Community Experience Distilled

Test-Driven Java Development

Invoke TDD principles for end-to-end application development with Java

Viktor Farcic
Alex Garcia

[PACKT] PUBLISHING open source* community experience distilled

Test-Driven Java Development

Invoke TDD principles for end-to-end application development with Java

Viktor Farcic

Alex Garcia

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Test-Driven Java Development

First published: August 2015

Production reference: 1240815

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-742-9

`www.packtpub.com`

Credits

Authors
Viktor Farcic
Alex Garcia

Reviewers
Muhammad Ali
Jeff Deskins
Alvaro Garcia
Esko Luontola

Commissioning Editor
Julian Ursell

Acquisition Editor
Reshma Raman

Content Development Editor
Divij Kotian

Technical Editors
Manali Gonsalves
Naveenkumar Jain

Copy Editors
Trishya Hajare
Janbal Dharmaraj

Project Coordinator
Neha Bhatnagar

Proofreader
Safis Editing

Indexer
Priya Sane

Production Coordinator
Shantanu N. Zagade

Cover Work
Shantanu N. Zagade

About the Authors

Viktor Farcic is a software architect. He has coded using a plethora of languages, starting with Pascal (yes, he is old), Basic (before it got the Visual prefix), ASP (before it got the .Net suffix) and moving on to C, C++, Perl, Python, ASP.Net, Visual Basic, C#, JavaScript, and so on. He has never worked with Fortran. His current favorites are Scala and JavaScript, even though he works extensively on Java. While writing this book, he got sidetracked and fell in love with Polymer and GoLang.

His big passions are test-driven development (TDD), behavior-driven development (BDD), Continuous Integration, Delivery, and Deployment (CI/CD).

He often speaks at community gatherings and conferences and helps different organizations with his coaching and training sessions. He enjoys constant change and loves working with teams eager to enhance their software craftsmanship skills.

He loves sharing his experiences on his blog, `http://TechnologyConversations.com`.

I would like to thank a lot of people who have supported me during the writing of this book. The people at Everis and Defie (companies I worked with earlier) provided all the support and encouragement I needed. The technical reviewers, Alvaro Garcia, Esko Luontola, Jeff Deskins, and Muhammad Ali, did a great job by constantly challenging my views, my assumptions, and the quality of the code featured throughout the examples. Alvaro provided even more help by writing the Legacy Code chapter. His experience and expertise in the subject were an invaluable help. The Packt Publishing team was very forthcoming, professional, and always available to provide guidance and support. Finally, I'd like to give a special thanks to my daughter, Sara, and wife, Eva. With weekdays at my job and nights and weekends dedicated to this book, they had to endure months without the support and love they deserve. This book is dedicated to "my girls".

Alex Garcia started coding in C++ but later moved to Java. He is also interested in Groovy, Scala, and JavaScript. He has been working as a system administrator and also as a programmer and consultant.

He states that in the software industry, the final product quality is the key to success. He truly believes that delivering bad code always means unsatisfied customers. He is a big fan of Agile practices.

He is always interested in learning new languages, paradigms, and frameworks. When the computer is turned off, he likes to walk around sunny Barcelona and likes to practice sports.

I did enjoy writing this book and I want to thank those people who made this possible. First of all, thanks to the staff at Packt Publishing for giving me this opportunity and the guidance along this difficult journey. Thanks to the technical reviewers, Alvaro Garcia, Esko Luontola, Jeff Deskins, and Muhammad Ali, for the tips and corrections; they added great value with their comments. Thank you, Viktor Farcic, for sharing this experience with me. It has been a pleasure to be your mate during this adventure. And finally, special thanks to my parents, my brother, and my girlfriend for being there whenever I need them. This book is dedicated with love to all of them.

About the Reviewers

Muhammad Ali is a software development expert with extensive experience in telecommunications and the air and rail transport industries. He holds a master's degree in the distributed systems course of the year 2006 from the Royal Institute of technology (KTH), Stockholm, Sweden, and holds a bachelor's honors degree in computer science from the University of Engineering & Technology Lahore, Pakistan.

He has a passion for software design and development, cloud and big data test-driven development, and system integration. He has built enterprise applications using the open source stack for top-tier software vendors and large government and private organizations worldwide. Ali has settled in Stockholm, Sweden, and has been providing services to various IT companies within Sweden and outside Europe.

I would like to thank my parents; my beloved wife; Sana Ali, and my adorable kids, Hassan and Haniya, for making my life wonderful.

Jeff Deskins has been building commercial websites since 1995. He loves to turn ideas into working solutions. Lately, he has been building most of his web applications in the cloud and is continuously learning best practices for high-performance sites.

