# TDD$^{HQ}$: Achieving Higher Quality Testing in Test Driven Development

Adnan Čaušević, Sasikumar Punnekkat and Daniel Sundmark
School of Innovation, Design and Engineering
Mälardalen University, Västerås, Sweden
{adnan.causevic, sasikumar.punnekkat, daniel.sundmark}@mdh.se

***Abstract*—Test driven development (TDD) appears not to be immune to positive test bias effects, as we observed in several empirical studies. In these studies, developers created a significantly larger set of positive tests, but at the same time the number of defects detected with negative tests is significantly higher than those detected by positive ones. In this paper we propose the concept of TDD$^{HQ}$ which is aimed at achieving higher quality of testing in TDD by augmenting the standard TDD with suitable test design techniques. To exemplify this concept, we present combining equivalence partitioning test design technique together with the TDD, for the purpose of improving design of test cases. Initial evaluation of this approach showed a noticeable improvement in the quality of test cases created by developers utilising TDD$^{HQ}$ approach.**

## I. Background

One of the most prominent, quality related, practices in agile is test driven development. Test driven development (TDD) gained on popularity with the introduction of the eXtreme Programming (XP) methodology [1]. As its name reveals, TDD is a development practice which essentially requires from the developers to first create a test case before writing the code for a given functionality. What is important to note here is that developers now have the full responsibility for creating executable test cases with an automated verdict for the unit or component level of the system that they are developing. In the systematic literature review [2], we catalogue the current body of knowledge with respect to empirical studies performed for the purpose of evaluating TDD. Seven potentially limiting factors of industrial adoption of TDD were identified, one of them being *developers inability to write efficient and effective test cases*. To gain better understanding of this problem and validate significance of such a limiting factor, we performed an empirical study as part of a master-level course on Software Verification & Validation. The main goal of this study [3] was to investigate how developers can benefit from additional knowledge on software test design techniques while performing TDD. Although, we could not establish any statistically significant difference between students performance before and after the course, we did notice a surprisingly high ratio of positive test cases within their test suites. This effect is known as "positive test bias" [4] where testing is done using more positive or specification defined inputs. However, such a result may not come as a surprise considering that TDD is designed in way to lead developers in the writing of positive tests which will guide them forward in the implementation of a correct functionality. What was noteworthy to investigate further was consequences on the quality of testing in TDD we might face if developers' focus is only on creating positive test cases.

A follow-up study [5] was designed to further investigate if there are any differences in the defect detection effectiveness of positive and negative test cases created when following the TDD approach. By the term *negative test case*, we refer to a test case that was created for the purpose of exercising a program in a way that was not explicitly specified in the requirement. On the other hand, a *positive test case* exercises a program behaviour as it is specified in the requirement. The results of our study again point out a significantly higher number of positive tests compared to negative ones, but at the same time a much higher defects detection effectiveness of negative tests compared to positive ones. We replicated our experiment in an industrial setting, using professional developers as subjects of study. The results of this study [6] showed a statistically significant difference between the number of positive and negative test cases. This way we could identify that positive test bias exists even when professional and experienced developers are following TDD. The difference in defect detection effectiveness, between positive and negative test cases, was also statistically significant. Around 29% of all test cases created by developers were negative, but at the same time contributing in revealing as much as 71% of all the defects found by all test cases. After further analysis of the solutions provided by the participants, we identified a general problem of when to consider convenient to write negative test cases during the test driven development process. We noticed that a more systematic approach in modifying TDD is needed and our idea was to investigate the possibility of extending test-driven development with a particular test design technique. This way we can facilitate consideration of unspecified requirements during the development to a higher extent and thus minimise the impact of a potentially inherent effect of the positive test bias in TDD.

