**Risk/Reward Analysis of Test-Driven Development**

by

Susan Hammond

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 3, 2019

Keywords: Test-Driven Development, TDD, agile development

Approved by

David A. Umphress, Chair, Professor of Computer Science and Software Engineering
James Cross, Professor of Computer Science and Software Engineering
Dean Hendrix, Associate Professor of Computer Science and Software Engineering
Tung Nguyen, Assistant Professor of Computer Science and Software Engineering

## Abstract

Test Driven Development (TDD) is a software development practice advocated by practitioners of the Agile development approach. Many developers believe that TDD increases productivity and product quality, while others have claimed that it delivers cleaner, more maintainable code. Many studies have attempted to evaluate these claims through various empirical methods. Most recently, Fucci, et al. have used the Besouro tool produced by Becker et al. to measure TDD conformance over a series of closely replicated experiments. Our approach used an originally-written analysis tool run against a cumulative series of homework assignments, providing measurements for the three dimensions identified by Fucci, et al: granularity, uniformity, and sequencing [Fucci, Nov 2016], along with other characteristics associated with TDD.

In addition to Test Driven Development, our study incorporates elements of the Transformation Priority Premise (TPP) set forth by Robert Martin. It proposes that as developers write their production code to make their test code pass (the green light phase of TDD), they are transforming their code (in contrast to refactoring). His premise is that there are a finite number of transformations that a developer can choose, and that there is a priority whereby some transformations should be preferred over others. This priority, in an isolated example, has been demonstrated to produce an algorithm with better performance than a comparable solution that ignored the priority order and produced a different algorithm.

A post-development analysis tool was written to examine student submissions for both TDD and TPP conformance. Submissions were committed using a special Eclipse plug-in to inject specific comments into a local git repository. The analysis tool then used those comments as indicators of the TDD phase in the student's process and evaluated how closely students followed the prescribed recommendations.

Our findings concur with recent studies, in that we did not find quantifiable proof that following the TDD process delivers a better product than not following the process. We also find that a concise yet meaningful definition of TDD is elusive and contributes to the difficulty in drawing definitive conclusions about its value as a software development practice.

Acknowledgements

I would like to thank many of my colleagues at Faulkner University: Dr. Idongesit Mkpong-Ruffin for her listening ear and professional advice, assistance, and patient support over the years, Dr. Leah Fullman for her statistics books and advice, and Dr. Grover Plunkett as we both worked toward completing our respective dissertations. I am grateful to my advisor, Dr. David Umphress for his guidance, patience and encouragement throughout the dissertation process.

I am most appreciative to my husband, Jeff Hammond, and my three children, Kathy, Brian, and David, as they have missed my presence at home, especially over the summers. They have encouraged, prodded, loved, and supported me, and I could not have completed this without my husband's patience and support.

In honor of my parents, Lanny and Arlene Poteet, who loved and supported me, but are no longer present in this life to celebrate this with me.

Table of Contents

# LIST OF TABLES

# LIST OF FIGURES

# 1. INTRODUCTION

Test Driven Development (TDD), also referred to as test-first coding or Test Driven Design, is a practice whereby a programmer writes production code only after writing a failing automated test case [Beck 2003]. This approach offers a completely opposite view of the traditional test-last approach commonly used in software development, where production code is written according to design specifications. Then, traditionally, only after much of the production code is written does one write test code to exercise it.

In 1999, Kent Beck coined the moniker Test Driven Development (TDD) in his book *Extreme Programming Explained: Embrace Change*. Beck began using the practice in the early 90's while developing in Smalltalk. But, as Beck himself points out, the idea of writing test code before production code is not original to him. While test-first coding is easily done with modern, sophisticated Integrated Development Environments (IDE's) where one can write test and production code and get immediate feedback, the practice was in use with earlier modes of programming. Beck describes being a child and reading a programming book that described test-first coding using paper tapes, where the programmer would produce the expected paper output tape first and then write code until the actual paper tape output matched the expected paper tape output [Beck 2001]. In fact, the earliest references to development teams using a TDD-style approach are in NASA's Project Mercury in the early 1960's, which employed test-first practices in the development of the spaceship's software. [Larman & Basili 2003]

Test Driven Development has been described using general guidelines [Beck 2003, Astels 2003], but does not provide rigorous guidance for its implementation. Some of Beck's initial instructions were as ambiguous as "never write a line of functional code without a broken test case." [Beck 2001]

William Wake adapted a traffic light metaphor to give developers an idea of how the practice should work. It has been modified and adapted numerous times, but his original example followed a green, yellow, red pattern. Starting to write the test was the initial green light. When the test failed to compile because there was no production code available, the programmer was experiencing the yellow light. Once a stub was written for the production code, the test code would fail because the stub did not do anything, resulting in a red light. Once the production code was written and the test case passed, the developer would return to a green light status. [Wake 2001]

The most common implementation of the traffic light example is shortened to Red, Green, Refactor [Beck 2003]. The red light represents the failing, or possibly non-compiling, test code. Green light is the end-result of writing the minimum amount of code to make that test code pass. Refactoring is used to eliminate any code duplication or bad programming techniques that were introduced by getting to green as expeditiously as possible.

After his 2001 book introducing Extreme Programming (XP), Beck wrote a second book to provide more detail about the practice of TDD, summing it up with a five-step process [Beck 2003]:

1. Write a new test case.
2. Run all the test cases and see the new one fail.
3. Write just enough code to make the test pass.
4. Re-run the test cases and see them all pass.
5. Refactor code to remove duplication.

TDD is a fundamental practice in the Extreme Programming approach to developing software [Beck 2000], but it can be used separately from the XP methodology. In a 2010 survey of self-described Agile practitioners, 53% of the 293 respondents used what the poll referred to as 'Developer TDD' for validating software [Ambler 2010]. Kent Beck polled attendees of the 2010 Leaders of Agile webinar regarding their use of TDD, and 50% of approximately 200 respondents indicated they used TDD [Beck 2010]. In both of these polls, the respondents are pre-disposed to TDD usage because they develop software in an Agile manner.

In a more mainstream poll, Forrester reported that 3.4% of 1,298 IT professionals polled used a TDD approach to development [West & Grant 2010]. The problem with this poll is the question: "Please select the methodology that most closely reflects the development process you are currently using." TDD was listed as a methodology along with Scrum, XP, Waterfall and other development methodologies. TDD is not really a methodology, but is a practice that could be used in conjunction with Scrum or XP, so this could result in an under-reported number of TDD practitioners. Alternately, since TDD is not well defined, it is possible that some respondents may have incorrectly claimed to use TDD, resulting in over-reporting. Regardless, the numbers indicate enough interest in TDD to treat it as a serious technique.

As TDD has entered into the software development vocabulary, academics have been assessing how to incorporate it into the university curriculum. In a 2008 paper, Desai and Janzen surveyed the results of 18 previous studies trying to assess the best time to introduce TDD into the curriculum [Desai & Janzen 2008]. Five studies were conducted with graduate students, five studies used undergraduate upper-classmen, seven used freshmen and one used a mix of all groups. Overall, the results were more promising at the higher levels, but Desai and Janzen argue the concepts can and should be introduced sooner. After assessing the methods used with

the freshmen group, they concluded that the key to teaching TDD at the lower levels was to introduce and reinforce instruction early in the course and then continuously model the technique throughout the remainder of the course. They followed up the survey paper with an experiment to validate their hypothesis along with a proposal for integrating TDD into the curriculum as early as the CS1/CS2 classes [Desai & Janzen 2009]. Kollanus & Isomöttönen conducted an experiment with upperclassmen and also concluded that reinforced teaching with examples is a necessary approach, and that instruction should begin early in the curriculum. Their recommendation was to introduce it after the first programming course [Kollanus & Isomöttönen 2008].

Numerous books have been written with extensive examples in an attempt to educate developers in the practice of TDD [Beck 2003, Astels 2003, Koskela 2008, Freeman & Pryce 2010]. Despite all these elaborations, TDD remains deceptively simple to describe, but deeply challenging to put into practice. One reason for this may be found in an experiment conducted by Janzen and Saieden [2007]. They found that while some programmers saw the benefits of the test-first approach, several of the programmers had the perception that it was too difficult or too different from what they normally do. Another study found that 56% of the programmers engaged in a TDD experiment had difficulty adopting a TDD mindset [George & Williams 2004]. For 23% of these developers, the lack of upfront design was identified as a hindrance. A third study compiled answers from 218 volunteer programmers who participated in an on-line survey. In the study, programmers self-reported on how often they implemented (or in some cases deviated from) specific TDD practices. The study found that 25% of the time, programmers admitted to frequently or always making mistakes in following the traditional steps in TDD. [Aniche & Gerosa 2010] Aniche and Gerosa observe that, while the technique is

simple and requires only a few steps, the programmer must be disciplined in his approach in order to garner the benefits. When the programmer is not disciplined, he does not experience the full benefit of TDD, and as a result he may be less inclined to use the practice.

The TDD practice at a unit test level also leaves many questions unanswered. John McGregor states, "Design coordinates interacting entities. Choosing test cases that will adequately fill this role is difficult though not impossible. It is true that a test case is an unambiguous requirement, but is it the correct requirement? More robust design techniques have validation methods that ensure the correctness, completeness, and consistency of the design. How does this happen in TDD?" [Fraser et al. 2003]

After producing a book with in-depth empirical research on the subject of TDD, Madeyski wrote, "a more precise definition and description of the TF (Test First) and TL (Test Last) practices, as well as a reliable and automatic detection of possible discrepancies from the aforementioned techniques, would be a valuable direction of future research." [Madeyski 2010].

Fucci, et al. ran a series of replicated experiments regarding TDD, and his conclusion is that there is no substantive difference between writing tests first or tests last, so long as the development is performed iteratively in small steps and automated test are written [Fucci 2016].

One aspect of TDD that is not frequently mentioned in academic research is the issue of mocks and other test doubles. According to [Meszaros 2007], "A Test Double is any object or component that we install in place of the real component for the express purpose of running a test." He defines a Mock Object as "an object that replaces a real component on which the SUT (software under test) depends so that the test can verify its indirect output." He further elucidates the definition by saying it is "also an object that can act as an observation point for the indirect outputs of the SUT."

Some proponents of TDD have conflated the two practices, basically insisting that to perform TDD properly, one must use mocks while writing tests. [Fowler 2007] describes Classical and Mockist TDD practitioners in a blog post. Classical TDD practitioners most often use the actual objects in their TDD process and will only mock objects when it is difficult to use the real objects. For example, making calls to a pay service while developing an application is not practical, so providing a mock object can be a better option. Mockist TDD practitioners will use a mock to separate all dependencies between all objects in the name of making the code more testable.

In a blog post, [Hannson 2014] declared that "TDD is dead." This spurred a series of Google Hangouts hosted by ThoughtWorks [ThoughtWorks 2014] exploring the question, "Is TDD Dead?" In the Hangouts, Kent Beck, Martin Fowler, and David Heinemeier Hannson debated the usefulness of TDD in modern software development. Hannson posited that the approach taken by Mockist TDD practitioners results in test-induced damage. By seeking to isolate code from its dependencies, many levels of indirection are introduced that make the code convoluted and difficult to understand and maintain. Fowler and Beck took a more Classical TDD approach, stating that they hardly ever use mocks. [ThoughtWorks 16 May 2014]. Beck indicated that TDD was all about feedback loops for him, and that if a developer found a piece of code hard to test, then there was a problem with the code design. So, in struggling to write the next test and finding it difficult to do so, the developer must ponder his design choices to understand the difficulty. This struggle can lead the developer to find a better, more testable design. [ThoughtWorks 20 May 2014].

A total of five Hangouts occurred, with little resolution between the camps. This entire conversation once again points out the confusion over what exactly TDD is and how one uses it to produce the best outcome.

This research examines TDD benefits from a different perspective: Can TDD be a useful tool for mitigating risk in the development process?

# 2. LITERATURE REVIEW

Kollanus performed a systematic literature review of TDD research where empirical methods were employed [Kollanus 2010]. She found 40 papers that report some kind of empirical evidence on TDD as a software development method. These empirical exercises resulted in inconsistent, and often contradictory, results from one study to the next.

The studies focused on different aspects of the practice and the resulting code, including defect density, programmer productivity, and object cohesion and coupling. In many of the articles, evidence was reported on TDD concerning 1) external quality, 2) internal code quality, and/or 3) productivity. Kollanus summarized her findings with regard to these three factors by stating there is [Kollanus 2010]:

1. weak evidence of better external quality with TDD.
2. very little evidence of better internal quality with TDD.
3. moderate evidence of decreased productivity with TDD.

A confounding complication Kollanus experienced in performing the systematic review was that frequently the TDD process was very briefly or not described within the source papers, so it was very difficult to compare the results. As mentioned previously, TDD has not been rigorously defined, and if the studies do not express how it was practiced, there is no guarantee the results between studies can be accurately compared. Another possible reason for the wide variation of findings may be an inconsistent understanding or application of TDD by the people participating in each individual study [Kou et al. 2010]. These studies have a "process

compliance problem" in that there is no validation that the participants actually used the TDD practices as set forth in the individual experiments.

Approximately half of these TDD experiments were conducted with college students who had training in software development and perhaps software design. Frequently, these students were then provided a brief introduction to TDD and given a "toy" solution to develop under a specific time constraint. Müller and Höfer found that studies such as these that use novices to perform TDD are not easily generalized because of inconsistent process conformance [Müller & Höfer 2007].

Even experienced programmers do not consistently conform to the process. Examples abound in Aniche's survey of TDD programmers. For instance, the survey revealed that, on a regular or frequent basis, 33% of programmers do not begin with the simplest test, a violation of one of the most fundamental concepts of TDD [Aniche & Gerosa 2010].

Another complication in the results of previous empirical studies lies in the skill and experience of the subjects. Ward Cunningham's statement that "test-first coding is not a testing technique" [Beck 2001] illustrates a major misunderstanding of TDD. To use the technique properly, the programmer must be developer and designer at the same time. This type of task is not for every developer at every skill level. Designing and developing code being led by specific tests requires a certain degree of maturity in understanding how software interacts.

Boehm [Boehm & Turner 2004] provides a classification system for programmers with regard to their ability to perform within a particular software development environment. Of his five levels, the top two have appropriate skills to function on an Agile team, the bottom two require a much more structured environment, and the middle level can perform well on an Agile team if there are enough people from the top two tiers to provide them guidance. The

implication of Boehm's classifications is that to be successful using an Agile approach, including using TDD to write code, requires people who are relatively experienced and very talented and flexible.

Latorre [2014] explored the impact of developer experience on TDD and found that both skilled and junior developers can quickly learn to adapt to the unit testing and short programming tasks associated with TDD. Junior developers struggle with the design aspect of TDD. Their performance suffers because they must make more frequent adjustments to their design choices than do more experienced developers.

## 2.1. Related Work

### 2.1.1. Agile Specification-Driven Development

Agile Specification-Driven Development is a technique that combines both TDD and Design-by-Contract (DbC) [Ostroff et al. 2004]. DbC is a concept introduced by Bertrand Meyer [Meyer 1991] which allows a language to explicitly declare pre- and post-conditions for a method and invariants for a class. Meyer created the programming language Eiffel to support DbC [Meyer 1991b], but more mainstream languages are being extended to include DbC as well. Leavens & Cheon [2006] introduce the Java Modeling Language as a way to support DbC in Java, and Microsoft introduced their version of DbC called Code Contracts into the .NET framework beginning with Visual Studio 2008 [Ricciardi 2009].

Meyer also introduced an approach to software development called the Quality First Model [Meyer 1997]. His approach pre-dates XP by 2-3 years, yet shares many similarities with it. The Quality First Model values working code above all else, but it also takes advantage of formal methods for development and tools that allow models to generate code and vice versa.

