Quality of Testing in Test Driven Development

Adnan Čaušević, Sasikumar Punnekkat and Daniel Sundmark
Mälardalen University, Sweden
firstname.lastname@mdh.se

***Abstract*—Test-driven development is an essential part of eXtreme Programming approach with the preference of being followed in other Agile methods as well. For several years, researchers are performing empirical investigations to evaluate quality improvements in the resulting code when test-driven development is being used. However, very little had been reported into investigating the quality of the testing performed in conjunction with test-driven development.**

In this paper we present results from an experiment specifically designed to evaluate the quality of test cases created by developers who used the test-first and the traditional test-last approaches. On an average, the quality of testing in test-driven development was almost the same as the quality of testing using test-last approach. However, detailed analysis of test cases, created by test-driven development group, revealed that 29% of test cases were "negative" test cases (based on non-specified requirements) but contributing as much as 65% to the overall tests quality score of test-first developers.

We are currently investigating the possibility of extending test-driven development to facilitate non-specified requirements to a higher extent and thus minimise the impact of a potentially inherent effect of positive test bias.

***Index Terms*—software testing; test case quality; test driven development; experiment;**

I. Introduction

Quality of agile methods is often a focus of empirical studies by researchers due to the inability to formally prove the benefits arising from the usage of such methods. Several factors may contribute to the overall software product quality when using agile methods, such as, usage of short cycles, close customer relationship, pair programming, test-driven development, continuous integration, and many more. When performing empirical investigations, researchers usually try to isolate one factor in particular and evaluate its effects on the code quality which is often the main metric of evaluation. However, when isolating test-driven development factor, researchers tend to omit another important metric, quality of test cases. We believe that measuring the quality and characteristics of test cases generated during test-driven development is an important step towards making it more industrially acceptable.

Test-driven development (TDD) was introduced as a practice within eXtreme Programming (XP) methododology [1]. Developers using TDD write automated unit tests before they write the actual code, and hence it is also referred as a test-first approach in literature [2]. Tests are written in the form of assertions and in TDD their purpose is to define code requirements. By using TDD, developers build the systems in cycles of test, development and refactoring.

Test-driven development was identified, in our industrial survey [3], as a most preferred but lesser used practice in industry. Interpretation of a main finding of this study could be: "Respondents would like to use TDD to a significantly higher extent than they actually do currently". This preference towards using TDD could be based on academic research results often pointing improvements of the code quality when TDD is used ([4]–[8]), but also due to the success of early adopters. As a follow up, we performed a systematic literature review [9] for the purpose of identifying any obstacles in the path of full scale adoption of TDD in the industry. Seven factors, which are potentially limiting full adoption of TDD, were identified and listed. Inability of developers to write automated test cases (in an efficient and effective way) is considered to be one of these limiting factors. In the current paper we are presenting analysis results of an experiment formulated and defined in a way to investigate the significance of such a limiting factor.

An experiment was conducted during the autumn semester in 2011 with master students enrolled in the Software Verification and Validation course at the Mälardalen University, with the intention of comparing testing efficiency and effectiveness of agile (test-first) and traditional (test-last) developers. This experiment allowed us to investigate the quality of testing in test-driven development by using the created test cases as a main metric of evaluation.

The remaining of this paper is organised as follows. Section II presents the related research work followed by the experimental design and its execution in section III. The analysis of quality attributes are presented in section IV followed by a detailed investigation on test cases in section V. In section VI, we discuss threats to validity of our study followed by conclusions and future research plans in section VII.

II. Related Work

During the identification of potential limiting factors of TDD adoption, our systematic literature review [9] listed 48 empirical studies that had effects of TDD as the focus of the investigation. Most of the studies had TDD as a primary focus of investigation, but in some cases effects of TDD were investigated in conjunction with some other practice, e.g. pair-programming. Goal of the studies investigating effects of TDD was related in most cases with respect to: (i) the internal or the external code quality improvements, (ii) performance improvements or (iii) a general perception of using TDD. However, we identified only one study [10] where the focus of the investigation was quality attributes of *test cases* when test-first approach was used.

Madeyski [10] investigated how TDD can impact branch coverage and mutation score indicators. In this 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.

