International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

http://ijpsat.ijsht-journals.org
Vol. 13 No. 1 February 2019, pp. 81-88

Usefulness and Effectiveness of Test-First Development and Junit Framework

Myint Myint Moe

University of Computer Studies (Hpa-an),
Kayin State, Myanmar

***Abstract* - Test-First development (TFD) 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. Test-First development is a software development practice where small sections of test code are used to direct the development of program units. Writing test code prior to the production code promises several positive effects on the development process itself and on associated products and processes as well. Thus, it is difficult to assess the potential process and product effects when applying test-First development. Test-First development can reduce the amount of introduced defects and lead to more maintainable code. Parts of the implemented code may also be somewhat smaller in size and complexity. While maintenance of test-First code can take less time, initial development may last longer. TFD defines a proven way to ensure effective unit testing.**

Keywords - TFD, Refactoring, Duplicating, JUnit.

I. INTRODUCTION

Test-First Development is a practice that can make batter programs. Test-First Development is a core part of the agile process called extreme Programming (XP) [2]. TFD is a software development technique in which tests are developed before the code, in short and incremental cycles. Test-First Development is known as Test-Driven Development. This technique proposes for the developer to create a new flawed test, and then to implement a little piece of code, in order to satisfy the current test set. Then, the code is refactored if necessary, to provide a better structure and architecture for the current solution executed, the code can be refactored so that its internal structure can be continuously evolved and improved. The tests help to verify if the behavior has not been modified during refactoring. This cycle is performed repeatedly until the tests added verify scenarios for all expected class requirements. TFD technique widely uses in industry.

In test-first software development processes the test cases are specified incrementally before the production code. Testing is taken into account from the initial development phases and it drives the full development process. It is based on iteration between writing unit tests and writing functional code. It involves working in very small steps, one small unit test at a time Before writing functional code, the developers write automated unit test cases for the new functionality they are about to implement. They write the functional code to pass these test cases. In other words, every line of new functional code is written in response to a unit test. New functionalities are not considered as proper implementations unless the new unit test cases and all the prior tests are passed. This technique was used for a long time but only in the last years it has been adopted as a key strategy in agile software development [5]. JUnit framework and its extension points JUnit is a Java open-source framework for the creation of unit tests. Its purpose is to be a basis for the creation of test automation code. It is used for the practice of Test-Last Development, and its model has been taken into account to create test frameworks for other languages. Some of the main features of these frameworks are the execution of test cases and the presentation of the execution results [4].

Corresponding Author: Myint Myint Moe

II. OBJECTIVES

Test-First Development is a process of developing and running automated test before actual development of the application. TFD starts with designing and developing tests for every small functionality of an application. In TFD approach, first, **the** test is developed which specifies and validates what the code will do. The simple concept of TFD is to write and correct the failed tests before writing new code (before development). This helps to avoid duplication of code as developers write a small amount of code at a time in order to pass tests.

The goal of TFD is not pure testing but instead to form the specifications necessary for development before attempting to develop **code**. Because the tests are the specifications, documentation is a secondary consideration of this approach. While studies have shown test-first development to have little impact on code coverage or mutation score, test-first development may reduce computational complexity and provide other design benefits. One of the key problems with this approach is that the goal is passing all the tests instead of having the desired behavior. Ideally the tests should reflect the desired behavior but this is not always the case. Tests are best for capturing low level desired behavior but fail to paint a larger picture for the program as a whole [6]. Unit tests are the center of TFD but basic unit testing and TFD are not the same concept. Unit testing verifies a behavior of a unit separately from other units in a code. With unit testing, developer tests single units, or methods, their code with a test program that gives input to each unit, or method, and checks if it returns the expected output. It is a common misunderstanding that using TFD, all tests are written before all the code. In TFD, the unit tests are written before any code fragment. It supports that the code is written correctly and how the code to be designed. Therefore, TFD generates a condition which allows the tests to drive the development while unit testing simply verifies a behavior of a unit [7].

