IRSTI 50.05.13

Zhazira Koppagambetova[1], Kamilla Umirserikova[2]
[1]University of Warwick, Coventry, United Kingdom
[2]Suleyman Demirel University, Kaskelen, Kazakhstan

TDD: WHEN NEEDED, AND MOST IMPORTANTLY, WHEN NOT

Abstract. Software testing is a procedure that allows to confirm or deny the performance of the code and the correctness of its work. Thus, testing plays an important role in software development, because the quality of the code and productivity depend on the choice of the testing method. Unit testing is the most common testing method that aims to test each piece of code. Among the various approaches to unit testing, Test-Driven Development (TDD) and Test Last Development (TLD) are the most common. Therefore, this paper discusses 2 main methods of software development: TDD and TLD. This work is aimed to analyze the effectiveness of TDD as opposed to TLD. As a result, the effectiveness of TDD slightly exceeds TLD. Granularity, uniformity, and refactoring showed positive key results in terms of code quality and performance, while sequencing did not significantly affect any of these factors. Thus, during software development, it is worth applying separate TDD processes, such as granularity, uniformity, and refactoring.

Keywords: Unit Test, Test-Driven Development, Test Last Development, External quality, Developer Productivity.

***

Аңдатпа. Бағдарламалық жасақтаманы тестілеу-бұл кодтың жұмысын және оның дұрыс жұмыс істеуін растау немесе жоққа шығару процедурасы. Осылайша, тестілеу бағдарламалық жасақтаманы әзірлеуде маңызды рөл атқарады, өйткені кодтың сапасы мен өнімділігі тестілеу әдісін таңдауға байланысты. Блокты тестілеу-бұл ең көп таралған тестілеу әдісі, оның мақсаты кодтың әр бөлігін тексеру болып табылады. Блокты тестілеудің әртүрлі тәсілдерінің ішінде ең көп тарағандары-тестілеуге негізделген даму (TDD) және тестілеуге негізделген даму (TLD). Сондықтан, бұл мақалада бағдарламалық жасақтаманы әзірлеудің 2 негізгі әдісі қарастырылады: TDD және TLD. Бұл жұмыс TLD-ге қарағанда TDD тиімділігін талдауға бағытталған. Нәтижесінде TDD тиімділігі TLD-ден сәл асады. Егжей-тегжейлі, біркелкілік және рефакторинг кодтың сапасы мен өнімділігі тұрғысынан оң нәтиже көрсетті, ал реттілік осы факторлардың ешқайсысына айтарлықтай әсер етпеді. Осылайша,

бағдарламалық жасақтаманы әзірлеу кезінде егжей-тегжейлі, біркелкілік және рефакторинг сияқты жеке TDD процестерін қолданған жөн.

Түйін сөздер: блокты тестілеу, тестілеуге негізделген даму, соңғы даму, сыртқы сапа, әзірлеушінің өнімділігі.

***

Аннотация. Тестирование программного обеспечения – это процедура, позволяющая подтвердить или опровергнуть работоспособность кода и правильность его работы. Таким образом, тестирование играет важную роль в разработке программного обеспечения, поскольку качество кода и производительность зависят от выбора метода тестирования. Модульное тестирование – это наиболее распространенный метод тестирования, целью которого является проверка каждого фрагмента кода. Среди различных подходов к модульному тестированию наиболее распространенными являются разработка на основе тестирования (TDD) и последняя разработка на основе тестирования (TLD). Поэтому в данной статье рассматриваются 2 основных метода разработки программного обеспечения: TDD и TLD. Эта работа направлена на анализ эффективности TDD в отличие от TLD. В результате эффективность TDD немного превышает TLD. Детализация, единообразие и рефакторинг показали положительные ключевые результаты с точки зрения качества и производительности кода, в то время как последовательность не оказала существенного влияния ни на один из этих факторов. Таким образом, при разработке программного обеспечения стоит применять отдельные процессы TDD, такие как детализация, единообразие и рефакторинг.

