An Overview of the Current State of the Test-First vs. Test-Last Debate

Chris M. Thomas
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
thom3706@morris.umn.edu

ABSTRACT

When it comes to software development, perhaps one of the most important and time consuming processes is that of software testing. In fact, early studies on software testing estimated that it could consume fifty percent or more of development costs for a product. Because of this, the ability to optimize testing to reduce testing costs can be very valuable. In this paper we compare two popular methods, test-last testing, often used in waterfall software development processes, and test-first testing, often used in Agile test driven software development methods, by reviewing recent studies on the subject. In this review we discuss the possible benefits of test-first and test-last testing and possible problems with the current data comparing these two testing methods. After that, we explore other methods in test-first testing besides test driven development, such as behavior driven development, in an attempt to find a better test-first testing model. In the end we discuss our results and potential future studies to help clarify current data.

Keywords

Test Driven Development, Behavior Driven Development, Test-First, Test-Last, Agile, Waterfall, Testing, LaTeX, text tagging

1. INTRODUCTION

When it comes to software development, perhaps one of the most important and time consuming processes is software testing. In fact, some early studies on software testing estimated that it could consume fifty percent or more of the development costs for a product [2]. Because of this, software developers have become increasingly interested in attempting to optimize testing to reduce development costs.

Although there are many testing methods that exist, they can be roughly classified into two categories: test-last testing and test-first testing. Test-last testing, used mostly in process oriented or waterfall development, is a testing method where testing is done after software is written to ensure that the software is working as intended. Test-first testing, used mostly in interval or agile oriented development, is a testing method where tests are written before the software being tested is written to ensure that the code to be written meets certain requirements.

Recently, there has been much debate in the testing community about whether or not test-first or test-last testing is superior. The goal of this paper is to attempt to give an overview of the current state of this debate by analyzing current research data concerning the advantages and disadvantages of each testing method. Because research often lags behind current implementations, and the field of test-first testing is currently changing due to its relatively new implementation, this paper will also explore new test-first methods. This paper will discuss the advantages and disadvantages of test-first testing versus test-last testing and explore new test-first methodologies in an attempt to determine the applicability of each methodology.

The paper is divided into four sections. In Section 2 we discuss what software testing is and current software development models with their supporting testing methods: test-first and test-last testing. In Section 3 we will provide an analysis of the data explaining the potential advantages and disadvantages of test-first testing compared to test-last testing. In Section 4 we discuss issues of using test driven development to implement test-first testing and will go over a new test-first methodology called behavior driven development. In Section 5 we will provide conclusions and suggestions for further research in the field.

2. BACKGROUND

2.1 Software Testing

Software testing, simply defined, is a branch of software engineering that uses a series of practices meant to either identify potential malfunctions or demonstrate functionality in a software system [2]. Software testing can be as simple as running a program to see if its results look correct or can be as complex as writing code to simulate scenarios in the real world.

When comparing different testing methods there is no one standard quantitative measurement that determines which method is superior. Because of this, many different types of measurement are used to argue that one testing method is better then another. In this paper we will focus on three attributes that are commonly found throughout research: code coverage, total development time, and code correctness. These three attributes are popular because they can

UMM CSci Senior Seminar Conference, December 2013 Morris, MN.

be quantitatively measured and are considered important within the testing community.

Code coverage refers to the percentage of lines of production code that are executed when a set of tests are run. For example, if one writes a set of tests that has 70% test coverage, it means that 70% of the lines of production code were executed when the tests were run. Total development time refers to how long it takes to finish a development project. Code correctness refers to how many errors are found within code after it is considered finished. This is often measured by running a very large all encompassing test set against the participants code in the attempt to find cases where their code fails to produce the desired result.

2.2 Waterfall Development and Testing

A popular software development model that was developed in the 1970's is the waterfall software development model, where software is developed in a series of phases. The waterfall development model is popular because it is simple to implement correctly and is time efficient. The first phase in the standard waterfall model is the requirements phase, where requirements are set by the customer or design company. Next the design phase occurs in which the product is designed. The product is then built in the implementation phase. The final phase is the verification phase where testing and debugging occur [13]. The phases were set up in this manner to have previous phases make later phases simpler to complete. Due to the fact that these processes often put testing at the end of development, a certain type of testing method, test-last testing, was the only testing method that made sense to use.

