Quality Metrics of Test Suites in Test-Driven Designed Applications

Hessah Alkaoud
Department of Computer Science
University of Colorado at Colorado Springs, USA
Email:halkaoud@gmail.com

Kristen Walcott-Justice
Department of Computer Science
University of Colorado at Colorado Springs, USA
Email:kjustice@uccs.edu

***Abstract*—As software engineering methodologies continue to progress, new techniques for writing and developing software have evolved. One of these methods is known as Test-Driven Development (TDD). In TDD, tests are written first, prior to code development, aiding in both maintenance and software evolution efforts. Due to the nature of TDD, no code should be written without first having a test to execute it. Thus, in terms of code coverage, the quality of test suites written using TDD should be high. However, it is unclear if test suites developed using TDD exhibit high quality in terms of other metrics such as mutation score. Moreover, the association between coverage and mutation score is little studied.**

In this work, we analyze applications written using TDD and more traditional techniques. Specifically, we demonstrate the quality of the associated test suites based on two quality metrics: 1) structure-based criterion represented by branch coverage and statement coverage, 2) fault-based criterion represented by mutation scores. We learn that test suites with high branch test coverage will also have high mutation scores, and we especially reveal this in the case of TDD applications. We found that Test-Driven Development is an effective approach that improves the quality of the test suite to cover more of the source code and also to reveal more faults based on mutation analysis.

I. Introduction

In software engineering, writing tests is vital to the development and maintenance processes because it verifies the correctness of the implementation of the software through program execution and analysis. During development, testing is a way to verify the completeness of the requirements. Testing also helps to ensure that the software has achieved an appropriate level of quality. In maintenance and software evolution, software testing is important when adding new functionality or refactoring to check the correctness of the changes. Moreover, testing is vital to verify that new features do not cause the program to regress during maintenance work. Due to this importance, new methods of developing programs and tests together have been devised to improve the evolution of software systems. One of these techniques is known as Test-Driven Development (TDD). TDD practices require that tests be written prior to the writing of code. In this methodology, no code should be written without first having a test case to cover such code. Thus, the quality of the code as measured using coverage should be high for TDD applications. However, there is little research in comparing the code coverage of such applications to other code quality metrics.

Traditionally, tests are written in an ad hoc fashion by the developers after the code is completed or in tandem with development by a quality assurance team. In TDD, however, tests are written prior to any code. The TDD process is also known as test-first or red-green-refactoring. Today TDD is being widely adopted in industry, including large software firms such as Microsoft and IBM [29]. TDD has also gained in popularity with the introduction of the eXtreme Programming XP methodology [7], and it is sometimes used as a stand-alone approach for software engineering maintenance tasks such as adding features to legacy code [13]. In TDD, the software is often developed in short iterative cycles and in each cycle, testing is repeated for each testable component of the program. The pattern of the cycles can be explained as the following:

- Write the test and run it to see that it fails (Red).
- If the test fails, implement the code that will cause the test to run and pass the test (Green).
- Alter the design of both the test and the code (Refactoring).

Refactoring is a process of altering the code for the purpose of enhancing its design, readability, maintainability, etc... without violating existing code functionality [14].

Once a test suite is written and complete, it is also important to evaluate the quality of the test suite. Many program-based adequacy criterion have been developed to determine whether or not a test suite adequately tests the application [35]. Program-based adequacy criterion exploit both structure-base criterion and fault-based criterion. A structure-based criterion requires the creation of a test suite that solely requires the exercising of certain control structures and variables within program [23]. Fault-based test adequacy criterion, on the other hand, attempts to ensure that the program does not contain the types of faults that are commonly introduced into software systems by programmers [12], [36]. One of the most common types of test quality evaluation in both industry and academic work is based on structurally-based criterion which is commonly analyzed in terms of statement or branch coverage [35].

When developing programs using TDD, code is only written when a test shows that it is needed. In theory, no production code in TDD should be developed unless it has at least one associated test. Therefore, the code coverage of TDD-based tests should be high for their associated applications by nature. Both the literature and practice indicate that the use of TDD yields several benefits. For instance, TDD leads to improved test coverage [5]. However, statement coverage and branch coverage have been shown to not necessarily give firm

evidence of high quality test suites [36].

In recent years, mutation testing/scores have been used as another method for evaluating test suite quality. Mutation testing is a fault-based technique which measures the fault-finding effectiveness of test suites on the basis of induced faults [12], [17]. Mutation testing is a well-known technique to design a new software tests or to evaluate the quality of existing software tests and test suites. The idea of using mutants to measure test suite adequacy was originally proposed by DeMillo et al. [12]. Mutation testing involves seeding faults in the source program. Each altered version of the source is called a mutant. When a test reveals the mutant then the mutant said to be killed. The ratio of killed mutants/generated mutants is known as the mutation score. In mutation testing, in a simple statement such as `if (a < b)`, the < sign will be replaced with all other possible relational operators such as `>, <=, >= , ==, !=`. The use of mutation operators yields results in the empirical assessment of quality for current testing techniques [3].

Given the rise in popularity of TDD approaches, the question is whether TDD actually promotes high quality test suites or not? In this work, we investigate if TDD methodology promotes higher test suite quality than traditional test design approaches. To show this, we analyze and compare three quality metrics: branch coverage, statement coverage, and mutation score when applied to two sets of programs: 1) Programs that were designed using the TDD methodology and 2) programs designed using standard testing techniques. In this work, we use the term *standard testing technique* to refer to any tests that were developed using methodologies that are not TDD oriented. We first analyze the size and complexity of the programs. Then we demonstrate the statement and branch coverage quality of the existing test suites. Next, we apply a mutation testing tool to determine the mutation score of the test suite. Finally, we analyze the association between test coverage and mutation scores in estimating overall test suite quality.

