passengers to choose their flights. Currently, we can add passengers to flights, but if we look at a `Passenger` object, we have no way to know which flight they are on.

To address this task, John considers introducing `Flight` as an instance variable of the `Passenger` class. This way, a passenger will be able to individually choose and join a flight, as the new interactive system requires. Note that there is a bidirectional reference between the `Flight` and `Passenger` classes. The `Flight` class already contains the set of `Passengers`:

```
private Set<Passenger> passengers = new HashSet<>();
```

John adds a test in the `FlightWithPassengersTest` class and modifies the existing `testAddRemovePassengers`.

Listing 13.6 testPassengerJoinsFlight from FlightWithPassengersTest

```
@Test
public void testPassengerJoinsFlight() {
    Passenger passenger = new Passenger("123-45-6789",
                                         "John Smith", "US");
    Flight flight = new Flight("AA123", 100);
    passenger.joinFlight(flight);
    assertEquals(flight, passenger.getFlight());
    assertEquals(1, flight.getNumberOfPassengers());
}

@Test
public void testAddRemovePassengers() throws IOException {
    Passenger passenger = new Passenger("124-56-7890",
                                         "Michael Johnson", "US");
    flight.addPassenger(passenger);
    assertEquals(1, flight.getNumberOfPassengers());
    assertEquals(flight, passenger.getFlight());

    flight.removePassenger(passenger);
    assertEquals(0, flight.getNumberOfPassengers());
    assertEquals(null, passenger.getFlight());
}
```

In this listing:

- Johns creates a passenger ① and a flight ②.
- He makes the passenger join the flight ③.
- He checks that the passenger has the previously defined flight assigned ④ and that the number of passengers from the flight is now one ⑤.
- In the existing `testAddRemovePassengers`, after the flight has added a passenger, he checks whether the flight has been set on the passenger's side ⑥. After the flight has removed a passenger, no flight is set on the passenger's side ⑦.

John also adds the following code to the `Passenger` class.

Listing 13.7 Changes to the Passenger class

```
[...]
private Flight flight;           ← ①
[...]

public Flight getFlight() {      | ②
    return flight;
}

public void setFlight(Flight flight) {  | ③
    this.flight = flight;
}

public void joinFlight(Flight flight) {
    Flight previousFlight = this.flight;
    if (null != previousFlight) {
        if(!previousFlight.removePassenger(this)) {
            throw new RuntimeException("Cannot remove passenger");
        }
    }
    setFlight(flight);
    if(null != flight) {
        if(!flight.addPassenger(this)) {
            throw new RuntimeException("Cannot add passenger");
        }
    }
}
```

In this listing:

- John adds a `Flight` field to the `Passenger` class ①.
- He creates a getter ② and a setter ③ for the newly added field.
- In the `joinFlight` method, he checks whether a previous flight exists for the passenger and, if so, removes the passenger from it. If the removal is not successful, the method throws an exception ④. Then it sets the flight for the passenger ⑤. If the new flight is not null, the method adds the passenger to it. If the passenger cannot be added, the method throws an exception ⑥.

John needs to push his code to the CI server located at 192.168.1.5. The local project is also under the management of a Git server; it is a clone of the code on the CI server. This clone was originally created using this Git command:

```
git clone \\192.168.1.5\ch13-continuous
```

To push the code, John executes a few Git commands, starting with this one:

```
git add *.java
```

The `git add` command includes updates to a particular file in the next commit. Executing this command will include all `.java` files that have been changed for the next commit. Changes are not actually recorded until we run `git commit`.

So, John will record the changes into his local repository by running the following command:

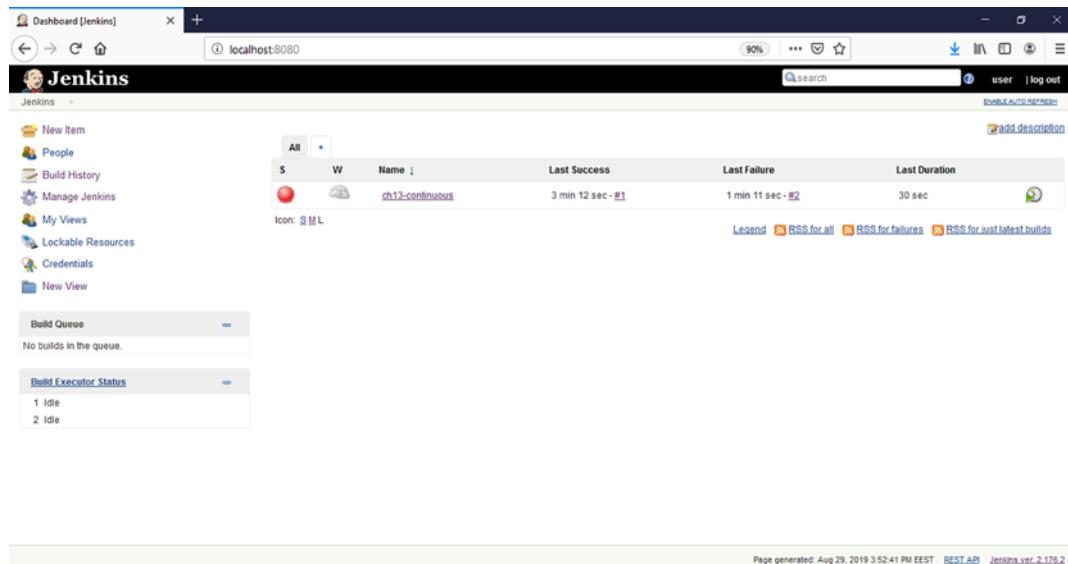
```
git commit -m "Allow the passenger to make the individual choice of a flight"
```

The changes are committed to the local repository with an informative message explaining the task the changes belong to: “Allow the passenger to make the individual choice of a flight.”

Now, John needs to do just one more thing in order for the code to arrive on the CI server. He executes this command:

```
git push
```

After the code is pushed to the CI server, a new build is launched on this machine, and it will fail (figure 13.15). By accessing the console of the project from Jenkins



The screenshot shows the Jenkins dashboard at `localhost:8080`. The left sidebar contains links for New Item, People, Build History, Manage Jenkins, My Views, Lockable Resources, Credentials, and New View. The main area displays a table of builds for the 'ch13-continuous' project. The table has columns for Status (S), Work (W), Name, Last Success, Last Failure, and Last Duration. The first build is marked as Failed (red icon), with the last success at '3 min 12 sec - #1' and the last failure at '1 min 11 sec - #2' (green icon). The last duration is '30 sec'. Below the table, there are links for RSS feeds: 'RSS for all' (green), 'RSS for failures' (orange), and 'RSS for just latest builds' (green). The bottom of the page shows a footer with the text 'Page generated: Aug 29, 2019 3:52:41 PM EEST' and 'Jenkins ver. 2.176'.

Figure 13.15 The result of the build execution after pushing the changes to the tests

(click the ch13-continuous link, then the build number dropdown icon in the Build History, and then Console Output), we can see that the modified test fails (figure 13.16).

```

Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/platform/junit-platform-commons/1.3.1/junit-platform-commons-1.3.1.jar
(78 kB at 187 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/platform/junit-platform-launcher/1.3.1/junit-platform-launcher-1.3.1.jar
(85 kB at 219 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-junit-platform/2.22.1/surefire-junit-platform-2.22.1.jar
(66 kB at 146 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/platform/junit-platform-engine/1.3.1/junit-platform-engine-1.3.1.jar
(135 kB at 288 kB/s)
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.manning.junitbook.ch13.flights.FlightTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0. Time elapsed: 0.022 s - in com.manning.junitbook.ch13.flights.FlightTest
[INFO] Running com.manning.junitbook.ch13.flights.FlightWithPassengerTest
[ERROR] Tests run: 3, Failures: 1, Errors: 0, Skipped: 0. Time elapsed: 0 s <<< FAILURE! - in
com.manning.junitbook.ch13.flights.FlightWithPassengerTest
[ERROR] testAddRemovePassengers Time elapsed: 0 s <<< FAILURE!
org.opentest4j.AssertionFailedError: expected: <com.manning.junitbook.ch13.flights.Flight@c46bcd4> but was: <null>
at com.manning.junitbook.ch13.flights.FlightWithPassengerTest.testAddRemovePassengers(FlightWithPassengerTest.java:29)
[INFO] Running com.manning.junitbook.ch13.passengers.PassengerTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0. Time elapsed: 0 s - in com.manning.junitbook.ch13.passengers.PassengerTest
[INFO] Results:
[INFO] -----
[ERROR] Failures:
[ERROR] FlightWithPassengerTest.testAddRemovePassengers:29 expected: <com.manning.junitbook.ch13.flights.Flight@c46bcd4> but was: <null>
[INFO]
[ERROR] Tests run: 8, Failures: 1, Errors: 0, Skipped: 0
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 34.761 s
[INFO] Finished at: 2019-08-29T15:52:09+03:00
[INFO] -----
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.22.1:test (default-test) on project ch13: There are test

```

Figure 13.16 The console of the Jenkins project showing the failure after running the build

The error message says the following:

```

[INFO] Running
com.manning.junitbook.ch13.flights.FlightWithPassengerTest
[ERROR] Tests run: 3, Failures: 1, Errors: 0, Skipped: 0,
Time elapsed: 0 s <<< FAILURE! - in
com.manning.junitbook.ch13.flights.FlightWithPassengerTest
[ERROR] testAddRemovePassengers Time elapsed: 0 s <<< FAILURE!
org.opentest4j.AssertionFailedError: expected:
<com.manning.junitbook.ch13.flights.Flight@c46bcd4> but was:
<null> at
com.manning.junitbook.ch13.flights.FlightWithPassengerTest.
testAddRemovePassengers(FlightWithPassengerTest.java:29)

```

After the build fails and the console output is investigated, the programmers realize that it is a problem of integration between Flight and Passenger. They need to make sure the relationship between Passenger and Flight is bidirectional: if the Passenger has a reference to a Flight, the Flight also has a reference to the Passenger, and vice versa.

Beth is the developer working on Flight, so she will take care of this issue. Beth's local project is also under the management of Git. It is a clone of the code on the CI server. This clone was originally created using this Git command:

```
git clone \\192.168.1.5\ch13-continuous
```

To get the updated code from John, Beth executes the following Git command:

```
git pull
```

This way, Beth has the latest updates on her machine. To fix the issue, Beth modifies the existing Flight class: to be precise, the `addPassenger` and `removePassenger` methods.

Listing 13.8 Modified Flight class

```
public boolean addPassenger(Passenger passenger) {
    if(getNumberOfPassengers() >= seats) {
        throw new RuntimeException("Not enough seats for flight "
            + getFlightNumber());
    }
    passenger.setFlight(this);           ↪
    return passengers.add(passenger);   ↪ 1
}

public boolean removePassenger(Passenger passenger) {
    passenger.setFlight(null);          ↪
    return passengers.remove(passenger); ↪ 2
}
```

Beth has added two lines of code to do the following:

- When a passenger is added to a flight, the flight is also set on the passenger's side ①.
- When a passenger is removed from a flight, the flight is also removed from the passenger's side ②.

Beth sends her changes to the CI server by executing the following Git commands:

```
git add *.java
git commit -m "Adding integration code for a passenger join/unjoin"
git push
```

After the code is pushed to the CI server, a new build is launched on this machine, and it succeeds (figure 13.17).

We can see the benefits of CI: integration problems are quickly signaled, and the developers can fix the issues immediately. JUnit 5 and Jenkins cooperate very effectively!

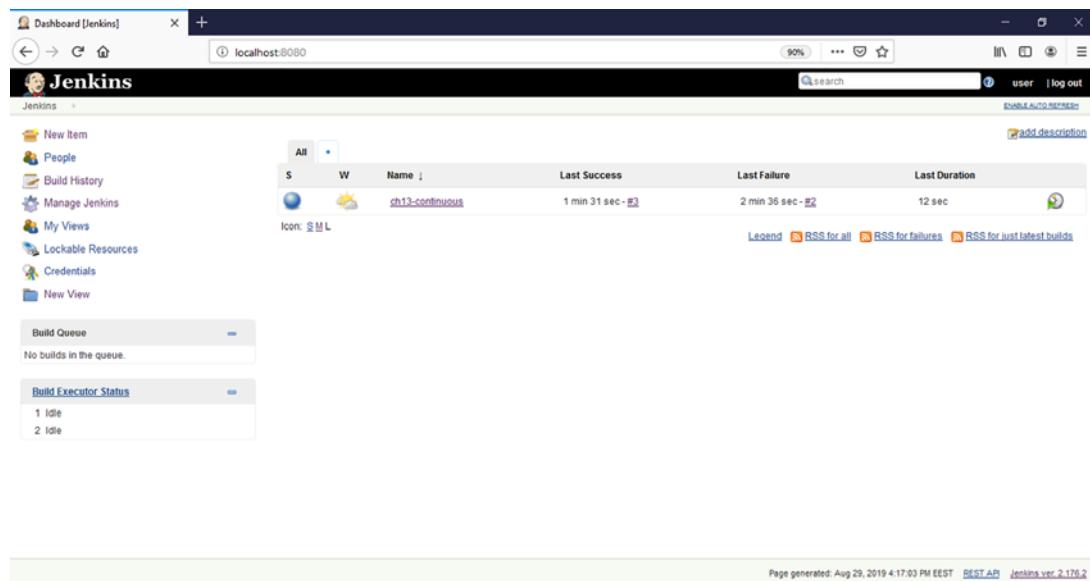


Figure 13.17 The Jenkins build after the necessary integration code is introduced in the `Flight` class

The next chapter will start part 4 of the book, which is dedicated to working with modern frameworks and JUnit 5. We'll begin by examining the JUnit 5 extension model.