Test-last testing is currently a popular testing method and is usually the first testing method that people tend to implement. Test-last testing is the practice of writing tests after code has been written to check the functionality of the written code. These tests are then used by the developer to fix their code until no further errors are found by the tests.

2.3 Agile Development and Testing

In the late 1990s, a group of software developers started to criticize the phase-oriented waterfall model, complaining that it was too brittle and inflexible to meet the demands of the most customers. In response to these critiques of the waterfall model, a new model for developing software emerged, the agile development model. This new development model, based on the tenets of the agile manifesto [3], promoted the idea that all actions of development should not occur within an ordered sequence of phases, but instead a series of time-boxed iterations where, in each iteration, developers set requirements, design, make , and test a subset of the end product's core functionality based on feedback from the previous iteration. The goal of each iteration is to produce a demonstrable sub-product to show a customer and to receive feedback on that sub-product. This process is complete when the sub-product meets all the demands of the customer and becomes the end product. Some current development practices that are considered agile are Extreme Programming and Scrum.

Due to the changes in Agile programming, test-last testing was pushed aside in favor of a different style known as test-first testing. In 2001, with the release of the agile development practice Extreme Programming, the idea of test-first testing, implemented in test driven development, started to become popular for the first time [5]. Test-first testing is the practice of writing tests before code has been written and then writing code to make the tests pass. It should be noted that since tests are written before production code, test-first testing tends to be heavily linked to development methods and thus the most common test-first models also include development elements as well.

The most well known and used test-first model is that of test driven development, or TDD for short. In the original TDD methodology, the developer uses a series of steps to develop his code. The series begins once all current tests pass, or a new project is started. When this occurs, the programmer writes a new failing test that tests the simplest functionality the programmer wishes to add to their code. Once the test has been written and the test fails, the programmer then writes the minimal amount of code to make the test pass. After this step the programmer streamlines his solution and integrates it with other parts of his code. This series of steps repeats until the code is complete [5]. In its current use in the field, TDD no longer has the uniform meaning that is described above. Instead it now refers to a loose collection of practices that roughly follow some or all of the guidelines given above. This means that although there is a specific definition for TDD, we can not assume that a study used the formal definition of TDD unless the TDD process is outlined within the study.

3. RESEARCH DATA ON TEST-FIRST VS TEST-LAST TESTING

In this section we will attempt to obtain useful comparisons between test-first and test-last testing by reviewing current research articles. We will first review three main studies. After that we will discuss the potential issues with summarizing the data given in the research articles and then draw potential useful comparisons from the data.

3.1 Data

This subsection contains summaries of important studies that will be used later in the paper to draw out important conclusions and comparisons between test-first and test-last testing. During this section we will make multiple references to test-last development methods which we will refer to as TLD.

3.1.1 Review by Kollanus

First, is a 2010 review by Kollanus [9]. In this review, Kollanus reviewed forty different experiments in scientific journals, magazines, and conference proceedings that provided empirical evidence comparing TDD to TLD. Each study was assigned to one of three categories: Controlled experiments, case studies, and others. Controlled experiment articles were articles that went over a conducted controlled experiment. Case study articles were articles that summarized received data from a group or situation over a period of time. Other articles were any articles that did not fit into the above categories and were either non-controlled experiments or surveys. Overall there were 14 controlled experiments, 14 case studies, and 12 studies defined as other.

In this review, Kollanus focuses on three different code quality measurements: external code quality, productivity, and internal code quality. External code quality, as defined by Kollanus, refers to how many errors are found in the resulting code. This definition is the same as the definition for

code correctness. Because of this, we can use External code quality as measurement of code correctness. Productivity, as defined by Kollanus, measures multiple properties related to how efficient the product code was to create. This includes the measurement of total development time. Although productivity is not strictly total development time, Kollanus wrote her conclusions in productivity based mainly on development time which suggests that the majority of productivity studies focused on total development time. This means that the data given could be useful as a rough estimate of the effectiveness of total development time. Internal code quality describes a wide set of measurements that measure code quality from a testing and development standpoint. Code coverage is one of these measurements but is only mentioned briefly in a summary of one article. Because of this, we are not able to assume that internal quality is a good estimate for code coverage. Thus, we will be ignoring the overall results of internal code quality and focus on the summary of the one study.

