A Strategy to Combine Test-Driven Development and Test Criteria to Improve Learning of Programming Skills

Bruno Henrique Pachulski Camara
Federal University of Technology - Paraná
PPGI, Cornélio Procópio, PR, Brasil
Faculdade Integrado, Campo Mourão, PR, Brazil
brunohenrique@grupointegrado.br

Marco Aurélio Graciotto Silva
Departamento Acadêmico de Computação
Federal University of Technology - Paraná
Campo Mourão, PR, Brazil
magsilva@utfpr.edu.br

ABSTRACT

Students usually have difficulties assimilating basic contents of introductory programming disciplines. Test-Driven Development (TDD) is an interesting approach to address this issue, but it does not focus on quality with respect to software testing. This study aims to verify the effectiveness of the use of software testing criteria for producing test cases with higher quality in the context of education with TDD. We used the test-driven development technique with a change in the refactoring step, adding an activity for creating test cases using software testing criteria. We performed two experimental studies to evaluate the effectiveness of the technique. Each study comprised two groups: one using traditional TDD and the other using TDD with test criteria, both having developed the same software. The results showed that groups applying the proposed intervention produced better test cases, with greater coverage, and not disrupting TDD. Moreover, we could observe that, in general, the test set for groups using TDD with criteria was more efficient, getting greater coverage with fewer test cases than groups using plain TDD.

Keywords

test-driven development; TDD; software testing; structural testing; computing education

1. INTRODUCTION

Reflection in action fosters learning with proper comprehension of problems [10], avoiding the usual acting on trial and error [10, 18]. Instead, doubts found on learning activities are addressed with micro-experiments, whose hypotheses are carefully studied before trying to solve them. In the introduction to programming, techniques relying on software testing or test-driven development support reflection in action, yet promoting incremental software development and early detection of defects [8, 10, 17].

On one hand, several studies report that requiring software testing techniques for programming courses allow students to produce better software, evidenced by higher test suite coverage and defects detection [4, 25, 26]. On the other hand, there are plenty of approaches using TDD for education: development of software for different (and more motivating) domains, such as Web [23], games [16]; intelligent tutoring systems [14], automatic assessment mechanisms [3, 9, 10, 27], and others. However, few studies consider integrating both. Software testing is often used to grade assignments, considering test cases coverage [8, 11]. Moreover, the explicit use of software testing techniques is not considered when designing test cases, even when using TDD.

In fact, TDD does not focus on applying software test techniques and creating effective test sets with respect to some test criteria. Recently, some studies focused on integrating TDD and software testing, evaluating improvements on the test set quality (coverage), time required for development and overall source code quality [6, 7]. Nonetheless, this has not been investigated in the educational context, thus devising a strategy that could be successfully used by students.

Considering this scenario, our study establishes an approach for integrating test-driven development with software testing techniques. Our goal is to improve reflection in action by using criteria when developing test cases. Thus, we have changed the TDD workflow, adding an activity to create test cases based upon control-flow test criteria. This action is complementary to test cases created at the beginning of TDD, avoiding major changes to the usual TDD workflow.

The impact of this change and the adoption of basic structural test criteria are intentional, reducing the need for introducing concepts that could be an obstacle for students. At the same time, we expect the approach to guide students in creating better test sets, avoiding *ad hoc* approaches and improving the reflection in action.

The remainder of this paper is organized as follows. We provide the main concepts regarding TDD and test criteria required by our approach on Section 2. We evaluated it by an experimental study with students from a graduation course on Information Systems, as described on Section 3. The students were organized in two groups, allowing the comparison between traditional TDD and TDD with test criteria. We could observe that students who employed TDD with test criteria produced software with better internal quality, *i.e.*, test sets that covered most test requirements, yet with similar test set size, as reported on Section 4. Threats to the validity of our study are analyzed on Section 5. We compared our study with related work on TDD with test criteria on Section 6, followed by conclusions and

SIGCSE '16, March 2–5, 2016, Memphis, TN, USA.
© 2016 ACM. ISBN 978-1-4503-3685-7/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2839509.2844633

considerations for further evaluation of the current strategy and improvements on Section 7.