Summary

This chapter has covered the following:

- Continuous integration, which is a software development practice that has members of a team integrate their work frequently.
- The benefits that CI provides for developing team Java applications: verifying each integration with an automated build that detects integration errors as quickly as possible, thus reducing integration problems and letting developers fix the issues immediately.
- Using Jenkins as a CI tool in collaborative projects. (Jenkins builds use JUnit 5 for execution).
- Demonstrating collaborative work on a team that practices CI: working on implementing tasks using Jenkins as a CI server and Git as a version control system.

Part 4

Working with modern frameworks and JUnit 5

This part of the book explores working with JUnit 5 and commonly used frameworks. Chapter 14 is dedicated to the implementation of JUnit 5 extensions as alternatives to JUnit 4 rules and runners. This is useful for working with custom test extensions as well as for easily working with modern frameworks that provide their own extensions for creating efficient JUnit 5 tests.

We continue by introducing HtmlUnit and Selenium in chapter 15. We will show you how to test the presentation layer with these tools. We go into detail about how to set up your projects and also some best practices for testing the presentation layer.

We dedicate chapters 16 and 17 to testing one of the most useful frameworks today: Spring. Spring is an open source application framework and inversion of control container for the Java platform. Spring is largely used today for creating Java SE and Java EE applications. It includes several separate frameworks, including the Spring Boot convention-over-configuration solution for creating applications that you can run directly.

Chapter 18 examines testing REST applications. Representational State Transfer represents an application program interface that uses HTTP requests to GET, PUT, PATCH, POST, and DELETE data. Chapter 19 discusses alternatives for testing database applications, including JDBC, Spring, and Hibernate.

JUnit 5 extension model

This chapter covers

- Creating JUnit 5 extensions
- Implementing JUnit 5 tests using available extension points
- Developing an application with tests extended by JUnit 5 extensions

The wheel is an extension of the foot, the book is an extension of the eye, clothing an extension of the skin, electric circuitry an extension of the central nervous system.

—Marshall McLuhan

In chapter 4, we demonstrated ways to extend the execution of tests. We examined JUnit 4 rules and JUnit 5 extensions face to face, and we analyzed how to migrate from the old JUnit 4 rules to the new extension model developed by JUnit 5. We also emphasized a few well-known extensions, such as `MockitoExtension` and `SpringExtension`. In chapter 8, we implemented tests using mock objects and `MockitoExtension`, and we'll implement more tests using `SpringExtension` in

chapter 16. In this chapter, we demonstrate the systematic creation of custom extensions and their applicability to creating JUnit 5 tests.

14.1 Introducing the JUnit 5 extension model

Although JUnit 4 provides extension points through runners and rules (see chapter 3), the JUnit 5 extension model consists of a single concept: the `Extension` API. `Extension` itself is just a *marker interface* (or *tag* or *token interface*)—an interface with no fields or methods inside. It is used to mark the fact that the class implementing an interface of this category has some special behavior. Among the best-known Java marker interfaces are `Serializable` and `Cloneable`.

JUnit 5 can extend the behavior of tests' classes or methods, and these extensions can be reused by many tests. A JUnit 5 extension is connected to an occurrence of a particular event during the execution of a test. This kind of event is called an *extension point*. At the moment when such a point is reached in a test's life cycle, the JUnit engine automatically calls the registered extension.

The available extension points are as follows:

- *Conditional test execution*—Controls whether a test should be run
- *Life-cycle callback*—Reacts to events in the life cycle of a test
- *Parameter resolution*—At runtime, resolves the parameter received by a test
- *Exception handling*—Defines the behavior of a test when it encounters certain types of exceptions
- *Test instance postprocessing*—Executed after an instance of a test is created

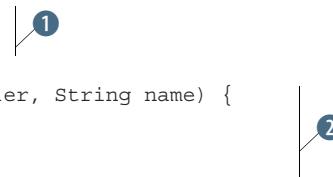
Note that the extensions are largely used inside frameworks and build tools. They can also be used for application programming, but not to the same extent. The creation and usage of extensions follow common principles; this chapter will present examples appropriate for regular application development.

14.2 Creating a JUnit 5 extension

Tested Data Systems is developing a flight-management application. Harry is working on this project, developing and testing the part related to passengers. Currently, the `Passenger` and `PassengerTest` classes look like listings 14.1 and 14.2, respectively.

Listing 14.1 Passenger class

```
public class Passenger {  
  
    private String identifier;  
    private String name;  
  
    public Passenger(String identifier, String name) {  
        this.identifier = identifier;  
        this.name = name;  
    }  
}
```



```
public String getIdentifier() {
    return identifier;
}

public String getName() {
    return name;
}

@Override
public String toString() {
    return "Passenger " + getName() + " with identifier: " +
        getIdentifier();
}
```

3

4

In this listing:

- A passenger is described through an identifier and a name ①.
- The Passenger constructor sets the identifier and name fields ②.
- The class defines getters for identifier and name ③.
- The `toString` method is overridden to allow passenger information (the name and identifier) to be displayed ④.

Listing 14.2 PassengerTest class

```
public class PassengerTest {

    @Test
    void testPassenger() throws IOException {
        Passenger passenger = new Passenger("123-456-789", "John Smith");
        assertEquals("Passenger John Smith with identifier: 123-456-789",
            passenger.toString());
    }
}
```

This class has a single test that checks the behavior of the `toString` method.

Harry's next task involves the conditional execution of tests depending on the context. There are three types of contexts—regular, low, and peak—depending on the number of passengers during a certain period. The task requires the tests to be executed only in the regular and low contexts. The tests are not executed during peak periods, as an overloaded system would cause problems for the company.

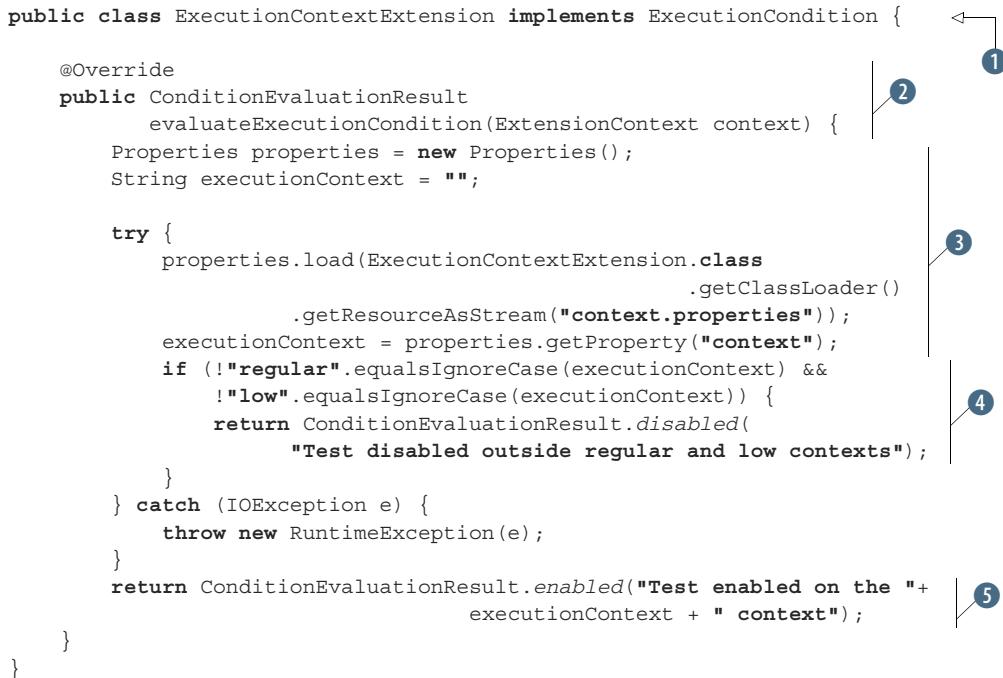
To fulfill this task, Harry creates a JUnit extension that controls whether a test should be run. Then, he extends the tests with the help of this extension. Such an extension is defined by implementing the `ExecutionCondition` interface. Harry creates an `ExecutionContextExtension` class that implements the `ExecutionCondition` interface and overrides the `evaluateExecutionCondition()` method.

The method verifies whether a property representing the current context name equals "regular" or "low" and, if not, disables the test.

Listing 14.3 ExecutionContextExtension class

```
public class ExecutionContextExtension implements ExecutionCondition { ← 1
    @Override
    public ConditionEvaluationResult
        evaluateExecutionCondition(ExtensionContext context) { 2
        Properties properties = new Properties();
        String executionContext = "";

        try {
            properties.load(ExecutionContextExtension.class
                .getClassLoader()
                .getResourceAsStream("context.properties"));
            executionContext = properties.getProperty("context");
            if (!"regular".equalsIgnoreCase(executionContext) &&
                !"low".equalsIgnoreCase(executionContext)) {
                return ConditionEvaluationResult.disabled(
                    "Test disabled outside regular and low contexts");
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return ConditionEvaluationResult.enabled("Test enabled on the " +
            executionContext + " context");
    }
}
```



In this listing:

- Harry creates a conditional test execution extension by implementing the `ExecutionCondition` interface ①.
- He overrides the `evaluateExecutionCondition` method, which returns a `ConditionEvaluationResult` to determine whether a test is enabled ②.
- He creates a `Properties` object that loads the properties from the resource `context.properties` file. He keeps the value of the `context` property ③.
- If the `context` property is something other than "regular" or "low", the returned `ConditionEvaluationResult` means the test is disabled ④.
- Otherwise, the returned `ConditionEvaluationResult` means the test is enabled ⑤.

The context is configured through the resources/context.properties configuration file:

```
context=regular
```

For the current business logic, the "regular" value means the tests will be executed in the current context.

The last thing to do is annotate the existing PassengerTest with the new extension:

```
@ExtendWith({ExecutionContextExtension.class})
public class PassengerTest {
[...]
```

Because the test executes in the current "regular" context configuration, the test will run just as it previously did. During peak periods, the context is set up differently (context=peak). If we try to execute the test during the peak period, we get the result shown in figure 14.1; the test is disabled, and it gives a reason.



Figure 14.1 The result of PassengerTest during the peak period, as extended with ExecutionContextExtension

We can instruct the Java Virtual Machine (JVM) to bypass the effects of conditional execution. To deactivate conditional execution, we set the `junit.jupiter.conditions.deactivate` configuration key to a pattern that matches the condition. From the Run > Edit Configurations menu, we can set `junit.jupiter.conditions.deactivate=*`, for example, and the result will be the deactivation of all conditions (figure 14.2). The result of the execution will not be influenced by any of the conditions, so all tests will run.

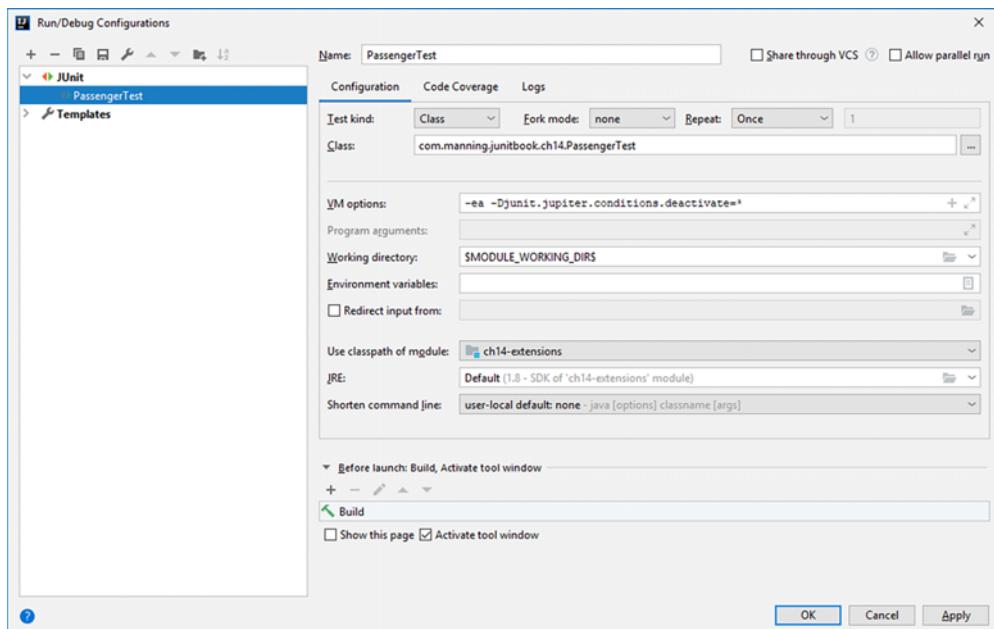


Figure 14.2 Deactivating conditional execution by setting the `junit.jupiter.conditions.deactivate` configuration key

14.3 Writing JUnit 5 tests using the available extension points

Harry is responsible for implementing and testing the passenger business logic, including persisting passengers to a database. This section follows his activity in implementing these tasks with the help of JUnit 5 extensions.