In the study, Kollanus concluded that there was weak support for improved external code quality in TDD methods compared to TLD methods. This conclusion was based on the fact that out of the 22 studies that focused on external code quality, only 6 studies concluded that TDD did not increase external code quality. However Kollanus is weary about this conclusion because out of the 7 controlled experiments that considered external code quality only 2 of them report an increase in code quality. This was concerning to Kollanus because the data from the controlled experiment articles are generally more accurate than the other two types of articles.

In terms of productivity, Kollanus suggests that TDD *may* be less productive. This conclusion was drawn from 23 studies where 11 of the studies claimed that TDD decreases productivity, 7 of the studies say that there was no difference in productivity, and 5 studies that say there was an increase of productivity. In this case the controlled experiment data accurately reflects these numbers as out of the 10 controlled experiments 2 of them claimed increased productivity, 4 of them claim no difference, and 4 of them claim decreased productivity. It is interesting to note that although decreased productivity was the most common result in the review, the majority of the studies in the review state that TDD does not decrease productivity.

There was one study that Kollanus mentions that considered code coverage. This mention occurs in the internal code quality section and noted that TDD improved test coverage.

3.1.2 Experiment by Lemos et al

In 2012, Lemos et. al [10] conducted a study on computer science students to see if test-first testing would significantly impact code coverage, code correctness, and/or total development time in auxiliary functions (functions with 10-200 lines of code). This study used 39 third-year computer science students knowledgeable in testing techniques. Each student took part in two 100 minute test-first training modules and were then asked to complete coding challenges over two sessions. In the first session, half the students used test-first methods while the other half used test-last methods. In the second session the students were asked to switch roles and given a different coding challenge.

Code coverage and total development time were measured as mentioned in Section 2. Code correctness was measured by running a set acceptance tests designed by the experimenter against a participant's code. Each code entry was then given a score based on how many acceptance tests passed. The scores were either 0 (all test cases fail), 0.5 (some test cases fail) or 1 (all test cases pass). This style of measuring code correctness has been used in previous studies and is called the *Functional Test Set Success Level* scale.

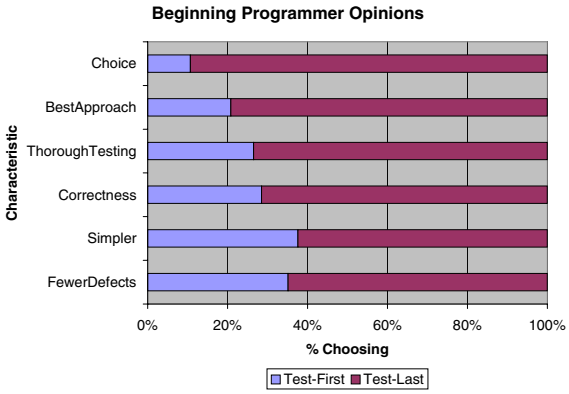

Figure 1: Results for beginner programmers in Janzen's study

It was found that code coverage on average was 40% higher when test-first testing was used. This percentage increase was found to be statistically significant. Thus, it was concluded that test-first methods produce higher test coverage then test-last methods. On total development time, it was found that test-first code took 12% longer to write then test-last code. This result was found to be statistically significant which lead to the conclusion that test-first methods took longer to implement then test-last methods. In terms of code correctness, the only difference found was that the test-last code had one more correct implementation of code then the test-first code. In the case of test last code correctness, two submissions scored a 0, thirty-three submissions scored a 0.5, and five submissions scored a 1. Test first code correctness scored slightly worse with two submissions that scored a 0, thirty-four submissions that scored a 0.5, and four submissions that scored a 1. These results were not found to be statistically significant and thus it was concluded that test first testing had no impact on code quality compared to test last testing.

3.1.3 Opinion Study by Janzen

In 2007 an opinion study was conducted by Janzen [7]. In this opinion study, Janzen polled participants from six experiments to determine whether they preferred testing with test-first methods or test-last methods. Five of these experiments were conducted on students at the University of Kansas while one of the experiments was conducted on professional programmers from a Fortune 500 company. Each participant was asked six questions where they had to choose between test-first and test-last testing. The questions were:

- which approach they would choose in the future (**Choice**)
- which approach was the best for the project(s) they completed (**BestApproach**)