2. TEST-DRIVEN DEVELOPMENT WITH TEST CRITERIA

Test-driven development (TDD) is an iterative software design technique organized in three stages [2, p. 9]. The first one requires the definition of test cases regarding the functionality to be implemented in the iteration. In the software testing community, this stage is known as test-first. Meanwhile, this stage is nicknamed as *red* by the TDD community, as every test case created will fail. The implementation of code required to pass the test case is the goal of the second stage (*green*). The last stage comprises refactoring the code, improving the solution design without changing the functionality just implemented. After refactoring, a new iteration can be commenced, selecting new functionalities for implementation [2, p. 9].

However, TDD was created as a design technique, contributing to high cohesion, and low coupling [18]. An important rule in TDD is: “If you can’t write a test for what you are about to code, then you shouldn’t even be thinking about coding” [13]. Indeed, testability is often asserted as a driver for (requirement) quality. For TDD, such quality attribute is enforced by extensive use of unit testing, driving the creation and modification of code. Nonetheless, despite the importance of test cases, they are not created with respect to some test criteria [1, 2].

Software testing criteria aim the reduction of the input domain while determining most errors [21, p. 43]. Criteria are defined based upon testing techniques which use some information source to derive test requirements. Finally, test cases should be defined to satisfy the requirements, defining a coverage measurement. For instance, source code can be used to derive requirements regarding control-flow using structural software testing techniques. In this study, we considered two criteria:

- All-nodes or statement coverage. Every code block must be covered by at least one test case [21]. For structural testing, the application is represented as a control flow graph, whose nodes represent a sequence of statements and edges represent changes of flow between blocks. In this paper, we use only the term “statement coverage”.

- All-edges or branch coverage. Every edge must be covered by at least one test case [21]. This criterion subsumes “All-nodes”, as covering edges requires covering their source and destination nodes, but usually provides more test requirements than the former. In this paper, we use only the term “branch coverage”.

In fact, test criteria are not considered throughout TDD [2], relying upon developers’ experience to derive test cases. Empirically, adding more test cases reveals errors and improves the software quality. However, *ad hoc* test technique increases the costs of development and can negatively impact the adoption of TDD. Recently, some studies have been conducted to address this, with promising results [5, 24]. Nonetheless, the definition and evaluation of test strategies that combine test techniques, employing a test criterion of increasing strength within each stage and iteration, must be further investigated.

Moreover, if we are willing to improve education on programming and testing to foster software quality, such strategy must consider the requirements regarding students, especially those who are commencing on Computing.

Our approach considers the adoption of test techniques to derive test cases, guiding students on when and which test criteria should be applied. Considering the usual TDD cycle, depicted on Figure 1, we supplemented the refactoring phase with activities related to the definition of new test cases, improving the coverage with respect to control-flow test criteria, step-wisely increasing, after each iteration, the criterion strength.

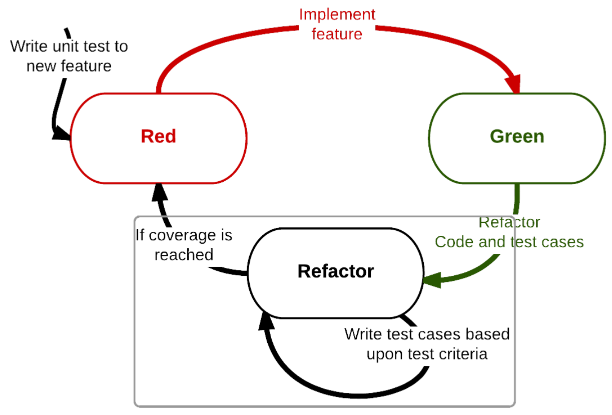
Write unit test to new feature
Implement feature
Red
Green
If coverage is reached
Refactor
Refactor Code and test cases
Write test cases based upon test criteria

Figure 1: Workflow for TDD augmented with test criteria.

The incremental strength of adopted test criteria not only fosters quality, but also supports scaffolding, as students learn simpler test techniques and improve their design and, after satisfying the test criterion, can use stronger ones. Thus, such strategy allies incremental quality improvement, provided by rigorous software testing, with development of design abilities, which is desirable for students.

3. METHOD

Considering the efficacy of test-driven development in the academic and enterprise settings, and promising results on adopting test techniques into TDD [5, 24], we defined a strategy guided by software test criteria applied to TDD aimed at undergraduate students. Thus, we proposed a change to the usual TDD workflow, adding an activity to create test cases while refactoring. The test set must satisfy a given coverage with respect to a test criterion, otherwise the refactoring activity cannot be finished and a new TDD iteration cannot start.