14.3.1 Persisting passengers to a database

Harry's next task is saving the passengers in a test database. Before the whole test suite is executed, the database must be reinitialized, and a connection to it must be open. At the end of the suite's execution, the connection to the database must be closed. Before executing a test, we have to set up the database in a known state so we can be sure that its content will be tested correctly. Harry decides to use the H2 database, JDBC, and JUnit 5 extensions.

H2 is a relational database management system developed in Java that permits the creation of an in-memory database. It can also be embedded in Java applications when required for testing purposes. JDBC is a Java API that defines how a client accesses a database; it's part of the Java Standard Edition platform.

To carry out this task, the first thing Harry needs to do is add the H2 dependency to the `pom.xml` file, as shown in the following listing.

Listing 14.4 H2 dependency added to the pom.xml file

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>1.4.199</version>
</dependency>
```

To manage the connection to the database, Harry implements the `ConnectionManager` class.

Listing 14.5 ConnectionManager class

```
public class ConnectionManager {
    private static Connection connection;

    public static Connection getConnection() {
        return connection;
    }

    public static Connection openConnection() {
        try {
            Class.forName("org.h2.Driver"); // this is driver for H2
            connection = DriverManager.
                getConnection("jdbc:h2:~/passenger",
                            "sa", // login
                            ""); // password
        }
        return connection;
    } catch(ClassNotFoundException | SQLException e) {
        throw new RuntimeException(e);
    }
}

public static void closeConnection() {
    if (null != connection) {
        try {
            connection.close();
        } catch(SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

In this listing:

- Harry declares a `java.sql.Connection` field and a getter to return it ①.
- The `openConnection` method loads the `org.h2.Driver` class, the driver for H2 ②, and creates a connection to the database with the URL `jdbc:h2:~/passenger` and the default credentials "sa" and " " ③. Normally, you do not want to store a user ID and password in plaintext in a configuration file, but it is OK

here because the database connection is only used for an in-memory database that is wiped after the tests are run.

- The `closeConnection` method tries to close the previously opened connection ④.

To manage the database tables, Harry implements the `TablesManager` class.

Listing 14.6 TablesManager class

```
public class TablesManager {

    public static void createTable(Connection connection) {
        String sql =
            "CREATE TABLE IF NOT EXISTS PASSENGERS (ID VARCHAR(50), " +
            "NAME VARCHAR(50));";

        executeStatement(connection, sql);
    }

    public static void dropTable(Connection connection) {
        String sql = "DROP TABLE IF EXISTS PASSENGERS;";

        executeStatement(connection, sql);
    }

    private static void executeStatement(Connection connection,
                                         String sql)
    {
        try(PreparedStatement statement =
            connection.prepareStatement(sql))
        {
            statement.executeUpdate();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```



In this listing:

- The `createTable` method creates the `PASSENGERS` table in the database. The table contains an `ID` and a `NAME`, both `VARCHAR(50)` ①.
- The `dropTable` method drops the `PASSENGERS` table from the database ②.
- The utility `executeStatement` method executes any SQL command against the database ③.

To manage the execution of the queries against the database, Harry implements the `PassengerDao` interface (listing 14.7) and the `PassengerDaoImpl` class (listing 14.8). A data access object (DAO) provides an interface to a database and maps the application calls to specific database operations without exposing the details of the persistence layer.

Listing 14.7 PassengerDao interface

```
public interface PassengerDao {
    public void insert(Passenger passenger); ①
    public void update(String id, String name);
    public void delete(Passenger passenger);
    public Passenger getById(String id); ④
}
```

In this listing, the `insert` ①, `update` ②, `delete` ③, and `getById` ④ methods declare the operations to be implemented against the database.

Listing 14.8 PassengerDaoImpl class

```
public class PassengerDaoImpl implements PassengerDao {

    private Connection connection; ①

    public PassengerDaoImpl(Connection connection) {
        this.connection = connection;
    }

    @Override
    public void insert(Passenger passenger) { ②
        String sql = "INSERT INTO PASSENGERS (ID, NAME) VALUES (?, ?);"

        try (PreparedStatement statement = connection.prepareStatement(sql)) { ③
            statement.setString(1, passenger.getIdentifier());
            statement.setString(2, passenger.getName());
            statement.executeUpdate();
        } catch (SQLException e) { ④
            throw new RuntimeException(e);
        }
    }

    @Override
    public void update(String id, String name) { ⑤
        String sql = "UPDATE PASSENGERS SET NAME = ? WHERE ID = ?;" ⑥

        try (PreparedStatement statement = connection.prepareStatement(sql)) { ⑦
            statement.setString(1, name);
            statement.setString(2, id);
            statement.executeUpdate();
        } catch (SQLException e) { ⑧
            throw new RuntimeException(e);
        }
    }

    @Override
    public void delete(Passenger passenger) { ⑨
        String sql = "DELETE FROM PASSENGERS WHERE ID = ?;" ⑩

        try (PreparedStatement statement = connection.prepareStatement(sql)) {
            statement.setString(1, passenger.getIdentifier());
        }
    }
}
```

```

        statement.executeUpdate();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

@Override
public Passenger getById(String id) {
    String sql = "SELECT * FROM PASSENGERS WHERE ID = ?";
    Passenger passenger = null;

    try (PreparedStatement statement = connection.prepareStatement(sql)) {
        statement.setString(1, id);
        ResultSet resultSet = statement.executeQuery();
        if (resultSet.next()) {
            passenger = new Passenger(resultSet.getString(1),
                resultSet.getString(2));
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    return passenger;
}
}

```

In this listing:

- The Connection field is kept inside the class and provided as an argument of the constructor ①.
- The insert method declares the SQL query to be executed ②, sets the identifier and name of the passenger as parameters of the query ③, and executes the query ④.
- The update method declares the SQL query to be executed ⑤, sets the identifier and name of the passenger as parameters of the query ⑥, and executes the query ⑦.
- The delete method declares the SQL query to be executed ⑧, sets the identifier of the passenger as a parameter of the query ⑨, and executes the query ⑩.
- The getById method declares the SQL query to be executed ⑪, sets the identifier of the passenger as a parameter of the query ⑫, executes the query ⑬, and creates a new Passenger object with the parameters returned from the database ⑭. Then the new object is returned by the method ⑮.

Now Harry needs to implement the JUnit 5 extension to do the following:

- 1 Before the execution of the entire test suite, reinitialize the database and open a connection to it.
- 2 At the end of the execution of the suite, close the connection to the database.
- 3 Before executing a test, make sure the database is in a known state so the developer can be sure its content is tested correctly.

Looking at the requirements, which ask for actions based on the life cycle of the test suite, it is natural for Harry to choose to implement life-cycle callbacks. To implement the extensions concerning the test life cycle, Harry must add the following interfaces:

- `BeforeEachCallback` and `AfterEachCallback`—Executed before and after the execution of each of the test methods, respectively
- `BeforeAllCallback` and `AfterAllCallback`—Executed before and after the execution of all test methods, respectively

Harry implements the `DatabaseOperationsExtension` class shown in the next listing.

Listing 14.9 The DatabaseOperationsExtension class

```
public class DatabaseOperationsExtension implements
    BeforeAllCallback, AfterAllCallback, BeforeEachCallback,
    AfterEachCallback {

    private Connection connection; ②
    private Savepoint savepoint;

    @Override
    public void beforeAll(ExtensionContext context) {
        connection = ConnectionManager.openConnection();
        TablesManager.dropTable(connection);
        TablesManager.createTable(connection);
    } ③

    @Override
    public void afterAll(ExtensionContext context) {
        ConnectionManager.closeConnection(); ④
    }

    @Override
    public void beforeEach(ExtensionContext context)
        throws SQLException {
        connection.setAutoCommit(false);
        savepoint = connection.setSavepoint("savepoint");
    } ⑤

    @Override
    public void afterEach(ExtensionContext context)
        throws SQLException {
        connection.rollback(savepoint); ⑥
    } ⑦
}
```

In this listing:

- The `DatabaseOperationsExtension` class implements four life-cycle interfaces: `BeforeAllCallback`, `AfterAllCallback`, `BeforeEachCallback`, and `AfterEachCallback` ①.

- The class declares a `Connection` field to connect to the database and a `Savepoint` field to track the state of the database before the execution of a test and to restore it after the test ②.
- The `beforeAll` method, inherited from the `BeforeAllCallback` interface, is executed before the whole suite. It opens a connection to the database, drops the existing table, and re-creates it ③.
- The `afterAll` method, inherited from the `AfterAllCallback` interface, is executed after the whole suite. It closes the connection to the database ④.
- The `beforeEach` method, inherited from the `BeforeEachCallback` interface, is executed before each test. It disables autocommit mode, so the database changes resulting from the execution of the test should not be committed ⑤. Then the method saves the state of the database before the execution of the test so that, after the test, the developer can roll back to it ⑥.
- The `afterEach` method, inherited from the `AfterEachCallback` interface, is executed after each test. It rolls back to the state of the database that was saved before the execution of the test ⑦.

Harry updates the `PassengerTest` class and introduces tests that verify the newly introduced database functionalities.

Listing 14.10 Updated `PassengerTest` class

```

@ExtendWith({ExecutionContextExtension.class,
            DatabaseOperationsExtension.class })
public class PassengerTest {

    private PassengerDao passengerDao;

    public PassengerTest(PassengerDao passengerDao) {
        this.passengerDao = passengerDao;
    }

    @Test
    void testPassenger(){
        Passenger passenger = new Passenger("123-456-789", "John Smith");
        assertEquals("Passenger John Smith with identifier: 123-456-789",
                    passenger.toString());
    }

    @Test
    void testInsertPassenger() {
        Passenger passenger = new Passenger("123-456-789",
                                             "John Smith");
        passengerDao.insert(passenger);
        assertEquals("John Smith",
                    passengerDao.getById("123-456-789").getName());
    }

    @Test
    void testUpdatePassenger() {

```

```

Passenger passenger = new Passenger("123-456-789",
    "John Smith");
    6
    passengerDao.insert(passenger);
    7
    passengerDao.update("123-456-789", "Michael Smith");
    8
    assertEquals("Michael Smith",
        passengerDao.getById("123-456-789").getName());
    9

}

@Test
void testDeletePassenger() {
    Passenger passenger = new Passenger("123-456-789",
        "John Smith");
    10
    passengerDao.insert(passenger);
    passengerDao.delete(passenger);
    assertNull(passengerDao.getById("123-456-789"));
    11
    12
}
    13
}

```

In this listing:

- The test is extended by `DatabaseOperationsExtension` ①.
- The constructor of the `PassengerTest` class receives a `PassengerDao` as argument ②. This `PassengerDao` will be needed to execute the tests against the database.
- The `testInsertPassenger` method creates a passenger ③, inserts them in the database with `passengerDao` ④, and checks whether they are found in the database ⑤.
- The `testUpdatePassenger` method creates a passenger ⑥, inserts them in the database with `passengerDao` ⑦, updates the information about them ⑧, and checks whether the update information is found in the database ⑨.
- The `testDeletePassenger` method creates a passenger ⑩, inserts them in the database with `passengerDao` ⑪, deletes them ⑫, and checks whether the passenger is found in the database ⑬.

If we run the tests at this time, we get the results shown in figure 14.3.

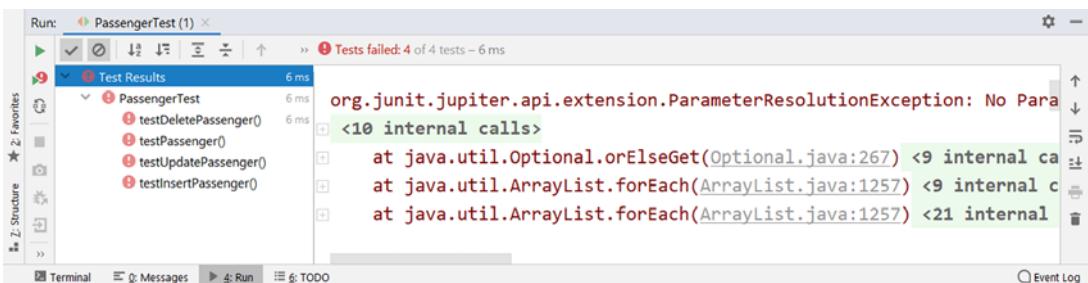


Figure 14.3 The result of the updated `PassengerTest` after it is extended with `DatabaseOperationsExtension`

The tests fail with this message:

```
org.junit.jupiter.api.extension.ParameterResolutionException:
No ParameterResolver registered for parameter
    [com.manning.junitbook.ch14.jdbc.PassengerDao arg0]
in constructor
```

This message appears because the constructor of the `PassengerTest` class is receiving a parameter of type `PassengerDao`, but this parameter is not provided by any `ParameterResolver`. To complete the task, Harry has to implement a `ParameterResolver` interface.

Listing 14.11 `DataAccessObjectParameterResolver` class

