An analysis of Mutation testing and Code coverage during progress of projects

Oskar Alfsson
Abstract

In order to deliver high quality software projects, a developing team probably needs a well-developed test suite. There are several methods that aim to evaluate test suites in some way, such as Code coverage and Mutation testing. Code coverage describes the degree of source code that a program executes when running a test suite. Mutation testing measures the test suite effectiveness.

More development teams use code coverage to a greater extent than mutation testing. With code coverage being monitored throughout a project, could the development team risk drop of the test suite effectiveness as the codebase getting bigger with each version?

In this thesis, a mutation testing tool called PIT is used during progress of four well known open source projects. The reason for this is to show that mutation testing is an important technique to ensure continuously high test suite effectiveness, and does not only rely on code coverage measurements. In general, all projects perform well in both code coverage and test suite effectiveness, with the exception of one project in which the test suite effectiveness drops drastically. This drop shows that all projects are at risk of low test suite effectiveness, by not using mutation testing techniques.
Acknowledgements

I would like to express my gratitude to my external supervisors, Björn Nyberg and Mattias Sällström at Omegapoint, for the idea and for their comments that greatly improved this thesis.

I would also like to thank my supervisor Pedher Johansson for his patience and guidance throughout this project.

Finally I wish to thank my loved ones for their endless support.
## Contents

1 Introduction 1  
1.1 Problem statement 1  
1.2 Related work 1  
2 Code Coverage 2  
3 Mutation Testing 3  
3.1 Mutation score 4  
3.2 Equivalent Mutation Problem 4  
4 Method 5  
4.1 Tools criteria 5  
4.2 Tools 6  
4.3 Criteria for the analysis of projects 6  
4.4 Selection process of projects 6  
4.5 Projects 6  
5 Results 7  
6 Discussion 10  
7 Future work 11  
References 13
1 Introduction

Testing is an important part of developing and maintaining a software project. It ensures the developers that the software behaves as they intended it to do. It does, however, require a well-developed test suite in order to yield a high degree of confidence in the correctness of existing functionality. There are several methods which aim to evaluate test suites in some way. This paper will look at two test effort evaluation methods; code coverage and mutation testing.

Code coverage is a measure used to describe the degree of source code that a program executes when running a test suite. A program with high code coverage have more of its source code executed in a test suite than a program with low code coverage. This intuitively suggest that a program with higher code coverage should contain less undetected bugs than a program with lower code coverage. Code coverage is fairly cheap in terms of time consumption, especially compared with mutation testing. Mutation testing measure the test suite effectiveness. Mutation testing analysis is expensive in terms of time consumption. In fact some projects will take hours to analyze.

Many programming teams use code coverage as a measure of test effort. By examining some of the more well known open source projects, it is also evident that usage of mutation testing techniques is much less common. This also shows that not as many programming teams uses mutation testing. Hopefully, all teams have the common desire to make high quality projects and thereby having high code coverage and effective test suites. Since code coverage is commonly being used as the sole quality measurement of a test suite, continued development of the existing codebase might run the risk of introducing new untested behavior. While this in many cases would have a lesser impact on code coverage, it might adversely affect the ability of the test suite to guarantee correctness of the evolved code. If this is indeed the case, this could indicate that mutation testing techniques might be an important tool for ensuring a continuously high test suite effectiveness, instead of relying only on code coverage measurements.

1.1 Problem statement

This thesis focuses on two type of test effort analysis, code coverage and mutation testing on a small number of longterm open source Java projects. The analysis methods will be used on at least four major or minor versions of the projects. The hypothesis is that the result of code coverage will be nearly constant for all versions, but the result of mutation testing will decrease with later versions compared to earlier versions of the projects.

By selecting a set of Java projects with a wide version history, and then run both code coverage and mutation testing analyses on the projects, it is possible to discuss and reason if the hypothesis is valid.

1.2 Related work

A lot of work have been done on the subject of code coverage and test suite effectiveness. An example is a study written by W. E. Wong, J. R. Horgan, S. London and A. P. Mathur [20] where they compare the correlation between fault detection effectiveness and block coverage, and fault detection effectiveness and the size of a test set. They describe block
coverage as a sequence of consecutive statements or expressions that include no branches except at the end, which means that if one element of the code is executed, all are. They show that there is a correlation between block coverage and fault detection effectiveness, that is higher than a correlation between fault detection effectiveness and the size of a test set.

