Tampere University of Applied Sciences

Test-driven development (TDD)

TDD techniques, advantages and disadvantages

Elina Widdowson

REPORT
March 2023

Bachelor's Degree Programme in Software Engineering

Key words: test-driven development, tdd

CONTENTS

GLOSSARY or ABBREVIATIONS AND TERMS (choose one or other)

TAMK	Tampere University of Applied Sciences
TDD	test-driven development

1 INTRODUCTION

There are many different testing methodologies in used in software development. Testing is done to ensure that the program functions as it is meant to and expected to. Tests reveal problems in the code and make it easier to pinpoint when and where in the code the problems are. Software that is rigorously tested is less likely to have unnoticed bugs or performance issues because tests help expose these problems. Having tests can also save time and make development more efficient because problems are easier to notice and find. In companies and commercial software products and services, proper testing can also help the company save money because they need to spend less time fixing bugs or releasing updates for badly performing software.

Testing can also be done manually or automated. Manual testing is where the tester interacts directly with the software interface or APIs, whereas automated tests are done by a machine that executes test scripts (Pittet, 2023). There are also many different kinds of tests. For example, unit tests are very low level and involve testing small code components, such as functions or class methods Pittet, 2023). Integration tests are used to test how the different modules of the application work with each other. Performance testing is used to test the application under different workloads to make sure it can scale and remain responsive. In addition, there are different approaches, methods and techniques used for software testing.

One testing technique that can be used in software development is test-driven development (TDD). According to Beck (2002), TDD is a set of techniques that encourage creating simple designs and test suites that can be run with confidence. There are specific steps that should be taken when using TDD and a cycle that guides testing and development with TDD. There are different advantages to using TDD, such as having a robust test suite that can be used to find problems as soon as they arise. However, there are also disadvantages to using TDD, such as time required to learn and implement TDD.

2 BASICS OF TEST-DRIVEN DEVELOPMENT

2.1 Introduction to test driven development

Test-driven development (TDD) is a set of techniques used for software testing and development. In TDD, the developer begins by writing a test case before writing any code. When not using TDD, developers usually write code first and then write tests for the code after they have created the desired functionality. TDD is largely centered around writing unit tests, which test a small component of the code, such as one function. Since TDD is a set of techniques that guide the way tests are written, TDD can be used with any programming language or testing framework. TDD is used to write automated tests, which can be easily run as a test suite. Automated tests are more efficient to run than manual tests.

According to Beck (2002), there are two main rules to follow when doing TDD. The first rule is that before you write any code you must first write a failing automated test for it. Since the TDD process begins with writing a test for code that does not exist yet, the test case will fail at the beginning. The second rule for TDD is to remove duplication from your code as you work on (Beck, 2002). The goal of TDD, as well as most software engineering, is to write clean code that works. Beck (2002) notes that with TDD the first priority is to make the code work and once it functions correctly then the developer should focus on making the code clean.

2.2 Clean code and testing

Clean code can be described in many different ways. According to Martin (2008), clean code is simple, orderly, well-written, and maintained well. Martin (2008) describes it as elegant, beautiful and easy to understand. Clean code consists of many different aspects such as using meaningful and understandable names for variables, functions, etc in the code; breaking down functionality into small parts; and avoiding duplication.

There are many different problems that arise from messy or badly written code. The first problem is that it is difficult for other developers to understand messy code (Martin, 2008). Since most developers work in teams or companies with multiple developers, they must work together. If the code is messy then takes a lot of time and effort for different developers to understand the code. It also difficult to modify the code because small changes can break other parts of the system, as Martin (2008) points out. Finding and fixing bugs is also much more difficult in messy code than clean code. Therefore, it is important for developers to write clean that functions well and is easy to understand and modify. The TDD techniques are designed to help developers write code clean as well as create robust tests that accurately and reliably test the code.

Martin (2008) also emphasizes the importance of writing clean tests in addition to writing clean code. According to Martin (2008), there are three things that make a test clean: “readability, readability, and readability”. Tests can be made more readable in the same manner that code is made more readable by focusing on “clarity, simplicity, and density of expression” (Martin, 2008). Since TDD involves writing a large number of tests, these tests can become unmanageable if they are not kept organized. The tests should be easy to go through and each test should be relevant to the current state of the code.

