How Does Test-Driven Development Affect the Quality of Developed Software?

Fatima Sjögren Alpha, Jan Zubac, Cecilia Lindskog, Isabella Gagner
Group G

Abstract—In recent years agile software testing has become widely used, which has increased the usage of the coding practice Test-Driven development (TDD). TDD is a coding practice where the developer writes a unit test for a specific feature before writing the production code for it. This paper evaluates how TDD affects the internal and external quality of the developed software when compared to the code produced using a more traditional Test Last Development (TLD). The internal and external quality correspond to verification and validation of the code, respectively. Studies have shown that projects using TDD often result in improved external quality. The studies conducted in regards to internal quality show contradicting results. There are approximately equally many studies that show positive, negative and neutral impact on internal quality when using TDD. The studies also show that TDD may affect the productivity negatively. There is a significant lack in studies conducted in an industrial setting which makes it difficult to draw conclusions of how it really is in the industry.

I. INTRODUCTION

Nowadays, many software companies have abandoned the traditional development process and transitioned to using a rational unified development process (RUP) and agile software development. Agile software development is a very broad term including areas such as Extreme Programming, Scrum, Feature Driven Development and Test-Driven Development. Agile testing is usually described as testing in short iterations, conducted in one of the previously mentioned areas [1]. This report will focus on the method Test-Driven Development (TDD), which is an agile development and testing style where the test cases are specified before their production code is written.

To be able to evaluate if there is a positive effect of using TDD, the impact on software quality will be discussed. Software quality can be divided into two very broadly defined areas, namely internal and external quality. Internal quality represents verification of the code and external quality represents validation of the code.

There are many studies conducted on the benefits and drawbacks of TDD. This paper aims to compile the results from several of these studies and evaluate the impact that TDD has on external as well as internal software quality.

II. DESCRIPTION

A. Test-Driven Development

This section aims to introduce the reader to how TDD works and to compare it to the traditional test-last development. TDD is a coding practice that is a central practice of Extreme Programming, and is thus an Agile software development method [2].

1) Definition: TDD is a coding practice where the developer writes a unit test for a specific feature before writing the production code for it. TDD relies on continuous regression testing and on tests that steer development activities. This is done in several short iterations [2].

Such an iteration may contain the following steps:
1) Write a unit test.
2) Make the unit test compile (with dummy code).
3) Watch it fail.
4) Make the unit test pass.

The first step is to write a unit test that will cover a specific feature that is going to be included in the product. At this step the unit test cannot compile since there is no production code written for the feature yet. The second step is to create just enough dummy code to make the unit test compile and watch it fail since there still isn’t enough production code written to pass the unit test. The next step is to implement just enough production code to make the unit test pass [2].

The same steps need to be followed when implementing additional features, this time with an additional step: refactoring. For example, the developer needs to refactor the production code and/or the test code when adding another feature that causes the previous unit test to fail.

The first step in this iteration is to write a new unit test for another desired feature. At this step the new unit test cannot compile since there is no production code written for the feature yet. The second step is to make the new unit test compile using dummy code. At this point, the old unit test may not pass since there were changes in the code. Dummy code is written to make the old unit test pass. Since only dummy code has been written, the new unit test will fail (as it should). The next step is to implement just enough production code to make the new unit test pass. If the old unit test fails, the production code needs to be refactored so that it passes again. The developer can move on to the next feature when all the written unit tests pass [2].

1) Write another unit test.
2) Make the new unit test compile (with dummy code).
3) Make the old unit test pass (with dummy code).
4) Watch the new unit test fail.
5) Make the new unit test pass.
6) Refactor.
2) **TDD vs. TLD:** The term "test-driven" can cause confusion. In traditional coding practices that use "test-oriented" development, small chunks of functionality are implemented followed by unit tests being written. This practice is however not the same as TDD [2].

