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 **test-driven development** (**TDD**) practices and we'll try to master them by solving **katas**. In karate, a 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** (**TTM**), 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 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 any 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
- There is 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
- The team is spending too much time trying to figure out what some method or a class does

TDD 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.

TDD 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 TTM, 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 of 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 have 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 TDD?* 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

TDD 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, the 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. An 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 optimal 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, refactoring 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 even to 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 the 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. TDD is the way we approach the design. It is the way to force us to think about the implementation and what the code needs to do before writing it. It is the way to focus on requirements and the 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—they are 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 its quality.

The main objective of TDD is testable code design with tests as a very useful side product.

Testing

Even though the main objective of TDD 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

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 the functionality of interfaces remains unchanged, tests should pass even if internals are changed. The tester is aware of what the program should do, but does not have the knowledge of how it does it. Black-box testing is the 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:

- It is efficient for large segments of code
- Code access, understanding the code, and ability to code are not required
- It offers separation between users and developers perspectives

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

- It provides limited coverage, since only a fraction of test scenarios is performed
- It can result in inefficient testing due to tester's lack of knowledge about software internals
- It can lead to blind coverage, since testers have 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**).

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.

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

- It is efficient in finding errors and problems
- Required knowledge of internals of the software under test is beneficial for thorough testing
- It allows finding hidden errors

- It encourages programmer's introspection
- It helps in optimizing the code
- Due to the required internal knowledge of the software, maximum coverage is obtained

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

- It might not find unimplemented or missing features
- It requires high-level knowledge of internals of the software under test
- It requires code access
- Tests are often tightly coupled to the implementation details of the production code, causing unwanted test failures when the code is refactored

White-box testing is almost always automated and, in most cases, take the form of unit tests.

When white-box testing is done before the implementation, it takes the form of TDD.

The difference between quality checking and quality assurance

The approach to testing can also be distinguished by looking at the objectives they are trying to accomplish. Those objectives are often split between **quality checking** (**QC**) and **quality assurance** (**QA**). While QC is focused on defects identification, QA tries to prevent them. QC is product-oriented and intends to make sure that results are as expected. On the other hand, QA is more focused on processes that assure that quality is built-in. It tries to make sure that correct things are done in the correct way.

While QC had a more important role in the past, with the emergence of TDD, **acceptance test-driven development** (**ATDD**), and later on BDD, focus has been shifting towards QA.

Better tests

No matter whether one is using black-box, white-box, or both types of testing, the order in which they are written is very important.

Requirements (specifications and user stories) are written before the code that implements them. They come first so they define the code, not the other way around. The same can be said for tests. If they are written after the code is done, in a certain way, that code (and the functionalities it implements) is defining tests. Tests that are defined by an already existing application are biased. They have a tendency to confirm what code does, and not to test whether a client's expectations are met, or that the code is behaving as expected. With manual testing, that is less the case since it is often done by a siloed QC department (even though it's often called QA). They tend to work on test definition in isolation from developers. That in itself leads to bigger problems caused by inevitably poor communication and the **police syndrome**, where testers are not trying to help the team to write applications with quality built in, but to find faults at the end of the process. The sooner we find problems, the cheaper it is to fix them.

Tests written in the TDD fashion (including its flavors such as ATDD and BDD) are an attempt to develop applications with quality built in from the very start. It's an attempt to avoid having problems in the first place.

Mocking

In order for tests to run quickly and provide constant feedback, code needs to be organized in such a way that the methods, functions, and classes can be easily replaced with mocks and stubs. A common word for this type of replacements of the actual code is **test double**. The speed of execution can be severely affected with external dependencies; for example, our code might need to communicate with the database. By mocking external dependencies, we are able to increase that speed drastically. Whole unit test suite execution should be measured in minutes, if not seconds. Designing the code in a way that can be easily mocked and stubbed forces us to structure that code better by applying a separation of concerns.

More important than speed is the benefit of the removal of external factors. Setting up databases, web servers, external APIs, and other dependencies that our code might need, is both time consuming and unreliable. In many cases, those dependencies might not even be available. For example, we might need to create a code that communicates with a database and have someone else create a schema. Without mocks, we would need to wait until that schema is set.

With or without mocks, the code should be written in such a way that we can easily replace one dependency with another.

Executable documentation

Another very useful aspect of TDD (and well-structured tests in general) is documentation. In most cases, it is much easier to find out what the code does by looking at tests than at the implementation itself. What is the purpose of some methods? Look at the tests associated with it. What is the desired functionality of some part of the application UI? Look at the tests associated with it. Documentation written in the form of tests is one of the pillars of TDD and deserves further explanation.