Therefore, when writing tests with TDD, developers should aim to make their tests clean and readable by expressing things as simply and concisely as possible. Clean tests are easier for other people to read and understand. They are also easier to modify in the future if there are changes to the functionality or design of the program.

3 IMPLEMENTATION TECHNIQUES FOR TDD

3.1 The TDD cycle

The general TDD cycle consists of three stages (Beck, 2002). It begins by the developer writing a test and inventing the ideal interface that they would like to

have. Although Beck (2002) notes that this interface may have to be changed later to make the code actually function as intended and fit the current design and system. The second stage consists of making the test run in the simplest and fastest way possible. At this stage it is acceptable for the developer to write bad, messy code for the sake of quickly creating the desired functionality.

In the final stage of the TDD cycle the developer should fix all of the badly written messy code to make it clean. At this stage they should also remove duplication and ensure that the code is easy to read and understand. Although it is acceptable for code to be messy during the initial stages of TDD, the code should never be left in this state as one of the goals of TDD is to write clean code (Beck, 2002). Therefore, TDD does not give developers an excuse to write code quickly and leave it as a mess. The code must always be checked over to ensure that it meets good development standards. Likewise, a test case may initially be written quickly and messily, however, the test must also be cleaned up afterwards to ensure that the tests follow the principles of clean code. Tests must also be easy for others to understand and they need to be easy to modify to make them flexible.

This TDD cycle is repeated constantly throughout the software development project for each new functionality that is introduced to the system. With TDD, functionality is added incrementally. For example, writing a small test and then writing a single method to make the test pass. Since tests are written before the code, all the new code that is written will have at least one test as well.

3.2 TDD steps

According to Beck (2002), the TDD process has five steps that a developer should do. The first step is to write a quick test case. The second step is to run all the current tests and see the newly written test case fail. The third step is to make a small change to the new test case. Then all the tests should be run again and they should pass. Finally, the developer should refactor their code and remove any duplication.

However, there is no hard rule for how much or little work a developer should do during each step in the cycle. Since developers are writing unit tests during TDD, these tests should be quite small and specific. However, often it is necessary to have many tests even for small software components, such as a function. The tests need to cover expected functionality and edge cases and often all the different aspects that need to be tested cannot be covered in a single test. Unit tests should also be testing very specific functionalities, therefore, a single test should not cover multiple different aspects to be tested. The developer must find a suitable scope for each test and then write the minimum amount of code to make the test pass.

Beck (2002) also explains that new design elements should only be introduced when they become necessary. Developers should not introduce new elements ahead of time because they may or may not be necessary in the future. This ensures that no extra code is being written. Beck (2002) also emphasizes the notion that developers should ask themselves how they want to test a specific feature and the moment and not how do they want to implement the feature. The testing is at the forefront of TDD and the tests drive the development of the code. According to Beck (2002), developers should aim to tell a story with their tests and this story describes the functionality and how it should be implemented.

3.3 Strategies for making tests run green quickly

During the TDD cycle, one goal is make tests run green as quickly as possible. This means that the developer should get their tests to pass successfully as fast as possible. Beck (2002) explains that each test should be testing something small. This means that it should be quick to write the test and afterwards the functionality because it is a small change that was made to the codebase. After the developer has written a new test, they should run it and look at the all the new errors that appear. Fixing all these errors one by one provides a clear list of things that must be added or modified in the code to get the test to run green. According to Beck (2002), the developer should first try to get the test to compile successfully and then after that prioritize making it pass. Getting a test to

compile could mean creating an empty class or dummy function that does not yet function as intended, however, gets the test to compile.

Beck (2002) also defines three different strategies that take a different approach to making tests pass quickly. The first strategy is fake it. When using this strategy, the developer can begin writing tests and code with constants and then gradually replace them with the real variables that will be needed to implement the correct functionality.

The second strategy is the obvious implementation. With this strategy the developer can write the real implementation straight away, however, this strategy should only be used if the real implementation is very obvious and easy to implement. If the solution seems complex then the work should be broken down into smaller steps and the final implementation written only after it has become clear.

The third strategy Beck (2002) describes is triangulation. With triangulation the developer should not generalize code straight away. Instead the developer should wait until there are two or more examples of a given solution and only then make the generalisation. This way a generalisation is not made before it becomes necessary, which allows the developer to save time and effort because no unnecessary work is done. Each of these strategies can be used at any point during the TDD cycle and Beck (2002) recommends using the strategy that best suits the current needs and shifting between the strategies as necessary.

