Evaluating the Quality of Student-Written Software Tests
with Curated Mutation Analysis

by

Braxton Hall

B. A., University of British Columbia, 2020

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

October 2022

© Braxton Hall, 2022
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Evaluating the Quality of Student-Written Software Tests with Curated Mutation Analysis**

submitted by **Braxton Hall** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Science**.

**Examinining Committee:**

Elisa Baniassad, Associate Professor of Teaching, Computer Science, UBC  
*Co-supervisor*

Reid Holmes, Associate Professor, Computer Science, UBC  
*Co-supervisor*

Meghan Allen, Associate Professor of Teaching, Computer Science, UBC  
*Supervisory Committee Member*

Ivan Beschastnikh, Associate Professor, Computer Science, UBC  
*Supervisory Committee Member*
Abstract

An important learning outcome in software engineering education is the ability to write an effective test suite that rigorously tests a target application. The standard approach for assessing test suites is to check coverage which can be problematic because coverage rewards code invocation, without checking test assertion correctness.

Mutation Analysis (injecting a small fault into a clone of a codebase) has been used in both industry and academia to check test suite quality. A mutant is killed if any tests in the test suite fail on the clone. More mutants killed indicates a stronger suite, as it is more sensitive to defects. Mutation Analysis has been limited in an educational setting because of the prohibitive cost in both time and compute power to run the students’ suites over all generated clones.

We employed Mutation Analysis to assess test suite quality in our upper-year Software Engineering course at a large research intensive university. This paper makes two contributions: (1) We show that it is feasible and effective to use a small sample of hand-written mutants for grading, and (2) We assess effectiveness for promoting student learning by comparing students graded with coverage to those graded with Mutation Analysis.

We found that mutation graded students write more correct tests, check more of the behaviour of invoked code, and more actively seek to understand the project specification.
Lay Summary

In a typical software engineering course, students write software tests. The quality of the student’s software tests can be difficult to measure. Mutation Analysis is one way of measuring the quality of software tests, however it has seen little use in educational contexts. This work aims to evaluate Mutation Analysis as a method of measuring the quality of software tests written by students. We show that it is feasible to employ Mutation Analysis using hand-written defects, and that Mutation Analysis is a strong indicator of student achievement of learning outcomes in testing.
Preface

All of the work presented henceforth was conducted in the Software Practices Laboratory at the University of British Columbia, Point Grey campus. All projects and associated methods fall under the category of course Quality Assurance as defined by the University of British Columbia’s Research Ethics Board.¹

I was the lead author of this work, responsible for all major areas of concept formation, data collection and analysis, as well as manuscript composition. This material is the result of ongoing research at the Software Practices Laboratory. Elisa Baniassad was the supervisory author on this project and was involved throughout the project in concept formation and manuscript composition. The material has not been published prior to this thesis.

¹https://ethics.research.ubc.ca/behavioural-research-ethics/breb-guidance-notes
# Table of Contents

Abstract ................................................................. iii  
Lay Summary ............................................................. iv  
Preface ................................................................. v  
Table of Contents ..................................................... vi  
List of Figures ............................................................ x  
Acknowledgments ....................................................... xi  
1 Introduction ............................................................ 1  
2 Related Work ........................................................... 4  
   2.1 Mutation Analysis as grading feedback ................. 4  
   2.2 The effect of Mutation Analysis on student learning ... 6  
3 Research Questions .................................................... 7  
   3.1 Research Context .................................................. 7  
   3.2 LO[1-4]: Learning to write an effective test suite ....... 8  
   3.3 LO5: Learning to carefully read and interpret a specification to then refine it into a test suite ...... 9  
   3.4 Further Exploration ............................................. 9
4 Implementation ...................................................... 10
  4.1 Coverage Grader ............................................. 10
  4.2 Mutation Grader ............................................. 11
    4.2.1 Reference Solution .................................... 11
    4.2.2 Handwritten mutants ................................... 11
    4.2.3 Receiving Feedback .................................... 12