III. RELATED WORK

In [7], the author proposed on Using Test-Driven Development to Improve Software Development Practices. This research study presents the results from seven semi-structured interviews with practitioners in Iceland working in the software industry who has adopted TDD in their projects. The interviews focused on the participants' point of view and experience of using TDD in software development. The data analysis of the interviews will be described along with a conclusion on how TDD can improve software development. The results from this research show that TDD has the ability to help practitioners throughout the software **development** implementation phase in many ways. Other studies on TDD were analyzed with five factors in mind and their outcomes compared to this research which showed very similar results. The main advantages of using TDD can be outlined as: improved code quality, less defects, easier maintenance, safety net and reliable software.

In [8], the authors described on "A Longitudinal Cohort Study on the Detainment of Test-Driven Development"; the authors conducted a (quantitative) longitudinal cohort study **with** 30 third-year undergraduate students in Computer Science at the University of Bari in Italy. Results: The use of TDD has a statistically significant neither on the external quality of software products nor on the developers' productivity. However, we observed that participants using TDD produced significantly more tests than those applying a non-TDD development process, and that the detainment of TDD is particularly noticeable in the amount of tests written.

IV. METHODOLOGY

Test-First Development (TFD) is a software development process that relies on the repetition of a very short development cycle: **requirements** are turned into very specific test cases, and then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements [10]. In Test-First Development, the developer writes automated unit tests for the new functionality, they are about to implement. It is a software engineering process that follows small development cycle. The automated unit tests for any application can be written in two ways: Before code implementation and after code implementation. If unit tests are written before any code implementation then it is known as Test-Driven Development (or) Test-First Development. If Unit tests are written after the code implementation then it is known as Test after Development (or) Test-Last Development [9].

4.1 Test-First Development Cycle

Test first development model cycle consists of:

1. Writing a Test: A unit test (manual or automated, preferably automated) is first written to exercise the functionality that is targeted for development. Before writing the **test**, the developer is responsible for understanding the requirements well. A unit test shall also

contain assertions to confirm the pass/fail criteria of the unit test.

2. Run to fail /make it compile: Since the feature is yet to be implemented, the unit test that was written in Step 1 is bound to fail. This step is essentially a validation step for the unit test written, as the test shouldn't pass even if there is no code written for it. Often unit tests are automated and there are chances that the tests fail because of syntax or compilation errors. Sanitization of the tests by removing these errors is also an essential part of this step.

3. Implementing the (complete /partial) functionality: This step involves developing the part of the functionality for which the unit test is written and will be validated.

4. Making tests to pass: Once the unit tests for the developed code have passed, the developer derives confidence that the code fulfills the requirements.

5. Code refactoring: The unit tests might have passed, but code refactoring may still be required for reasons including handling errors elegantly, reporting the results in the required format, or carving a subroutine out of the written code for re-usability.

6. Repeating the cycle: The unit test/set of unit tests is/are refactored to cater to new functionality or push towards completion of the functionality.