In test-driven development, the tests lead or guide the production activity. A set of tests are written for a set of features that are to be implemented. Ideally in TDD, the cycle is much shorter where only a single test is written followed by implementing just enough functionality to pass the test. The previously written tests need to be supervised as well to make sure that they still pass after the latest implemented feature. This results in testable software where each function can be exercised independently [2]. See Figure [1] for a visualisation of how TDD works.

A traditional test-last method can be said to work in the opposite way - the high level and detailed design guide the code implementation and the tests are written and executed when the code has been written. See Figure [2] for a visualisation of how test-last development works. This test-last method requires a lot of work in the first two processes, since all definitions need to be quite specific for the implementation to start off correctly [3]. Compare this to TDD - when the high level design is specified, test writing is initialised immediately and coding can commence. This method rather allows the detailed design to be elaborated through the test and coding iterations [3].

3) **Misconceptions:** There are some common misconceptions about TDD. David S. Janzen and Hossein Saeidjan have conducted a study where they interviewed participants from different companies who claimed to be using TDD. They discovered a couple of misconceptions. One misconception that they discovered is that TDD is equivalent to automated testing. Another misconception is that TDD is about writing all the tests first before writing any production code [3]. Many people also believe that TDD is the same as plain unit testing. The main reason behind these misconceptions (and many other misconceptions about TDD) is the adjective "test-driven". It can confuse many people and let them think that it has more to do with testing and quality assurance and less to do with development [2].

4) **Why use TDD?:** TDD can be used for different purposes. The presence of tests allows the system to be changed without the fear of accidentally breaking it. It also lets the low level design evolve instead of having to be decided in advance. If TDD is used properly it can help many developers become more effective, this however requires discipline [2].

---

**B. Software Quality**

According to ISO 9126-1 [4], there are two quality characteristics; internal and external quality. These correspond to verification and validation of the code, respectively. When measuring how TDD affects the quality of the developed software, different studies have measured quality differently - some measure external quality, some internal, and some both. To be able to compare studies to each other, where different methods have been used and where either or both internal and

---

**Fig. 1:** The development flow for Test-Driven Development as described by Janzen and Saeidjan [3].

**Fig. 2:** The development flow for traditional Test-Last Development as described by Janzen and Saeidjan [3].
external quality has been measured, this section will expand on the concept of Software Quality. The aim is to introduce some commonly used concepts and measurements within Software Quality when it comes to studies on TDD.

In a study conducted by Turhan et al., external quality was only measured by functional correctness, and they did not use other measures of external quality such as usability or usefulness [5]. To be able to measure the functional correctness, they divided the software development tasks into smaller tasks called user stories, where each user story corresponded to one functionality or feature from the end-users point of view. The functional correctness was then calculated using the number of passed acceptance tests of each user story, where the acceptance tests were developed by the researchers leading the study [5]. The final measure of the external quality was then the average percentage correctness.

Comparing this methodology to the one used in a study by Rafique and Misic, the latter consisted of a more complex analysis - in addition to looking into the number of acceptance tests that passed, they also counted the number of defects per KLOC, the total number of tests that passed, as well as the quality mark given by the client [6].

The majority of the studies analysed in this report have only tested the impact of TDD on external quality, and according to a study conducted by Bissi et al., the most common way of measuring the internal quality in TDD research is through looking at the test code coverage [7]. However, in one study measuring internal quality impact of TDD, carried out by Janzen and Saiedian, the internal quality of the developed code was measured by looking at four things; code size, code complexity, cohesion and coupling [3]. The code complexity was measured partly by looking at the size of each class or method, where the authors stated that a larger code size often equates more complex code. However they also measured cyclomatic complexity and nested block depth when evaluating the code complexity. Cyclomatic complexity is evaluated by looking at the number of nodes, edges and discrete connected components [9]. and Janzen and Saiedian state that higher cyclomatic complexity as well as nesting leads to less maintainable code, which would then mean lower code quality. When measuring the coupling and cohesion, they measured the number of connections between objects for coupling and the value of LCOM5 for cohesion [3].