Our results show that test-driven development is significantly different from the other development approaches on mutation score. We demonstrate a relationship between the mutation score and traditional coverage metrics, statement coverage and branch coverage, and that the correlation with branch coverage is stronger. A percentage of 54% of the programs reflects a relationship with statement coverage and more as 62% reflects a relation with branch coverage.

In summary, the main contributions of this research are as follows:

- An examination of the quality of TDD test suites based on two quality metrics, code coverage and mutation score.
- An empirical evaluation of the quality of TDD test suites considering mutation and coverage scores.
- A discussion of the observed trends and comparison of the two quality metrics: coverage and mutation.

II. Background

In this section, we discuss background regarding test-driven development practices and two program-based criteria that are used for evaluating test suite quality: code coverage and mutation scores.

A. TDD Practices

Test-driven development is a software development process that is based on repeating a very short development cycle. There are several variations of test-driven development, although the main method is called Test-First or testing via a red-green-refactoring cycle. Refactoring is a process of altering the code for the purpose of enhancing its design, readability, maintainability, etc. without violating existing code functionality [14]. The TDD process is performed in the following six steps:

- Add test code for a new functionality or a specific desired improvement.
- Run the test code and see the test fail (red).
- Implement the production code according to the test.
- Run all tests again and make sure they pass (green).
- Repeat 3, 4 until the test passes.
- Refactor the production code and the test.
- Repeat steps 1 through 7 to program development completion.

Test-driven-development implies quality benefits to the software. It is true that TDD requires more test-based code than without TDD because of the thoroughness of the unit test code, but the large numbers of tests also contributes to limiting the number of defects in the code. The frequent and early nature of the TDD testing helps to reveal defects early in the development process, reducing the number of bugs that may be observed later in time. Moreover, TDD tests that are written prior to code development and are run repeatedly are more likely to detect any regression of the code's behavior. This allows developers to discover problems that may arise when a later change in the implementation unexpectedly affects other functionality.

In software engineering, behavior-driven development BDD is another software development process that is built based on test-driven development (TDD). The original developer of BDD, Dan North [30], came up with the notion of BDD because he was dissatisfied with the lack of any specification within TDD of what should be tested and how. Astels notes, "Behavior-Driven Development is what you were doing already if you're doing Test-Driven Development very well [6]." In this work, we will only focus on TDD-specific applications.

B. Value of Coverage

Code coverage metrics are common for evaluating the quality of test suites. Coverage metrics check how many elements of the source code have been *executed* by the test. Code coverage falls under structurally-based criterion that requires the creation of a test suite that solely requires the exercising of certain control structures, statements, or variables within the program[23]. There are many coverage metrics types that have been used and proposed, some of which include statement coverage, branch coverage, and condition coverage. Branch coverage criteria and statement coverage criteria are the two most popular examples of structural adequacy criteria that are used in existing coverage analysis tools and by industry[35].

Statement Coverage metrics require executing each statement in the program at least once during testing. It can be defined as follows:
Definition: the coverage of all nodes in the flow graph [36].

Branch Coverage metrics require the execution of each control transfer (true,false) of each branch (e.g. if statements, loops) in the program under test at least once. Branch coverage can be defined as follows:
Definition: the coverage of all edges in the flow graph [36].

To distinguish the differences between statement and branch coverage, consider the following code:

```
public int returnInput(int input, boolean
    condition1, boolean condition2, boolean
    condition3) {
  int x = input;
  int y = 0;
  if (condition1)
    x++;
  if (condition2)
    x--;
  if (condition3)
    y=x;
  return y;
```

with the following associated test case:
Test 1: `shouldReturnInput(x, true, true, true)`

In this case, statement coverage would be 100% as every statement would be executed. Test case 1 will guarantee that each statement in the code will be exercised. However, the branch coverage is only 50% because branch coverage requires checking that both edges of each branch execute. To improve the branch coverage to 100%, a second test case could be added as follows:
Test 2: `shouldReturnInput(x, false, false, false)`

From these examples of test cases, we observe that branch coverage implies statement coverage, because exercising every branch leads to executing every statement [11]. Branch coverage is a good indicator of test quality, but still branch coverage is not enough to truly estimate test quality because it does not imply the execution of all possible paths. Consider the following example: This simple method has two tests, and the branch coverage calculation reports 100% branch coverage even though there is no test to catch when `i ==0`.

```
public static String foo(int i) {
  if ( i <= 0 ) {
     return "foo";
  } else {
     return "bar";
  }
}
@Test
public void shouldReturnBarWhenGiven1() {
  assertEquals("bar", foo(1));
}
@Test
public void shouldReturnFooWhenGivenMinus1() {
  assertEquals("foo", foo(-1));
}
```

Therefore, high code coverage does not necessarily imply full coverage and thorough testing overall.

C. Mutation Score

Mutation analysis is an alternative approach for determining code quality. Mutation analysis was originally introduced in [12], [8]. Mutation testing is a fault-based technique which measures the fault-finding effectiveness of test suites on the basis of introduced faults [12], [17]. Fault-based test adequacy criterion attempt to ensure that the program does not contain the types of faults that are commonly introduced into software systems by programmers [12]. Mutation testing involves modifying a program's source code in small ways in order to seed in faults [31]. Each modified version of the program is called a *mutant*, and each mutant includes only one potential fault. Fault planting is based on a set of mutation operators. For example, mutation operators may include:

- Statement removing.
- Replace the boolean sub-expression with true and false.
- Replace the arithmetic operation with another, e.g. + with *, - and /.
- Replace the boolean relation with another, e.g. $>$ with $\geq$, == and $\leq$.
- Replace the variables with another declared variable in the same scope that has the same type.