```
public class DataAccessObjectParameterResolver implements
    ParameterResolver{  

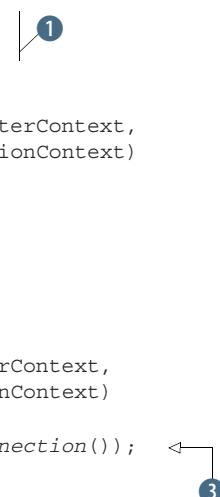
    @Override
    public boolean supportsParameter(ParameterContext parameterContext,
                                    ExtensionContext extensionContext)
        throws ParameterResolutionException {
    return parameterContext.getParameter()
        .getType()
        .equals(PassengerDao.class);
}  

    @Override
    public Object resolveParameter(ParameterContext parameterContext,
                                ExtensionContext extensionContext)
        throws ParameterResolutionException {
    return new PassengerDaoImpl(ConnectionManager.getConnection());
}  

}  

}
```



In this listing:

- The class implements the `ParameterResolver` interface ①.
- The `supportsParameter` method returns `true` if the parameter is of type `PassengerDao`. This is the missing parameter of the `PassengerTest` class constructor ②, so the parameter resolver supports only a `PassengerDao` object.
- The `resolveParameter` method returns a newly initialized `PassengerDaoImpl` that receives as a constructor parameter the connection provided by the `ConnectionManager` ③. This parameter will be injected into the test constructor at runtime.

Additionally, Harry extends the `PassengerTest` class with `DatabaseAccessObjectParameterResolver`. The first lines of the `PassengerTest` class are shown next.

Listing 14.12 Extending PassengerTest

```
@ExtendWith({ExecutionContextExtension.class,
            DatabaseOperationsExtension.class,
            DataAccessObjectParameterResolver.class})
public class PassengerTest {
    [...]
```

The result of executing PassengerTest is shown in figure 14.4. All tests are green; the interaction with the database works properly. Harry has successfully implemented the additional behavior of the tests executed against a database with the help of the life-cycle callback and parameter-resolution extensions.

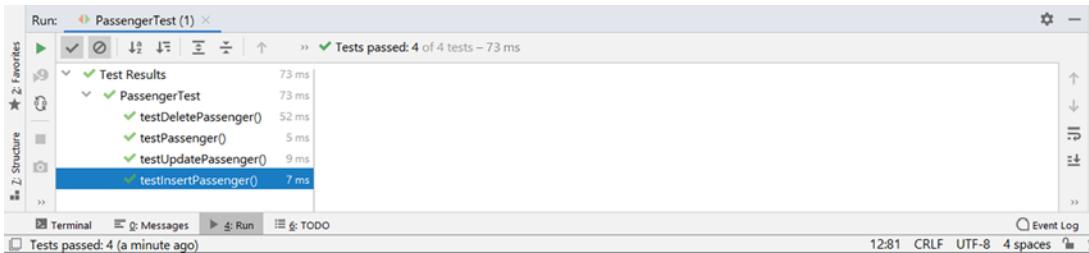


Figure 14.4 The result of the updated PassengerTest after it is extended with DatabaseOperationsExtension and DatabaseAccessObjectParameterResolver

14.3.2 Checking the uniqueness of passengers

Next, Harry must prevent a passenger from being inserted into the database more than once. To implement this requirement, he decides to create his own custom exception, together with an exception-handling extension. He wants to introduce this custom exception because it is more expressive than the general SQLException. First, Harry creates the custom exception.

Listing 14.13 PassengerExistsException class

```
public class PassengerExistsException extends Exception {
    private Passenger passenger;
    1
    public PassengerExistsException(Passenger passenger, String message) {
        super(message);
        this.passenger = passenger;
    }
    2
    3}
```

In this listing:

- Harry keeps the existing passenger as a field of the exception ①.

- He invokes the constructor of the superclass, retaining the `message` parameter **2**, and then sets the passenger **3**.

Next, Harry changes the `PassengerDao` interface and the `PassengerDaoImpl` class so that the `insert` method throws `PassengerExistsException`.

Listing 14.14 Modified insert method from PassengerDaoImpl

```
public void insert(Passenger passenger) throws PassengerExistsException {
    String sql = "INSERT INTO PASSENGERS (ID, NAME) VALUES (?, ?)";

    if (null != getById(passenger.getIdentifier()) ) {
        throw new PassengerExistsException
            (passenger, passenger.toString());
    }

    try (PreparedStatement statement = connection.prepareStatement(sql)) {
        statement.setString(1, passenger.getIdentifier());
        statement.setString(2, passenger.getName());
        statement.executeUpdate();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

The `PassengerDaoImpl` `insert` method checks whether the passenger exists. If so, it throws `PassengerExistsException` **1**.

Harry wants to introduce a test that tries to insert the same passenger twice. Because he expects this test to throw an exception, he implements an exception-handling extension to log it.

Listing 14.15 LogPassengerExistsExceptionExtension class

```
public class LogPassengerExistsExceptionExtension implements
    TestExecutionExceptionHandler {
    private Logger logger = Logger.getLogger(this.getClass().getName());
}

@Override
public void handleTestExecutionException(ExtensionContext context,
    Throwable throwable) throws Throwable {
    if (throwable instanceof PassengerExistsException) {
        logger.severe("Passenger exists:" + throwable.getMessage());
        return;
    }
    throw throwable;
}
```

In this listing:

- The class implements the `TestExecutionExceptionHandler` interface **1**.

- Harry declares a logger of the class ② and overrides the handleTestExecutionException method inherited from the TestExecutionExceptionHandler interface ③.
- He checks whether the thrown exception is an instance of PassengerExistsException; if so, he simply logs it and returns from the method ④. Otherwise, he rethrows the exception so it can be handled elsewhere ⑤.

The updated PassengerTest class is shown in the following listing.

Listing 14.16 Updated PassengerTest class

```
@ExtendWith({ExecutionContextExtension.class,
            DatabaseOperationsExtension.class,
            DatabaseAccessObjectParameterResolver.class,
            LogPassengerExistsExceptionExtension.class})
public class PassengerTest {
    [...]
```

The result of PassengerTest is shown in figure 14.5. All tests are green; the PassengerExistsException is caught and logged by the new extension. Table 14.1 summarizes the extension points and interfaces provided by JUnit 5.

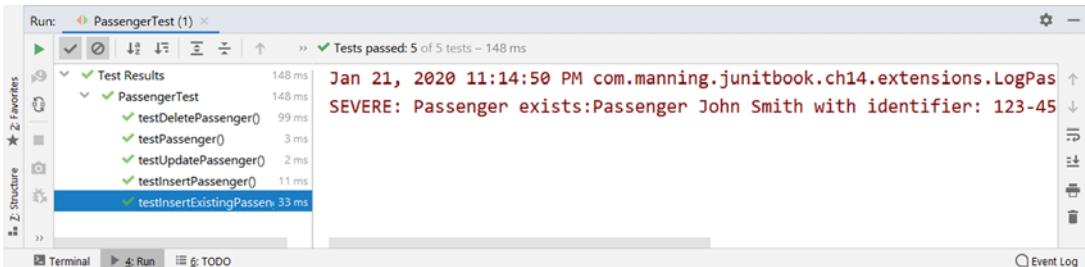


Figure 14.5 The result of the updated PassengerTest after it is extended with LogPassengerExistsExceptionExtension

Table 14.1 Extension points and corresponding interfaces

Extension point	Interfaces to implement
Conditional test execution	ExecutionCondition
Life-cycle callbacks	BeforeAllCallback, AfterAllCallback, BeforeEachCallback, AfterEachCallback
Parameter resolution	ParameterResolver
Exception handling	TestExecutionExceptionHandler
Test instance postprocessing	TestInstancePostProcessor

Chapter 15 is dedicated to presentation-layer testing and finding bugs in an application's graphical user interface (GUI).

Summary

This chapter has covered the following:

- Introducing the JUnit 5 extension model and the available extension points: conditional test execution, life-cycle callbacks, parameter resolution, exception handling, and test instance postprocessing
- Demonstrating the development tasks that require the creation of JUnit 5 extensions: a context extension for conditional test execution; a passenger database setup extension for life-cycle callbacks; a parameter resolver for parameter resolution; and a logging exception extension for exception handling
- Using extensions to implement JUnit 5 tests required by the development scenario

15

Presentation-layer testing

This chapter covers

- Developing HtmlUnit tests
- Developing Selenium tests
- Comparing HtmlUnit and Selenium

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

—Edsger Dijkstra

Simply stated, presentation-layer testing involves finding bugs in the graphical user interface (GUI) of an application. Finding errors here is as important as finding errors in other application tiers. A bad user experience can lose a customer or discourage a web surfer from visiting your site again. Furthermore, bugs in the user interface can cause other parts of the application to malfunction.

Due to its nature and user interaction, GUI testing presents unique challenges and requires its own set of tools and techniques. This chapter will cover the testing of web application user interfaces. We address what can be objectively—that is,

programmatically—asserted about the GUI (by writing explicit Java code). Subjective elements like fonts, colors, and layout are outside the scope of this discussion.

Testing interaction with websites can be challenging from the point of view of stability. If the website content changes over time, or if you encounter problems with your internet connection, this kind of test may occasionally or permanently fail. The tests presented in this chapter have been designed to have high long-term stability by accessing known websites with less probability of changing in the near future.

We can test the following:

- The content of web pages to any level of detail (including spelling)
- The application structure or navigation (following links to their expected destination, for example)
- The ability to verify user stories with acceptance tests¹

A *user story* is an informal, natural-language description of one or more features of a software system. An *acceptance test* is a test conducted to determine whether the requirements of a specification are met. We can also verify that the site works with the required browsers and operating systems.

15.1 Choosing a testing framework

We will look at two free, open source tools to implement the presentation-layer tests in JUnit 5: HtmlUnit and Selenium. HtmlUnit is a 100% Java headless browser framework that runs in the same virtual machine as our JUnit 5 tests. A *headless* browser is a browser without a GUI. We use HtmlUnit when our application is independent of OS features and browser-specific implementations of JavaScript, DOM, CSS, and so on.

Selenium drives various web browsers programmatically and checks the results from executing JUnit 5 tests. We use Selenium when we require validation of specific browsers and OSs, especially if the application takes advantage of or depends on a browser-specific implementation of JavaScript, DOM, CSS, and so on. Let's start with HtmlUnit.

15.2 Introducing HtmlUnit

HtmlUnit (<http://htmlunit.sourceforge.net>) is an open source Java headless browser framework. It allows tests to imitate the user of a browser-based web application programmatically. HtmlUnit JUnit 5 tests do not display a user interface. In the remainder of this section, when we talk about “testing with a web browser,” it is with the understanding that we are really testing by emulating a specific web browser.

15.2.1 A live example

We'll first introduce the `ManagedWebClient` base class that we are going to use as the base class for HtmlUnit tests using JUnit 5 annotations.

¹ To learn about extreme programming acceptance tests, see www.extremeprogramming.org/rules/function-altests.html.

Listing 15.1 ManagedWebClient base class

```
[...]
import com.gargoylesoftware.htmlunit.WebClient;
[...]

public abstract class ManagedWebClient {
    protected WebClient webClient; ①

    @BeforeEach
    public void setUp() {
        webClient = new WebClient();
    } ②

    @AfterEach
    public void tearDown() {
        webClient.close();
    } ③
}
```

In this listing:

- We define a protected `WebClient` field to be inherited by the subclasses ①. The `com.gargoylesoftware.htmlunit.WebClient` class is the main starting point of an HtmlUnit test.
- Before each test, we initialize a new `WebClient` object ②. It simulates a web browser and will be used to execute all the tests.
- After each test, we make sure that the simulated browser is closed when the test is executed with a `WebClient` instance ③.

Let's jump in with the example shown in listing 15.2. If you can connect to the internet, you can test. We will go to the HtmlUnit website and test the home page. We will also navigate to the Javadoc and make sure a class appears at the top of the list of documented classes.

Listing 15.2 Our first HtmlUnit example

```
[...]
public class HtmlUnitPageTest extends ManagedWebClient {

    @Test
    public void homepage() throws IOException {
        HtmlPage page = webClient.
            getPage("http://htmlunit.sourceforge.net");
        assertEquals("HtmlUnit - Welcome to HtmlUnit",
                    page.getTitleText()); ①

        String pageAsXml = page.asXml();
        assertTrue(pageAsXml.
                    contains("<div class=\"container-fluid\">")); ②
    }
}
```

```

String pageAsText = page.asText();
assertTrue(pageAsText.contains(
    "Support for the HTTP and HTTPS protocols"));
}

@Test
public void testClassNav() throws IOException {
    HtmlPage mainPage = webClient.getPage(
        "http://htmlunit.sourceforge.net/apidocs/index.html");
    HtmlPage packagePage = (HtmlPage)
        mainPage.getFrameByName("packageFrame").getEnclosedPage();
    HtmlListItem htmlListItem = (HtmlListItem)
        packagePage.getElementsByTagName("li").item(0);
    assertEquals("AboutURLConnection", htmlListItem.getTextContent());
}
}

```

In this listing:

- In the first test, we access the HtmlUnit website home page and check that the title of the page is as expected ①.
- We get the home page of the HtmlUnit website as XML and check that it contains a particular tag ②.
- We get the home page of the HtmlUnit website as text and check that it contains a particular string ③.
- In the second test, we access a URL from the HtmlUnit website ④.
- We get the page from inside the packageFrame ⑤.
- Inside the page obtained at ⑤, we look for the first element tagged as "li" ⑥.
- We check that the text of the element obtained at ⑥ is "AboutURLConnection" ⑦.