5 Evaluation ........................................................... 14
  5.1 Cohorts .......................................................... 14
  5.2 The Mutation Oracle .......................................... 15
  5.3 RQ1: Is our Mutation Grader an accurate indicator in assessing
         student test quality? .................................. 15
  5.4 RQ2: Does student test quality improve under Mutation Analysis?
         ............................................................. 15
    5.4.1 Fault Finding Ability .................................. 16
    5.4.2 Asserting on invoked code ............................ 17
    5.4.3 Over Asserting ........................................ 18
    5.4.4 Under Asserting ...................................... 18
    5.4.5 Summary ................................................. 19
  5.5 RQ3: Do students strive to understand software specification better
         under Mutation Analysis? .............................. 19
    5.5.1 Summary ................................................. 20
  5.6 RQ4: Does the velocity of student implementation improve after
         Mutation Analysis? ......................................... 21
  5.7 RQ5: What do student tests look like in practice under Mutation
         Analysis? .................................................... 22
    5.7.1 Evaluation Infrastructure .............................. 22
    5.7.2 Test Density ............................................. 22
    5.7.3 Assertion Density ....................................... 23
    5.7.4 Assertion Complexity .................................. 23
    5.7.5 Assertion Categories .................................. 25
    5.7.6 Summary ................................................. 26

6 Discussion .......................................................... 27
List of Tables

Table 3.1 Learning outcomes evaluated by RQ2, and their respective means of evaluation. .............................. 8
Table 5.1 Student posts to the forum .............................................. 19
Table 5.2 Grader scores at different points in development ............... 21
Table 5.3 Proportion of student test cases containing 0, 1 or more assertions in Coverage Term and Mutant Term .................. 24
Table 5.4 Categories of assertions encountered in student tests. Categorized from 9,846 assertions from the Coverage Term, and 13,176 assertions from the Mutant Term. An assertion may appear in multiple categories, and categories with less than 1% representation in both terms have been excluded. ........... 25
List of Figures

Figure 4.1 An example of automated feedback from the Mutation Grader.
Section refers to a portion of the specification, Kills is the number of mutants killed. Mutants may appear in more than one section. ............................... 13

Figure 5.1 Distribution of Mutation Oracle scores achieved by test suites from both the Coverage and Mutant terms, normalized for class size. .................................................. 16

Figure 5.2 Student score versus Oracle Score ............................... 17

Figure 5.3 Tests per test suite across all students ............................... 23

Figure 5.4 Distribution of the number of keywords in assertions in both the Coverage Term and Mutant Term. ............................... 24
Acknowledgments

I’d like to thank the following people, who supported me most during the last two years

1. the homies in my phone: Halina and Jin
2. the homies in my computer, recc centre and bob’s server
3. the homies in splablablablablablablab (best lab)
Chapter 1

Introduction

Software testing is a core concept in a standard undergraduate Software Engineering curriculum. Five important learning outcomes in testing are:

- **LO1**: Learning to write an effective test suite that finds faults in a target application.
- **LO2**: Learning to check the behaviour of all code that is invoked.
- **LO3**: Learning to write tests that do not over assert behaviour.
- **LO4**: Learning to write tests that do not under assert behaviour.
- **LO5**: Learning to carefully read and interpret a specification to then refine it into a test suite.

The standard approach for assessing student test suite quality is to check coverage metrics, such as line coverage, or branch coverage. In our large Software Engineering course, we initially used the standard coverage grading approach in the Test Driven Design part of our course project.

However, coverage scores have been shown to be poor indicators of a test suite’s fault finding ability [4]: they can be artificially inflated by rewarding invoking code without correctly validating the invoked code behaved as required. This makes coverage scores poor indicators in a student’s achievement of the learning outcomes, and our students knew it:
Does a test have to be correct to increase coverage? Or does it only matter that the necessary conditions were triggered to increase code coverage?

