THE NELSON MANDELA AFRICAN INSTITUTION OF SCIENCE AND

TECHNOLOGY (NM-AIST)

School of Computational and Communication Science and Engineering (CoCSE)

Group Assignment

Title: Challenges and opportunities of Test-Driven Development

S/N	Name	Reg No	Course
1.	Desire Asiimwe	M078/UG21	EMoS
2.	Carmel Nkeshimana	M068/BI21	EMoS
3.	Dickson Msaky	M028/T21	EMoS

Course Code : EMoS 6308

Course Name : System Development Methodology

Lecturer : Dr. Devotha Nyambo

Challenges and opportunities of Test-Driven Development

Abstract

Software testing plays a significant role in determining software quality. Through software testing, the developer can find defects and errors. However, not every developer is interested in the software testing phase. The most used testing methodology by many developers is the traditional testing practice, where testing is done after completely developing the software. Because of this, the consequences can be significant in terms of time, financials, and software quality. To respond to this situation, Kent Beck, a software developer, introduced a programming practice called Test-Driven Development focused on creating clean, simple code that satisfies the requirements. Test-Driven Development is a software programming practice that emphasizes test-first development, where the developer comes up with the tests before the code. The practice derives its foundation from Agile software development principles and extreme programming. This paper explores the opportunities and challenges of using Test-Driven Development practice. According to most literature, the practice provides many opportunities, such as improved code quality, better application quality, and minimal time spent on debugging. The challenges associated are minimal compared to opportunities, including the time it takes to be understood and increased development time.

Keywords: Test-driven development (TDD), Development team, Refactoring

1. Introduction To Test-Driven Development

Testing is one of the most important aspects of any software development project. (Khan & Khan, 2014) Through a comprehensive software testing approach, you can find the defects, errors, and false assumptions made earlier. (Gaur et al., 2017) In order to make good software, a variety of testing techniques, including regression testing, unit testing, and integration testing, need to be adequately utilized in the software development process. (Medewar, 2022)

Traditionally, software testing is done after the developer has finished writing the code. The software is then checked for defects and errors at the end, where it can take a lot of time to examine all the software components (Yang et al., 2006). Unfortunately, the amount of time that developers spend testing the software is severely limited by their need to meet deadlines and financial restrictions. To respond to this situation, *Kent Beck*, a software developer, introduced a programming practice called TDD with a focus on creating clean, simple code that satisfies the requirements. (Beck, 2002) In TDD, the developer has an additional role, not only as a coder but also as a tester.

TDD is a software programming practice that emphasizes test-first development. The developer comes up with the tests before the code. (Nanda, 2021) This forces developers to focus on a comprehensive examination of the requirements before writing code for a particular functionality; as a result, these tests drive the design of the software. (Janzen & Saiedian, 2008) The development process starts by writing the test case, then the simplest code to pass the test, and finally refactoring, i.e., refining the codes by checking and removing some duplications in the complete code. In short, TDD refers to the practice of creating test cases first and then writing enough code to make them pass.

2. The cycle of TDD

The cycle of TDD follows the Red-Green-Refactor cycle (Beck, 2002), which is characterized by repetitive processes, as shown in Fig. 1.

i. Red - Write a failing test.
ii. Green - Write code that is just enough to make the test pass.
iii. Refactor - Optimize, then clean up the design.

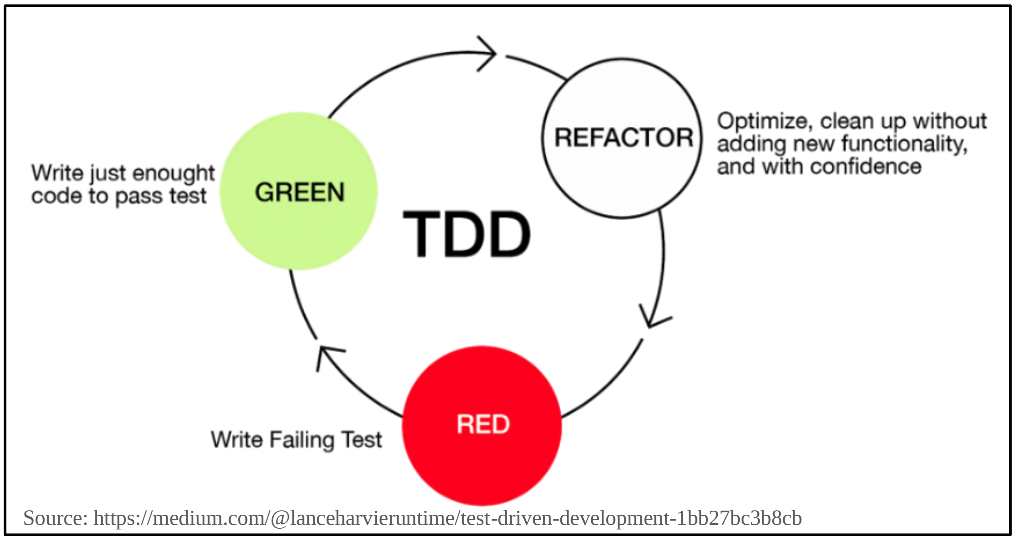
REFACTOR
Optimize, clean up without adding new functionality, and with confidence
Write just enought code to pass test
GREEN
TDD
Write Failing Test
RED
Source: https://medium.com/@lanceharvieruntime/test-driven-development-1bb27bc3b8cb