In the test execution process, if the test revealed the introduced fault then it is said that the test *killed* the mutant. Relating the total number of the killed mutants to the total number of generated mutant is a powerful way to measure the quality of the test suites. The number of mutants killed given the number of mutants create results in the calculation of a mutation score, which is a testing criterion to measure the effectiveness or ability of a test suite to detect faults. The equation to calculate mutation score can be written as follows:

$$mutation\_score = \frac{number\_of\_mutants\_killed}{total\_number\_of\_mutants}$$

The mutation score determines whether the test is good enough to detect seeded faults. Mutation testing is not meant as a replacement for code coverage; rather, it is a complementary approach that is useful for finding code that is covered by tests based on coverage metrics but may still not be effectively tested [26]. While traditional code coverage measures how much of code is executed by the test suite, the mutation score measures the ability of tests running on developed code to reveal random faults. The simplest example of the usefulness of mutation adequacy is this conditional logic statement: `if (a < b)`. It is useful to include mutation operators that manipulate the relational operators found within a program. Thus, the '$<$' operator may be replaced will other possible relational operators such as ($>$,$\leq$,$\geq$,==,$\neq$) [16]. Ideally, *mutation* operators should produce the kinds of mutant programs that software engineers are most likely to create.

Note that some mutants may not actually introduce faults, but rather create an equivalent program given the feasible paths in the program. In this work, we assume that mutants may or may not be equivalent to the original program, given the mutation tools we use. Thus, a mutation score of 100%

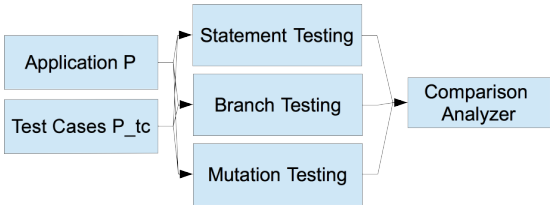

Fig. 1. The analysis approach.

may not be possible for all programs. If equivalents could be easily assessed, a mutation score of 100% would be the goal. However, in this work, we will aim for "high" mutation scores, where "high" will be judged relative to the mutation scores of other test suites and programs.

III. Analyzing TDD Approaches

In this research, we compare the statement coverage, branch coverage, and mutation scores of programs that have been developed using TDD and other practices. Our process can be seen in Figure 1. Each application and its associated test suite is executed using a coverage analyzer tool and a mutation analyzer tool. Given those results, we compare the coverage and mutation scores for different types of applications.

We expect that TDD approaches will exhibit high code coverage in terms of both statement and branch coverage, given the practices of TDD. However, there is little known about the correlation between coverage and mutation score. Applications that were developed using other methods are analyzed with the same tools. We expect that coverage and mutation scores will be lower than those developed using TDD, especially for applications that are not developed within a large community or are not widely used.

IV. Empirical Evaluation

In this research, we compare the quality of TDD test suites to other those developed using more standard testing techniques. First, we observe the trends of quality in test suites that are written using TDD methodology. Next, we inspect the trends of quality when generating test suites using other development techniques. Then, we compare the quality of TDD test suites to other test suites developed using traditional techniques and observe the overall impact of code coverage criteria and mutation score criteria. In summary, we analyze:

- The relationship between coverage and mutation scores when applied to programs developed using TDD
- The relationship between coverage and mutation scores when applied to programs developed using non-TDD methods
- Potential relationships between coverage and mutation scores given program properties

Overall, we examine how traditional software testing methodologies compare to newer testing methodologies, specifically test-driven-development (TDD) with regard to test suite quality. We answer three key questions:

Q1: What is the impact of TDD on test quality metrics?
In the way the TDD process is designed, it directly implies good coverage, as described in Sections II. However, there is a little understanding in current research as to how quality can be measured other than coverage. Another potential quality metric includes mutation metrics and, more specifically, mutation score. If we know that the code coverage is high in TDD applications, does that imply that the mutation score is also high, nearing 100%? We have also observed that TDD applications claim high "code coverage." In most work, it is unclear if code coverage refers to statement coverage, a lesser metric, or to branch coverage. Thus, we also will also analyze if TDD test suites are creating high quality test suites in terms of both statement and branch coverage.

Q2: What are the differences in the quality of test suites between TDD and other testing methodologies?
Once both quality metrics, coverage and mutation score, are taken into consideration, we next ask what qualities traditional test suites exhibit. Traditional test suites are often developed in ad hoc fashions. Some may develop the tests in tandem with the writing of code, or the code may be written and then passed off to an external (away from the developer) Quality Assurance (QA) team for evaluation. QA teams often test the code in an ad hoc, white-box, behaviorally or integration testing driven way. For example, "if I click this button, then I should arrive at this view and see these results." Tests written in this way account little for individual units of code or subtle integrations between components.
We examine traditionally created test suites for applications and evaluate the code coverage and mutation scores for each. We hypothesize that the coverage and mutation scores will be lower than that of applications generated using TDD.

Q3: What are the trends in test quality metrics?
There are also some debatable questions that are related to the common quality metric, coverage. Coverage metrics, specifically those based on structure alone, are cheap to evaluate and thus, are most frequently used in industry. For example, the avionics industry standard `DO-254` [34] demands that close to 100% statement coverage be achieved. The avionics industry standard `DO-178B` [33] and automotive industry standard `IEC 61508` [2] detail similar requirements. However, it is unclear if these metrics alone actually lead to higher fault finding ability.