## II. TDD$^{HQ}$ - Higher Quality Testing in TDD

Although, in simple words, TDD is about writing test cases before writing the code, there are few noteworthy steps in the process that should be mentioned here. Execution of test cases happens twice and this at first might seems like a redundant task. However, it is needed for a developer to see that the newly added test case is failing (usually displayed in red) in order to continue working on creating minimal changes to the code to pass that test (this time displayed in green) [1]. In addition, after fully implementing a set of features, developer have to perform refactoring of the code. Refactoring is a process of modifying the code for the purpose of improving its design, readability, maintainability, etc. without violating existing functionality. Because of these mentioned aspects,

TDD is sometimes refereed as a ”red-green-refactor” process. In the definition of TDD there is no explicit guidance on how a developer should design and add a new test case. The reason for this problem is very obvious. TDD is a development practice and not a test design technique. Test data is usually formed from usage examples and serve as a safety net to detect potential miss-implementation of required functionality. In other words, test cases created using TDD could be considered as a by-product of the development process. But, often they are not. Many agile teams see the value in having a high number of automated test cases created at such an early (unit level) stage of software product development and value those tests as a considerable addition to the overall testing effort. It is known that benefits, mostly in terms of cost, are significant when defects are detected at the early stage of development [7]. However, as we discussed in the previous section, test cases created in TDD are *not designed to detect defects* but rather to provide confidence in the correctness of implemented functionality. In order to increase a defect detection effectiveness of test cases created when using test driven development, we are proposing a modification to the standard TDD process flow, named *TDD$^{HQ}$ - Higher Quality Testing in Test Driven Development*, detailed in Figure 1.

- **A** - Choosing Quality Improvement Aspect

When testing software product, members of quality assurance teams investigate various aspects of product quality: functionality, performance, security, usability, robustness, etc. For them, it is important to cover both functional and non-functional quality features. However, developers tend to focus mainly on the functional aspects of software quality, as it was noticed in our previous empirical studies. This is why it is needed for a developer to explicitly choose an aspect of quality improvement during test driven development which will further guide designing of subsequent test cases.

- **B** - Selecting Test Design Technique

After deciding on the quality improvement aspect that should be in the focus of the current iteration, one of the appropriate test design technique should be selected. It is however important that this test design technique directly contribute to the previously chosen quality improvement aspect. In case there is a possibility to select two or more complementary test design techniques, developers could choose to iterate the process flow with the same quality improvement aspect but each time using a different test design technique for the same feature.

- Check Whether More TCs for **B** in **A**

Once the quality improvement aspect and the test design technique are determined, a classical *red-green* phase of TDD is conducted. Since a particular test design technique could require (by design) creation of several test cases, it is important to reflect if more test cases are needed for a given test design technique selected in **B** to satisfy a quality improvement aspect selected in **A**. If more test cases are needed, an additional *red-green* phase should be conducted for each test case individually.

- Check Whether All Q.I. Aspects Covered

After fully implementing a feature with the specific quality improvement aspect in mind and using the appropriate test design technique(s), developers could iterate through the implementation of the selected feature with a new **A-B** combination, or choose to continue with adding new features, if they exists. Sometimes, the decision on usage of the set of A-B combinations could be taken upfront, for all features by developer itself, or it could be enforced at the organisational level as a development policy implied to all development teams.