Ostroff, et al., summarize the quality-first approach as [Ostroff et al. 2004]:

1. Write code as soon as possible, because then supporting tools immediately do syntax, type, and consistency checking.
2. Get the current unit of functionality working before starting the next. Deal with abnormal cases, e.g., violated preconditions, right away.
3. Intertwine analysis, design, and implementation.
4. Always have a working system.
5. Get cosmetics and style right.

The Quality First Model has at its core the DbC approach, and it shares some commonalities with conventional TDD practice. For instance, the emphasis in step two is to finish one unit of functionality before moving on to the next. This is also a tenet of TDD. However, a big difference is also immediately obvious in the second step because TDD asks the developer to focus on the most common case, whereas DbC expects the developer to focus on abnormal cases first.

In concept, both TDD and DbC are specification tools. Ostroff, et al., point out that the two approaches actually can be complementary. They state that using TDD early in the development cycle allows the developer to describe the functionality and formalize the collaborative specifications, while adding contracts later provides an easier way to document and enforce pre- and post-conditions [Ostroff et al. 2004].

This approach was being explored further in blogs by Matthias Jauernig [2010] and David Allen [2010]. The introduction of Code Contracts in .NET stimulated the interest in combining the use of TDD and DbC for these bloggers. Specifically, they encouraged developers who currently use TDD to incorporate the use of Code Contracts into their development practices.

### 2.1.2. Behavior-Driven Design (BDD)

BDD was first introduced by Dan North [North 2006]. The initial reason for the name shift from *Test*-Driven Development to *Behavior*-Driven Development was to alleviate the confusion of TDD as a testing method versus TDD as a design method. Astels notes, "Behavior-Driven Development is what you were doing already if you're doing Test-Driven Development very well" [Astels 2006]. By changing the developer's focus to the behavior of the code, it was posited that the developer would shift his mind-set away from validation to the design aspects of the technique.

To illustrate the need for BDD, the BddWiki provides a life-cycle to the learning and adoption of TDD [Rimmer 2010]:

1. The developer starts writing unit tests around their code using a test framework like JUnit or NUnit.
2. As the body of tests increases the developer begins to enjoy a strongly increased sense of confidence in their work.
3. At some point the developer has the insight (or is shown) that writing the tests before writing the code, helps them to focus on writing only the code that they need.
4. The developer also notices that when they return to some code that they haven't seen for a while, the tests serve to document how the code works.
5. A point of revelation occurs when the developer realizes that writing tests in this way helps them to "discover" the API to their code. TDD has now become a design process.
6. Expertise in TDD begins to dawn at the point where the developer realizes that TDD is about defining behaviour rather than testing.
7. Behaviour is about the interactions between components of the system and so the use of mocking is fundamental to advanced TDD.

Rimmer asserts that most developers, with some assistance, reach step four, but very few progress beyond it. He does not provide any empirical research to support this learning curve; it is merely based on observation and experience.

Proponents of BDD have developed new testing frameworks that change the nomenclature in order to support the approach and to get developers to think beyond the viewpoint that TDD is about testing. The rationale is based on the Sapir-Whorf theory that the language used influences one's thought [Astels 2006]. Dan North developed a Java-based framework called JBehave that removed any reference to testing and modified the language to focus on behavior. For example, instead of using the testing terminology 'assertion', the framework uses 'ensureThat'; for example 'ensureThat(actual, equalTo(expected); [jBehave 2011]. For Ruby, the framework is called rSpec, and it replaces the 'assertion' terminology with a sentence-type structure where the actual object is the subject and the assertion statement is the verb; for example, 'actual.should.equal expected' [Astels 2006].

Some key aspects of BDD include [North 2006]:

1. The word 'test' should not be included in the test name.
2. Test method names should be sentences and should begin with the word 'should', for instance, 'shouldTransferBalanceFromAccountWithSufficientFunds'.
3. Tests should not correspond one-to-one with classes, but should focus on behaviors. This makes it easier to change the structure as classes grow and need to be broken out into separate classes.

### 2.1.3. Acceptance Test-Driven Development (ATDD)

Traditional TDD focuses on the smallest test that one could possibly write [Beck 2003]. The programmer begins with the failing test case and writes code until the test case passes. Test cases are written to assist in the development of various low-level requirements, but there is no mechanism by which a developer can put the requirements in context. Approaching development from this low-level viewpoint can produce results like Madeyski experienced, where programmers' code was significantly less coupled than comparable Test-Last code, but the percentage of acceptance tests passed was not significantly impacted. [Madeyski 2010]. One

could say the code quality was better, but the code didn't necessarily meet the high-level requirements of the project.

In order to ensure that the code is actually providing the functionality that the users desire, some software engineers have been focusing on a higher level test to set the context for the overall development. Beck mentions the concept of application test-driven development [Beck 2003], whereby the users would write the tests themselves. There have since been off-shoots of TDD that focus on allowing the customer to write acceptance tests that the developers can use to see if their software is providing the correct functionality. As described above, BDD is one of those approaches. ATDD is another.

Ward Cunningham introduced the concept of FIT tables in 2002 [Wiki 2011]. The general idea is that the customers enter test data into a table using a standard word processor or spreadsheet, and the developers write a test fixture that uses the data to test the software under development [Shore 2005]. This approach has been extended with multiple variations and tools, including Fitnesse [Martin, R. et al. 2008], which allows the customer to enter test cases into a wiki and then translates the data through a FIT client and server process into standard FIT fixtures, to get a FIT environment up and running very quickly. Custom fixtures written by the development team complete the configuration [Martin, R. et al. 2008]. Other solutions include Easy Accept, which is a script interpreter and runner offering another approach to automating the creation of the acceptance tests [Sauve et al. 2006] and AutAT, which focuses on acceptance testing of web applications [Schwarz et al. 2005].

Beck objects to ATDD because tests would not be under the control of the developers. He predicts that users and/or organizations will not be willing or able to take on this

responsibility in a timely enough manner for the software development. He objects to the lag in time between test and feedback, preferring the more immediate feedback of TDD [Beck 2003].

### 2.1.4. Growing Objects, Using Tests

Freeman & Pryce [2010] advocate a combination of the developer control and immediate feedback of TDD with the functional focus of ATDD. The developer begins with a functional test case that is derived from a user story that represents a system requirement. This differs from the ATDD approaches in that the tests are written by the developers, not by the end users. It puts the onus on the developer to make sure that he understands the user stories sufficiently to create the correct test. But it addresses Beck's concern that ATDD would create delays or cause the developers to lose needed control.

The fundamental TDD cycle is envisioned to take on the activities depicted in Figure 2.1.

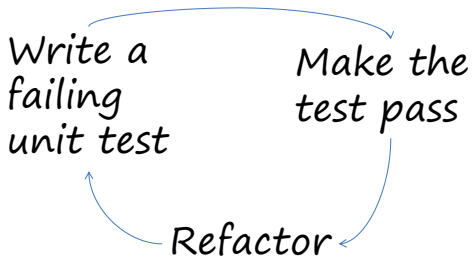



**Figure 2.1: Fundamental TDD Cycle [Freeman & Pryce 2010]**

By adding the functional test, the cycle takes the form of Figure 2.2.

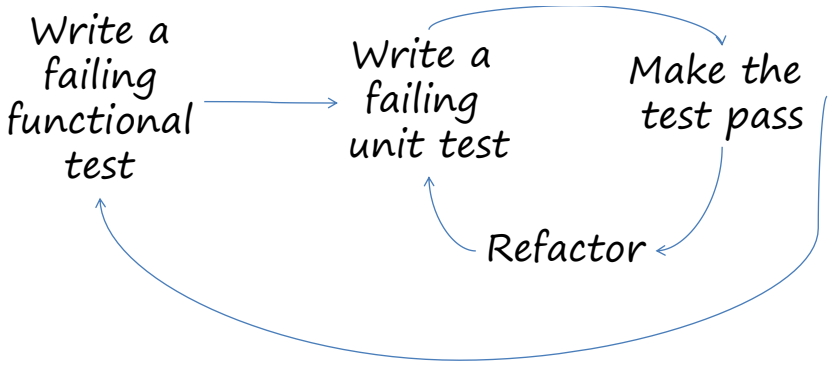



**Figure 2.2: Functional Test TDD Cycle [Freeman & Pryce 2010]**

### 2.1.5. Transformation Priority Premise

Robert Martin (colloquially known as "Uncle Bob") is a programmer and author, particularly in the realm of Agile Planning and Programming. He is a staunch proponent of the practice of Test-Driven Development, going so far as to say that "it could be considered *unprofessional* not to use it." [Martin, R. 2011 p. 83]. He has developed an approach to TDD that he calls the Transformation Priority Premise [Martin, R. 2013].

To understand this premise, the words "transformation" and "priority" must be examined in the context. Martin draws a parallel between transformations and refactorings. In the red/green/refactor cycle of TDD, a red condition indicates a failing test case, a green condition indicates all test cases currently pass, and refactoring is a situation where the structure of the code is modified or "cleaned up" without changing the behavior of the code. This "clean up" is designed to make the code more readable and/or maintainable. Martin Fowler referred to this process as removing "bad smells" in the code [Fowler 1999, p.75].

Transformations are used to bring the production code from a red condition to a green condition. They change the behavior of the code without significantly changing the structure of the code. Martin uses the TDD philosophy "As the tests get more specific, the code gets more

generic" [Martin, R. 2009]. He sees the transformations as a way to *generalize* behavior of the production code without changing the structure. His initial list of Transformations was expressed as follows [Martin, R. 2013]:

- **({}–>nil)** no code at all->code that employs nil
- **(nil->constant)**
- **(constant->constant+)** a simple constant to a more complex constant
- **(constant->scalar)** replacing a constant with a variable or an argument
- **(statement->statements)** adding more unconditional statements.
- **(unconditional->if)** splitting the execution path
- **(scalar->array)**
- **(array->container)**
- **(statement->recursion)**
- **(if->while)**
- **(expression->function)** replacing an expression with a function or algorithm
- **(variable->assignment)** replacing the value of a variable.

In a later video, the list was simplified [Martin, R. 2014]:

1. Null
2. Null to Constant
3. Constant to Variable
4. Add Computation
5. Split Flow
6. Variable to Array
7. Array to Container
8. If to While
9. Recurse
10. Iterate
11. Assign
12. Add Case

In both lists, the ordering of the Transformations is important. As one traverses the list from top to bottom, the transformations progress from lesser complexity to greater complexity.

Micah Martin wrote in his blog the following, "As I pondered these transformations, I found it simpler to think about them in terms of the resulting code, and that led me to the short list below [Martin, M. 2012]:

1. constant : a value
2. scalar : a local binding, or variable
3. invocation : calling a function/method
4. conditional : if/switch/case/cond
5. while loop : applies to for loops as well

Regardless of which list is used, Martin's Priority Premise is that programmers should opt to perform transformations to their code that are higher up on the list. When following this order, Martin hypothesizes that "better" algorithms will emerge than if transformations are selected from further down the list. In other words, given the option to use a simpler transformation or a more complex transformation, the programmer should select the simpler transformation.

This Priority Premise has implications both with transforming the code to achieve a green light state, but also with the subsequent tests to be written. As the production code is transformed to make the tests pass using higher priority transformations, so the next test written should lead the developer to choose a higher priority transformation. Martin believes that this should lead to simpler tests which should lead to simpler code.

## 2.2. Related Tools

### 2.2.1. ComTest

ComTest is an Eclipse plug-in that allows students (its intended target audience) to express test conditions in JavaDoc, which are then translated into JUnit tests [Lappalainen et al. 2010]. The rationale is that students should be introduced to the concept of testing early in their education, but the concepts used in the xUnit tools are too complex for beginners. ComTest uses a very short and simple syntax which is reminiscent of FIT tables. The benefits of ComTest are that they are shorter and easier to write than xUnit tests; they keep the test and production code together; and they serve as documentation for method usage.

While ComTest promotes itself as a TDD tool, it would be more accurately described as a primitive Design by Contract tool. Its focus is primarily on the interfaces and validating pre- and post-conditions. TDD is a practice that encourages the developer to go back and forth between test code and production code to "grow" the functionality. But ComTest encourages the developer to write out the tests in JavaDoc, generate the test code, and run it against the production code. The tests may be written first, but this does not conform to the traditional approach to TDD.

### 2.2.2. Zorro

Zorro was developed to monitor the process compliance problem. The monitoring is intended to allow researchers to determine how closely TDD was followed, and thus to determine if the results of their research reflects an accurate picture of the validity of TDD [Kou et al. 2010]. Kou provides a "precise, operational definition of TDD practice" using an "episode classification algorithm". He developed the Zorro system to assess how closely a developer is following the TDD practice based on this algorithm. The results are expressed as a percentage, for instance, "Developer X used TDD 65% of the time during this phase of development."

The "operational definition" of TDD that is given by Zorro focuses primarily on the order in which activities are performed. The Zorro software, using the traditional definition of TDD, asks questions such as, were test cases written first? Did the test cases fail prior to writing code? Was the time spent in code editing too long, suggesting that the developer has not broken down the test into a small enough "chunk"?

Zorro currently includes twenty-two episode types that are subdivided into eight categories: Test First, Refactoring, Test Last, Test Addition, Regression, Code Production, Long, and Unknown. When Zorro assesses TDD conformance, all of the Test First episodes were considered conformant, and all of the Test Last, Long and Unknown were considered non-

conformant. Within the other categories, the episodes were determined to be conformant based on the context of the episodes preceding and succeeding them. For instance, refactoring can be performed as a part of a TDD practice, but it can also be done in a Test-Last context. So, if a Refactoring episode is preceded by a Test First episode, the Refactoring is deemed TDD Conformant. If it is preceded by a Test Last episode, the Refactoring is deemed non-TDD Conformant. See Appendix B for a complete description of the Zorro classification system rules.

### 2.2.3. TDD-Guide

The TDD-Guide was developed by Oren Mishali as a component of his PhD Thesis [Mishali 2010]. The primary focus of his research was how to use aspect-oriented programming (AOP) to support the software engineering process. The TDD-Guide makes use of an AOP framework that Mishali developed and is implemented as an Eclipse IDE plug-in. His work on the TDD-Guide was intended to demonstrate his AOP framework in the development of a non-trivial software component, as opposed to providing a rigorous definition of TDD.

The TDD Guide supports a rule-based approach to implementing various software processes. TDD events and the necessary responses are defined as rules and placed in the tool's repository. A single TDD event is often defined as a series of several, more basic events. The AOP framework tracks the series of low-level events to enable the TDD-Guide to interpret the higher-level TDD event. In addition to events, timing information is tracked as well.

As an example, one TDD event was defined as *OneTestAtaTime*, which supports the TDD philosophy that a developer should only be working on one test at a time. The low-level events that would support this would be: TestCaseModified, TestCaseExecuted, and ProdCodeModified. This sequence would represent conformance to the *OneTestAtaTime* TDD event. As a negative example, a sequence of TestCaseModified, TestCaseExecuted (with a failure of more than one test case), and ProdCodeModified would represent a deviation to

*OneTestAtaTime* because there are multiple test case failures. Appropriate positive/negative responses are displayed in the Eclipse IDE based on process deviation or conformance.

While the focus of the research was not on TDD, it touched on several relevant issues. The TDD-Guide tool was developed to be very flexible: rules can be added incrementally to the repository; both positive and negative feedback can be given to the user; and the tool can either make recommendations or can strictly enforce the TDD behavior desired. Developers interact with Eclipse in the normal fashion, and only receive alerts and notifications as their behavior does or does not conform to the TDD behaviors defined by the system.

