Software Testing and Reverse Engineering
Mutation Testing
Final Report

April 10, 2016

Raies Saboerali - 4080335 - r.a.a.saboerali@student.tudelft.nl
Samuel Austin - 4005996 - s.e.austin@student.tudelft.nl
Brynjolfur Mar Georgsson - 4404548 - b.m.georgsson@student.tudelft.nl

Contents

1 Introduction

Traditionally, line coverage and branch coverage are used to determine the quality of a test suite. The idea behind this is that a test suite should run all possible lines of code and all possible branches to make sure that it behaves as intended and there are no unexpected inputs which could cause a problem. The more that is covered the more certain one can be that the program is behaving as it should be. However, faking branch and line coverage can easily be achieved in order to get nice results.

Mutation testing is a testing method which assesses the quality of a test suite by analyzing the behavior of the test suite on modified versions of the source code which the test suite was written for. The desired behavior is for the test suite to contain at least one failing test if a modification was made to the source code. In mutation testing, many modifications, called mutants, are created and for each mutant the test suite is run to see if it fails. The more tests of the test suite fail, the higher the mutations coverage and therefor the higher the quality of the test suite.

A disadvantage of mutation testing is the high performance cost due to the necessity of running the test suite for each individual mutant. It is important to know whether performing a mutation test is worth that high cost. If no extra information or value is gained from running mutation tests over the branch and line coverage information gained from the unit tests than there is no point in running mutation tests.

In this report the correlation between branch coverage and mutation coverage is analyzed to see if there is a consistent connection between the two or not. This should help give insight into what aspects of a test suite make it good and why.

In the second section, the research questions will be defined, which will provide a goal on what look for. The third section will discuss the tools and techniques used in order to perform the investigation, and a list of projects used for this project. The fourth section will present the results, while in the fifth section the analysis of the results is performed.

2 Research Questions

In order to gain more understanding about mutation testing, several research questions have been defined.

Branch coverage vs Mutation coverage

Testing is important when developing software, as it helps us understand the code better and find faults. Getting high line and even branch coverage can be easily achieved by creating tests that do not test for faults, but instead just cover the lines. As an example, lets assume this piece of code:

```
if (a <= 5) do something..
```

The idea here is to test on boundary conditions of the integer a with a being either $4, 5, 6$. If a programmer makes a mistake in the condition then this can be found using the tests. This is a very simple example, but this can be applied to more complex situations where it is more tempting to write tests to cover code instead of finding faults.

The assumption is that, if the test cases are strong then the mutated program should most likely be killed if such a part of the program is mutated. However, if it is not the case, then the mutant survives. This will of course result in a lower mutation score. What the team actually wants to look for, is if having a high branch coverage also means that one would have a high mutation score.

The question regarding this point: *Does a high branch coverage guarantee a high mutation coverage?*

High branch coverage and Low mutation coverage

As a follow up on the first research question: *Is it possible to have high branch coverage, but a low mutation coverage? If there is low mutation coverage, what is the reason behind this?*

The goal here is too look for cases where the mutation coverage is lower than the branch, and finding out why this happens.

Low branch coverage and High mutation coverage

A low branch coverage would indicate a low amount of unit tests, but would it be possible to still achieve a high mutation score? The goal here is to find test suited that do not cover much, but are still strong enough to produce a high mutation score.

The third question: *Is it possible to have a high mutation coverage, but a low branch coverage?*

Branch coverage and mutation coverage per mutation operator

Previous research has shown that certain sets of mutation operators are enough in order to achieve high mutation coverage in most cases. [1] In this experiment, the team wants to go a bit further and look at the correlation between certain operators and the branch coverage. For example, does a certain operator correlate with high or low branch coverage. Mutation operators which produce a low mutation coverage compared to the branch coverage would indicate that that operator is more effective at identifying weak tests than mutation operators which produce a high mutation coverage with the same branch coverage.

This defines the fourth research question: *Does the mutation coverage of certain mutation operators correlate with branch coverage better than others?*

Assertions and Mutation coverage

Finally, it would be valuable to know whether certain types of assertions result in better mutation coverage than others. If certain types of assertions result in better mutation coverage than others this could help identify which type of assertions result in better tests and should be used over others whenever possible. A few examples of types of assertions are:

- Asserting the returned value of a method
- Asserting the value of a private member of an object
- Asserting the number of times a method is called
- Asserting whether or not an exception is thrown

The fifth question becomes: *Do certain types of assertions result in better mutation coverage?*

3 Method

In this section, first the tools used during this project are discussed. After the tools, the techniques which were used to gather the data, and analyze the code and data are discussed.