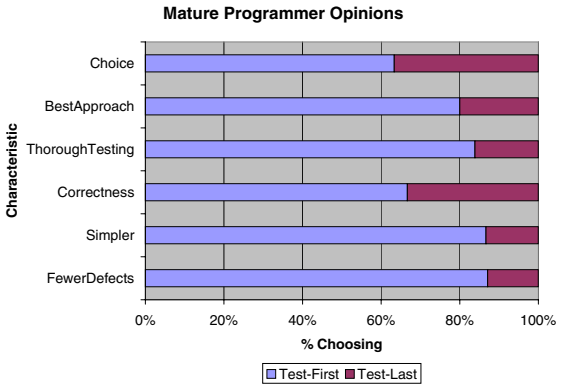

Figure 2: Results for Mature programmers in Janzens study

- which approach would cause them to more thoroughly test a program (**ThoroughTesting**)
- which approach produces a correct solution in less time (**Correctness**)
- which approach produces code that is simpler, more reusable, and more maintainable (**Simpler**)
- which approach produces code with fewer defects (**FewerDefects**)

The results were divided between a beginner group (students within their first few programming classes) and a mature group (students nearing undergraduate graduation and professionals). The results for the beginner group can be seen in Figure 1 and the results for the mature group can be seen in Figure 2. Each graph shows the distribution of all of the participants answers to the six questions where each question is referenced by its bold keyword. With the data given in the figures on the choice question, Janzen was able to conclude that beginner programming students have a statistically significant preference toward using test-last testing while more mature programming students and industry professionals showed a preference toward using test-first testing though preference was not found to be statistically significant.

3.2 Understanding Contradictions

Many summaries of TDD research [5, 6, 8, 9] have noted that studies that compare test-first to test-last testing tend to contradict one another. Kollanus points out that articles in her review contradicted other articles which made it hard to make sound conclusions from the research given, especially on the subject of productivity. The amount of contradictions concerned Kollanus enough that she noted the amount of contradictions as a potential confounding factor for her results. Contradictions also existed between our review by Kollanus and our experiment by Lemos et al. Although both articles seem to agree that code coverage is increased with test-first methods neither study seems to totally agree on the attributes of code correctness or total development time. Since contradictions occur often in the results of our study we considered it to be important to explore some current theories on why these contradictions exist. In particular we will be exploring two potential causes: method difficulty and improper study implementation/ documentation.

Method difficulty is considered a potential confounding factor when comparing test-first and test-last testing [5, 8]. In particular, there is concern that test-first testing may be more difficult for beginning coders than test-last testing. This is a concern because most study data comes from two sources with very different skill levels: industry professionals and college students. In the study by Janzen, figures 1 and 2 show very different results between beginning and mature programmers. In particular, beginner programmers did not report an overall beneficial experience with test-first testing compared mature programmers. This suggests that beginning coders may not be getting all the benefits from test-first methods that the more mature coders received. If this is the case, then any summary method with mixed results between beginning and advanced programmers has a potential to have contradictory results.

The other problem that has plagued summary papers of test-first testing is the lack documentation and conformance of TDD methods in studies. Kollanus, in her review paper, acknowledges a frustrating lack of information in many studies on how TDD was implemented. Most articles, claimed Kollanus, had perhaps one or two lines describing their TDD methodology or only claimed that they used TDD. Considering that TDD can be implemented in many ways, this lack of documentation for TDD implementation creates a problem in accurately comparing two studies. This is an issue because multiple TDD methods may not preform equally, causing potentially contradicting results. This problem could be avoided if studies documented their TDD process. Another problem with TDD studies is that very few confirm their participants correctly implemented TDD [5]. For example, in the study by Lemos et. al, the researchers acknowledged that one of their confounding factors was that they only asked students to write tests before code, thus a variety of different test-first methods might have been used within the study making its data less conclusive. Overall the lack of documentation of how researchers specifically planned to implement TDD in their study and their lack of some sort of monitoring of whether the implementation was actually occurring greatly reduces the credibility of the experiment being done and allows for contradictory results to occur.

3.3 Conclusion

Due to current issues in contradictory data it is hard to make solid conclusions about the advantages and disadvantages of using test-first testing instead of test-last testing. That being said, current trends exist within the research that allow us to guess some of the traits of test-first methods compared to test-last methods. One result which was very clear in the research was that of code coverage. The results from Lemos et al and Kollanus suggest that test-first testing tends to produce more code coverage compared to test-last testing. This seems to be a fairly uncontested conclusion as [8, 5] reached the same conclusion and no contradictions were found in other articles. Another convincing trend that has occurred is that test-first testing seems to be harder to implement than test-last testing. This trend is apparent from the results in Janzen and also from stud-

ies [1, 4] which will be discussed in the next section. Although these trends seem to be clear, the rest seem to be more muddled. Code correctness, for example, seems to have only a vague trend that agrees that test-first methods do not have worse code quality than test-last methods. The debate about whether or not test-first testing produces better code correctness though has yet to be resolved due to conflicting data. Another muddled trend is that of total development time. Currently, the only thing that can be concluded on this topic is that test-first likely takes at least as much time as test-last methods to implement though it is unclear if test-first methods take significantly longer than test-last ones.