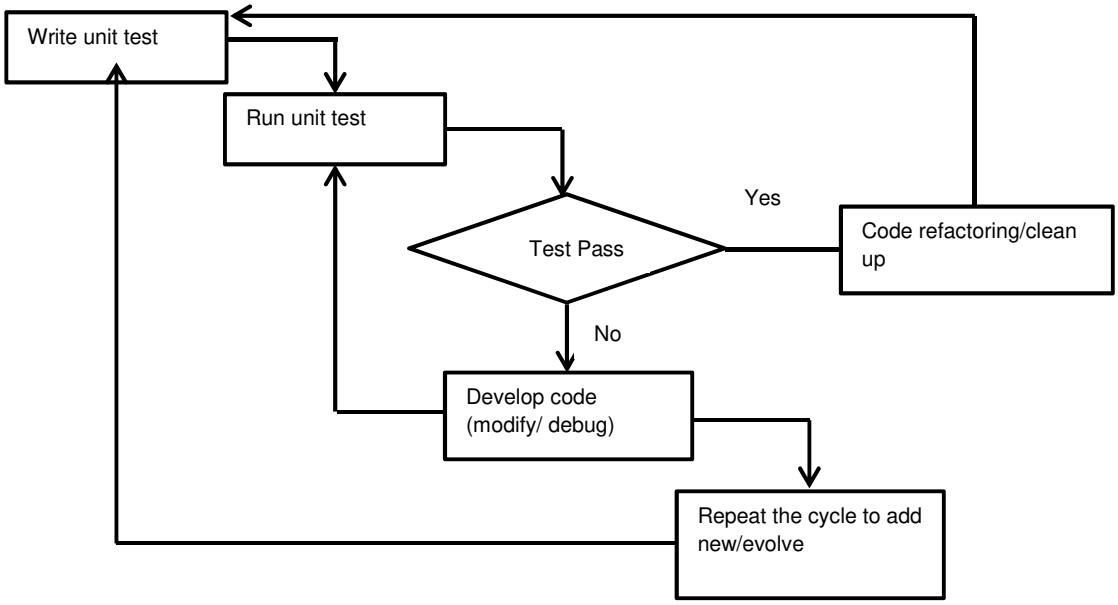


Figure 1: Test-First Development cycle

4.2 Write Test

Tests in TFD are somewhat like unit tests with the difference that they are written for behaviors, not for methods. It is important that tests are written so that they are order independent, i.e. the result remains the same regardless of the sequence in which the tests are run. When writing the tests, it should be kept in mind that the tests should concentrate on testing the true behaviors, i.e. if a system has to handle multiple inputs, the tests should reflect multiple inputs.

4.3 Run Test

The test is run to verify if test written is useful or not. In case, test case passes, this indicates that test is worthless. This also validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code. The new test should also fail for the expected reason. This increases confidence that it is testing the right thing, and will pass only in intended cases.

4.4 Write Code

In TFD, the code writing is actually a process for making the test work, i.e. writing the code that passes the test. TFD cycle may be employed, for example, by replacing the

return value of some method with a constant. It provides a quick way to make the test pass. It has an effect while giving the programmer confidence to proceed with refactoring and it also takes care of the scope control by starting from one concrete example and then generalizing from there. The abstract implementation is driven through sensing the duplication between the test and the code. Implementation can give a push towards the right solution, if the programmer really does not know where to begin to write the code. After reaching the abstract implementation, the other assertion becomes redundant with the first one and it should be eliminated. The obvious implementation is used when the programmer is confident and knows for sure how to implement some operation. Constant practicing of "obvious implementation" can be exhaustive, since it requires constant perfection. When the tests start to fail consecutively, it is recommended to practice until confidence returns [9].

4.5 Refactor

Refactoring is a process of improving the internal structure by editing the existing working code, without changing its external behavior. It is essential, because the design integrity of software scatters over time due to the accumulated pressure of modifications, enhancements and bug fixes. Now the code can be cleaned up as necessary. The idea of refactoring is to carry out the modifications as a series of small steps without introducing new defects into to the system. By re-running the test cases, the developer can be confident that code refactoring is not damaging any existing functionality. This research would be doing comparative study of Test-first development with traditional techniques and discussing pros and cons of these techniques. This study will reveal factors that encourage the use of Test First Development in industry and would try to find the weaknesses of TFD that limits their use in industry.

4.6 Duplication

Duplicated code in its various forms is the death of good code. Duplication is such a problem that several people have warned against it at length. If developers detest duplicated code, developers need to look closely to detect what it's doing. It's likely that there is something useful or important being done. Developers need to put it in one place. Developers can do this by extracting duplicate code into separate methods that can then be called from multiple locations. If the duplication is in an inheritance hierarchy, Developer may be able to push the duplicated code up the hierarchy. If the structure of some code is duplicated but not the details, Developers can extract the differing parts and make a template method of the common structure. Some cases of duplication will be simple, others won't be. Some will be obvious, but some will be very subtle. An example of a simple case of duplication is an expression (or sequence of expressions) that appears in multiple methods. In this case, it is simply a matter of moving the duplicated expression(s) into a separate method and replacing the original occurrences with calls to the new method [2].