Figure 1: The Cycle of TDD

RED – This is the first step of the cycle where the software developer starts by writing a test that captures the new requirement. At this stage, the test is expected to fail but should be related to the functionality implemented. (Beck, 2002)

GREEN – This is the second step of the cycle where the software developer proceeds to write minimum code to satisfy the test written in the first step. At this step, the developer is not worried about making the software pretty good and improving it, just to meet the requirement and pass the test. (Beck, 2002)

REFACTOR – This is the third step of the cycle where the software developer revises the codes to improve the quality of the software, e.g., readability and maintainability. The software developer also improves code structure by optimizing and cleaning up without adding new functionality. (Beck, 2002)

3. TDD Implementation Processes

The different processes involved in a TDD software programming practice are as follows; (Illustrated in Fig 2.)

3.1 Write the test

The developer starts by writing a test that defines a small requirement of the software. This is an essential part since it involves analyzing what needs to be achieved by understanding the requirements which are presented.

3.2 Run test to see if it fails

Once the test is created, the next step is to run them for the first time, where it should fail. It is expected since there is no code for the test to run on. This help ensures the test is not implemented in a way it always passes.

3.3 Write the functionality

At this point, the developer begins writing the code that ensures the test pass and nothing more. The developer should keep it as simple as possible as long it can pass the tests. The code written at this stage is not final, and refactoring for improvements will come in later stages.

3.4 Run the test again

The developer should rerun the test using the codes written in the previous step. The written code must be optimized if the test cases fail until they do. This helps ensure that the newly added code meets the test requirement.

3.5 Refactor the code

Once the tests pass, the developer should now clean the written codes to make sure it adheres to standards. The developer should look to remove any duplications and inconsistencies that may have been introduced during the previous step. To ensure that the refactoring process won't change any current functionality, the developer should run the test cases repeatedly during each refactoring phase, as shown in Fig 2.

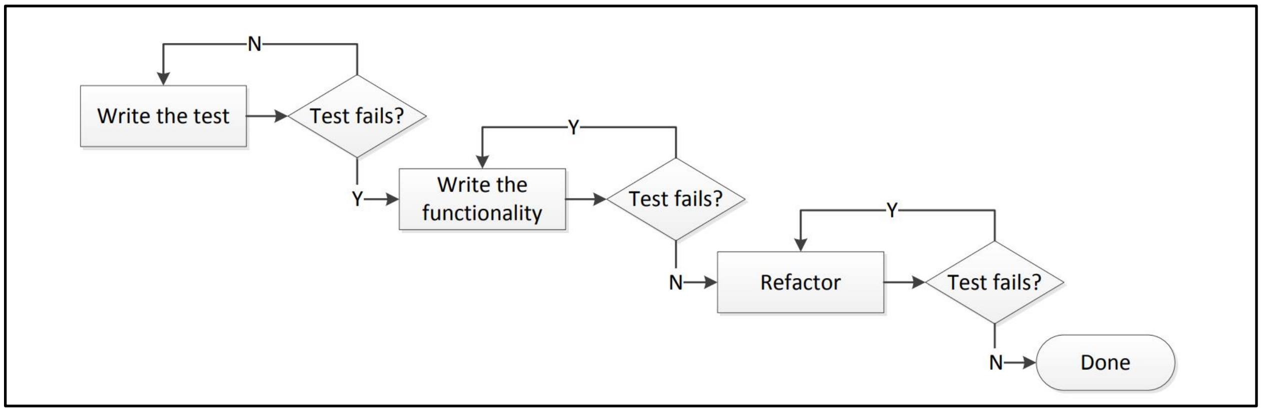

Figure 2: Test-Driven Development activities. Source*:* (Madeyski & Kawalerowicz, 2013)

