Test Reduction Visualization

UNDERGRADUATE HONOURS THESIS

FACULTY OF SCIENCE (COMPUTING SCIENCE)

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Blair Wiser

SUPERVISORS:

Jeremy Bradbury

Christopher Collins

April 2016

Abstract

During the development of software it is important to maintain a comprehensive test suite to ensure quality of the code produced. However, agile development calls for quick builds to enable the iterative development that can quickly adapt to unpredicted circumstances. A great amount of research has been done in order to find methods that optimize the quality of the project given a small amount of time. One such way to optimize quality is through a test suite that has little to no overlap in coverage. Since tests are run many times throughout the development of software it is important for the process to be fast while still ensuring that all of the code is covered by tests. We propose a tool that will aid a user through the process of reducing tests in a test suite. We developed a tool that uses visualization techniques to show a representation of a test suite. The tool uses mutation testing data to display to the user the mutant score of individual tests along with how many of the mutants detected were uniquely killed by the test and how many mutants were killed by other tests in the test suite. The tool uses a greedy algorithm to find subsets of the given solution in order to optimize the mutant score with respect to the run time of the test suite. Usage of this tool can overcome test case glut that can occur during development as more and more tests are added to the project.

Table of Contents

List of Figures

1. Introduction

The purpose of this thesis is to examine how visualization techniques aid in the task of optimizing a test suite by removing redundant tests. We define a redundant test as a test that only detects bugs which are detected by other tests in the test pool. With a large test set it can be difficult to manually examine them to determine overlap between them. Through the use of visualization techniques the user will be able to see how tests overlap at a glance. Evaluation of tests to discover redundancy is done through mutation testing. Our tool displays the results of the mutation testing process and suggests to the user which tests should remain and which tests should be removed.

1.1 Motivation

Test suite optimization is an important aspect of agile development where testing is done many times throughout the work day. However, we must be careful in the optimization process. There is a natural tradeoff between the speed of the testing process and the test coverage. We must balance both aspects. If the testing is not thorough enough, bugs will be left undetected which will cause problems later in development. If the testing is not fast enough the process can cause delays for the project. While there has been work done to automate the process of reducing tests, little work has been done with visualization of the process. The user often has valuable insight into the nature of the test data and that insight can be useful in the test reduction process. Through a human in the loop system we believe the process of test reduction can be streamlined and produce better results.

1.2 Goals

The goal of this thesis is to produce a prototype for a tool that aids developers in optimizing test suites.

The goals of the tool are:

- Time and minimum score parameters to be set by user
- User can add or remove tests from the test pool
- System initializes with a solution based on input parameters

1.3 Stakeholders

The main stakeholders of the tool are developers with knowledge and intuition behind the code and the test set to be optimized. The developer must also have some knowledge of mutation testing practices in order to understand the evaluation of the tests. We aim to make the test reduction process as easy and as painless as possible. Manipulating large data sets can often be tedious so we must be aware of the requirements of the developers using this tool. Secondary stakeholders are other individuals involved in the project that might view the visualization. The visualization aspect of the tool must be intuitive and understandable. With stakeholders identified we are able to evaluate the tool based on their unique requirements.

1.4 Thesis Overview

In section 2 background subject material will be identified and examined. An overview on mutation testing and software visualization will be given along with the importance of optimized testing in agile development. Section 3 covers the design and implementation of the prototype tool, including a look at the work that serves as a basis for this thesis. Section 4 will examine the resulting tool, how it is used, and how it helps development. Finally, section 5 will conclude with a summary of what has been done and suggestions for future work.

2. Background

This section gives a brief overview of the different concepts that this thesis utilizes. Many of these concepts come from software testing such as test suite optimization and mutation testing. Software visualization techniques are also used to present the data.

2.1 Test Suite Optimization

Ten minute builds is a concept from agile development. It dictates that code should be built and tested within a ten minute time period. This is because agile development calls for building and testing to be done several times through a day. Because the testing process is done many times throughout the day it must be an efficient process. In other words the test suite must be optimal. There are three general approaches to optimizing a test suite. Test case reduction (also called minimization) aims to identify and remove redundant tests from the test suite. Test case selection seeks to identify which test cases are related to changes made in the base code and to change the test cases accordingly, thus alleviating the buildup of test cases that tends to occur as development progresses. Finally, test case prioritization reorders the test suite in such a way that most errors are detected at the beginning of the testing process. This thesis focuses primarily on the first method of test case reduction.

2.2 Mutation Testing

Mutation testing is used to evaluate the quality of tests. A program is mutated by making various changes to it. The tests are then run on the original code and on the mutants. If the test gives different output for the original code and the mutated code the test is said to have killed (detected) the mutant. A test's mutation score is the number of mutants it is able to kill. Mutants

are created through various means. A common way to create a mutant is to swap an operator for another. Both arithmetic and logical operators can be swapped.

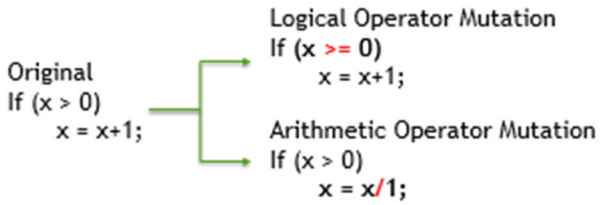

Figure 1: Examples of mutation

2.3 Unit Testing

Unit testing is a method of testing where individual blocks of code are tested. It is viewed as the smallest testable part of the code. Unit testing is often done by using a test harness such as JUnit for Java. Because of this, unit testing is very easy to automate. The benefits of unit testing have been studied extensively [3] showing a decrease in defects early on in a project's life cycle decreasing the cost of fixing the defect later.

2.4 Software Visualization

Software visualization is the use of representing data as visual graphics in order to explore and interpret it. It is often used to create graphical representations of programs, algorithms and processed data. Its main challenge involves finding an effective way to represent these concepts as visual graphics. However, when done correctly a visualization of data can make it much easier to understand. Visualization helps people in finding patterns and relationships within data.

3. Implementation

The primary stakeholders of the tool are software developers so it is important for the tool to integrate into a software development work flow with little frustration. The tool must also be easy to learn and present information in a clear manner. Because of these requirements we selected to implement the tool as an Eclipse plugin.

3.1 Architecture

The plugin consists of three parts. The activator serves as the basis for the Ecplise plugin handling the life cycle as well as the integration of the other parts. The actions model the data and perform operations on the data in response to the user's actions. The view displays the data to the user in a new panel created by the plugin.

A test result class was created to store all the data of how tests detected the mutants. The class consists of an ID, the time it took a test to run, and three array lists representing which mutants were detected. The array lists are

- Detected Mutants: a list of all the mutants detected by the test
- Unique Mutants: a list of mutants that were detected by the test first
- True Unique Mutants: a list of all mutants that are only detected by the test.

3.2 Input

The plugin reads in output from test cases run with JUnit. Each test's output must be in its own file. After the plugin reads a file it creates a test result object to store the mutants that it detects.

Once all the test files have been read the unique mutants and true unique mutants lists are calculated for each test result.

3.3 Test Reduction Algorithm

We developed an algorithm that suggests to the user a test suite based on parameters input by the user. The parameters input are the maximum time the test suite is allowed to take when run and the minimum mutant score the test suite is allowed to have as a whole. The test score for the test suite is defined as the sum of the unique mutants detected by each test plus the sum of the true unique mutants found by each test. These parameters are stored in a configuration file in JSON format which are then read by the tool. The algorithm takes a greedy approach to optimizing the mutant score. First the test suite is pruned down to a size so that it meets the time constraint set by the user. Tests are removed from the end of the list. Next a greedy switch is made. The algorithm first finds the test in the test suite that detects the lowest number of mutants uniquely by checking the size of its true unique mutant list. Next the algorithm finds the test in the unused test group that detects the most number of mutants uniquely. Similarly to before, this is found by checking the true unique mutant list of each test in the unused group. The final step is to swap the found tests. The algorithm continues to make greedy swaps until a swap would result in a lower mutant score.

3.4 Visualization

The visualization window consists of two timelines. The top timeline represents the test suite while the bottom timeline is the group of unused tests. A single test is represented by a rectangle on the timeline. Tests are displayed in the order that they are run in from left to right. The rectangle's width represents the time it takes to run. The height of the rectangle is the number of

mutants that the test detected. The coloured sections of each rectangle represent different categories of mutant. The darkest blue section represents the unique mutants only found by that test. The medium blue sections corresponds to the mutants that have been detected by this test first and are detected by a test later on. Finally, the lightest blue portion is the number of mutants detected that have already been detected by a test in the test suite. The tests in the unused group are coloured as if they were at the end of the test suite timeline.

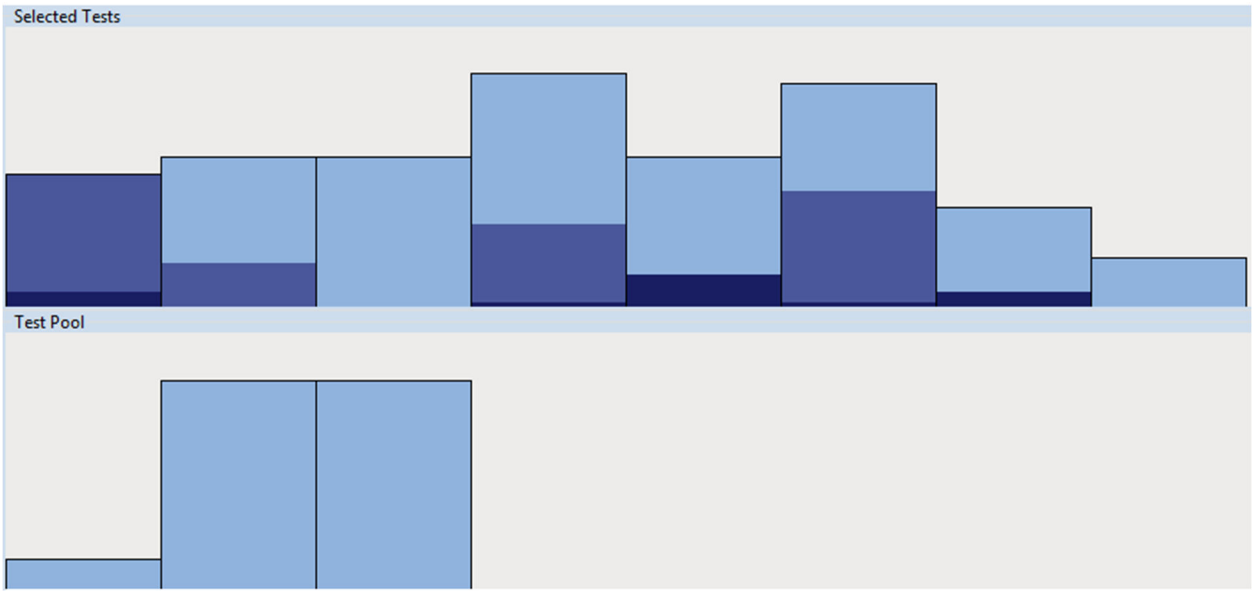

Figure 2: Test timelines

The user can select a specific test by clicking on a test in the timeline. Selecting a test allows the user to visualize overlap between tests. When a test is selected sections of other tests in the timeline are highlighted with a fuchsia and the selected test is outlined. Highlights indicate which other tests are able to detect the mutants found by the selected tests. Using this feature the user can quickly see how similar a test is to other tests in the test suite.

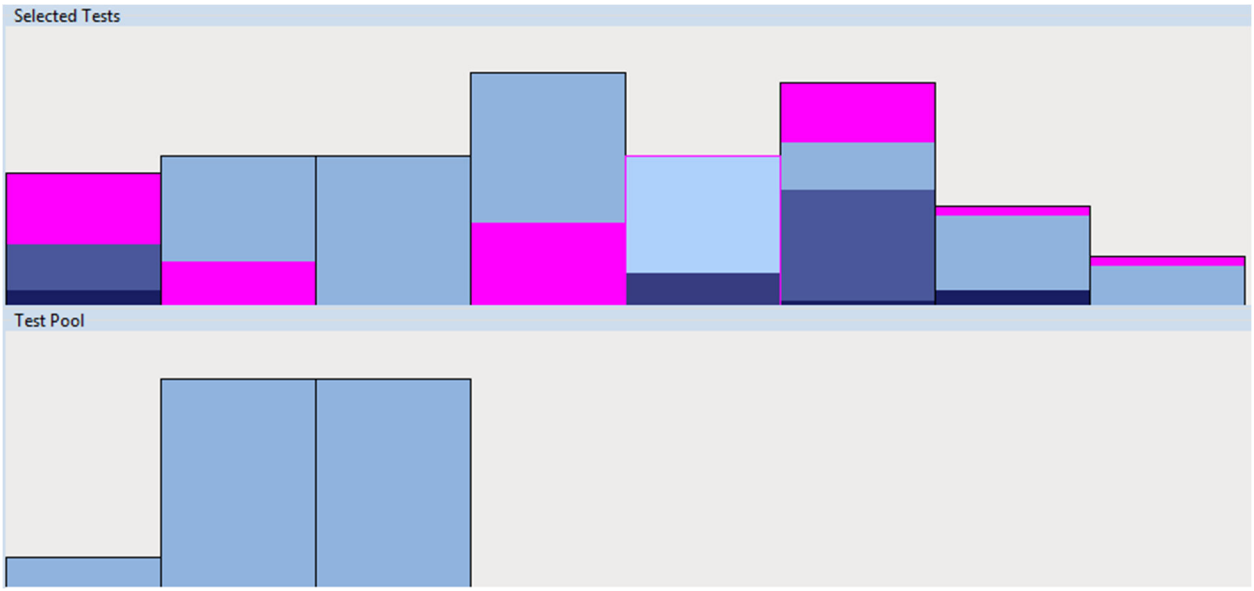

Figure 3: Test selection

More information about a test will be displayed when the user hovers the mouse over the test as a tool tip. The tool tip contains the sizes of the mutants detected, mutants newly detected and the mutants uniquely detected lists. The tool tip also displays the time it takes for a test to run.

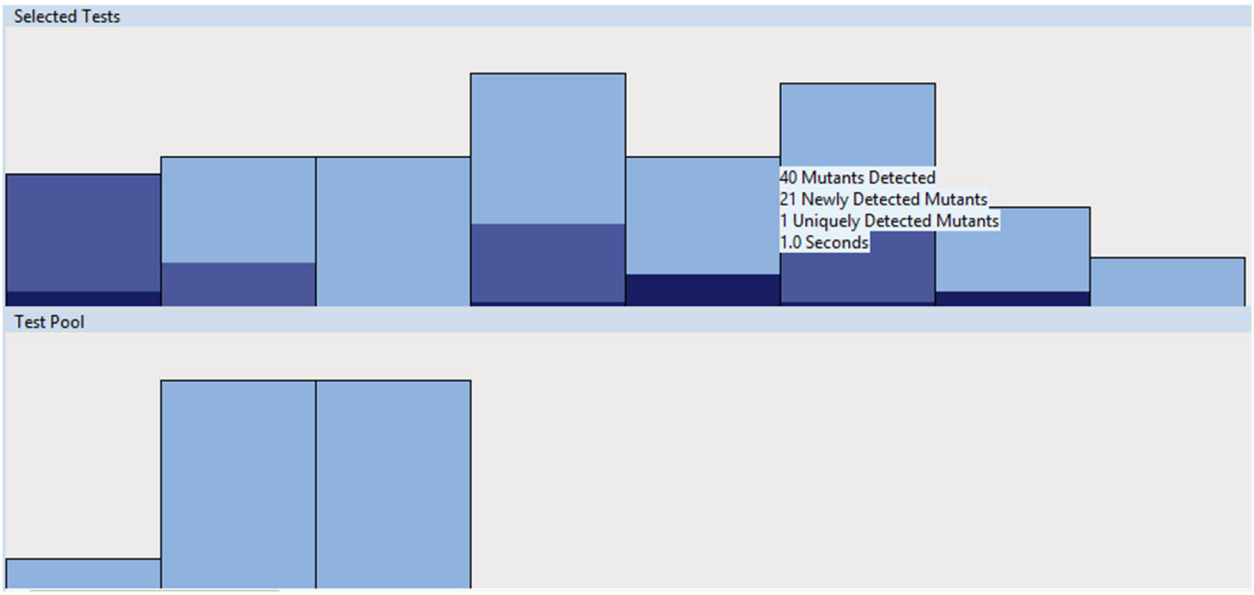

Figure 4: Test tool tip

The user can remove tests from the test suite by right clicking on them. Once a test is removed it will appear in the unused test group.

4. Results

An open source Java project was utilized to test the plugin. First, mutants were created from the code by utilizing a tool called µJava. Using µJava a total of 159 mutants were created using 3 mutation operators.

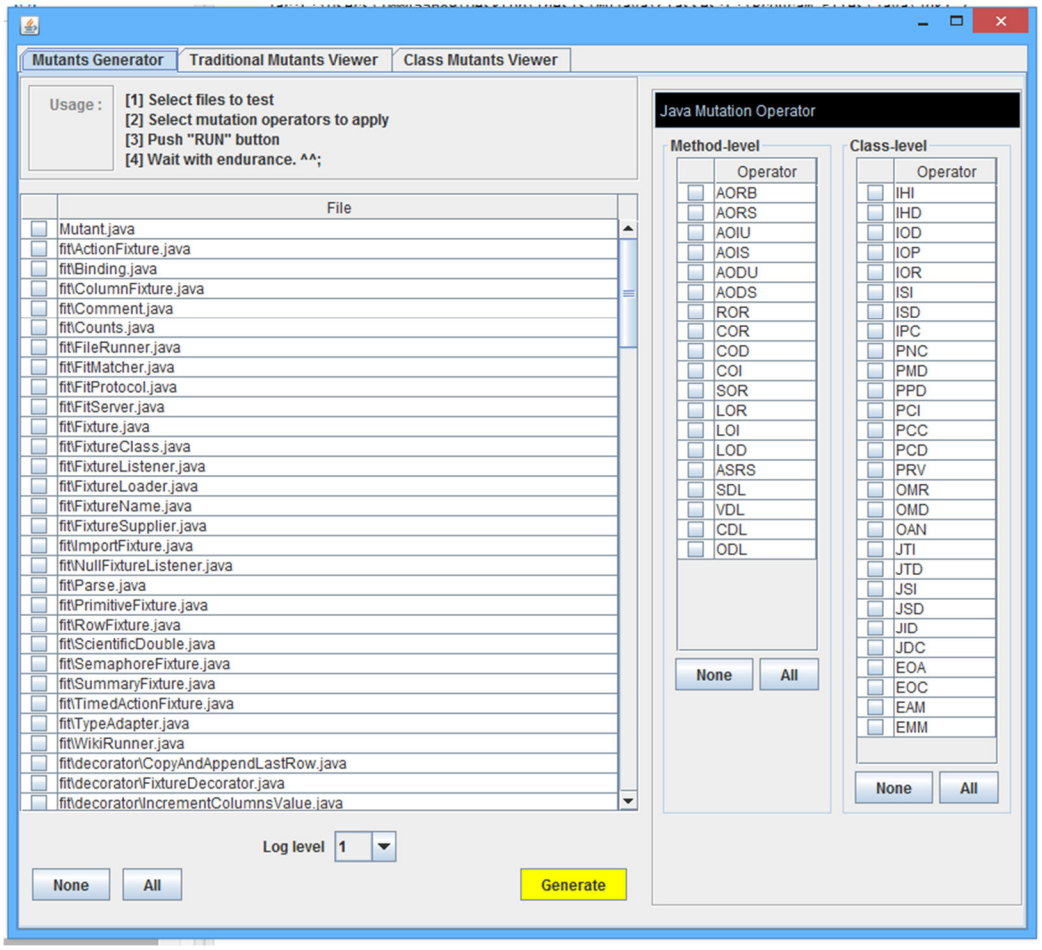

Figure 5: µJava mutant generation tool

Next, tests had to be separated into their own files in order to separate the test results. A batch script was used to run the tests. Each test had to be run on each mutant. A batch script was used to use JUnit to run each test on all the mutants and writing the results to an output file for that test. Once all the output files were prepared they were used as input for the plugin.

During testing we had to make a decision regarding the vertical scale of the bars. The total height of the window is based on the maximum possible height that a rectangle can reach (the total number of mutants). However, for the test data no single test discovers a large portion of the mutants resulting in small bars. An alternative is to scale the maximum height of the window to be the maximum number of mutants that is discovered by the tests plus some percentage of buffer space. This approach results in bars that are overall larger and easier to see.

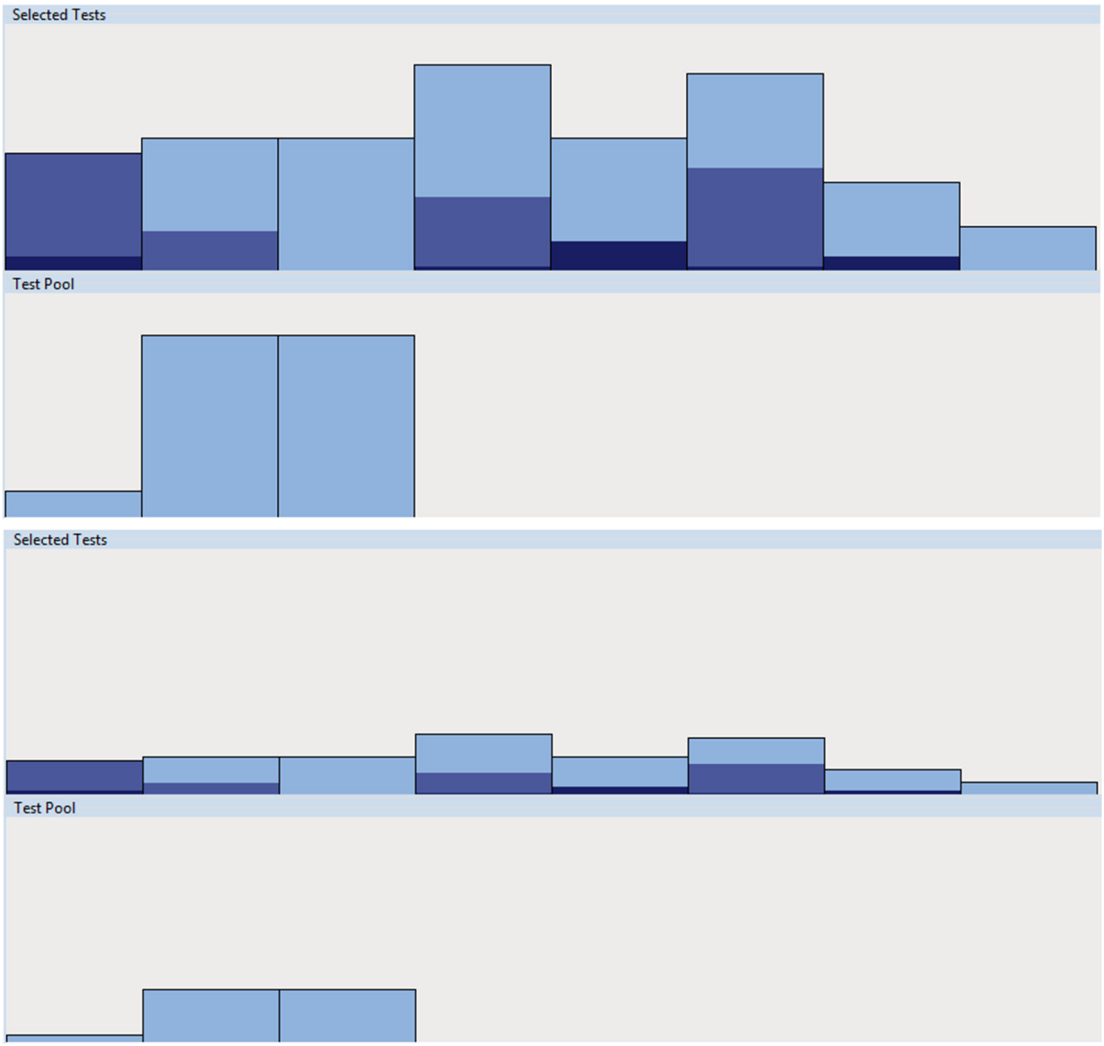

Figure 6: Comparison of vertical scales

The downside to this approach is that it is misleading at first glance. Until the user inspects the results by hovering over the bars it appears that the individual tests are able to detect a large portion of the mutants, which is not the case.

From the initial results we learn that it is possible to gain insight into the task of test reduction by using visualization techniques. The visualization used makes it easy to see the overlap in test coverage by selecting a test and highlighting the other tests that overlap with the current one. The colour scheme allows the user to see how much each test is contributing by showing the unique mutants and the number of mutants first detected by the test. As time goes on tests tend to contribute less as previous tests will have caught most mutants.

The visualization could be further evaluated in a formal manner. In order to test its effectiveness it should be used to in a real project. For comparison a second test suite should be maintained by more traditional means. The tool's usefulness will be measured by comparing the two test suites after a set amount of development cycles. The development team could also be surveyed to get their opinions on the tool and how useful they find it.

5. Conclusion

In summary, we set out to explore how visualization can aid in the task of test reduction. Starting with an existing visualization aimed at test prioritization we extended it to work with real unit testing data. In addition, we developed a greedy algorithm that is able to read real data and create a suggested, minimized solution for the user. This tool is implemented as an open source Eclipse plugin and is the main contribution of this thesis.

5.1 Limitations

The main limitation for this project was the use of a pre-existing project as a base. The previous project was implemented in such a way that it was not easily extended. Most of the classes were implemented as static classes in order for new data to be included. However, this caused numerous issues when extending the code to use real data, such as when user parameters were to be used by the algorithm instead of statically set values. Overall, implementing changes to the existing code in order to correctly implement the functionally we needed took more time than what was initially estimated and resulted in some planned aspects for the project to be abandoned.

5.2 Future Work

A large amount of preprocessing of data is required by the user in the plugin's current form. This includes generating the mutants from the base code and running the tests against these mutants to detect them. Ideally this processing would be done within the tool. Future work to make the plugin a more user friendly experience should focus on automating the preprocessing for the user

Early on we had considered using a genetic algorithm to create the initial solution instead of a greedy one. A genetic algorithm uses a process based on natural selection to iterate through

generations of possible solutions. High ranked solutions pass on traits to the next generation while low ranked solutions are discarded. The implementation of a genetic algorithm is more time consuming than a greedy one so we shifted focus in order to meet deadlines. While the use of a genetic algorithm does not guarantee a better solution, we think it would be an interesting experiment to implement a genetic algorithm for test suite reduction and compare its results with those of the greedy algorithm implemented in our plugin.

References

1. Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel, “Test Case Prioritization: A Family of Empirical Studies”, IEEE Trans. Softw. Eng. 28, 2, February 2002.

2. Yoo, S. and Harman, M., “Regression testing minimization, selection and prioritization: a survey” Software, Testing, Verification and Reliability, March 2010.

3. Williams, Laurie, Gunnar Kudrjavets, and Nachiappan Nagappan. "On the effectiveness of unit test automation at microsoft." *20th international symposium on software reliability engineering*. IEEE, 2009.

4. Eltoweissy, Mohamed, Matković, Krešimir and Gračanin, Denis, “Software Visualization”, Innovations in Systems and Software Engineering, Septermber 2005.