Ключевые слова: модульное тестирование, разработка основанная на тестировании, последняя разработка, внешнее качество, производительность разработчика.

I. Introduction

The importance of testing in software development is undeniable, as it contributes to improving the reliability, quality, and performance of the software[1]. It also allows the developer to check whether the software is working correctly and make sure that it is doing what it is intended to do[2]. Thus, nowadays there is a wide variety of testing techniques[3], so the question of choosing an effective testing method remains relevant at all times.

One of the important aspects of software testing is the ability to fix software errors (bugs) at the initial stage of development, which reduces the risk of defects in the final product[3]. Consequently, the earlier the process starts, the earlier bugs are detected, and the lower the cost of fixing them [4].Therefore, among the various types of software testing, unit testing is the most common, since it is

aimed at testing small parts of the code, such as functions or methods of a class[5]. Unit testing also has various approaches, among which the most fundamental are Test-Driven Development (TDD) and Test Last Development (TLD) [5].

As a result, the purpose of this literature review was to evaluate the effectiveness of the application of Test-Driven Development, as well as analyzing and determining its benefits and challenges.

This paper has the following structure: Section 2 presents general insight into Unit Testing, Test-Driven Development (TDD) and Test Last Development (TLD). Also, a comparison of TDD and TLD is reflected in this section. The review of various research on the effectiveness of the TDD is reported in Section 3. Section 4 concludes the paper.

2. Software Development Techniques

This literature review is focused on the software development methods, therefore, in the following sections, definitions and the basic principles of Unit Tests, Test-Driven Development (TDD), and Test-Last Development (TLD) are given, the main differences of which are the rigidity of the structuring and the sequence of execution [5].

2.1 Unit testing

The importance of providing quality code and improving development efficiency increases as the scale and complexity of software increases [6], consequently one of the most important stages of software development is software testing [1]. The main task of software testing is to find defects in the software in the early stages for the purpose of decreasing software development costs and increasing the reliability of software [4]. Finding the software defects means correct and fast identification of the root of the error, and this can be achieved through unit testing [7]. Therefore, unit testing with the ability to check the correctness of a single unit of functional code is one of the main approaches to software testing [5].

Writing tests for each individual function or method is the main concept of unit testing. Testing is carried out with high granularity, since small parts of the system are tested, rather than the entire system as a whole [4]. This allows to quickly check whether the next code change has led to regression [7], that is, to the appearance of errors in already tested places in the program, and also makes it easier to detect and eliminate such errors [8]. Among the various applications of unit tests, Test-Driven Development and Test Last Development are two of the most basic [5].

2.2 Test-Driven Development (TDD)

Test-Driven Development is a software development strategy based on repeated short development cycles [9]. The essence of TDD is that first a test is written that covers the desired changes. Next, a program code is written that executes the desired behavior of the system and allows the test to pass. After that, the written code is refactored with constant testing of passing tests [8]. In

other words, the philosophy of TDD is that tests are a specification of how the program should behave. This method is also known as the red-greenrefactor cycle [4], which was proposed by Kent Beck in 2003.

TDD application involves the following three steps:

1. Red: writing a failing test for a small piece of functionality.

2. Green: implementation of the functionality that successfullypasses the test.

3. Refactor: refactoring old and new code to keep it in a wellstructured and readable state.

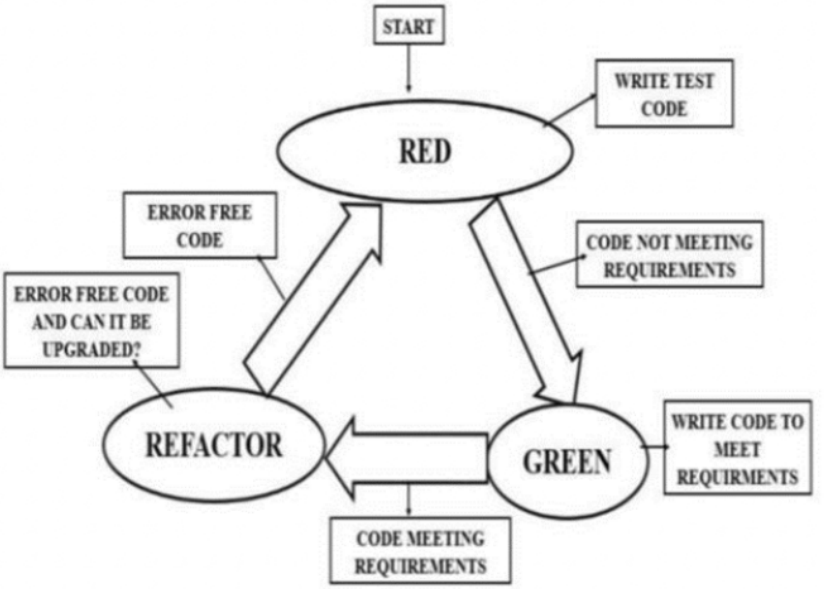

Figure 1: Test-Driven Development Red Green Refactor Cycle [4].

2.3 Test Last Development (TLD)

In contrast to Test Driven Development, developers follow the classic method of software development, specifically Test Last Development (TLD) [10]. At the stage of requirements analysis, based on which the code should be developed, there is clear documentation of all the requirements, functionality, and states [4]. After the software plan (technical task) for all steps of the software development process is approved, the code

Figure 2: Flowchart of Traditional Testing [4].

is developed and, after the developer is confident that he has fulfilled all the technical conditions, tests are written to check whether the code meets all the requirements [9].

2.4 Comparing TDD and TLD

Micro-iterative test-coding cycles define the TDD's fine-grained approach [9] - in contrast to traditional approaches, where " systems are generally built up-front and then tested", define it as a coarse-grained approach [11]. The traditional method of development (TLD) involves writing test codes at the last step of software development, that is, after writing the production code [11, 12]. As opposed to TLD, in TDD first of all, a small test code is written to check the required functionality. At this stage, the test will fail because the functioning code has not been written yet, but at the further stages of development, this test will be used to check the correctness of the code. Developing the code itself is the third step. To check whether the existing functionality of the code corresponds to the desired one, the former one is tested for all cases. An indicator that the existing code is working correctly is the successful passing of the code on all tests [4,5,8].

The next important step in TDD is code refactoring, that is, modifying the code in various ways, such as removing duplicates or improving the design structure, the main goal of which is the absence of behavioral changes in the software [9, 12]. Accordingly, refactoring provides high-quality and easily readable code with a well-designed structure that allows the software developers to make new changes in the code without breaking and changing the functional requirements of the final product [8].

It is necessary to continue running the existing code for all tests until there is no need to change the functionality of the code. Then and only then, refactoring should be stopped [4]. Refactoring is an additional, but at the same

time an important stage in software development, however, this stage is skipped in the traditional method of development (TLD) [12].

Thus, the TLD method assumes the presence of a program plan, thought out in advance for each stage of development, which is consistently implemented with testing at the last stage [10]. Whereas TDD obeys a more flexible and iterative development approach, where continuous code development, refactoring, and testing are observed [4].

3. Literature review

This section highlights the main findings of the authors of various studies on the application of TDD in practice.

In their research work, Yogesh and Vimala (2020) reflected the main aspects of TDD by identifying methods and internal processes and comparing it with the traditional testing method. Thus, the authors of the study concluded that the implementation of TDD in practice was effective since TDD improved software performance by writing test codes that increase test coverage as well as reduce errors. Yogesh and Viamala noted that the primary writing of test code in TDD, followed by writing production code, contributed to a deeper logical understanding of the functionality of the code. Thus, authors pointed out the following advantages of using TDD: the absence of ambiguity in the code due to writing test codes, the ability to make changes in the code at any time during software development without breaking or changing the behavior of the final product, and a sharp decrease in the number of errors. One of the drawbacks that were mentioned by Yogesh and Vimala was the complexity of the application of TDD at the initial use.