3.1 Tools and Techniques

In order to perform mutation testing, Pitest [1] was used. Pitest can be used to mutate Java code and can be configured to run with ANT, Maven or Gradle. Pitest runs a default set of mutation operators on the compiled code. After the run, a report is generated containing the line coverage and mutation coverage per package and class. The default mutators which were used are: `VoidMethodCallMutator`, `ReturnValsMutator`, `NegateConditionalsMutator`, `MathsMutator`, `IncrementsMutator` and `ConditionalBoundaryMutator`.

For the research questions, branch coverage was preferred instead of line coverage, in order to measure this Jacoco was used. Jacoco runs the unit tests and outputs the results. The projects were configured to output HTML which made it easy to read, and also in csv format in order to process them easily. EclEmma for eclipse was also used as a second plugin in order to generate the branch coverage reports.

A python CSV parser [2] was written in order to generate useful graphs, which will be shown later in the report. The parser creates graphs per mutation operator, and plots the mutation coverage against the branch coverage. These graphs were used in order to help answer the research questions.

3.2 Projects

The following projects were used for analysis:

- JPacman Framework [3]
- Cobertura Plugin [4]

[1] http://pitest.org/
[2] https://github.com/TUDelft-CS4110/2016-team-mutation/tree/master/ReportParser
[3] https://github.com/TUDelft-CS4110/2016-team-mutation/tree/master/mutation_projects/jpacman-framework-5.1.0
[4] https://github.com/jenkinsci/cobertura-plugin

- Mockito [5]
- RxJava [6]
- Cloudstack [7]

The JPacman framework is used in the Bachelor course of Software Testing which two of the group members followed. The other projects listed were all gathered from various repositories on GitHub.

3.3 Problems

Getting Pitest to run for the first time took a while (a day) for the whole team. At first nothing would work, every project produced a different error. Some would not even run, others would say that the test suite was not green, even though the tests passed when running them with Maven or JUnit in Eclipse. A single failure of a test would cause the whole Pitest suite to fail. If a module was not mutable, e.g. did not contain valid Java tests which Pitest could understand, then a failure would occur stating that there are no tests. After a while of struggle, the team found out that failing modules could be skipped. Luckily most of the modules that caused failures were not part of the core modules and could be skipped easily. In some instances, the core module would fail. Excluding the core of course made no sense, instead the test that failed had to be fixed or removed. Finding the failing test was not always easy. A few test cases where failing when trying to read files from the project, which were somehow not found when run via Pitest. For such test cases, the tests where removed and the project had to be rebuild. In order to find out which test fails, the debug log had to be printed, which was quite large and slowed down the Pitest run, especially for large projects as RxJava and Cloudstack. After the failing test was identified and dealt with, the project had to be build again, because Pitest makes use of the compiled code. Re-running Pitest after the build, even with history enabled, took quite some time. This cycle of fixing a failing test, rebuilding and running Pitest had to be repeated for each test failure. The Pitest tool doesn't support the Gradle building framework out of the box. The website for Pitest does recommend the following tool for Gradle projects [8] which is a Gradle plugin developed and maintained by a third party. The Gradle-Pitest plugin offers a decent getting started documentation although for already established projects, like RxJava, some intuition was required to get mutation reports.
The Pitest tool also has support for the ANT building framework, this feature was tried on one project. The Apache Tomcat [9] project uses ANT to build its source code. Due to the size and complexity of the Tomcat structure the plugin did not give any results. The Tomcat project is split into several modules, however only the core module and the relevant tests were of interest for mutation. Any effort to exclude modules did not result in any success. In the end it would skip all the modules resulting in a empty mutation report. The problem was that several test classes were failing, removing the problematic test classes allowed the plugin to run further. However the process ran for several hours and either caused the computer to lock up or ran in loops without reaching any conclusion. Therefor the idea of actually analyzing the Tomcat testing suite was scrapped. Jacoco was mainly used for generating the branch coverage reports. However, for some projects the plugin would run, but not output the results. A lot of time was tried fixing this issue, unfortunately without success. Finally EclEmma was used in Eclipse in order to generate the reports for the projects that failed to work with Jacoco. Strangely, EclEmma did work and the team was able to output coverage reports for all projects.

