Impact of Test Design Technique Knowledge on Test Driven Development: A Controlled Experiment

Adnan Čaušević, Daniel Sundmark, and Sasikumar Punnekkat

Mälardalen University, Sweden
`firstname.lastname@mdh.se`

Abstract. Agile development approaches are increasingly being followed and favored by the industry. Test Driven Development (TDD) is a key agile practice and recent research results suggest that the successful adoption of TDD depends on different limiting factors, one of them being insufficient developer testing skills. The goal of this paper is to investigate if developers who are educated on general testing knowledge will be able to utilize TDD more effectively. We conducted a controlled experiment with master students during the course on Software Verification & Validation (V&V) where source code and test cases created by each participant during the labs as well as their answers on a survey questionnaire were collected and analyzed.
Descriptive statistics indicate improvements in statement coverage. However, no statistically significant differences could be established between the pre- and post-course groups of students. By qualitative analysis of students' tests, we noticed a lack of test cases for non-stated requirements ("negative"tests) resulting in a non-detection of bugs. Students did show preference towards TDD in surveys.
Although further research is required to fully establish this, we believe that identifying specific testing knowledge which is complementary to the testing skills of a new TDD developer would enable developers to perform their tasks in a more efficient manner.

Key words: test driven development; controlled experiment; software testing

1 Motivation

Test Driven Development (TDD), also known as test-first programming, is an essential part of eXtreme Programming (XP) [1]. TDD requires the developers to construct automated unit tests in the form of assertions to define code requirements before writing the code itself. In this process, developers evolve the systems through cycles of test, development and refactoring. In a recent industrial survey [2], we examined the difference between the preferred and the actual level of usage for several test-related practices. Among the 22 examined practices, surprisingly, TDD gained the highest score of 'dissatisfaction'. This means that the accumulated absolute difference between the preferred and the actual levels of usage was highest in the case of TDD. The nature of this dissatisfaction could be stated as "Respondents would like to use TDD to a significantly higher extent than they actually do currently".

Subsequently we explored the current body of knowledge through an empirical systematic literature review [3] to identify the limiting factors which prevents the successful adoption of TDD. Insufficient developer testing skills was identified as one of the important limiting factors as part of the study. By developer testing skill, we refer to the developer's ability to write efficient and effective automated test cases.

1.1 Problem Statement

TDD in its essence teaches developers on how to perform software development providing some indirect basic testing skills, for example based on *positive testing* (i.e. testing to show that the software "works" using valid input). We are interested in identifying specific testing knowledge which is complementary to the already mentioned testing skills of a new TDD developer. We believe that such a strategy would enable developers to perform their tasks in a more efficient manner resulting in higher quality of software products.

1.2 Research Objective

In the form suggested by Wohlin et al. [4], the research objective of this study can be expressed as follows:

To analyze *the effect of testing knowledge on TDD*
for the purpose of *evaluation of factors affecting the outcome of TDD*
with respect to the *factors' limiting effect on the usage of TDD*
from the point of view of *the software developer*
in the context of *eXtreme Programming software development.*

1.3 Context

To perform analysis with respect to the above objective, an experiment was organised as laboratory activities with master students enrolled in the Software Verification and Validation course at Mälardalen University during the autumn semester in 2010.

1.4 Paper Outline

This paper is structured according to the reporting guidelines provided in [5] (although some minor deviations from the reporting guidelines were made). In section 2 we present the related research works followed by the experimental design in section 3. Section 4 presents the details of execution of our experiment. The treatment and analysis of the collected data are given in section 5. In section 6, we present statistical inferences followed by conclusions and future research planned in section 7.

2 Related Work

Test-driven development is a practice derived from experience and without any ground theory it makes it very difficult to prove its efficiency in a formal way. This is one of the reasons why many experiments on TDD are conducted in order to provide empirical evidence of its claimed quality improvements.

In this section we present related work on empirical investigations of TDD identified in our recent systematic literature review [3], grouped w.r.t two aspects: (i) related to testing knowledge and (ii) general experiments on TDD.

2.1 TDD and testing knowledge

Sfetos et al. [6] performed an industrial survey on advantages and difficulties that software companies experienced when applying XP. Test-first was among the investigated practices. During interviews, developers identified difficulties in writing tests at the very beginning of the project.