This example covers the basics: getting a web page, navigating the HTML object model, and asserting results. We'll look at more examples in the next section.

15.3 Writing HtmlUnit tests

When we write an HtmlUnit test, we write code that simulates the action of a user sitting in front of a web browser: we get a web page, enter data, read the text, and click buttons and links. Instead of manually manipulating the browser, we programmatically control an emulated browser. At each step, we can query the HTML object model and assert that values are what we expect. The framework will throw exceptions if it encounters a problem, which allows our test cases to avoid checking for these errors, thus reducing clutter.

15.3.1 HTML assertions

We know that JUnit 5 provides a class called `Assertions` to allow tests to fail when they detect an error condition. Assertions are the bread and butter of any unit test. HtmlUnit can work with JUnit 5, but it also provides a class in the same spirit called `WebAssert`, which contains standard assertions for HTML like `assertTitleEquals`, `assertTextPresent`, and `notNull`.

15.3.2 Testing for a specific web browser

As of version 2.36, HtmlUnit supports the browsers listed in table 15.1.

Table 15.1 HtmlUnit-supported browsers

Web browser and version	HtmlUnit <code>BrowserVersion</code> constant
Internet Explorer 11	<code>BrowserVersion.INTERNET_EXPLORER</code>
Firefox 5.2 (deprecated)	<code>BrowserVersion.FIREFOX_52</code>
Firefox 6.0	<code>BrowserVersion.FIREFOX_60</code>
Latest Chrome	<code>BrowserVersion.CHROME</code>
The best-supported browser at the moment	<code>BrowserVersion.BEST_SUPPORTED</code>

By default, `WebClient` emulates `BrowserVersion.BEST_SUPPORTED`, which, at the time of writing this chapter, is Google Chrome (but this may change in the future, depending on the evolution of each browser). To specify which browser to emulate, we provide the `WebClient` constructor with a `BrowserVersion`. For example, for Firefox 6.0, we use

```
WebClient webClient = new WebClient(BrowserVersion.FIREFOX_60);
```

15.3.3 Testing more than one web browser

Our example company, Tested Data Systems, has many customers. Naturally, each customer uses its preferred browser to access the pages of applications developed for it. Consequently, the engineers at Tested Data Systems would like to test their applications with more than one browser version. John is in charge of writing tests for this purpose. He will define a test matrix to include all HtmlUnit-supported web browsers.

Listing 15.3 uses JUnit 5 parameterized tests to drive the same test with all browsers in the test matrix. The JUnit 5 parameterized tests will use Firefox 6.0, Internet Explorer 11, the latest Chrome (79 at the time of writing), and the best-supported browser (again, Chrome 79 at the time of writing, but this may change in the future).

Listing 15.3 Testing for all HtmlUnit-supported browsers

```
[...]
public class JavadocPageAllBrowserTest {

    private static Collection<BrowserVersion[]> getBrowserVersions() {
        return Arrays.asList(new BrowserVersion[][] {
            { BrowserVersion.FIREFOX_60 },
            { BrowserVersion.INTERNET_EXPLORER },
            { BrowserVersion.CHROME },
            { BrowserVersion.BEST_SUPPORTED } });
    }

    @ParameterizedTest
    @MethodSource("getBrowserVersions")
}
```

```

public void testClassNav(BrowserVersion browserVersion)
                        throws IOException
{
    WebClient webClient = new WebClient(browserVersion);

    5   HtmlPage mainPage = (HtmlPage) webClient
        .getPage(
            "http://htmlunit.sourceforge.net/apidocs/index.html");
    6   WebAssert.notNull("Missing main page", mainPage);

    7   HtmlPage packagePage = (HtmlPage) mainPage
        .getFrameByName("packageFrame").getEnclosedPage();
    8   WebAssert.notNull("Missing package page", packagePage);

    9   HtmlListItem htmlListItem = (HtmlListItem) packagePage
        .getElementsByTagName("li").item(0);
    10  assertEquals("AboutURLConnection",
        htmlListItem.getTextContent());
}
}

```

In this listing:

- John creates a `private static` method that will provide the collection of all web browsers supported by HtmlUnit ①.
- He creates the `testClassNav` method that will receive as a parameter the `BrowserVersion` ④. This is a parameterized test, as indicated by the `@ParameterizedTest` annotation ②. The `BrowserVersion` parameters are injected by the `getBrowserVersion` method, as indicated by the `@MethodSource` annotation ③.
- He creates a `WebClient` receiving as a constructor argument the injected `BrowserVersion` ⑤, accesses a page on the internet ⑥, and checks that the page exists ⑦.
- Inside the page that John previously accessed, he looks for a frame with the name `packageFrame` ⑧ and checks that it exists ⑨.
- Inside the `packageFrame`, he looks for a list of all `HtmlUnit` classes and checks whether the first class on the list is named `AboutURLConnection` ⑩.

15.3.4 Creating standalone tests

You may not always want to use the actual URLs of pages as input for tests because external pages may be subject to change without notice. A minor change in a website can make the tests break. In this section, we will show how to embed and run HTML in the unit test code itself.

The framework allows us to plug a mock (see chapter 8) HTTP connection into a web client. In listing 15.4, we create a JUnit 5 test that sets up a mock connection (an instance of the class `com.gargoylesoftware.htmlunit.MockWebConnection`) with a default HTML String response. The JUnit 5 test can then get this default page by using any URL value. We will test the title of the HTML response obtained by a web

client with a mocked connection that avoids using an actual URL that may change without notice.

Listing 15.4 Configuring a standalone test

```
[...]
public class InLineHtmlFixtureTest extends ManagedWebClient {

    @Test
    public void testInLineHtmlFixture() throws IOException {
        final String expectedTitle = "Hello 1!";
        String html = "<html><head><title>" +
                      expectedTitle +
                      "</title></head></html>";
        MockWebConnection connection = new MockWebConnection();
        connection.setDefaultResponse(html);
        webClient.setWebConnection(connection);
        HtmlPage page = webClient.getPage("http://page");
        WebAssert.assertTitleEquals(page, expectedTitle);
    }
}
```

In this listing:

- We create the expected HTML page title ① and the expected HTML response ②.
- We create a MockWebConnection ③ and set the HTML response as the default response for the mock connection ④. We then set the web client connection to our mock connection ⑤.
- We get the test page ⑥. Any URL is fine here since we set up our HTML response as the default response.
- We check that the page title matches our HTML response ⑦.

To configure a JUnit 5 test with multiple pages, we call one of the `MockWebConnection` `setResponse` methods for each page. The code in the next listing sets up three web pages in a mock connection. Again, we will test the titles of the HTML responses obtained by a web client with a mocked connection, thus avoiding using an actual URL that may change without notice.

Listing 15.5 Configuring a test with multiple page fixtures

```
@Test
public void testInLineHtmlFixtures() throws IOException {
    final URL page1Url = new URL("http://Page1/");
    final URL page2Url = new URL("http://Page2/");
    final URL page3Url = new URL("http://Page3/");

    MockWebConnection connection = new MockWebConnection();
    connection.setResponse(page1Url,
```

```

        "<html><head><title>Hello 1!</title></head></html>") ;
connection.setResponse(page2Url,
        "<html><head><title>Hello 2!</title></head></html>") ;
connection.setResponse(page3Url,
        "<html><head><title>Hello 3!</title></head></html>") ;
webClient.setWebConnection(connection);

④   HtmlPage page1 = webClient.getPage(page1Url);
WebAssert.assertTitleEquals(page1, "Hello 1!");

HtmlPage page2 = webClient.getPage(page2Url);
WebAssert.assertTitleEquals(page1, "Hello 2!");

HtmlPage page3 = webClient.getPage(page3Url);
WebAssert.assertTitleEquals(page1, "Hello 3!");
}

```

In this listing:

- We create three pages ① and a mock connection ② and set three responses for each of the URL pages ③.
- We set the web client connection to our mock connection ④.
- We test getting each page and verifying each page title ⑤.

WARNING Do not forget the trailing / in the URL! `http://Page1/` will work, but `http://Page1` will not be found in the mock connection. The result will be an `IllegalStateException` with the message “No response specified that can handle URL.”

15.3.5 Testing forms

HTML form support is built into the `HtmlPage` API, where form elements can be accessed with `getForms` (returns `List<HtmlForm>`) to get all form elements and `getFormByName` to get the first `HtmlForm` with a given name. We can call one of the `HtmlForm` `getInput` methods to get HTML input elements and then simulate user input with `setValueAttribute`.

The following example focuses on the `HtmlUnit` mechanics of driving a form. First, we create a simple page to display a form with an input field and a Submit button. We include form validation via JavaScript alerts.

Listing 15.6 Example form page

```

<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<script>
function validate_form(form) {
    if (form.in_text.value=="") {
        alert("Please enter a value.");
        form.in_text.focus();
        return false;
    }
}

```

```

}
</script>
<title>Form</title></head>
<body>
<form name="validated_form" action="submit.html"
      onsubmit="return validate_form(this);" method="post">
  Value:
  <input type="text" name="in_text" id="in_text" size="30"/>
  <input type="submit" value="Submit" id="submit"/>
</form>
</body>
</html>

```

This form looks like figure 15.1 when we click the OK button without input.

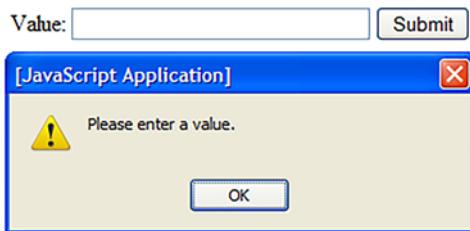


Figure 15.1 Sample form with one input and one Submit button, and the alert triggered when we click the button without input

The next listing tests normal user interaction with the form.

Listing 15.7 Testing a form

```

[...]
public class FormTest extends ManagedWebClient {

  @Test
  public void testForm() throws IOException {
    HtmlPage page =
      webClient.getPage("file:src/main/webapp/formtest.html");
    HtmlForm form = page.getFormByName("validated_form");
    HtmlTextInput input = form.getInputByName("in_text");
    input.setValueAttribute("typing...");
    HtmlSubmitInput submitButton = form.getInputByName("submit");
    HtmlPage resultPage = submitButton.click();
    WebAssert.assertTitleEquals(resultPage, "Result");
  }
}

```

In this listing:

- We use the web client inherited from the parent ManagedWebClient class to get the page containing the form ①, and then we get the form ②.
- We get the input text field from the form ③, emulate the user typing in a value ④, and then get and click the Submit button ⑤.

- We get back a page from clicking the button, and we make sure it is the expected page ⑥.

If at any step the framework does not find an object, the API throws an exception, and the test automatically fails. This allows us to focus on the test and let the framework handle failing the test if the page or form is not as expected.

15.3.6 Testing JavaScript

HtmlUnit processes JavaScript automatically. Even when, for example, HTML is generated with `Document.write()`, we follow the usual pattern: call `getPage`, find an element, click it, and check the result.

We can toggle JavaScript support on and off in a web client by calling

```
webClient.getOptions().setJavaScriptEnabled(true);
```

or

```
webClient.getOptions().setJavaScriptEnabled(false);
```

HtmlUnit enables JavaScript support by default. We can also set how long a script is allowed to run before being terminated, by calling

```
webClient.setJavaScriptTimeout(timeout);
```

To deal with JavaScript alert and confirm calls, we can provide the framework with callback routines. We will explore these next.

TESTING JAVASCRIPT ALERTS

Our tests can check which JavaScript alerts have taken place. We will reuse our example for testing forms from listing 15.7; it includes JavaScript validation code to alert the user about empty input values.

The test in listing 15.8 loads our form page and checks calling the alert when the form detects an error condition. Our test installs an alert handler that gathers all alerts and checks the result after the page has been loaded. The class `CollectingAlertHandler` saves alert messages for later inspection.

Listing 15.8 Asserting expected alerts

```
[...]
public class FormTest extends ManagedWebClient {
[...]

@Test
public void testFormAlert() throws IOException {
    CollectingAlertHandler alertHandler =
        new CollectingAlertHandler();
    webClient.setAlertHandler(alertHandler);
    HtmlPage page = webClient.getPage(
        "file:src/main/webapp/formtest.html");
    HtmlForm form = page.getFormByName("validated_form");
    HtmlSubmitInput submitButton = form.getInputByName("submit");
    [...]
```

```

6   HtmlPage resultPage = submitButton.click();
7   WebAssert.assertTitleEquals(resultPage, page.getTitleText());
8   WebAssert.assertTextPresent(resultPage, page.asText());
9
10  List<String> collectedAlerts =
11    alertHandler.getCollectedAlerts();
12  List<String> expectedAlerts = Collections.singletonList(
13      "Please enter a value.");
14  assertEquals(expectedAlerts, collectedAlerts);
15}

```

In this listing:

- We create the alert handler ①, which we install in the web client inherited from the parent class ②.
- We get the form page ③, the form object ④, and the Submit button ⑤, and we click the button ⑥. This invokes the JavaScript, which calls the alert.
- Clicking the button returns a page object, which we use to check that the page has not changed by comparing the current and previous page titles ⑦. We also check that the page has not changed by comparing the current and previous page objects ⑧.
- We get the list of alert messages that were raised ⑨, create a list of expected alert messages ⑩, and compare the expected and actual lists ⑪.