Projects with low or no mutation coverage:

- crawler4j [10]
- libgdx [11]
- Jenkins [12]
- Maven [13]

[5] https://github.com/mockito/mockito
[6] https://github.com/ReactiveX/RxJava
[7] https://github.com/apache/cloudstack
[8] http://gradle-pitest-plugin.solidsoft.info/
[9] https://github.com/apache/tomcat
[10] https://github.com/yasserg/crawler4j
[11] https://github.com/libgdx/libgdx
[12] https://github.com/jenkinsci/jenkins
[13] https://github.com/apache/maven

- Tomcat [14]
- Hive [15]
- Guava [16]

Ultimately, the number of projects that were useful enough was lower than expected. After spending hours trying to get Pitest to work, one would find that the projects did not produce any mutation coverage or really low mutation coverage. For example, in crawler4j, which is a really small project, the tests were not able to run with Pitest, even though they were JUnit tests. This probably has to do with the environment Pitest runs in, which may not be suitable for some test cases. This resulted in zero mutation coverage, which is not at all useful for this project. In case of of libgdx, Jenkins and Maven a similar situation occurred. At first these projects looked quite promising, e.g. for Jenkins, there were more than 3000 tests present. Unfortunately, after running the the Pitest suite, the results were not that exciting. Both projects resulted in a really low mutation coverage (less than 15). The team did not find it useful in order to include this many projects which yielded zero or low mutation coverage, which is the reason why these are excluded. The less useful results of these runs are still visible in the repository. Looking at the reports, one might observe that Cloudstack is included even though the project has low mutation coverage. However, this project was still included as it contained a few sub modules which still had a higher mutation coverage than 15. The projects Tomcat, Hive and Guava were not included because these did not yield any positive result. These projects failed when trying to run Pitest, because of either JUnit failure or failure of the core modules.

4 Results

This section contains the results obtained from the mutation and branch coverage testing that was performed on the various projects. In mutation testing, mutants can be categorized into three groups; survived, killed and no coverage. In this report the no coverage mutants were ignored so the mutation coverage was calculated based on only the killed and survived mutants.

4.1 Branch coverage vs Mutation coverage

The initial focus of this research was to determine whether there is a correlation between the branch coverage of a test suite and its mutation coverage. As a first step the overall branch and mutation coverage for the projects were investigated and the coverage results per class for each project was plotted. From the results in figure 1 and 2 some correlation between branch coverage and mutation coverage can be observed, although there are also outliers which do not follow this pattern.

4.1.1 High branch coverage and low mutation coverage

A number of classes were found for which there was a high branch coverage and a low mutation coverage. A number of classes and their coverage statistics are listed in table 1.

Project	Class	Branch Coverage	Mutation Coverage
Mockito	AtLeast	83%	20%
RxJava	Functions	100%	63%
RxJava	CompositeException	100%	58%
Cloudstack	LoggingExclusionStrategy	100%	0 %
JPacMan	Game	73%	42%

Table 1: Classes with high branch coverage and low mutation coverage

4.1.2 Low branch coverage and high mutation coverage

As a sub-question defined previously in section 2, this report also investigates the cases where there is low branch coverage and a high mutation coverage. There are a few instances where this occurred. E.g. in Cloudstack,

[14] https://github.com/apache/tomcat
[15] https://github.com/apache/hive
[16] https://github.com/google/guava