These measurements, as well as code-density metrics, which examines the size (in terms of LOC) of classes, methods or features developed are used in a study conducted by Shull et al. [9].

A possible reason that few of the studies linking software quality to TDD examine the internal software is presented in another study by Janzon and Saiedian [10]. Here, the authors suggest that this could be because some of the tests produced when applying TDD generally correlate to external quality, but also that internal quality is a measurement that is more prone to debate.

In conclusion, software quality can be divided into internal and external quality. External quality has been measured using one or several of the following metrics;
- functional correctness, i.e. how many acceptance tests that passed, compiled into an average percentage correctness.
- number of defects per KLOC,
- the total number of passed tests,
- a quality mark given by the client.

Internal quality, being generally perceived as more of a subjective metric, has been measured using the following metrics;
- code coverage,
- code size,
- code complexity (McCabe’s cyclomatic complexity),
- cohesion,
- coupling.

III. ANALYSIS

When it comes to evaluation of TDD, it is important to note that it is hard to make conclusions about the benefits of TDD without having several factors in mind. The most important high level factors to take into consideration are software quality (external and internal) and productivity. These high level factors can each be divided into several other factors that are important to consider when evaluating TDD, such as the granularity of the project, meaning how long the development cycles are. The longer cycles, the larger the granularity. Another important factor to consider is uniformity, which correlates to how well the time schedule is followed in each development cycle, e.g. a high uniformity means that each cycle is approximately the same length. It is also important to consider re-factoring effort, e.g. how many development cycles it takes to re-factor a given task [5].

A. Benefits

This section aims to summarise the analysis of the reports that were used in this report, presenting the benefits that TDD has proved to have on external and internal quality.

1) External Quality: In a study conducted by Fucci et al. and Turhan et al., where it is compared whether it matters to test first (TDD) or test last (TLD), it was noted that there were several benefits with using TDD. In projects with low granularity (short development cycles) that had very low variation in cycle time (uniformity) it was observed that the external software quality was significantly improved. External software quality was here defined as the extent to which the system matched the functional requirements of the system by evaluation of the quality of each user story and comparing that to the total amount of user stories [5].

In a systematic literature review conducted by Bissi et al. in regards to how TDD affects internal quality, external quality and productivity, it was also observed that the external quality was improved when using TDD. In the study, they compiled the results from 17 studies. Out of these 17 studies, only one study showcased a negative impact on external software quality and one study came to the conclusion that TDD neither improved nor decreased the external software quality. This means that approximately 88% of the studies came to the conclusion that TDD improved external software quality, which is a significantly positive result. The remaining 12%,
which were two studies that did not see a positive impact on external quality when using TDD, were conducted in an experimental academic setting, meaning that the programmers in the study were students and did not develop a product meant for the market. One of the studies showed a neutral impact on the quality, that is, no benefit or drawback from using TDD. A study conducted by Pancur et al. and Trampus et al., where an empirical evaluation of TDD development in a university environment was studied, was the only study that showed a negative impact on the external software quality when using TDD [7].

In another study based in an industrial setting, it was observed that the code produced using TDD resulted in significantly higher pass-rate for functional tests (which would then correspond to a higher external quality). The observed numbers were anywhere from 18% increased pass-rate for functional tests to an impressive 50% increased pass-rate. It was also observed that software produced through TDD resulted in a higher overall code coverage as well (which could indicate a higher internal quality). When tests are written before development, the features are defined before they are actually realised, which leads to the customers being more satisfied [11].

2) Internal quality: According to a comparative case study on the impact of TDD on internal quality (measured by test coverage and program design) conducted by Maria Siniaalto and Pekka Abrahamsson, all (four) studies in the paper that were conducted in an industrial setting found that usage of TDD resulted in a reduced defect density. What classified the studies to be in an industrial setting according to Siniaalto and Abrahamsson was the fact that they were primarily conducted by professional developers that were developing a product for the industry [12]. Two of the four studies mentioned were conducted at Microsoft in the United States of America, one was conducted at IBM in the United States of America and one was conducted in China [12]. A decreased defect density implies an improvement of both internal and external software quality, as fewer defects make more tests pass which means improved internal quality, and a decrease of defects also means that the customer will be more satisfied with the final product, which is an indication of improved external quality.