The TDD-Guide was developed iteratively, and by its last iteration, it considered four basic behaviors, noted as follows:

1. Basic TDD Cycle
    a. Complete TDD Cycle - Conformance
    b. Developer continues with coding after a cycle is ended - Deviation
    c. Developer continues with testing after a cycle is ended - Conformance
    d. Developer moves to code without executing the test - Deviation
    e. Developer moves to code after executing the test - Conformance
2. One test at a time
    a. Developer starts coding with one failing test - Conformance
    b. Developer starts coding with several failing tests - Deviation
    c. Developer creates the second consecutive unit test - Deviation
3. Simple initial cycle
    a. Developer spends too much time on the first test - Deviation
    b. Developer finishes the first cycle on time - Conformance
4. TDD coding standards
    a. Developer violates test naming convention - Deviation
    b. Developer violates code naming convention - Deviation

In comparing the TDD-Guide to Zorro, Zorro's primary focus is TDD and provides a more comprehensive analysis of the TDD behavior, while the TDD-Guide was only written to demonstrate another lower-level framework and only evaluates a limited set of TDD behaviors. Also, Zorro is strictly a background process to provide feedback to researchers, while TDD-Guide seeks to provide feedback to the developer to encourage TDD-compliant behavior.

### 2.2.4. Besouro

A subsequent stand-alone tool named Besouro was adapted from the Zorro tool [Becker 2014]. Zorro is a rules-based system that requires an underlying platform of Hackystat and SDSA (both developed and supported through the University of Hawaii) [Johnson & Paulding 2005]. Hackystat is an open-source framework used for automated collection and analysis of software engineering data based on programmer actions. SDSA is a statistical application that uses the data collected by Hackystat to provide further analysis of the collected data. Hackystat uses sensor plug-ins to the developer's IDE to collect the data. Besouro is a plug-in that uses the Zorro code as its base, but it is a stand-alone plug-in available only for the Eclipse IDE. Besouro replaces the collection functionality that Hackystat provided, eliminating the dependency on that platform.

The more significant difference between Zorro and Besouro from an analysis viewpoint is the approach to the TDD Conformance evaluation. As described in section 2.2.2, Zorro recognizes twenty-two episode types that are subdivided into eight categories. Certain episodes are definitively categorized as either Test First or Test Last. Within the categories of Refactoring, Test Addition, Regression, and Code Production, the conformance issue can be ambiguous. Zorro uses the categorization of the episode immediately preceding the current one to determine its conformance. This approach appears to lead more frequently to false negatives than to false positives. If, upon entering the IDE, the developer fixes a problem in production code that was left over from the previous development session and then runs the test code, this will be evaluated as Test Last. If he then proceeds to perform several TDD compliant yet conformance-ambiguous activities, then the entire session may end up being classified as non-TDD compliant. As of their 2014 article, the creators of Besouro were investigating

programmatic methods to evaluate Refactoring, Test Addition, and Regression independently of the neighboring episodes.

**Experiments using Besouro**

Beginning in the Fall of 2012, Davide Fucci and other co-researchers began a series of replicated experiments reviewing various aspects of TDD, and after the initial experiment, added the use of the Besouro tool to measure TDD process conformance. In early iterations, the experiment typically used upper division undergraduate students and junior graduate level students as the test subjects but since 2016 has been performed in industrial settings with professional developers. There were six three-hour sessions, during which the students were taught unit testing concepts, Eclipse, jUnit, and TDD. Analysis was performed on the results from the last of the three-hour sessions. Subjects were provided with a template program containing 30 lines of code that included an API for acceptance tests to run successfully [Fucci May 2014].

Initially, Fucci and Turhan performed a differentiated and partial replication of an experiment first performed by [Erdogmus 2005], namely a modified version of the Bowling Scorekeeper coding kata originally set forward by Robert Martin. Their goal was to evaluate external quality, "defined as the percentage of acceptance tests passed for the implemented stories" and productivity, "measured as the percentage of implemented stories." [Fucci & Turhan, 2014]. They found a correlation between TDD (when it delivered a higher number of tests) and higher productivity, but they found no significant correlation between number of tests and external code quality.

In the next iteration of experiments, Fucci, et al. added the use of the Besouro tool to measure process conformance while performing the experiment. [Fucci, Sept 2014]. They

evaluated quality, productivity, number of tests, and test coverage, and found no correlation between the measured variables and TDD process conformance. In [Fucci, May 2014], the experiment subjects were divided into TDD or TLD (test-last development) groups. In this round, TLD had better results in quality and productivity, whereas TDD had a slightly higher number of overall tests.

In [Fucci et al., April 2014], the researchers investigated the effect that developers' skills had on quality and productivity in a TDD context. They found that developers' skills had a significant impact on productivity, but not on external quality while performing TDD to accomplish the task.

A multi-site blind analysis was then performed by replicating the same experiment at two other universities [Fucci, Sep 2016]. The conclusion drawn was similar to previous results: TDD did not affect testing effort, software external quality, or developers' productivity.

The most recent experiment was conducted with industrial partners using a week-long workshop about unit testing and TDD. It also divided the participants into two groups: one using TDD and the other using incremental test-last (ITL). The focus of this study was to look at three different dimensions of an iterative development process (granularity, uniformity, and sequencing), while still evaluating quality and productivity. The results of their experiment imply that granularity and uniformity are more important than sequencing, and the authors assert that it is not the test-first aspect of TDD that is most important, but rather its emphasis on taking very small steps in the development process and covering those small steps with tests, whether before or after writing the code [Fucci, Nov 2016].

In summary, the trend of the findings from the Fucci-related experiments was that there is little support for the proposition that the use of TDD improves quality or productivity.

While Fucci and his colleagues are to be applauded for their re-use of an experiment in order to compare results over a number of different implementations, there are some inherent flaws and assumptions with the original experiment set-up. Colyer points out several issues with the design in this series of experiments. According to Colyer, "one of the expected benefits of the TDD process (is) helping you to shape and refine the design/interface." [Colyer 2017]. By providing the API for the precise method signatures to the test subjects, they are nullifying one of the benefits of TDD.

Another issue Colyer points out is the relationship between refactoring and the chosen definition of external quality and productivity.

> Refactoring also shows up in both models as a negative factor – i.e., the more refactoring you do, the worse the productivity and quality. For productivity at least, this seems to me to be an artefact [sic] of the chosen measurements. Since productivity is simply a measure of passing assert statements over time, and refactoring doesn't change the measure, time spent on refactoring must by definition lower productivity. Not accounted for at all are the longer term impacts on code quality (and hence future productivity) that we hope to attribute to refactoring. The model is set up to exclusively favour short-term rewards. [Colyer 2017]

Separate from Colyer's concerns is an issue related to novices in the development environment and the way granularity and uniformity are assessed. Granularity is defined as a "cycle duration typically between 5 and 10 minutes." For a novice who is just learning a toolset and a new development process, this appears to be a very small timeframe. Frequently a novice must stop and consult documentation or notes regarding the assignment at hand. This could easily exceed a five to ten-minute time window. Beck, in the "Is TDD Dead?" discussion [ThoughtWorks 2014 20 May], described getting up and going outside to think about his next steps. The concept of granularity being measured in time allows no time for thought about design of the code. And while uniformity is to be desired from an experienced TDD practitioner,

someone less experienced cannot be expected to maintain a relatively constant development time to each iteration. As with granularity, a developer is effectively penalized in their performance assessment for stopping to consider design implications or look at documentation or requirements. As Frederick Brooks said, "think-time [is] the dominant activity in the programmer's day." [Brooks, 1987] Penalizing a programmer for stopping to think is surely not the best measure of their effectiveness.

Another concern with the experiment is with its definition of quality. It only measures quality in the context of external quality, using a set of pre-defined acceptance tests as the measurement. Internal quality is never considered or assessed. One of TDD's supposed benefits is that it produces smaller, more focused classes with looser coupling. One can certainly imagine having poorly designed code that can pass acceptance tests. Indeed, if refactoring is seen as reducing a developer's productivity measurement, the implication is that refactoring provides no value. If there is no measurement of internal code quality, and thus no incentive to refactor rapidly-written and possibly poorly designed code, the resulting code could be a maintenance nightmare.

# 3. RESEARCH DESCRIPTION

Previous TDD studies have focused on external quality and productivity, with the results being inconsistent and mixed. TDD appears to improve quality when compared to a waterfall-style test-last mode of development, but shows no distinct advantage when matched against an iterative approach in which tests are written at the end of each iteration. Quality, then, appears to derive more from development cycles that are short, iterative and employ automated tests than from testing philosophies.

What is missing is the question of how well TDD was used. Previous studies examined TDD under the assumption that developers competently and consistently selected failing test cases (i.e., performed the red light portion of the TDD cycle) so as to inform the quality of their production code (i.e., the green-light element of the TDD cycle). Unless red light tests arise out of a conscious decision to identify missing functionality that will, at the end of the green-light phase, result in software that reduces the developer's uncertainty of a defect, they come about in an ad hoc fashion that fails to take advantage of TDD's engineering discipline. Put simply, TDD, as observed in studies to date and as currently practiced, amounts to short cycles in which quality might or might not be driven by tests. Tests serve to push production code to completion, rather than guide its construction by addressing areas of uncertainty as soon as possible.

This research sought to examine how quality is affected by *how* TDD is applied, not *whether* it is applied. We took the position that previous researchers have not shown a correlation between use of TDD and improved quality because failing test cases were not

identified in such a way as to use TDD for what it was intended: to address functionality that has the greatest likelihood of failure. We surmised that TDD would lead to better quality if developers were to construct a pre-defined amount of functionality (meaning, a module, component, or other design unit) through a series of red light/green light cycles that starts with the most fundamental statement possible and adds detail in minimal increments. The best-defined approach which supports this concept is the Transformation Priority Premise (TPP). It postulates that desired functionality can be achieved by successively applying a set of generalizing transformations to a simple base case. In general, the TPP operationalizes the canonical control structures expressed by [Böhm & Jacopini, 1966] by suggesting that *sequence* addresses functionality of a single data item, *alternation* generalizes functionality to take into consideration computation on a single data item that differs based on its value, and *iteration* generalizes functionality further by addressing a stream of multiple data items. The ultimate goal of TPP is to produce a "better" algorithm with simpler code [Martin, R. 2013]. In theory, this would lead to less complex code, which would be more maintainable and reduce technical debt.

## 3.1 Hypotheses

We hypothesized that following the steps of TPP would reduce risk in software development. We defined *risk* as the extent to which desired functionality fails to meet specifications. Because risk cannot be measured directly, we chose two proxies to provide us an approximation: code coverage and code increment size. Our rationale was that test cases that drive successive TPP transformations would yield better coverage and that the amount of production code affected by successive applications of transformations would be minimal. Our research hypotheses were as follows:

1. For code coverage:

   H0-1: Students conforming to the TPP process will have no better code coverage than students who do not conform to the TPP process.

   H1-1: Students conforming to the TPP process will have better code coverage than students who do not conform to the TPP process.

2. For TPP compliance:

   H0-2: Students conforming to the TPP process will have no difference in the size of their code changes from one commit to the next than students who do not conform to the TPP process.
   H1-2: Students conforming to the TPP process will have more-consistently sized and smaller sized code changes from one commit to the next than students who do not conform to the TPP process.

## 3.2 Case Study environment/history

This research was carried out as a case study on assignments submitted over 2 semesters by students at Auburn University enrolled in COMP5700/6700/6706, *Software Process*. *Software Process* is an upper-division course that is taken by Computer Science majors who are seniors in the undergraduate program and graduate students at all levels. All levels of students complete the same homework assignments, with graduate students having additional work beyond those assignments. The primary objective of the course is to expose students to best practices in software engineering, including how to perform Test Driven Development, with at least one lecture devoted to the Transformation Priority Premise. At least three homework assignments required the use of the TDD process, and the final two homework assignments expected the use of TPP.

Homework submissions from COMP 5700/6700/6706 were analyzed for TDD compliance and students received feedback informing them of how well they employed the TDD process. In contrast to previous research using Besouro in which elapsed time was used as a indicator of the amount of functionality built over a TDD cycle, we used net lines of code added

per cycle as the measure of the output of each TDD increment. TPP emphasizes simpler code, so one would anticipate that fewer lines of code would be required to produce the desired result.

Assumptions for the research approach were as follows:

1. Test Driven Development (TDD) is a desirable software development practice because it encourages building code in testable increments.
2. The Transformation Priority Premise (TPP) shapes the effectiveness of TDD by suggesting the order in which to implement functionality.
3. Code coverage can be used as a proxy measurement for risk in software development. For example, high code coverage reduces risk of defects in software, while low code coverage increases risk of defects.

The case study environment included Python [Python 2.7. https://www.python.org], the Eclipse IDE [Eclipse Foundation. https://www.eclipse.org], the PyDev [PyDev. http://www.pydev.org] plugin for tailoring Eclipse for Python use, and a local git repository linked to a GitHub [GitHub, inc. https://github.com] account.

The environment also included an Eclipse plug-in written specifically for this research to track code built in each TDD cycle. The plug-in added buttons to the Eclipse menu bar that allowed students to indicate when they were running red light and green light tests. The buttons automatically committed changes to the local git repository, flagging each commit with a message that stated whether the commit was intended as a red light test or a green light test.

Post-processing software analyzed each git commit to determine the degree to which the student complied with TDD and TPP. Specifically, the analysis application examined the order and number of the transformations performed per commit; the size of the commit in lines of code; and what files were changed. The transformations per commit pointed to the amount of functionality added per TDD cycle and the commit size indicated the complexity of the functionality. The file changes gave an indication of the integrity of the TDD method based on the concept that the red light phase should limit changes to test code and the green-light phase

should limit changes to production code. The analysis software also examined the order in which red light and green light phases occurred as a way of assessing TDD skill acquisition. Red light indicators should alternate with green light indicators in the ideal situation, indicating the creation of a failing test case and the subsequent creation of sufficient production code to make the test case pass. Multiple, sequential red light or green light indicators signaled possible confusion about TDD, disregard for employing TDD, or a defect in either the test code or the production code.

The analysis application generated a report for each student showing TDD-related statistics, including a TDD Score. A separate code coverage report provided an individual report for each student with code coverage percentages per submitted file, while a composite report lists all the students and their overall code coverage scores.

Information from the git logs provided snapshots of development activity at the end of each TDD cycle. The git commit message named the intent of the TDD cycle, git log entries with a leading minus sign indicated lines that were deleted from the previous commit, and entries with a leading plus sign indicated lines added to the previous commit. Individual lines were examined further to determine the nature of the code change in an attempt to map the change to a TPP transformation. The table below shows how the various keywords and symbols were interpreted to determine the specific transformation:

Table 1: TPP Transformations and the Python conditions that define them

| **Transformation** | **Statement patterns that suggest a transformation** |
|---|---|
| Null | + pass<br>+ return None<br>+ return |
| Null to Constant | - pass<br>- return None<br>- return<br>+ return with a number, or a string literal, or empty list |
| Constant to Variable | - return with a number, or string literal, or empty list<br>+ return with variable name |
| Add Computation | + string containing either +, -, *, /, %, or math. |
| Split Flow | + if |
| Variable to Array | |
| Array to Container | |
| If to While | - if (record conditional values)<br>+ while (if it contains the same conditional values as if) |
| Recurse | + Method name is called within the method of that name |
| Iterate | + for |
| Assign | + Parameter is assigned a new value inside the method's code. |
| Add Case | + else or elif |

In the process of evaluating the transformations, any patterns opposed to the TPP were also noted. Anti-Transformations imply that the developer jumped past a lower-order transformation to a higher-order transformation. Skipping steps adds to the riskiness of the code development. The following Anti-Transformations were recorded:

Table 2: Anti-Transformations and the Python conditions that define them

| **Anti-Transformation** | **Statement patterns that indicate an anti-transformation** |
|---|---|
| Constant Only | + return with a number, or a string literal, or empty list with no corresponding deleted return or return None |
| Straight to Variable | + return with variable name with no corresponding deletes consistent with Constant or Null |
| While with no If | + while (with no corresponding deleted if statement containing the same conditional values) |

The analysis software generated a text-based report for each student. Appendix A contains a brief excerpt of the student report, as well as a UML diagram describing the components of the system.