Difficulties in adapting and applying TDD were also observed by Santos et al. (2018), whose research work was based on an analysis of experimental tasks performed by developers and a survey at the end of the experiment. The experimental task consisted of three levels, namely:

'Bowling-Score Keeper', 'Mars-Rover' and 'Spread-Sheet', wherein each of the levels developers needed to apply the TDD, adhering to all sequential development steps, as well as the traditional development method. Thus, developers with different skill levels were randomly divided into 3 groups to perform an experiment. In an analytical analysis, researchers first provided descriptive statistics (mean, standard deviation, and median) of the traditional method and the TDD. Then they analyzed the experimental items using a Linear Mixed Model. As a result, they concluded that the effectiveness of the application of the traditional method and TDD was the same.

Unlike Yogesh and Vimala, Fucci et al. (2017) studied and analyzed the impact of individual unique TDD processes such as sequencing, granularity, uniformity (order, length, variation), and refactoring effort on the productivity and quality of code, rather than the entire TDD technique. The analysis was carried out based on 82 data points provided by 39 professionals, each of which indicated the process used by the professionals while performing specific

development tasks. Fucci et.al. provided the following observations based on the results of the regression analysis performed: granularity and uniformity had a positive effect on improving code quality and productivity in general. While sequencing the order in which the test and production code was written did not have a significant impact. And refactoring, in turn, negatively affected both quality and productivity. Thus, the claimed merits of TDD were due to its fine-grained approach, which allows monitoring of the sequential development flow, rather than its distinctive first test dynamics.

Moe and Oo (2020) also used regression analysis to assess the effectiveness of TDD. In their research work, regression analysis was used to analyze the relationship between 2 dependent variables, namely QLTY (Quality) and PROD (Productivity), and 1 independent variable, TEST. QLTY denoted the percentage of passed tests for implemented solutions to problems, and PROD was defined as the percentage of solutions completed tasks. According to the analysis, QLTY and TEST showed a positive trend, which was characterized in the relationship between the number of tests and the quality of the external code. In addition, in the relationship between TEST and PROD, there was a certain negative relationship, which was characterized by a certain decline in developer productivity when using TDD methods. Thus, the researchers concluded that, in contrast to the traditional method, TDD took 16% more time, and the number of writing test codes increased by 52%. This affected the productivity of the developer and the quality of the written code. Researchers agreed with the generally accepted opinion that TDD helps to reduce defects in functional code, which positively affects the quality of the code, while they did not exclude some decline in developer productivity.

Papis et al. (2020) also aimed to assess the impact of development methods (TDD and TLD) on code and testing quality. In a threeweek experiment, 19 participants of different levels participated in solving various blocks of tasks using TDD and TLD. The researchers chose Linear Models (LMM - Linear Mixed Model) as an analysis tool. Based on the tasks performed by the developers, the researchers presented the following observations: TDD showed 1.8 fewer errors than TLDs and test quality was 5% higher for TDD than for TLD. As with previous studies, they noted that TDD was difficult to use, especially for beginners. Hence, TLD rules were easier to follow than TDDs.

Karac and Turhan (2018), in their work, "What We (Really) Know About Test-Driven Development," questioned the effectiveness of TDD by analyzing various studies. They argued that there was not strong enough evidence that TDD was better than another development method. Thus, they emphasized that while choosing a development method, it was necessary to proceed from the expected results. Also, the authors of the work noted that working with short cycles of code, namely with small, clearly formulated tasks, had a positive effect on productivity than the order of test execution.

Teschner (2020), as well as Karac and Turhan, strongly agreed that "simply writing tests for the sake of it will not automatically increase the value of production code. Tests should concentrate on the business logic and expected behavior". In the research carried out by Teschner, code coverage was traced by developing a project in C ++, based on the meson assembly system, the built-in functionality of which allows to support software testing. The project was based on a library of linear algebra for solving a linear system of equations using the iterative Conjugate Gradient (CG - Conjugate Gradient) method. As a result, he claimed that with the use of TDD processes, code coverage was increased, as well as code quality was improved. Thus, Teschner confirmed that by using TDD processes, the probability of missing errors by mistake was minimal, and the software met the stated requirements. Like other researchers, Teschner found that personal aptitude was a major barrier when writing test codes for production code. Hence, he concluded that writing effective test code primarily depended on the developer's level of competence. Teschner disagreed with the fact that the increase in development time was a disadvantage of the TDD method, on the contrary, he argued that the increased time was compensated by the decrease in the time required for debugging.