4.7 Solve Problems

Test-First Development solves all of these problems: The programmer does the testing, working with tests while the code is freshly in mind. In fact, the code is based on the tests, which guarantees testability, helps ensure exhaustive test coverage, and keeps the code and tests in sync. All tests are automated. They are run quite frequently and identically each time. Exhaustive test coverage means that if a bug is introduced during debugging, the test scaffolding finds it immediately and pinpoints its location. And the test-debug cycle is kept quite short: there are no lengthy delays between the discovery of a bug and its repair. Finally, when the system is delivered, the exhaustive test scaffolding is delivered with it, making future changes and extensions to it easier [11].

In software industry, testing is the process of confirming the application works as it is expected to be. Test Last Development (TLD) and Test First Development (TFD) are two major testing processes. In TLD, testing is done at end after coding whereas in TFD, testing is done before writing codes. Both of them have their own strengths and weakness. One doesn't surpass other in all aspects. The objective of the article is to present the strengths and weakness of these processes and help understand both in order to look to opt for better alternative according to the need and situation.

4.8 Strengths and Weakness of TFD

TFD has strengths and weakness. Due to their contrasting nature, one's strength is another weakness. Their strengths and weakness are below:

4.8.1 Time Required

Development time is relatively high in TFD. It nearly takes as much as more time. This is because of its iterative process between testing, coding, refactoring. Developers have to constantly move to and from between code and test cases. They cannot just focus on writing code and then testing but has to focus on writing failing test first and just enough code to pass the test.

4.8.2 Learning Curve

TFD has a learning curve. It is not just a testing technique but a process of test, code and then refactor in forward sequence. Test before coding forces developers to think of test cases before implementation, which they are unfamiliar to. This change requires a lot of practice and discipline. Moreover, refactoring comes up with good design principles like use of interfaces, design patterns, abstraction etc. These all comes up with experience, learning and lot of practice. Developers study and practice during courses in university. It also doesn't demand refactoring. The task of refactoring is upon developers' discretion.

4.8.3 Maintenance Cost and Productivity

TFD decreases the maintenance cost and overall increases the productivity. Maintenance becomes cheaper and easier if the products are highly stable and reliable. Its inseparable relationship between testing and coding recursively is the result of highly reliable and stable product. All codes are vigorously tested during development. This also helps in bug fixing, further development and maintenance by making sure no new bugs are inserted during the process.

4.8.4 Code Size

The inclusion of many more test cases in TFD increases the size of the code. But it can be also argued that as TFD only includes the minimum amount of code that are required for the product it has the smallest size possible and test cases are only used during development and not in production environment resulting into much cleaner and better code.

4.8.5 Code Change

Code changes are common in software development and during maintenance. With TFD, one can ensure that new changes don't impact the product in an unwanted way. This is by repeated continuous running test cases. Additionally, test cases also act as a good starting point for debugging, which easies change.

4.8.6 Code Simplicity

TFD is relatively complex. Complexity is mainly due to the use of various design principles like interface, design pattern, etc.

4.8.7 Unit Testing Tool of TFD: Junit

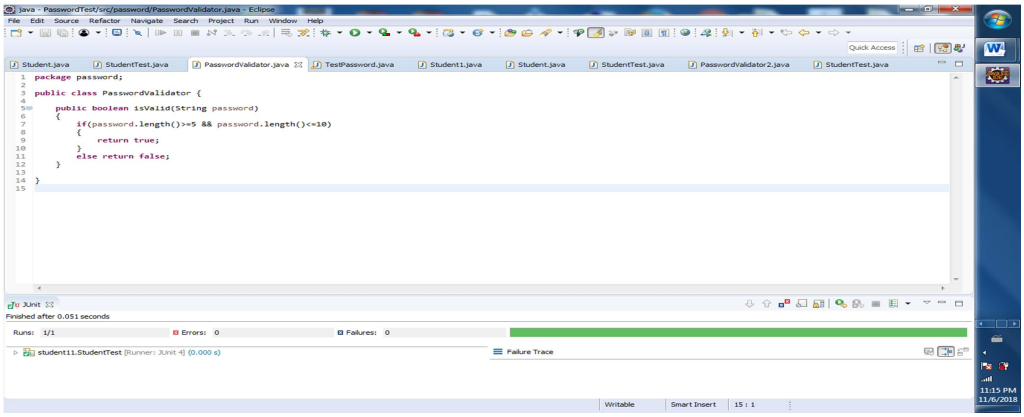
JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development and is one of a family of unit testing frameworks, which is collectively known as xUnit that originated with SUnit. JUnit is mostly used by developers. JUnit is designed for unit testing, which is really a coding process, not a testing process. But many testers or QA engineers are also required to use JUnit for unit testing. Writing more tests will make more productive, not less productive. Tests should be done as soon as possible at the code unit level. JUnit makes unit testing easier and faster. JUnit classes are important classes, used in writing and testing JUnits. Some of the important classes are Assert, TestCase and TestResult. Assert contains a set of assert methods. Test Case contains a test case that defines the fixture to run multiple tests.