The research hypothesis is that the quality of the software using TDD can be improved with the adoption of test techniques and test criteria. Moreover, we want to evaluate whether such intervention might impair TDD.

The strategy is targeted toward undergraduate students with basic programming skill, but who do not practice test-driven development and are not familiar with software testing. Our technique, by combining TDD and simple test criteria, provides a scaffolding approach to develop skills that can significantly improve the quality of the software developed by the students, and whose results can endure throughout their academic and professional life.

In this study, we defined a strategy based upon structural testing and unit testing. We have changed the activities developed at the refactoring stage: the student should, besides refactoring, create test cases that cover test requirements with respect to statement criteria, and, yet in the same iteration, reach most requirements for branch coverage. The group could start a new TDD iteration only when both coverage thresholds were reached.

The strategy was evaluated with a controlled experiment [22]. We controlled the problem to be solved by the students (Bowling Game), the programming language (Java), test framework to be used when defining test cases (JUnit), the environment for development (Eclipse) and submission of source code for evaluation. For each subject (student) of this study, we established its profile, identifying previous experience on programming, TDD and software testing. The subjects were organized in two groups:

- "Group A" (control). Subjects that will implement software requirements (functionalities) using plain TDD.
- "Group B" (intervention). Subjects that will implement software requirements using TDD based upon test criteria. Besides implementing at least one test case for each functionality, every subject must develop a test set, in the refactoring stage, that satisfies some test criteria, as defined in our strategy.

At the beginning of the experiment, every student was trained on TDD, basic concepts on software testing, and control-flow test criteria. After training, we evaluated whether the students correctly understood the concepts and developed the required abilities on TDD and software testing.

The subjects that succeeded were randomly assigned to groups A and B. Then, the instructor supplied the description of the software. Resembling a typical TDD workflow, the development was organized in iterations. As suggested by Shelton *et al.* [24], each iteration comprised of a single functionality to be implemented. The description of a new functionality was delivered to the group only after finishing the current iteration. This approach is similar to a typical software development environment and conceals information from future iterations that could be a confounding factor.

Both groups were assigned to the same project and, for each iteration, they were given the same functionality to implement. The groups did not know they were using different TDD strategies, and could not interact with one another, avoiding influences between groups.

For each iteration and software delivered at its end, we collected the source code of the application and test cases, and processed related data, such as date and time of the end of the iteration. Using these data, we could extract information considered relevant to analyze TDD experiments: size of the application and test cases (number of line of codes), internal quality (statement and branch coverages) [20].

4. RESULTS

Considering the method described in the previous section, we conducted two experiments with students from a graduation program on information systems. The first trial took place on April 2014 with a cohort of four students, and the second trial was on November 2014 with a cohort of sixteen students. The organization into groups, training and analysis of the results are discussed in the following subsections.

4.1 Grouping and training

The subjects of the first trial were four students of the second-to-last term of the course. Overall, they have similar grades for Mathematics and Computing courses, easing the provision of balanced groups. Thus, the subjects were randomly assigned to groups A (control) and B (intervention).

In this trial, training was initiated with video-lectures on TDD, demonstrating traditional software development, and development using TDD, highlighting the benefits of the latter technique. Along the lecture, the instructor developed an example, which was also implemented by the students. Afterward, another activity was proposed, exercising the concepts just learned. Finally, the instructor evaluated the results of the exercises, which he considered satisfactory.

The subjects of the second trial were sixteen students of the last term of the course. Differently from the first trial, instead of a video, the professor taught TDD as usual. After lecturing, the students were assigned six activities on TDD, which were evaluated by the instructor. Based on this evaluation, four students were excluded from the trial, as their proficiency on TDD was inadequate.

4.2 Activity and intervention

The software developed in both trials was "Bowling Game" [19], an application commonly used in studies about TDD [5, 6, 12]. This application is expected to provide the score of a player in a bowling game using a set of rules regarding tries and knocked down pins. In our experiment, we considered a subset of its functionality, organizing it in three software development iterations:

- First iteration: The game has ten tries and, for each try, the player can roll the ball twice. Each roll can achieve at most ten points (number of pins knocked down). The score should be provided at any moment of the game (before or after every roll).
- Second iteration: If the player gets ten points in the first roll of a try, he gets a strike. The player cannot play the second roll of the try, but the current try will be awarded the points of the next one.
- Third iteration: The player gets two extra rolls if he gets a strike on the tenth try. If the player gets a spare (knock down all pins with two rolls) in the tenth try, he gets one extra roll.