From our results from Q1 and Q2, we analyze the relationship between test suite quality metrics, code coverage and mutation score, across test methodologies that were used to develop the test suites. In our research, we answer two key questions: 1) What is the relationship between coverage and mutation score? 2) Does the comparison differ between different types of coverage? Finally, we discuss if it is enough to judge the test suite quality by traditional coverage metrics, or if other metrics, such as mutation score, should be considered. Lastly, we discuss the differences observed between TDD generated applications versus traditionally generated applications.

A. Experiment Design and Metrics

In this section, we describe the experiment design including tools, metrics, and benchmarks that are used in our work, and then we explain the experiment evaluation.

1) Experiment design: In our experiments, two sets of benchmarks will be evaluated using three key quality metrics: statement coverage, branch coverage, and mutation score. In

TABLE I. TEST APPLICATIONS AND STATISTICS.

Benchmarks			Test			Program proprieties		
name	version	group	**Statement coverage**	**Branch coverage**	**Mutation score**	**SLOC**	**Test-LOC/ Source-LOC**	**CCN**
Tyburn	1.1	TDD	92	88	86	554	1.2	1.33
Jbehave-core	3.9	TDD	88	81	100	9726	0.8	1.57
Helium	0.1	TDD	94	99	92	1018	1.7	1.95
Trove	3.0.3	Non_TDD	7	7	59	2080	5.0	2.24
Commons-lang	3.2	Non_TDD	94	90	75	15825	1.8	3.31
Commons-io	2.4	Non_TDD	86	85	82	6037	2.0	2.52
Jdom	2.0.0	Non_TDD	92	86	77	12038	1.4	2.58
Numerics4j	1.2	Non_TDD	99	97	81	1836	1.5	2.27
lavalamp		Non_TDD	98	35	72	1039	1.3	1.28
netweaver		Non_TDD	16	2	15	17953	0.1	2.7
jaxen	1.1.6	Non_TDD	79	59	50.41	8428	0.8	3.21

addition to these metrics, the benchmarks will be analyzed for their size, complexity, and other standard metrics. Each metric will be evaluated using common tools. Experiments were executed using a 3rd Gen Intel Core i7-3770 processor 3.40GHz with 16GB DDR3 SDRAM.

2) Benchmarks: The experiments are executed on Java language open source projects. There are a number of challenges in selecting such applications. Many open source projects are not well tested. In our work, we focus on a set of open source projects that are often used in software engineering research, the SF100 [1], as well as several frequently used programs from Apache Commons. The SF100 is a set of 100 open source Java projects selected from SourceForge. We also considered programs that are known to be developed using TDD [25]

In our experiment, we have incorporated three TDD projects. These benchmarks include `Tyburn`,`Jbehave` and `Helium`. We also evaluate five non-TDD projects including `trove`, `Commons Lang`, `JDOM`, `Commons IO`, `numerics4j`, and `jaxen` along with `netweaver` and `lavalamp` from SF100.

3) Metrics: We measure branch coverage, statement coverage, and mutation score on all projects. Each of these metrics is described in Section II. Two main tools are used for these measurements: Jacoco and MAJOR. Jacoco [18] is an open source coverage library for Java, which has been created by the EclEmma team. Jacoco reports code coverage analysis (.e.g. line, branch, instruction coverage) from the bytecode. MAJOR [22] is a mutation testing and score tool developed by René Just. The Major mutation framework enables fundamental research on mutation testing as well as efficient mutation analysis of large software systems. The mutation framework provides the following three components:

- Compiler-integrated mutator
- Mutation analysis back-end for JUnit tests
- A domain specific language to configure the mutation process

We also calculate program and test attribute measurements to help us recognize the distinct trends in the code projects. We measure project characteristics including: number of non-commented source code lines (SLOC), complexity of the source code, and test lines of code per source line of code (test-LOC/source-LOC). These calculations are performed using JavaNCSS. JavaNCSS [24] is a command line tool that can be used to count the non-commented lines of source code, as well as the McCabe cyclomatic complexity of the source code.

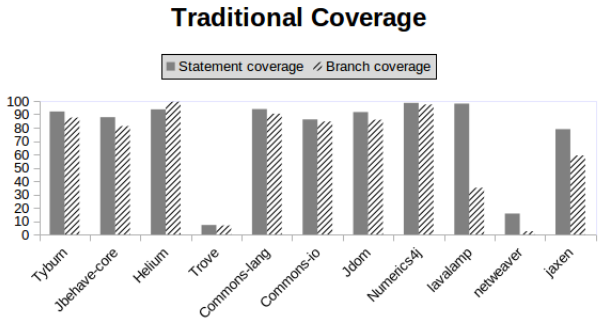

Fig. 2. Statement and Branch coverage of benchmarks

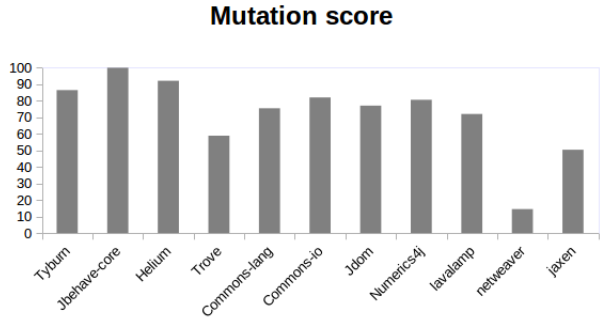

Fig. 3. Mutation score of benchmarks with standard deviation

The cyclomatic complexity [27] metric for software systems is adapted from the classical graph theoretical cyclomatic number and can be defined as the number of linearly independent paths in a program. Larger numbers of lines of code and increased complexity can imply poor design and greater challenges to thorough testing.