TestResult contains methods to collect the results of executing a test case. JUnit TestCase is the base class in JUnit framework. A test case defines the fixture to run multiple tests. To define a test case: Implement a subclass of TestCase; Define instance variables that store the state of the fixture; Initialize the fixture state by overriding setup; Clean-up after a test by overriding teardown. Each test runs in its own fixture so there can be no side effects among test runs. JUnit TestSuite is a container class in JUnit framework. Test Suite allows grouping multiple test cases into a collection and running them together. TestSuite class is no longer supported in JUnit. To run only the selected test, position the cursor on the test method name and use the shortcut. To see the result of a JUnit test, Eclipse uses the JUnit view which shows the results of the tests. Select individual unit tests in this view, right-click on them and select Run to execute them again. [11]

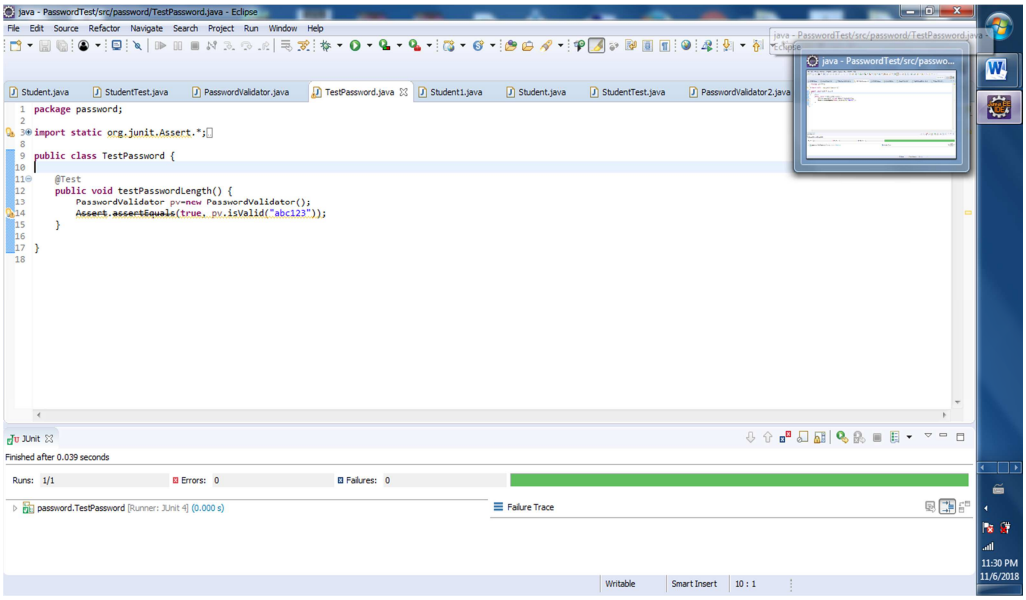
4.9 Example of TFD

A condition for Password acceptance: The password should be between 5 to 10 characters.