– Student on the class forum

Additionally, coverage scores are highly implementation dependent; overly verbose components under test increase the coverage score, and in turn inflate a student’s grade. This makes them unreliable at reporting how much of a specification has been exhausted, which would be an invaluable indicator in a student’s achievement of LO5.

In industry, Mutation Analysis has been used to check test suite quality and the literature suggests that it is a strong indicator of test suite effectiveness [7]. In Mutation Analysis, a clone of a program is made, with one small error introduced [3]. The error, or mutant, is usually a small typo, such as changing a “<” to a “<=” in a binary expression. The test suite is then run on the mutated version of the program, and if any of the tests in the suite fail because of the mutant, then the mutant has been killed. In an industrial setting, thousands of clones are made with faults injected into them, with the test suite ran once for each mutant independently. Running the test suite against all the clones is time and computationally intensive. Its use in an educational setting has been limited because of the barrier of the effort required for its implementation (in creating and mutating the clones) and deployment (affording the time and compute power to run the suites over a meaningful number of mutated applications).

The thesis of this work that a small number of hand-written mutants, against which students’ Testing submissions are graded in place of coverage metrics, is a feasible implementation of Mutation Analysis, and positively affects student learning outcomes.

We validate this thesis with the introduction of the Mutation Grader, composed of hand-written mutants, and through the investigation of the following research questions:

- **RQ1**: Is our Mutation Grader an accurate indicator in assessing student test quality?

- **RQ2**: Does student test quality improve under Mutation Analysis?
• **RQ3:** Do students strive to understand software specification better under Mutation Analysis?

• **RQ4:** Does the velocity of student implementation improve after Mutation Analysis?

• **RQ5:** What do student tests look like in practice under Mutation Analysis?

We compared the coverage-graded cohorts and mutation-graded cohorts to determine whether the presence of Mutation Analysis resulted in a change in students’ software quality, test quality, or practice during their Testing submission development.

We find promising results, demonstrating that students write more correct tests, check more of the behaviour of invoked code, and more actively seek to understand the project specification.

This paper makes two contributions: (1) We show that by using a small sample of hand-written mutants it is feasible to employ Mutation Analysis as an assessment approach in an educational setting, and (2) We report on the results of the comparison success of the cohorts to determine if Mutation Analysis positively affects learning outcomes.
Chapter 2

Related Work

In this chapter, we examine related work on Mutation Analysis in an educational setting. These works largely relate to our Mutation Grader in (1) their interrogation of the feasibility and efficacy of Mutation Analysis as a means of grading feedback, or (2) the effect of Mutation Analysis on student learning.

2.1 Mutation Analysis as grading feedback

Using Mutation Analysis as a means of assessing student software test suites has a long history of proposals beginning in the 1990s [6]. Aaltonen et al., in their assessment of Mutation Analysis as a means of automated assessment, demonstrated that Mutation Analysis provided a stronger capacity for finding deficiencies in student test suites than the traditionally used metric of code coverage [1]. Highlighting weaknesses of Mutation Analysis as a means of assessment, Aaltonen et al. name

- **Incomparability.** Generated mutants are student and implementation specific. Thus scores between students with similar test suites may vary wildly.

- **Gaming.** A student could write pathological implementations to inflate their mutation score.

- **Inaccuracy.** Without manual inspection of every mutant, it is unknown which mutants actually cause a failure, and which are isomorphic to the original implementation. Automated systems estimate this.
Building off the work of Aaltonen et al., Shams and Edwards suggest practical solutions to the shortcomings of Mutation Analysis as a means of automated feedback [10]. (1) Use a reference solution. Using a reference solution ensures that the implementation is complete and correct. The reference solution ensures that student scores are comparable, and prevents gaming. (2) Use the results of student test suites over time to prove that mutants cause a failure. If there is at least one test in all the students’ test suites that kills a mutant, the mutant must not be isomorphic to the original implementation, thus directly proving that the mutant causes a fault.