In another study A. S. Namin and J. H. Andrews [16] looks at the relationship between three properties of test suites: coverage, size and fault-finding effectiveness. Their experiments indicates that coverage in some cases have correlation with the effectiveness when the size is controlled. They also observed that using both size and coverage leads to a better prediction of the effectiveness then by only using the size. They suggest that both coverage and size are important to test suite effectiveness. The experiment indicated that no linear relationship exists between the properties coverage, size, effectiveness.

In a study made by L. Inozemtseva and R. Holmes [12] they evaluated the relationship between coverage, the size of a test suite and the test suite effectiveness for large Java projects. They say that their study is the biggest that has been done on the subject. The results indicates that there is a low to moderate correlation between coverage and test suite effectiveness when the number of tests in the test suite are controlled for. Also, they show that a higher coverage does not provide greater insight into the effectiveness of the test suite. They suggest that code coverage should not be used as an indicator of the test suite effectiveness.

2 Code Coverage

The first publication about code coverage was made by Miller and Maloney in 1963 [15]. From that day until today it has been well used by developers as a testing measurement. Schoolbooks often recommend code coverage as a technique to measure the quantity of code tested from a test suites [19]. A goal with code coverage could be that every statement from the source code has at least one test associated to it. Maintaining a high degree of code coverage ensures that tests run against all or most of the code as an attempt of reducing the number of bugs present in the project. Measurements can be represented in several ways such as; branch coverage, statement coverage, function coverage and conditional coverage.

Branch coverage provides information about what code paths of a control structure has been executed. An example of a control structure is an if statement. Statement coverage, also known as Line coverage, measures how many of the program statements that are being executed. Function coverage is the result of how many of the program functions that are called. Conditional coverage, also known as Predicate coverage, this measure how many of the boolean sub-expressions, both true and false, in the program that been executed.
As an example, consider the following C function as part of a bigger program.

```c
int foo (int x, int y) {
    int z = 0;
    if ((x>0) && (y>0)) {
        z = x;
    }
    return z;
}
```

After running a test suite, the following four conditions must be met in order to satisfy each type of coverage. For functional coverage, the function 'foo' must be called at least once during the test run. Statement coverage is satisfied by calling 'foo(1,1)', were every line in the function will be called. In order to get branch coverage satisfied, two tests can be called with 'foo(1,1)' and 'foo(1,0)'. In the first test, the two 'if' conditions are met, which executes the branch. The second test prevents the 'if' branch to be executed. A way to satisfy conditional coverage could be to have tests that call 'foo(1,0)' and 'foo(0,1)'. In the first test, condition 'x>0' is true and condition 'y>0' is false and vice versa for the second test. This does not satisfy branch coverage since neither test meet the 'if' condition.

3 Mutation Testing

The first publication of mutation testing was done by R. A. DeMillo, R. J. Lipton and F. G. Sayward [10] in 1978. Since then, its been mostly used by scientist for a research purpose. But with today’s constant increase in computer power, tools start to be useful to common developers.

Mutation testing measures the effectiveness of a test suite, known as its mutation score. It is done by modifying some part of the code, resulting in a new version of the program. This version is called a Mutant and is supposed to change the behavior of the original program. A new version of the program, or mutant, is obtained by introducing a single defect into the code; a mutation. A mutation is generated by applying some mutation operator, where each type of operator represents a certain kind of simple behavioral modification to the original code. The number and type of mutation operators varies between tools. An example of a mutation operator is changing a relational operator with its negation, see the example on the next page.
The relational operator, ‘==’ in the original version, is changed to ‘!=’ in the mutant version. The behavior of the versions is changed, since variable ‘z’ would not get the value of 2 in the mutant version.

The original version of the program and its test suite is supposed to have a test suite where every test are passing. The mutant analysis is done by running the test suite against each mutant, one at a time. If at least one test fail in a test suite run, we say that the mutant is killed. Running a full mutant analysis on a program is considered to be expensive because each mutant must run at least part of the test suite individually. Some mutation testing tools support changing the mutant operators on executable code. By doing this, the time for compilation on a mutation analysis is reduced significantly.

Some mutation testing tools use line coverage to determine which tests it should run. As an example, consider a program with low line coverage. Those mutant operators that are modified on lines that are not covered by a test suite, will not be executed in an analysis. Instead, the user will see in the report that there are some mutants alive and not covered by the test suite.

3.1 Mutation score

As mentioned above, the result of a mutation analysis is called mutation score. It is calculated by dividing the number of killed mutants with the total number of mutants. The mutation score is supposed to help the programming teams develop more effective test suits, as the mutations tries to mimic typical programming errors, such as using wrong variable name or operator.