Next, we rewrite the original form test from listing 15.7 to make sure the form's normal operation doesn't raise alerts.

Listing 15.9 Asserting no alerts under normal operation

```

[...]
public class FormTest extends ManagedWebClient {
[...]

    @Test
    public void testFormNoAlert() throws IOException {
        CollectingAlertHandler alertHandler =
            new CollectingAlertHandler();
        webClient.setAlertHandler(alertHandler);
        HtmlPage page = webClient.getPage(
            "file:src/main/webapp/formtest.html");
        HtmlForm form = page.getFormByName("validated_form");
        HtmlTextInput input = form.getInputByName("in_text");
        input.setValueAttribute("typing...");
        HtmlSubmitInput submitButton = form.getInputByName("submit");
        HtmlPage resultPage = submitButton.click();
        WebAssert.assertTitleEquals(resultPage, "Result");
        assertTrue(alertHandler.getCollectedAlerts().isEmpty(),
            "No alerts expected");
    }
}

```

In this listing:

- We install a `CollectingAlertHandler` in the web client ①.
- We simulate a user entering a value ② and, at the end of the test, check that the alert handler list of messages is empty ③.

To customize the alert behavior, we need to implement our own `AlertHandler`. Setting `AlertHandler` as shown next will cause our test to fail when a script raises the first alert.

Listing 15.10 Custom alert handler

```
webClient.setAlertHandler((page, message) ->
    fail("JavaScript alert: " + message));
```

Now we apply the same principles to test JavaScript confirm calls by installing a confirm handler in the web client with `setConfirmHandler`.

Listing 15.11 Asserting the expected confirmation messages

```
[...]
public class WindowConfirmTest extends ManagedWebClient {

    @Test
    public void testWindowConfirm() throws FailingHttpStatusCodeException,
        IOException {
        String html = "<html><head><title>Hello</title></head>
                     <body onload='confirm(\"Confirm Message\")'>
                     </body></html>";
        URL testUrl = new URL("http://Page1/");
        MockWebConnection mockConnection = new MockWebConnection();
        final List<String> confirmMessages = new ArrayList<String>();
        1
        2
        3
        4
        webClient.setConfirmHandler((page, message) -> {
            confirmMessages.add(message);
            return true;
        });
        mockConnection.setResponse(testUrl, html);
        webClient.setWebConnection(mockConnection);
        5
        6
        7
        8
        9
        10
        HtmlPage firstPage = webClient.getPage(testUrl);
        WebAssert.assertTitleEquals(firstPage, "Hello");
        assertEquals(new String[] { "Confirm Message" },
                     confirmMessages.toArray());
    }
}
```

In this listing:

- We create the HTML page, including the confirm message ① and the test URL to be accessed ②.
- We create a mock connection ③ and initialize an empty confirm message list ④.

- We set up the `webClient` inherited from the superclass with a confirm handler defined as a lambda expression that will simply add the message to the list ⑤. We set the response of the connection when accessing the test URL ⑥ and then set the web client connection to our mock connection ⑦.
- We get the page ⑧ and then check its title ⑨ and the array of confirming messages ⑩.

Next, we will modify the previous code to introduce a function in the JavaScript of the emulated website and a handler to collect alerts.

Listing 15.12 Asserting expected confirmation messages from a JavaScript function

```
public class WindowConfirmTest extends ManagedWebClient {

    @Test
    public void testWindowConfirmAndAlert() throws
        FailingHttpStatusCodeException, IOException {
        String html = "<html><head><title>Hello</title>
                     <script>function go(){
                         alert(confirm('Confirm Message'))
                     }</script>\n" +
                     "</head><body onload='go()'></body></html>";
        URL testUrl = new URL("http://Page1/");
        MockWebConnection mockConnection = new MockWebConnection();
        final List<String> confirmMessages = new ArrayList<String>();
        webClient.setAlertHandler(new CollectingAlertHandler());
        webClient.setConfirmHandler((page, message) -> {
            confirmMessages.add(message);
            return true;
        });
        mockConnection.setResponse(testUrl, html);
        webClient.setWebConnection(mockConnection);

        HtmlPage firstPage = webClient.getPage(testUrl);
        WebAssert.assertTitleEquals(firstPage, "Hello");
        assertArrayEquals(new String[] { "Confirm Message" },
                         confirmMessages.toArray());
        assertArrayEquals(new String[] { "true" },
                         ((CollectingAlertHandler)
                          webClient.getAlertHandler())
                          .getCollectedAlerts().toArray());
    }
}
```

The code is annotated with four numbered callouts:

- ① A callout pointing to the `confirm('Confirm Message')` line in the JavaScript code.
- ② A callout pointing to the `CollectingAlertHandler` implementation in the `setAlertHandler` and `setConfirmHandler` blocks.
- ③ A callout pointing to the `assertArrayEquals` call in the test code, comparing the `confirmMessages` list with the expected array.
- ④ A callout pointing to the `CollectingAlertHandler` implementation in the `getCollectedAlerts` block.

In this listing:

- We create the HTML page, including a confirm message provided by a JavaScript function ①.
- We set an alert handler for the `webClient` inherited from the superclass ②. A `CollectingAlertHandler` is a simple alert handler that keeps track of alerts in a list.

- In addition to the list of confirmation messages ③, we also check the collected alerts ④.

TIP Note that when you run some of the HtmlUnit tests in these sections, you may get a series of warnings. The problem is not in the test, but in the CSS of the web page that is accessed.

15.4 Introducing Selenium

Selenium (<https://seleniumhq.org>) is a free, open source tool suite used to test web applications. Selenium's strength lies in its ability to run tests against a real browser on a specific OS; this is unlike HtmlUnit, which emulates the browser in the same VM as your tests. Selenium lets you write tests in various programming languages, including Java with JUnit 5.

Selenium WebDriver is the name of the key interface against which tests should be written. Selenium WebDriver is a W3C Recommendation and is implemented by many current browsers, including Chrome, Firefox, and Internet Explorer. WebDriver makes direct calls to the browser, using each browser's native support for automation. How these direct calls are made, and the features they support, depends on the browser you are using. Every browser has different logic for performing actions on the browser. Figure 15.2 shows the various components of the Selenium WebDriver architecture.

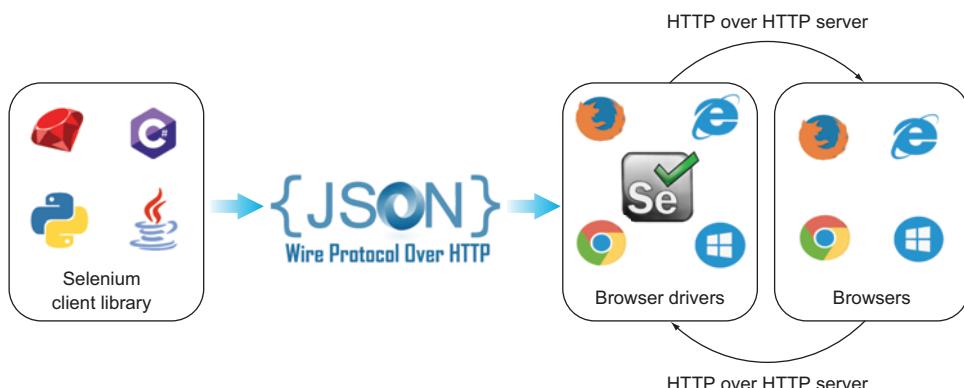


Figure 15.2 The Selenium WebDriver architecture, including the Selenium client library, the browser drivers, the browsers themselves, and HTTP communication

WebDriver includes the following four components:

- *Selenium client libraries*—Selenium supports multiple libraries for programming languages such as Java, C#, PHP, Python, Ruby, and so on.
- *JSON Wire Protocol over HTTP*—JavaScript Object Notation (JSON) is used to transfer data between servers and clients on the web. JSON Wire Protocol is a REST API that transfers the information to the HTTP server. Each WebDriver

(such as FirefoxDriver, ChromeDriver, InternetExplorerDriver, and so on) has its own HTTP server.

- *Browser drivers*—Each browser has a separate browser driver. Browser drivers communicate with the respective browser without revealing the internal logic of the browser functionality. When a browser driver receives a command, that command is executed on the browser, and the response goes back in the form of an HTTP response.
- *Browsers*—Selenium supports multiple browsers such as Firefox, Chrome, Internet Explorer, and Safari.

15.5 Writing Selenium tests

In this section, we will explore writing individual tests with Selenium. We will look at how to test for more than one browser and how to navigate the object model, and we will work through some example JUnit 5 tests. The first thing to do to set up a Selenium Java project is to include the Maven dependency into the project.

Listing 15.13 Selenium dependency in the pom.xml configuration

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
</dependency>
```

As of version 3, Selenium supports the browsers listed in table 15.2.

Table 15.2 Browsers supported by Selenium

Web browser	Browser driver class
Google Chrome	ChromeDriver
Internet Explorer	InternetExplorerDriver
Safari	SafariDriver
Opera	OperaDriver
Firefox	FirefoxDriver
Edge	EdgeDriver

To be able to test for a specific browser, we will need to download the Selenium drivers for that specific browser and include the access path to it on the OS path. The links to download each driver can be found by visiting <https://selenium.dev/downloads/>, navigating to the Browsers section, and choosing the browser you are interested in.

For demonstration purposes, we are going to use three of the most popular browsers: Google Chrome, Internet Explorer, and Mozilla Firefox. So, we download the Selenium drivers for them and copy them into a dedicated folder, as shown in figure 15.3.

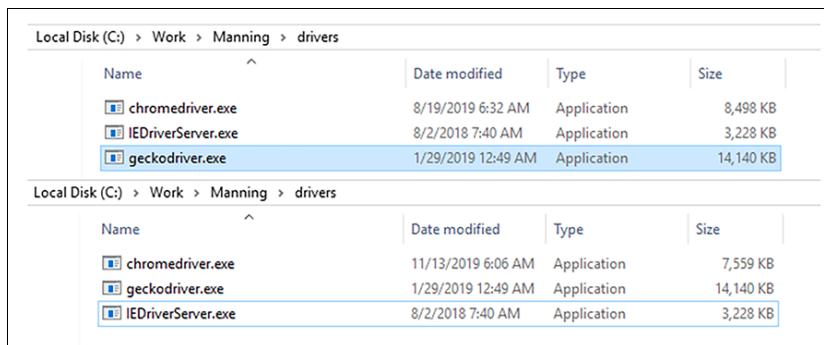


Figure 15.3 The folder containing the Selenium drivers

NOTE You must use the driver corresponding to the exact browser installed on your computer. For example, if your Google Chrome version is 79, you will only be able to use version 79 of the driver—versions such as 77, 78, and 80 will not work.

We need to include the folder with the Selenium drivers on the OS path. On Windows, we do this by choosing This PC > Properties > Advanced System Settings > Environment Variables > Path > Edit (figure 15.4). To do this on other OSs, consult the documentation.

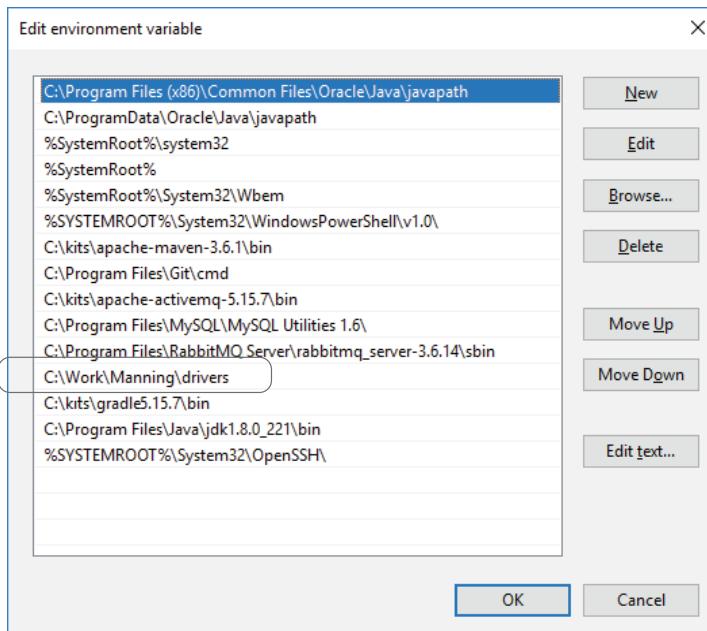


Figure 15.4 The path to the Selenium drivers folder added on the OS path

We have now set up the environment and are ready to write Selenium tests.

15.5.1 Testing for a specific web browser

Our tests will use specific browsers at first. We'll try two tests that access the Manning home page and the Google home page and verify that their titles are as expected. Listings 15.14 and 15.15 use Chrome and Firefox in two separate test classes along with JUnit 5 annotated methods.

Listing 15.14 Accessing the Manning and Google home pages with Chrome