Our experiment investigation relates to the work of Madeyski, since we are also measuring the code coverage and the mutation score indicators. However, both those indicators are considered as an internal quality attributes of a test suite. Main difference in our experiment is the enforcement of a programing interface, which allowed us to execute test cases of one individual participant on the code of all other experiment participants. This, as a result, provided us with the insight of the defect-finding ability of participants test cases, thus creating an external quality attribute used for an additional analysis. In the following section we are discussing how internal and external quality attributes are used to make a judgement on the overall quality of participants tests.

III. Experiment Design and Execution

In this section the study design and the process of execution of the experiment are described. Complete study design of the experiment is published in [11]. Specific details of the study (instruction material and code skeleton) can be found at the first author's website[1].

A. Quality of Testing

To evaluate the quality of testing in test-driven development, an experiment was conducted to investigate:

> *Is there a significant difference between the quality of test cases, produced using test-first and test-last approaches?*

The main goal of the experiment was to compare several quality attributes that could relate to the overall quality of testing, when developing software using test driven development (test-first) and traditional (test-last) approaches. Those attributes are grouped as *internal* and *external* quality attributes of test cases.

Internal quality attributes are measuring, by using a specific criteria, effectiveness of test cases which are *accompanied* with the particular source code. For this experiment we used two internal quality attributes:

- Code coverage - measuring to what extent a provided set of test cases is exercising statements in the accompanied source code.
- Mutation score - measuring to what extent a provided set of test cases can detect seeded faults within the accompanied source code.

External quality attributes are measuring effectiveness of test cases on *any given* source code. However, it is assumed that the same functionality is implemented as in the accompanied source code of the particular set of test cases. For this experiment we used one external attribute:

- Defect detecting ability - measuring total number of failing occurrence of test cases on any given source code implementing the same functionality.

In addition to measuring total number of defects found by all tests, we grouped test cases into *positive* and *negative* and calculated how many defects each group can reveal. By a *positive* test case, we refer to a test case designed to test the code for explicitly stated requirement. By a *negative* test case, we refer to a test case designed to test how the program is behaving for a non-given requirement.

B. Hypotheses, Parameters and Variables of the Experiment

In order to test the goal of the experiment, following null and alternative hypotheses were formulated:

- **Test Artefact Quality:**
  - $\mathbf{H}^t{}_0$. There is no significant difference between the quality of the test artefacts produced by test-first or test-last developers.
  - $\mathbf{H}^t{}_a$. Test-first developers produce test artefacts of a higher quality.
- **Code Artefact Quality:**
  - $\mathbf{H}^c{}_0$. There is no significant difference between the quality of the code artefacts produced by test-first or test-last developers.
  - $\mathbf{H}^c{}_a$. Test-first developers produce code artefacts of a higher quality.

The *test artefact quality* and *code artefact quality* are operationalized in a list of response variables, provided in Table I.

C. Subjects of the Experiment

The participants of the experiment were software engineering master students enrolled in the Software Verification and Validation (V&V) course at Mälardalen University during the autumn semester of 2011. The experiment was part of the laboratory work within the V&V course, and the participants earned credits for their participation. They were informed that the final grade for the course will be obtained from the written exam and their performance during the laboratory work will not affect the final grade, although they had to fully complete the laboratory work.

D. Object of the Experiment

The task given to the students was to completely implement and test (to the extent they consider sufficient) a bowling game score calculation algorithm. The specification for this problem was based on the Bowling Game Kata (i.e., the problem also used by Kollanus and Isomöttönen to explain TDD [12]).

E. Experiment Execution

Fourteen participants were randomly grouped in two groups, the test-first and the control (test-last) group. The Eclipse [13] integrated development environment (IDE) was used to create software solution in the Java programming language and the jUnit [14] testing framework was used for writing executable

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

TABLE I
EXPERIMENT RESPONSE VARIABLES

Construct	Variable name	Description	Scale type
Code Artefact Quality	Defects in Code	Number of defects found in a particular code implementation.	Ratio
Test Artefact Quality	Coverage	Statement coverage of test suite when applied to code implementation.	Ratio
Test Artefact Quality	Mutation	Mutation score indicator of test suite when applied to code implementation.	Ratio
Test Artefact Quality	Defects Detected	Number of defects found by test suite in all code implementations.	Ratio

tests. After completely finalising their implementations, the subjects answered a set of questions using an online survey system.

Participants in the test-first group were instructed to use TDD to develop software solutions. Instructions for TDD were given as prescribed by Flohr and Schneider [15]. Participants in the test-last (control) group were instructed to use traditional (test-last) approach for software development. To avoid problems with subjects' unfamiliarity with the jUnit testing framework and/or Eclipse IDE, subjects were given an Eclipse project code skeleton with one simple test case.