For each iteration, Group A used traditional TDD and Group B employed TDD with test criteria to develop the software, as defined in Figure 1. For our experiment, Group B created test cases with respect to statements and branches coverage while in the refactoring stage. Only after the students noticed they had covered most statements or branches (in our study, this represents at least a 90% coverage), the intervention group could start a new iteration.

After each iteration, groups submitted the code to the instructor through the institution virtual learning environment. The instructor collected coverage information for each submission employing Eclemma, extracting data as shown in Table 1. The name of each group was suffixed with the trial number to ease the interpretation of the data.

4.3 Analysis

We analyzed the data for internal quality attributes: number of test cases, and coverage of test requirements.

Table 1: Results from experiments.

			Iteration 1			**Iteration 2**			**Iteration 3**		
			Statements	Branches	Test cases	Statements	Branches	Test cases	Statements	Branches	Test cases
Trial 1	Student 01	A1	93.3	54.8	5	96.7	93.3	6	95.4	71.4	7
	Student 02	A1	91.2	44.3	2	87.7	43.0	3	88.6	44.4	4
	Average	**A1**	**92.25**	**49.55**	**3.5**	**92.20**	**68.15**	**4.5**	**92.00**	**57.90**	**5.5**
	Student 09	B1	95.4	78.6	2	82.7	75.0	4	95.1	89.6	6
	Student 10	B1	98.7	100.0	2	98.9	100.0	4	98.7	100.0	4
	Average	**B1**	**97.05**	**89.30**	**2.0**	**90.80**	**87.50**	**4.0**	**96.90**	**94.80**	**5.0**
	Diff. B1 A1		**4.85**	**39.75**	**-1.5**	**-1.40**	**19.35**	**-0.5**	**4.90**	**36.90**	**-0.5**
Trial 2	Student 03	A2	77.8	33.3	1	85.2	66.7	2	82.2	68.8	3
	Student 04	A2	93.4	100.0	1	96.4	100.0	2	96.4	100.0	4
	Student 05	A2	89.4	62.5	1	79.2	77.8	1	89.0	61.8	1
	Student 06	A2	91.1	66.7	4	85.1	66.7	2	89.1	71.4	3
	Student 07	A2	80.0	50.0	2	94.9	83.3	7	95.5	83.3	10
	Student 08	A2	91.3	75.0	11	92.8	73.1	11	97.1	73.3	12
	Average	**A2**	**87.17**	**64.58**	**3.3**	**87.17**	**77.93**	**4.1**	**91.55**	**76.43**	**5.5**
	Student 11	B2	89.6	100	1	80.1	100	1	93.6	100	4
	Student 12	B2	88.2	100	2	92.9	100	5	87.1	80.6	9
	Student 13	B2	93.2	100	3	94.2	100	4	96.9	100	8
	Student 14	B2	98.2	90	3	99.4	81.2	3	99.0	88.7	3
	Student 15	B2	100	100	3	100	100	4	99.8	88.8	5
	Average	**B2**	**93.84**	**98.00**	**2.4**	**93.32**	**96.24**	**3.4**	**95.28**	**91.62**	**5.8**
	Diff. B2 A2		**6.67**	**33.42**	**-0.9**	**6.15**	**18.31**	**-0.7**	**3.73**	**15.19**	**0.3**

* Group A is control (traditional TDD) group and Group B is intervention (TDD+Test Criteria) group.

For this study, we considered statement and branch coverage for test requirements evaluation. Overall, groups using TDD with test criteria developed test sets with higher coverage with respect to statements and branches coverage and with less test cases than groups using traditional TDD.

In Table 1, considering the first trial, we can notice that, in the first iteration, Group A1 developed on average 3.5 test cases covering 92.25% statements and 49.55% branches, while Group B1 developed on average 2 test cases, covering 97.05% statements and 89.30% branches. For the next iterations, Group A1 added on average one test case per iteration. Meanwhile, Group B1 doubled their test set size (although yet smaller than the one of Group A) and, in the third iteration, they added just one test case. Concerning coverage of test requirements on the second and third iteration, Groups A1 and B1 kept similar averages for statements coverage, but the same does not apply for branch coverage: Group B1 always kept a higher average than Group A1, on a minimum of 20% difference.