```
[...]
public class ChromeSeleniumTest {

    private WebDriver driver; 1

    @BeforeEach
    void setUp() {
        driver = new ChromeDriver(); 2
    }

    @Test
    void testChromeManning() {
        driver.get("https://www.manning.com/");
        assertThat(driver.getTitle(), is("Manning | Home")); 3
    }

    @Test
    void testChromeGoogle() {
        driver.get("https://www.google.com");
        assertThat(driver.getTitle(), is("Google")); 4
    }

    @AfterEach
    void tearDown() {
        driver.quit(); 5
    }
}
```

Listing 15.15 Accessing the Manning and Google home pages with Firefox

```
[...]
public class FirefoxSeleniumTest {

    private WebDriver driver; 1'

    @BeforeEach
    void setUp() {
        driver = new FirefoxDriver(); 2'
    }

}
```

```

@Test
void testFirefoxManning() {
    driver.get("https://www.manning.com/");
    assertThat(driver.getTitle(), is("Manning | Home"));
}

@Test
void testFirefoxGoogle() {
    driver.get("https://www.google.com");
    assertThat(driver.getTitle(), is("Google"));
}

@AfterEach
void tearDown() {
    driver.quit();
}

```

In these listings:

- We declare a web driver ① ①', and we initialize it as ChromeDriver ② and as FirefoxDriver ②', respectively.
- The first test accesses the Manning home page and checks that the title of the page is “Manning | Home” ③ ③'. We do this using Hamcrest matchers (see chapter 2).
- The second test accesses the Google home page and checks that the title of the page is “Google” ④ ④'. We also do this using Hamcrest matchers.
- After executing each JUnit 5 test, we make sure we call the `quit` method of the web driver, which closes all the open browser windows. The driver instance becomes garbage collectible, meaning it is available to be removed from memory ⑤ ⑤'.

If we run each of these tests, the execution opens Chrome and Firefox, respectively, accesses the Manning and Google websites, and validates their titles.

15.5.2 Testing navigation using a web browser

Our next test will access the Wikipedia website, find an element on the page, and click it. We'll use JUnit 5 annotations and Firefox to do this.

Listing 15.16 Finding an element on a page with Firefox

```

[...]
public class WikipediaAccessTest {

    private RemoteWebDriver driver;           ← 1

    @BeforeEach
    void setUp() {
        driver = new FirefoxDriver();          ← 2
    }
}

```

```

    @Test
    void testWikipediaAccess() {
        driver.get("https://en.wikipedia.org/");
        assertThat(driver.getTitle(),
                   is("Wikipedia, the free encyclopedia"));
    }

    WebElement contents = driver.findElementByLinkText("Contents");
    assertTrue(contents.isDisplayed());

    contents.click();
    assertThat(driver.getTitle(),
               is("Wikipedia:Contents - Wikipedia"));
}

@AfterEach
void tearDown() {
    driver.quit();
}

```

In this listing:

- We declare a web driver as `RemoteWebDriver` ①. `RemoteWebDriver` is a class that implements the `WebDriver` interface and is extended by browser classes as `FirefoxDriver`, `ChromeDriver`, or `InternetExplorerDriver`. We declare the web driver as `RemoteWebDriver` in order to be able to call the method that finds elements on a web page and that is not declared by the `WebDriver` interface.
- We initialize the web driver as `FirefoxDriver` ②.
- We access the Wikipedia home page and check that the title of the page is “Wikipedia, the free encyclopedia” ③.
- We look for the Contents element and check that it is displayed ④.
- We click the Contents element and check that the newly displayed page has the title “Wikipedia:Contents – Wikipedia” ⑤.
- After executing the test, we make sure we call the `quit` method of the web driver, which closes all the open browser windows. The driver instance becomes garbage collectible, meaning it is available to be removed from memory ⑥.

TIP You can easily change this program to use another web driver, such as `ChromeDriver` or `InternetExplorerDriver`. Try doing this as an exercise.

15.5.3 Testing more than one web browser

To emphasize the advantages of the new JUnit 5 features and put them in practice in our Selenium tests, we will run the same test class with more than one browser. The following listing reworks our example of accessing the Manning and Google home pages as JUnit 5 parameterized tests running with different browsers.

Listing 15.17 Accessing Manning and Google with different browsers

```

public class MultiBrowserSeleniumTest {
    private WebDriver driver;                                1

    public static Collection<WebDriver> getBrowserVersions() {
        return Arrays.asList(new WebDriver[] {new FirefoxDriver(), new
            ChromeDriver(), new InternetExplorerDriver()});
    }

    @ParameterizedTest
    @MethodSource("getBrowserVersions")
    void testManningAccess(WebDriver driver) {                2
        this.driver = driver;
        driver.get("https://www.manning.com/");
        assertThat(driver.getTitle(), is("Manning | Home"));
    }

    @ParameterizedTest
    @MethodSource("getBrowserVersions")
    void testGoogleAccess(WebDriver driver) {                3
        this.driver = driver;
        driver.get("https://www.google.com");
        assertThat(driver.getTitle(), is("Google"));
    }

    @AfterEach
    void tearDown() {                                       4
        driver.quit();                                     5
    }
}

```

In this listing:

- We define a WebDriver field that will be initialized during test execution ①.
- We create the getBrowserVersions method that will serve as a method source for injecting the argument of each parameterized test ②. The method returns a collection of web drivers (FirefoxDriver, ChromeDriver, and InternetExplorerDriver) that are injected one by one into the parameterized tests.
- We define two parameterized tests, for which the getBrowserVersions method provides the test arguments ③. This means each of the parameterized methods is executed a number of times equal to the size of the collection returned by getBrowserVersions—and it uses a different browser each time it runs.
- The first test saves a reference to the web driver in the field variable, accesses the Manning home page, and checks that the title of the page is “Manning | Home” ④. We do this using Hamcrest matchers (see chapter 2). The JUnit 5

parameterized test is executed three times: once for each of the web drivers provided by the `getBrowserVersions` method. After executing each test, we make sure we call the `quit` method of the web driver, which closes all the open browser windows.

- The second test saves a reference to the web driver in the field variable, accesses the Google home page, and checks that the title of the page is “Google” ⑤. The test is also executed three times, once for each of the web drivers provided by `getBrowserVersions`. We also do this using Hamcrest matchers.
- After executing each test, we call the `quit` method of the web driver ⑥.

If we run this test class, the execution opens the Firefox, Chrome, and Internet Explorer browsers for each test, accesses the Manning and Google websites, and validates their titles. Two tests are executed through three browsers each—this is a total of six test executions.

15.5.4 Testing Google search and navigation using different web browsers

Next, we will see how to test a Google search by clicking one of the links we obtain from the search engine and navigating in the result. In the following listing, we search for “en.wikipedia.org” using Google, jump to the first result, and then navigate to an element inside it.

Listing 15.18 Testing Google search and navigating the Wikipedia website

```
public class GoogleSearchTest {
    private RemoteWebDriver driver; ①

    public static Collection<RemoteWebDriver> getBrowserVersions() {
        return Arrays.asList(new RemoteWebDriver[] {new FirefoxDriver(),
            new ChromeDriver(), new InternetExplorerDriver()}); ②
    }

    @ParameterizedTest
    @MethodSource("getBrowserVersions")
    void testGoogleSearch(RemoteWebDriver driver) { ③
        driver.get("http://www.google.com");
        WebElement element = driver.findElement(By.name("q"));
        element.sendKeys("en.wikipedia.org"); ④
        driver.findElement(By.name("q")).sendKeys(Keys.ENTER); ⑤

        WebElement myDynamicElement = (new WebDriverWait(driver, 10))
            .until(ExpectedConditions
            .presenceOfElementLocated(By.id("result-stats"))); ⑥

        List<WebElement> findElements =
            driver.findElements(By.xpath("//*[@id='rso']//a/h3")); ⑦
    }
}
```

```

        findElements.get(0).click();

        assertEquals("https://en.wikipedia.org/wiki/Main_Page",
                     driver.getCurrentUrl());
        assertThat(driver.getTitle(),
                   is("Wikipedia, the free encyclopedia"));

        WebElement contents = driver.findElementByLinkText("Contents");
        assertTrue(contents.isDisplayed());

        contents.click();
        assertThat(driver.getTitle(),
                   is("Wikipedia:Contents - Wikipedia"));
    }

    @AfterEach
    void tearDown() {
        driver.quit();
    }
}

```

In this listing:

- We declare a `RemoteWebDriver` field that will be initialized during test execution ①.
- We create the `getBrowserVersions` method that serves as a method source for injecting the argument of each parameterized test ②. The method returns a collection of `RemoteWebDrivers` (`FirefoxDriver`, `ChromeDriver`, and `InternetExplorerDriver`) that will be injected one by one into the parameterized tests. We need `RemoteWebDrivers` in order to call the method that finds elements on a web page and that is not declared by the `WebDriver` interface.
- We define one JUnit 5 parameterized test, for which `getBrowserVersions` provides the test argument ③. This means the parameterized method will be executed a number of times equal to the size of the collection returned by `getBrowserVersions`.
- We access the <http://www.google.com> website and look for the element with the “q” name ④. This is the name of the input edit box where we need to insert the text to search for; we can obtain it by right-clicking the element and choosing Inspect or Inspect Element (depending on the browser). We get the result shown in figure 15.5.
- We insert the text “en.wikipedia.org” to search for using Google, and we press Enter ⑤.
- We wait until the Google page shows the result, but no longer than 10 seconds ⑥.
- We use an XPath to get all the elements returned by the Google search ⑦. XPath is a query language for finding elements in XML. Right now, all you

need to know is that the `//*[@@id='rso']//a/h3` XPath provides the list of all elements returned by the Google search. For more about working with XPath in Selenium, see www.guru99.com/xpath-selenium.html.

- We click the first element on the list and check the URL and the title of the newly accessed page ⑧.
- On this new page, we look for the element with the “Contents” text and check that it is displayed ⑨ Then, we click this element and check that the title of the newly accessed page is “Wikipedia:Contents—Wikipedia” ⑩.
- After executing the test, we call the `quit` method of the web driver, which closes all the open browser windows. The driver instance becomes garbage collectible, meaning it is available to be removed from memory ⑪.

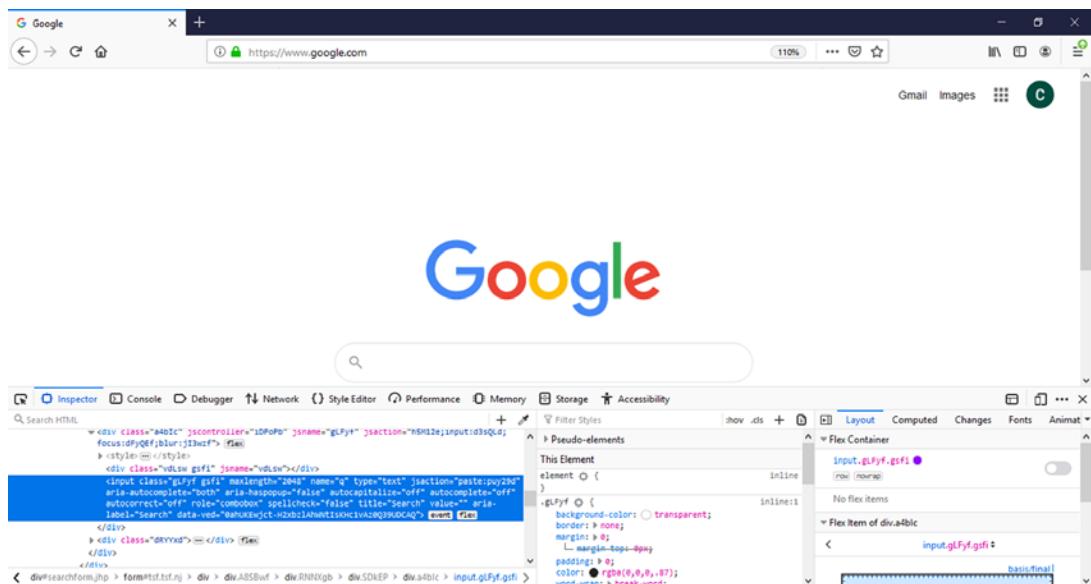


Figure 15.5 Inspecting the name of the input edit box on the Google home page with the help of Firefox

If we run this test, the execution opens the Firefox, Chrome, and Internet Explorer browsers. We have one test executed through three browsers—a total of three test executions.

15.5.5 Testing website authentication

Many of Tested Data Systems’ customers interact with their applications using websites where they need to authenticate. Authentication is critical for such applications because it verifies the identity of the user and allows access. Consequently, John needs to test authentication by writing tests to follow both successful and unsuccessful

scenarios. John chooses Selenium, as he would also like to follow the interaction with the website visually.

John will use the website <https://the-internet.herokuapp.com>, which provides many functionalities that can be automatically tested, including form authentication. The class in the following listing represents the interaction with the <https://the-internet.herokuapp.com> home page (figure 15.6).

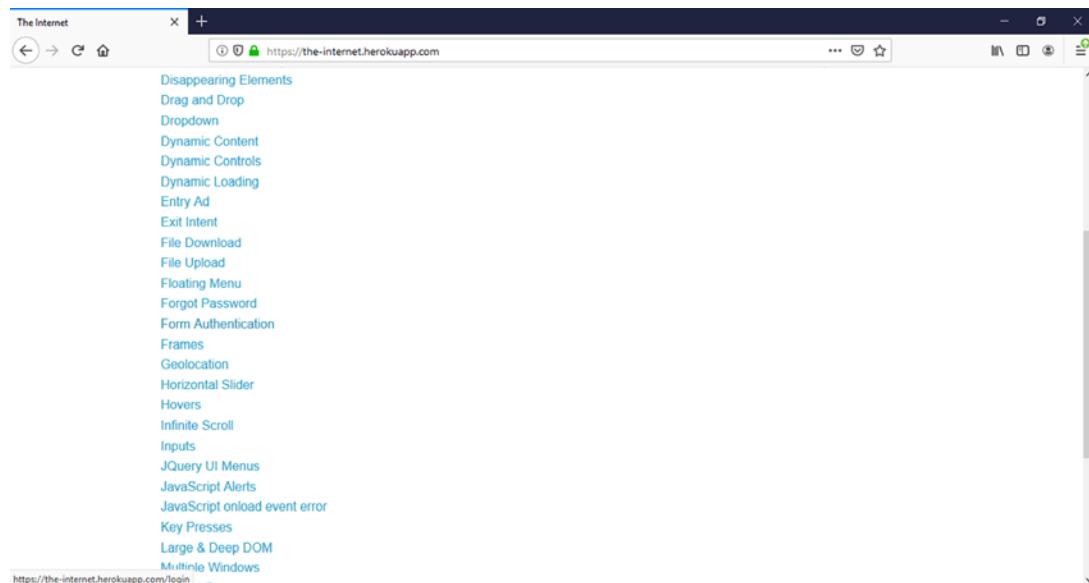


Figure 15.6 The <https://the-internet.herokuapp.com> home page

Listing 15.19 Class describing the home page of the tested website