IV. ANALYSIS OF QUALITY ATTRIBUTES

In this section, an analysis of the experiment data is performed and presented. Quality attributes are grouped, defined and discussed in separate subsections. Overall descriptive statistics of quality attributes are presented in Table II. First, the quality of code is presented as a separate quality attribute which is not directly defining a quality of testing but it does indirectly contribute to it. Subsequently, three quality attributes which are directly defining quality of testing (quality by code coverage, quality by mutation and quality of test cases) are presented and analysed.

A. Quality of Code (Q_{code})

In addition to measuring number of defects, a particular test case can detect in various source codes, we need to know of what quality a particular source code is. This way we can differentiate test cases which are failing on the code of higher quality. In order to calculate the quality of the code indicator, we use the following formula:

$$Q_{code}(i) = 1 - \frac{N_{FTC}(i)}{N_{TC}}$$

where, i represents a specific individual participant of the experiment, N_{TC} - total number of test cases created by all participants and $N_{FTC}(i)$ - number of test cases which are failing on the code of participant i. Code quality of test-first group was on an average 5.66% higher than the test-last group. Looking at the number of defects found in the code of test-first and the test-last developers, the difference is much higher (244 errors in test-first code and 330 errors in test-last code). This results align to the overall impression that usage of test-driven development does improve the quality of the resulting code.

B. Quality by Code Coverage ($Q_{coverage}$)

The first internal quality attribute of test suite (particular set of test cases) used in our experiment is code coverage. Code coverage is calculated as a percentage of statements that accompanied tests exercise within the total number of statements in a given participants code. This way we have a measurement to what extent a provided set of test cases is exercising statements in the accompanied source code. This data is collected using EclEmma [16] plug-in for Eclipse.

On an average, code coverage that was achieved by test-first and test-last group is nearly the same and has a relatively high value. This, as a result, creates difficulties to derive any conclusions or reason about the difference in quality of groups' test cases by using this particular quality attribute.

C. Quality by Mutation ($Q_{mutation}$)

The second internal quality attribute of a test suite used in our experiment is a mutation score indicator. Mutation score indicator is measuring to what extent a provided set of test cases can detect seeded faults within the accompanied source code. Faults are seeded with Judy [17] mutation testing tool following next approach:

1) Compiled code is provided as an input
2) Compiled set of test cases (test suite) is also provided as an input. However, it is a requirement that all test cases are not failing on the original source code.
3) N variations of the original program were generated with the Judy tool, by using a set of default mutation operators. These variations of programs are referred as *mutants* in literature.
4) A complete test suite is executed for each mutant $n \in N$
5) If any test case within the test suite fails during the execution, current mutant n is marked as "killed". This means that the test suite recognised the modification in a particular variation of the original program.
6) Mutation score is calculated as $m/|N|$ (total number of killed mutants m ($m \leq |N|$) divided by total number of variations of the original program $|N|$)

Average mutation score indicators for both the groups were similar, thus making it again difficult to identify if the test cases of any group could be considered to be of a better quality by using this quality attribute.

D. Quality of Test Cases ($Q_{testing}$)

The only external quality attribute of a test suite we used in our experiment is a defect detecting ability attribute. For each participants test suite we calculated a total number of defects discovered in all other participants source code. As a result,

TABLE II
DESCRIPTIVE STATISTICS OF QUALITY ATTRIBUTES

Construct	Variable	Dev. Approach	Mean	Median	Sum	Std. Dev.	Std. Error	Min	Max
Code Artefacts	Defects Found in Code	Test-First	34.86	44	244	24.04	9.09	3	58
		Test-Last	47.14	24	330	48.54	18.35	9	143
	Code Quality	Test-First	83,94%	79,72%	-	11,08%	4,19%	73,27%	98,62%
		Test-Last	78,28%	88,94%	-	22,37%	8,45%	34,10%	95,85%
Internal Test Artefacts	Code coverage	Test-First	96.34%	98.78%	-	5.89%	2.23%	83.28%	100.00%
		Test-Last	96.18%	98.83%	-	4.22%	1.60%	89.4%	100.00%
	Mutation Score	Test-First	81.90%	85.12%	-	11.75%	4.44%	57.54%	92.57%
		Test-Last	83.29%	84.31%	-	5.80%	2.19%	75.49%	91.30%
External Test Artefacts	Defect Detecting Ability	Test-First	40.57	17	284	47.34	17.89	13	144
		Test-Last	41.43	31	290	26.68	10.09	17	86
	Tests Quality	Test-First	27.30	9.45	191.07	35.46	13.40	6.70	104.63
		Test-Last	27.27	18.59	190.92	18.68	7.06	9.76	61.35