4. Conclusion

Analyzing the effectiveness of the use of TDD as opposed to TLD, it appeared that TDD contributes to improving the quality of software and increasing customer satisfaction. The latter is due to the extended coverage of tests when using TDD, the developer can be sure that the software works as expected, meets all the requirements, and also reduces the possibility of making mistakes in the functional code. It should be noted that using TDD, in contrast to the traditional development method, development time increases, since TDD implies writing test code for each separate module of the program code. However, the increased time is compensated by the time spent on debugging diagnostics.

Observing the positive effect of using TDD, it is worth noting that the effectiveness of its use does not greatly exceed the traditional development method. The distinctive results in terms of code quality and performance are granularity, uniformity, and refactoring, while sequencing does not affect any of these factors. Based on the analyzes of various researchers, it can be concluded that the efficiency of TDD is explained by the approach to the task, namely, finegrained, stable steps, and not by the order of execution of individual TDD processes. Therefore, it is advised for developers to focus on dividing tasks into as small and uniform steps as possible. Thus, maximum efficiency is achieved during the performance of individual processes such as granularity, uniformity, and code refactoring, rather than the entire TDD method.

References

1 Borle, N., Feghhi, M., Stroulia, E., Grenier, R. and Hindle, A. (2018), [journal first] analyzing the effects of test driven development in github, in '2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)', pp. 1062–1062.

2 Alkawaz, M. H. and Silvarajoo, A. (2019), A survey on test case prioritization and optimization techniques in software regression testing, in '2019 IEEE 7th Conference on Systems, Process and Control (ICSPC)', pp. 59–64.

3 Umar, M. A. (2020), 'Comprehensive study of software testing: Categories, levels, techniques, and types'.

4 Yogesh, T. and Vimala, P. (2020), Test-driven development of automotive software functionality, in '2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)', pp. 1162–1165.

5 Papis, B. K., Grochowski, K., Subzda, K. and Sijko, K. (2020), 'Experimental evaluation of test-driven development with interns working on a real industrial project', IEEE Transactions on Software Engineering pp. 1–1.

6 Sun, B., Shao, Y. and Chen, C. (2019), Study on the automated unit testing solution on the linux platform, in '2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)', pp. 358–361.

7 Laghari, G. and Demeyer, S. (2018), Poster: Unit tests and component tests do make a difference on fault localisation effectiveness, in '2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)', pp. 280–281.

8 Kampmann, A. and Zeller, A. (2019), Carving parameterized unit tests, in '2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)', pp. 248–249.

9 Moe, M. M. and Oo, K. K. (2020), Consequences of dependent and independent variables based on acceptance test suite metric using test driven development approach, in '2020 IEEE Conference on Computer Applications(ICCA)', pp. 1–6.

10 Nanthaamornphong, A. and Carver, J. C. (2018), 'Test-driven development in hpc science: A case study', Computing in Science Engineering 20(5), 98–113.

11 Teschner, T.-R. (2020), 'A practical guide towards agile test-driven development for scientific software projects'.

12 Santos, A., Spisak, J., Oivo, M. and Juristo, N. (2018), Improving development practices through experimentation: An industrial tdd case, in '2018 25th Asia-Pacific Software Engineering Conference (APSEC)', pp. 465–473.

13 Fucci, D., Erdogmus, H., Turhan, B., Oivo, M. and Juristo, N. (2017), 'A dissection of the test-driven development process: Does it really matter to test-first or to test-last?', IEEE Transactions on Software Engineering 43(7), 597–614.

14 Karac, I. and Turhan, B. (2018), 'What do we (really) know about test-driven development?', IEEE Software 35(4), 81–85.