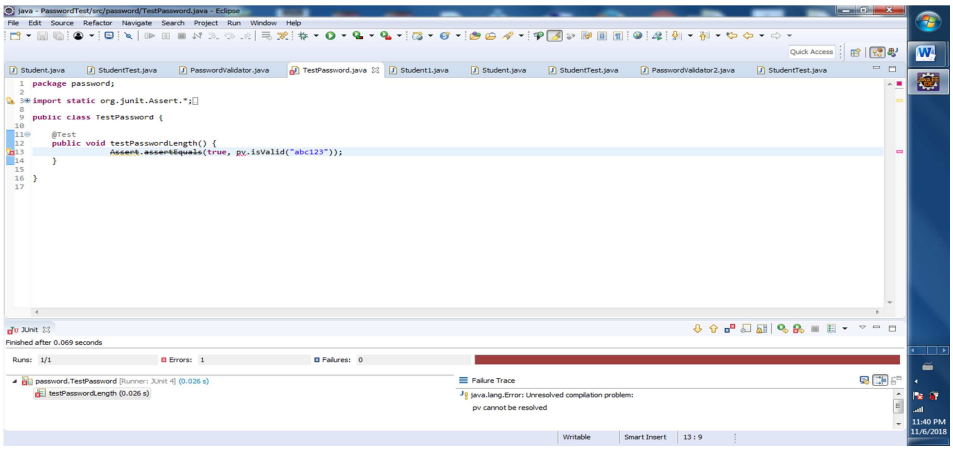
1. Create class Password Validator



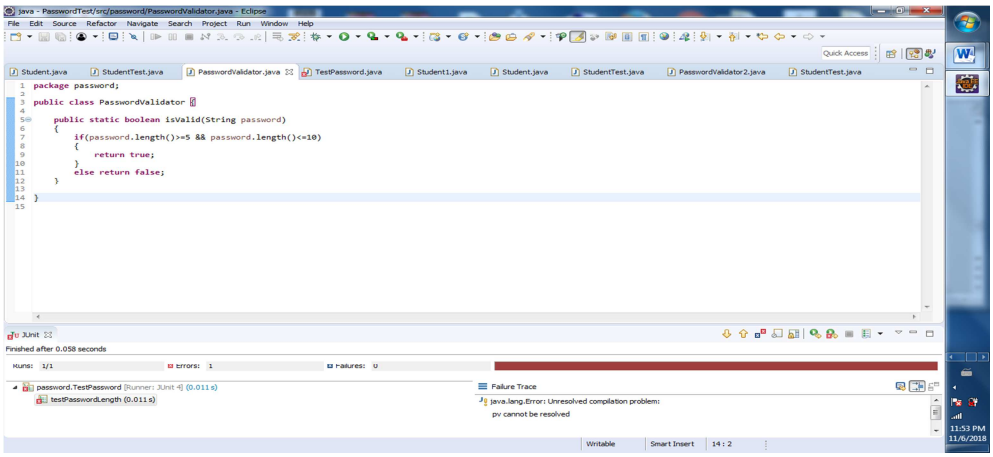
2. Write the code that fulfills all requirements using JUnit framework.



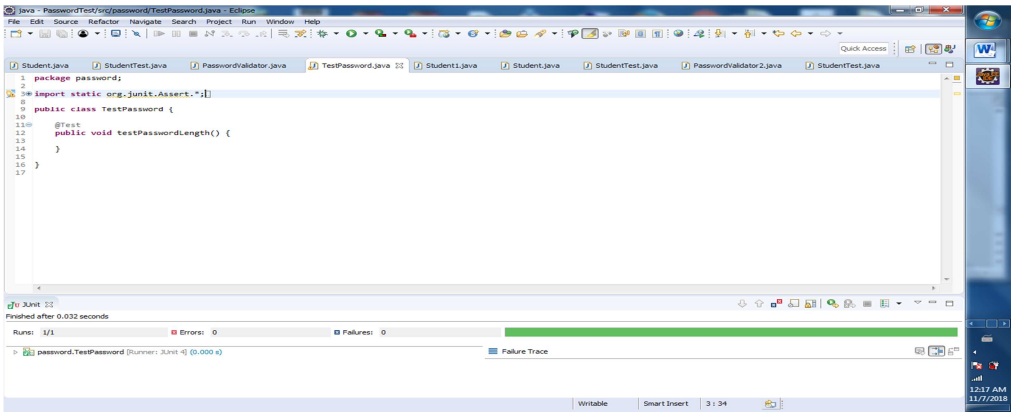
3. To do refactor (change code), remove class PasswordValidatorpv= new Password Validator()



4. Need to change this method by adding “static” word before Boolean as public static Boolean isValid (String password). Refactoring Class PasswordValidator () to remove above error to pass the test.