Geras et al. [7] performed an experiment with professionals in academic environment providing subjects with two programs for development, one using test-last and one using test-first process. One of the conclusions made from the experiment is that without adequate training and having proper testing skills it is risky to adopt TDD.

Kollanus & Isomöttönen [8] analysed students perceptions and difficulties on TDD in an educational context experiment. As part of their conclusions they present different difficulties students had when designing tests. Generally, students find it difficult to design appropriate test cases and to design tests in small steps.

2.2 Experiments in TDD

In Table 1 we present experiments in TDD selected from [3] outlining experiment environment (industrial or academic) and type of subjects (students, professionals or mixed). A brief description of the aim and results of each TDD study is also presented.

3 Experimental Design

This section details the design of the experiment. Further practical experiment setup information, e.g., for replication purposes, can be found at the first author's webpage[1].

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

AUTHORS	YEAR	EXPERIMENT SETTINGS	SUBJECTS
Müller & Hagner [9]	2002	Academic	Students
AIM: To evaluate benefits of test-first programming compared to traditional approach. RESULTS: Test-first does not accelerate programming, produced programs are not more reliable but test-first supports better understanding of program.			
George & Williams [10]	2003	Industrial	Professionals
AIM: To evaluate quality improvements of test-driven development compared to a waterfall-like approach. RESULTS: Test-driven development produces higher quality code with the tendency of developers spending more time on coding.			
Geras et al. [7]	2004	Academic	Professionals
AIM: To investigate developer productivity and software quality when comparing test-driven and traditional development approaches. RESULTS: There were little or no differences in developer productivity but frequency of unplanned test failure was lower for test-driven development.			
Erdogmus et al. [11]	2005	Academic	Students
AIM: To evaluate functional tests in test-driven development when compared to traditional test-last approach. RESULTS: Test-first students created on an average more tests and tended to be more productive. There was no significant difference in quality of produced code between two groups.			
Flohr & Schneider [12]	2006	Academic	Students
AIM: To investigate the impact of test-first compared to clasical-testing approach. RESULTS: No significant differences could be established, but students did show a preference towards test-first approach.			
Janzen & Saiedian [13]	2006	Academic	Students
AIM: To examine the effects of TDD on internal quality of software design. RESULTS: Positive correlation between productivity and TDD, but no differences in internal quality. Perception on TDD was more positive after the experiment.			
Müller & Höfer [14]	2007	Academic	Mixed
AIM: To investigate the conformance to TDD of professionals and novice TDD developers. RESULTS: Experts complied more to the rules of TDD and produced test with higher quality.			
Janzen et al. [15]	2007	Academic	Professionals
AIM: To investigate effects of TDD on internal code quality. RESULTS: Programmers' opinions on TDD improved after the experiment but internal code quality had no significant difference between test-first and test-last approach.			
Gupta & Jalote [16]	2007	Academic	Students
AIM: To evaluate the impact of TDD on designing, coding and testing when compared with traditional approach. RESULTS: TDD improves productivity and reduce overall development effort. Code quality is affected by test effort regardless of the development approach in use.			
Kollanus & Isomöttönen [8]	2008	Academic	Students
AIM: To improve understanding on TDD in educational context. RESULTS: Students expressed difficulties with following TDD approach and designing proper tests. Regardless, they believed in the claimed benefits of TDD.			
Höfer & Philipp [17]	2009	Academic	Mixed
AIM: To compare conformance to TDD of experts and novice programmers. RESULTS: Experts refactored their code more than novice programmers, but they were also significantly slower.			
Huang & Holcombe [18]	2009	Academic	Students
AIM: To investigate the effectiveness of test-first approach compared to the traditional (test-last) development. RESULTS: Test-first teams spent more time on testing than coding compared to test-last teams. There was no linear correlation between effort spent on software testing and the software external quality.			
Vu et al. [19]	2009	Academic	Students
AIM: To investigate how test-first and test-last methodologies affects internal and external quality of the software. RESULTS: Test-last team was more productive and created more tests. Students indicate preference towards test-first approach.			
Madeyski [20]	2010	Academic	Students
AIM: To investigate how Test-first programming can impact branch coverage and mutation score indicator. RESULTS: The benefits of the Test-first practice can be considered minor in the specific context of this experiment.			