3.2 Equivalent Mutation Problem

Some mutants can not be detected by a test suite and thereby not be killed. The reason for this is that the mutant version is behaviorally equivalent to the original version. These are called equivalent mutants. This is considered as one of the biggest problem with mutation testing. In a literature study, L. Madeyski, W. Orzeszyna, R. Torkar and M. Józala writes about 17 techniques from 22 articles that are related to the Equivalent Mutation Problem (EMP) [14].
EMP can be explained by the following two statements:

```java
//original version
int i = 3;
if ( i >= 2 ) {
    return "foo";
}

//mutant version
int i = 3;
if ( i > 2 ) {
    return "foo";
}
```

In the first statement (original version) the code is unchanged. In the second statement (mutant version) the mutant operator has been changed from `>=` to `>`. The result are two statements that behaves equivalent. The mutant can not be killed by a test suite because it would not be detected.

To detect these equivalent mutant, the developers have to manually iterate through the mutants that are alive and then decide whether the mutants are equivalent or not. This however, could consume a lot of time even for small programs.

### 4 Method

Tools are selected that could run code coverage and mutation analysis, with reference to certain criteria, see section 4.1 Tools criteria. With the selected tools, see section 4.2 Tools, project can be analyzed. In section 4.3 Criteria for the analysis of projects, criteria of such analysis are listed. A set of Java projects, all with a wide version history, are selected through a selection process, see section 4.4 Selection process of projects. If a project pass the selection process, data will be collected from code coverage and mutation analysis. It will be collected from a set of a selected versions from the passing project. There will also be other forms of collected data such as: number of test in the test suit and source line of code (SLOC). With the resulting data, it is possible to discuss and reason if the hypothesis is valid.

#### 4.1 Tools criteria

There could be a lot of projects tested in the selection process. Because of this a project should be easy to set up with a tool in a development environment.

Considering a mutation analysis could take hours on big projects, a tool with features that reduces the time of an analysis is considered a great advantage when choosing a mutation testing tool.
4.2 Tools

There are several mutation testing tools for Java programs, e.g. PIT [9], Javalanche [17] and MuJava [13]. Many tools are written to meet the needs of academic research rather than real development teams. As a mutation testing tool for this thesis, PIT is chosen for several reason. PIT is open source, fast and easy to setup to a project as it integrates with build automation utilities such as: Ant [1], Gradle [5] and Maven [3]. For this type of experiment, dozens of projects could be analyzed and with an easy setup for each project, a lot of time could be saved. Other mutation testing tools that were in consideration, did not integrate with any build automation utilities. Another reason why PIT were chosen is because it creates each mutant by manipulating the compiled byte code of the original version. This feature makes it faster than tools that creates each mutant by manipulating the source code, compile and then run each mutant. Javalanche have this feature as well, but it does not integrate with any build automation utilities which makes PIT a better choice. PIT also uses code coverage to determine which tests it should run, which means that only the tests that are able to affect a given mutant are rerun for that mutant. This is helpful for this experiment as code coverage would been collected anyway.

In order to be able to determine the number of lines of source code, a tool called SLOC [8] is chosen. The developers describes it as a simple tool to count SLOC (source line of code). It is fast as it takes seconds to count projects with 80000 lines of source code. It is open source and easy to use by a terminal.

4.3 Criteria for the analysis of projects

The EMP, mentioned in previous section 3.2 Equivalent Mutation Problem, is ignored due to time constraints.

Some projects are already using mutation testing tools. This could affect the mutation score and will be taken into account during the discussion section.

4.4 Selection process of projects

In a study [18], A. Shi, A. Gyori, M. Gligoric, A. Zaytsev and D. Marinov set up a method on how to select projects to analyze through a mutation testing tool. In this thesis, the selection process is inspired by that method.

A project must satisfy following five conditions: the project is written in Java and built through Maven [3]; the GitHub repository has more than 100 commits; the already chosen tools can run the latest version of the project; and the tools can run on at least four versions of the project; at last the size increases with each later version. Size as in source line of code and number of test in the test suite.

4.5 Projects

In the study [18] mentioned in section 4.4 Selection process of projects, the authors setup a method on how to select projects to analyze through PIT. Initially, they selected the 2000 most popular GitHub projects which was written in Java. They ended up with 17 projects that satisfied all steps of the method. That 17 projects are the initial set of projects to be tested through the selection process in this thesis. The latest version of each project have
probably been changed since 2014 when the study was made.

Mutation analysis was done with PIT on all projects, each with at least four release versions. The following four open source projects was the result when using the selection process described in section 4.4 Selection process of projects: Commons-Lang [2], Dropwizard [4], JOPT-Simple [6] and JSQLParser [7]. Commons-Lang is a package of Java utility classes for the classes that are in java.lang’s hierarchy. Dropwizard is a Java library for building RESTful web services [11]. JOPT-Simple is a Java library for parsing command line options. The last project, JSQLParser is a SQL statement parser. It parses and translates a SQL statement into a hierarchy of Java classes.