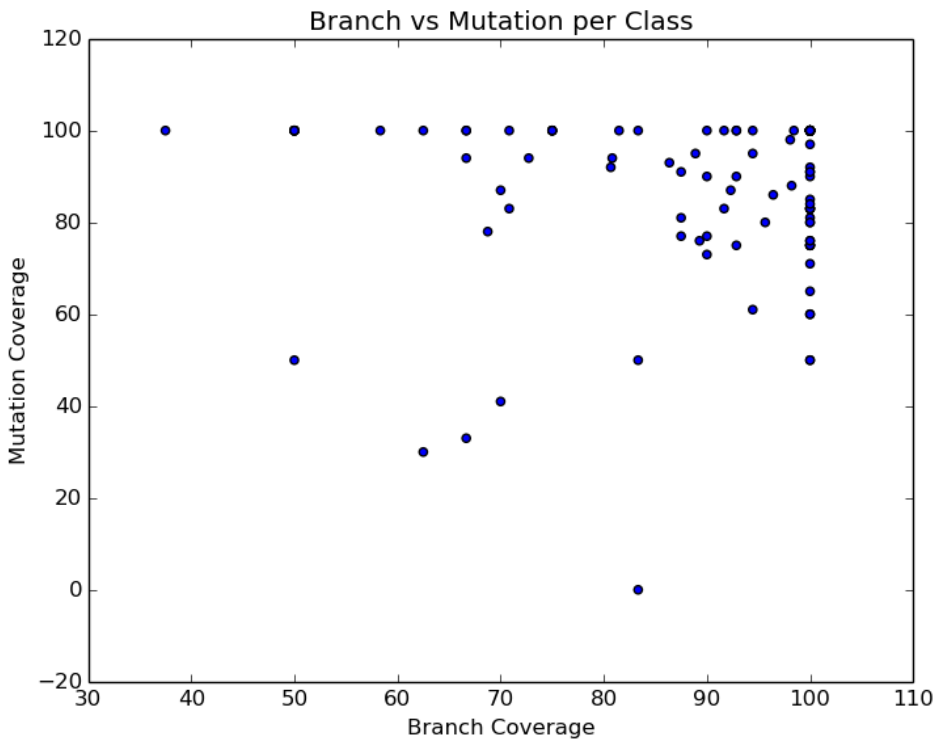

Figure 1: Branch coverage vs mutation coverage per class for the Mockito project.

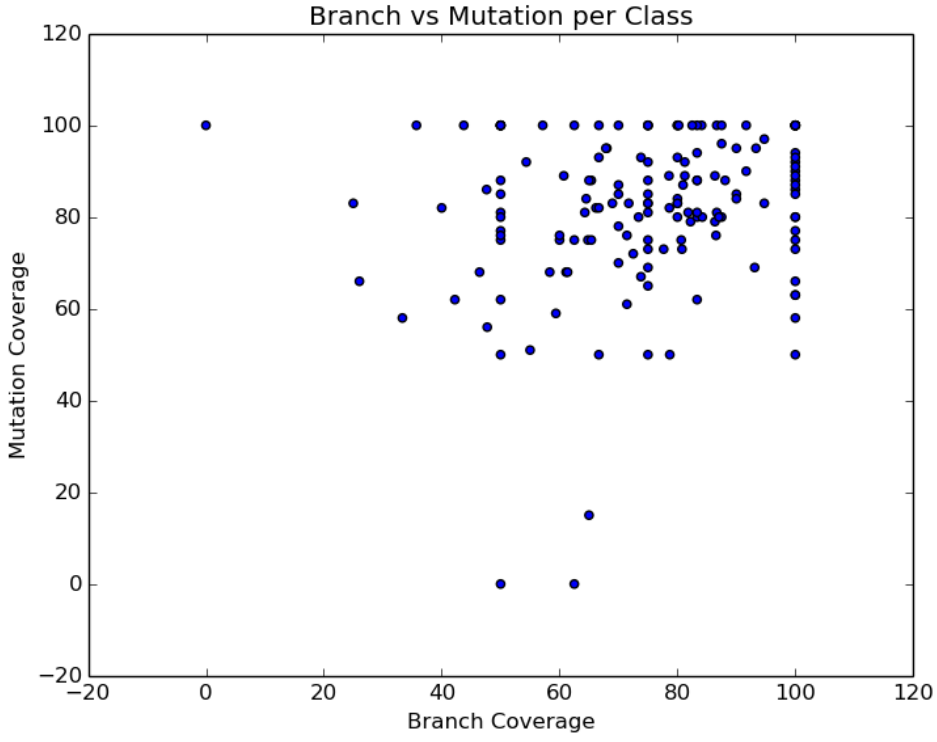

Figure 2: Branch coverage vs mutation coverage per class for the RxJava project.

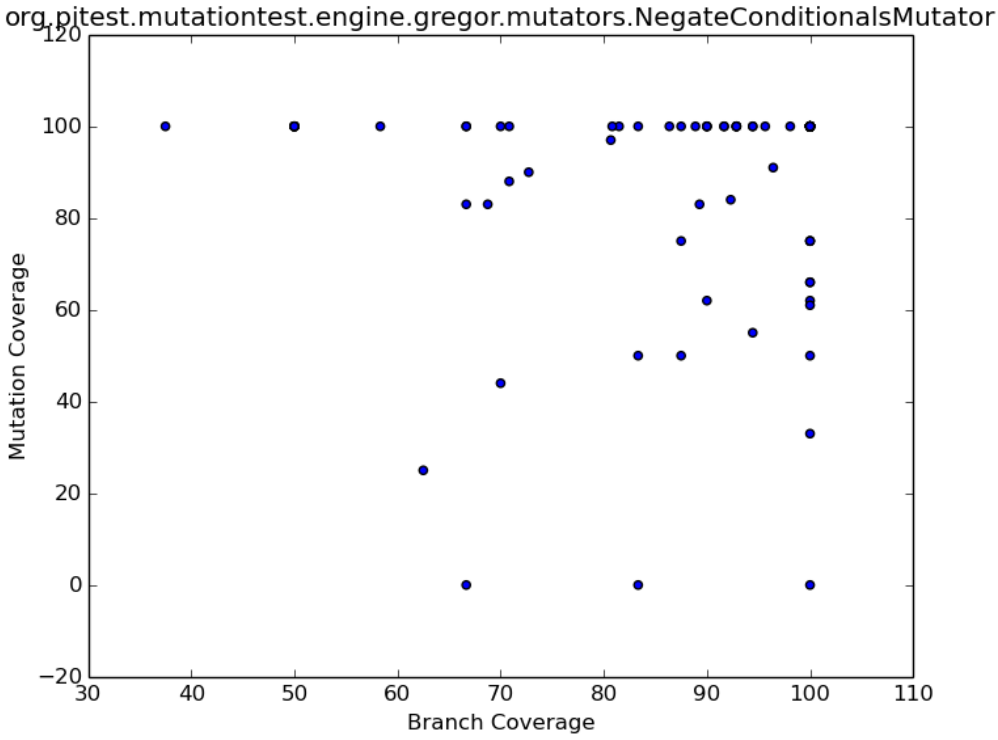

Figure 3: Branch coverage vs mutation coverage per class of the negate conditionals mutator for the Mockito project.

`com.cloud.agent.transport`, there are two instances were such a case occur. In JPacman, this occurred once in `org.jpacman.framework.factory.MapParser`. Similar instances can also be found in the other projects of which the detailed results are available in the repository [17].

4.2 Branch coverage and mutation coverage per mutation operator

When looking at individual mutation operators in figure 3 and 4 one can see similar results to the overall results. The general tendency seems to be that a high branch coverage will result in a high mutation coverage regardless of the operator but again there are some outliers which do not follow this pattern.

Table 2 shows the average branch coverage and average mutation coverage for each mutation operator used. Detailed results for each project can be found in appendix A. These results and the graphs generated for each project will be used to get to a conclusion. The graphs for each mutation operator per project is a lot, which why they are not included here. Instead they can be found in the repository [18]

Project	Avg. Branch Coverage	Avg. Mutation Coverage
VoidMethodCallMutator	58.03	69.69
IncrementsMutator	80.92	74.96
ReturnValsMutator	84.70	61.75
ConditionalsBoundaryMutator	59.12	74.172
NegateConditionalsMutator	72.27	71.90
MathMutator	67.99	62.972

Table 2: Branch coverage and mutation coverage per mutation operator

[17] https://github.com/TUDelft-CS4110/2016-team-mutation
[18] https://github.com/TUDelft-CS4110/2016-team-mutation/tree/master/mutation_reports

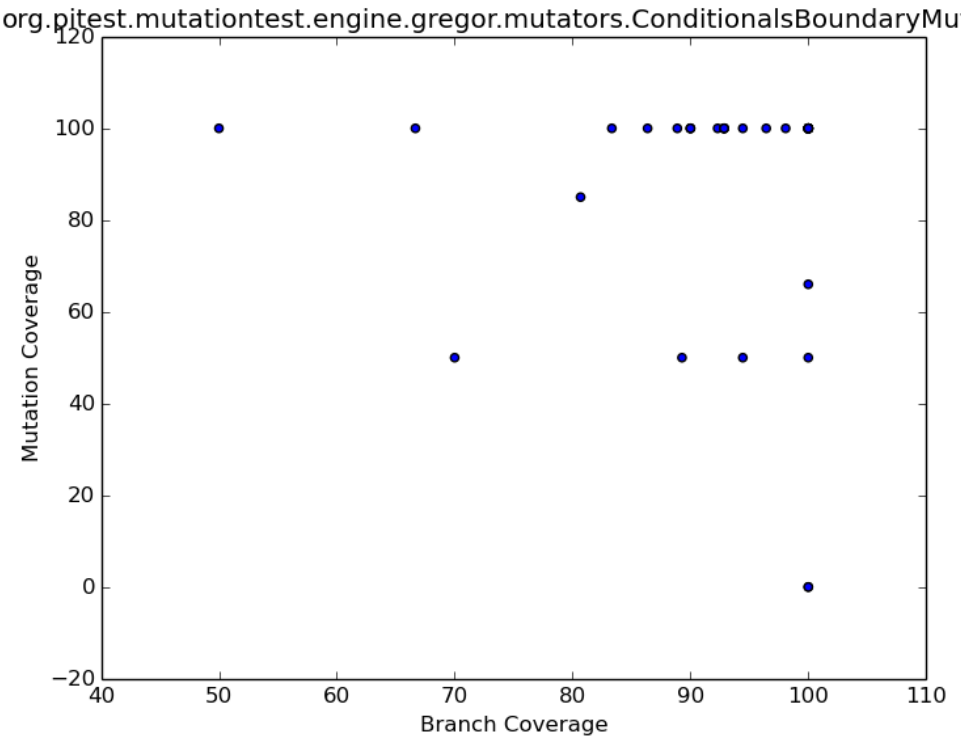

Figure 4: Branch coverage vs mutation coverage per class of the conditionals boundary mutator for the Mockito project.

4.3 Assertions and mutation coverage

To analyze how mutation coverage relates to the type of assertion, the classes from table 1 with high branch coverage and low mutation coverage will be reused. Two of these classes could be paired with a similar class with a high mutation coverage.

The first class is `AtLeast` which has a high branch coverage (83%) but a low mutation coverage (20%). A similar class is `Only` which has a high branch coverage (100%) and a relatively high mutation coverage (56%).

The second class is `Functions` which has a branch coverage of 100% and a mutation coverage of 63%. The class that is similar is `Actions` which has a 100% branch coverage and a 100% mutation coverage.

5 Analysis

In this section the research questions defined in the second section will be answered based on the results and collected data.

5.1 Relation between branch coverage and mutation coverage

From the results we can see that generally branch and mutation coverage do correlate, but there are certain cases where this does not hold. An explanation for the correlation between branch coverage and mutation coverage could be the following. In cases where there is a high branch coverage, when you apply a mutation the chance of the location of the mutation being covered by one of the test is high, which in turn means that there is a high chance of a test failing due to this. In the case of low branch coverage, there is a high chance that a mutation will be applied in a branch which is not covered by a test and therefor the mutant will survive.

5.2 High branch coverage and low mutation coverage

There were some cases where there was a high branch coverage, but a low mutation coverage listed in table 1. These cases seem to contradict the more common cases discussed in the previous section. Discovering what causes this discrepancy could help in gaining an understanding of when tests can be considered good or bad. In the case of the `AtLeast` class in Mockito there were very few tests for testing that class and the assertions in the tests were focused on verifying whether exceptions were being thrown or not. A large majority of the branch coverage came from tests which were made for testing other classes, but happened to also run functionality from the `AtLeast` class.

The same can be seen for the `Functions` class in RxJava. Although there were more test methods than for `AtLeast` and the branch coverage was not being created indirectly, all tests were expecting exceptions to be thrown.

For `CompositeException` there was also good branch coverage provided by the test class however the tests were all aimed at asserting the results of two of the methods. These two methods also have a good mutation coverage, however, the `printStackTrace` method was also called in the tests but its results were never checked. This caused the mutation coverage to be very low for that method which brought the overall mutation coverage down for the class.

In the case of `LoggingExclusionStrategy` and `Game` there were simply no test classes. The branch coverage was being provided by tests which indirectly covered the functionality of the two classes but were asserting the results of other classes. This reason for high branch coverage and low mutation coverage was also observed in classes not listed in this report and seemed to be fairly common.

5.2.1 Low branch coverage and High mutation coverage

As stated in the previous section, there are a few cases were the mutation coverage is higher than the branch coverage. The goal behind this, is to investigate why this happens. By looking at the program code and test code of these instances, there is one conclusion that can be reached, which is: "the tests are good". Even in instances where the branch coverage was not a 100% or close to it, one can achieve high mutation coverage. By looking at vast majority of these cases, on can observe that the test cases are good enough to survive the mutation, which probably means that they are "good" tests. However, this only puts a value on the quality of the tests, it does not mean that the tests meet the product requirements.

5.3 Branch coverage and mutation coverage per mutation operator

In this section the relationship between branch coverage and individual mutation operators are investigated. The idea is to see if the mutation coverage of certain mutation operators correlate with branch coverage more than others. The results of each mutation operator were compared to each other for each project it was used in.

Higher mutation coverage

In table 2, the average coverage for `VoidMethodCallMutator` can be found. The average result for this mutator is around 58% branch coverage and 70% mutation coverage. On average the mutation coverage for this operator, is 12% higher.

The results for `ConditionalBoundaryMutator` are also positive on average. For most projects the average mutation coverage resulted in a higher number than the average branch coverage, this can be seen in table 2. Here the average branch coverage was around 60%, and average mutation coverage around 74%.

These results probably indicate that these types of vault are most likely better tested than others.

Lower mutation coverage

The other four mutation operators yielded lower mutation coverage than branch coverage on average. The `ReturnValsMutator` performed worse, with the average branch coverage being around 85% and average mutation coverage around 62%, which is a difference of 23%. The `IncrementsMutator` and `MathMutator` each resulted a 5% lower in mutation coverage than branch coverage.

The `ReturnValsMutator` had the lowest mutation coverage when compared to the average branch coverage, which makes interesting to look at. This mutator is used to test return values. For example: inverting Boolean return values. More details can be found on the Pitest website [19]. A few examples of such occurrences can be found in JPacman, the `Food` class, which contains the `getPoints()` method. This is a simple function which returns

[19] http://pitest.org/quickstart/mutators/#RETURN_VALS

the value of a piece of food in the game. In Pitest the return value was mutated to an incorrect value. The test still survived. In the test suite this code is not directly tested, instead higher methods in the hierarchy are tested. Similar examples can be found in the same package in classes like `Game` and `Player`. A few more examples are also found in Cloudstack's `DhcpConfigEntry` and Mockito's `Timer` class. In each of these cases a similar behavior can be observed, the functions were not directly unit tested. In case of JPacman the lines were also covered, because of unit tests that spawned the GUI in order to perform tests using the UI, however the core functionality of such functions are mostly not tested enough.

By looking at the mutation operators which result in lower mutation coverage than branch coverage, on can find "weak" tests faster. The idea behind this reasoning was explained in section 2. According to the results, the `ReturnValsMutator`, `IncrementsMutator` and `MathMutator` can be used to find weak tests faster.

Important issue

One important issue with these results are the assert statements in the source code. As far as the team could find, the usage of assert statements was only found in JPacman. This may be due to the fact that JPacman is the only academic project on the list in this report. Since version 0.27, Pitest avoids mutating asserts. This was introduced when users started complaining about assert mutations which not covered, because by default the assertions are not enabled and therefore Pitest does not run with asserts. However, in JPacman there are asserts in the form of methods. While these statements are not mutated, the internal of the method functioning as an assert is mutated. This resulted in surviving mutations. As stated, this issue was only found in JPacman and not in the other projects. One example of this can be found in the `Board` class of JPacman where the constructor makes use of `titleInvariant()` as a form of an assert.

5.4 Assertions and Mutation coverage

An analysis was made of the difference between the assertions in the tests for classes with a high and low mutation coverage. The reason for this is that it is possible to achieve a high branch coverage without doing any assertions. This would result in your tests never failing regardless of any changes made to the code. This indicates the cases where a high branch coverage and a low mutation coverage were found, which are caused by low quality assertions in the tests for classes with a low mutation coverage.

An example of this can be found in Mockito. The class `Only` has a high branch coverage (100%) and a relatively high mutation coverage (56%). A similar class, `AtLeast`, has a high branch coverage (83%) but a low mutation coverage (20%). When looking at the unit tests of the `Only` class we see that there are tests for multiple scenarios of the same method which all assert the resulting value of the functionality. In the case of the `AtLeast` class, there are fewer tests and the tests do not assert for expected values. Instead, they only expect an exception to be thrown or not and in one case there was an assertion inside of a catch block.

In RxJava, the classes `Functions` and `Actions` both have a branch coverage of 100% but a mutation coverage of 63% and 100% respectively. A clear difference could be seen in the test methods of the two test classes. While all test methods for `Functions` were asserting for exceptions being thrown, only one test for `Actions` was doing this. All other tests for `Actions` were asserting the values that were expected to be produced by the called functionality. This observation leads us to believe that asserting expected values in a test is a more powerful way of testing than checking for the presence of a thrown exception.

6 Conclusion

On the whole, it can be concluded that there is some correlation between branch coverage and mutation coverage if the tests are of decent quality. The most important aspect of the tests are the assertions in this case. This does not mean that branch coverage can be used as an alternative for mutation coverage for measuring the quality of a test suite. As can be seen in the report, it is quite easy to create tests which have a high branch coverage yet have a low mutation coverage. Three reasons were identified for this:

- The assertions in the test are of low quality, meaning they do not assert values produced by the functionality being tested.
- Called functionality is not being asserted by the tests
- The branch coverage is being created indirectly by tests that are testing other functionality.

This shows that the quality of a test can not only be measured by its branch coverage, but rather the quality of a test is determined by the combination of its branch coverage and the quality of its assertions. What was identified as a good assertion is an assertion which verifies either a return value of the function being tested or a value in some object that is being modified by the function. A type of assertion which seemed to perform badly are assertions which verify exceptions being thrown.

In this report, no analysis was done involving tests which use assertions on the number of calls to a method. We believe these types of assertions would also result in a lower mutation coverage than assertions which verify returned values, but more research would need to be done to verify this.

Looking at the correlation between branch coverage and mutation coverage per operator, based on the results in this report the mutation operators `ReturnValsMutator`, `IncrementsMutator` and `MathMutator` yield either equal or lower mutation coverage. The mutation operators that resulted in lower mutation coverage can therefore be used in order to find weak tests earlier than the mutators that resulted in higher mutation coverage. If a test suite already has high coverage, the tests are probably less interesting to look at. According to this report, these operators can be used in order find weak tests faster. Looking at the results there seems to be a tendency to badly test return values, what this report did not cover, is the mentality of the coder behind this. In further research it would be interesting to look at reasons why this happens.

Overall the journey of this project was interesting, as it was nice to analyze how branch coverage and mutation coverage correlate with each other and using that data to find if mutation testing is actually suitable. However, this ride had quite some bumps when trying to get the mutation tool to work. The work presented in this report might be limited, because it was conducted for just a few weeks, however it might provide interesting ideas for others to dive into greater depth. As a final remark one can say that there is a future for mutation testing, but there is also a lot of work to be done yet.

References

[1] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

A Branch coverage vs Mutation coverage per mutator

The tables in this appendix present the average branch coverage and the average mutation coverage per mutation operator for each project.

Project	Avg. Branch Coverage	Avg. Mutation Coverage
JPacman	36.90	51.59
Cobertura plugin	55.11	64.82
Mockito	68.41	92.34
RxJava	76.38	75.17
Cloudstack	53.33	64.52
Average	58.03	69.69

Table 3: Branch coverage and mutation coverage for: VoidMethodCallMutator

Project	Avg. Branch Coverage	Avg. Mutation Coverage
JPacman	75	61.98
Cobertura plugin	66.67	75.99
Mockito	93.10	92.72
RxJava	86.53	79.46
Cloudstack	83.33	64.52
Average	80.92	74.96

Table 4: Branch coverage and mutation coverage for: IncrementsMutator

Project	Avg. Branch Coverage	Avg. Mutation Coverage
JPacman	83.45	50.70
Cobertura plugin	90.74	59.99
Mockito	94.01	89.36
RxJava	88.37	75.25
Cloudstack	66.91	33.47
Average	84.70	61.75

Table 5: Branch coverage and mutation coverage for: ReturnValsMutator

Project	Avg. Branch Coverage	Avg. Mutation Coverage
JPacman	62.5	61.98
Cobertura plugin	32	75.60
Mockito	83.81	91.51
RxJava	52.29	77.34
Cloudstack	65	64.52
Average	59.12	74.172

Table 6: Branch coverage and mutation coverage for: ConditionalsBoundaryMutator

Project	Avg. Branch Coverage	Avg. Mutation Coverage
JPacman	61.61	57.04
Cobertura plugin	84.43	67.37
Mockito	90.33	89.61
RxJava	90.76	75.74
Cloudstack	34.21	69.76
Average	72.27	71.90

Table 7: Branch coverage and mutation coverage for: NegateConditionalsMutator

Project	Avg. Branch Coverage	Avg. Mutation Coverage
JPacman	39.29	46.53
Cobertura plugin	100	79.66
Mockito	88.24	86.57
RxJava	74.91	73.06
Cloudstack	37.5	29.04
Average	67.99	62.972

Table 8: Branch coverage and mutation coverage for: MathMutator