4. DIFFICULTY AND TDD

Due to TDD's popularity, almost all test first testing studies claim to have been done using TDD. Because of this, the data for test-first testing has the potential to reflect an attribute of TDD that is not an attribute of test-first testing. Although all the results found in the previous section have the potential to contain this issue, one particular result seems to suggest it may be a attribute of TDD and not test-first testing. This attribute is the difficulty of test-first testing.

4.1 Studies noting TDD difficulty

In 2012 Hammond et al did a summary paper on the current state of TDD. In this paper Hammond et al states that "TDD remains deceptively simple to describe but deeply challenging to put into practice effectively" [5]. Hammond et al defended their claim by summarizing several studies briefly that showed the complexity of TDD. In this section we will take a more in-depth look at some of the studies Hammond et al used to draw their conclusion. We will then discuss in the next section why these studies prove that the complexity is caused by the design of TDD and not test-first testing.

One of the studies that Hammond et al summarizes is the opinion study by Janzen. In particular Hammond focuses on a quote from Janzens study: "A few of the participants of the studies found TDD either too difficult or too different from what they normally do". This quote refers to an interesting relation found between the choice and best approach category. It was noted by Janzen that in every experiment, there were more people who thought test-first was the best approach than people who would choose to implement test-first testing. When Janzen looked into why this phenomena had occurred he found multiple statements in the comment section that claimed the difficulties of TDD.

Another study that Hammond summarizes is an experiment survey by George and Williams [4]. In the study, multiple programming professionals from John Deere, RoleModel Software, and Ericsson participated in an experiment comparing TDD to waterfall development. In this study a 9 question survey was given out asking programmers what they thought about TDD and what was difficult with TDD. In this survey 56% of the professionals noted that they had difficulty adapting to the TDD mindset when participating in the study. In addition 23% of the participants noted that they felt that the lack of upfront design in TDD was more of a hindrance than a help.

The last study Hammond summarizes in this portion of the paper is an online survey done by Aniche and Gerosa [1]. In the survey, 218 TDD programmers of differing skill levels were surveyed about their TDD practices and mistakes. In the survey, it was found that TDD was not easy to follow as about 25% of the programmers admitted to frequently or always making mistakes in following the traditional steps of TDD. Two examples of these mistakes include: forgetting to clean up their code after a test passes and writing tests that are too complex for effective TDD.

4.2 Discussion

The data above shows fairly strong evidence that TDD is at least somewhat difficult to implement but none of the data above shows that test-first testing is the problem. In the first study, Janzen reveals that TDD was difficult for some people to implement. In the second study, George and Williams reveal that 56% of the participants found the TDD mindset hard to adapt while only 23% of the participants found testing before designing as a problem, suggesting that something other then test-first testing caused difficulty for at least 33% of the participants. The third study showed that programmers had trouble implementing the steps of TDD, which has nothing to do with testing before designing. With these studies in mind, difficulty in using TDD does not seem to be directly linked with test-first testing.

One potential issue with TDD that is brought up by many developers is that it fails to explain how to write good test-first tests. This is best represented in a quote from Dan North's article [11]: "While using and teaching agile practices like test-driven development (TDD) on projects in different environments, I kept coming across the same confusions and misunderstandings. Programmers wanted to know where to start, what to test and what not to test, how much to test in one go, what to call their tests, and how to understand why a test fails." In this quote Dan North expresses his and other coders frustration with TDD and its failure to specify how and what tests should be written. This problem is also mentioned in articles [8, 12]. Currently there is no empirical evidence that this could be the cause of the TDD difficulty, but it is currently suspect by knowledgeable experts in the field [12, 8, 11].

