The Current State of Test-First vs. Test-Last Testing Draft

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 methods, 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.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Delphi theory

Keywords

ACM proceedings, $\LaTeX$, 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 development costs for a product [?]. 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, two particular testing methods are currently very popular: 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 the software code has been written to ensure that the code works correctly. Test-first testing, used mostly in interval or agile oriented development, is a testing method where testing is done before the software code is written to ensure that the future code meets certain requirements.

this transition needs to be rewritten 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 of these testing methods. 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 where new test-first methods are going and compare them to the research given. This paper will therefore discuss the advantages and disadvantages of test first testing versus test last testing and explore new emerging test-first methodologies in an attempt to determine the applicability of each methodology.

The paper is divided into four sections. In section two of the paper we discuss what software testing is and current software development models with their supporting testing methods: test-first and test-last testing. In section three 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 four we will go over new test-first based testing methodologies and in section five we will state our 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 entails a series of practices meant to either identify potential malfunctions or demonstrate functionality in a software system [?]. Software testing can be as simple as running a program to see its results or can be as complex as writing code to simulate scenarios in the real world.

When it comes to judging the effectiveness of certain test-

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

ing methods it is important to note that there is no one standard quantitative measurement. Because of this, many different types of measurement are used in an attempt to prove that one testing methodology is better then another. Although there are many different attributes that are used for arguing the superiority of one testing method over another, we will focus on three attributes that are commonly found throughout research; code coverage, total development time, and code correctness. Code coverage refers to how many lines of code percent-wise are currently being tested. For example, if I write a test with 70% test coverage it means that 70% of the lines of code that were wrote are being tested. 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. These three attributes are popular to use because they can be quantitatively measured and are considered important within the testing community.

2.2 Waterfall Development and Testing

present tense this paragraph From as early as the 1970s, a popular software development model was the waterfall model, where software was developed in a series of phases. The waterfall development model is used because it is simple to implement correctly and is also considered time efficient because completing previous phases helps later phases become more 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 and then the product is built in the implementation phase. The final phase is the verification phase where testing and debugging occurred wikipedia waterfall model. Due to the fact that these processes often put testing at the end of development, a certain type of testing method, the test-last method, became very popular to use with the waterfall development method.

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 errors are found by the tests. /mycommentthis last sentence may no longer be needed Test last testing allows the programmer to focus on writing tests based on areas of his code that he may be concerned about.

2.3 Agile Oriented Development and Testing

In the late 1990s, some software developers started to criticize the sequence-oriented waterfall model, complaining that it was too brittle and inflexible to meet the demands of the standard customer. 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 cite manifesto here!, promoted the idea that all actions of development should not occur within an ordered sequence of phases, but in a series of time-boxed iterations where in each iteration developers set requirements, design a product, make it, and test it. The goal of each iteration is to produce a demonstrable working product to show a customer and to receive feedback on the 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 1996 might be 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 [?]. 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 occur before development 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 most used test-first model is that of test driven development, or TDD for short. In its current use in the field, the phrase TDD is mostly used as a blanket term for any sort of development practice that uses test-first testing to drive code development. That being said it is important to note that although TDD has been turned into a vague term, in this paper we will focus more on its original, specific definition. merge some of the ideas of this paragraph with the intro to the paragraph

Test driven development, as mentioned earlier is currently the most widely used test first testing and design methodology. Although in the previous cases we have been using TDD as a vague term in this section we will focus on the the original TDD methodology introduced in Extreme Programming. In the original TDD methodology, the developer repeats a step of processes to develop his code. The process begins once all current tests pass, or the new code has just been 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 with the other parts of his code. This series of steps repeats until the code is complete.

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 observing current research articles. In order to accomplish this, this section starts with a data subsection that goes over three main studies that will be used to draw our conclusions. Next, an analysis subsection summarizes the results of the studies in terms of our three main empirical measurements: code coverage, code correctness, and total development time. Afterwards a discussion subsection talks about the potential issues with summarizing the data given in the research articles and then the conclusion subsection talks about potential useful comparisons we can draw 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.