in Table II, we can see that test-first developers test cases discovered in total 284 defects, while test-last developers test cases discovered 290 defects. This result could be interpreted that our experiment participants, regardless of the development approach used, had almost the same testing ability. However, it is important to differentiate test cases which are failing on the code of higher quality from those which are failing on the code of lower quality. For that reason we calculated quality value for each test case of every participant. Quality of test cases for a participant (i) is calculated as a sum of a quality of each test case (j) from a set of test cases (n) of that participant (i):

$$Q_{testing}(i) = \sum_{j=1}^{n} Q_{TC}(i.j)$$

To calculate the quality of an individual test case (j) of a participant (i) we need to know on which participants' code this test case is failing ($m \in M$). The logic we follow is that making an otherwise good quality code fail will give higher quality value for a test case. Sum of the code quality values (Q_{code}) of those participants will define the quality of a particular test case (j):

$$Q_{TC}(i.j) = \sum_{k=1}^{m} Q_{code}(k)$$

Here again, on an average, overal quality of test cases is nearly the same (0.03) for both test-first and test-last group of developers. This means that ability to write automated test cases is of the same quality for developers using test-driven development and developers using traditional test-last approach. However it still remains unclear if quality of testing of our experiment participants, on a global scale, is satisfying this property or not. Basically, we need to investigate if TDD developers are "as good as" or "as bad as" test-first developers in testing. Currently we can only infer that they are on the same level, but cannot confirm whether this level is high or low.

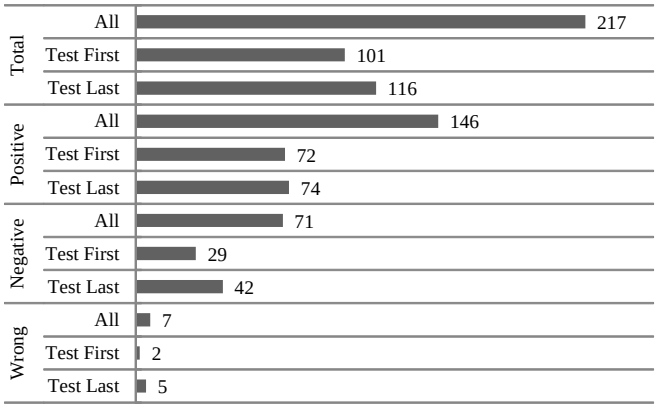

Fig. 1. Distribution of Test Cases

V. IMPACT OF "POSITIVE TEST BIAS" ON TEST CASES

From our experiment, it was difficult to obtain some further understanding about the quality of testing in test-driven development by only performing measurement of various quality attributes. Since the results from our previous investigation [18] pointed out that students have a very small focus on "negative" test cases, this experiment had a built-in mechanism of differentiating whether a particular test case is of positive or negative type.

By a negative test case, we refer to a test case that was created for a purpose of exercising a program in a way that was not specified in the requirements. Positive test case, on the other hand, is exercising a program in a way that was specified in the requirements. In literature, phenomenon of more positive approach to testing is known as a "positive test bias" [19], [20].

In the following subsections, analysis of test cases is performed to identify if positive test bias is present and what effect it has on the quality of testing in test-driven development.

A. Overall Distribution of Test Cases

In Fig. 1, an overview of the total number of test cases is presented. All participants created 217 test cases, out of which

TABLE III
A COMPLETE OVERVIEW OF TEST CASES

		Test-First		Test-Last	
		Positive TCs	Negative TCs	Positive TCs	Negative TCs
All Code	Failing Test Cases	63	23	64	35
	Occurrence	123	161	103	187
Test-First Code	Failing Test Cases	20	23	24	35
	Occurrence	26	83	24	107
Test-Last Code	Failing Test Cases	63	23	64	29
	Occurrence	93	78	79	80