B. Results

We evaluate the number of source lines of code (SLOC), test to source lines of code (test-LOC/source-LOC), and the average cyclomatic complexity (CCN) of production code for all projects. Also, we evaluate the statement coverage, branch code coverage, and mutation scores of all test suites of our selected benchmark programs, as can be seen in Table I.

1) TDD vs Traditional: In this section we aim to answer these two questions:
Q1: What is the impact of TDD on other test quality metrics?

Tests of Normality

	group	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
		Statistic	df	Sig.	Statistic	df	Sig.
Branch Coverage	Non-TDD	.262	8	.113	.858	8	.115
	TDD	.225	3	.	.984	3	.756

Tests of Normality

	group	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
		Statistic	df	Sig.	Statistic	df	Sig.
Mutation Score	Non-TDD	.265	8	.105	.801	8	.029
	TDD	.204	3	.	.993	3	.843

Fig. 4. Results of normality test

Q2: What are the differences in the quality of test suites between TDD and other testing methodologies?

Figure 2 illustrates the statement and branch coverage for each program including programs of both groups the TDD and the non-TDD programs. From these results, we observe that the TDD applications (`Tyburn`, `Jbehave`, `and Helium`) exhibit high branch coverage of more than 80% and statement coverage of more than 88%. Several of the non-TDD applications also exhibit very high branch and coverage scores, namely `Commons-lang`, `Commons-io`, `Jdom`, and `Numerics4j`. We suspect that these higher coverage metrics are due to the fact that these four applications are under active development in large communities that have high standards for acceptance of new features. `Trove` has a high Test-LOC/Source-LOC because nearly all testing effort is focused on one significant package. The other packages are mostly ignored, thus yielding low coverage results when the entire application is considered.

We next measured the mutation scores using MAJOR [19], [20], [22]. In Figure 3, we visualise the results of all benchmarks. For the TDD applications under consideration, `Tyburn` (non-TDD) exhibited the lowest mutation score at 86% whereas `Jbehave` (TDD) had a score of 100%. On the other hand, the mutation score in non-TDD programs ranged from 15% in `Netweaver` to 82% for `commons-io`. Moreover, from Figures 3 and 2), we see that the quality of tests developed using TDD is not only high in terms of statement and branch coverage but also in terms of the mutation score metric.

Then, we explored if TDD and non-TDD applications are significantly different in term of their tests. To show this, we compare the means of the two groups using a two sample t-test for the normal groups and a non-parametric test on the data that the normality test shows as not normal. Thus, we first test the normality of our data using the Shapiro-Wilk test. We explored the normality of two variables: branch coverage and mutation score. We test both the mutation score metric and the branch coverage metric assuming that branch coverage will already includes statement coverage [11].

As show in Figure 4, specifically the normality indicator p-value in the last column, the results of the normality test shows that the data is normal for both groups,TDD and non-TDD, except in the mutation score of non-TDD group which was 0.029,because the value is less than 0.05, it indicates that the there is an evidence that the data tested are not from a normally distributed population.

Next, according to the normality test, we compare the means of the two groups using either a ”two sample t-test” for the normal data or ”non-parametric test” for the data that is not normally distributed. Comparing the two groups of applications in terms of the branch coverage metric, the p-value was 0.059 which is not significant, but is approaching statistically significant results. Therefore, we can speculate that with more TDD programs we might be able to conclude a statistically significant difference between the application groups, where the TDD group has higher branch coverage than the non-TDD group. On the other hand, the Shapiro-Wilk test shows that the mutation score of non-TDD group is not normally distributed. Therefore, to compare the means, we used a non-parametric two-Sample test on this data. Namely, we applied the non-parametric independent samples in a Mann-Whitney U test and learned that the results of this tests had a p-value equal to 0.012. Therefore, we can conclude that the mutation score of the two groups are significantly different, where the TDD approach scored higher than the non-TDD approaches.