Table 1. Research publications on experiments in TDD

3.1 Goals, Hypotheses, Parameters, and Variables

The goal of the experiment was to test the effect of knowledge in software testing on *development speed*, *artefact quality* and *developer perception* when using TDD. In order to do so, the following null and alternative hypotheses were formulated:

- **Development Speed:**
  - $\mathbf{H}^{s}{}_{0}$. When using TDD, there is no significant difference between the development speed of developers with or without knowledge in software testing.
  - $\mathbf{H}^{s}{}_{a}$. When using TDD, developers with knowledge in software testing develop faster.
- **Artefact Quality:**
  - $\mathbf{H}^{q}{}_{0}$. When using TDD, there is no significant difference between the quality of the artefacts produced by developers with or without knowledge in software testing.
  - $\mathbf{H}^{q}{}_{a}$. When using TDD, developers with knowledge in software testing produce artefacts of a higher quality.
- **Developer Perception:**
  - $\mathbf{H}^{p}{}_{0}$. There is no significant difference in the perception of TDD between developers with or without knowledge in software testing.
  - $\mathbf{H}^{p}{}_{a}$. Developers with knowledge in software testing have higher preference towards TDD than those without knowledge in software testing.

The *development speed*, *artefact quality* and *developer perception* are operationalized in a list of response variables, provided in Table 2.

Construct	**Variable name**	**Description**	**Scale type**
Development Speed	User Stories	Number of user stories finished within lab session.	Ratio
Artefact Quality	Defects	Number of defects found in code implementation by independent test suite.	Ratio
Artefact Quality	Coverage	Statement coverage of test suite when applied to code implementation.	Ratio
Artefact Quality	Complexity	Cyclomatic complexity of the code implementation.	Ratio
Developer Perception	Ease of use	The ease of use with which the steps of TDD could be followed.	Ordinal
Developer Perception	Preference	Subjects' perception of TDD.	Ordinal

Table 2. Experiment Response Variables

In this experiment, the factor of *knowledge in software testing* is operationalized using a 10-weeks half-time advanced-level academic course in Software Verification and Validiation. Some topics that are covered by course are: introduction to software testing and testing fundamentals, the test processes, how to practically write test cases, code inspection and security testing, test design techniques, static program analysis and real-time testing. The course content has been inspired partly by industrial certification courses (e.g., the International Software

Testing Qualification Board (ISTQB) foundation- and advanced-level certification courses [21]), and partly by scientific courses and syllabi (e.g., the software testing course contents proposed by Ammann and Offutt [22]). For the purpose of this experiment, a subject is said to have knowledge in software testing if (s)he has taken part in the course lectures and exercises, and not to have knowledge in software testing if (s)he has not.

3.2 Experiment Design

The experiment design is detailed in Figure 1. Two groups of subjects (Group A and Group B) worked on two different problems (Problem 1 and Problem 2) as part of the labs, once before and once after the course (using TDD on both the occasions). During both the labs they used the Eclipse [23] integrated development environment (IDE) to create *working* software solutions in the Java programming language and the jUnit [24] testing framework for writing executable tests. Upon completion of each of the labs, the subjects answered a set of questions in an online survey system.

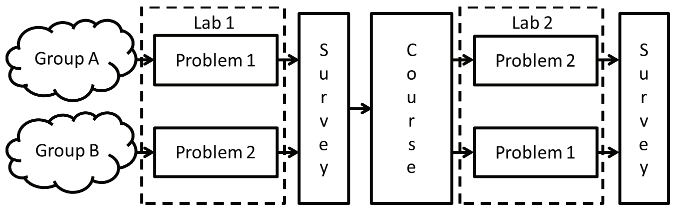

Fig. 1. Design of Experiment.

3.3 Subjects

The subjects of the experiment were software engineering master students enrolled in the Software Verification and Validation course at Mälardalen University during the autumn semester of 2010. The experiment was part of the laboratory work within the V&V course, and the subjects earned credits for participation. Students were informed that the final grade for the course will be obtained from the written exam and their performance during labs would not affect their grades.

3.4 Objects

As stated above, the experiment used two specific software development problems for the experiment, namely: (i) Roman numeral conversion (Problem 1) and (ii) a bowling game score calculator (Problem 2). The specifications for Problem 1 were written by us (in the form of a list of user stories) for the purpose of this experiment, whereas the specifications for Problem 2 (also a list of user stories) were based on the Bowling Game Kata (i.e., the problem also used by Kollanus and Isomöttönen to explain TDD [8]). Detailed information about the problems and their user stories are provided on first author's webpage[2].