The summary report generated for each student listed the transformations that were detected in each of his or her commits and indicated compliance with general TDD recommendations. In the first iteration, the report showed number of commits; red light commits; green light commits; average lines of code and transformations per commit; added and deleted/modified lines of both production and test code; and the ratio of production to test code.

The summary report led to additional questions about the students' TDD performance. Did the students alternate between red and green light commits? If they were not alternating, why not? Were their tests failing in unexpected ways on the red light commit, resulting in consecutive red lights? Was their production code not passing on their green light commit, resulting in a number of consecutive green lights? Were they adding too much code at one time, violating the TDD principle that one should only write enough code to make a single test pass? Or in TPP terms, were they making more than one Transformation per red light/green light cycle?

A TDD scoring criteria was developed to assist in examining the questions raised in the prior paragraph. A score was assigned per commit by examining each red and green light commit and whether they appropriately created production or test code in it, by looking at how many transformations they performed, and by how large the commit was. The scoring range for each criteria was 0 to 100. The following chart shows the breakdown:

Table 3: TDD Scoring Criteria for a Commit

| Criteria (per commit) | Condition | Score |
|---|---|---|
| 1a. Red Light - does it contain Production Code? | 0 | 100 |
| | 1 and above | Deduct 25 points for every prod file in RL |
| 1b. Green Light - does it contain Test Code? | 0 | 100 |
| | 1 and above | Deduct 10 points for every test file in GL |
| 1c. "Other" Commit w/large number of lines of code changed (student bypassed Red/Green/ Refactor button) | | Commit received the Large Commit score for this portion of the grading. Data observation indicated that many students would create a very large commit with many changes, but not classify it as Red or Green. |
| 1d. "Other" Commit w/multiple refactorings | 3 or below | 100 |
| | Above 3 | Commit received the Number of Transformations score for this portion of the grading. Data observation indicated that many students would have net lines of code change below the large commit threshold, but would have made numerous transformations, but not classify the commit as either Red or Green. |
| 2. Number of Transformations (calculated for production code only) | 1 | 100 |
| | 2 and above | Deduct 10 points for every additional transformation<br>Anti-transformations – deduct 20 points. With an anti-transformation, the subject skipped a transformation according to the given order and is subsequently penalized. |
| 3. Large Commits | Less than 10 net LOC added | 100 |
| | Increments of 10 | Deduct 5 points for each increment of 10 LOC above 10 |

The appropriate grade from Criteria 1 was averaged along with the grades from Criterias 2 and 3 to produce a TDD score for each commit.

If a student performed TDD per the recommendations, there should have been many commits. In an ideal TDD cycle, TDD would be manifested by alternating red light and green light commits. Consecutive red and green light commits indicated either a difficulty with test or production code or a misunderstanding of the TDD process. We calculated an average length of consecutive red and green lights documented throughout the entire assignment, then incorporated that score into an overall assignment score that included an overall average from all the individual TDD Commit scores. The following table indicates the scoring criteria for an overall assignment:

Table 4: TDD Grading Criteria for an Assignment

| **Criteria (per assignment)** | **Condition** | **Score** |
|---|---|---|
| Average of all commit grades in Assignment | | Average of Commit Scores |
| Average length of Consecutive Red Lights | 1 | 100 |
| | 2 and above | Deduct 5 points for every number above average length of 1 |
| Average length of Consecutive Green Lights | 1 | 100 |
| (The criterion for Green Lights was more lenient because students were given instructions to hit the Green Light button until the test passed) | 5 and above | Deduct 5 points for every number above average length of 5 |

The three elements were averaged together to produce a TDD score for each assignment.

A phenomenon that emerged as the students submitted their homework involved the occurrence of "Other" commits. The students would bypass using the Red/Green/Refactor buttons provided and run and/or commit using the Eclipse facilities available. This resulted in a large number of commits that did not fall into the scoring criteria, thus nullifying the scoring for the average length of consecutive red/green lights. After analyzing the individual result reports,

the scoring criteria was adjusted to address this behavior. If a student's average number of red lights or green lights fell below one standard deviation from the class average, their red or green light score was reduced to 70. If their average fell more than 2 standard deviations below the mean, their corresponding score was reduced to a 50.

As a side note: TDD literature advocates following each red light/green light cycle with a refactoring activity. Refactoring is a key component of the TDD Cycle, but as noted above, it is used to clean up the code without changing its behavior. In this analysis, the focus was placed on the transformations that actually implement the behavior between the red light and green light step. Students also had the opportunity to press a Refactor button. The data for the Refactor commits do not factor into the overall TDD Score.

## 3.3 Spring 2018 Case Study Results

The first set of data used in this analysis was from the Spring 2018 semester. We examined a total of two hundred fifteen assignments across sixty students. Students were given three TDD-related assignments, and the final semester assignment was used for the purposes of analyzing the hypotheses, as it represented the culmination of TDD practice throughout the semester. Results of the analysis were only available for students who were able to successfully install and use the Eclipse plug-in described earlier. It was not possible to gather the correct data from students for whom the plug-in did not work. During the Spring 2018 case study, the plug-in only worked for students who were using a Mac system. This reduced the available sample size to twenty subjects. As seen in the descriptive statistics given in Table 5, the mean TDD score was 83.35, with a median of 87, so scores trended high.

Table 5: Descriptive Statistics for the variables under study (Spring 2018)

| | *TDD Score* | *Code Coverage* | *Avg LOC / Trans / Commit* |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Mean | 83.35 | 53.7 | 8.67 |
| Standard Error | 3.343 | 8.7 | 1.84 |
| Count | 20 | 19 | 19 |
| Median | 87 | 67 | 8.53 |
| Mode | 96 | 89 | 0 |
| Standard Deviation | 14.95 | 37.94 | 8.04 |
| Sample Variance | 223.607 | 1439.76 | 64.7 |
| Skewness | -1.586 | 0.368 | 1.06 |
| Range | 52 | 94 | 28 |
| Minimum | 44 | 0 | 0 |
| Maximum | 96 | 94 | 28 |
| 1st Quartile | 77 | 6.5 | 2.5 |
| 3rd Quartile | 94.5 | 89 | 11 |

The statistics indicate that the TDD scores are not normally distributed, as graphically illustrated in Figure 3.1 and 3.2. TDD conformance is right-skewed, which is influenced by the scoring criteria for "Other" commits. The lower bounds of "Other" commits was set at 50 based on lower limits placed in the scoring rubric for the course. Another observation related to TDD score is that students could make dozens of commits that conformed to TDD and then have one very large commit where they broke from the pattern and wrote many lines of code. Because the TDD average was divided by the number of commits, a student could still receive a relatively high average even when a small percentage of commits had a very low score, sometimes even zero. These two factors contributed to relatively high TDD scores.

Figure 3.1

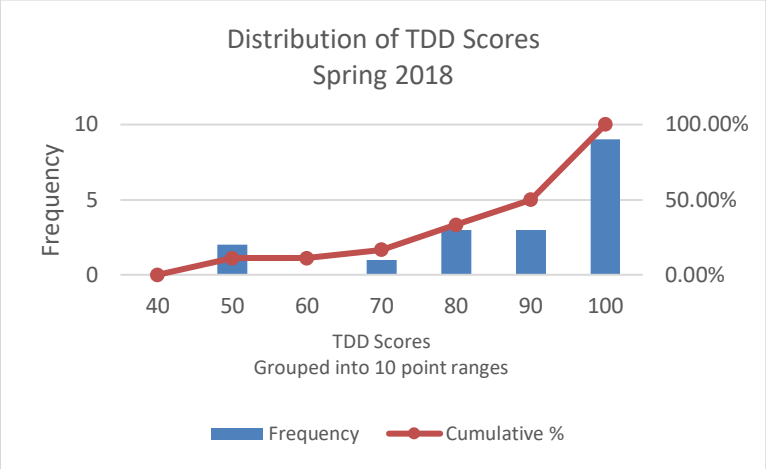
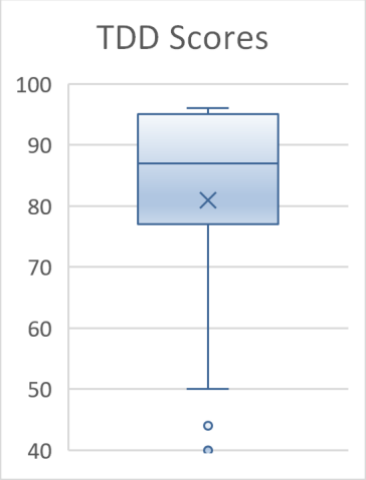



Figure 3.2




From the skewness number in the descriptive statistics, we know that Code Coverage is skewed slightly to the left, however, the graph of the percentages indicates a multimodal distribution, illustrated by Figure 3.3. The range is very wide at 94. The lower scores are attributed partially to failing test cases, which interrupted the code coverage analysis tool. Thus, there appears not one, but three groups represented by the data. On the lower end are students whose unit tests would not pass or who truly had poor code coverage, and on the upper end are students who were successful at writing and passing their own unit tests.

Figure 3.3

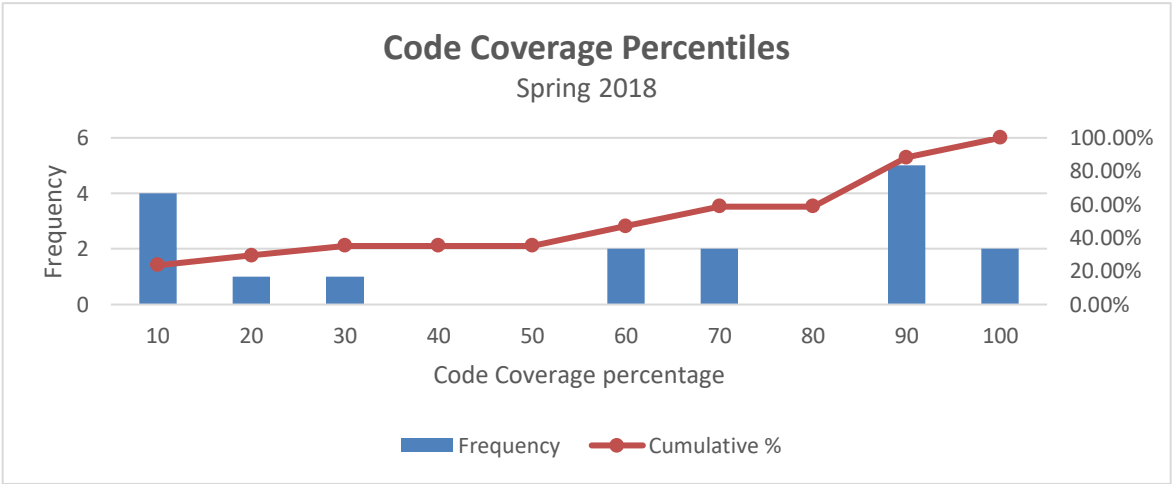
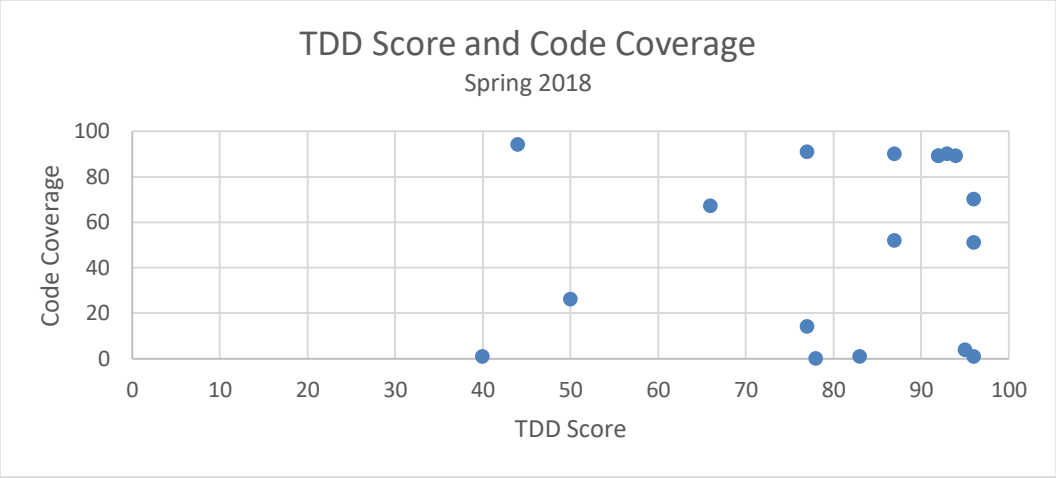


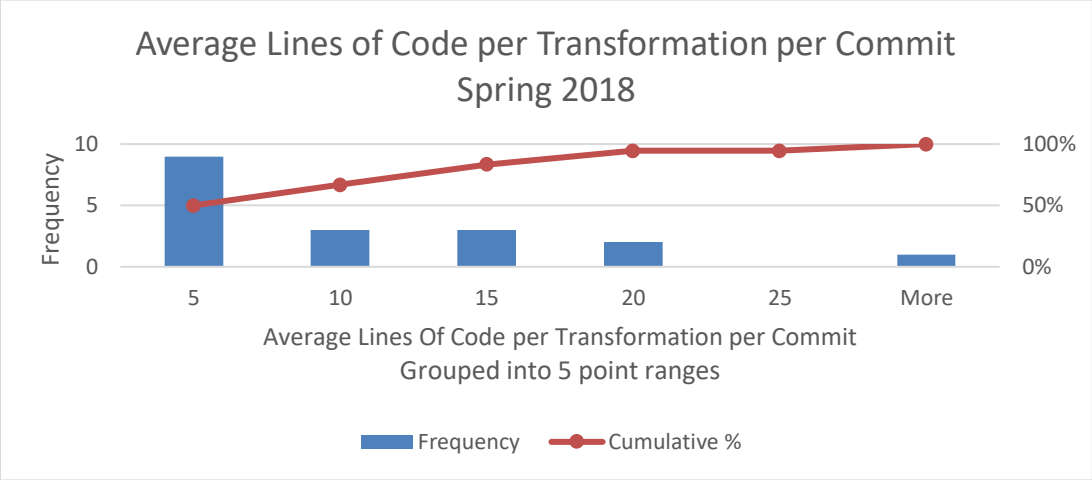
Figure 3.4 provides a scatter plot illustrating the intersection of the students' TDD Scores and Code Coverage percentages. While there is a cluster of students who demonstrate both high TDD Scores and high Code Coverage scores, no discernable trend is present. Because the code coverage percentages were not normally distributed and the TDD scores were skewed so significantly to the right, the correlation score associated with the TDD score and code coverage ($r = .16$) is basically meaningless. Therefore, with respect to H0-1, we must conclude that the null hypothesis holds.