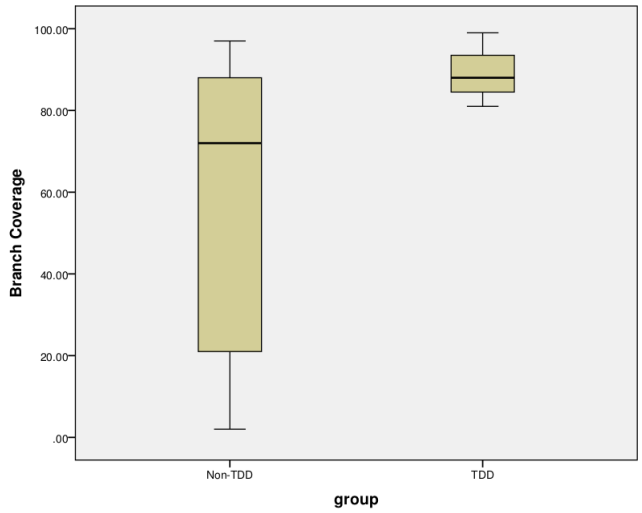

Fig. 5. Differences in branch coverage between TDD and Non-TDD applications

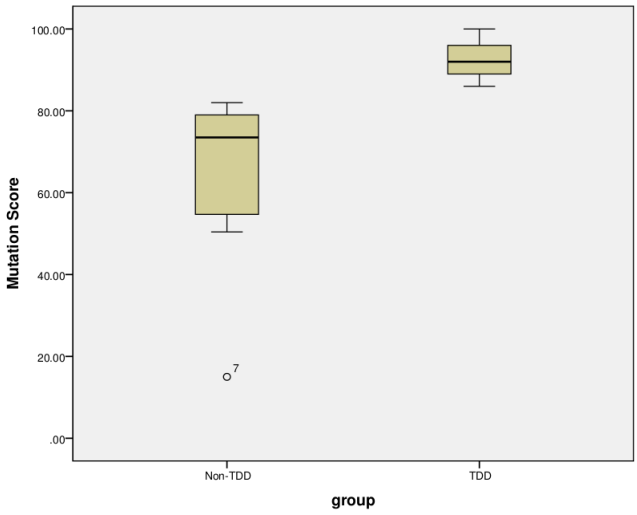

Fig. 6. Differences in mutation score between TDD and Non-TDD applications

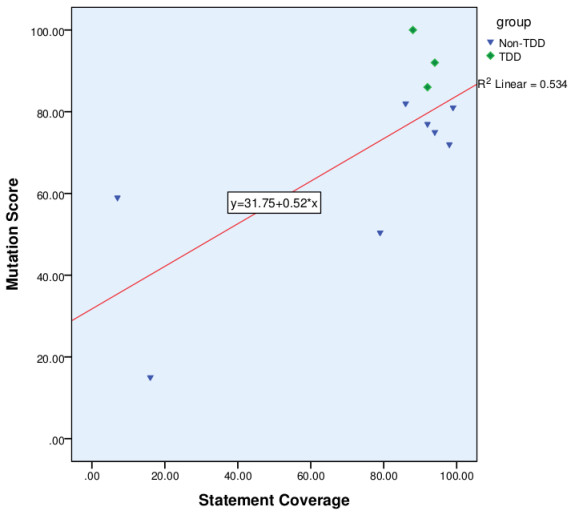

Fig. 7. The relationship between mutation score and statement coverage.

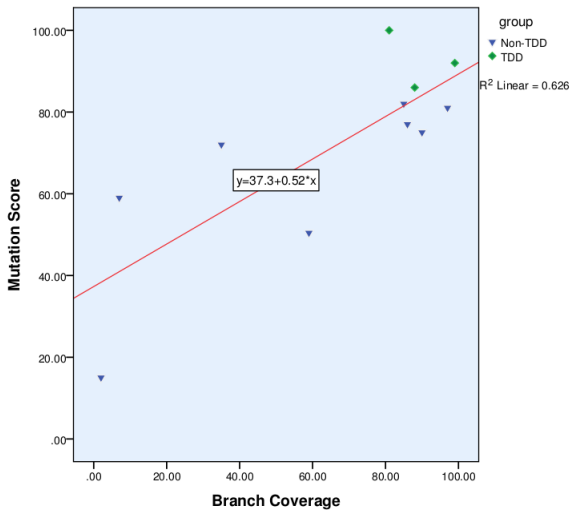

Fig. 8. The Relationship between mutation score and branch coverage

Figures 5 and 6 demonstrate the relationship between mutation score and the two coverage types. As seen in the first box plot in Figure 5, the mean, the standard deviation, and the confidence interval for TDD group on branch coverage is (M=89.33, SD=9.07 , CI 95%: 66.79 to 111.87) while the values for the non-TDD group are (M=57.63, SD=38.40, CI 95%: 25.52 to 89.73). The second plot, shown in Figure 6, demonstrates that the TDD group scores higher based on mutation score (M=92.67, SD=7.02 , CI 95%: 75.22 to 110.11) than the non-TDD group (M=63.93, SD=22.61, CI 95%: 45.03 to 82.83), where M, SD, and CI stand for the mean, standard deviation, and confidence interval respectively.

2) Mutation Score versus Coverage: We use the correlation coefficient Pearson's r test to analyze the strength of the correlation between the metrics. The correlation coefficient is a value that ranges between $-1 \leq r \leq 1$. In our results shown in Table II, the correlation r value for mutation score is .791 with branch coverage and 0.731 with statement coverage. The correlation coefficient indicates the existence of a relationship between these test quality metrics. Also, when analyzing the statistical significance of the resulting correlation, p-value of the correlation with branch coverage is 0.004 and it is 0.0.011 for the correlation with statement coverage (shown in the second row of Table II). Therefore, the correlation is statistically significant because the p-value is below 0.05.

We also calculate the coefficient of determination R-squared. As shown in Figure 7, the mutation score is dependent on the statement coverage, and it can be predicted based on the statement coverage, the independent variable. As visualized in the figure, the rounded R-squared value of 0.54 indicates an existent relationship between the statement coverage and the mutation score, where almost 54% of the variation in statement coverage can be explained by the variation in mutation score.

Similarly, Figure 8 reflects a relation where the coefficient of determination is 0.62. Thus, 62% of the programs that had high branch coverage also achieved a high mutation score.

Despite the fact that 62% of the programs can be explained by the relationship between mutation score and branch coverage, 36% of the programs in our study cannot be reliably explained with regard to this relation. To illustrate, in `lavalamp` branch coverage scores were 35% whereas the mutation score reaches 75%. We believe this was because the statement coverage of this program was as high as 98%. Other programs cannot be explained by either by branch or statement coverage. For example, in `common-lang` the branch coverage of `common-lang` is equal to 90% and the statement coverage is 94%. However, its mutation score is 75%.

V. Discussion

In TDD, we found a trend between test-LOC per Source-Line of Code with high branch coverage and statement coverage. In other words, as more tests are written in TDD, branch and the statement coverage improve. The R-squared value is a linear relationship of 0.89 corresponding to statement coverage and 0.99 to branch coverage. This indicates that TDD programs tend to focus test writing based on code branches. It is known that executing every branch executes every statement as well unless there is a break or a jump [18]. Therefore, as a consequence of the high branch coverage in TDD programs, the statement coverage and the mutation score of TDD programs is also high.