TDD Steps:

1. Write one single test-case
2. Run this test-case. If it fails continue with step 3. If the test-case succeeds, continue with step 1.
3. Implement the minimal code to make the test-case run
4. Run the test-case again. If it fails again, continue with step 3. If the test-case succeeds, continue with step 5.
5. Refactor the implementation to achieve the simplest design possible.
6. Run the test-case again, to verify that the refactored implementation still succeeds the test-case. If it fails, continue with step 5. If the test-case succeeds, continue with step 1, if there are still requirements left in the specification.

Fig. 2. TDD steps for development.

3.5 Instrumentation

As one way of ensuring that subjects properly followed the steps of TDD, we provided the instructions for TDD prescribed by Flohr and Schneider [12] (see Figure 2). To avoid problems with subjects' unfamiliarity of jUnit testing framework and/or Eclipse IDE, subjects were given an Eclipse project code skeleton with one simple test case. Since this was all located in a subversion (SVN) repository, an instruction on how to obtain code from SVN and import it in Eclipse was also provided to students.

3.6 Data Collection Procedure

Teams were instructed to upload their source codes in a SVN repository. This way the lab instructor has a complete log of subjects' activities and the option to obtain code from a specific point in time.

Subjects answered survey questions using quiz assignments in the Blackboard learning management system for the course. Data from surveys is then exported in comma separated values (.csv) file format.

4 Execution

4.1 Sample

Twenty-eight students participated in the experiment. Students were informed that their work in computer laboratory would be used for the experiment, but

[2] `http://www.mrtc.mdh.se/~acc01/tddexperiment`

they were not provided any details on the goal of the experiment itself. Also, we explicitly stated that their performance would not influence the final grade of the V&V course in any way. The final grade was determined by the written exam.

4.2 Preparation

Team numbers were assigned in sequential order based on the time of receipt of the e-mail requested by the lab instructor. Problems for the teams were assigned in an alternating manner between the two immediate teams (ex., if team i was assigned problem 1, one team i+1 was assigned problem 2 and team i+2 was assigned problem 1 again etc.).

Since the lab work was time-boxed to 3 hours, a Java code skeleton was created for students. It contained a program class with one empty method returning zero and a test class with one assert statement validating the previous mentioned method. This skeleton was made to be directly imported into Eclipse as an existing project.

For each team a corresponding subversion (SVN) repository was created with read/write permissions assigned only to students within the given team and to the lab instructor. To avoid difficulties in setting up SVN and importing project in Eclipse, an instruction on the usage of SVN and Eclipse was provided to the students.

4.3 Data Collection Performed

As explained to students in the lab instruction document, after creating a new test or after changing code in order to pass the existing tests, an SVN commit command had to be executed. This way the lab instructor had a complete log of activities during the lab and an ability to obtain source code of the team at any given point in time. The absence of some students from any of the lab sessions were clearly visible from their SVN repository since the date of source code was not the same as the date of the lab. Such data was excluded from the analysis.

5 Analysis

5.1 Descriptive Statistics

Based on initial experimental plan of response variables (see Table 2) a descriptive analysis was performed for each variable independently.

First, considering the development speed construct, Figure 3 presents the percentage of user stories finished during the experiment sessions as mean values with standard error deviation. As the figure shows, the development speed was relatively unaffected in both groups before and after the course.

Second, considering the artefact quality construct, Figures 4, 5, and 6 present percentage of statement coverage of students test suite, cyclomatic complexity