Prior to his Internet development career, he worked for 13 years as a television news photographer. He continues to provide Internet solutions for different television stations through his website, `www.tvstats.com`.

I would like to thank my wife for her support and patience through the many hours of me sitting behind my laptop learning new technologies. Love you the most!

Alvaro Garcia is a software developer who firmly believes in the eXtreme Programming methodology. He's embarked on a lifelong learning process and is now in a symbiotic exchange process with the Barcelona Software Craftsmanship meet-up, where he is a co-organizer.

Alvaro has been working in the IT industry for product companies, consulting firms, and on his own since 2005. He occasionally blogs at `http://alvarogarcia7.github.io`.

He enjoys reading and reviewing technology books and providing feedback to the author whenever possible to create the best experience for the final reader.

Esko Luontola has been programming since the beginning of the 21st century. In 2007, he was bitten by the TDD bug and has been test-infected ever since. Today, he has tens of projects under his belt using TDD and helps others get started with it; some of his freely available learning material includes the TDD Tetris Tutorial exercise and the Let's Code screencasts. He is also fluent in concurrency, distributed systems, and the deep ends of the JVM platform. In recent years, his interests have included Continuous Delivery, DevOps, and microservices.

Currently, Esko is working as a software consultant at Nitor Creations. He is the developer of multiple open source projects such as Retrolambda for back porting Java 8 code to Java 5-7 and the Jumi Test Runner in order to run tests faster and more flexibly than JUnit.

Table of Contents

Preface

Test-driven development has been around for a while and many people have still not adopted it. The reason behind this is that TDD is difficult to master. Even though the theory is very easy to grasp, it takes a lot of practice to become really proficient with it.

Authors of this book have been practicing TDD for years and will try to pass on their experience to you. They are developers and believe that the best way to learn some coding practice is through code and constant practice. This book follows the same philosophy. We'll explain all the TDD concepts through exercises. This will be a journey through the TDD best practices applied to Java development. At the end of it, you will earn a TDD black belt and have one more tool in your software craftsmanship tool belt.

What this book covers

Chapter 1, Why Should I Care for Test-driven Development?, spells out our goal of becoming a Java developer with a TDD black belt. In order to know where we're going, we'll have to discuss and find answers to some questions that will define our voyage.

Chapter 2, Tools, Frameworks, and Environments, will compare and set up all the tools, frameworks and environments that will be used throughout this book. Each of them will be accompanied with code that demonstrates their strengths and weaknesses.

Chapter 3, Red-Green-Refactor – from Failure through Success until Perfection, will help us develop a Tic-Tac-Toe game using the red-green-refactor technique, which is the pillar of TDD. We'll write a test and see it fail; we'll write a code that implements that test, run all the tests and see them succeed, and finally, we'll refactor the code and try to make it better.

Chapter 4, *Unit Testing – Focusing on What You Do and Not on What Has Been Done*, shows that to demonstrate the power of TDD applied to unit testing, we'll need to develop a Remote Controlled Ship. We'll learn what unit testing really is, how it differs from functional and integration tests, and how it can be combined with test-driven development.

Chapter 5, *Design – If It's Not Testable, It's Not Designed Well*, will help us develop a Connect4 game without any tests and try to write tests at the end. This will give us insights into the difficulties we are facing when applications are not developed in a way that they can be tested easily.

Chapter 6, *Mocking – Removing External Dependencies*, shows how TDD is about speed. We want to quickly demonstrate some idea/concept. We'll continue developing our Tic-Tac-Toe game by adding MongoDB as our data storage. None of our tests will actually use MongoDB since all communications to it will be mocked.

Chapter 7, *BDD – Working Together with the Whole Team*, discusses developing a Book Store application by using the BDD approach. We'll define the acceptance criteria in the BDD format, carry out the implementation of each feature separately, confirm that it is working correctly by running BDD scenarios, and if required, refactor the code to accomplish the desired level of quality.

Chapter 8, *Refactoring Legacy Code – Making it Young Again*, will help us refactor an existing application. The process will start with creation of test coverage for the existing code and from there on we'll be able to start refactoring until both the tests and the code meet our expectations.

Chapter 9, *Feature Toggles – Deploying Partially Done Features to Production*, will show us how to develop a Fibonacci calculator and use feature toggles to hide functionalities that are not fully finished or that, for business reasons, should not yet be available to our users.

Chapter 10, *Putting It All Together*, will walk you through all the TDD best practices in detail and refresh the knowledge and experience you gained throughout this book.

What you need for this book

The exercises in this book require readers to have a 64 bit computer. Installation instructions for all required software is provided throughout the book.