5. EVOLUTIONS OF TDD: BDD

New test-first methods are starting to appear in the agile community that contain some of TDD's main tenets but are different enough from TDD that they are starting to receive their own names and classifications. These new methods are starting to appear because many developers are not satisfied with the current implementation of TDD. In this subsection we will discuss one of these popular TDD spin-offs, behavior driven development, which was developed to combat the problem noted in the previous paragraph.

Behavior Driven Development, or BDD for short, was created by Dan North as a substitution for TDD that helps the user determine what to test and how to test it. The main difference between BDD and TDD is that TDD focuses on testing the code itself while BDD focuses more on testing a code's intended behavior. In order to do this, BDD users start by considering their code's intended functionality. They then write sentences that use their native language to define what they want their code to achieve. These sentences are then converted from the programmers native language to tests in a standard testing language using various programming tools and methods that have been developed

specifically for BDD [12, 11, 5]. This differs from TDD because the focus is no longer on testing new features but on defining what these new features should do. This shift in focus and use of native language to define problems, according the Dan North, is what makes BDD simpler to implement [11]. However, there have not been enough studies done to check this hypothesis.

6. CONCLUSION

6.1 Summary

In this paper we talked about and compared two common test methods: test-first testing and test-last testing. Within this discussion we reviewed current research in the field in an attempt to make conclusions about the advantages and disadvantages of test-first and test-last methods. In this part of the paper we were somewhat able to conclude that test-first testing increases code coverage, is more difficult to implement, has at least as much code correctness, and takes at least as long to develop as test-last testing. These conclusions must be considered weak though as we noted that studies were very contradictory to one another due to various factors including method difficulty and poor study design/documentation. We also discussed how there seems to be evidence that being harder to implement is not a problem with test-first testing but with TDD, the way test-first testing is usually implemented. We concluded by exploring a new test-first method, BDD, that had the potential to fix this issue with TDD.

6.2 Suggested Further Research

For people who are looking to increase clarity in the debate between test-first and test-last testing there are a few studies that would be quite useful in its current state. One helpful study that could be implemented is a summary study of TDD and test-first testing articles which are divided up by participant coding expertise to see if participants with different programing experience achieve differing results. Another useful study that could be implemented would be to compare the new BDD method against TDD to see if participants find BDD simpler to implement than TDD. Also any new research on the debate that has a well documented TDD method and has some way to confirm that participants followed this method would increase the amount of credible data on TDD.

7. REFERENCES

[1] M. F. Aniche and M. A. Gerosa. Most common mistakes in test-driven development practice: Results from an online survey with developers. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 469–478, Washington, DC, USA, 2010. IEEE Computer Society.

[2] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[3] K. B. et al. Manifesto for agile software development, 2013. [http://agilemanifesto.org/ ; accessed 17-November-2013].

[4] B. George and L. Williams. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 1135–1139, New York, NY, USA, 2003. ACM.

[5] S. Hammond and D. Umphress. Test driven development: the state of the practice. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE '12, pages 158–163, New York, NY, USA, 2012. ACM.

[6] T. Hellmann, A. Sharma, J. Ferreira, and F. Maurer. Agile testing: Past, present, and future – charting a systematic map of testing in agile software development. In *Agile Conference 2012*, AGILE 13, pages 55 – 63, 2012.

[7] D. S. Janzen and H. Saiedian. A leveled examination of test-driven development acceptance. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 719–722, Washington, DC, USA, 2007. IEEE Computer Society.

[8] V. Kettunen, J. Kasurinen, O. Taipale, and K. Smolander. A study on agility and testing processes in software organizations. In *PGood/Questionableroceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 231–240, New York, NY, USA, 2010. ACM.

[9] S. Kollanus. Test-driven development - still a promising approach? In *Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology*, QUATIC '10, pages 403–408, Washington, DC, USA, 2010. IEEE Computer Society.

[10] O. A. L. Lemos, F. C. Ferrari, F. F. Silveira, and A. Garcia. Development of auxiliary functions: should you be agile? An empirical assessment of pair programming and test-first programming. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 529–539, Piscataway, NJGood/Questionable, USA, 2012. IEEE Press.

[11] D. North. Introducing BDD, 2006. [http://dannorth.net/introducing-bdd/; accessed 25-November-2013].

[12] M. Soeken, R. Wille, and R. Drechsler. Assisted behavior driven development using natural language processing. In *Proceedings of the 50th international conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 269–287, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] Wikipedia. Waterfall model — wikipedia, the free encyclopedia, 2013. [Online ; accessed 17-November-2013].