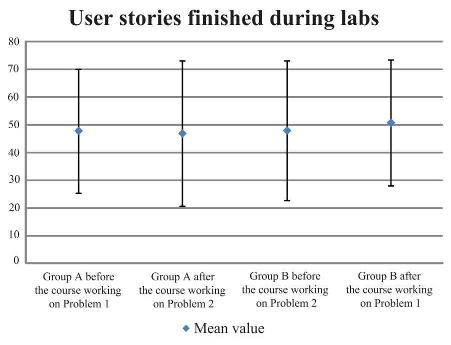

Fig. 3. Performance mean values

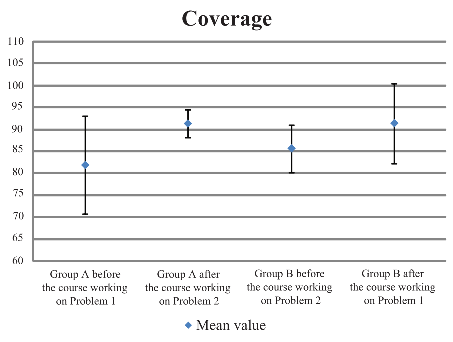

Fig. 4. Code coverage mean values

of the code, and the number of defects detected by an independent test suite respectively. These measures are given as mean values with standard error deviations. In the case of code coverage, it can been seen that both post-test groups had better mean values than the pre-test groups. In the complexity and defects metrics, the differences between the experiment objects seem to obscure such visible results, if they exist.

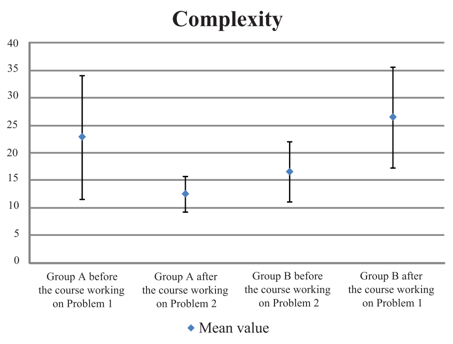

Fig. 5. Code complexity mean values

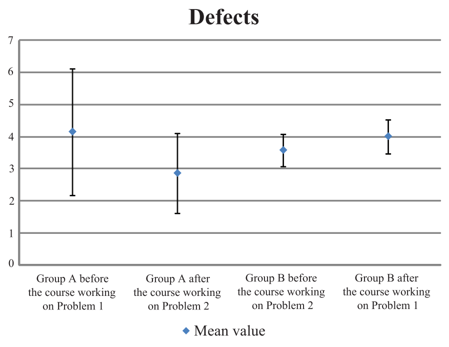

Fig. 6. Defects found mean values

Finally, Figures 7 and 8 provide results related to the developer perception construct. The first of these figures presents the sum of student responses on the ease of use with which the steps of TDD are followed in labs. Possible responses vary from 1 to 8 where 1 means impossible to follow and 8 means it was straightforward. Data is presented for both instances of labs. Figure 8 presents the sum of student responses on the perception of TDD. Possible responses varies from 1 to 8 where 1 means they will not consider using TDD in future developments and 8 means they will always use TDD. Data is presented for both instances of labs. Generally, students found TDD to be a preferable development method that is easy to use. However, there is no obvious difference between the pre-experiment and post-experiment perceptions on this matter.

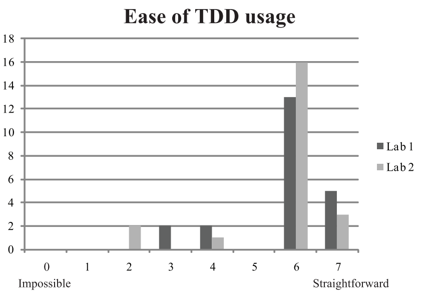

Fig. 7. How difficult was it to follow TDD

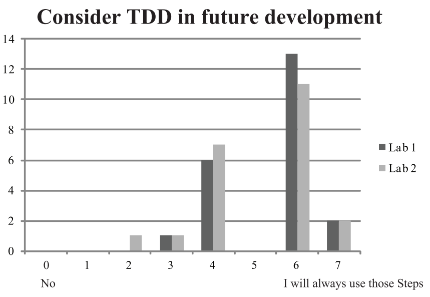

Fig. 8. Students perception of TDD

5.2 Data Set Reduction

Source codes of 17 teams (9 from Group A and 8 from Group B) and 28 student responses in survey questionnaires were collected for analysis. The difference of 6 students were due to the fact that some students did not fill in the questionnaire but did perform the lab.

When the actual source code analysis was performed additional data points had to be removed. The projects of teams 4 and 13 were excluded due to several syntax errors which made the complete solution uncompilable and irrelevant for any of the analysis. During code coverage analysis a huge deviation occurred with Team 14. A detailed analysis revealed that students did not write any test cases during the lab but they subsequently submitted tests in SVN. Since this was opposite from the TDD practice stated in their lab instructions, data from this team was also excluded. After removing data from those three teams, finally we had data points from:

– 14 teams (7 from Group A and 7 from Group B) for source code analysis and
– 22 student responses for survey questionnaire analysis.