For the second trial, the difference for statements and branches coverage was clearer. Group A2 achieved an average of 64.58% for branch coverage while Group B had 98.00% using fewer test cases for the first iteration. For statements coverage, the difference was smaller, but Group B2 still had a better coverage.

Based on the average checked on Table 2 of the two groups in the first iteration, group A averaged 3.38 test cases with 88.44% statement coverage and 60.83% branch coverage against an average of 2.29 test cases with 94.76% statement coverage and 95.51% branch coverage. For the second and third iterations, the difference is smaller, with both groups developing on average 5 test cases, but yet with significant difference between coverage of test requirements: 89.75% versus 92.60% regarding statement coverage, and 75.49% versus 93.74% for branch coverage. Indeed, the difference of statement coverage between groups A and B is not statistically significant. This result agrees with studies that evaluate coverage for test cases developed by students using TDD [11]. However, for branch coverage, the difference is significant at a level of 0.05 applying Mann-Whitney Test U (merging data from both trials). Merging data from all iterations, statement coverage is also statistically significant.

An expected result is the improvement of software quality due to application of test criteria without impairing TDD, despite the additional effort to create better test cases and impact of this activity in the refactoring state. So far, our results show that test criteria act as a guide to create test cases, allowing the definition of more consistent test sets with respect to number of test cases and quality (coverage). In the short-term, the effort saved by defining fewer test cases is an interesting result of the technique. On the longer term, the higher quality of the test set can contribute to reduce the effort required on specific software testing activities or enable the usage of more sophisticated test techniques, considering test criteria with better strength and comprehensive integration and system testing.

After the experiment, we surveyed the students about the use of traditional TDD and TDD with test criteria. Overall, they perceived test cases as an integral part of the software, which they could not deliver it without implementing the

Table 2: Summary of the results from the experiment.

	Statements				**Branches**				**Test cases**			
	TDD	TDD+TC	Diff.	p-value	TDD	TDD+TC	Diff.	p-value	TDD	TDD+TC	Diff.	p-value
Iteration 1	88.44	94.76	6.32	0.094	60.83	95.51	34.69	**0.006**	3.38	2.29	-1.09	1.000
Iteration 2	89.75	92.60	2.85	0.397	75.49	93.74	18.26	**0.031**	4.25	3.57	-0.68	0.860
Iteration 3	91.66	95.74	4.42	0.189	71.80	92.53	20.73	**0.014**	5.50	5.57	0.07	0.682
Iter. 1 + 2 + 3	89.95	94.37	3.85	**0.007**	69.37	93.93	24.56	**0.00009**	4.38	3.81	-0.57	0.818

* TDD is control (traditional TDD) group and TDD+TC is intervention (TDD+Test Criteria) group.

required functionalities and test cases (keeping them all "green"). With respect to TDD with criteria, the groups considered satisfying test requirements as much a goal as satisfying software requirements, *i.e.*, while the requirements for the iteration are not fulfilled, the software is not ready for delivery.

Moreover, when questioned about the efficacy of applying test criteria with TDD, the students asserted the approach had helped them to better define their test cases and check which part of the code required further testing. One student, that had previous experience on TDD, complemented that he was accustomed to *ad hoc* techniques to create test cases, while the current approach provided a better experience, establishing "guidelines" on writing test cases.

5. THREATS TO VALIDITY

An important threat is the evaluation of whether students correctly applied TDD. We tried to address this by training sessions. For instance, on the second trial, we excluded two students that could not perform TDD as expected after those training sessions.

The correct application of control-flow test criteria should be considered when analyzing results. For both trials the students did not use tools to directly evaluate the coverage with respect to the criteria in question. Despite this, we could observe that the coverage data are consistent with the hypothesis that students considered the statements and branches criteria when developing test cases. Analyzing the data, we could notice that coverage did not reach 100% mostly due to "getters" and "setters" methods created by the students with automatic assistance of the development tool, and for which they constantly forgot to design test cases. Nonetheless, future work should provide the students tools to address this threat. We are modifying an Eclipse plugin, based on Eclemma [15], which will compute this information, and automatically collect and send it for analysis by the instructor. Furthermore, we will analyze the impact of code automatically created and verify their impact on test coverage.