Mutation score metrics have a relationship not only with branch coverage but also with statement coverage. Both statement and branch coverage will complimentarily predict mutation score. Looking at `Lavalamp` as an example,the statement coverage is 98%, the branch coverage is 35% and the mutation score is 72%. Despite the low branch coverage of `LavaLamp`, which was 35%, the mutation score of `Lavalamp` is 72%, which is not low. This could at first glance bring up a question about the relationship between the mutation score and the branch coverage. However, we show in our results that the mutation score can be predicted from both branch and statement coverage in most cases. In this case, the non-killed mutants of `Lavalamp` were due to the uncovered branches that have not been tested.

TABLE II. THE RELATIONSHIP BETWEEN MUTATION SCORE METRIC WITH STATEMENT AND BRANCH COVERAGE

		Branch Coverage	Statement Coverage
Mutation Score	Pearson Correlation	.791**	0.731*
	Sig. (2-tailed)	0.004	0.011
	N	11	11

*. Correlation is significant at the 0.05 level (2-tailed)
**. Correlation is significant at the 0.01 level (2-tailed).

Our results conclude that writing more test cases does not necessary ensure good test suite quality on both metrics. We found no relation between Test-LOC/source-LOC. The coefficient of determination of Test-LOC with the mutation score coverage was 0.02 and was 0.05 with branch coverage. Writing more tests could give little extra value, such as in `trove` where the TLOC/SLOC is the highest among the projects as seen in Table I, but the tests did not reflect a very high mutation score or coverage because most of `trove`'s tests were written for one or two modules of the source, leaving the other modules not tested. Also, `Lavalamp` exhibited low branch coverage at 35% while its Test-LOC/source-LOC is not low. One possible explanation is that most of the tests were written to cover the statements, not the branches. There is approximately one line of test per line of source code. However as we discussed in Section II, if the tests cover 100% of the lines, that does not indicate anything about branch coverage. When most of the branches of a program are covered, its statements are mostly covered.

We also want to know if the program size in our experiments impacts the mutation score that we are getting for each program. We found that larger sized programs do not correlate to lower mutation scores. Only 24% of applications can be explained by this connection. For example, `netweaver` falls under one of the 24% of programs in our benchmarks that can explained from this relation. However, this relation also is questionable since the most accurate interpretation of the low mutation score of `netweaver` is not because of the program size but because `netweaver` was not tested enough according to our calculated test-line-of-code to source-line-of code metric, which was only 0.1 for this project.

VI. THREATS TO VALIDITY

The primary threats to validity of this research are internal and involve the benchmarks selected and the tools used. We carefully selected TDD programs, `Tyburn, Jbehave` and `Helium`, from a list of TDD applications identified by Mark Levison [25]. Research into these applications reveal that they were developed using TDD practices based on documentation and code commits. The other applications were also considered based on their documentation and code commits, and there is no indication that TDD practices were used. Some programs, such as those included in Apache Commons, are well accepted and tested and thus could impact our results based on their high level of acceptance and support. Other programs, such as those selected from the SF100, are less well-maintained or supported at this time, giving better evidence of "normal" programs. In future work, a wider range of programs will be considered, including more TDD and non-TDD applications.

The tools used for coverage and mutation analysis also lead to a potential internal threat to validity. Jacoco was used for all coverage analysis, and MAJOR was used for mutation analysis. Jacoco is a well-founded tool for analyzing Java programs, based on Emma. This is a standard tool for analyzing Java programs [28]. MAJOR was developed by René Just at the University of Washington [21]. This mutation tool has been built into the Java compiler. However, other options such as PIT [10] could be used for comparison.

VII. RELATED WORK

TDD is a subject for many researchers. TDD techniques have been addressed from many angles. Most of them revolve around three topics: internal quality, external quality and productivity. Only recently have researchers started to conduct experiments on TDD with the use of mutation analysis. In this section, we will discuss some of these studies and how they pertain to our work. There are three major studies comparing TDD to non-TDD practices and their impacts of code coverage and mutation score. Madeyski [26] investigated how TDD can impact branch coverage and mutation score indicators. In this experiment, twenty-two third and fourth-year graduate MSc software engineering students were divided in two groups: test-first (TF) and the test- last(TL). The participants were asked to develop a web based conference paper submission system. The main result of this study is "that the TF programming practice, used instead of the classic TL technique, does not significantly affect branch coverage and mutation score indicator" [26]. Following this work, Pancur and Ciglaric [32] conducted a family of controlled experiments comparing test-driven development to iterative test-last development with focus on productivity, code properties (external quality and complexity) and (code coverage and fault-finding capabilities) of tests. In their study, the minimum TDD mutation score was 22.0 and the maximum was 88.8. The results of the research state that "the effect of TDD on mutation is small and in the positive direction (based either only on ID2, r = 0.149)" [32].

Finally, Cauevic and Punnekkat [9] also measured code coverage and mutation score indicators. However, those two metrics were utilized for an analysis to external attribute, a defect detecting ability attribute. For each participant's test suite they calculated a total number of defects discovered in all other participants source code. Fourteen participants were randomly grouped in two groups; the first group used test-first development and the second was used as a control (test-last) group. Participants of the test-first group were guided to use TDD to develop software solutions. The task of both groups were to completely implement and test the same project, a bowling game score calculation algorithm. This research and that performed by Madeyski [26] show similar results, concluding that there are no statistically significant differences in branch coverage and mutation score indicators between the test-first and the test-last groups.