The approach used by Shams and Edwards is not amenable to live, incremental feedback. As our students automatically receive feedback on submissions throughout their development period, using the results of student test suites over time to weed out isomorphic mutants would result in inaccurate scores for students until the end of the Testing phase. Students are also prone to submit later in the development period, which means that students who start early are the most impacted, as they will not be able to estimate their scores until the remainder of their classmates has made submissions as well [5]. Instead, we finalize the set of mutants before the Test development period, and manually ensure that no mutants are isomorphic. This fixed set of mutants ensures that student scores are consistent throughout the Testing phase.

Edwards and Shams go on to use a variant of all-pairs approach where all student tests (and some instructor tests) are run against all student code, effectively using handwritten student code in place of automatically generated mutants [4]. They go on to show that student test suites largely fail to exhaustively test a specification, test “happy path” scenarios, and fail to uncover real hand-written faults even after achieving high coverage scores. While we also use hand-written faults in our Mutation Grader approach, we elected to use instructor curated faults, as an all-pairs approach is not suitable for incremental feedback.

To overcome the challenge of the computational expense of Mutation Analysis, Kazerouni et al. propose using a subset of mutants when evaluating student tests [8]. They select a subset of kinds of mutations that can be performed on software under test. They show that their selected subset strongly correlates with a complete, computationally expensive Mutation Analysis. Similarly to Kazerouni et al., we utilise a reduced set of mutants to approximate a more thorough Mutation Anal-
ysis for the sake of computing resources at our institution. Promisingly, we find similarly strong correlations between reduced and complete analyses. However, Kazerouni et al. only attempt their analysis on corpora of software tests and have yet to deploy their Mutation Analysis in a classroom.

2.2 The effect of Mutation Analysis on student learning

Mutation Analysis as an exercise has been used to augment student learning, both in their understanding of program implementation and software testing practices.

Oliveira et al. introduce Pascal Mutants, a program for interactively exploring mutants from Pascal source code [9]. When introduced to Pascal Mutants, undergraduate students tasked with understanding undocumented code were shown to demonstrate a more intimate understanding of the code than their counterparts who did not have access to Pascal Mutants. Unlike Pascal Mutants, our Mutation Grader does not make source code nor mutants known to students, however our findings complement the work of Oliveira et al., demonstrating that when reading a specification through the lens of Mutation Analysis, students demonstrate evidence of striving to more intimately understand program requirements and behaviour.

Delgado-Pérez et al. analyze the effect of Mutation Analysis on students through their perceptions of their own test suite effectiveness [2]. They show that students perceive their test suites as less effective after using Mutation Analysis to evaluate it. Similarly, we see that student behaviour surrounding their own test suites are affected by exposure to Mutation Analysis. Promisingly, we see students using more assertions to bolster their test suite effectiveness. Delgado-Pérez et al. limit students’ Mutation Analysis exposure to a single test suite scoring event, while we show how students are affected by Mutation Analysis as live feedback.
Chapter 3

Research Questions

In this work we evaluate whether a small number of hand-written mutants, against which students’ test suites are graded, is a feasible implementation of Mutation Analysis, and how Mutation Analysis affects whether students are obtaining the core learning outcomes of the course. We decompose this high level assessment into five research questions, which are motivated in this chapter.

3.1 Research Context

We run a large (~300-person) third-year software engineering class at a large research intensive university.

The course employs a semester-long project in which students are asked to create a small query language to query publicly available enrollment data stored on disk in an archive. The project is released in two major phases:

1. Testing (2 Weeks): A test driven design based assessment, in which students are provided a specification for the language, and their tests are graded against a reference solution.

2. Implementation (10 Weeks): This phase is itself split into three portions, roughly corresponding to a base backend implementation, an augmented backend implementation, and finally a frontend implementation.