The first research summarized research article is a 2010 article by Kollanus [?]. In this article, Kollanus, using standard literary review methods, found and reviewed forty different experiments in scientific journals, magazines, and conference proceedings that provided empirical evidence about the effectiveness of using TDD compared to test-last development methods. In the review Kollanus focuses on three different measurements: external code quality, productivity, and internal code quality. In the article, Kollanus describes external code quality ahttp://start.fedoraproject.org/s the measurement of the number of passed researcher acceptance tests in a test set when run against code written by participants. Since this definition is equivalent to our definition for code-correctness this data can be used for code-correctness comparisons. In the study, Kollanus concluded that there was weak support for improved external code quality in TDD methods compared to test-last development methods. This conclusion was based on consistent evidence found in case studies that reported better external code quality in TDD methods. However, the support was considered weak because there were contradictory results found in some higher quality controlled experiments that recorded external code quality. Productivity, as defined by Kollanus, is a measurement that measures how much code one writes in comparison to time spent. Since this definition is similar to that of total development time's, this data can tentatively be used for total development time comparisons. The research viewed by Kollanus heavily suggests that TDD methods are less productive and therefore take more time overall to write. This conclusion was drawn from the fact that the majority of studies and experiments showed a decrease in productivity. That being said multiple studies seem to contradict the majority. Internal code quality describes a wide set of measurements that measure code quality from a development standpoint. Although code coverage is one of these measurements and there is a mention that one article found increased code coverage in TDD methods, this sections conclusions will not be discussed as it does not fit with our paper's themes.

In 2012, Hellmann et. al conducted a summary on the current status of research of TDD. Within Hellmann's background there is a mention of a summary study produced by Jeffries and Mitnik that found in general that TDD largely resulted in an increase in quality, but one study they identified showed instead that TDD resulted in a strong negative impact on quality. Additionally while they showed that TDD could reduce the amount of effort required by up to 27%, most studies found an increase in effort of up to 100%.

In 2012, Lemos et. al [?] conducted a study on third year computer science undergraduate 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) compared to test-Last testing. In this study 39 third year undergrad computer science students, knowledgeable in testing techniques and participants of two 100 minute long test first modules, were asked to complete coding challenges over two sessions. In the first session, half the students implemented testing-first testing while the other half implemented test-last testing to help solve the problem. In the second section the students were asked to switch roles and given a different coding challenge. In order to reduce domain knowledge bias amongst the coding challenges three different challenge question types were handed out at each session. Code coverage was measured by comparing the percentage of tested lines in participants code. Total development time was measured by how long it took participants to finish their code. Code correctness was measured by running acceptance tests against a participants development code and giving the code a score of 0(all test cases fail), .5(some test cases fail) or 1(all test cases pass). Note this style of measuring code correctness has been used in previous studies and is call the *Functional Test Set Success Level* scale. The results of the study were analyzed for statistical significance using the Wilcoxon/Mann-Whitney non-parametric signed-rank paired test. For the results of code coverage, it was found that code coverage on average was 40% higher when test-first testing was used. On total development time, it was found that test-first code took 12% longer to write then test-last code. Both of these results were statistically significant. In terms of code correctness, the only difference found was that the test-last code had one more correct implimentation of code then the test-first code. This result was not found to be statistically significant.

need to add two paragraphs for the studies by Janzen and Kettenun

3.2 Analysis

Two of the three studies covered the topic of code coverage. In both shttp://start.fedoraproject.org/tudies by Kollanus and Lemos et al. it was acknowledged that increased test coverage had occurred in test-first testing methods compared to their test-last counterparts. All three papers brought up total development time and noted that test-first testing tended to increase development time, although both KollanusâĂŹs study and HellmannâĂŹs study provide notable contradictory studies to this conclusion. All three studies in one way or another also brought up code correctness in the form of external code quality in Kollanus or in the form of code quality in the case of Hellmann et al. In this case the results were across the board where Lemos et al. stated that there was no code correctness difference between test first and test last whereas Kollanus said there was weak evidence that test-first testing, in the form of TDD, produces more correct code then test-last testing. Hellmann et al provided contradictory evidence backing that in most studies test-first testing has shown better code correctness then test last testing but there was a noted study where test-first testing was less code correct then test-last testing. In summary, with the studies given, test-first testing produces more code coverage and is likely to take more time to code then test-last testing but due to contradictions in the research given we can not draw a conclusion on code correctness.

3.3 Discussion

Many summaries of TDD research [?, ?, ?] , including the literary review done by Kollanus [?], have noted that it is hard to draw conclusions with the given research because that many studies that compare test-first to test-last testing are contradictory to one another. For example in [?], there is a mention of previous summary studies that have had issues with conflicting data. This is then followed by a series of examples of data that has conflicted in other studies in the past. In KollanusâĂŹs literature review [?], Kollanus points out that many articles in her study contradicted other articles which made it hard to make sound conclusions from the research given and may be a confounding factor in her study. This seems to occur because of two main factors: participant experience and improper study implementation and documentation.