5.3 Hypothesis Testing

Hypothesis testing was performed in two steps: First, the **Mann-Whitney** non-parametric test was used to ensure that the differences in response variable data between the experiment groups and between the experiment objects were statistically nonsignificant. The α was set to 0.05, and consequently a resulting z score of more than 1.96 or less than -1.96 was required to show a significant difference between the objects or the groups.

The result of this analysis is shown in Table 3. As can be seen from the table, there were no significant differences between the experiment objects or groups, with the exception of a significant difference in object complexity. This parameter is consequently omitted from further analysis.

Second, on the basis of the nonsignificant differences between experiment objects and groups, the **Wilcoxon** signed rank test for paired nonparametric data was used in order to test the null hypotheses of the experiment. As in the Mann-Whitney case, the α was set to 0.05. The result of this analysis is shown in Table 4. For a null hypothesis to be rejected, it is required that $\boldsymbol{min(}W_+, W_-\boldsymbol{)} \leq$ **Critical** W holds. As shown in the table, none of the experiment's null hypotheses can be rejected based on the collected data.

	Development speed	Artefact quality			Developer perception	
	User Stories	Defects	Coverage	Complexity	Ease of use	Preference
Group A vs. Group B	-0.16	-0.80	-0.34	-1.36	-0.30	1.34
Roman vs. Bowling	0.02	-1.91	0.05	**-2.64**	0.19	0.09

Table 3. Mann-Whitney z scores for differences between experiment groups and objects. A significant difference in complexity between the experiment objects is found.

Construct (Null hypothesis)	Parameter	W_+	W_-	min(W_+, W_-)	Critical W
Development speed ($\mathbf{H}^s{}_0$)	User Stories	52.5	52.5	52.5	21 (14 non-zero differences)
Artefact quality ($\mathbf{H}^q{}_0$)	Defects	22.5	13.5	13.5	4 (8 non-zero differences)
Artefact quality ($\mathbf{H}^q{}_0$)	Coverage	25	80	25	21 (14 non-zero differences)
Artefact quality ($\mathbf{H}^q{}_0$)	Complexity	Not tested			
Developer perception ($\mathbf{H}^{pe}{}_0$)	Ease of use	30	25	25	8 (10 non-zero differences)
Developer perception ($\mathbf{H}^{pp}{}_0$)	Preference	30	15	15	6 (9 non-zero differences)

Table 4. Testing of null hypotheses of the experiment

6 Interpretation

6.1 Evaluation of Results and Implications

When looking at the descriptive statistics results of the code coverage variable we can notice a positive increase in performances of both the groups when comparing before and after the course results. Even though there were no statistically significant differences in code coverage values (null hypothesis could not be rejected), we think this was a borderline case. What we want to emphasise is that, on an average, the best performing group before the course was still worse than the worst group after the course:

$$\boldsymbol{max(}A, B\boldsymbol{)}_{precourse} < \ \boldsymbol{min(}A, B\boldsymbol{)}_{postcourse}$$

The level of complexity of the students program solutions changed for both groups from one lab to another, but this change had one direction for Group A and another for Group B. What we can only conclude from this data is that solutions for Problem 1 are of higher complexity than solutions for Problem 2.

We expected the number of defects variable to provide us with a direct way of evaluating the impact of testing knowledge. An independent suite of test cases for each problem was created but we could not use it to the full extent since different teams finished different numbers of user stories. Every team had on an average four bugs and in most cases those could have been found by test cases designed using a negative test design technique.

Students claimed they adhered to TDD practice during the experiment to a high extent (Figure 9). The ease of usage of TDD practice was also reported to a high extent (Figure 7) but interestingly students did not feel the same about their preference of using TDD in future development (Figure 8).