5 Results

Table 1 shows the result of the analysis made on the four projects. The name of each project is shown in Column 1. Column 2 show the data point number, were 1 is an earlier release than 2 and so on. In this thesis, a data point 1 of a project is the earliest version and a data point 4 is the latest version to be analyzed. Column number 3 shows the version name of the specific version or data point. The next column (4) shows the date of a release. Column 5 shows source line of code for each data point. Column 6 is the total number of tests in a projects test suite. Column 7 shows the total number of mutants that were created during the mutation analysis. Column 8 and 9 is the result of line coverage and mutation score.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons-Lang</td>
<td>1. 3.3.1</td>
<td>140318</td>
<td>67 127</td>
<td>2 513</td>
<td>11 274</td>
<td>94.2</td>
<td>85.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. 3.4</td>
<td>150406</td>
<td>69 661</td>
<td>3 534</td>
<td>11 684</td>
<td>94.0</td>
<td>85.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. 3.5</td>
<td>161020</td>
<td>76 197</td>
<td>3 806</td>
<td>12 881</td>
<td>93.7</td>
<td>85.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. 3.6 RC1</td>
<td>170417</td>
<td>78 573</td>
<td>3 985</td>
<td>13 214</td>
<td>94.7</td>
<td>86.0</td>
<td></td>
</tr>
<tr>
<td>Dropwizard</td>
<td>1. 0.8.4</td>
<td>150826</td>
<td>25 693</td>
<td>51</td>
<td>220</td>
<td>88.2</td>
<td>84.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. 0.9.2</td>
<td>160120</td>
<td>30 316</td>
<td>52</td>
<td>227</td>
<td>86.9</td>
<td>83.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. 1.0.3</td>
<td>161028</td>
<td>37 624</td>
<td>66</td>
<td>235</td>
<td>89.2</td>
<td>87.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. 1.1.0</td>
<td>170321</td>
<td>42 188</td>
<td>69</td>
<td>244</td>
<td>85.6</td>
<td>57.0</td>
<td></td>
</tr>
<tr>
<td>JOPT-Simple</td>
<td>1. 3.3</td>
<td>110522</td>
<td>7 259</td>
<td>560</td>
<td>492</td>
<td>98.6</td>
<td>95.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. 4.0</td>
<td>111016</td>
<td>7 502</td>
<td>581</td>
<td>495</td>
<td>99.0</td>
<td>96.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. 5.0.3</td>
<td>160925</td>
<td>10 198</td>
<td>807</td>
<td>658</td>
<td>99.0</td>
<td>95.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. 6.0 A</td>
<td>161206</td>
<td>10 331</td>
<td>808</td>
<td>663</td>
<td>98.7</td>
<td>94.7</td>
<td></td>
</tr>
<tr>
<td>JSQLParser</td>
<td>1. 0.8.4</td>
<td>130827</td>
<td>6 696</td>
<td>176</td>
<td>4 622</td>
<td>79.2</td>
<td>64.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. 0.9</td>
<td>140508</td>
<td>8 398</td>
<td>243</td>
<td>5 392</td>
<td>83.0</td>
<td>66.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. 0.9.5</td>
<td>160314</td>
<td>11 900</td>
<td>414</td>
<td>7 448</td>
<td>82.8</td>
<td>68.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. 1.0</td>
<td>170325</td>
<td>13 721</td>
<td>531</td>
<td>8 759</td>
<td>80.8</td>
<td>66.1</td>
<td></td>
</tr>
</tbody>
</table>

The time between releases for each project varies. The Commons-Lang project had thirteen months between data point 1 and 2. Then there was one year and five month to point 3 and another six months to the last point. Dropwizard had five, ten and five months between all four data points. JOPT-Simple project had the biggest time differences between its releases. At first there was five months to data point 2. The largest gap in time was between point 2 and 3 with five years and nine months. Then there was three months between data point 3
and 4. JSQLParse project had nine, twenty two and twelve months between its releases.

Size in terms of SLOC and number of tests, did increase with all versions for all projects. Also notable is that the number of mutants also increased. Figure 1 shows SLOC collected for all projects at the four data points.