Figure 3.4



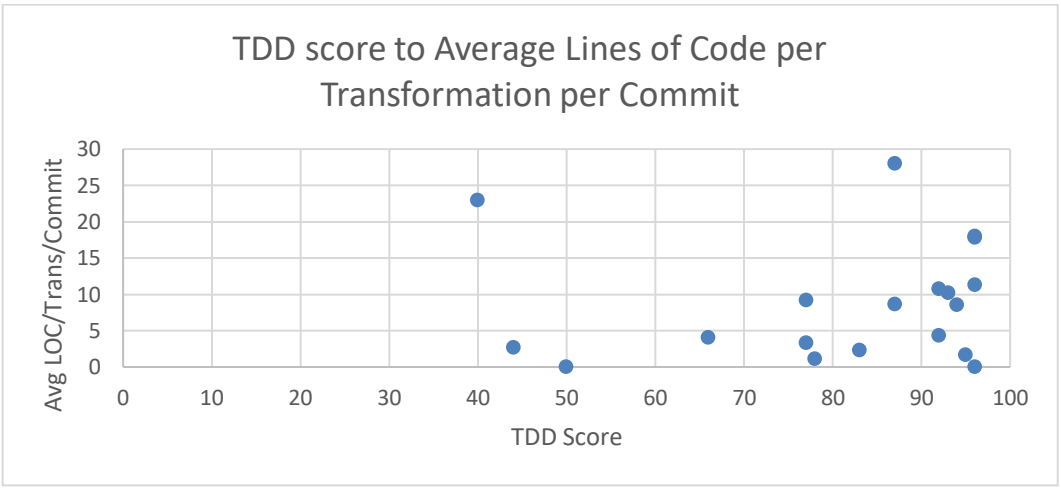

As can be seen in Figure 3.5, the average lines of code per transformation per commit are left-skewed, indicating the students tend toward smaller sizes of commits in the final assignment, which is the desired effect.

Figure 3.5




However, there is no correlation seen between TDD Score and Average Lines of Code per Transformation Per commit (r = .07). As was seen with TDD in relation to Code Coverage, there is a cluster of scores to the bottom right on the scatter plot seen in Figure 3.6, but it is not strong enough to suggest an influence. Thus, we also cannot reject the null hypothesis found in H0-2.

Figure 3.6



## 3.4 Fall 2018 Case Study Results

The second set of data used for analysis was from the Fall 2018 semester. We examined a total of 460 assignments across 102 students, which provided a much larger sample size than was available in Spring 2018. In the Fall 2018 case study, students were assigned a total of 4 TDD-related programming assignments. The assignments represented iterations that would eventually produce a software representation of a Rubik's cube. The first assignment contained starter code with the method signature that would be used for acceptance testing in the subsequent assignments. For this assignment, students were expected to set up their development environment according to the course requirements and to return the software representation of a solved Rubik's cube. Students had been instructed on TDD during class time, including an in-class demonstration of the TDD technique, and also received a series of 6 screen-cast examples on starting TDD. Each subsequent TDD assignment added additional functionality to the Rubik's cube. After each assignment, students received feedback related to their performance of TDD and were asked to incorporate that feedback into their TDD practice in the next assignment.

As with the Spring 2018 data, the final semester assignment was used for the purposes of analyzing the hypotheses. The Eclipse plug-in worked correctly on all Operating Systems used by the students, which provided a higher percentage of class participants in the case study. Fifteen students did not submit the final assignment; this reduced the available sample size to eight-seven subjects. The descriptive statistics are given in Table 6. As compared to Spring 2018, a larger percentage of students had failing test cases that resulted in a zero-score assignment from the code coverage tool. This significantly skewed the descriptive statistics of the code coverage results. Because of this, the descriptive statistics contains two columns for

Code Coverage: one with numbers reflective of the sample containing 0 scores and a smaller sample containing only non-0 scores.

Table 6: Descriptive Statistics for the variables under study (Fall 2018)

| | *TDD Scores* | *Code Coverage (including 0's)* | *Code Coverage* | *LOC / Trans / Commit* |
|---|---|---|---|---|
| Mean | 87.2 | 33.23 | 45.6 | 0.26 |
| Standard Error | 1.8 | 3.63 | 4.067 | 0.07 |
| Count | 87 | 87 | 61 | 87 |
| Median | 97 | 20 | 46 | 0.13 |
| Mode | 97 | 0 | 1 | 0.05 |
| Standard Deviation | 16.86 | 33.84 | 31.76 | 0.645 |
| Sample Variance | 284.4 | 1145 | 1008.9 | 0.42 |
| Skewness | -1.47 | 0.45 | -0.046 | -1.31 |
| Range | 67 | 99 | 98 | 6.69 |
| Minimum | 33 | 0 | 1 | 0.00 |
| Maximum | 100 | 99 | 99 | 3 |

Again, the statistics indicate that the TDD Score data are not normally distributed, as graphically illustrated in Figure 3.7 and 3.8. TDD scores for the Fall 2018 case study were even higher than the Spring cohort, with a mean of 87.2 and a median and mode of 97. TDD conformance is right-skewed, for the same reasons as discussed in the previous section.

Figure 3.7

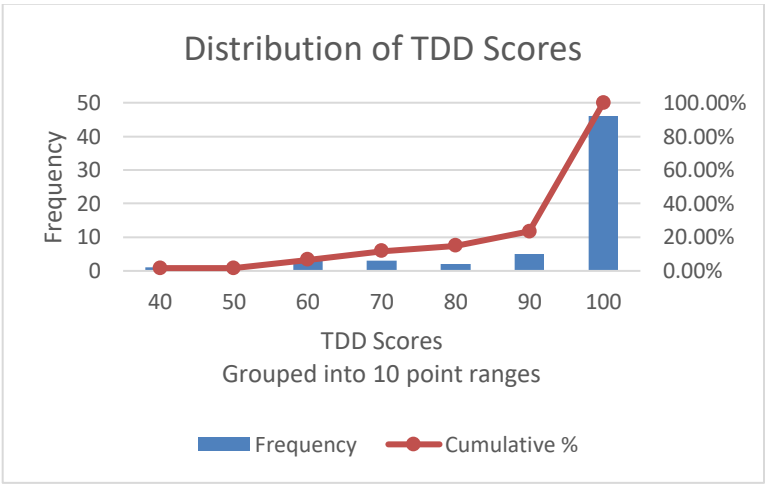
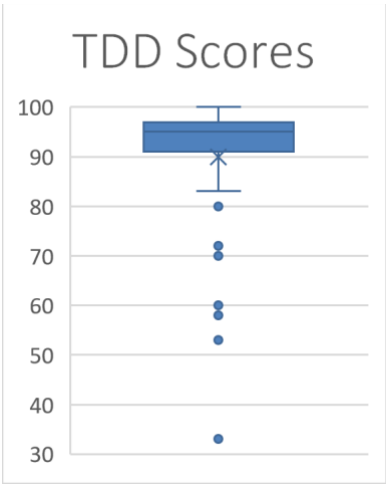



Figure 3.8



Having a larger sample size in the Fall 2018 class results in a Code Coverage percentages graph that indicates a bimodal distribution, illustrated by Figure 3.9. Even excluding the 0 values as shown in Figure 3.10, the range is even wider than in Spring at 98. Once again, there is a contingency of students on the lower end whose unit tests would not pass or who truly had poor code coverage, and on the upper end are students who were successful at writing and passing their own unit tests. Perhaps the scatterplot in Figure 3.11 best sums up the Code Coverage scores; they are truly scattered with no appearance of a normal distribution.

Figure 3.9

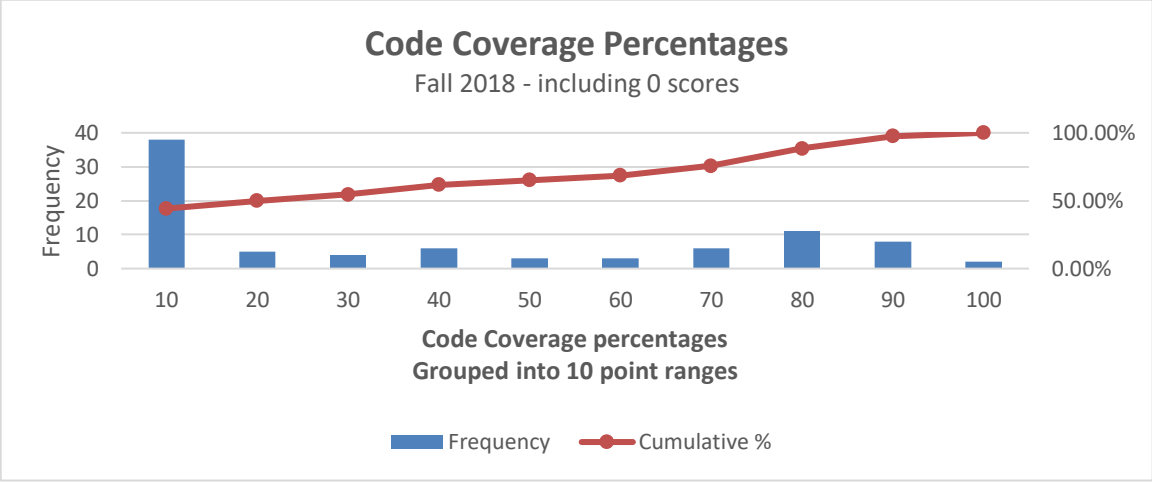


Figure 3.10

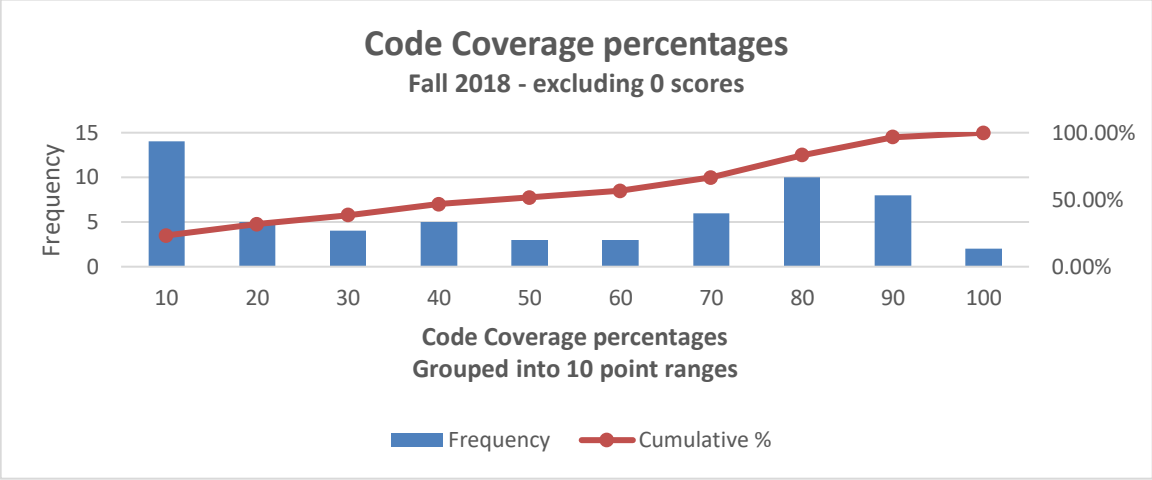
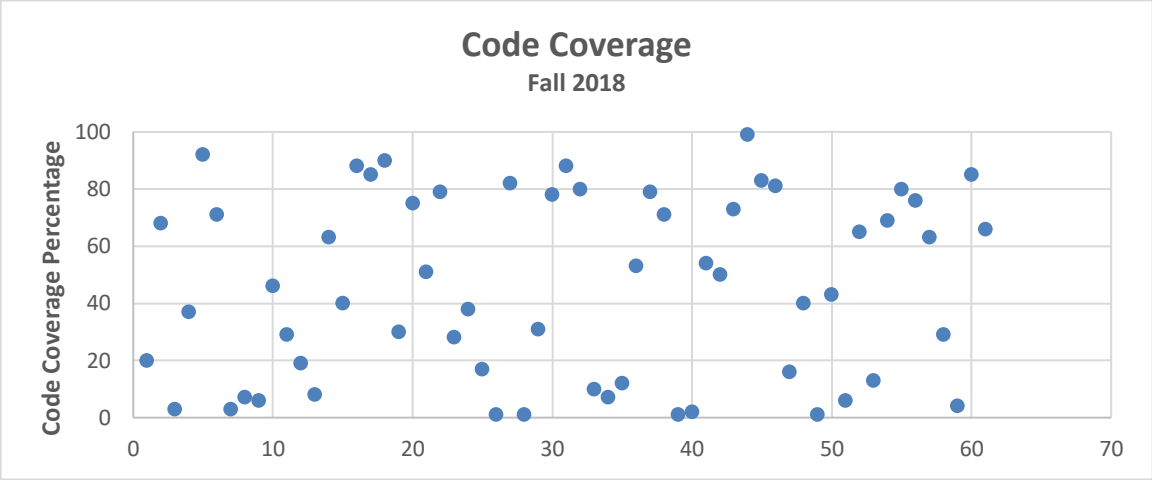


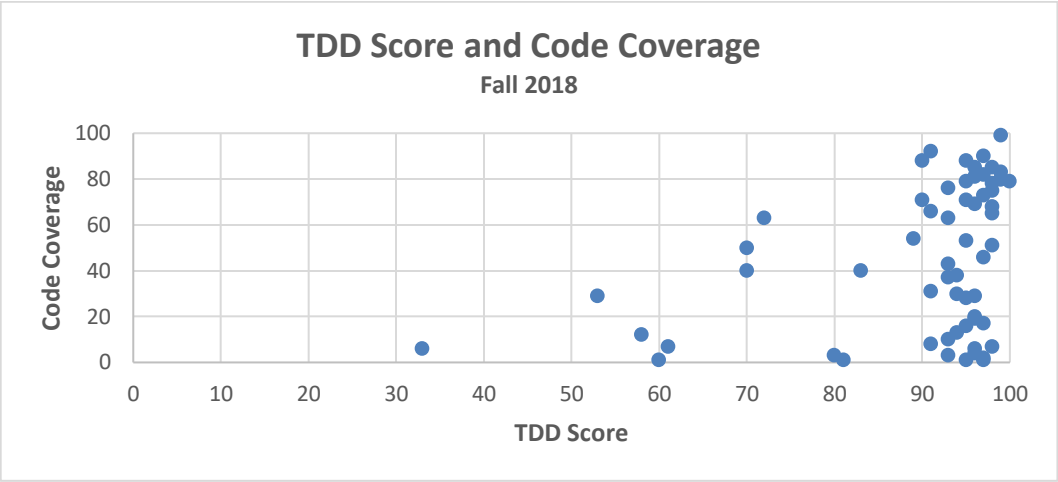

Figure 3.11




Taking the TDD score assigned by the analysis software and correlating it to the students' code coverage resulted in a .357 correlation coefficient. For the purposes of the correlation analysis, students with a 0 score were omitted as this skewed the calculation. As can be seen in Figure 3.12, many students had a high TDD score but still had a very low Code Coverage score. As before, the code coverage percentages were not normally distributed and the TDD scores were skewed significantly to the right, but perhaps because of the larger sample size, the correlation score between the TDD score and code coverage produces a moderate correlation

value. But because of the non-normal data, with respect to H0-1, we have insufficient evidence to disprove the null hypothesis.

Figure 3.12




After further investigation, the students with low code coverage scores despite a high TDD score appeared to have written tests that exercised the same code over and over. Their tests did not branch out to evaluate other parts of the code. So, while they appeared to have a high TDD score because they were writing code using the alternating red and green light approach, they were writing repetitive tests. When they did add production code, they would write it in larger chunks that were not exercised by the tests they had actually written.