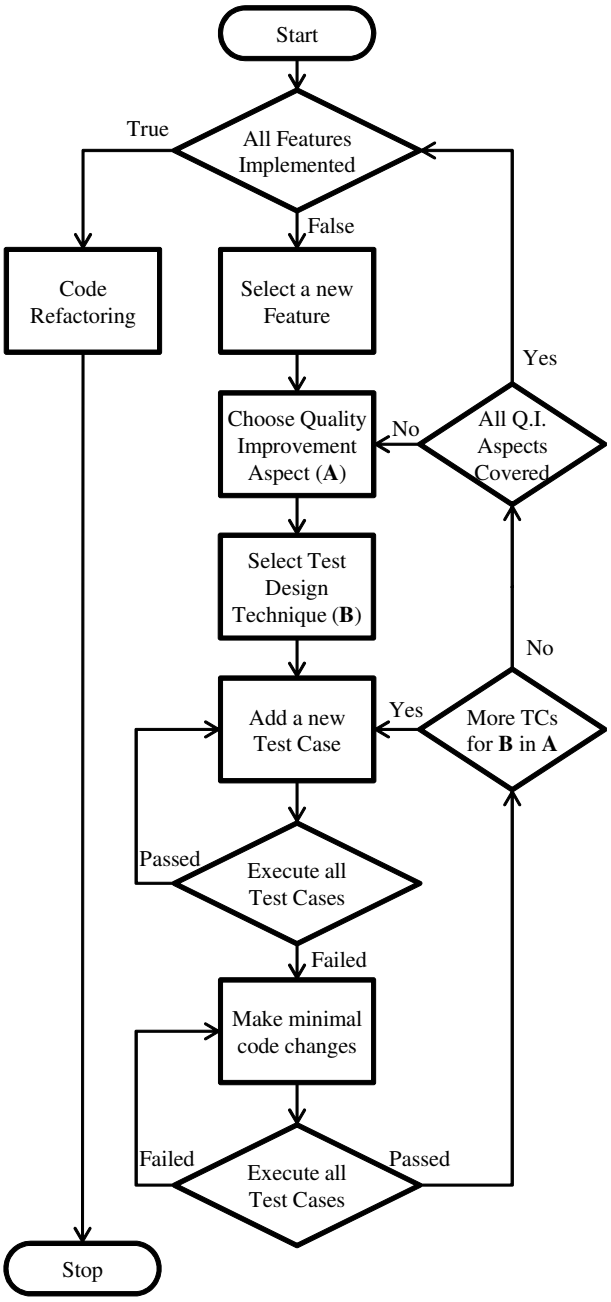


Fig. 1. TDD$^{HQ}$ additions to standard TDD Process Flow

### A. Combinations of *A* and *B*

By using the general TDD$^{HQ}$ process flow (Figure 1), a specific instance of the process could be created purely based on the combinations of quality improvement aspects and test design techniques selected (**A** and **B**). Let us assume

that a developer would like to use a *classical* TDD process for implementing a new feature. Focusing on functionality as quality improvement aspect and using functional testing as a test design technique, developers can continue working using accustomed traditional TDD flow. If, on the other hand, developers would like to increase robustness of their software product, set of combinations provided in Table I could be one example of how to achieve such a goal. Those combinations are used as well in the empirical evaluation of our approach, detailed in Section III.

TABLE I. **A-B** COMBINATION FOR INCREASED ROBUSTNESS IN TDD

| Step | **A** | **B** |
|---|---|---|
| 1 | Functional | Functional testing |
| 2 | Robustness | Equivalence partitioning |

## III. EVALUATION OF THE PROPOSED APPROACH

To better understand the potential improvements and the overall effect that our proposed approach might have on the software product quality, an empirical study was conducted in an academic environment. The experiment was setup with one goal in mind. We wanted to investigate if choosing set of **A-B** combinations for increased robustness in TDD (Table I) when following the TDD$^{HQ}$ process flow, will significantly increase the defect detection effectiveness of the produced test cases. For that purpose, a following research question was defined:

> **RQ:** *Does the usage of TDD$^{HQ}$ improve defect detection effectiveness of test cases?*

By a defect detection effectiveness of test cases, we consider test case's ability to detect a defect in the code. This way, for each test case, we have a number of how many defects a particular test case can reveal. In order to realistically measure the quality of testing we need to essentially have access to an ideal test suite which is capable of finding all the defects. Our approach here will be to approximate such an ideal test suite by combining all the test suites developed by several individual developers working on the same problem. Given such a set of multiple implementations and associated test suites, we are then able to cross-compare the ability of test cases to find defects. This is a similar approach we used in our previous empirical studies ( [5], [6]). In order to test the goal of the experiment, with respect to the stated research question, the following null and alternative hypotheses were formulated:

- $\mathbf{H}^1{}_0$ There is a difference in the quality of tests of *TDD* and *TDD$^{HQ}$* participants.
- $\mathbf{H}^1{}_a$ There is no difference in the quality of tests of *TDD* and *TDD$^{HQ}$* participants.