The other problem that has plagued summary papers of test first testing is that most studies either lack documentation in key locations or do not effectively handle the participant conformance issue which comes up in TDD studies. In the literary review by Kollanus, she acknowledges a frustrating lack of information in many studies on how TDD was implemented. Most studies, claimed Kollanus, had maybe one or two lines describing their TDD methodology or just simply claimed that they used TDD. Considering that TDD can be implemented in many ways, as discussed earlier in the background, Kollanus states that the lack of documentation for TDD implementation means that, as a reviewer, she has no idea if the test methods used to produce the results were similar or different thus potentially confounding some of her conclusions. Another issue similar to this is the conformance issue, brought up in (cite here). The conformance issue in TDD testing refers to the fact that few researchers make sure that TDD was implemented correctly or at all by its participants. 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 research being done and allows for confounding factors to potentially occur.

In the case of participant experience there seems to be a skill gap between test first and test last testing. In an opinion study by [?] it was found that a majority college students said they were not comfortable with and had trouble with comprehending TDD, even after doing a coding exercise with it. The majority of students also said that they were much more comfortable implementing test last methods and would prefer to use test last methods in the future. Coding professionals, on the other hand, actually preferred test first programming and were much more comfortable with it than students. They also stated they felt it was relatively easy to understand while students did not. In another opinion study [?], by Kettunen et al, a common theme throughout the paper was examples of individual testimonies and research conclusions that pointed to test first testing being harder to implement than test last testing. Considering that most conducted experiments focus on college students and industry professionals, who have vast experience differences, the fact that test first testing may be harder to implement for college students compared to industry professionals may create differing results between studies. This idea is considered in [?] and [?], but no summary papers where found considering this result and comparing student participant experiment results to professional participant experiment results.

4. TEST DRIVEN DEVELOPMENT

Test-first testing in most of these studies are being done in various forms of TDD. Because of this, the data for test-first testing is biased toward any strengths or weaknesses that TDD contains. Although there are many small strengths and weaknesses in using TDD, one particular weakness of TDD has become noted directly or indirectly now in multiple studies. That weakness is best described in Hammond et. al which states that: "TDD remains deceptively simple to describe but deeply challenging to put into practice effectively" '[?].

4.1 Studies noting TDD difficulty

note this study either needs to be previously discussed or this paragraph needs to be expanded to include the study The fact that TDD is challenging to use effectively can be seen in multiple studies.

In the study by Janzen 2007, there was an interesting relation between two different measurement aspects of the opinion study. It was noted by Janzen that in every experiment, slightly less people said they would be more likely to implement TDD than people who said that TDD was a superior method to test-last methods. This means that more people thought that TDD was the better method to implement than people who said they would actually implement TDD. Janzen attributed this phenomena as a testament to the difficulty of TDD as he found in the comments section multiple survey participants mentioned that they felt that TDD was too difficult or too different from what they normally do.

In a study done by George and Williams, cite this article! multiple programming professionals from John Deere, Role-Model 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 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 then a help.

rewrite this after reviewing article, the summary given by hammond kinda sucks In an online survey done by Aniche and Gerosa [?], 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.

A conclusion paragraph would be nice here

4.2 Reasons for TDD Difficulty

The main problem that makes TDD so difficult to implement correctly is that it fails to explain the best way to implement tests to test your codes wanted functionality. This is best represented in a quote from Dan North's article cite Introducing BDD here. "While using and teaching agile practices like test-driven development (TDD) on projects in different environments, I kept coming across the same confusion 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." Another quote that reveals this difficulty can be found in [?]. "Design coordinates interacting entities. Choosing test cases that will adequately fill this role is difficult. It is true that a test case is an unambiguous requirement, but is it the correct requirement?" This difficulty causes many problems when people attempt to implement TDD and potentially makes test first testing harder to implement correctly then it needs to be.

5. EVOLUTIONS TO TDD

this section is currently a work in progress

As can been seen in the above section there is an issue where TDD can be difficult to implement correctly. Due to this issue, new methods are starting to appear in the agile community that contain some of TDD's main tenets but have shifted enough away from TDD at some fundamental level that they are starting to receive their own names and classifications. Although there are many of TDD spin-offs, for this paper we chose to focus on two more popular spin offs, acceptance test driven development and behavior driven development.