A study conducted by Janzen and Saedidian [3] shows that TDD could not be shown to increase the internal quality when compared to a TLD. There were even worries as to how the quality would be affected if tests were not implemented as they should be when applying TDD - i.e. how the quality would be affected if the development group did not follow the TDD methodology.

B. Drawbacks

When it comes to drawbacks with using Test-Driven Development, it can be observed that the refactorability of the code was impacted negatively by TDD [5]. It was also observed that TDD had a negative or neutral impact on the internal software quality in approximately 40% of the TDD studies that were conducted in an experimental academic setting. Furthermore, 8 of the 17 studies conducted on TDD showcase that the productivity decreased when using TDD and five of the studies showed indifferent results in regards to productivity when applying TDD. Only the papers conducted in an academic setting showed positive or no impact on productivity using TDD, while the rest stated negative results [7]. Even though productivity does not directly correspond to software quality, it is important to mention that there are other drawbacks when using TDD that might "cancel out" the quality benefits.

According to a comparative study on the impact of TDD conducted by Pekka Abrahamsson and Maria Siniaalto, a study conducted by Geras et al. in Chicago in the USA showcased that TDD in some cases led to a false security in regards to the quality of the code. The false sense of security lead to problems at the acceptance test level which indicates a negative impact on quality in some cases [12]. The study that showcased this was conducted in a semi-industrial setting where the developers were experienced programmers, however, they were not developing a real product.

C. Recommendations & Further Research

After analysing the research available on the quality impact of TDD, it is apparent that there is a lack of coverage on the impact of TDD in regards to different frameworks and programming languages used. As mentioned by Bissi et al., most of the research that compares TDD and TLD is conducted where the developers are using Java. The reason for this is that Java has great tools for TDD, such as JUnit [7]. However, it would be interesting to see how the effects of TDD were when using other languages that did not have that much IDE support, meaning that automate tests would be a more difficult task.

Additionally, as described by Munir et al., there is a lack of studies conducted with high rigour and high relevance [13]. These are the studies that are giving the most important results, as they are the studies that are most similar to real industrial practices. It is important to consider the experimental and academic studies as well, however, there is a significant lack in studies conducted in an industrial setting which prevent accurate conclusions about how TDD impacts development conducted by professional developers that are developing a product for the market. As a reason of this, it is recommended that more studies are conducted at real companies that are practising TDD.

IV. Conclusion

To conclude, there are numerous factors to take into consideration when it comes to evaluating how agile testing impacts software quality. As stated in the drawback section of the paper, it was observed that the refactorability of the code was impacted negatively, hence some parts of the quality was negatively impacted by TDD. A reason why TDD impacted refactorability negatively might be that the developers are so focused on getting the tests for the current unit that they are writing to pass that they are missing to consider the software architecture as a whole. This leads to the code being difficult to refactor as every module is a stand alone without having scaleability in mind.
There is an issue with several factors playing a role in the measured quality of a software project. For example, if you conduct a study to measure the external software quality using TDD and this study is conducted in a somewhat different environment than the study you wish to compare it to, then you cannot know whether the differences in outcome is due to using TDD or to the fact that the environment differs and causes the programmers to perform differently. This seems to be the issue with many of the studies that has been examined in this report, where the environment differs as well as the targeted group and other factors that may influence the result. This makes it difficult to compare the studies with each other in order to gain an overall picture of the improvement of quality using TDD.