Subjects of the experiment are 22 master students admitted to the course on Software Verification & Validation (V&V) at Mälardalen University in Sweden during the autumn semester of 2012. The experiment was part of the hands-on laboratory assignment within the V&V course, and the subjects earned credits for the participation. Students were informed that the final grade for the course will be calculated only based on the written exam and their performance during the laboratory work will not affect the final grade. However, they had to fully complete their assignment. Similarly as in our previous studies, this experiment used a bowling game score calculator problem. The specification for this was based on the Bowling Game Kata. Participants in the TDD$^{HQ}$ group were instructed to use TDD in combination with equivalence partitioning test technique to develop software solutions. Participants in the TDD group were instructed to use traditional TDD approach for software development. Detailed information about the problem and instructions are provided on first authors webpage[1]. To avoid problems with subjects' unfamiliarity with the JUnit testing framework and/or Eclipse IDE, a dedicated lecture on the usage of both in TDD was provided to students as well as several hands-on assignments. In addition, subjects were given an Eclipse project code skeleton with one simple test case at the beginning of the experiment. Subjects worked individually on the implementation and were randomly assigned to follow either TDD or TDD$^{HQ}$ approach. Their work was not time-boxed and subjects were given an opportunity to work on the implementation until they have enough quality confidence in the submitted solution. Experiment participants were instructed, upon finalising their software implementation, to upload the source code together with the test cases to Blackboard, the course management tool.

Table II presents aggregated data of our participants test cases categorised by type of the test case and experiment group who created them. Looking at the overall number of test cases there is no significant difference, since TDD$^{HQ}$ group created around 5% less test cases than TDD group. By using just positive test cases from both groups we detected 355 defects, while negative test cases contributed with 1268 defects. This is one more evidence how valuable negative test cases are as part of any test suite. Looking at the differences between the groups we can observe that TDD$^{HQ}$ group detected around 17% more defects in total, but if we look only at the negative test cases, the increase in defect detection is around 20%. With this study we are mainly focusing on the quality of produced test cases by measuring number of defects they can detect in the code. Since this code is also written by our participants, we can as well observe number of defects found in the code of our experiment participants' groups. In total, code of TDD$^{HQ}$ group had around 26% less defects than code of TDD group. Only looking at the defects detected by negative tests, TDD$^{HQ}$ group had as much as 31% less than TDD group. Interestingly, more positive defects (around 8%) are detected in TDD$^{HQ}$ group.

TABLE II. AGGREGATED DATA

| | Test Cases Distribution | | Defects Detected by Test Cases | | Defects Found in Code | |
|---|---|---|---|---|---|---|
| Test Type | *TDD* | *TDD$^{HQ}$* | *TDD* | *TDD$^{HQ}$* | *TDD* | *TDD$^{HQ}$* |
| *Positive* | 124 | 116 | 169 | 186 | 171 | 184 |
| *Negative* | 108 | 103 | 576 | 692 | 749 | 519 |
| Total | **232** | **219** | **745** | **878** | **920** | **703** |

This data is also used to address the research question. The **Mann-Whitney** nonparametric test was used in order to test the $\mathbf{H}^1{}_0$ null hypotheses with $\alpha = 0.05$. We can not reject stated hypothesis (**p-value is 0.2370851**), leading to the conclusion that there is no statistically significant difference in the quality of tests produced by participants of our experiment when following TDD and TDD$^{HQ}$ approach. Similarly to several previously published experiments on TDD, this study was also performed in an academic setting. Therefore, external

[1] http://www.mrtc.mdh.se/~acc01/tddhq/

validity and the possibility to generalise the results of this study, is threatened with following limitations: (i) using students as subjects, (ii) using small scale objects of investigation, and (iii) having short duration of the experiment.

## IV. Related Work

Our work in this paper consists of two major aspects which should be considered when relating it to the current body of knowledge: (i) theoretical proposal for achieving higher quality of testing in test driven development - TDD$^{HQ}$ and (ii) empirical study designed to measure defect detection effectiveness as a quality attribute of a test case. When looking at empirical studies with TDD as the main focus of investigation, we can observe that most of them are performed for the purpose of detecting potential benefits in the quality of the produced code. This was as well one of the finding in our systematic literature review [2] listing 48 empirical studies on TDD. One study did however had the focus on quality attributes of *test cases* when a test-first approach was used. Madeyski [8] investigated how usage of TDD can impact branch coverage and mutation score indicators. In his experiment, 22 students were divided in two groups: the test-first and the test-last, with the task of developing a web based conference paper submission system. This experiment shows no statistically significant differences in branch coverage and mutation score indicators, between the test-first and the test-last groups. We identified one additional study with the focus on developer's testing ability when following test driven development approach [9]. This was an industrial observational experiment where developers performed programming tasks in their own offices without the control of researchers. Once developers submitted their code and test cases, researchers performed mutation testing analysis to identify complementary test cases to the ones created during TDD process. Those unit tests, created by researchers, were still able to find several software faults in the developers' submitted code.