Figure 3.13

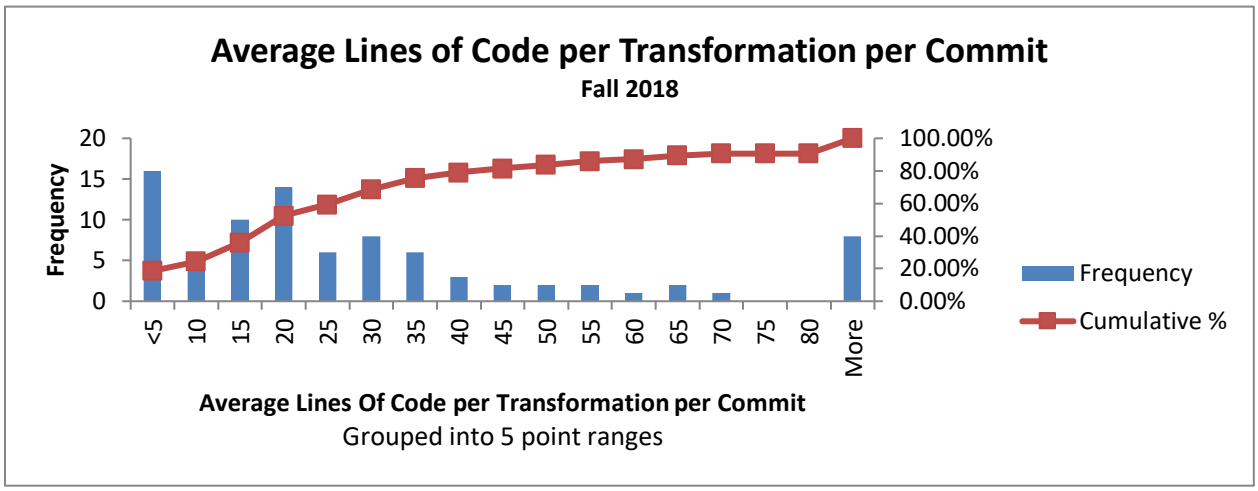
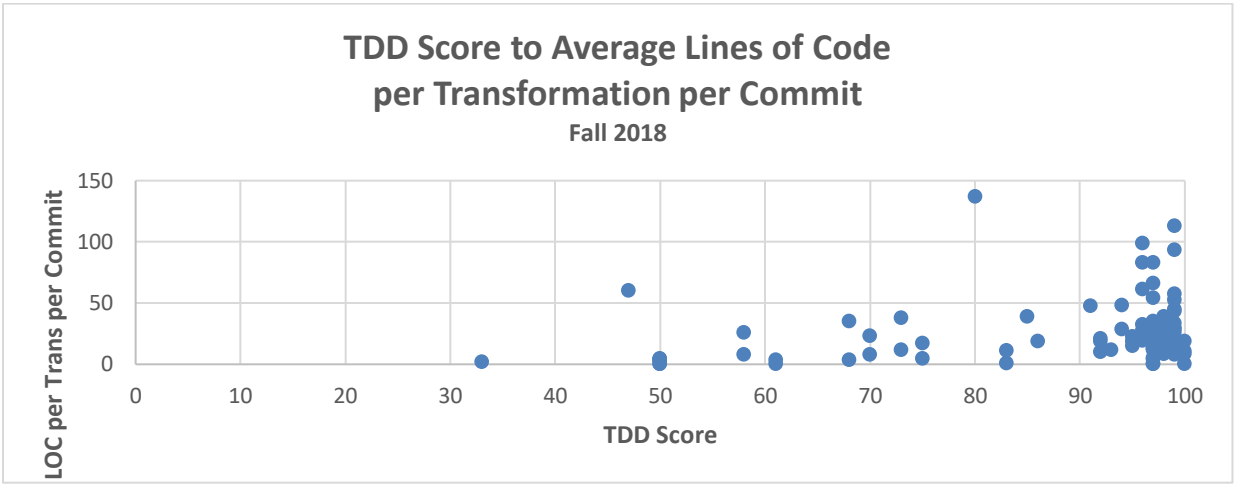



Figure 3.14




As in the spring, there is basically no correlation between the TDD Score and the Average lines of code per transformation per commit (r = 0.259). The "More" category in Figure 3.13 counts outliers who either wrote or copied in large chunks of code to complete the final assignment and had not been writing code incrementally throughout the course or had skipped an assignment or two and were trying to write the code for the entire cumulative assignment in one or two continuous sittings at the end of the semester. Even if the outliers were eliminated from

the correlation calculation, r is still small at 0.317. Thus, the null hypothesis holds for H0-2 in the Fall results as well.

Further analysis was performed using the available data. In the final assignment, the students averaged an acceptance test passage rate of 72%. Regression tests of the acceptance tests from Assignments Four through Six resulted in a class average of 78% passage. When the correlation analysis was applied between the students' TDD scores and their acceptance test passage rate, there was a correlation coefficient of .46 for Assignment Seven and a factor of .65 for the scores of the combined acceptance tests from the previous three assignments. This would appear to imply that following TDD helped students to pass the course's acceptance tests.

## 3.5 Fall 2018 Survey Results

Before the semester began, students filled out a pre-course survey. Full results of the pre- and post-course survey questions related to TDD can be found in Appendix C. The one question that related to TDD merely asked "On a scale of 0 to 5 (where 0 = no proficiency and 5 = expert), how would you rate your Test Driven Development (TDD) skills?" Over three-fourths of the class rated themselves a two or lower, meaning they had little to no TDD skills. This implies a relatively clean slate, so most class participants would have no prior misconceptions about what TDD is or how to perform it.

In the post-course survey, eleven questions were asked. Over eighty percent of students felt that TDD improved the quality of their code, so the majority of students appear to have a positive opinion of the practice. However, only sixteen percent said they would like to make TDD a part of their baseline skillset, and only thirty-five percent would want to use it in a less strict form. This implies that while they appreciate the value of the practice, the majority aren't convinced of its usefulness in software development.

Over ninety percent said they understood what they should do during the red light and green light phases of TDD. However, while they intellectually understood what should be done, their self-admitted adherence was far less than that. More than fifty percent of students admitted to writing more green light production code than was called for by the red light test. In their description of their adherence to TDD in the final programming assignment, only sixty percent indicated either a strict adherence or that they slipped a few times in performing TDD. Forty-five percent found TDD to be very easy or easy to use by the final assignment.

To test the students' self-awareness related to their TDD compliance, a comparison was drawn between the students self-reported adherence and the TDD score. Roughly seventy-eight percent of students self-reported a level of adherence that corresponded to their TDD score. For instance, if a student self-reported very strict adherence or slipping only occasionally with their TDD, and their TDD score was greater than 80, this was counted as a corresponding result. This seems to imply that most of the students had a good appreciation for whether or not they were complying with the TDD guidelines that were given to them.

Three questions related to the TPP aspect of the case study. While over three-fourths of the class understood the purpose of TPP, only sixty-four percent thought they understood how to apply it. The number who believed they used it appropriately dropped to fifty-two percent.

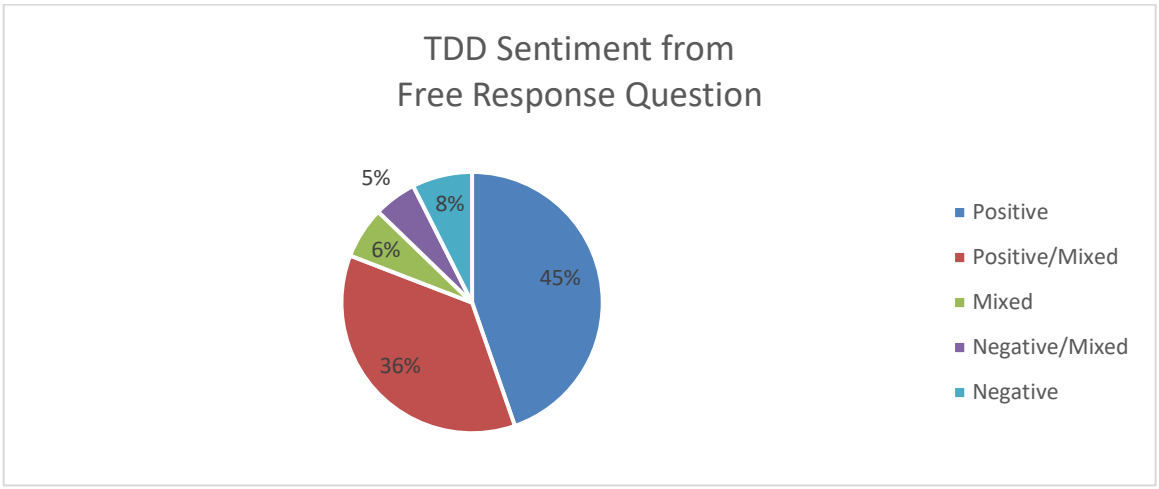
The survey data provided the opportunity to answer other questions. For instance, would prior TDD experience have any bearing on students' anticipated use in the future? There was basically no correlation between prior experience and anticipated future use ($r = -.04$). Another question: would students who found TDD to be easy to use by the last assignment be more

likely to say that they would use TDD in the future? There was a low correlation between ease of use and anticipated use of TDD in the future, with a correlation coefficient of .358.

The survey provided a free response question related to TDD: "What is your overall opinion of TDD? Please provide an insight into how useful you found TDD and TPP." The most frequently used word/words in the responses were helped (25) and helpful (13). When looking at the words in context, helped was normally used with phrases like "write better code" or "understand requirements" or "catch mistakes sooner." More than sixty percent of the time it was used in a positive context. About half of the time, the word helpful seemed to come with a caveat. For instance, it was helpful, but "time consuming" or "more tedious than helpful" or a less-positive comment about the strictness of TDD. The word good occurred twenty-two times, and over two-thirds of the time it was used in a positive way. The other third of the time contained caveats like "difficult," "time-consuming," "too hindering," or additional comments related to strict adherence. The most frequently used but less-positive words occurring in the responses were the words "tedious" and "difficult" which were used nine times each, and the word "hard" (eight times).

To summarize, the overall tenor of each comment was assigned a Likert-style scale ranging from Positive, Positive/Mixed, Mixed, Negative/Mixed, and Negative. When assigning Positive/Mixed or Negative/Mixed, the comments fell predominantly into the first category, but contained some kind of caveat. Figure 3.13 illustrates that the overwhelming majority of the class (81%) had a Positive sentiment about the practice as expressed through the free-form question.

Figure 3.15


TDD Sentiment from
Free Response Question
5%
8%
6%
45%
36%
Positive
Positive/Mixed
Mixed
Negative/Mixed
Negative

# 4. THREATS TO VALIDITY

We use the categorization recommended by Wohlin et al [Wohlin 2012] to discuss the threats to validity.

## 4.1 Threats to Internal Validity.

This threat has to do with the internal design of the study and impacts that would have had on the outcome. In this series of case studies, subjects self-reported red/green/refactor events. While the analysis tool was in place to measure process conformance, investigation of individual students' submissions unearthed many reasons that they frequently strayed from a strict adherence to TDD. As noted in Section 3.5, while ninety percent of students said they understood what they should do in the red and green light phases, they frequently combined changes to production and test code into one event. Could this imply that there are situations where it makes more sense to combine changes to both in one "step"? Or is this a manifestation of the *maturation threat* where subjects react differently over time and become tired or bored with the process under study? Another possibility is related to homework deadlines; as students got closer to the homework deadline, they may have decided to ignore the process in favor of completing the project. While this phenomenon is a part of the academic landscape, it is applicable to an industrial setting. As deadlines approach for completing a software development cycle, there are certainly occasions where professional developers bypass certain process expectations to make the deadline.

In addition to modifying both test and production code within a single commit, there were numerous instances where subjects had consecutive red or green light instances between commits. It is easy to see how someone who is learning a new practice and/or language might make mistakes in writing the test cases or even in writing the code intended to make that test case pass. In the pre-course survey, thirty percent of the students indicated they had little to no proficiency in Python, the language of choice for this course, so in addition to other course content, these students were learning Python as well. If the test case or production code did something unexpected for a novice, the student has to choose whether to press the red or green light to self-report the phase. In our study, students were instructed to continue to press the green light button until they were able to get the individual test to pass. The TDD Scoring criteria took this into account when evaluating consecutive green light episodes. How would a tool like Besouro evaluate these types of situations? Should these concerns impact the definition of TDD conformance in future experiments?

### 4.2 Threats to Construct Validity.

This threat evaluates the relationship between theory and observation. Students may have had an *evaluation apprehension threat* with regards to their TDD performance. While students were evaluated by Teaching Assistants for their TDD conformance, the results from the analysis tool did not have any direct bearing on their grades. But the fact that they were being evaluated on how well they attempted to perform TDD did cause some to resent the practice, as evidenced by some of the comments in the post-course survey.

### 4.3 Threats to Conclusion Validity

This is used to evaluate the relationship between the dependent and independent variables. The study is subject to a *threat of random heterogeneity of participants* because most

of the graduate students in the class came from different universities with drastically different backgrounds from the perspectives of cultural, language, and education. Some students may have failed to fully comprehend the instructions due to language barriers or a deficiency in their prior education related to the software matters covered in the course.

Another aspect of Conclusion Validity is *reliability of measures*. The assessment tool for this experiment was written from scratch in Python. The underlying measurements are objective in nature (lines of code, numbers of commits, specific types of code constructs, etc.) and have been verified through manual comparison of the assessment tool output and the code under evaluation. The number of points deducted in the scoring process was somewhat subjective in nature. In order to validate the approach of the assessment tool TDD score, comparisons were drawn with the Product and Process grades given by the Teaching Assistants. TDD scores that were widely inconsistent with the TA grades were examined individually, and the scoring algorithm was adjusted as described in Section 3.2 based on observations drawn from those comparisons.

## 4.4 Threats to External Validity

These threats are directly related to the ability to generalize the results. The subjects for the experiment were upper-division undergraduates and graduate students. Trying to generalize the results to professional developers could represent a *threat of interaction of setting and treatment.* One attempt to mitigate this threat was to use an on-going assignment over the course of the semester rather than one "toy" assignment over a short period of time (a time frame of weeks as opposed to hours). By the end of the semester, the mean size of the students' code base for those who completed all four assignments was 996 lines of production code. Students submitted their assignments, received the results of the acceptance tests, and were expected to

make corrections to their code before proceeding to the next assignment. This correlates better to the expectations of professional developers whose code is subjected to automated testing and fixes must be applied before the code can proceed into production. However, at the end of the semester, the code is indeed thrown away with no expectation of being placed in production.

The threats described above are believed to have minimal impact on the overall results of the study. Thus, we are comfortable with the outcomes as presented in this paper.

# 5. CONCLUSIONS AND FUTURE WORK

To state that TDD results in a better product is a naïve claim. One of the reasons that studies continue to have inconclusive results is that TDD is just not that simple to precisely define and measure. Even the Besouro tool being used for measurement in the series of experiments carried out by Fucci, et al. must infer whether a developer is performing TDD based on a series of events, and the tool still contains some ambiguity in its operational definition of TDD. There is no hard and fast measure that says, "This is TDD."

Some of the earliest studies conducted about TDD compared it to a more traditional waterfall approach where the majority of code was written and then tests were applied against the code later. In this context, TDD produced far superior results. When comparisons are drawn against the current mode of iterative development, Fucci et al. submit that TDD does not appear to be superior for development and testing than an Iterative Test-Last Approach.

The results of our study concur with the results of the studies conducted by Fucci and his associates in that the practice of TDD does not appear to be the contributing factor for a "better" product in the short term. In our studies, conformance to TDD did not have a measurable influence on code coverage, which we used as a proxy for risk, or on the size of the commits, our measure for TDD granularity.

Beyond the definition of TDD, there are no agreements on the definition of how to best evaluate quality in a software product. Previous TDD experiments have focused on external quality as measured by automated test passage rates with a total disregard for the internal quality

of the code product itself. But a number of the purported benefits of TDD speak to cleaner, more maintainable code, which currently has no universally agreed upon measurement standards.

Based on numerous studies, Test Driven Development does not appear to be superior to Iterative Test-Last approaches with respect to empirically-measurable values of quality or productivity, at least in the short term. Some arguments in favor of TDD claim that it provides more maintainable code, or another way to describe it is to reduce technical debt. A future source of investigation is to compare open source projects that advertise the use of TDD and open source projects that have unit tests but do not use the TDD process. Several tools are currently available to provide analysis of technical debt in a code repository. This type of analysis would provide one way to measure internal quality of a code base over a longer timeframe, and perhaps provide a measurable answer to the "better quality" aspect of the value of Test Driven Development.