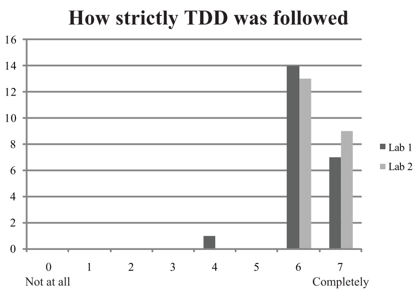

Fig. 9. How strictly TDD was followed

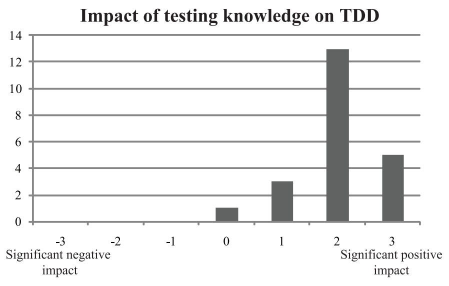

Fig. 10. Students opinion on the impact

6.2 Limitations of the Study

Typically, four types of validity are discussed in empirical research (i.e., *construct validity*, *internal validity*, *external validity* and *reliability*) [4].

Construct validity refers to the correctness in the mapping between the theoretical constructs that are to be investigated, and the actual observations of the study. Some of the constructs investigated in this study are not trivially defined, and may be subject to debate (particularly in the case of *artefact quality* and *testing knowledge*). In order to mitigate this problem, we have used standard software engineering metrics (e.g., complexity and coverage), and provided detailed information on the operationalization of each construct involved in the experiment.

Internal validity concerns the proper analysis of data. The statistical strategy used in this paper was to first eliminate the possibility of major confounding variables affecting the result (i.e., testing for differences between experiment objects or groups), and second, to test the null hypotheses. Furthermore, as the normality of the data could not be assumed, we used non-parametric tests to conduct these hypothesis tests. However, regardless of the strategy used, it is without question a fact that the sample size of the data was small, which is a major limitation for statistical analysis (and potentially also a cause for the inability for null hypothesis rejection). The only way to resolve this matter is through replications of the experiment.

External validity relates to the possibility to generalize the study results outside its scope of investigation. As many of the previously published experiments on TDD (see Table 1), this experiment is performed in a course setting and suffers from the consequent threats to external validity (e.g., *student subjects, small scale objects, short experiment duration*). It is, however, uncertain to what extent this affects the results, as we are not examining a practice (TDD) directly, but rather assessing whether the practice improves given the acquisition of a certain knowledge.

Reliability concerns the degree of certainty with which a replication of this study, e.g., by a different set of researchers, would yield the same study outcome. Here, as the experiment package and guidelines are made available for replication purposes, the major reliability threat relates to the replicated execution of the V&V course. On the other hand, without having any deeper insight as to what specific testing knowledge would be beneficial for TDD, this needs to be considered future work.

7 Conclusions and Future Work

In this section a summary of the study results with directions for future work are presented.

7.1 Relation to Existing Evidence

In the related works section we mentioned three research papers where participants of their studies expressed difficulties with testing and/or constructing test cases. Opinions of the subjects of our study pointed out that testing knowledge had a relatively significant positive impact on how they performed TDD as can be seen in Figure 10. However, based on qualitative data from our experiment, we also inferred that our respondents had problems with creating negative test cases.

7.2 Impact

A growing number of research publications empirically evaluating TDD implicitly suggest that TDD will most likely provide benefit of higher code quality to the organisation which decide to implement this development process. However, to the best of our knowledge, there are no reports on failure of implementing or adopting TDD within a specific organisation. In this context a more relevant research question could be: where and why will TDD not work and how to overcome those factors?

Our experiment is a initial attempt to address this research question from an orthogonal perspective by evaluating specifically whether testing knowledge can support TDD in practice or it could be considered as a limiting factor (as stated in [3]). Though the present study is inconclusive, it opens up several interesting challenges for the research community. We believe that identifying specific testing knowledge, which is complementary to the testing skills of a new TDD developer, is essential. Such a knowledge would enable developers to achieve performance efficiency and higher quality of software products. Additionally, it will have a great impact on the industrial adoption of TDD.

7.3 Future Work

In this study we presented a detailed experiment with students as subjects, making it more accessible for other researchers to replicate or perform a similar experiment. Alongside of providing more evidence on how general testing knowledge supports TDD in practice, we think an evolving experiment should be created with more specific focus. This experiment would be a possibility to directly investigate the effect of knowledge of negative testing on TDD practice. It could be designed in a way to provide education to subjects specifically on how to design test cases for unspecified system behaviours and use that knowledge when performing TDD of software systems.