Who this book is for

If you're an experienced Java developer and want to implement more effective methods of programming systems and applications, then this book is for you.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
public class Friendships {
    private final Map<String, List<String>> friendships = new
      HashMap<>();

    public void makeFriends(String person1, String person2) {
        addFriend(person1, person2);
        addFriend(person2, person1);
    }
```

Any command-line input or output is written as follows:

```
$> vagrant plugin install vagrant-cachier
$> git clone thttps://bitbucket.org/vfarcic/tdd-java-ch02-example-
  vagrant.git
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Once we type our search query, we should find and click the **Go** button."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Why Should I Care for Test-driven Development?

This book is written by developers for developers. As such, most of the learning will be through code. Each chapter will present one or more TDD practices and we'll try to master them by solving katas. In karate, kata is an exercise where you repeat a form many times, making little improvements in each. Following the same philosophy, we'll be making small, but significant improvements from one chapter to the next. You'll learn how to design and code better, reduce time-to-market, produce always up-to-date documentation, obtain high code coverage through quality tests, and write clean code that works.

Every journey has a start and this one is no exception. Our destination is a Java developer with the **test-driven development** (**TDD**) black-belt.

In order to know where we're going, we'll have to discuss, and find answers, to some questions that will define our voyage. What is TDD? Is it a testing technique, or something else? What are the benefits of applying TDD?

The goal of this chapter is to obtain an overview of TDD, to understand what it is and to grasp the benefits it provides for its practitioners.

The following topics will be covered in this chapter:

- Understanding TDD
- What is TDD?
- Testing
- Mocking
- Executable documentation
- No debugging

Why TDD?

You might be working in an agile or waterfall environment. Maybe you have well-defined procedures that were battle-tested through years of hard work, or maybe you just started your own start-up. No matter what the situation was, you likely faced at least one, if not more, of the following pains, problems, or causes for unsuccessful delivery:

- Part of your team is kept out of the loop during the creation of requirements, specifications, or user stories
- Most, if not all, of your tests are manual, or you don't have tests at all
- Even though you have automated tests, they do not detect real problems
- Automated tests are written and executed when it's too late for them to provide a real value to the project
- There is always something more urgent than dedicating time to testing
- Teams are split between testing, development, and functional analysis departments, and they are often out of sync
- An inability to refactor the code because of the fear that something will be broken
- The maintenance cost is too high
- The time-to-market is too big
- Clients do not feel that what was delivered is what they asked for
- Documentation is never up to date
- You're afraid to deploy to production because the result is unknown
- You're often not able to deploy to production because regression tests take too long to run
- Team is spending too much time trying to figure out what some method or a class does

Test-driven development does not magically solve all of these problems. Instead, it puts us on the way towards the solution. There is no silver bullet, but if there is one development practice that can make a difference on so many levels, that practice is TDD.

Test-driven development speeds up the time-to-market, enables easier refactoring, helps to create better design, and fosters looser coupling.

On top of the direct benefits, TDD is a prerequisite for many other practices (continuous delivery being one of them). Better design, well-written code, faster time-to-market, up-to-date documentation, and solid test coverage, are some of the results you will accomplish by applying TDD.

It's not an easy thing to master TDD. Even after learning all the theory and going through best practices and anti-patterns, the journey is only just beginning. TDD requires time and a lot of practice. It's a long trip that does not stop with this book. As a matter a fact, it never truly ends. There are always new ways to become more proficient and faster. However, even though the cost is high, the benefits are even higher. People who spent enough time with TDD claim that there is no other way to develop a software. We are one of them and we're sure that you will be too.

We are strong believers that the best way to learn some coding technique is by coding. You won't be able to finish this book by reading it in a metro on the way to work. It's not a book that one can read in bed. You'll have to get your hands dirty and code.

In this chapter, we'll go through basics; starting from the next, you'll be learning by reading, writing, and running code. We'd like to say that by the time you're finished with this book, you'll be an experienced TDD programmer, but this is not true. By the end of this book, you'll be comfortable with TDD and you'll have a strong base in both theory and practice. The rest is up to you and the experience you'll be building by applying it in your day-to-day job.

Understanding TDD

At this time, you are probably saying to yourself "OK, I understand that TDD will give me some benefits, but what exactly is test-driven development?" TDD is a simple procedure of writing tests before the actual implementation. It's an inversion of a traditional approach where testing is performed after the code is written.

Red-green-refactor

Test-driven development is a process that relies on the repetition of a very short development cycle. It is based on the test-first concept of **extreme programming** (**XP**) that encourages simple design with a high level of confidence. The procedure that drives this cycle is called **red-green-refactor**.