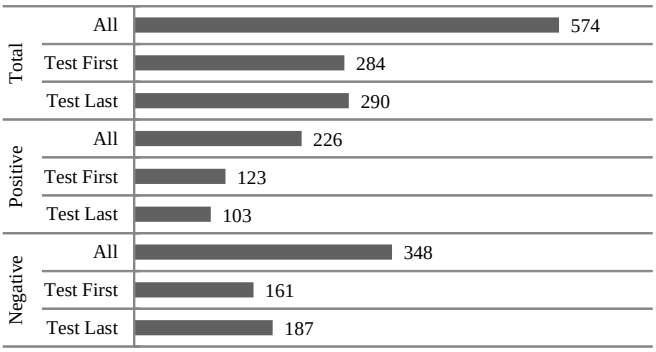

Fig. 2. Failing Occurrence for Test Cases

101 was created by test-first developers and 116 by test-last developers. There were 146 positive test cases and 71 negative test cases in total.

First observation we can make from this data is that both groups of developers created less negative than positive test cases. Additionally, test-first developers created less negative test cases compared to the test-last developers. Interestingly, participants created 7 wrong test cases. These test cases were excluded from our analysis and the total number of test cases (217) does not contain these 7 test cases. By a wrong test case we refer to a test case that has a wrong verdict for the provided input.

B. Failing Occurrence of Test Cases

Since the test cases of each individual experiment participant were executed on all the other participants code, it is often a case that one test case is failing on several occasions. Fig. 2 presents numbers of failing occurrence of the test cases which are failing at least once on any of the participants code. Total number of failing occurrences of test cases (or a total number of occasions where error was detected) was 574 for all participants put together. Both test-first and test-last groups of developers have nearly the same number of failing occurrence (284 and 290 respectively). This, once again, align with the results of the quality attributes, where quality of testing for both groups is relatively the same.

TABLE IV
$Q_{testing}$ FOR POSITIVE AND NEGATIVE TEST CASES

	Test-First Group	**Test-Last Group**	**All**
Positive	66,45	50,47	116,92
Negative	124,62	140,46	265,08
Sum	191,07	190,93	382,00

However, when comparing overall distribution of test cases and number of failing occurrences we can make one interesting observation:

> 146 positive test case detected 226 errors, while
> 71 negative test cases detected 348 errors.

Even though there are nearly 50% less negative test cases, they are still detecting 50% more errors than positive ones. To double-check this observation in Table IV we listed $Q_{testing}$ quality attribute values in groups by: (i) type of test cases (positive and negative) and (ii) approach used for the development (test-first and test-last). When looking at the data of test-first group of developers, we notice:

> 72 positive test case (Fig. 1) with $Q_{testing} = 66,45$,
> and 29 negative test cases had $Q_{testing} = 124,62$

This means that negative test cases (29% of all test cases) were contributing as much as 65% to the overall quality of testing score for the test-first developers.

Table III represent a complete overview of the testing efforts from several perspectives. It is presented in a way to identify test cases (both positive and negative) created by test-first and test-last developers and the effects they had when executed on the code of all participants or on the code of the test-first or the test-last participants. For each set of test cases it is possible to distinguish number of failing test cases and their occurrence. For example, there are 23 negative test cases of test-first developers that are failing on the test-first as well as on the test-last code. However, those test cases are detecting 83 errors in test-first code and 78 errors in test-last code.

VI. THREATS TO VALIDITY - RESERVATIONS

The analysis presented in this study is based on the data from the experiment in [11]. Similar to the previously published experiments on TDD [9], this experiment was also

performed in an academic setting. Therefore, external validity is threatened with some known academic limitations: (i) using students as subjects, (ii) using small scale objects of investigation, and (iii) having short duration of the experiment. Additionally, due to the low number of participants no statistically significant conclusions could be drawn based on the collected data. By using standard software engineering metrics for calculating quality attributes (e.g., coverage and mutation indicator) we addressed eventual construct validity of our empirical research. Additionally, by providing sufficient information about the experiment we are addressing reliability threats of this study.

VII. Conclusions and Future Work

We can relate findings of our analysis to the result of the related study in [10]. Mainly, we can see that difference in test cases between the test-first and the test-last participants is almost non-existant, if code coverage and mutation score indicators are used for comparison. Interestingly, an additional quality attribute that we introduced ($Q_{testing}$) also could not make any distinction between the test cases created by the test-first and the test-last participants.