```
[...]
public class Homepage {
    private WebDriver webDriver; 1

    public Homepage(WebDriver webDriver) {
        this.webDriver = webDriver; 2
    }

    public LoginPage openFormAuthentication() {
        webDriver.get("https://the-internet.herokuapp.com/");
        webDriver.findElement(By.cssSelector("[href=\"/login\"]"))
            .click();
        return new LoginPage(webDriver); 3
    }
}
```

Annotations:

- 1: A callout arrow points to the declaration of the `private WebDriver webDriver;` field.
- 2: A callout arrow points to the constructor `public Homepage(WebDriver webDriver) { this.webDriver = webDriver; }`.
- 3: A callout arrow points to the `return new LoginPage(webDriver);` statement.
- 4: A callout arrow points to the `webDriver.get("https://the-internet.herokuapp.com/");` line.
- 5: A callout arrow points to the `webDriver.findElement(By.cssSelector("[href=\"/login\"]"))` line.

In this listing:

- The Homepage class contains a private WebDriver field that is used to interact with the website ①. It is initialized by the constructor of the class ②.
- In the openFormAuthentication class, John accesses <https://the-internet.herokuapp.com/> ③. He looks for the element having the login hyperlink (href) and clicks it ④. He returns a new LoginPage ⑤ (described in the next listing) and receives the web driver as an argument of the constructor.

The class in the next listing represents the interaction with the <https://the-internet.herokuapp.com/login> login page (figure 15.7).

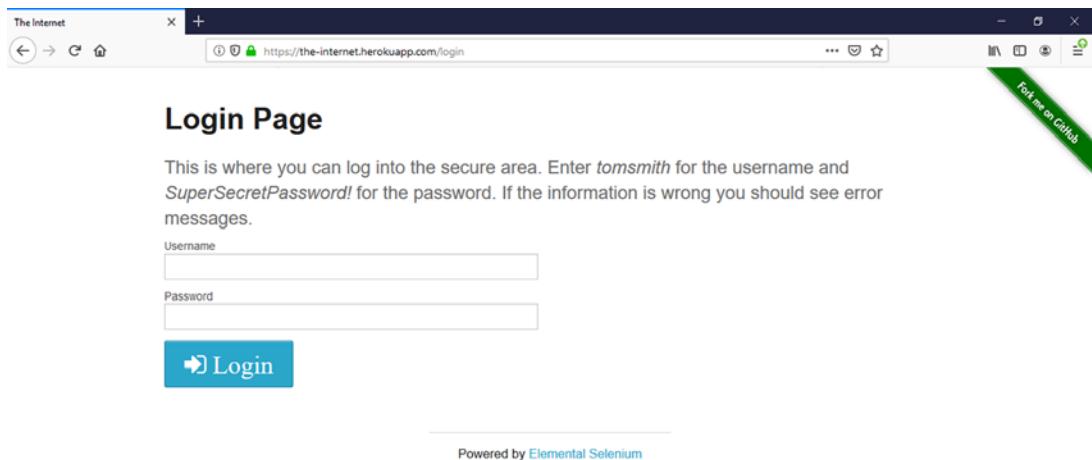


Figure 15.7 Login page at <https://the-internet.herokuapp.com/login>

Listing 15.20 Class describing the login page of the tested website

```
[...]
public class LoginPage {
    private WebDriver webDriver; 1

    public LoginPage(WebDriver webDriver) {
        this.webDriver = webDriver; 2
    }

    public LoginPage loginWith(String username, String password) {
        webDriver.findElement(By.id("username")).sendKeys(username);
        webDriver.findElement(By.id("password")).sendKeys(password); 3
    }
}
```

```

    webDriver.findElement(By.cssSelector("#login button")).click(); ← 4

    return this;
} ← 5

public void thenLoginSuccessful() {
    assertTrue(webDriver
        .findElement(By.cssSelector("#flash.success"))
        .isDisplayed());
    assertTrue(webDriver
        .findElement(By.cssSelector("[href=\"/logout\"]"))
        .isDisplayed());
}

public void thenLoginUnsuccessful() {
    assertTrue(webDriver.findElement(By.id("username"))
        .isDisplayed());
    assertTrue(webDriver.findElement(By.id("password"))
        .isDisplayed());
}
}

```

In this listing:

- The LoginPage class contains a private WebDriver field that is used to interact with the website ①. It is initialized by the constructor of the class ②.
- The loginWith method finds the elements with the IDs username and password and writes in their content the string arguments username and password from the method ③. Then, John finds the Login button using a CSS selector and clicks it ④. The method returns the same object, as John intends to execute the thenLoginSuccessful and thenLoginUnsuccessful methods on it ⑤.
- He checks for a successful login by verifying that the elements with the CSS selector #flash.success (the shaded bar in figure 15.8) and the hyperlink (href) logout (the Logout button in figure 15.8) are displayed on the page ⑥. Remember that you can get the name of an element by right-clicking it and choosing Inspect or Inspect element (depending on the browser).
- He checks for an unsuccessful login by verifying that the elements with the IDs username and password are displayed on the page ⑦. This is because after an unsuccessful login attempt, we remain on the same page (figure 15.9).

DEFINITION *CSS selector*—The combination of an element selector and a selector value that identifies web elements within a web page. For now, you only need to know that the #login button CSS selector finds this element on the web page. For more about CSS selectors, see https://www.w3schools.com/cssref/css_selectors.asp.

John uses the Homepage and LoginPage classes that he has created to run the successful and unsuccessful login testing scenarios, as shown in listing 15.21. Tom Smith

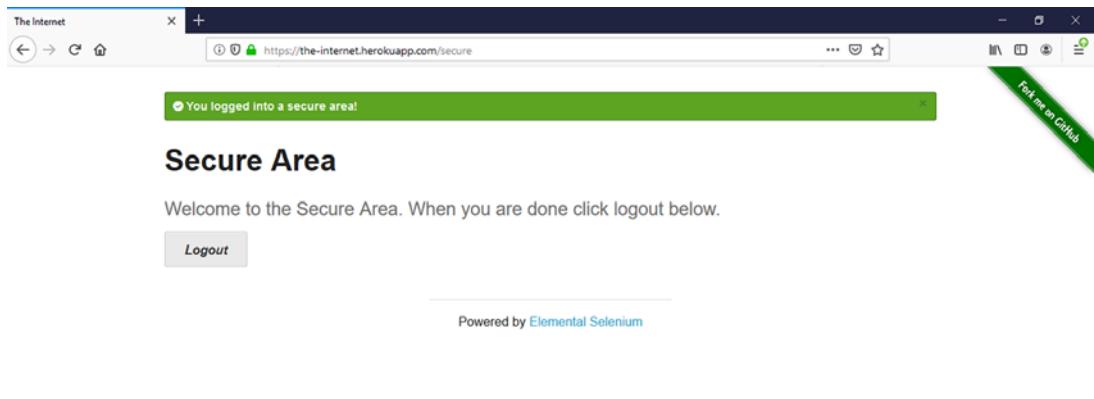


Figure 15.8 Successful login at <https://the-internet.herokuapp.com/secure>

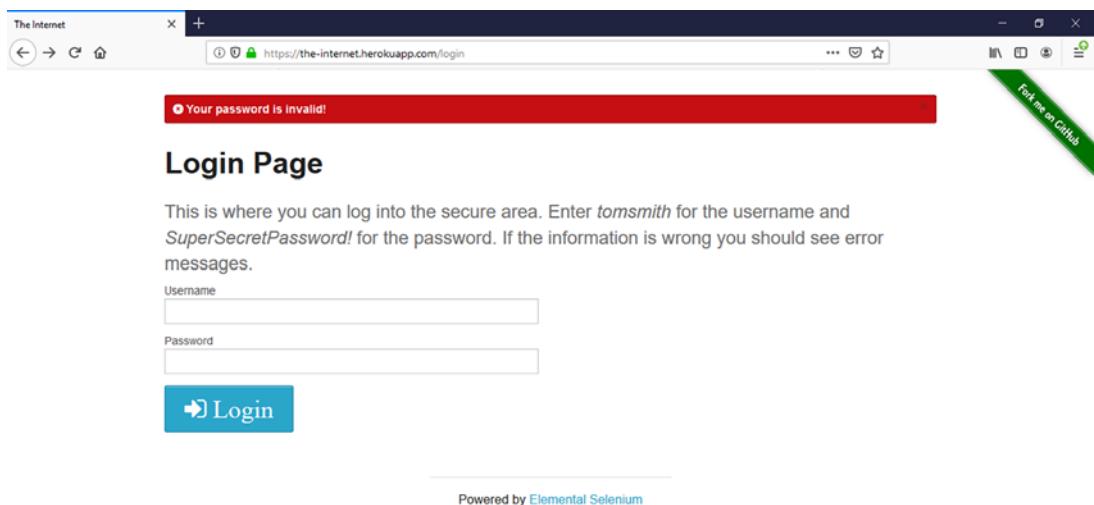


Figure 15.9 Unsuccessful login at <https://the-internet.herokuapp.com/secure>

is a test user who accesses a web interface for the example flight-management application. The credentials that allow the login are username `tomsmith` and password `SuperSecretPassword!`. John will work with this test user (never use real credentials in a test—doing so is a breach of security): he will create a test with the test user's

correct credentials (which should never be changed, as the test will fail otherwise) and another test with incorrect credentials.

Listing 15.21 Successful and unsuccessful login tests

```
[...]
public class LoginTest {

    private Homepage homepage;
    private WebDriver webDriver; 1

    public static Collection<WebDriver> getBrowserVersions() {
        return Arrays.asList(new WebDriver[] {new FirefoxDriver(),
                                              new ChromeDriver(), new InternetExplorerDriver()});
    } 2

    @ParameterizedTest
    @MethodSource("getBrowserVersions")
    public void loginWithValidCredentials(WebDriver webDriver) { 3
        this.webDriver = webDriver;
        homepage = new Homepage(webDriver); 4
        homepage
            .openFormAuthentication()
            .loginWith("tomsmith", "SuperSecretPassword!")
            .thenLoginSuccessful();
    } 5

    @ParameterizedTest
    @MethodSource("getBrowserVersions")
    public void loginWithInvalidCredentials(WebDriver webDriver) { 3
        this.webDriver = webDriver;
        homepage = new Homepage(webDriver); 4
        homepage
            .openFormAuthentication()
            .loginWith("tomsmith", "SuperSecretPassword")
            .thenLoginUnsuccessful();
    } 6

    @AfterEach
    void tearDown() { 7
        webDriver.quit();
    }
}
```

In this listing:

- John declares a `Homepage` field and a `WebDriver` field that will be initialized during test execution ①.
- He creates the `getBrowserVersions` method that will serve as a method source for injecting the argument of the parameterized tests ②. The method returns a collection of `WebDrivers` (`FirefoxDriver`, `ChromeDriver`, and