Line coverage for all projects was never more than 2.3 % from its average value. Figure 2 shows the result of line coverage for all project. JOPT-simple and Commons-Lang had a steady line coverage at 99 % and 94 %, respectively. The other two projects, Dropwizard and JSQLParse, did vary more in results of line coverage. Dropwizard had an average line coverage of 87.5 %. The biggest change was between data point 3 and 4 were it dropped 3.6 % from 89.2 % down to 85.6 %. JSQLParse had an average line coverage of 81.5 %. This project varied the most from its average value. At data point 1, line coverage was at 79.2 % which is 2.3 % from average 81.5 %.

The mutation score was nearly constant for three of the projects: Commons-Lang, JOPT-Simple and JSQLParse. Figure 3 shows the mutation score for all projects. Dropwizard had the biggest change on mutation score. It dropped from 89.2 % down to 57.0 % between data point 3 and 4.

Figure 1: SLOC collected for all projects at four data points.
Figure 2: Line coverage collected for all projects at four data points.

Figure 3: Mutation score collected for all projects at four data points.


6 Discussion

It is clear that Dropwizard did change the most at mutation score. Although it did not exactly follow the hypothesis by dropping mutation score for all version, but it did a big drop with 32.2% at its last version. This confirms that it is possible for coverage measurements and mutation score to diverge quite drastically. Other projects that are unaware or ignorant of mutation testing techniques might risk the same drop in mutation score.

When examines mutation reports of Dropwizard there is a lot more mutants marked as timeouts in the first three reports, than in the last report. This could potentially be a reason for Dropwizards big change in mutation score. There are two things that could cause a mutant to be marked as timeout: (1) a mutant causes an infinite loop; (2) PIT thinks that a mutant causes an infinite loop but being wrong. If PIT is correct at detecting infinite loops, then the mutation scores are correct. But if PIT is wrong about detecting infinite loops, then the timeout marked mutants in the first three reports could be alive mutants, i.e. bringing the mutation score down.

It is obvious that JOPT-Simple did outperform all the other projects when it comes to both mutation score and line coverage. In fact, JOPT-Simple have PIT as a plugin dependency in Maven. PITs result have probably been monitored through out the development process. Notably, some classes and methods are ignored at a mutation analysis, which could give it a better score. Also it is possible that all mutants have been iterated and checked if they are equivalent or not.

Only four projects was monitored in the section 5 Results. The reason for this is that other project did not make it through the selection process of projects. Some projects, simply did not have a test suite where all test are passing, making a mutation analysis impossible. Other projects had various issues with PIT. It is possible that those project require some tweaks, e.g. ignore classes or methods, in order to work with PIT. But then some projects would have been customized, making the result harder to analysis. The four resulting projects in this thesis have been selected with the same selection process, 4.4 Selection process of projects, and then been analyzed with referenced to the same criteria, 4.3 Criteria for the analysis of projects.

In the selection process of projects, the last condition was that the size should increase with each later version. The reason for this is that it is more likely that that project follows the hypothesis of this thesis, since intuitively, a bigger project should require more effort to keep the test suite effective than a smaller project. Although, the first version of each project, starts with thousands of lines of source code, making every project quite mature project already from the start. This could be the reason why the test suite effectiveness is as steady as it is. The developers have developed a technique on how they should make tests. Then the same technique is used throughout the project.

In conclusion, all projects are well known open source projects that preforms well and steady, with the exception of Dropwizard, in both line coverage and mutation score. The result of Dropwizard shows that mutation testing techniques might be a solution to ensure continuously high test suite effectiveness, instead of relying only on code coverage measurements. A developing team should consider using mutation testing as a technique to deliver high quality projects.
7 Future work

With only four projects, each with four version, the result is rather limited. If more projects would fit the selection process, then there is a great chance that more projects would have a more unsteady results. By extending the initial set of projects to be evaluated in the selection process, or by customizing some of the existing 17 initial projects, the number of projects and versions in the experiment could have been higher.

Dropwizards big change in mutation score could have been caused by mutants incorrectly marked as timeouts. It would be of great interest to investigate those mutants to see if the mutation score are correct, since this thesis conclusion rely on the mutation score of Dropwizard.

All projects starts with thousands of source line of code, making each project rather mature. As an example, Commons-Lang starts at version 3.3.1 and JSQLParser at version 0.8.4 in this experiment. Would version 1.0 for Commons-Lang or version 0.1 for JSQLParser turn out differently, than the steady result of line coverage and mutation score this experiment resulted in?

In order to get the true mutation score of each project, EMP, mentioned in section 3.2, have to be taking into account. Iterate through all undetected mutants and check if they are equivalent mutants. If they are equivalent then exclude them. The number of equivalent mutants may vary between the projects and versions. By including EMP, there is a good chance of better mutation score for all projects.
References