4. Levels of TDD

TDD consists mainly of two levels, Acceptance TDD and Developer TDD. Acceptance TDD is a practice in which the tests are written from the users' point of view, fulfilling the software requirements. In developer TDD, the tests are written from the developer's point of view, which satisfies the software requirement. Developer TDD aims to define a detailed design for the execution of the software. (Nanda, 2021)

5. Opportunities of TDD

5.1 Improved code quality

Compared to traditional software development TDD is simple and produces clean and more meaningful code. This comes as a result of a deeper understanding developed by the programmer while writing the test code for the specified functionality. Test code writing helps to uncover uncertainties that are confirmed first by the client before proceeding to build the real production code. (Buchan, Li, G., & MacDonell, 2011)

5.2 Refactoring leads to perfectly well-working code

Refactoring is the process of restructuring code while not changing its original functionality. According to (Buchan, Li, G., & MacDonell , 2011), TDD not only helps in building the developers' confidence but also increases their readiness to advance the code design.

5.3 Better application quality

Using TDD, the application developed is released with fewer defects, leading to increased software reliability. The quality is attributed to the tests the codes go through before implementation. TDD provides an increased likelihood of product acceptability, which helps to manage and evaluate the cost of the project while reducing the uncertainty about unwanted side effects (Parsons, Lal, & Lange, 2011). (Nanthaamornphong & Carver, 2017) in their work mention that TDD helps developers to achieve better quality software; particularly in terms of reliability and functionality.

5.4 Produces code that is easy to maintain

The code developed using refactoring is considered more readable and easier to maintain. The process of writing tests before coding makes it possible to obtain constant comments on what to improve at the developers' end, which leads to an improvement in software quality, thus making it easy to maintain the future. (Parsons, Lal, & Lange, 2011).

TDD allows embedding documentation within the code and using these comments, it is easy to maintain the software developed without causing it to malfunction (Canfora et al., 2006).

5.5 Team productivity and motivation

TDD invokes the developers to think more critically and work together as a team and this builds team cohesion. When the developed code finally passes the tests, developers are motivated to continue working because they have the assurance that what they have developed is what the client needs (Buchan, Li, G., & MacDonell, 2011).

5.6 TDD allows room for changes to the software as per the customer's needs

Focus is given to only one feature of the software until it meets the needs of the users. This ensures that development is requirement-driven and the standpoint of the customer is not neglected as the development process goes on (Parsons, Lal, & Lange, 2011).

5.7 TDD enables incremental development

TDD gives a provision for the development team to focus on one function of the system at a time. This implies that even when all functionalities of the system are not known at the beginning of the project, the team can work on what is known at the time, and there is room for scalability (Janzen & Saiedian, 2005).

5.8 Less time spent on debugging

Since the developers are already familiar with what exactly the users' needs through the testing process, less bugs are encountered during the coding process compared to when traditional development models are used and thus less time is spent on debugging the code (DESPA, 2014) Testing new functionality while using TDD is quicker as there is less revision made to the test codes in order to come up with a functionality that will pass the test and much as writing test code may appear time-consuming at the beginning, the cost to fix bugs immediately is way cheaper compared to when discovered months later.

6. Challenges in TDD

6.1 Takes time for TDD to be understood by everyone on the development team

One major issue with TDD is that it takes time to make it understandable to everyone on the development team, so many engineers or project managers only begin to reap its benefits much later. This means that if you want to introduce TDD into a company recently, it will cost more time to get everyone on board, especially developers who weren't familiar with its usability before. It is challenging to learn TDD, particularly if you are a developer without prior TDD expertise. (Buchan et al., 2011) Teams require continuous reminders of the advantages of their work before they become used to it.

6.2 Team leader explanation/ Team leader biasness

It is common for the development team leader to base his explanation of TDD's advantages solely on his personal experience without taking into account the opinions of his team and this is an obstacle to the adoption of TDD procedures. The development team can spend one year to integrate and internalize it while also becoming aware of the productivity and code quality gains. (Buchan et al., 2011)

6.3 Time to accommodate/ Developers' perspective on TDD

Some developers require a long time to understand how TDD works and they occasionally experience disruption from a new team member who has no prior experience with TDD. It takes new team members several months to shift their perspective toward TDD. His presence might also lead to disagreement and lower morale (Buchan et al., 2011)

To adopt TDD, the development team must be skilled at refactoring and writing high-quality tests. To adhere to the TDD technique, which demands more formal training and can boost productivity, they must continue to acquire refactoring and developing skills. (Nanthaamornphong & Carver, 2017)

6.4 Misunderstanding from the upper management

Upper management occasionally misinterprets the TDD process when they claim that developers spend more time creating tests than working on functional code. And this is accurate because the team spends more time providing functionality than they did in the past when they were using non-TDD approaches (Buchan et al., 2011). To overcome this, the development team leader needs to create a certain level of awareness among the top management concerning TDD in order to win their full support and trust.