The subjects of our study were students from the last year of their graduation courses. This reduces the impact of lack of programming experience, improving the effectiveness of TDD. Therefore, such population do not have the same characteristics of those students from introductory computing classes, who we would also like to evaluate. That is a factor we must consider when applying the strategy for beginners (CS1 and CS2 students). The test strategy we adopted in this study may also require use of different test criteria. Thus, we are considering to evaluate functional test criteria, such as equivalence partitioning and boundary value analysis.

6. RELATED WORK

As we have discussed on Section 2, the usual focus of research related to test-driven development is improvement of software design, helping programmers to define better solutions for the scenario to be addressed [1, 2, 10]. Software testing is not an explicit goal, at least with respect to a test criterion or test phase.

Even with no explicit goal on software testing, test cases must be somehow defined. Considering this fact, Shelton *et al.* [24] evaluated test cases with respect to mutant analysis at the end of each development iteration. Their goal was to increase the quality of the test set after completing refactoring, enabling the developer to assess the mutation score and to define new test cases to improve that score. Another goal was to evaluate the impact of such activity in the TDD workflow. Their study was conducted along with three professional programmers, who were proficient on TDD and use of software testing frameworks. The results showed improvement at the quality of the test, discovering more defects after using mutation analysis. The statement coverage, originally at 91%, reached 100%, and the mutation score ranged from 40% to 96.9% [24]. Overall, the programmers were receptive to the novel TDD approach, stating that the development pace was not affected, and that they found it easier to refactor code and find defects for the application with improved test set [24].

Causevic *et al.* defined a method to improve quality of software within TDD: TDD^{HQ} (Test-Driven Development High Quality). Before starting an iteration, a quality improvement aspect should be chosen. Then the programmer selects a software testing technique that can contribute toward satisfying that aspect. At the end of the iteration, the programmer restarts the workflow, choosing between implementing a new functionality and improving another quality aspect [5]. The efficacy of the approach on defect detection was evaluated with a controlled experiment. Two students' groups were organized: one who used TDD^{HQ} and another that employed traditional TDD. The chosen quality improvement aspects were functionality and robustness, which were evaluated with equivalence partitioning test criterion. The results of the study revealed improvement of the quality of the test set developed by the TDD^{HQ} group, showing test cases that had exercised invalid input data (*bad-path*) were more effective at finding defects than those that had considered only valid input (*happy-path*), as usual in traditional TDD. However, it was not possible to rebut the hypothesis that TDD^{HQ} has better results than traditional TDD to find defects [5].

Differently from those studies, our approach was designed to help students initiating on Computing. For instance, Causevic *et al.* [5] define a flexible model that supports several quality assurance activities, but they evaluated it with graduated students (ongoing master's degree programs). Shelton *et al.* [24] adopts mutation analysis, which is a more sophisticated software testing technique, requiring more effort from the student to grasp the rationale for detection of defects and the development of reflection in action.

7. CONCLUSIONS

The hypothesis we have evaluated was whether the application of test criteria to test-driven development could improve the quality of software developed by students, and foster reflection in action. The development of test cases, by itself, contributes toward this, but, in our study, we could observe that the use of test criteria improved the quality of the test set without significant disturbance to the TDD workflow.

Likewise, in related studies the experiment was rather short, comprising just three iterations. Longer term effect of the approach should be evaluated, employing more iterations and measuring the effort for developing test cases using *ad hoc*, structural or another testing technique. Considering test criteria, can we observe the saturation effect regarding coverage and use such information to employ criterion with greater strength than control-flow ones, and, if so, can students cope with such criteria? Another question is whether there are other improvements besides test set quality with respect to coverage. For instance, can the usage of TDD with criteria contribute to even better design than traditional TDD? Does the effort for improved testing alleviate the effort for testing the software?

Our next step comprises of further evaluation of the current strategy for longer iterations, and applying functional test criteria. We will also focus on students from introductory computing courses, and evaluate more students (cohorts of up to 40 students), reducing bias concerning profiles and enabling the proper statistical evaluation of our hypothesis.

8. REFERENCES

[1] M. F. Aniche and M. A. Gerosa. How the practice of TDD influences class design in object-oriented systems: Patterns of unit tests feedback. In *26th Brazilian Symposium on Software Engineering*, pages 1–10, 2012.