5.1 Acceptance Test Driven Development

Perhaps the closest spin off to TDD, acceptance test driven development (ATDD), follows many of the ideas and expectations that regular TDD does, like writing tests first and following and using tests to define the development code. Unlike TDD though ATDD believes that in order to produce higher quality code and better customer satisfaction customers should write or define ”acceptance” tests that must be passed before the product is considered finished. This part of ATDD is noticeably different from the TDD ideology because in the TDD ideology only the developer should writes the tests and the tests should be written from simplest to hardest. [?]

Although some people claim ATDD is a step in the right direction for TDD developers their are also many people who have complaints about ATDD. One of the major complaints is that ATDD is an ideal practice at best because customers will not take the time to create useful acceptance tests. [?] Yet others state that ATDD is very hard for both customers and developers to implement correctly because their is no effective common language between the two factions.(site new IEEE article here) Because of these varied complaints some of the promoters of ATDD have moved away from it to support a newer similar TDD spin off, Behavior Driven Development or BDD.

5.2 Behavior Driven Development

BDD is a new spin off of TDD that focuses on how to correctly implement the fundamental usefulness of TDD. In BDD the most important thing that it argues is that TDD is unsuccessful because it is vague and it focuses more on ”testing the code” then showing the codes behavior. Thus BDD argues that solving the problem is to stop thinking about code in terms of testing but to understand code in terms of behavior. Because of this shift in ideas BDD uses different testing tools, such as JBehave for java or Cucumber for Ruby [?], compared to TDD that breaks code testing down into standard verbal sentences instead of basic testing language. The argued pros for this type of language usage is that it actually reduces much of the difficulty of tdd by allowing us to define the purpose of our code in more natural terms thus removing much of the confusion about what is important to test in TDD and the other pro is that now it is easier to communicate goals and functionality of the code to non computer science majors (IE Managers, Customers, Sales Reps, etc.) which allows for higher customer satisfaction. this is just a blurb on BDD and what it is and what perceived advantages it contains, this paragraph will get reworked as it does not fit in the current paper well

6. 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. What can be done instead is we can consider certain trends in research that can suggest potential answers. Most studies and summaries seem to agree that test-first testing tends to produce more test coverage compared to test-last testing and that test first testing tends to take longer to write tests compared to test first testing. Two of the studies we looked at, the experiment by lemos and the literary review by Kollanus, point out these two trends. Also one or both trends have been noted through multiple papers [?, ?, ?] with no major contradictions found. Another convincing trend that has occurred is that in general test first testing seems to be harder to implement than test last testing as mentioned earlier. Although these trends seem to be clear, the rest seem to be more muddled. Code correctness for example seems to have no clear trend as multiple studies have reached contradictory conclusions to other studies. Another hot button topic that fails to show a trend one way or the other is the amount of time spent simply developing code as this again seems to have results across the board. From these trends we can consider that test first testing overall probably takes more time to write, increases code coverage, and is harder to implement than test-last testing and that other advantages or detriments may exist but they are unclear in current research.

7. CONCLUSION

this section is currently a work in progress

7.1 Conclusion intro

In this paper we talked about two common test methods: test-first testing, often found in waterfall development models and test-last testing, often found in agile development models. In section two we defined these test methods as well as provided background information to the reader. In section three we explored recent research into the comparison of the two different testing methods and explored potential research problems with the current data. In particular we found that data was contradictory but we still were able to tentatively conclude that test-first testing increases test coverage, increases time taken to test, and is harder to implement than test-last testing. In this section we also suggested two potential reasons for the contradictory data: different participant experience levels and poor documentation/implementation of specific test first methods. In section four we explored different implementations of test first testing.

7.2 Discussion of Implications of Section 3

stubbed for now

7.3 Discussion of Implications of Section 4

stubbed for now

7.4 Suggestion for Further Research

For people who are looking to advance the field of TDD it would be potentially very interesting to see how the new TDD spin-offs such as Behavior Driven Development and Agile Specification-Driven Development compare to that of a well documented TDD process like the one suggested for

Extreme Programming. Another useful study that could be implemented would be to compare the results of studies thing that could also help advance the field is to do research comparing the effectiveness of waterfall testing and TDD testing between groups with notably different experience levels as some research tentatively points that this may be a relevant variable.

This paragraph is a rough idea on the further research topic

conclusion brain storm

-TDD increases code test coverage

-TDD perhaps takes more skill to preform

-TDD may have more effective and less effective practices

-More study is needed.

-Better study methods should be implimented