6.5 Learning TDD process gets monotonous and boring

Learning TDD methodology and implementing it can get boring because the process of writing tests for an existing system, refactoring application code, refactoring tests, and determining and deciding on areas of the application that can be extended becomes monotonous as time goes on. (Mugridge, n.d.)

6.6 Increase development time

Considering the time spent to implement a certain set of requirements, it is difficult to measure the duration spent while developing a software product. As time increases, an up-front loss might overshadow a long-term gain at the side of the organization for which the software is being developed. This can be considered a business-critical factor for adopting new practices within the organization (Nanthaamornphong & Carver, 2017).

Conclusion

This paper explored the opportunities and challenges of using TDD practice. Based on the discussed opportunities, TDD provides more benefits to development teams and the organization as a whole since it comes with many opportunities to the team in terms of developing good quality software within a short time and also the ease of maintenance of the software developed.

Considering the crucial importance to some software application i.e. in banking, industries, hospital and etc., a security flaw can cause a significant impact. Since TDD focuses on creating clean, simple code that satisfies the requirements, it can help address such issues early since tests drive the software design. Also, TDD practice when compared to prescriptive process models such as the Waterfall model provides more benefits for the critical system since it does not follow the linear sequential activities. In the waterfall model, each phase is fully completed before the beginning of the next phase, and testing is done at the after coding.

References

Beck, K. (2002). *Test Driven Development: By Example* (1st ed.). Addison Wesley.

Gaur, J., Goyal, A., Choudhury, T., & Sabitha, S. (2017). A walk through of software testing techniques. *Proceedings of the 5th International Conference on System Modeling and Advancement in Research Trends, SMART 2016*, 103–108. https://doi.org/10.1109/SYSMART.2016.7894499

Janzen, D. S., & Saiedian, H. (2008). Does test-driven development really improve software design quality? *IEEE Software*, *25*(2), 77–84. https://doi.org/10.1109/MS.2008.34

Khan, M. E., & Khan, F. (2014). *Importance of Software Testing in Software Development Life Cycle*. www.IJCSI.org

Madeyski, L., & Kawalerowicz, M. (2013). Continuous Test-Driven Development-A Novel Agile Software Development Practice and Supporting Tool. *ENASE 2013 - Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*, 283. https://doi.org/10.5220/0004587202600267

Medewar, S. (2022). *Types of Software Testing: Different Testing Types with Details*. https://hackr.io/blog/types-of-software-testing

Nanda, V. (2021). *What is Test-Driven Development (TDD)?* https://www.tutorialspoint.com/what-is-test-driven-development-tdd

Yang, Q., Li, J. J., & Weiss, D. (2006). A survey of coverage based testing tools. *Proceedings - International Conference on Software Engineering*, 99–103. https://doi.org/10.1145/1138929.1138949

Buchan, J., Li, L., G., S., & MacDonell . (n.d.). Causal Factors, Benefits and Challenges of Test-Driven Development:. *18th Asia-Pacific Software Engineering Conference*, (p. 2011).

Janzen, D., & Saiedian, H. (2005). Test-Driven Development: Concepts, Taxonomy, and Future Direction. *Computer*. doi:10.1109/MC.2005.314

Parsons, D., Lal, R., & Lange, M. (2011). Test Driven Development: Advancing Knowledge by Conjecture. *Future Internet* .

Mugridge, R. (n.d.). *Challenges in Teaching Test Driven Development*.

Canfora, G., Cimitile, A., Garcia, F., Piattini, M., & Visaggio, C. A. (2006). Evaluating advantages of test driven development: A controlled experiment with professionals. *ISESE'06 - Proceedings of the 5th ACM-IEEE International Symposium on Empirical Software Engineering*, *2006*, 364–371. https://doi.org/10.1145/1159733.1159788

Nanthaamornphong, A., & Carver, J. C. (2017). Test-Driven Development in scientific software: a survey. *Software Quality Journal*, *25*(2), 343–372. https://doi.org/10.1007/S11219-015-9292-4