## V. Conclusions and Future Work

This paper present TDD$^{HQ}$ concept, an approach for achieving higher quality testing in test driven development. TDD$^{HQ}$ approach is focused on augmenting the standard TDD methodology with a suitable test design technique, for the purpose of satisfying a criteria of a specific quality improvement aspect we would like to achieve. Classical test driven development is designed in way to lead developers in writing of positive tests which will guide them forward in the implementation of a correct functionality. By using TDD$^{HQ}$, developers do not necessarily focus only on verifying functionality, as it is defined in the given requirements, but they can as well increase security, robustness, performance and many other quality improvement aspects for the given software product, without interfering the classical "red-green-refactor" flow. To exemplify this concept, we defined a set of quality improvement aspects and test design techniques combinations, for the purpose of improving robustness of the produced software solution. This was achieved by combining equivalence partitioning test design technique together with the test driven development. Initial evaluation of this approach was performed in an academic experiment, where the group who followed this approach created 5% less test cases but at the same time detected 17% more defects compared to a group of participants using classical test driven development. Such an increase in defects detection at an early stage of development (unit level) could have a very strong impact on the overall development process. Detecting defects at later stages (integration and system level testing) comes with a much greater cost [7]. However, since we could not show a statistically significant difference in the quality of test cases, between our group of participants, a new study needs to be conducted with a larger sample size. Since this empirical study investigated only one specific combination of quality improvement aspect (A) and accompanied test design technique (B), an additional research effort is needed to fully list, categorise and map sets of appropriate A-B combinations for each particular domain where TDD$^{HQ}$ could be fully utilised. Once this list is complete, a further investigation of its significance should be performed in the form of several empirical studies. Design of those studies should, in addition, account for investigation of the significance of possible threats when using TDD$^{HQ}$, specifically threats to developers productivity and efficiency. The proposed improvements (TDD$^{HQ}$) in the process flow of performing test driven development is expected to result in not just higher quality of produced tests but also in the higher quality and productivity in software systems, in terms of robustness and early defects detection.

## Acknowledgment

This work was supported by the Mälardalen Real-Time Research Center (MRTC) and SSF-funded Synopsis project at Mälardalen University in Sweden.

## References

[1] K. Beck, *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[2] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, march 2011, pp. 337 –346.

[3] A. Causevic, D. Sundmark, and S. Punnekkat, "Impact of Test Design Technique Knowledge on Test Driven Development: A Controlled Experiment," in *XP*, ser. Lecture Notes in Business Information Processing, C. Wohlin, Ed., vol. 111. Springer, 2012, pp. 138–152.

[4] L. M. Leventhal, B. Teasley, D. S. Rohlman, and K. Instone, "Positive Test Bias in Software Testing Among Professionals: A Review," in *Selected papers from the Third International Conference on Human-Computer Interaction*. London, UK: Springer-Verlag, 1993, pp. 210–218.

[5] A. Causevic, S. Punnekkat, and D. Sundmark, "Quality of Testing in Test Driven Development," in *Quality of Information and Communications Technology (QUATIC), 2012 Eight International Conference on the*, September 2012.

[6] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark, "Effects of Negative Testing on TDD: An Industrial Experiment," in *XP*, ser. Lecture Notes in Business Information Processing, H. Baumeister and B. Weber, Eds., vol. 149. Springer, 2013 (to be presented).

[7] R. Megen and D. Meyerhoff, "Costs and benefits of early defect detection: experiences from developing client server and host applications," *Software Quality Journal*, vol. 4, no. 4, pp. 247–256, 1995.

[8] L. Madeyski, "The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment," *Inf. Softw. Technol.*, vol. 52, pp. 169–184, February 2010.

[9] W. Shelton, N. Li, P. Ammann, and J. Offutt, "Adding Criteria-Based Tests to Test Driven Development," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 878–886.