5. Using JUnit framework, add “static” word before Boolean as public static Boolean isValid (String password). Refactoring Class PasswordValidator () to remove above error to pass the test. So refactor code as there is no need of creating instance of class "Assert.assertEquals (true, PasswordValidator. IsValid (“Abc123”)”.



V. CONCLUSION

TFD applies testing and analysis to the test suites generative by tool to assess their fault detection effectiveness. It compares that effectiveness with those of the test suites generated by other comparable tools. It compares the effort and time spent in using those tools. Also, the execution time of tool to automatically generate test case code is very fast. This system compares the associated efficiency to manual coding when tool is used in large-scale projects. TFD is a development approach that drives the design of a code and supports developers in their work with confidence, and more reliable software. TFD has given the developers deeper logical understanding of their code and has supported them to improve their development skills. TFD counts the bugs and defects over a period of time. TFD approach enables thorough unit testing which improves the quality of the software and enhance customer satisfaction. TFD is an effective tool to improve productivity in long run and can significantly reduce the defect density of developed software either immediately or in the long run. This research will hopefully guide some software companies, managers and developers to understand the benefits and drawbacks of TFD and how to make a decision on whether or not to adopt TFD.

REFERENCE

[1] Viktor Farcic, Alex Garcia; Test-Driven Java Development; Invoke TDD principles for end-to-end application development with Java; Copyright © 2015 Packt Publishing; First published: August 2015; Production reference: 1240815; Published by Packet Publishing Ltd. ISBN 978-1-78398-742-9; www.packtpub.com

[2] By David Astels; Publisher: Prentice Hall PTR; Test-Driven Development: A Practical Guide; Pub Date: July 02, 2003; ISBN: 0-13-101649-0

[3] Jürgen Münch, Simo Makinen; "Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies"; © Springer 2014; Proceedings of the 6th International Conference Software Quality Days (SWQD 2014), Vienna,Austria, January 14-16, 2014.

[4] André A. S. Ivo1,2* , Eduardo M. Guerra2, Sandy M. Porto2, Joelma Choma2 and Marcos G. Quiles3; An approach for applying Test-Driven Development (TDD) in the development of randomized algorithms; Ivo et al. Journal of Software Engineering Research and Development (2018) 6:9; https://doi.org/s40411-018-0053-5; © The Author(s). 2018 Open Access

[5] Roberto Latorre, "A successful application of a Test-Driven Development strategy in the industrial environment"; © Springer Science+Business Media New York 2013

[6] Corrigan Johnson; "SPEST - A TOOL FOR SPECIFICATION-BASED TESTING"; the Faculty of California Polytechnic State University, San Luis Obispo, January 2016

[7] Raquelita Ros Aguilar, "Using Test-Driven Development to Improve Software Development Practices"; School of Computer Science; REYKJAVIK UNIVERSITY, SPRING 2016; T-622-UROP.

[8] Davide Fucci, Simone Romano, Maria Teresa Baldassarre; "A Longitudinal Cohort Study on the Retainment of Test-Driven Development"; ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '18), October 11–12, 2018, Oulu, Finland. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3239235.3240502

[9] Shaweta Kumar, Sanjeev Bansal; "Comparative Study of Test Driven Development with Traditional Techniques"; International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume-3, Issue-1, March 2013 352

[10] Naomi Unkelos-Shpigel (&) and Irit Hadar; "Test First, Code Later: Educating for Test Driven Development, Teaching Case"; Information Systems Department, University of Haifa, Carmel Mountain, 31905 Haifa, Israel.