[2] K. Beck. *Test-Driven Development: By Example.* Addison-Wesley Professional, USA, 1 edition, Nov. 2002.

[3] K. Buffardi and S. H. Edwards. Exploring influences on student adherence to test-driven development. In *17th Conference on Innovation and Technology in Computer Science Education*, pages 105–110, 2012.

[4] K. Buffardi and S. H. Edwards. Effective and ineffective software testing behaviors by novice programmers. In *9th Conference on International Computing Education Research*, pages 83–90, New York, NY,USA, 2013.

[5] A. Causevic, S. Punnekkat, and D. Sundmark. TDD^{HQ}: Achieving higher quality testing in test driven development. In *39th Euromicro Conference Series on Software Engineering and Advanced Applications*, pages 33–36, 2013.

[6] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark. Effects of negative testing on TDD: An industrial experiment. In *14th Conference on Agile Software Development*, pages 91–105, 2013.

[7] A. Causevic, D. Sundmark, and S. Punnekkat. Impact of test design technique knowledge on test driven development: A controlled experiment. In *13th Conference on Agile Software Development*, pages 138–152, 2012.

[8] D. M. de Souza, S. Isotani, and E. F. Barbosa. Teaching novice programmers using ProgTest. *International Journal of Knowledge and Learning*, 10(1):60–77, 2015.

[9] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *18th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 148–155, New York, NY, USA, 2003. ACM.

[10] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *35th Technical Symposium on Computer Science Education*, pages 26–30, New York, NY,USA, 2004.

[11] S. H. Edwards and Z. Shams. Do student programmers all tend to write the same software tests? In *19th Conference on Innovation and Technology in Computer Science Education*, pages 171–176, New York, NY,USA, 2014.

[12] D. Fucci and B. Turhan. On the role of tests in test-driven development: A differentiated and partial replication. *Empirical Software Engineering*, 19(2):277–302, Apr. 2014.

[13] B. George and L. Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, Apr. 2004.

[14] M. Hilton and D. S. Janzen. On teaching arrays with test-driven learning in WebIDE. In *17th Conference on Innovation and Technology in Computer Science Education*, pages 93–98, 2012.

[15] M. R. Hoffmann et al. EclEmma - Java code coverage for Eclipse. Software, Aug. 2006.

[16] V. Isomöttönen and V. Lappalainen. CSI with games and an emphasis on TDD and unit testing: piling a trend upon a trend. *Inroads*, 3(3):62–68, Sept. 2012.

[17] D. Janzen and H. Saiedian. Test-driven learning in early programming courses. In *39th Technical Symposium on Computer Science Education*, pages 532–536, 2008.

[18] D. S. Janzen and H. Saiedian. Test-driven learning: intrinsic integration of testing into the CS/SE curriculum. In *37th Technical Symposium on Computer Science Education*, pages 254–258, 2006.

[19] R. C. Martin. The Bowling Game Kata. Site: http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata, 2005.

[20] H. Munir, M. Moayyed, and K. Petersen. Considering rigor and relevance when evaluating test driven development: A systematic review. *Information and Software Technology*, 56(4):375–394, Apr. 2014.

[21] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, New York, NY,USA, 2 edition, 2004.

[22] S. L. Pfleeger. Design and analysis in software engineering: the language of case studies and formal experiments. *Software Engineering Notes*, 19(4):16–20, Oct. 1994.

[23] S. Schaub. Teaching CS1 with web applications and test-driven development. *Inroads*, 41(2):113–117, June 2009.

[24] W. Shelton, N. Li, P. Ammann, and J. Offutt. Adding criteria-based tests to test driven development. In *5th International Conference on Software Testing, Verification and Validation*, pages 878–886, 2012.

[25] D. M. d. Souza, J. C. Maldonado, and E. F. Barbosa. ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. In *24th Conference on Software Engineering Education and Training*, pages 1–10, 2011.

[26] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez. Experiences with Marmoset: designing and using an advanced submission and testing system for programming courses. In *11th Conference on Innovation and Technology in Computer Science Education*, pages 13–17, New York, NY,USA, 2006.

[27] M. Thornton, S. H. Edwards, R. P. Tan, and M. A. Pérez-Quiñones. Supporting student-written tests of GUI programs. In *39th Technical Symposium on Computer Science Education*, pages 537–541, 2008.