And what of the Transformation Priority Premise? Compared to Test Driven Development, this concept is still in its infancy, and deserves more exploration. Robert Martin, in his original blog post about TPP, had these specific questions [Martin, R., 2013] (The parenthetical remarks below are his):

> Are there other transformations? (almost certainly)
> Are these the right transformations? (probably not)
> Are there better names for the transformations? (almost certainly)
> Is there really a priority? (I think so, but it might be more complicated than a simple ordinal sequence)
> If so, what is the principle behind that priority? (some notion of "complexity")
> Can it be quantified? (I have no idea)
> Is the priority order presented in this blog correct? (not likely)
> The transformations as described are informal at best. Can they be formalized? (That's the holy grail!)

In [Martin, R. 2014], he demonstrated the TPP concept using a sort routine. In the first half of the illustration, while using TDD to write the routine, he chose a lower-level priority (Assign) over a higher priority transformation (Add Computation). Once the algorithm was complete, the resulting code was a bubble sort with $O(n^2)$ performance. By backing up to a specific point in the TDD process, and this time choosing the higher priority transformation instead, the resulting algorithm was a quicksort with markedly better performance. Are there other, similar algorithms where making a specific decision of choosing one transformation over another would result in a better or worse algorithm result? Do the priorities hold real value in constructing better code from a performance perspective? Martin's guiding principle is that when you have a choice to make, always choose the highest priority transformation over a lower-level transformation. Experimentation in this area might be able to address these questions. Using our analysis tool to track the specific transformations, we could identify choices that led down different paths in the code. The resulting code could be analyzed for performance and then compared to the transformation choices of the authors.

Or coming at the premise from another angle, rather than give the entire list of transformations, one group could be instructed to simply prefer one of the higher-level transformations over other potential choices, while a control group is simply told to follow TDD. Using one of the algorithms identified as having a pivot point in the development, we could investigate whether the experiment group followed the priority and if that led their algorithms to result in better performance. This might provide some validation for the priority aspect of the Transformation Priority Premise.

In summary, many professional developers choose to use TDD and believe that it helps them develop better code. The students in our study had an overwhelmingly positive viewpoint

of the practice, albeit many with caveats. While the studies do not support TDD as being better than practices like Iterative Test-Last, they also do not suggest that practicing TDD is detrimental to producing quality code. More research is needed to more fully understand its contribution to the Software Engineering field.

# REFERENCES


[Allen 2010] Allen, D. (2010, February 2). *More on the synergy between Test-Driven Development and Design by Contract*. Retrieved July 11, 2011, from Software Quality: http://codecontracts.info/2010/02/02/more-on-the-synergy-between-test-driven-design-and-design-by-contract/

[Ambler 2010] Ambler, S. (2010). *How Agile Are You? 2010 Survey Results*. Retrieved June 22, 2011, from http://www.ambysoft.com/surveys/howAgileAreYou2010.html.

[Aniche & Gerosa 2010] Aniche, M. F., & Gerosa, M. A. (2010). Most Common Mistakes in Test- Driven Development Practice: Results from an Online Survey with Developers. *Third International Conference on Software Testing, Verification, and Validation Workshops* , 469-478.

[Astels 2003] Astels, D. (2003). *Test-Driven Development: A Practical Guide.* Upper Saddle River, NJ: Pearson Education, Inc.

[Astels 2006] Astels, D. (2006, March 17). Google TechTalk: Beyond Test Driven Develoment: Behavior Driven Development. Retrieved July 12, 2011, from http://www.youtube.com/watch?v=XOkHh8zF33o.

[Beck 2000] Beck, K. (2000). *Extreme Programming Explained.* Addison-Wesley.

[Beck 2001] Beck, K. (2001, September/October). Aim, Fire. *Software* , 87-89.

[Beck 2003] Beck, K. (2003). *Test-Driven Development: By Example.* Addison-Wesley Professional.

[Beck 2010] Beck, K. (2010, July 8). *CD Survey: What practices do developers use?* Retrieved June 29, 2011, from Three Rivers Institute: http://www.threeriversinstitute.org/blog/?p=541

[Becker et al. 2014] Becker, K., Pedroso, B., Pimenta, M., & Jacobi, R. (2015) Besouro: A framework for exploring compliance rules in automatic TDD behavior assessment. Information and Software Technology, ISSN: 0950-5849, Vol: 57, Issue: 1, Page: 494-508.

[Boehm & Turner 2004] Boehm, B., & Turner, R. (2004). *Balancing Agility and Discipline: A Guide for the Perplexed.* Pearson Education, Inc.

[Böhm & Jacopini 1966] Böhm, C. & Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM, Volume 9 Issue 5, May 1966,* 366-371.

[Brooks 1987] Brooks, Frederick (1987). No Silver Bullet – Essence and Accident in Software Engineering. *Computer, Volume 20 Issue 4, April 1987,* 10-19.

[Colyer 2017] Colyer, Adrian (2017, June 13). *An interesting/influential/important paper from the world of CS every weekday morning, as selected by Adrian Colyer.* Retrieved July 3, 2018, from **the morning paper**: https://blog.acolyer.org/2017/06/13/a-dissection-of-the-test-driven-development-process-does-it-really-matter-to-test-first-or-test-last/

[Desai & Janzen 2008] Desai, C., & Janzen, D. S. (2008). A Survey of Evidence for Test-Driven Development in Academia. *inroads - SIGCSE Bulletin , 40* (2), 97-101.

[Desai & Janzen 2009] Development into CS1/CS2 Curricula. *SIGCSE'09* (pp. 148-152). Chattanooga, TN: ACM.Desai, C., & Janzen, D. S. (2009). Implications of Integrating Test-Driven

[Erdogmus et al. 2005] Erdogmus, H., Morisio, Maurizio, & Torchiano, Marco (2005). On the Effectiveness of the Test-First Approach to Programming. *IEEE Transactions on Software Engineering, 31(3),* 226-237.

[Fowler 1999] Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code.* Boston: Addison-Wesley.

[Fowler 2007] Fowler, Martin (2007). *Mocks Aren't Stubs.* Retrieved November 13, 2018 from https://martinfowler.com/articles/mocksArentStubs.html.

[Fraser et al. 2003] Fraser, S., Astels, D., Beck, K., Boehm, B., McGregor, J., Newkirk, J., et al. (2003). Discipline and Practices of TDD (Test Driven Development). *OOPSLA '03* (pp. 268-269). Anaheim, CA: ACM.

[Freeman & Pryce 2010] Freeman, S., & Pryce, N. (2010). *Growing Object-Oriented Software, Guided By Tests.* Boston, MA: Pearson Education, Inc.

[Fucci & Turhan 2014] Fucci, D. & Turhan, B. Empir Software Eng (2014) 19: 277. https://doi.org/10.1007/s10664-013-9259-7.

[Fucci et al. Apr 2014] Fucci, Davide & Turhan, Burak & Oivo, Markku. (2014). On the Effects of Programming and Testing Skills on External Quality and

Productivity in a Test-Driven Development Context. 10.1145/2745802.2745826.

[Fucci et al. May 2014] Fucci, Davide & Turhan, Burak & Oivo, Markku. (2014). Conformance factor in test-driven development: Initial results from an enhanced replication. ACM International Conference Proceeding Series. 10.1145/2601248.2601272.

[Fucci et al. Sept 2014] Fucci, Davide & Turhan, Burak & Oivo, Markku. (2014). Impact of process conformance on the effects of test-driven development. International Symposium on Empirical Software Engineering and Measurement. 10.1145/2652524.2652526.

[Fucci et al. 2016] Fucci, Davide & Scanniello, Giuseppe & Romano, Simone & Shepperd, Martin & Sigweni, Boyce & Uyaguari, Fernando & Turhan, Burak & Juristo, Natalia & Oivo, Markku. (2016). An External Replication on the Effects of Test-driven Development Using a Multi-site Blind Analysis Approach. 10.1145/2961111.2962592.

[Fucci et al. Nov 2016] Fucci, Davide & Erdogmus, Hakan & Turhan, Burak & Oivo, Markku & Juristo, Natalia. (2016). A Dissection of Test-Driven Development: Does It Really Matter to Test-First or to Test-Last?. IEEE Transactions on Software Engineering. 43. 1-1. 10.1109/TSE.2016.2616877.

[George & Williams 2004] George, B., & Williams, L. (2004). A structured experiment of test-driven development. *Information & Software Technology , 46* (5), pp. 337-342.

[Hannson 2014] Hannson, D. (2014). *TDD is dead. Long live testing.* Retrieved November 13, 2018 from http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html

[Janzen & Saiden 2007] Janzen, D. S., & Saieden, H. (2007). A Leveled Examination of Test-Driven Development Acceptance. *29th International Conference on Software Engineering.* IEEE.

[Jauernig 2010] Jauernig, M. (2010, January 17). *Specification: By Code, Tests, and Contracts*. Retrieved July 11, 2011, from Mind-driven Development: http://www.minddriven.de/index.php/technology/dot-net/code-contracts/specification-by-code-tests-and-contracts

[jBehave 2011] jBehave. (n.d.). *Candidate Steps*. Retrieved July 12, 2011, from jBehave: http://jbehave.org/reference/stable/candidate-steps.html

[Johnson & Paulding 2005] Johnson, PM, Paulding, MG (2005). Understanding HPCS development through automated process and product measurement with Hackystat. In: Second Workshop on Productivity and Performance in High-End Computing (P-PHEC), URL http://csdl.ics.hawaii.edu/techreports/04-22/04-22.pdf

[Kollanus 2010] Kollanus, S. (2010). Test-Driven Development - Still a Promising Approach? *2010 Seventh International Conference on the Quality of Information and Communications Technology* (pp. 403-408). IEEE.

[Kollanus & Isomöttönen 2008] Kollanus, S., & Isomöttönen, V. (2008). Test-Driven Development in Education: Experiences with Critical Viewpoints. *ITiCSE* (pp. 124-127). Madrid, Spain: ACM.

[Koskela 2008] Koskela, L. (2008). *Test Driven: Practical TDD and Acceptance TDD for Java Developers.* Greenwich, CT: Manning Publications Co.

[Kou et al. 2010] Kou, H., Johnson, P. M., & Erdogmus, H. (2010). Operational definition and automated inference of test-driven development with Zorro. *Automated Software Engineering* , 57-85.

[Lappalainen et al 2010]. Lappalainen, V., Itkonen, J., Kollanus, S., & Isomöttönen, V. (2010). ComTest: A Tool to Impart TDD and Unit Testing to Introductory Level Programming. *ITiCSE '10* (pp. 63-67). Ankara, Turkey: ACM.

[Larman & Basili 2003] Larman, C., & Basili, V. R. (2003, June). Iterative and Incremental Development: A Brief History. *Computer* , pp. 47-56.

[Latorre 2014] Latorre, R. (2014, April). Effects of Developer Experience on Learning and Applying Unit Test-Driven Development. IEEE Transactions on Software Engineering 40(4): 381-395.

[Leavens & Cheon 2006] Leavens, G. T., & Cheon, Y. (2006, September 28). *Design by Contract with JML*. Retrieved July 7, 2011, from The Java Modeling Language: http://www.eecs.ucf.edu/~leavens/JML//jmldbc.pdf

[Madeyski 2010] Madeyski, L. (2010). *Test-Driven Development: An Empirical Evaluation of Agile Practice.* Berlin: Springer-Verlag.

[Martin, R. et al. 2008] Martin, R. C., Martin, M. D., & Wilson-Welsh, P. (2008, October). *OneMinuteDescription*. Retrieved June 29, 2011, from

Fitnesse.UserGuide: http://www.fitnesse.org/FitNesse.UserGuide.OneMinuteDescription

[Martin, R. 2009] Martin, R. C. (2009). *As the Tests get more Specific, the Code gets more Generic.* Retrieved May 15, 2014 from Clean Coder: https://sites.google.com/site/unclebobconsultingllc/home/articles/as-the-tests-get-more-specific-the-code-gets-more-generic

[Martin, R. 2011] Martin, R. C. (2011). *The Clean Coder.* Upper Saddle River, NJ: Prentice-Hall.

[Martin, M. 2012] Martin, Micah (2012). *Transformation Priority Premise Applied.* Retrieved May 15, 2014 from http://blog.8thlight.com/micah-martin/2012/11/17/transformation-priority-premise-applied.html

[Martin, R. 2013] Martin, R. C. (Posted 2013, May, Written 2012, December). *The Transformation Priority Principle.* Retrieved May 15, 2014 from https://blog.cleancoder.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html.

[Martin, R. 2014] Martin, R. C. *Clean Code, Episode 24: The Transformation Priority Premise, Parts 1 and 2*. Directed by Robert C. Martin, Clean Coders, January, 2014.

[Meyer 1991] Meyer, B. (1991). Design by Contract. In D. Mandrioli, & B. Meyer (Eds.), *Advances in Object-Oriented Software Engineering* (pp. 1-50). Prentice Hall.

[Meyer 1991b] Meyer, B. (1991). *Eiffel: The Language.* Prentice Hall.

[Meyer 1997] Meyer, B. (1997, May). Practice to Perfect: The Quality First Model. *Computer*, pp. 102-106.

[Mishali 2010] Mishali, O. (2010, 05). *Usings Aspects to Support the Software Process*. Retrieved 3 28, 2011, from http://ssdl-linux.cs.technion.ac.il/wiki/ images/0/03/Mishalithesis.pdf

[Müller & Höfer 2007] Müller, M. M., & Höfer, A. (2007). The effect of experience on the test- driven development process. *Empirical Software Engineering* , 593-615.

[North 2006] North, D. (2006, March). Introducing BDD. *Better Software*.

[Ostroff et al. 2004] Ostroff, J. S., Makalsky, D., & Paige, R. F. (2004). Agile Specification- Driven Development. In J. Eckstein, & H. Baumeister, *Lecture Notes in Computer Science* (pp. 104-112). Berlin, Germany: Springer-Verlag.

[Ricciardi 2009] Ricciardi, S. (2009, June 26). *Introduction to Microsoft Code Contracts with Visual Studio 2008*. Retrieved July 7, 2011, from Steffano Ricciardi: On Software Development and Thereabouts: http://stefanoricciardi.com/ 2009/06/26/introduction-to-microsoft-code-contracts-with-visual-studio-2008/

[Rimmer 2010] Rimmer, C. (2010, September 15). *Introduction*. Retrieved July 11, 2011, from Behaviour-Driven Development: http://behaviour-driven.org/Introduction

[Sauve et al. 2006] Sauve, J. P., Abath Neto, O. L., & Cirne, W. (2006). EasyAccept: A Tool to Easily Create, Run and Drive Development with Automated Acceptance Tests. *AST* , 111-117.

[Schwarz et al. 2005] Schwarz, C., Skytteren, S. K., & Ovstetun, T. M. (2005). AutAT - An Eclipse Plugin for Automatic Acceptance Testing of Web Applications. *OOPSLA '05.* San Diego, CA: ACM.

[Shore 2005] Shore, J. (2005, March 1). *Introduction to Fit*. Retrieved June 29, 2011, from Fit Documentation: http://fit.c2.com/wiki.cgi?IntroductionToFit

[Siniaalto & Abrahamsson 2007] Siniaalto, M., & Abrahamsson, P. (2007). A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage. *First International Symposium on Empirical Software Engineering and Measurement* , 275-284.

[ThoughtWorks 9 May 2014] "Is TDD Dead? [Part I]" *YouTube*, 9 May 2014, https://www.youtube.com/watch?v=z9quxZsLcfo.

[ThoughtWorks 16 May 2014] "Is TDD Dead? [Part II]" *YouTube*, 16 May 2014, https://www.youtube.com/watch?v=z9quxZsLcfo.

[ThoughtWorks 20 May 2014] "Is TDD Dead? [Part III]" *YouTube*, 20 May 2014, https://www.youtube.com/watch?v=z9quxZsLcfo.