The experimental data analysis is indicative that the code of test-first group is of better quality as compared to that of the test-last group. This is an interesting observation considering that both the groups had same quality of test cases used to test the implementation. However, even though test-first group has less failing test cases than test-last group on their code, both groups still have a relatively high number of errors in the code.

Main implication of this study is a finding that test-first participants have more positive test cases than negative, which is probably a result of the "inherent"positive test bias of test-driven development approach. By nature, test-driven development is a development methodology and not a test design technique. That is why test cases are "driving"a developer towards implementing required functionality in a constructive rather than destructive way.

The analysis of the results from an experiment, presented in this paper, underlines the importance of having negative test cases as part of the test suite. Test efficiency for test-first developers was highly dependant on the negative test cases which represented 65% of overall testing effort. By calculating the quality of test cases, our study very clearly indicated the impact of a "positive test bias" factor on TDD.

Since there is no underlying theory behind TDD, it is not possible to formally validate its claimed benefits by any other means except performing empirical investigations. In a long term investigation process, this study should be replicated and conducted as a fully controlled experiment, with a higher number of participants in order to validate statistical significance of the presented results.

We are currently investigating the possibility of extending test-driven development to facilitate consideration of unspecified requirements during development to a higher extent and thus minimise the impact of a potentially inherent effect of positive test bias.

Acknowledgments

This work was supported partly by the SWELL (Swedish software Verification & Validation ExceLLence) research school and the SYNOPSIS project at MDH.

References

[1] K. Beck, *Extreme programming explained: embrace change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[2] L. Koskela, *Test driven: practical tdd and acceptance tdd for java developers*. Greenwich, CT, USA: Manning Publications Co., 2007.

[3] A. Causevic, D. Sundmark, and S. Punnekkat, "An Industrial Survey on Contemporary Aspects of Software Testing," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, 2010, pp. 393–401.

[4] B. George and L. Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, no. 5, pp. 337 – 342, 2003.

[5] H. Erdogmus, M. Morisio, and M. Torchiano, "On the Effectiveness of the Test-First Approach to Programming," *IEEE Transactions on Software Engineering*, vol. 31, pp. 226–237, 2005.

[6] D. S. Janzen and H. Saiedian, "On the Influence of Test-Driven Development on Software Design," *Software Engineering Education and Training, Conference on*, vol. 0, pp. 141–148, 2006.

[7] A. Gupta and P. Jalote, "An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development," in *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 285–294.

[8] J. H. Vu, N. Frojd, C. Shenkel-Therolf, and D. S. Janzen, "Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project," in *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 229–234.

[9] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review," in *Proceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST)*, 2011.

[10] 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.

[11] A. Causevic, D. Sundmark, and S. Punnekkat, "Test Case Quality in Test Driven Development: A Study Design and a Pilot Experiment," in *International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)*, May 2012.

[12] S. Kollanus and V. Isomöttönen, "Understanding TDD in academic environment: experiences from two experiments," in *Proceedings of the 8th International Conference on Computing Education Research*, ser. Koli '08. New York, NY, USA: ACM, 2008, pp. 25–31.

[13] Eclipse, http://www.eclipse.org.

[14] jUnit Framework, http://www.junit.org.

[15] T. Flohr and T. Schneider, "Lessons Learned from an XP Experiment with Students: Test-First Needs More Teachings," in *Product-Focused Software Process Improvement*, ser. Lecture Notes in Computer Science, J. Mnch and M. Vierimaa, Eds. Springer Berlin / Heidelberg, 2006, vol. 4034, pp. 305–318.

[16] EclEmma - Java Code Coverage for Eclipse, http://www.eclemma.org.

[17] Judy - Java mutation tester, http://www.java.mu.

[18] A. Causevic, D. Sundmark, and S. Punnekkat, "Impact of Test Design Technique Knowledge on Test Driven Development: A Controlled Experiment," in *Agile Processes in Software Engineering and Extreme Programming - 13th International Conference, XP 2012, Malmö, Sweden, May 20-25, 2012. Proceedings*, ser. Lecture Notes in Business Information Processing. Springer, 2012 (to appear).

[19] B. E. Teasley, L. M. Leventhal, C. R. Mynatt, and D. S. Rohlman, "Why Software Testing Is Sometimes Ineffective: Two Applied Studies of Positive Test Strategy," *Journal of Applied Psychology*, vol. 79, no. 1, pp. 142 – 155, 1994.

[20] 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.