The Mutation Grader is employed only during the Testing phase of the project.
Table 3.1: Learning outcomes evaluated by RQ2, and their respective means of evaluation.

<table>
<thead>
<tr>
<th>Learning Outcome</th>
<th>Evaluation Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO1: Learning to write an effective test suite that finds faults in a target application</td>
<td>Mutant scores between terms using a more rigorous Mutation Analysis than the Mutation Grader</td>
</tr>
<tr>
<td>LO2: Learning to check the behaviour of all code that is invoked</td>
<td>The correlation between mutant scores and coverage scores</td>
</tr>
<tr>
<td>LO3: Learning to write tests that do not over assert behaviour</td>
<td>The rate at which tests fail against the reference solution</td>
</tr>
<tr>
<td>LO4: Learning to write tests that do not under assert behaviour</td>
<td>The rate at which tests fail against program stubs</td>
</tr>
</tbody>
</table>

3.2 LO[1-4]: Learning to write an effective test suite

To assess whether the Mutation Grader is an appropriate tool in encouraging and evaluating student achievement in LO[1-4], we focused on validating whether the Mutation Grader accurately assessed student test quality, and whether student test quality improved under Mutation Analysis.

RQ1: Is our Mutation Grader an accurate indicator in assessing student test quality? Since Mutation Analysis is typically run with a large number of mutants (in the thousands), we examined whether our approach of using 16 hand-written mutants was effective for assessing student test suite quality, and in turn in assessing a student’s achievement of LO1: Learning to write an effective test suite that finds faults in a target application. We look at whether mutation-graded scores correlate with the scores obtained from a larger set of mutants produced by an independent instructor of prior offerings of this course.

RQ2: Does student test quality improve under Mutation Analysis? We examine whether test suite quality improves when students are graded using mutation scores. We defined quality as demonstrating achievement in LO[1-4], and compare metrics between the cohort graded by coverage and the cohort graded by Mutation Analysis. The learning outcomes are restated, paired with their respective evalua-
tion metrics in Table 3.1.

3.3 LO5: Learning to carefully read and interpret a specification to then refine it into a test suite

To compare students’ achievements of this learning outcome, we evaluate whether students sought better understanding of the project specification, and the velocity of students’ implementation.

*RQ3: Do students strive to understand software specification better under Mutation Analysis?* Clarifying the requirements of the project specification is evidence that students are closely engaging with it, and in turn evidence that they are working toward LO5. To evaluate how much the students attempt to clarify the project specification, we examine the frequency at which cohorts ask specification clarifying questions on the class forum.

*RQ4: Does the velocity of student implementation improve after Mutation Analysis?* We posit that Mutation Analysis would encourage a deeper understanding of the project specification, leading to faster success at implementation. We look at whether students attain higher implementation scores more quickly when graded by the Implementation Grader.

3.4 Further Exploration

*RQ5: What do student tests look like in practice under Mutation Analysis?* Unrelated to any specific learning outcome, and because to our knowledge there has been no detailed analysis of how test-writing style changes in the face of Mutation Analysis in a pedagogical setting, we performed a comparative examination of the style and textural qualities of student test suites.

We examine the density of student tests within a test suite, and the assertions in those tests. We examine assertions in terms of count per test, complexity, and semantic category.
Chapter 4

Implementation

We have an automated testing framework that runs tests against an implementation. This Autograder is written in TypeScript for Node.js. During the Testing phase of the project, the students’ tests are run against our reference solution (described in Section 4.2.1). During the implementation phase of the project, our tests are run against the student’s implementation. To ensure that the project properly compiles during the grading of either phase, students are provided with an API interface and a list of allowed dependencies to develop with.

We have three configurations for the Autograder:

- **Coverage Grader**: Student tests are run against our reference implementation, and the coverage score is captured.

- **Mutation Grader**: Student tests are run against a set of clones containing handwritten mutants.

- **Implementation Grader**: Student implementations are run against our set of tests.