3.4 Refactoring and removing duplication

Refactoring code and removing duplication is an important part of the TDD process. According to Beck (2002), the developer should always check their code after writing new tests/code and having the tests pass. Refactoring code and removing duplication are part of writing clean code and because the goal of TDD is to produce clean code, it is important for developers to constantly improve their code through refactoring and removing duplication when necessary.

Removing duplication the code also reduces dependencies. Beck (2002) points out that when developers must remember to remove duplication both from the code and duplication between the test data and the code data.

Since TDD is centred around writing tests first and code second, when refactoring code the developer should also write tests first (Beck, 2002). The developer should first write tests for how they want the refactored code to work. This way there is certainty that refactoring the code will not break anything because the tests are already in place and will show any errors encountered. If there are no tests in place before refactoring code then problems caused by the refactoring may not be noticeable until later during the development process. Beck (2002) notes that this makes bugs more difficult to find and fix as there will be new functionality and dependencies as well. If problems are found and fixed as soon as they occur then they are easier to deal with.

While refactoring the developer should also delete any old tests that have become redundant due to added functionality or other new modifications that were made to the code (Beck, 2002). This ensures that all the tests in the project are reliable and useful. Old tests can also cause confusion later if it is unclear whether or not they are actually testing any current functionality. When using TDD, there will be a very large number of tests in the software project, which also increases the need for keeping tests tidy and up to date. This makes it easier to work with the tests in the future. In addition to deleting unnecessary tests, Beck (2002) notes that it is important to modify tests to reflect changes in the code after new functionality is added. Modifying old tests is also easier if unnecessary tests have already been removed. This also ensures that developers do not waste time and effort modifying test cases that are no longer useful for the project.

In some cases tests can also be refactored. For example, the code base could contain a Dollar class and Euro class with each having their own tests. These tests were first written before the Euro and Dollar class were created, as is the standard with TDD. However, then a parent Currency class is created for all currencies because the developer wants to have many different currencies and abstracting this idea to a new parent class gives the program a better design.

The test cases dealing with Euros or Dollars can then be converted to using the new Currency class instead of each child class because testing Euros and Dollars separately is redundant as they both now inherit from Currency and function the same.

3.5 Considerations after implementation is finished

There are several things developers should also consider after they have finished their project. Beck (2002) notes that even though there is always room for improvement at some point all the desired features will have been implemented and the project is finished. At this point, Beck (2002) advises developers to review their design and make sure it is cohesive. In addition, developers should also look for any lingering duplication in the code and remove it in order to keep the code the clean.

Beck (2002) also advises developers to remember that TDD does not replace all other types of testing and developers should still remember to use other relevant testing methods and tools. For example, performance, stress, and usability testing are advisable to do in addition to TDD. These tests will cover features that were not tested in the tests written during the TDD cycle. For example, unit tests only test the functionality of the code components and do not test whether or not different modules operate correctly with each other. For this purpose, integration tests should also be written. Unit tests also do not show how the program performs under different work loads or numbers of users, therefore, performance testing should be done for this purpose if these factors are also a concern for the application being developed.

4 PROS AND CONS OF TEST-DRIVEN DEVELOPMENT

4.1 Pros of TDD

There are many benefits of using TDD during software development. According to Beck (2002), “by saying everything two ways—both as code and as tests—we hope to reduce our defects enough to move forward with confidence.” Another benefit of TDD is that you can modify your codebase with confidence. Often it is difficult to know beforehand whether a change will result in bugs or break other parts of the code. With TDD, developers do not need to spend time speculating or debating the possible effects of a change to the code. Instead, a developer can make their desired modifications, run all the current tests and instantly see whether or not the tests are still passing. This gives instant feedback about the functionality of new code.

Martin (2008) also claims that unit tests are the tool that keeps code reusable, maintainable and flexible. When software has sufficient unit tests in place, developers cannot accidentally make breaking changes to the system, which gives them the freedom to make changes whenever necessary without worrying about negative consequences elsewhere in the code. It is due to this that the presence of unit tests allows developers to create flexible code that is easy to change in the future.