TDD per se provides an excellent opportunity for improving code quality by imbibing "test culture" in the development community. Adherence to TDD results in the generation of automated and executable test cases during the development phase itself, thus improving the testability of the system requirements. However, as indicated by our study, TDD needs to be supplemented with new process steps or test design techniques, which could potentially further enhance the robustness and the reliability of the system.

In a long term research perspective, we also intent to perform an industrial case study investigating how experienced developers could benefit from testing knowledge and what kind of specific testing knowledge they need in order to increase the quality of the code artefacts they produce.

Acknowledgments

This work was supported by SWELL (Swedish software Verification & Validation ExceLLence) research school and OPEN-SME research project.

References

1. Beck, K.: Extreme programming explained: embrace change. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
2. Causevic, A., Sundmark, D., Punnekkat, S.: An industrial survey on contemporary aspects of software testing. In: Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation. ICST (2010) 393–401
3. Causevic, A., Sundmark, D., Punnekkat, S.: Factors limiting industrial adoption of test driven development: A systematic review. In: Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation. ICST (2011) 337–346
4. Wohlin, C., Runesson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering – An Introduction. Kluwer Academic Publishers (2000)
5. Jedlitschka, A., Pfahl, D.: Reporting guidelines for controlled experiments in software engineering. In et al., R.J., ed.: Proceedings of the 4th International Symposium on Empirical Software Engineering (ISESE 2005), IEEE Computer Society (2005) 94–104
6. Sfetsos, P., Angelis, L., Stamelos, I.: Investigating the extreme programming system - an empirical study. Empirical Software Engineering **11** (2006) 269–301

7. Geras, A., Smith, M., Miller, J.: A prototype empirical evaluation of test driven development. In: Proceedings of the Software Metrics, 10th International Symposium, Washington, DC, USA, IEEE Computer Society (2004) 405–416
8. Kollanus, S., Isomöttönen, V.: Understanding tdd in academic environment: experiences from two experiments. In: Proceedings of the 8th International Conference on Computing Education Research. Koli '08, New York, NY, USA, ACM (2008) 25–31
9. Muller, M., Hagner, O.: Experiment about test-first programming. Software, IEE Proceedings - **149**(5) (October 2002) 131 – 136
10. George, B., Williams, L.: A structured experiment of test-driven development. Information and Software Technology **46**(5) (2003) 337 – 342
11. Erdogmus, H., Morisio, M., Torchiano, M.: On the effectiveness of the test-first approach to programming. IEEE Transactions on Software Engineering **31** (2005) 226–237
12. T. Flohr and T. Schneider: Lessons learned from an xp experiment with students: Test-first needs more teachings. In Mnch, J., Vierimaa, M., eds.: Product-Focused Software Process Improvement. Volume 4034 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 305–318
13. Janzen, D.S., Saiedian, H.: On the influence of test-driven development on software design. Software Engineering Education and Training, Conference on **0** (2006) 141–148
14. Müller, M., Höfer, A.: The effect of experience on the test-driven development process. Empirical Software Engineering **12** (2007) 593–615
15. Janzen, D.S., Turner, C.S., Saiedian, H.: Empirical software engineering in industry short courses. Software Engineering Education and Training, Conference on **0** (2007) 89–96
16. Gupta, A., Jalote, P.: 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. ESEM '07, Washington, DC, USA, IEEE Computer Society (2007) 285–294
17. Höfer, A., Philipp, M.: An empirical study on the tdd conformance of novice and expert pair programmers. In Aalst, W., Mylopoulos, J., Sadeh, N.M., Shaw, M.J., Szyperski, C., Abrahamsson, P., Marchesi, M., Maurer, F., eds.: Agile Processes in Software Engineering and Extreme Programming. Volume 31 of Lecture Notes in Business Information Processing. Springer Berlin Heidelberg (2009) 33–42
18. Huang, L., Holcombe, M.: Empirical investigation towards the effectiveness of test first programming. Inf. Softw. Technol. **51** (January 2009) 182–194
19. Vu, J.H., Frojd, N., Shenkel-Therolf, C., Janzen, D.S.: 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) 229–234
20. Madeyski, L.: The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. Inf. Softw. Technol. **52** (February 2010) 169–184
21. The International Software Testing Qualifications Board (ISTQB). `http://www.istqb.org`
22. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge, UK (2008) ISBN 0-52188-038-1.
23. Eclipse. `http://www.eclipse.org`
24. jUnit Framework. `http://www.junit.org`