The studies conducted on the benefits and drawbacks of TDD show contradicting results in terms of productivity as well as internal and external quality improvement. The studies conducted in an academic setting are generally the studies that contradict the results of studies conducted in a semi-industrial or industrial setting. The reason for this might be that TDD grants great benefits for professional developers with great experience, however, it proves to have a negative impact on student developers most likely due to inexperience in software development as well as them not being able to fully adapt to the usage of TDD.

The concluded quality of the software that may be acquired by using TDD varies depending on the quality being in focus of the study, whether it’s the internal quality or the external quality that is being examined. In the studies conducted by Fucci et al. and Turhan et al. where external quality is the main focus, great benefits could be found using TDD in regards to software quality improvement. The same applies for the review conducted by Bissi et al. where a majority of the reviewed studies showed a benefit from using TDD to assure external software quality. Overall, one can conclude from these studies that using TDD when developing software is mainly beneficial from the perspective of external quality assurance. Some examples where this is shown are in higher functional correctness, better evaluated quality of user stories and higher pass-rate for functional tests.

The benefits and drawbacks of TDD in relation to the previously mentioned metrics used to measure internal quality can be evaluated as following:

- Functional correctness: TDD is beneficial,
- Number of defects per KLOC: TDD is beneficial,
- The total number of passed tests: TDD is beneficial,
- A quality mark given by the client: TDD is beneficial.

As far as internal software quality goes, the studies conducted show a more unfocused result. While the study made by Maria Siniaalto and Pekka Abrahamsson showed that TDD improved internal quality by reducing defect density, another study conducted by Janzen and Sacidian did not show any improvements in internal software quality compared to TLD. Other studies even showed that TDD had a negative impact on the internal software quality. It is hence difficult to draw a conclusion regarding the improvement of internal quality while using TDD, and more studies would need to be conducted in order to reach a more coherent result as the current results point in different directions.

The benefits and drawbacks of TDD in relation to the previously mentioned metrics used to measure internal quality can be evaluated as following:

- Code coverage: TDD is beneficial,
- Code size: TDD is neutral or not beneficial,
- Code complexity: TDD is neutral or not beneficial,
- Cohesion: TDD is neutral or not beneficial,
- Coupling: TDD is neutral or not beneficial.

Since the benefits of using TDD when developing software seems to vary when external quality respectively internal quality is the focus of the study, no conclusion can be drawn regarding the general improvement of software quality using TDD instead of TLD. Rather one has to bear in mind the various aspect of software quality when using different coding techniques, and make an individual evaluation regarding which aspects of the software quality that is of most importance for the software project in question. If external software quality is valued the most, then TDD seems to be the preferred programming technique in the majority of cases. However if internal software quality is regarded most important, the choice of programming technique is less obvious, where both TDD and TLD would make a good candidate. If refactorability or productivity is the main focus of a software project, TDD might not be the best choice, and if chosen, possible negative impacts should be taken into consideration.

Despite of TDD being far from flawless, with several detected drawbacks, it has made an important impact on today’s software development. By shifting focus from producing functioning code fast, no matter the size or complexity, a well thought out programming practise could be introduced, where every piece of code is necessary and every test is derived from a requirement of the user. TDD has lifted the importance of software testing as well as having well defined goals and requirements before producing the code. Research like the one we have examined in this report is a key step towards understanding the benefits and drawbacks of TDD, and along with further research the practice of TDD can be improved and refined to be able to achieve an even higher level of software quality than today.

V. CONTRIBUTION STATEMENT

Jan, Isabella, Cecilia and Fatima wrote the report, each having their own main responsibility area. These main responsibilities were Analysis (Jan), Conclusion (Cecilia), Test-Driven Development and References (Fatima), Software Quality (Isabella). Fatima contributed to the report by arranging the sources in order and making sure that each reference is correctly presented.

Each member helped each other with proof reading and adding necessary sections to each others areas, to make sure the report followed an easy to read structure. The introduction and the abstract were both written by Jan and Fatima. The majority of the conclusion was written by Cecilia but Jan also contributed to the first part of the conclusion.
REFERENCES