The main problem with (traditional) software documentation is that it is not up-to-date most of the time. As soon as some part of the code changes, the documentation stops reflecting the actual situation. This statement applies to almost any type of documentation, with requirements and test cases being the most affected.

The necessity to document code is often a sign that the code itself is not well-written. Moreover, no matter how hard we try, documentation inevitably gets outdated.

Developers shouldn't rely on system documentation because it is almost never up-to-date. Besides, no documentation can provide as detailed and up-to-date a description of the code as the code itself.

Using code as documentation does not exclude other types of documents. The key is to avoid duplication. If details of the system can be obtained by reading the code, other types of documentation can provide quick guidelines and a high-level overview. Non-code documentation should answer questions such as what the general purpose of the system is and what technologies are used by the system. In many cases, a simple `README` is enough to provide the quick start that developers need. Sections such as project description, environment setup, installation, and build and packaging instructions are very helpful for newcomers. From there on, code is the bible.

Implementation code provides all needed details while test code acts as the description of the intent behind the production code.

Tests are executable documentation with TDD being the most common way to create and maintain it.

Assuming that some form of **continuous integration** (**CI**) is in use, if some part of the test documentation is incorrect, it will fail and be fixed soon afterwards. CI solves the problem of incorrect test documentation, but it does not ensure that all functionality is documented. For this reason (among many others), test documentation should be created in the TDD fashion. If all functionality is defined as tests before the implementation code is written and the execution of all tests is successful, then tests act as a complete and up-to-date information source that can be used by developers.

What should we do with the rest of the team? Testers, customers, managers, and other non-coders might not be able to obtain the necessary information from the production and test code.

As we saw earlier, two of the most common types of testing are black-box and white-box testing. This division is important since it also divides testers into those who do know how to write or at least read code (white-box testing) and those who don't (black-box testing). In some cases, testers can do both types. However, more often than not, they do not know how to code so the documentation that is usable by developers is not usable by them. If documentation needs to be decoupled from the code, unit tests are not a good match. That is one of the reasons why BDD came in to being.

BDD can provide documentation necessary for non-coders, while still maintaining the advantages of TDD and automation.

Customers need to be able to define new functionality of the system, as well as to be able to get information about all the important aspects of the current system. That documentation should not be too technical (code is not an option), but it still must be always up-to-date. BDD narratives and scenarios are one of the best ways to provide this type of documentation. The ability to act as acceptance criteria (written before the code), be executed frequently (preferably on every commit), and be written in a natural language makes BDD stories not only always up-to-date, but usable by those who do not want to inspect the code.

Documentation is an integral part of the software. As with any other part of the code, it needs to be tested often so that we're sure that it is accurate and up-to-date.

The only cost-effective way to have accurate and up-to-date information is to have executable documentation that can be integrated into your CI system.

TDD as a methodology is a good way to move along in this direction. On a low-level, unit tests are a best fit. On the other hand, BDD provides a good way to work on a functional level while maintaining understanding that is accomplished by using natural language.

No debugging

We (authors of this book) almost never debug applications we're working on!

This statement might sound pompous, but it's true. We almost never debug because there is rarely a reason to debug an application. When tests are written before the code and the code coverage is high, we can have high confidence that the application works as expected. This does not mean that applications written using TDD do not have bugs—they do. All applications do. However, when that happens, it is easy to isolate them by simply looking for the code that is not covered by tests.

Tests themselves might not include some cases. In those situations, the action is to write additional tests.

With high code coverage, finding the cause of some bug is much faster through tests than spending time debugging line by line until the culprit is found.

Summary

In this chapter, you got the general understanding of TDD practice and insights into what TDD is and what it isn't. You learned that it is a way to design code through a short and repeatable cycle called Red-Green-Refactor. Failure is an expected state that should not only be embraced, but enforced throughout the TDD process. This cycle is so short that we move from one phase to another with great speed.

While code design is the main objective, tests created throughout the TDD process are a valuable asset that should be utilized and severely impact our view of traditional testing practices. We went through the most common of those practices, such as white-box and black-box testing, tried to put them into the TDD perspective, and showed benefits that they can bring to each other.

You discovered that mocks are very important tools that are often a must when writing tests. Finally, we discussed how tests can and should be utilized as executable documentation and how TDD can make debugging much less necessary.

Now that we are armed with theoretical knowledge, it is time to set up the development environment and get an overview and comparison of different testing frameworks and tools.