[ThoughtWorks 4 June 2014] "Is TDD Dead? [Part IV]" *YouTube*, 4 June 2014, https://www.youtube.com/watch?v=z9quxZsLcfo.

[Wake 2001] Wake, W. (2001, January 2). *The Test-First Stoplight*. Retrieved June 21, 2011, from http://xp123.com/articles/the-test-first-stoplight/

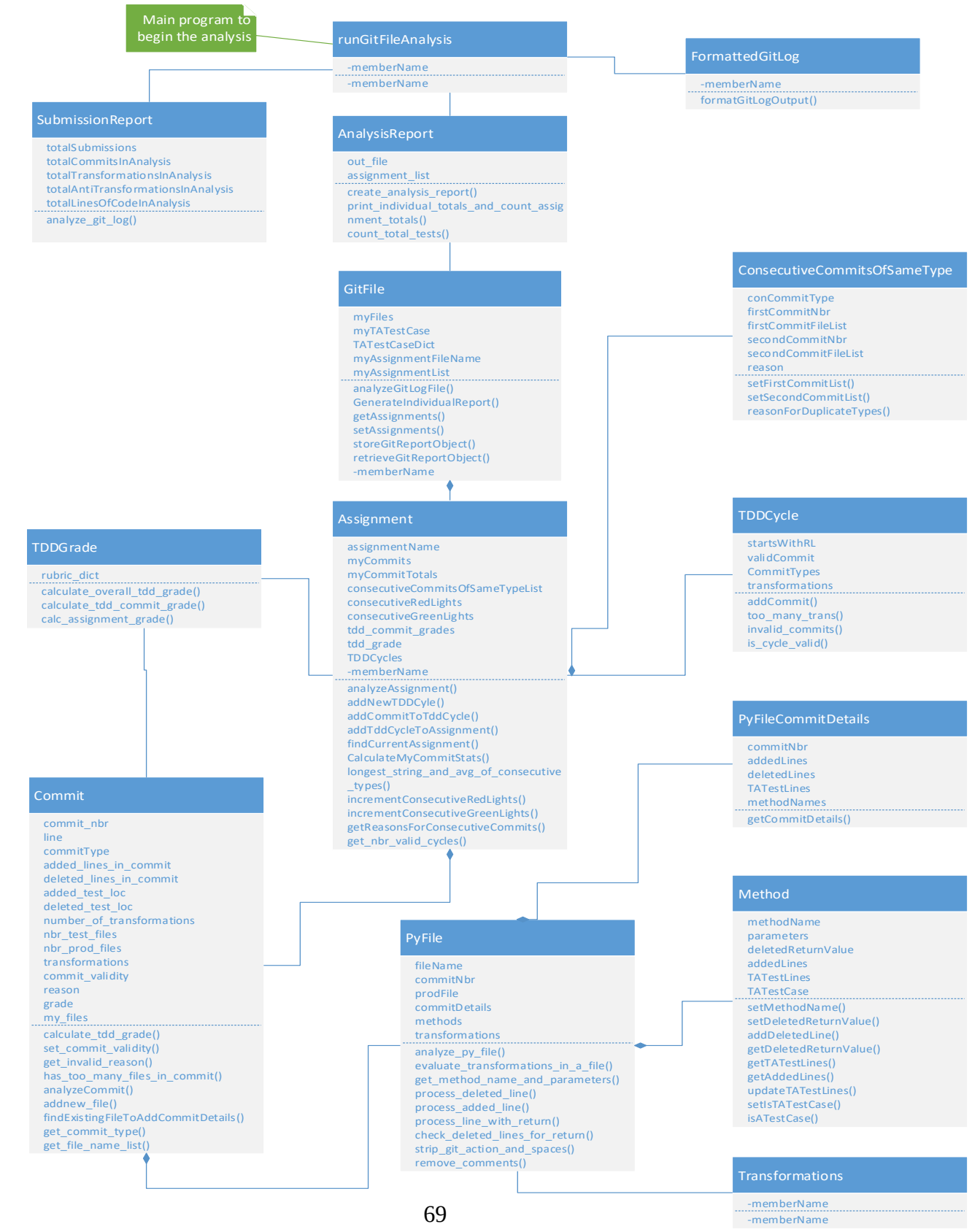
[West & Grant 2010] West, D., & Grant, T. (2010). *Agile Development: Mainstream Adoption Has Changed Agility.* Cambridge: Forrester.

[Wiki 2011] *Framework for Integrated Test*. (2011, February 24). Retrieved July 12, 2011, from Wikipedia: http://en.wikipedia.org/wiki/Framework_for_Integrated_Test

[Wohlin 2012] Wohlin, C., Runeson, P., Hst, M, Ohlsson, M., Regnell, B. & Wesslin, A. (2012). *Experimentation in Software Engineering*, Springer.

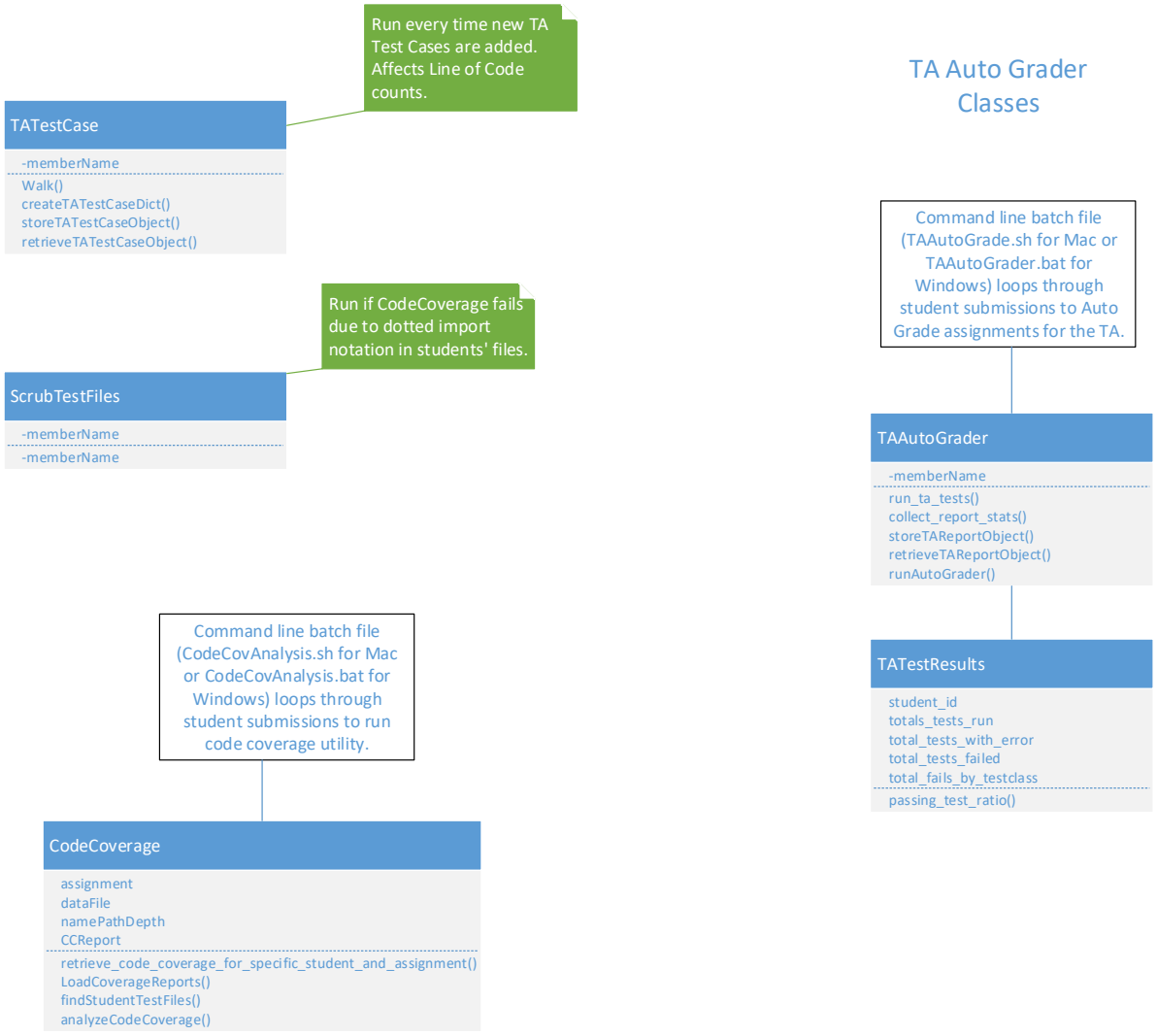
# Appendix A

## TDD Analysis System



# Appendix A

## Supporting Utilities



# Appendix A

Sample Student Report

```
**********
Assignment Name:  assignment4
**********
Number of TDD Cycles:  5
Longest Streak of Consecutive Green Lights:  25   Average Length of Consecutive Green Light Streaks:  6
Longest Streak of Consecutive Red Lights:  1      Average Length of Consecutive Red Light Streaks:  1
---
        Commit Number:1     Commit type: Red Light        Commit TDD Score:  100
Commit Feedback
"Red Light event should only contain test code."

        Added lines:2.  Deleted lines:0.
          Added test lines:0  Deleted test lines:0.
          Test files:3.  Production files:1.  Number of Transformations:  1.


        Transformations to file:  dispatch.py  (Prod)
        Split Flow

.
.
.
.

(Summary portion at the end of the student report)
==========
Total test code lines added:290
Total production code lines added:416
Total test code lines deleted:106
Total production code lines deleted:98
Ratio of test code to production code:0.70:1
TDD Score:  97
Grade Components:  Average Red Light Length - 1;  Average Green Light Length - 6;  Average of TDD Commit Scores - 96
==========
```

# Appendix B

### Classification Rules for Zorro

| Type | Definition | TDD Conformant |
|---|---|---|
| Test-first | Test creation -> Test compilation error -> Code editing -> Test failure -> Code editing -> Test pass<br>Test creation -> Test compilation error -> Code editing -> Test pass<br>Test creation -> Code editing -> Test failure -> Code editing -> Test pass<br>Test creation -> Code editing -> Test pass | Yes |
| Refactoring | Test editing -> Test pass<br>Test refactoring operation -> Test Pass<br>Code editing (number of methods, or statements decrease) -> Test pass<br>Code refactoring operation -> Test pass<br>Test Editing && Code editing (number of methods or statements decrease) -> Test pass | Context Sensitive |
| Test Addition | Test creation -> Test pass<br>Test creation -> Test failure -> Test editing -> Test pass | Context Sensitive |
| Regression | Non-editing activities -> Test pass<br>Test failure -> Non-editing activities -> Test pass | Context sensitive |
| Code Production | Code editing (number of methods unchanged, statements increase) -> Test pass<br>Code editing (number of methods/statements increase slightly (source code size increase <= 100 bytes) -> Test pass | Context sensitive |
| | Code editing (number of methods/statements increase significantly (source code size increase -> 100 bytes) -> Test pass | No |
| Test Last | Code editing -> Test editing -> Test pass<br>Code editing -> Test editing -> Test failure -> Test pass | No |
| Long | Episode with many activities (>200) -> Test pass<br>Episode with a long duration (>30 minutes) -> Test Pass | No |
| Unknown | None of the above -> Test pass<br>None of the above | No |

# Appendix C

## Fall 2018 Survey Results

In a pre-course survey, students were asked a number of questions. One question was related to TDD, and the question and subsequent answer summary is below:

On a scale of 0 to 5 (where 0 = no proficiency and 5 = expert), how would you rate your Test Driven Development (TDD) skills?

| Answer Text | Number of Respondents | Percent of respondents selecting this answer |
|---|---|---|
| 0 | 28 | 28% |
| 1 | 29 | 29% |
| 2 | 25 | 25% |
| 3 | 9 | 9% |
| 4 | 8 | 8% |
| 5 | 1 | 1% |

At the end of the semester, students from the fall 2018 course were surveyed to understand their perception of TDD and their conformance to the process. The questions and subsequent summary of the responses are given below.

TDD improved the quality of my code.

| Answer Text | Number of Respondents | Percent of respondents selecting this answer |
|---|---|---|
| Strongly Agree | 31 | 31% |
| Agree | 51 | 52% |
| Undecided | 10 | 10% |
| Disagree | 5 | 5% |
| Strongly Disagree | 1 | 1% |
| Not Applicable | 1 | 1% |

# Appendix C

I understood what should be built during the red-light TDD cycle and what should be built during the green-light TDD cycle.

| | | |
|---|---|---|
| Strongly Agree | 55 | 56% |
| Agree | 37 | 37% |
| Undecided | 6 | 6% |
| Disagree | 1 | 1% |
| Strongly Disagree | | 0% |
| Not Applicable | | 0% |

I wrote more code during the green-light TDD cycle that what my red-light code tested.

| | | |
|---|---|---|
| Strongly Agree | 15 | 15% |
| Agree | 42 | 42% |
| Undecided | 10 | 10% |
| Disagree | 22 | 22% |
| Strongly Disagree | 8 | 8% |
| Not Applicable | 2 | 2% |

The amount of production code I wrote in each red-light/green-light TDD cycle generally decreased over time (i.e., the number of lines of production code committed to git decreased over the span of an assignment).

| | | |
|---|---|---|
| Strongly Agree | 13 | 13% |
| Agree | 38 | 38% |
| Undecided | 15 | 15% |
| Disagree | 28 | 28% |
| Strongly Disagree | 4 | 4% |
| Not Applicable | 1 | 1% |

# Appendix C

I used TDD to perform low-level design on my assignment.

| | | |
|---|---|---|
| Strongly Agree | 15 | 15% |
| Agree | 52 | 53% |
| Undecided | 15 | 15% |
| Disagree | 12 | 12% |
| Strongly Disagree | | 0% |
| Not Applicable | 5 | 5% |

I understood the purpose of TPP.

| | | |
|---|---|---|
| Strongly Agree | 23 | 23% |
| Agree | 54 | 55% |
| Undecided | 12 | 12% |
| Disagree | 5 | 5% |
| Strongly Disagree | 2 | 2% |
| Not Applicable | 2 | 2% |
| No Answer | 1 | 1% |

I understood how to apply TPP.

| | | |
|---|---|---|
| Strongly Agree | 18 | 18% |
| Agree | 46 | 46% |
| Undecided | 21 | 21% |
| Disagree | 8 | 8% |
| Strongly Disagree | 3 | 3% |
| Not Applicable | 3 | 3% |

I used TPP to guide my TDD efforts.

| | | |
|---|---|---|
| Strongly Agree | 9 | 9% |
| Agree | 43 | 43% |
| Undecided | 28 | 28% |
| Disagree | 12 | 12% |
| Strongly Disagree | 2 | 2% |
| Not Applicable | 5 | 5% |

# Appendix C

Which of the following best describes your adherence to TDD by the last programming assignment?

| | | |
|---|---|---|
| Very strict | 20 | 20% |
| I slipped a few times | 40 | 40% |
| I slipped occasionally | 23 | 23% |
| I slipped often | 9 | 9% |
| I generally wrote code first then the tests afterward | 7 | 7% |

How difficult was it to use TDD by the last programming assignment?

| | | |
|---|---|---|
| 0 - very easy | 21 | 21% |
| 1 - easy | 24 | 24% |
| 2 - neutral | 31 | 31% |
| 3 - difficult | 18 | 18% |
| 4 - very difficult | 4 | 4% |
| 5 - not used | 1 | 1% |

How likely are you to voluntarily use TDD in the future?

| | | |
|---|---|---|
| 0 - No way | 5 | 5% |
| 1 - Maybe in limited circumstances | 22 | 22% |
| 2 - Perhaps, but I'm undecided | 20 | 20% |
| 3 - Yes, but not regularly or faithfully | 35 | 35% |
| 4 - I would make it part of my baseline skillset | 16 | 16% |
| No answer | 1 | 1% |