The procedure itself is simple and it consists of a few steps that are repeated over and over again:

1. Write a test.
2. Run all tests.
3. Write the implementation code.
4. Run all tests.
5. Refactor.
6. Run all tests.

Since a test is written before the actual implementation, it is supposed to fail. If it doesn't, the test is wrong. It describes something that already exists or it was written incorrectly. Being in the green state while writing tests is a sign of a false positive. Tests like these should be removed or refactored.

[While writing tests, we are in the red state. When the implementation of a test is finished, all tests should pass and then we will be in the green state.]

If the last test failed, implementation is wrong and should be corrected. Either the test we just finished is incorrect or the implementation of that test did not meet the specification we had set. If any but the last test failed, we broke something and changes should be reverted.

When this happens, the natural reaction is to spend as much time as needed to fix the code so that all tests are passing. However, this is wrong. If a fix is not done in a matter of minutes, the best thing to do is to revert the changes. After all, everything worked not long ago. Implementation that broke something is obviously wrong, so why not go back to where we started and think again about the correct way to implement the test? That way, we wasted minutes on a wrong implementation instead of wasting much more time to correct something that was not done right in the first place. Existing test coverage (excluding the implementation of the last test) should be sacred. We change the existing code through intentional refactoring, not as a way to fix recently written code.

[Do not make the implementation of the last test final, but provide just enough code for this test to pass.]

Write the code in any way you want, but do it fast. Once everything is green, we have confidence that there is a safety net in the form of tests. From this moment on, we can proceed to refactor the code. This means that we are making the code better and more optimum without introducing new features. While refactoring is in place, all tests should be passing all the time.

If, while refactoring, one of the tests failed, refactor broke an existing functionality and, as before, changes should be reverted. Not only that at this stage we are not changing any features, but we are also not introducing any new tests. All we're doing is making the code better while continuously running all tests to make sure that nothing got broken. At the same time, we're proving code correctness and cutting down on future maintenance costs.

Once refactoring is finished, the process is repeated. It's an endless loop of a very short cycle.

Speed is the key

Imagine a game of ping pong (or table tennis). The game is very fast; sometimes it is hard to even follow the ball when professionals play the game. TDD is very similar. TDD veterans tend not to spend more than a minute on either side of the table (test and implementation). Write a short test and run all tests (ping), write the implementation and run all tests (pong), write another test (ping), write implementation of that test (pong), refactor and confirm that all tests are passing (score), and then repeat—ping, pong, ping, pong, ping, pong, score, serve again. Do not try to make the perfect code. Instead, try to keep the ball rolling until you think that the time is right to score (refactor).

[Time between switching from tests to implementation (and vice versa) should be measured in minutes (if not seconds).]

It's not about testing

T in TDD is often misunderstood. Test-driven development is the way we approach the design. It is the way to force us to think about the implementation and to what the code needs to do before writing it. It is the way to focus on requirements and implementation of just one thing at a time—organize your thoughts and better structure the code. This does not mean that tests resulting from TDD are useless—it is far from that. They are very useful and they allow us to develop with great speed without being afraid that something will be broken. This is especially true when refactoring takes place. Being able to reorganize the code while having the confidence that no functionality is broken is a huge boost to the quality.

[The main objective of test-driven development is testable code design with tests as a very useful side product.]

Testing

Even though the main objective of test-driven development is the approach to code design, tests are still a very important aspect of TDD and we should have a clear understanding of two major groups of techniques as follows:

- Black-box testing
- White-box testing

The black-box testing

Black-box testing (also known as functional testing) treats software under test as a black-box without knowing its internals. Tests use software interfaces and try to ensure that they work as expected. As long as functionality of interfaces remains unchanged, tests should pass even if internals are changed. Tester is aware of what the program should do, but does not have the knowledge of how it does it. Black-box testing is most commonly used type of testing in traditional organizations that have testers as a separate department, especially when they are not proficient in coding and have difficulties understanding it. This technique provides an external perspective on the software under test.

Some of the advantages of black-box testing are as follows:

- Efficient for large segments of code
- Code access, understanding the code, and ability to code are not required
- Separation between user's and developer's perspectives

Some of the disadvantages of black-box testing are as follows:

- Limited coverage, since only a fraction of test scenarios is performed
- Inefficient testing due to tester's lack of knowledge about software internals
- Blind coverage, since tester has limited knowledge about the application

If tests are driving the development, they are often done in the form of acceptance criteria that is later used as a definition of what should be developed.

[Automated black-box testing relies on some form of automation such as **behavior-driven development** (**BDD**).]

The white-box testing

White-box testing (also known as clear-box testing, glass-box testing, transparent-box testing, and structural testing) looks inside the software that is being tested and uses that knowledge as part of the testing process. If, for example, an exception should be thrown under certain conditions, a test might want to reproduce those conditions. White-box testing requires internal knowledge of the system and programming skills. It provides an internal perspective on the software under test.