In each of these papers, only small programs were used, and these were developed within a controlled, academic setting. No industrial-sized programs were considered. Our work expands upon these studies by analyzing industry-level,

open source applications. To the best of our knowledge, no studies have been conducted comparing test-first and test-last techniques with regard to final code quality based coverage and mutation score on larger, already produced and in-use applications. Very few studies, other than the ones mentioned above, have considered the correlation between code coverage and mutation scores for estimating code quality. Andrews et al [4] analyzed four common control and data flow testing criteria on middle sized industrial programs with a comprehensive pool of test cases and known faults. However, mutation testing and mutation score are not considered. Fraser and Zeller [15] present an automated approach to generate unit tests that detect mutations for object-oriented classes. While their tool, μtest, does produce tests with strong fault-finding ability, there is no link back to how mutants relate to less expensive, code coverage-based analyses.

VIII. Conclusion and Future Work

In this paper, we performed a test suite quality based evaluation for the TDD testing process. We demonstrated the effect of TDD practices on test suite quality and compared that to other testing methodologies. Test suite quality is measured based on branch and statement coverage compared to mutation scores. Finally, we discuss the correlation between code coverage quality and mutation scores.

We have quantitatively evaluated Test-Driven-Development (TDD) by applying test quality metrics to three open source projects. Our study gives an evidence that the Test-driven development is indeed an effective approach for writing high quality tests to support the maintenance and evolution of software.

We analyzed differences in test suite quality between TDD groups and other in-use development techniques. The difference between the TDD and non-TDD groups is approaching significance on branch coverage, ($t = 2.179, p = 0.059$). The mutation score metric of the TDD application group is significantly different from the non-TDD group, as shown by the results of the Mann-Whitney U test ($p = .012$).

Finally, we evaluated the correlation between code coverage quality and mutation scores. We discovered that mutation scores are significantly correlated to both types of coverage quality. The relationship between mutation scores and branch coverage was significant ($r = 0.79, p < 0.05$), as was the relationship between mutation scores and statement coverage ($r = 0.73, p < 0.05$). We determined that the mutation score often can be predicted from both traditional metrics, statement and branch coverage. Both metrics,when examined together, indicate a higher likelihood to kill mutants. 64% of the programs' mutation scores can be anticipated based on branch coverage. A lower percentage of 54% of the programs' mutation scores can be predicted by the statement coverage.

Some of the programs in our experiment cannot be predicted either by branch or statement coverage. Therefore, there are factors other than the branch coverage and statement coverage that impact mutation score. Those factors are not included in our study and are left for future research to explain program cases such as `Jbehave` and `common-lang` programs. In future work, we will analyze such programs and their associated tests further and incorporate a larger range of TDD and non-TDD developed applications.

Acknowledgment

The authors would like to thank Dr. René Just, the developer of MAJOR, which we used as our mutation analysis tool, for giving us access to the tool and for his help and guidance in its usage, and Dr. Gregory M. Kapfhammer for his guidance and help in the tools and research for this project.

References

[1] Sf100 benchmarks. http://www.evosuite.org/sf100/.

[2] Iec 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems. IEC, 1998.

[3] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments?[software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411. IEEE, 2005.

[4] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, Aug 2006.

[5] Dave Astels. *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference, 2003.

[6] Dave Astels. Google techtalk: Beyond test driven develoment: Behavior driven development, 2006.

[7] Kent Beck and Cynthia Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[8] Timothy Alan Budd. Mutation analysis of program test data. 1980.

[9] Adnan Cauevic, Sasikumar Punnekkat, and Daniel Sundmark. Quality of testing in test driven development. In *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*, pages 266–271. IEEE, 2012.

[10] Henry Coles. Pit. http://pitest.org/, 2014.

[11] Steve Cornett. Code coverage analysis, 1996-2011.

[12] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[13] Michael Feathers. *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[14] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[15] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on*, 38(2):278–292, March 2012.

[16] Dick Hamlet and Joe Maybee. *The Engineering of Software: A Technical Guide for the Individual*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[17] Richard G. Hamlet. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on*, (4):279–290, 1977.

[18] Marc R Hoffmann and Gilles Iachelini. Code coverage analysis for eclipse. *Eclipse Summit Europe*, 2007.

[19] R. Just, G.M. Kapfhammer, and F. Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 720–725, April 2012.

[20] R. Just, G.M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 11–20, Nov 2012.

[21] René Just, Gregory M Kapfhammer, and Franz Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 50–56. ACM, 2011.

[22] Rene Just, Franz Schweiggert, and Gregory M Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 612–615. IEEE Computer Society, 2011.

[23] Gregory M. Kapfhammer. *The Computer Science and Engineering Handbook*, chapter Chapter 105: Software Testing. CRC Press, Boca Raton, FL, second edition, 2004.

[24] Clemens Lee. Javancss-a source measurement suite for java, 2005. *URL: http://javancss.codehaus. org*.

[25] Mark Levinson. Misconceptions with test driven development. http://agilepainrelief.com/notesfromatooluser/2008/11/misconceptions-with-test-driven-development.html, Nov 2008.

[26] Lech Madeyski. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology*, 52(2):169–184, 2010.

[27] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[28] Co. KG Mountainminds GmbH and Contributors. Jacoco java code coverage library. lhttp://www.eclemma.org/, 2014.

[29] Nachiappan Nagappan, E Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3):289–302, 2008.

[30] Dan North. Introducing bdd. *Better Software, March*, 2006.

[31] A Jefferson Offutt. A practical system for mutation testing: help for the common programmer. In *Test Conference, 1994. Proceedings., International*, pages 824–830. IEEE, 1994.

[32] Matjaž Pančur and Mojca Ciglarič. Impact of test-driven development on productivity, code and tests: A controlled experiment. *Information and Software Technology*, 53(6):557–573, 2011.

[33] SC-167. Do-178b: Software considerations in airborne systems and equipment certification. RTCA, 1992.

[34] SC-180. Do-254: Design assurance guidance for airborne electronic hardware. RTCA, 2004.

[35] Elaine J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, 1988.

[36] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.