In addition, when using TDD, functionality is added slowly in small increments (Beck, 2002). Due to this, there will never be unnecessary code written and implemented before it is useful. This stops the developer wasting time creating implementations just in case they are useful later and allows them to focus on the most important features and functionality. Beck (2002) also explains that with TDD developers can check their assumptions about how they think the code should work or will work by writing tests for these assumptions. Sometimes the assumptions underpinning a design choice are false and when using TDD it is quick to notice when this happens and the developer can then modify their approach.

Finally, using TDD means that the code will be thoroughly tested because writing tests is at the forefront of the TDD cycle. Therefore, developers can be sure that there are sufficient tests for the code and can rely on these tests to show problems with the code as soon as these problems arise. Having tests in place before writing the code can also allow the developer to think about implementation details that they might not have thought of without having spent time creating tests first.

4.2 Cons of TDD

Although there are many benefits to using TDD, there are also disadvantages to this way of working and testing software. The biggest disadvantage of TDD is that it is time-consuming (Beck, 2002). It takes time to practice TDD and become comfortable producing software using this method. There are no specific rules for how much a developer should do in one step or how quick and drastic changes to the code should be. Each developer must figure this out and do what is appropriate for their current situation. This is a skill that must be learned and practised.

In addition, it takes time to write tests. The time spent writing tests is time that cannot be spent writing code. In companies this use of time might be problematic because developers have deadlines to meet and must balance their time between writing tests and implementing functionality and bug fixes in the code. Using TDD can, however, allow developers to become more efficient at writing code. If this is the case, they can balance their time between writing code and tests and still get the same amount of code written in less time. However, this requires practice and there is no guarantee that using TDD will always allow developers to be more efficient and quick at writing code.

Furthermore, it may be difficult to use TDD in projects where multiple developers must collaborate on the same codebase and everyone on the team is not using TDD. If this is the case then there will not be test cases already in place for all the functionality in the project. During TDD, developers rely on their tests to cover all the functionality in the project and alert them as soon as a

problem arises. If the tests in the project do not cover all the functionality then the tests cannot be relied upon to detect problems as soon as they arise.

5 DISCUSSION

Test-driven development is one approach to testing in software development. TDD is a technique where the developer always writes tests before writing any code. Using this technique ensures that all the code is covered by suitable tests and gives developers the freedom and confidence to change their code without worrying about breaking changes. Having tests in place with TDD allows developers to quickly and easily test their entire codebase by constantly running all the tests that have been written so far.

However, it is important to note that TDD is focused on writing unit tests, which test the smallest parts of the program. Therefore, using TDD to write tests does not replace the need for other types of testing, such as integration tests or usability tests. Due to this, developers using TDD to write unit tests should also create additional tests that test other features of the program that are not covered with unit tests.

Using TDD to develop software and write tests has several benefits. Firstly, all production code is already covered by unit tests, which means that bugs and other problems are quickly seen and can be fixed as soon as they arise. Using TDD also encourages developers to write code simply and succinctly because writing simple tests that can be quickly made green is the main work flow in TDD. Developers can also test their assumptions with TDD as the tests immediately show the developer whether new additions or modifications function as they should. Refactoring and removing duplication are also an important part of the TDD cycle, which encourages developers to keep their code clean.

However, TDD also has some disadvantages. The biggest disadvantage of TDD is that it is more time consuming that just writing code. Writing tests takes time that cannot be spent writing code. Although TDD is meant to encourage

developers to write more efficiently, hence reducing the time needed to write code, achieving this level of efficiency takes practice and time. TDD can also be difficult to implement properly when working in a team unless all team members are motivated and dedicated to following the TDD cycle and always writing tests before code. If TDD is not implemented properly and there are tests missing for code then the benefits of using TDD are diminished as the tests cannot be relied upon to accurately evaluate how all the parts of the program are working.

Therefore, TDD has both advantages and disadvantages. When TDD is implemented properly it can be used to create robust and reliable unit tests that accurately evaluate whether or not the program is functioning as intended. It also encourages good software development habits and writing clean code and clean tests. However, developers will need time and practice to become efficient at using TDD to write tests and code.

REFERENCES

Beck, K. 2002. Test Driven development by Example. Addison-Wesley Professional. Electronic edition available on O'Reilly.

Martin, R. C. 2008. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson. Electronic edition available on O'Reilly.

Pittet, S. 2023. "The different types of software testing". Atlassian. Read on 01.03.2023. Available at: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing