Investigating ways to make Mutation Testing feasible in agile environments

Mark Anthony Cachia

Department of Computer Science and Artificial Intelligence

University of Malta

May 2012

Submitted in partial fulfilment of the requirements for the degree of B.Sc. (Hons.) ICT
Plagiarism Declaration
The plagiarism declaration form can be found in any one of the hard copies.
Abstract

In Agile environments, code is written in cycles and tests are developed to ensure the code is correct and to protect against regressions. However, tests might give a false sense of security. Mutation Testing is a technique which analyses the thoroughness of a test suite and helps identify which lines are not tested exhaustively. Unfortunately, Mutation Testing is very costly both in terms of execution time and the time it takes a developer to analyse mutation results. Many have tried to make Mutation Testing more efficient by optimising the generation and execution of mutants, or reducing the mutants by some form of heuristic. In my work, I successfully introduce and evaluate the concept of Localised Mutation. Localised Mutation exploits the fact that in this modern era of agile software development, code is written in iterations. By only considering the additions or modifications to the source code, the number of mutants generated is drastically reduced. This makes Mutation Testing more feasible and in turn reduces the cost of software development as Mutation Testing can be used to detect bugs in earlier stages of development where bugs cost much less to fix.
Acknowledgements
I would like to thank Dr Mark Micallef for his guidance throughout this work. Also, I would like to thank Apache Commons for the BCEL and CLI libraries used in this work.
# Table of Contents

Plagiarism Declaration .................................................................................................................. 1

Abstract ........................................................................................................................................ 2

Acknowledgements ......................................................................................................................... 3

Tables and lists ................................................................................................................................. 8

  Table of Algorithms ..................................................................................................................... 8
  Table of Equations ......................................................................................................................... 8
  Table of Figures ............................................................................................................................. 8
  Table of Tables .............................................................................................................................. 8

1  Introduction ................................................................................................................................ 10

  1.1  Overview of the Mutation Process ......................................................................................... 10

  1.2  Current problems in the area of Mutation Testing ................................................................. 11

    1.2.1  Objective ........................................................................................................................... 11

    1.2.2  Computationally Expensive ............................................................................................. 11

    1.2.3  Human Oracle Problem ..................................................................................................... 12

  1.3  Expectations ............................................................................................................................. 12

  1.4  Research question ...................................................................................................................... 12

    1.4.1  Contribution ....................................................................................................................... 13

  1.5  Document Structure .................................................................................................................. 13

2  Background and Literature Review ............................................................................................. 14

  2.1  Introduction ............................................................................................................................... 14

  2.2  Agile Development ..................................................................................................................... 14

    2.2.1  Agile Manifesto ................................................................................................................... 15

    2.2.2  Testing in Agile environments ............................................................................................ 16

  2.3  Test Suite Adequacy Criteria .................................................................................................. 18

  2.4  Mutation Testing ....................................................................................................................... 20
2.4.1 Introduction

2.4.2 The Mutation Testing Process

2.4.3 Mutation operators

2.4.4 Mutation Score

2.4.5 Code coverage analysis versus Mutation Testing

2.4.6 Existing cost reduction techniques

3 Specification and Design

3.1 Introduction

3.2 Chosen optimisations from the literature

3.3 Other optimisations – Localised Mutation

3.4 Discarded optimisations from literature

3.5 Discarded techniques from literature

3.5.1 Object Oriented Mutation Operators

3.6 Assumptions

3.7 Main Components of the proposed Mutation Testing tool

3.7.1 Change Detection – For Localised Mutation

3.7.2 Test Relevance – for Localised Testing

3.7.3 Mutator

3.7.4 Test Runner

3.7.5 Front end

3.8 Implementation Procedure

3.8.1 Initial program design

3.8.2 Incremental improvements

4 Implementation

4.1 Technologies and Skills acquired

4.1.1 Understanding of the Java Virtual Machine and bytecode

4.1.2 Byte Code Engineering Library (BCEL)
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.3</td>
<td>JUnit test runner</td>
<td>34</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Other notes</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Challenges Encountered</td>
<td>35</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Mutation</td>
<td>35</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Detection of changes</td>
<td>35</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Detection of tests</td>
<td>36</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Execution of tests</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Background on existing evaluation techniques</td>
<td>38</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Metrics</td>
<td>38</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Evaluation Methodologies</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Chosen evaluation technique</td>
<td>39</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Evaluation Setup</td>
<td>40</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Procedure</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Results</td>
<td>44</td>
</tr>
<tr>
<td>5.4</td>
<td>Discussion</td>
<td>46</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Execution Time</td>
<td>46</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Mutation Score</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparison with other work</td>
<td>48</td>
</tr>
<tr>
<td>5.6</td>
<td>Conclusions</td>
<td>49</td>
</tr>
<tr>
<td>6</td>
<td>Future work</td>
<td>50</td>
</tr>
<tr>
<td>6.1</td>
<td>Mutation</td>
<td>50</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Further optimisations on Localised Mutation</td>
<td>50</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Mutation in General</td>
<td>50</td>
</tr>
<tr>
<td>6.2</td>
<td>Further optimisations in Localised Testing</td>
<td>50</td>
</tr>
<tr>
<td>6.3</td>
<td>Others</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>51</td>
</tr>
</tbody>
</table>
Tables and lists

Table of Algorithms
Algorithm 1 determining if a test is relevant to a method ........................................36
Algorithm 2 illustrating the procedure of the evaluation..............................................44

Table of Equations
Equation 1 illustrating the time limit to run a test [9].................................................36
Equation 2 Mutation Score [2]..................................................................................38
Equation 3 the percentage decrease in Mutation Score ..............................................45
Equation 4 the performance gain.............................................................................45

Table of Figures
Figure 1 Program mutation .......................................................................................10
Figure 2 illustrating the agile process adapted from text [25] .............................15
Figure 3 illustrating the testing pyramid adapted from [19] [28] [36].................17
Figure 4 a screenshot illustrating statement, branch and condition coverage in the
Eclipse IDE ...........................................................................................................19
Figure 5 illustrating the Mutation Testing process, from [1]...............................20
Figure 6 illustrating the percentage of publications on optimisation techniques of
Mutation Testing [1] ...........................................................................................24
Figure 7 illustrating the bytecode structure [30] .....................................................26
Figure 8 the activity diagram of the system .............................................................30
Figure 9 illustrating the system flow ......................................................................32
Figure 10 illustrating the statistics TortoiseSVN tool ..........................................43
Figure 11 illustrating the Execution Time against Iteration ......................................46
Figure 12 illustrating the performance gain in comparison with other work ..........48

Table of Tables
Table 1 illustrating a few types of mutation operators [1] .........................................22
Table 2 implemented mutation operators .................................................................35
Table 3 illustrating the features against Iterations ........................................39
Table 4 illustrating the specifications of the machine on which the evaluation was run ..................................................................................................................41
Table 5 illustrating the experiment data ..........................................................................................................................42
Table 6 showing the changes performed in revisions which signified a period of work ..................................................................................................................................................43
Table 7 illustrating the results for Experiment 1 – 2 hours of work ..................44
Table 8 illustrating the results for Experiment 2 – a day’s work .......................44
Table 9 illustrating the results for Experiment 3 – a week’s work ......................44
Table 10 illustrating the Mutation Score combined results ............................45
Table 11 illustrating the Execution Time combined results .............................45
1 Introduction

In agile programming, substantial effort is invested in writing comprehensive unit tests [9]. These unit tests have multiple purposes. Apart from assessing the code’s compliance to functional requirements, unit tests also make source code easier to understand and use, as tests are examples of the usage of the unit [8]. Furthermore, it can be said that it can serve as detailed specification of the unit. However, the value of testing can be called into question if there is no measure of the quality of unit tests [9].

Mutation Testing is a method designed to identify whether a test suite is satisfactory. In turn, such tests lead to finding bugs within the code [1]. Irvine et al [9] in their work to determine the effectiveness of unit tests [9] found that developers using agile methods are committed to having effective unit tests and are motivated to improve on them. Furthermore, the developers who participated in their study were astonished by how ineffective some of their testing practices were.

1.1 Overview of the Mutation Process

The mutation process was described in detail by Jia and Harman [1], and is summarised below.

- **Mutation** (Figure 1) – A number of mutants are generated (P₁,n) by performing syntactic changes (seeded faults) to the user’s original program P [1],
- **Checking correctness of the test suite** – all test cases are run on the original program P to verify that the tests pass, if tests do not pass, the mutation process for this test suite stops at this point [1],
- **Running tests on generated mutants** – Tests in the test suites are run on the mutant programs P₁,n. If the test fails when run on a mutant, this is known as a
*killed mutant*. This is positive, as it means the test suites were able to detect the syntactic modification. If, on the other hand, the tests all pass, this means that the test suites were not able to detect the syntactic changes made, and thus the test suites can be further improved [1].

From the killed mutant information collected, Mutation Testing determines a “Mutation Score”, also known as an “adequacy score” [1], which represents the quality of the tests.

### 1.2 Current problems in the area of Mutation Testing

#### 1.2.1 Objective

“Although Mutation Testing is able to effectively assess the quality of a test set, it still suffers from a number of problems.”[1] The objective is thus to identify the key bottlenecks in Mutation Testing thus enabling researchers to identify solutions to these problems. These problems can be sub-divided into two categories, the computational cost and cost of manual labour due to the Human Oracle Problem.

#### 1.2.2 Computationally Expensive

Mutation Testing by its very nature is very computationally expensive and this hinders Mutation Testing from being used on everyday basis in practical situations [4] [7]. This is due to the fact that the computational complexity of Mutation Testing is $O(n^2)$ where $n$ is the number of operations in the source code [3]. Modern hardware today has made great advances; however, it is still not enough to make the “traditional”, that is without optimisations, Mutation Testing feasible [7]. The key computational bottlenecks of Mutation Testing are generating mutants, equivalent mutants and running tests, as described below.

**1.2.2.1 Large amount of generated mutants**

During Mutation Testing, a large number of mutants are generated and this is computationally expensive [4] [7].

**1.2.2.2 Equivalent Mutants**

Certain mutants are said to be equivalent mutants. These are mutants which either leave the semantics of a program unaltered (such as mutating dead code), mutations that supress speed improvements, alter state, or which cannot be triggered [1] [6].
Equivalent mutants are expensive to detect, and furthermore, the generation and execution of tests on equivalent mutants are a waste of computation time [6].

1.2.2.3 Running Tests
All tests in a program have to be tested on each mutant [1] [7]. This is very expensive [1] [2] [3] [4] [7] [29]. In fact, Untch and Offutt [2] quote Krauser et al [10] “current implementations of mutation tools are unacceptably slow and are only suitable for testing relatively small programs” [2]. This belief is also shared by Bogacki and Walter [8].

1.2.3 Human Oracle Problem
The Human Oracle Problem is a problem found in many computing concepts [1]. The Human Oracle Problem can be defined as the process of manually calculating the output of the program (which in the case of Mutation Testing is each test case). In each test case, the developer has to come up with the input and then manually calculate the output which will be automatically asserted by the testing framework. The Human Oracle Problem particularly applies to Mutation Analysis, as after Mutation Testing is complete, and the Mutation Score is not adequate, the developer has to generate more tests hence undergoing the Human Oracle Problem until the Mutation Score obtained is adequate [7]. Hence, it can be said that a lot of manual labour is involved in Mutation Analysis.

1.3 Expectations
Mutation Testing promises to help determine the thoroughness of a test suite and hence the users have high expectations, but Mutation Testing is very expensive [1]. However, there a number of solutions for the above mentioned bottlenecks which may be found in literature. It is believed that Mutation Testing will be feasible if the optimisations on Mutation Testing are implemented in one single solution [7].

1.4 Research question
The goal of this work is to answer the question, “Can Mutation Testing be made efficient to be practical for everyday use, particularly for agile environments?” This dissertation presents the design, implementation and evaluation of a holistic solution aimed towards answering this research question.
1.4.1 Contribution
The main contribution of this work is the introduction of Localised Mutation. This concept exploits the iterative nature of software development to drastically reduce the number of mutants generated in agile environments.

1.5 Document Structure
In the Background and Literature Review section, the concepts of Agile environments, testing and bytecode as well as further details on Mutation Testing are introduced. The Specification and Design section discusses the new concept of Localised Mutation, the chosen optimisation techniques from literature and highlights how the tool which tests this new technique is divided. The Implementation section exposes the techniques and technologies required to test the concept of Localised Mutation Testing. The Evaluation section discusses the choice of the metrics and methodologies to assess the improvement by Localised Mutation and other experiments' results. The Future work section lists some further implementation, specific improvements and suggestions. The last section, Conclusions sets to explain the successful introduction of the concept of Localised Mutation as well as the positive implications.
2 Background and Literature Review

2.1 Introduction
In today’s modern world of software development, agile techniques are used to help make the development process more responsive with feedback and less stringent so as to allow developers to be more creative [26]. Developers write test suites to ensure that their code is both valid and correct [27]. Test Driven Development is an agile approach which relies heavily on testing, and involves writing the tests before the code in order to have more concise, elegant and structured code [27]. In order to analyse the effectiveness of their test suites, developers tend to rely on code coverage analysis, a technique discussed in section 2.3. However, having a high statement coverage may lead to a false sense of security, as even though a test suite seems to extensively test a codebase, the test suite might not test all the branches or it might leave some special input cases untested [30].

Mutation Testing is a technique which determines if a test suite thoroughly tests a codebase [1]. However, Mutation Testing is extensively expensive in terms of computation time, which in turn makes it unattractive [4][7].

This chapter serves as a detailed background to the concepts mentioned above as well as a survey of previous work related to making Mutation Testing feasible.

2.2 Agile Development
Agile software development is about eliminating the traditional approach of software development and replacing it with activities which provide control [25], “feedback and change” [26].
There are six main variants of agile software development; these are: Crystal methodologies, Dynamic software development method, Feature-drive development, Lean software development, Scrum, and Extreme programming [25]. All variants tackle software development in iterations, also known as sprints. This is essential in modern software development as visibility of the progress and changes are critical but most importantly easily available. Furthermore, the agile mentality encourages developers to be more creative in their work and thus being more innovative [25].

Agile software development is performed in short iterations of two or three weeks in length. At each iteration, the client tests the features implemented and provides feedback as illustrated by Figure 2.

### 2.2.1 Agile Manifesto

To try to maintain a certain standard for agile environments, the Agile Manifesto was established. This manifesto is an informal gathering of guidelines towards the development of quality software in an agile manner and is based on the following four principles [27].

*“Individuals and interactions over processes and tools”* [33] – The key players such as the developers, managers and other stakeholders’ interactions are more important than the defined processes and tools of traditional software development.
Furthermore, tests will be driven by customer requirements; hence, it is vital to know the customers’ expectations [27].

“We Working software over comprehensive documentation” [33] – It is imperative in agile development to have valid and correct working code, even at the cost of not having top notch documentation. Others argue that many test techniques need extensive documentation to properly test a codebase. However, it is fundamental to note that working software is not simply having the codebase with suitable levels of documentation but also having the codebase extensively tested [27].

“Customer collaboration over contract negotiation” [33] – Testers are the crucial collaborators with the customers. This is due to the fact that the tester elaborates on the customers’ expectations to ensure the system is capable to handle all the possible inputs the customer will be providing to the live system. Although contract negotiation is important especially for financial reasons and customer expectation management, it is vital to ensure the requirements arriving to the development team is in sync with the customers’ expectations [27].

“Responding to change over following a plan” [33] – In the modern world, due to rapid changes in lifestyles and markets, the customers’ requirements tend to change rather frequently even during the development process. This might cause project instability and invalid code. To try to overcome this issue, code is written in an agile manner with short iterations of around two to three weeks. At the end of each iteration, the client provides feedback on the implemented features. If the additions or changes are not accepted, compared to traditional development, the costs of ensuring customer satisfaction by adhering to his requirements are minimal [27].

2.2.2 Testing in Agile environments
Through testing an indication of a program’s completeness, validity and correctness is obtained. This information is essential to the process of agile development, as relaying this information to the developer before considering an iteration to be complete informs the developer that a feature is not up to standard and the developer would fix the feature instead of consider it to be complete. It also helps confirm the creative approaches and more importantly serves as an indication that the current addition to the codebase is correct [27]. This section discusses how testing is employed in agile environments at different levels and test analysis techniques.
2.2.2.1 Terminology
The following terms are commonly used in testing environments:

- **Test case** – A test case is a set of tests which test a particular feature or operation in a codebase [34].
- **Test suite** – A set of test cases which test holistically a codebase [35].

2.2.2.2 Test Driven Development
Test driven development is a concept in agile software development. Tests are written before sections of code. Then, enough code is written for the test to pass. Test driven development might tend to be frustrating and slow; however, it produces elegant and modular code. As such tests are short and test a particular feature, tests may be considered as documentation in terms of an example of its use. Furthermore, code written using the test driven development technique tends to have less test failures [27].

2.2.2.3 Levels of testing
There are four levels of testing used in agile development; unit, integration, system [19] and user acceptance [28] testing.

![Testing Pyramid Diagram](image)

Figure 3 illustrating the testing pyramid adapted from [19] [28] [36]

The different levels of testing test different aspects of a codebase, as illustrated by Figure 3. Although this dissertation focuses on unit testing, it is helpful to also briefly discuss integration, system and user acceptance testing.
2.2.2.3.1 Unit testing

The most basic form of testing is Unit testing. Unit testing is testing performed on the smallest or smaller set of code. This usually consists of testing a method or a few related methods. This white-box (testing done by a person who knows how the system works) testing method tests only one particular sub-feature of a system and hence can be parallelised [19]. Furthermore, this testing strategy individually tests the sub-parts of a system [28].

2.2.2.3.2 Integration testing

Integration testing seeks to determine if multiple components of the system work correctly together. Hence, this tests data transmissions and conversions between independent modules of the system [19]. Two modules can pass their separate unit tests, which imply they work correctly independently, but fail integration tests, which usually show their interconnection is flawed [28].

2.2.2.3.3 System testing

System testing is a technique to ensure that the system works correctly overall and all modules are successfully communicating and performing correctly [19] [28].

2.2.2.3.4 User Acceptance testing

User acceptance testing is the final set of tests and is performed at the top most level. This is usually done manually by the client (or product manager), not the developers or testers themselves [19]. The aim of user acceptance testing is to determine if the system developed is what was required and if all the features are present and working correctly [28].

2.3 Test Suite Adequacy Criteria

Test Suite Adequacy Criteria are techniques which attempt to illustrate the thoroughness of test suites performed at different levels. One common technique is known as code coverage analysis. Code coverage analysis is an automated technique which illustrates which statements, branches conditions and paths of code are covered by tests [28] [37].
Statement Coverage – a test technique designed to ensure developers write tests which execute all the statements in a codebase [28] [37]. Statement coverage is calculated as a percentage, as shown in the lower part of Figure 4. Yellow and green highlighting indicates covered statements whilst red indicate uncovered statements.

Branch Coverage – this technique is designed to ensure all the outcomes of each branch are tested [28] [37]. The yellow area in Figure 4 illustrates that in this case, not all branches were tested. If all branches were tested, the highlighting would be green in colour, otherwise if no tests were testing the branches, such highlighting would be red.

Condition coverage – condition coverage is similar to branch coverage, however the branching is not performed on an operation but on a variable directly [38]

Path Coverage – a test technique which determines which of the paths created by branching instructions have been tested [37].

Most tools only provide statement and path coverage analysis. Such analysis is not thorough enough to ensure that the test suite is adequate and hence provides a false sense of security [28].
2.4 Mutation Testing

2.4.1 Introduction
Mutation Testing has a number of benefits, which are of even more importance in agile environments where testing is a key part of the development process [9]. Such benefits include exposing the false sense of security provided by code coverage analysis and the fact that Mutation Testing makes it easier to create some tests which one would not have initially thought of [29]. Further to Section 1 Introduction, the following information is required for a holistic understanding of Mutation Testing.

2.4.2 The Mutation Testing Process

![Image of the Mutation Testing process](image-url)

Figure 5 illustrating the Mutation Testing process, from [1]

A brief description of Mutation Testing process was provided in Section 1 Introduction. A more detailed description of Mutation Testing was provided in Jia and Harman [1]’s work, as illustrated in Figure 5.

In Mutation Testing, from an original program P, a number of modified programs P’ called mutants are produced by applying a few syntactic modifications to the developers’ program P. These syntactic modifications are achieved by means of mutation operators as described in the next section. Simultaneously, the Test Set T is
run on the original program \( P \), so as to verify that the test suite is correct. Correctness is defined as a test suite which is “successfully executed” [1] against program \( P \).

If correctness is not met, \( P \) needs to be fixed. However, if this condition is met, \( T \) is run on the mutants, \( P' \). As previously described, if the test fails when run on a mutant, this is known as a \textit{killed mutant}. If all mutants \( P' \) are killed, the test suite is thorough and Mutation Testing stops. If some mutants are not killed, the tests are not testing the code exhaustively enough [1]. However, in some cases a mutant can never be killed as it will “always produce the same output as the original program” [1]. Examples include mutants which leave the semantics of a program unaltered (such as mutating dead code), mutations that suppress speed improvements alter state, or which cannot be triggered [1] [6]. Such mutants are called equivalent mutants [1].

2.4.3 Mutation operators
A mutation operator is the operation which injects a small defect in a program’s source code by replacing or removing an operator.

2.4.3.1 Justifying the benefits of only seeding small defects in Mutation Testing
Back in 1978, DeMillo et al [39] established the Competent Programmer Hypothesis. This hypothesis assumes that the programmers are competent, hence the code they develop is either correct or incorrect however very nearly correct. Hence, if the program is incorrect, only one or few modifications are necessary to fix the fault. Using similar reasoning, during Mutation Testing, only minor syntactic changes are required from mutant operators, as such slight changes change the semantics of the code entirely [1].

DeMillo et al [39] expanded the Competent Programmer Hypothesis to the Coupling Effect. The coupling effect states that “Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors” [39]. This means that tests should detect tiny flaws which change the desired output. Thus, such tests should also expose larger or more complex flaws in this process.

Offutt [40] expanded the Coupling Effect to the Coupling Effect Hypothesis and the Mutation Coupling Effect Hypothesis. The Coupling Effect Hypothesis states that “complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex
faults” [40]. By the Coupling Effect Hypothesis, Offut meant that larger flaws are made up of multiple individual small faults. Thus detecting the individual smaller bugs would in turn would detect the larger bug.

The Mutation Coupling Effect Hypothesis states “Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will also detect a large percentage of the complex mutants” [40]. This is similar to the Coupling Effect but in perspective with Mutation Testing. This means that if a test suite is able to detect a simple mutant, knowing that a simple bug might be part of a more complex bug, a test suite will most probably also detect a more complex mutant.

### 2.4.3.2 Types of mutation operators

Table 1 illustrating a few types of mutation operators [1]

<table>
<thead>
<tr>
<th>Mutation operator type</th>
<th>Description</th>
<th>Mutation operator example</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional mutation operators</td>
<td>A simple arithmetic or branching change [1].</td>
<td>AND to OR</td>
<td>if (a &gt; 0 &amp;&amp; b &gt; 0) becomes if (a&gt;0 [b &gt; 0)</td>
</tr>
<tr>
<td>Constant substitution</td>
<td>Replacing one constant with another [29]</td>
<td>-</td>
<td>double pi = 3.14159; becomes double pi = 0;</td>
</tr>
<tr>
<td>Object Oriented mutation operators</td>
<td>Introduced by Kim et al [23] to improve mutation in object oriented programs</td>
<td>Type replacement operator – replace a type with compatible types</td>
<td>Number x = new Float(); becomes Number x = new Integer();</td>
</tr>
<tr>
<td>Higher order mutation</td>
<td>Introduced by Jia and Harman [1], inserts more than one defect in any given mutant [1].</td>
<td>Mutate equality and addition</td>
<td>if (x == true) y++; becomes if (x != true) y--;</td>
</tr>
</tbody>
</table>

The traditional mutation operators (arithmetic and branching operators) were designed to modify either variables or expressions as shown in Table 1. Certain operators found in source code can generate more mutants than others [1]. This is because mutation operators can only replace operators of the same type and operators which have the same number of parameters [32], of which there may be more than one.
Due to new programming concepts such as Object Oriented programming, traditional mutation operators are not sufficient to adequately test a test suite’s rigorousness, due to faults generated in the Object Oriented world [1]. Hence, twenty new mutation operators were created by Kim et al [23] to test Object Oriented aspects [23]. One such mutation operator is changing constants, such as replacing zero with one [29].

The mutation operators previously described are known as “First Order Mutants” [1]. Jia and Harman [1] introduced the concept of Higher Order Mutation. This technique inserts more than one defect in any given mutant [1] as described in further detail in section 2.4.6.1.

### 2.4.4 Mutation Score

As mentioned in the Introduction, Mutation Testing determines a “Mutation Score”, also known as an “adequacy score” [1], which represents the quality of the tests. The Mutation Score is the ratio of the number of killed mutants\(^1\) over the total number of non-equivalent mutants. The objective of Mutation Analysis is to increase the Mutation Score to 1, which is achieved by the developer thoroughly testing the code, signifying the test set is adequate to detect all the errors caused by the mutants [1] [7].

### 2.4.5 Code coverage analysis versus Mutation Testing

As already established, statement coverage provides the percentage of lines of code covered by tests. However, in certain cases, statement coverage is not enough and Mutation Testing comes into play to test the thoroughness of tests [29]. In a study by Smith and Williams [29], it was established that when a developer improves the Mutation Score through the addition of tests, this also increases statement coverage. However, increasing statement coverage might not necessarily increase the Mutation Score.

Aaltonen et al [30] found in their study that there were several codebases where statement coverage was close to 100% but mutation score varied from 55% to 98%. Also, Aaltonen et al [30] concluded that if test quality is determined by statement coverage, it is easily cheated. On the other hand, Mutation Score reveals the true thoroughness of the tests.

---

\(^1\) A killed mutant is a mutant which is able to detect a change to the original program [1].
2.4.6 Existing cost reduction techniques

![Percentage of publications on optimisations of Mutation Testing](image)

Figure 6 illustrating the percentage of publications on optimisation techniques of Mutation Testing [1]

Figure 6 illustrates the publications on different Mutation Testing optimisations. Localised Mutation or a similar approach was not mentioned in this Systematic Literature Review, published a year ago at the time of writing. An optimisation’s goal can be described as trying to achieve one of the following, to “do fewer”, “do smarter” or “do faster” [7]. These three classifications for mutation optimisations were proposed by Offutt and Untch [7]. Others applied optimisations which can also be categorized in these three categories. The relevant methodologies are summarised below.

2.4.6.1 “Do fewer” [7]

A lot of mutation operators are not effective. In fact, in one particular study, 6 out of 22 mutation operators were responsible for 40%-60% of mutation kills [7]. Thus the concept of Selective Mutation was born. Only the mutation operators which are very effective at detecting flaws should be applied [1] [3] [7]. Offutt et al [3] found Selective Mutation to save time by up to 30% when discarding two ineffective mutant operators and exceeded 60% when discarding six, whilst the mutation score only dropped by less than 2%. This philosophy is applied in the Javalanche mutation suite [5].
Mutant Sampling is another technique which uses a discriminator to skip testing on mutants. This discriminator is either some form of heuristic or a random discriminator [1]. Using a random discriminator is however not as effective as using a heuristic discriminator. Such a technique was found to reduce the effectiveness by just around 16% [7].

Mutant Clustering is a subset of Mutant Sampling. The mutant operators are chosen using clustering algorithms [1].

Also, Jia and Harman’s [1] Higher Order Mutation, inserts more than one defect in any given mutant. A Higher Order mutant was said to be harder to kill than a First Order Mutant (a mutant with only one defect). This is thus a more effective approach [1].

2.4.6.2 “Do smarter” [7]

Weak Mutation is a process of comparing a program’s state to the mutant’s state at particular code locations. If states are not identical, a mutant can be said to be effective [7].

Mutant Schema Generation is a technique which encodes the mutants into one source level program. The program is then compiled normally with the native compiler and is executed in the same environment at ordinary speeds. Mutant Schema systems do not need to provide any run-time semantics and environment configurations, and thus are significantly less complex and easier to build than interpretive systems, thus remaining portable [2].

Schuller and Zeller [5] use code coverage analysis to only run tests relevant to the mutated code [5]. This was achieved by initially running every test and determining the lines each test covered. When a statement was mutated, only the tests covering that statement were executed. This technique is employed in Javalanche to limit the overheads in running irrelevant tests [5]. Furthermore code mutation is performed directly on bytecode, which is defined and described below, as this eliminates the overhead of compilation [5].
2.4.6.2.1 Java Bytecode

Like modern languages, the Java platform converts the source code to intermediary code, more commonly known as bytecode [30].

![Diagram of Bytecode Structure](image)

**Figure 7 illustrating the bytecode structure [30]**

The typical bytecode is split up in three sections. These sections are the array of variables, the operands and the constant pool as illustrated in Figure 7. The arrays of local variables are the class attributes or fields. The operand stack contains the class’ opcodes (the technical term for a Java Virtual Machine operator) together with their line numbers. Every method has its own scope. Certain operators may also push addresses or primitive values to the stack. The constant pool contains the class’ constants as well as metadata on the class. The constant pool also contains some external metadata such as the mapping from bytecode line numbers to the .java source files [30].

2.4.6.3 “Do faster” [7]

“A lot of previous work has focused on techniques to reduce computational cost.” [1]

To avoid compilation slowdowns, Krauser et al [10], developed a compiler-integrated mutation application that avoids a lot of the overhead of the compilation bottleneck but is able to execute compiled code. In this method, the program being mutated is compiled by a custom compiler. During the compilation process, mutants are generated as code substitutes. Execution of a particular mutant requires only that the
appropriate code “patch” be applied prior to execution. Replacing the code substitutes is inexpensive and the mutant executes at compiled speed [1] [7].

As the execution of tests against each mutant is fully independent, the running of tests should be parallelised to significantly reduce the running time of execution of tests [1] [7]. Javalanche exploits this fact [5].
3 Specification and Design

3.1 Introduction
In this work, the agile nature of software development will be exploited. Although modern development environments employ agile development, this consideration was not reflected in Mutation Testing. Hence, this work introduces the concept of Localised Mutation to attempt to optimise Mutation Testing. This will be explored in further detail in section 3.3, Other optimisations – Localised Mutation.

3.2 Chosen optimisations from the literature
The optimisations found in previous literature as described in the literature review were analysed. From those the below were chosen to be included in the tool.

- Bytecode manipulation - Mutations will be performed directly on bytecode also similar to Javalanche [5]. Due to the nature of the Java Virtual Machine (JVM)\(^2\), this is more suitable than mutant schemas, interpretive systems or compiler-integrated methodologies as there is no need for any configurations whatsoever once bytecode manipulation has occurred. Hence bytecode manipulation was chosen.
- Localised testing - Similar to Javalanche [5], only tests which directly test a given mutant are executed. This is of great advantage in terms of execution time as it is considerably reduced.
- Concurrent test execution - Concurrent execution of tests was implemented [5], but as described later in section 4.2 Challenges Encountered, this was discarded due to the misbehaving of mutants as discussed in section 4.2.4.

3.3 Other optimisations – Localised Mutation
In this work, the concept of Localised Mutation is introduced. No reference to such a concept was found in the literature surveyed. This concept utilises the fact that agile development is performed in iterations or short cycles. In literature, this concept is not exploited as mutation is always performed on large sections of code. In Localised Mutation, mutation is performed on the code developed in a given cycle. Hence, the number of mutants generated will be significantly smaller. Therefore, the Execution

---

\(^2\) The program responsible for executing Java bytecode [12]
Time will also be sizably smaller and should make Mutation Testing much more feasible than it is currently today.

3.4 Discarded optimisations from literature

As the main focus of this work was to reduce the execution time, the following points were discarded as they were not conducive to reaching the desired goal, or not implementable in the timeframe.

- Selective Mutation – Recall selective mutation reduces the number of mutations by selecting only the mutant operators which have the highest mutation score [3]. Although this could contribute to a considerable reduction in Execution Time, this optimisation was not implemented for two reasons. Firstly, the Localised Mutation optimisation should be significantly more effective and secondly, as this work is more of a proof of concept, only a few mutant operators were implemented.

- Weak Mutation – Recall Week Mutation is the process of comparing a program’s state to the mutated code’s state at particular code locations [7]. This is very inefficient as comparing the state of the whole application under test for each mutant is very expensive.

3.5 Discarded techniques from literature

This section highlights some techniques or approaches discarded from literature.

3.5.1 Object Oriented Mutation Operators

As Kim et al [23] described in their work, the introduction of Object Oriented programming generate the need to create new mutation operators. Whilst the concept of object oriented mutation operators is indeed interesting, it was deemed to be beyond the scope of this work, especially in the context of an FYP.

3.6 Assumptions

The list below summarises the assumptions taken in this work.

- As mutants are generated in a systematic way and higher order mutation is not performed, it can be assumed that no Equivalent Mutants were generated provided the codebase did not contain dead code,

- Source and test classes are in their respective folders, as per Java convention,
- Test classes, as per Java convention, are in the same package as the source class they are testing,

### 3.7 Main Components of the proposed Mutation Testing tool

The tool is split up into five components; Change detection, Test relevance, Mutator, Test runner and the Front end.

![Figure 8 the activity diagram of the system](image)

Figure 8 illustrates through a UML activity diagram the flow of execution of the system. The components referred to in the figure are described below.
3.7.1 **Change Detection – For Localised Mutation**  
This module listens on source classes to detect changes. By comparing the time of last modification attribute of the class’ files to the time of the last modification, a change to a class level can be detected. The class is then compared to the previous version which is cached to pinpoint the change to a method level.

3.7.2 **Test Relevance – for Localised Testing**  
This module parses the tests to get the tests which directly call the modified methods.

3.7.3 **Mutator**  
The mutator module mutates the modified classes with the implemented mutation operators. The modified classes are then stored on secondary storage.

3.7.4 **Test Runner**  
The test runner uses the JUnit test runner to run JUnit tests which test the mutated classes.

3.7.5 **Front end**  
The front end module has two main roles. Firstly, it provides easy access to three configurables, namely the path to the project and the relative paths to the source and test folder. Secondly, it provides detailed statistics on the passed mutants by providing the line number and the operation modified which tests did not detect.
The development of the mutation testing tool was performed in an agile manner and hence split up in iterations. The development was performed in this manner so as to compare the potential improvements in each of the 3 iterations. At each iteration, a different optimisation was introduced. Also during each iteration, the optimal implementation given the techniques chosen were employed. This is to say that from
one iteration to the next, the only improvements were those documented. Figure 9 illustrates the flow of the tool at all Iterations, including which optimisations were added for each iteration.

3.8.1 Initial program design
The base program was needed, as to be able to employ incremental iterations. This program is known as Iteration 0, as described below.

3.8.1.1 Iteration 0: The inefficient (traditional) approach
Iteration 0 is the traditional Mutation Testing approach, and hence is very inefficient. However Iteration 0 will serve as the benchmark. In this iteration, the complete source is mutated directly using bytecode since as established beforehand; this is very efficient as it bypasses compilation bottlenecks. All the tests are run against all mutants. However, once a test fails, the mutant is immediately considered to be killed, that is, no further running of tests are required.

3.8.2 Incremental improvements
The following incremental improvements were then employed.

3.8.2.1 Iteration 1: The Localised Mutation approach
During this iteration, changes in the sources are detected. Mutation is only performed on changes or new methods. Work in this iteration will attempt to make Mutation Testing feasible for agile environments hence answering the research question.

3.8.2.2 Iteration 2: The Localised Mutation and Localised Testing approach
In this iteration, Localised Testing [5] was the additional optimisation to the Localised Mutation. Only the tests which directly tested the localised mutant were run.
4 Implementation

4.1 Technologies and Skills acquired

The below points are some of the specific technologies and concepts acquired to adequately design and implement the tool required for the evaluation.

4.1.1 Understanding of the Java Virtual Machine and bytecode

As established earlier, mutations performed directly on bytecode (.class files) are much more efficient than recompilation [5]. Hence understanding of how the JVM works particularly with reference to bytecode was mandatory. In short, the process of compilation is the following. After compilation, a class file per Java file is created. Each class file contains JVM instructions (or bytecode), symbol tables, class constants [12], and other helpful information such as the mapping from bytecode line number to source code line number.

4.1.2 Byte Code Engineering Library (BCEL)

The Apache Commons BCEL [11] library plays two vital roles in the tool developed for this evaluation. Firstly, the library is used to efficiently mutate some JVM operators. Secondly, when a change is detected at a class level, the BCEL library allows a user to obtain the bytecode of a method. This was used to pinpoint the method or methods causing the change by comparing the latest methods’ bytecodes to their previous’ version.

4.1.3 JUnit test runner

As testing is framework specific, the tool only supports JUnit tests. This is so as JUnit is the most used testing framework in the agile environment [15]. Hence, the tool uses the JUnit runner to run a test class’ method and report back the tests’ results.

4.1.4 Other notes

Other than the major components; BCEL [11], JUnit test runner library [21] and the Apache Commons IO (used for file operations) [20], for maximum efficiency all other components of the Mutation Testing tool were programmed from the ground up. This approach was chosen so as to prevent using external libraries. Using external libraries forces a developer to use pre-defined data structures resulting in utilising computation time to translate data types due to compatibility reasons instead of actually performing Mutation Testing.
4.2 Challenges Encountered

During the development of the Mutation Testing tool, various challenges were encountered. Such obstacles are documented in this section.

4.2.1 Mutation

Table 2 implemented mutation operators

<table>
<thead>
<tr>
<th>Bytecode operator</th>
<th>Mutated To</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>idiv</td>
<td>imul</td>
<td>Integer division to multiplication</td>
</tr>
<tr>
<td>imul</td>
<td>idiv</td>
<td>Integer multiplication to division</td>
</tr>
<tr>
<td>iinc</td>
<td>iinc</td>
<td>Integer addition parameter multiplied by -1</td>
</tr>
<tr>
<td>if_icmple</td>
<td>if_icmpgt</td>
<td>Integer less or equal to greater than</td>
</tr>
<tr>
<td>icmpgt</td>
<td>if_icmple</td>
<td>Integer greater than to less or equal</td>
</tr>
<tr>
<td>ifle</td>
<td>ifgt</td>
<td>Integer less or equal to greater than</td>
</tr>
<tr>
<td>ifgt</td>
<td>ifle</td>
<td>Integer greater than to less or equal</td>
</tr>
<tr>
<td>ifge</td>
<td>iflt</td>
<td>Integer greater or equal to less than</td>
</tr>
<tr>
<td>iflt</td>
<td>ifge</td>
<td>Integer less than to greater or equal</td>
</tr>
<tr>
<td>if_icmpeq</td>
<td>if_icmpne</td>
<td>Integer equals to not equal</td>
</tr>
<tr>
<td>if_icmpne</td>
<td>if_icmpeq</td>
<td>Integer not equal to equals</td>
</tr>
<tr>
<td>ifeq</td>
<td>ifne</td>
<td>Boolean equals to not equals</td>
</tr>
<tr>
<td>ifne</td>
<td>ifeq</td>
<td>Boolean not equals to equals</td>
</tr>
<tr>
<td>if_acmpeq</td>
<td>if_acmpne</td>
<td>Object equals to not equal</td>
</tr>
<tr>
<td>if_acmpne</td>
<td>if_acmpeq</td>
<td>Object not equal to equals</td>
</tr>
<tr>
<td>ifnonnull</td>
<td>ifnonnull</td>
<td>Reference null to reference not null</td>
</tr>
<tr>
<td>ifnonnull</td>
<td>ifnonnull</td>
<td>Reference not null to reference null</td>
</tr>
</tbody>
</table>

The tool employs 17 mutation operators. Table 2 shows the list of JVM operators together with their corresponding mutated operator. These mutation operators were selected to be included in the tool as they are all simple and hence, by the Coupling Effect as described in the Background and Literature Review section 2.4.3.1, such simple defects are usually enough to also expose larger defects.

“An instruction A can be replaced by an instruction B if A and B operate on the operand stack in the same way - they expect the same number and type of arguments on the stack before the operation and leave the same number and type of arguments on the stack after operation.”[9] Such instructions had to be manually identified. The mutation operators implemented are summarised in Table 2.

4.2.2 Detection of changes

To efficiently detect changes on code at a class level, an individual thread is kept listening on each class file generated. Once a change in a class is detected, the same independent thread identified the method or methods changed. This was done so as to
prevent queuing of detection of changes and decrease the running times of both Iterations 1 and 2.

4.2.3 Detection of tests
In order to reduce execution time, the use of call graphs to detect relevant tests had to be abolished. This was necessary as the available tools such as Soot [22] took long to generate the call graph of a method and used huge amounts of memory. This approach would have been used to detect tests which directly test a modified method. Instead, test source code was parsed to detect the tests directly testing the modified methods. As the Java convention is that tests are placed in the same package as source files, when a method was altered, test parsing was used to determine the relevant tests in the test folder in the same package as the source. This process is described in Algorithm 1.

∀ test sources in the same package as the source
    ∀ the methods in the test class
        if method contained a call to the method modified
        test is relevant

Algorithm 1 determining if a test is relevant to a method

Even though detecting tests by parsing source is much more efficient than building call graphs, the process is still relatively slow. As this process is easily parallelisable per class, each class detected the tests relevant to its changes on a separate thread for efficiency. This gave a significant speed boost in Iteration 2 as once the tests on a modified class have been executed, the tests could immediately commence on the next modified class.

4.2.4 Execution of tests

Equation 1 illustrating the time limit to run a test [9]

\[
\text{Runtime Limit} = 10 \times \text{Original runtime} + 2s
\]

Equation 1 was devised by Irvine et al [9] as a way to limit the running time of each test on misbehaving mutants, such as mutants which contained infinite loops due to mutation.

This methodology was not taken in consideration as if it had to be employed; a run of all the tests has to be performed before running Localised Mutation and testing. Such an approach would defeat the purpose of limiting the Execution Time. However, a
similar approach was taken. During the evaluation of Iteration 0 (the traditional Mutation Testing approach), it was observed that test executions always took less than a second to run. Therefore, in this study, a misbehaving mutant is any test takes more than three seconds to complete.

4.2.4.1 Number of tests
Another challenge encountered is the amount of tests a method is covered by. As the point of Mutation Testing is to detect if a suite’s unit tests are thorough, if a method is detected to be modified and there are only a few relevant tests, the Mutation Score is likely to be appalling. However, a more “severe” example is when the mutant has no relevant tests. This might happen when the developer has just written his new code and committed the code without writing tests. In this instance the method is not tested at all, and thus the mutant is immediately considered to be not killed.

4.2.4.2 Speedup of execution of tests
Execution of parallelised tests proved to be tricky and ineffective. As the Java Class Loader loads classes and resources in relative directories, each mutant test execution had to have a copy of all the classes and resources to a separate location as otherwise mutants would be mixed up. This was performed and results were discouraging, as initially expected. This is because in iterations 1 and 2, the execution time of tests is so short that the I/O time required to copy the files was around 5 times longer than the execution itself. Furthermore, the amount of memory required when handling misbehaving mutants due to infinite loops also scaled. At one point, two misbehaving tests where running simultaneously and the memory usage rose to an alarming 6GB. For these reasons, parallelisation of the execution of tests was not implemented.
5 Evaluation

5.1 Background on existing evaluation techniques

The literature was surveyed to determine what similar works used as evaluation methodologies and metrics. The following were identified.

5.1.1 Metrics

Below are the metrics employed in previous works. These metrics are standard and no other metrics are used in the field.

- **Mutation Score** [3] [4] [5] - also known in literature as the “adequacy score” [1], “kill rate” [6], or “mutation adequacy score” [2]. The Mutation Score is the percentage of the number of killed mutants divided by the total number of non-equivalent mutants [2] (Equation 2). The objective of Mutation Testing is to increase the Mutation Score to 100%. This is achieved by the developer thoroughly testing the code, signifying the test suite is adequate and thus able to detect all the errors seeded by the mutants [1] [3]. Recall that a killed mutant is a mutant which is able to detect a change to the original program [1]. Mutation Score can also be quoted as a ratio instead of a percentage [1] [3].

\[
\text{Equation 2 Mutation Score [2]} \\
\text{Mutation Score (\%) } = \frac{\sum \text{killed mutants}}{\sum \text{mutants} - \sum \text{equivalent mutants}} \times 100\%
\]

- **Execution time** [2] [5] - the time taken to perform multiple mutations and execute tests on each mutant.

5.1.2 Evaluation Methodologies

Below are the different evaluation methodologies used in literature:

- **Comparison with another Mutation Testing tool** [2] - Untch and Offutt [2] compared Mothra and IMSCU based on execution time. This methodology was not selected as the main focus of this work is to attempt to make Mutation Testing feasible not by simply making the current approaches evaluate faster but by creating a new approach to reduce the number of mutants.

- **Comparison between different mutation algorithms** [4] - Offutt et al [4] compared selective mutation with non-selective mutation based on Mutation
Score. This methodology was chosen as different iterations contain different algorithms. Further detail can be found in section 5.2 Chosen evaluation technique.

- **Evaluating using the same algorithm on different codebases** [3] [5] - Mathur and Wong [3] evaluated 4 codebases with randomly generated tests. However when random generation failed, tests were manually created. Schuler and Zeller [5] calculated metrics on 7 codebases. This methodology is ideal, however due to time constraints, it was considered infeasible.

- **Human-seeded faults** [23] – This is not applicable for this work as it involves human effort in “mutation” which would make the process inefficient at best. Human-seeded faults generation consists of inserting faults in the system and identifying which mutants were killed by the faults by hand. This is a repetitive task and is thus ideal for automated computation. Andrews et al [23] stated that their method of evaluation was likely to underestimate the effectiveness of the system, especially in the case of cost-effectiveness of Mutation Testing.

- **Developer Experience and Acceptance** [9] – Irvine et al [9] used their Mutation Testing tool Jumble in a developer environment. They found that developers, when they applied themselves, could reach mutation scores of 95%, which resulted in approximately as much test code as code developed. They also made mutation scores public which incentivised developers to maintain high mutation scores, as the scores could be seen by peers and management. This evaluation method is not relevant as it does not help answer the research question.

### 5.2 Chosen evaluation technique

<table>
<thead>
<tr>
<th>Table 3 illustrating the features against Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Iteration 0</td>
</tr>
<tr>
<td>Iteration 1</td>
</tr>
<tr>
<td>Iteration 2</td>
</tr>
</tbody>
</table>

The purpose of this evaluation is to answer the research question, “Can Mutation Testing be made efficient to be practical for everyday use, particularly for agile environments?” This will be addressed in two ways. Recall, Iteration 0 is the traditional Mutation Testing approach, which mutates the whole codebase and runs all
the tests on each mutant. Iteration 1 mutates only the changed code but runs all the
tests. Iteration 2 mutates only the changes and runs only the tests which directly test
the modified methods. A summary of the iterations can be seen in Table 3.

Firstly, the target is to achieve a faster Execution Time per iteration whilst having a
lower Mutation Score in Iteration 2 than Iteration 1. This is due to the fact that
Mutation Testing is about determining the extent to which unit tests of a program are
thorough. Hence, as Iteration 2 runs only the tests which directly test a method, tests
run in Iteration 2 are the unit tests. Therefore, if a mutant is killed in Iteration 1 and
not in 2, the mutant was killed by an integration test not a unit test and thus does not
satisfy the concept of Mutation Testing. This signifies an incomplete test suite. Since
agile development is iterative in nature and relies on the concept of short frequent
feedback loops, it would be desirable to enable developers to get frequent feedback
from their Mutation Testing tool. The Execution Time should be in the seconds range
and the Mutation Score in Iteration 2 should be lower than the score obtained in
Iteration 1. The chosen evaluation technique consists of a comparison between
different mutation approaches developed per iteration, similar to Offutt et al’s [4]
evaluation. Due to time constraints, comparison of the same algorithm on different
codebases and comparison with another Mutation Testing tool was deemed unfeasible
in the context of this work.

Secondly, the relative time taken and cost required for a developer to add sufficient
tests so as to kill passing mutants will be reviewed. This is so as in a practical
environment, it is futile to only consider the time it takes to evaluate the rigorousness
of a test suite through Mutation Testing and not consider the time required to add
further tests to increase the Mutation Score.

5.2.1 Evaluation Setup
A number of considerations were taken before evaluation was started.

- Codebase – the Java project source the tool will mutate and test upon,
- Metrics Calculation – what metrics this evaluation will employ,
- Evaluation machine – on what machine this evaluation will be run on,
- Experiment Data – in depth statistics on the codebase used.
- Procedure – the procedure undergone in this evaluation as previously explained
  in Section 3 Specification and Design.
5.2.1.1 Codebase

For the evaluation, a search was performed for a codebase which was open-source and had access to a revisioning system such as Subversion (SVN) or Concurrent Version Systems to be able to retrieve all the version of the codebase. Due to the fact that no Object Oriented mutation operators were implemented, it was ideal if the codebase used minimal Object Oriented programming concepts and extensively used branching and arithmetic operators. After some research, the Apache Commons CLI library [13] was chosen. This API provides functionality for parsing command line arguments and is also able to display print help messages on the options available for a given command line tool [13]. The CLI library is approximately 5,000 lines of code in length. This length of the code is ideal for this evaluation, as the running time of mutation and execution of tests on the generated mutants in Iterations 0 is limited, hence the running of the whole test suite is a number of minutes instead of a number of hours.

5.2.1.2 Evaluation machine

The evaluation was performed on the machine specified in Table 4.

<table>
<thead>
<tr>
<th>Machine type</th>
<th>Virtual</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i7 (Sandy Bridge) running at 2.8GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>4GB</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 64bit</td>
</tr>
<tr>
<td>Java</td>
<td>JRE 1.7 64bit</td>
</tr>
</tbody>
</table>

The evaluation was purposely evaluated on a virtual environment so as to keep background operations to a minimum, which otherwise would have a direct influence on the Execution Time. A Linux environment was chosen as it is a robust and efficient environment. Furthermore, the 64bit edition was used to capitalize on the machine’s performance.
5.2.1.3 Metrics Calculation

The application handled the calculation of the Execution Time and Mutation Score calculation. The execution time was calculated programmatically by obtaining the time in milliseconds from the point when the user initiates mutation until testing is complete. The Mutation Score was also calculated by the application in the manner specified in section 5.1.1 Metrics.

To calculate these metrics, two code base instances are required. The code base X is checked out from the SVN representing the code at a certain point in time. Code base Y is checked out representing the code at a later point in time. The size of the time frame between codebases X and Y is discussed in the next section. In Iteration 0, as the tool mutates the complete codebase, one can expect a large number of mutants. However in Iterations 1 and 2, as only methods whose code has been changed is mutated, one can expect a considerably smaller number of mutants. If the changed method is short, there will be very few mutations. It is to be noted that due to the small number of mutations, statistically, the result would be sensitive, as the sample size is smaller and might be too small to be conclusive. For instance, if 2 mutants are generated and both are killed, the Mutation Score would be 100%.

5.2.1.4 Experiment Data

5.2.1.4.1 Overview

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Revision X</th>
<th>Revision Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1, representing a 2 hours of work</td>
<td>1091552</td>
<td>1091575</td>
</tr>
<tr>
<td>E2, representing a day of work</td>
<td>759392</td>
<td>779054</td>
</tr>
<tr>
<td>E3, representing a week of work</td>
<td>780163</td>
<td>955156</td>
</tr>
</tbody>
</table>

Three experiments per iteration, E1, E2 and E3 were carried out. The changes performed by the developer between SVN repository revisions X and Y simulate the work performed in the given timeframe, as shown in Table 5. The first experiment roughly reflected a developer’s 2 hours’ worth of work, from revision Xe1 to Ye1. The second reflected a day’s work, from revision Xe2 to Ye2. The third reflected a week’s worth of work; Xe3 and Ye3. The application uses revision X as a baseline to identify what changes have been made until revision Y.
5.2.1.4.2 Identifying revisions

Revisions were identified with the help of the TortoiseSVN tool [14] to be used as codebase changes the experiments.

Figure 10 illustrating the statistics TortoiseSVN tool

SVN repository revision logs and commits count per quarter charts using the Statistics tool as shown in Figure 10 were very important for the selection of revisions since the peak of changes within the source were identified. The peaks exposed the major changes in the source. Furthermore, the Compare revisions tool was vital to review changes between any two revisions on a line-by-line and file-by-file level.

Table 6 showing the changes performed in revisions which signified a period of work

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Source changes</th>
<th>Test changes</th>
<th>Configuration changes</th>
<th>Class documentation changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;5</td>
<td>&lt;5</td>
<td>0</td>
<td>&lt;5</td>
</tr>
<tr>
<td>2</td>
<td>15-20</td>
<td>10-15</td>
<td>0</td>
<td>10-15</td>
</tr>
<tr>
<td>3</td>
<td>50-55</td>
<td>40-45</td>
<td>&lt;5</td>
<td>55-60</td>
</tr>
</tbody>
</table>

Table 6 shows the criteria for selecting the revisions for experiments 1 to 3.
5.2.2 Procedure
The procedure depicted in Algorithm 2 illustrates the iterative approach the evaluation undertook for all three iterations.

∀ iteration i
∀ experiment e
1. The iteration radio button i was selected on the application (for repeatability of this experiment, in the provided application, one can select the algorithm at different levels of optimization i.e. the iteration number),
2. the application was initialized on CLI revision $X_e$,
3. changes were made to update the codebase to revision $Y_e$,
4. Mutation Score and Execution Time were obtained from the application.

Algorithm 2 illustrating the procedure of the evaluation

5.3 Results
The evaluation was performed three times on the setup and procedures as described in sections Evaluation Setup and Procedure respectively. The Execution Time in the results below are the average of the three runs.

Table 7 illustrating the results for Experiment 1 – 2 hours of work

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Mutants</th>
<th>Killed</th>
<th>Mutation Score</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>349</td>
<td>314</td>
<td>90%</td>
<td>58s</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>30</td>
<td>100%</td>
<td>9s</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>25</td>
<td>83%</td>
<td>5s</td>
</tr>
</tbody>
</table>

Table 8 illustrating the results for Experiment 2 – a day’s work

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Mutants</th>
<th>Killed</th>
<th>Mutation Score</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>253</td>
<td>238</td>
<td>94%</td>
<td>42s</td>
</tr>
<tr>
<td>1</td>
<td>95</td>
<td>83</td>
<td>87%</td>
<td>23s</td>
</tr>
<tr>
<td>2</td>
<td>95</td>
<td>44</td>
<td>46%</td>
<td>5s</td>
</tr>
</tbody>
</table>

Table 9 illustrating the results for Experiment 3 – a week’s work

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Mutants</th>
<th>Killed</th>
<th>Mutation Score</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>340</td>
<td>243</td>
<td>71%</td>
<td>79s</td>
</tr>
<tr>
<td>1</td>
<td>183</td>
<td>183</td>
<td>100%</td>
<td>37s</td>
</tr>
<tr>
<td>2</td>
<td>183</td>
<td>57</td>
<td>31%</td>
<td>9s</td>
</tr>
</tbody>
</table>
Table 10 illustrating the Mutation Score combined results

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Mutation Score % decrease from Iteration 1 to 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17%</td>
</tr>
<tr>
<td>2</td>
<td>47%</td>
</tr>
<tr>
<td>3</td>
<td>63%</td>
</tr>
<tr>
<td>Mean</td>
<td>42%</td>
</tr>
</tbody>
</table>

Table 11 illustrating the Execution Time combined results

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Execution Time performance gain from Iteration 0 to 1</th>
<th>Execution Time performance gain from Iteration 1 to 2</th>
<th>Execution Time performance gain from Iteration 0 to 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.4x</td>
<td>1.8x</td>
<td>11.6x</td>
</tr>
<tr>
<td>2</td>
<td>1.8x</td>
<td>4.6x</td>
<td>8.4x</td>
</tr>
<tr>
<td>3</td>
<td>2.2x</td>
<td>4.1x</td>
<td>8.8x</td>
</tr>
<tr>
<td>Mean</td>
<td>3.5x</td>
<td>3.5x</td>
<td>9.6x</td>
</tr>
</tbody>
</table>

Table 7, Table 8 and Table 9 show the results obtained in Experiments 1 to 3 respectively.

Equation 3 the percentage decrease in Mutation Score

\[
\text{Percentage decrease in MS} = \text{Percentage before} - \text{Percentage after}
\]

Equation 4 the performance gain

\[
\text{Performance gain} = \frac{\text{Running time before}}{\text{Running time after}}
\]

The results were analysed and a statistical summary of the results can be found in Table 10 and Table 11 which were calculated using Equation 3 and Equation 4 respectively.
5.4 Discussion

5.4.1 Execution Time

![ET against Iteration](image)

Figure 11 illustrating the Execution Time against Iteration

As sought for, in this work, the execution time was reduced over each Iteration. This is clearly illustrated in Figure 11.

5.4.1.1 Localised Mutation

The average performance gain over the use of Localised Mutation was of 3.5x. However as in a realistic agile environment, one commits more frequently, around three to four times a day, the 6.4x performance gain from Localised Mutation is more realistic. The performance gain in Localised Mutation in Experiment 1 was the largest, as expected. This is due to the fact that only a few changes were performed between Iterations 0 and 1, the ratio of mutants generated over the changed code, 30 mutants, to the mutants generated over the whole codebase, 349 mutants, is significant. The other two experiments had lower improvements as expected since the ratio of mutants generated over changes to the mutants generated over the whole codebase was lower at 95:253 for Experiment 2 and 183:340 for Experiment 3.

Unexpectedly, in Experiment 3 however, the performance gain increased from 1.8x in Experiment 2 to 2.2x. This may be due to the fact that the number of mutants generated directly depends on the number of occurrences of each Java operator used,
whose respective mutant operator was implemented. Hence, if the methods altered employed only Java operators whose mutation operator were not implemented, then no mutants would be generated.

5.4.1.2 Localised Testing
The performance gained from Iteration 1 to 2 was on average 3.5x, which was lower than expected. Experiments 2 and 3 which had the largest amount of changes had similar performance gains, at 4.6x and 4.1x respectively. The difference in performance between these experiments was of only 11%.

The performance gain in Experiment 1 was smaller than that of the other experiments as the methods altered had a high number of tests compared to the rest of the methods. Hence the ratio of relevant methods to the complete set of methods was considerably closer to 1:1 i.e. the method had a higher amount of tests relative to other methods. This instance made Localised Testing seem to be less effective as the number of mutants generated was smaller, and as noted earlier, the statistics are more sensitive. However due to the results obtained in the other two experiments which had approximately 8 and 12 times more mutants generated; it can be said that the improvement is consistent.

5.4.1.3 All Optimisations
The overall mean performance gain was of 9.6x. The average execution time of mutation and execution of tests was of 6.3s. As initially thought, the best performance gain was from Experiment 1 as shorter sections of code had to be tested as would the case if such an approach would be used in a realistic agile environment.

5.4.2 Mutation Score
The target for Mutation Score is that it decreases from Iteration 1 to 2 due to the fact that integration tests may be killing some of the mutants in Iteration 1 which would have otherwise passed. Recall, an integration test is a test which does not directly test the method under analysis but is indirectly tested as other methods called by the test depend on the former’s output. All experiments registered a drop in Mutation Score between Iterations 1 to 2, the mean drop being 42%.

In both Experiments 1 and 3, the 100% Mutation Score in Iteration 1 was due to the kills from integration testing. This statistic in Experiment 3 is a little bit unexpected, as 100% mutation score in 183 mutants seems optimistic. Both the source code and tests
were analysed in detail. The vast majority of changes in the source code were in core functionality. Hence, nearly all other operations in the changed methods relied on these changes. Thus tests revealed that through integration tests, indirectly, the core functionality was nearly always being tested. This is due to the fact that in the test classes’ setUp method, the modified core methods were always called and during these initialisation steps, the mutant would be killed. With high probability, if a core functionality method is flawed and its output is required in higher level operations, the result would be based on incorrect inputs, leading to an incorrect result.

5.5 Comparison with other work

<table>
<thead>
<tr>
<th>Performance gain in comparison with other work</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Localised Mutation in realistic agile environments</td>
<td>6.4</td>
</tr>
<tr>
<td>Localised Mutation</td>
<td>3.5</td>
</tr>
<tr>
<td>Localised Mutation and Localised Testing in realistic agile environments</td>
<td>11.6</td>
</tr>
<tr>
<td>Localised Mutation and Localised Testing</td>
<td>9.6</td>
</tr>
<tr>
<td>Selective Mutation [4]</td>
<td>4.2</td>
</tr>
<tr>
<td>Mutant Schemata [2]</td>
<td>4.1</td>
</tr>
</tbody>
</table>

Figure 12 illustrating the performance gain in comparison with other work

In order to gauge the potential impact of this work, results were compared to those of similar research exercises in the literature. The two areas of work whose targets are most aligned to the work presented in this dissertation are Mutant Schemata [2] and Selective Mutation [4]. Other works mentioned in the Background were excluded from the comparison exercise due to the fact that they focused more on Mutation Score rather than Execution Time. Apart from creating new approaches to attempt to make Mutation Testing feasible in agile environments, the performance gain of Localised

3 The Java convention for a test class’ initialisation [16]
Mutation in a realistic agile environment was superior to the other surveyed literature. Results of the comparison exercise are summarised in Figure 12.

5.6 Conclusions

The results obtained both in terms of Mutation Score and Execution Time were positive.

Canfora and Cimitile [17] state that as usually the maintainer is rarely the code author, the time spent on comprehending someone else’s code is 50%-90% the time required to fix a bug. Arcuri [18] states that 50% of the cost of development is spent on testing. Also, he states that the money and effort spent on adequate testing is worth it, as in the USA alone, around $20 billion dollars could be saved if better testing was performed before the release of software. The cost of fixing a bug during testing is at least 3x more expensive than fixing the issue during development. Additionally, it is at least 15x more expensive to fix a bug after release [19]. Hence, it would be ideal to fix a bug at the development stage rather than in the later stages of the software development life cycle.

Ultimately, it is optimal if Mutation Testing is performed multiple times until Mutation Score is satisfactory before committing code so as to ensure the code developed is thoroughly tested. If this procedure is followed, over time, the time taken to perform Mutation Testing will decrease and also adding additional tests to increase the Mutation Score as well as fixing the bugs themselves will be performed faster. Hence the likelihood that code would still contain bugs would decrease and in turn, so will the cost of development.
6 Future work

The following are some considerations how this approach could be made more efficient.

6.1 Mutation

6.1.1 Further optimisations on Localised Mutation

The changes on the code should be identified to an operation level not method level. That is to say that if a line of code contains a number of operations such as assignment and subtraction, and only one of them is changed, only that operation would be mutated. If so, the number of mutants generated would be considerably smaller.

6.1.2 Mutation in General

The mutated class should not be stored and executed on secondary storage. Instead, the mutant should be read directly by the Class Loader from memory. This approach could save time wasted performing I/O operations.

More mutation operators particularly Object Oriented operators should be implemented. Kim et al [23] described that traditional mutation operators do not test for faults which may be introduced through Object Oriented operations, hence not thoroughly testing the Object Oriented aspect.

6.2 Further optimisations in Localised Testing

Detection of relevant tests should be performed in a cleaner way ideally using an efficient call graph. A linking from source methods to test methods should be created and tests should be listened for changes so as to eliminate the detection of the relevant tests if no changes are performed.

6.3 Others

The tool should have a feature to re-run the mutants once additional tests have been implemented so as to ensure the latest tests developed are thorough enough.

The encapsulation of the tool in an IDE plugin would make the use of the tool more accessible and would also make integration between the tool’s reporting and source code seamless.
7 Conclusions

In this work, progress was made towards making Mutation Testing feasible for agile environments. The research question, “Can Mutation Testing be made efficient to be practical for everyday use, particularly for agile environments?” was answered by building a tool which reduced the running time considerably.

Through researching previous work, it was found that there were various attempts in the past to make Mutation Testing feasible. Approaches were based on 3 basic principles, “do fewer”, “do faster” and “do smarter” [7]. Approaches included were on how Mutation Testing could be sped up by using faster mutation techniques and ways to reduce the number of mutants. Some of the approaches in literature were applied. The main techniques include direct Bytecode manipulation and Localised Testing.

However, it was felt that one important characteristic of modern software development was ignored in optimisation approaches; the iterative nature of modern development [19] which is particularly popular in agile environments. Hence, the need to introduce the concept of Localised Mutation was felt. This approach exploits the fact that code is written in cycles from a code repository. Hence, mutants were only generated on changes or additions to the repository not on the whole codebase.

This lead to the design of a tool which employed the above optimisation techniques. The tool is split up in various modules. These are, IO module which performs listening to detect changes in source (used for Localised Mutation), a Mutation module which performs mutation on complete methods, Test Parsing used to detect relevant test methods (used for Localised Testing) and Test Runner used to execute tests.

During the development phase of the tool, various challenges were encountered. Challenges include the handling of misbehaving mutants, detecting relevant tests, detecting changes in code and concurrent test execution. With the exception of concurrent test execution, these issues were all overcome.

The tool was then evaluated on Mutation Score and Execution Time. The number of mutants generated was reduced considerably. Also the Mutation Score decreased as expected. This is because executing all tests resulted in a number of mutants being killed by integration tests. When Localised Testing was put in the picture, the Execution Time of Mutation Testing plummeted.
With the addition of Localised Mutation to the vast choice of optimisations, it is believed that Mutation Testing is now efficient enough to be feasible in terms of Execution Time. Mutation Testing is now feasible to also reduce the overall cost of development, as enforcing developers to create suitable levels of thorough tests, help detect bugs at the development stage, which in turn make the bugs much cheaper to fix.
8 Appendix

The source code together with a readme file can be located on the CD provided.
9 References


[38] “Certification Authorities Software Team”, Position Paper, 2002