4.1 Coverage Grader

The Coverage Grader executes students’ tests against the reference solution. A student test suite’s score is calculated using the statement coverage score achieved.
Additionally, student tests are run against an unimplemented project stub. If any student tests pass when run against the unimplemented project, the student’s feedback contains a warning with a list of test names which passed on the stub. The student is advised make the test stronger, however it has no direct impact on their grade.

4.2 Mutation Grader

The Mutation Grader executes students’ tests against the non-mutated reference solution, and then once for each of the mutated clones. A mutant is graded as killed if some test passes on the non-mutated reference solution, and fails for the mutant. A student test suite’s Mutation Grader score is the number of killed mutants over the total number of mutants.

4.2.1 Reference Solution

The reference solution has been thoroughly tested by the instructors and is assumed correct. As suggested by Shams et al., the use of a reference solution ensures that scores from different students’ test suites are comparable, and sidesteps the time cost of mutant generation [10].

An online instance of the reference solution is made available to students in both terms to model correct behaviour when creating test cases. Students interact with the API through a course hosted website called the Reference User Interface.

4.2.2 Handwritten mutants

Mutant generation is commonly achieved by automatically manipulating code at the AST level. As a result, components of software with a larger AST, i.e. more verbose implementation, are susceptible to more mutants, and in turn are “worth” more of the mutant score [1]. Similarly to coverage-based scoring, this makes mutant scores highly implementation dependent. We require a specification dependent score to assess \textbf{LO5}: Learning to carefully read and interpret a specification to then refine it into a test suite.

To achieve specification dependent scores, we elect to use handwritten mutants. This affords us as educators the ability to precisely weight how much spe-
cific requirements and sections of the project specification are “worth” in students’ Mutation Grader scores.

We isolate key project requirements from the project specification that we deem should be tested by a thorough test suite. We then select methods in the reference solution responsible for the correct implementation of these project requirements. The reference solution is supplied with functions which are clones of these existing methods, but with minor changes applied to make them faulty. Each mutant function is associated with a unique ID. At the program’s entrypoint, the process’s environment variables are used to select a mutant function using its ID. The corresponding class method is reassigned with the mutant function before the tests are executed.

The actual implementation of class method overriding was achieved in \( \sim 100 \) lines of code, and has been made publicly available on GitHub.\(^1\)

As our project contains frequent disk accesses, as well as archive decompression, executing a single test suite just once often can take over 100 seconds. With this challenge in mind, the set of mutants is restricted to 16 to ensure the Mutation Grader’s response time is fast enough to provide a response to students as they are developing their solution.

The 16 mutants were validated against our Implementation Grader to ensure that they all induce real faults. We assume that the Implementation Grader, written by course instructors, is correct. We repeat this evaluation with an empty test suite, to ensure that mutants live and are killed appropriately.

### 4.2.3 Receiving Feedback

Students invoke the Mutation Grader by commenting on commits in their GitHub repository. The Mutation Grader clones their repository, scores their tests, and replies in their commit comments with feedback as seen in Figure 4.1. The header titles from the project specification are listed in the feedback, along with counts of how many mutants pertaining to each section of the project specification are still alive. In addition, the Mutation Grader’s feedback includes a list of tests that failed against the reference solution, and students are encouraged to reevaluate the named

---

\(^1\) https://github.com/braxtonhall/nodejs-class-mutator/
**Figure 4.1**: An example of automated feedback from the Mutation Grader. Section refers to a portion of the specification, Kills is the number of mutants killed. Mutants may appear in more than one section.
Chapter 5

Evaluation

This chapter reports on our findings for each of our research questions defined in Chapter 3.

5.1 Cohorts

In our evaluation, we perform our analyses on student submissions from two subsequent offerings of our software engineering course:

- **The Coverage Term** ($n = 290$): offered in Winter 2021, the final semester prior to the introduction of the Mutation Grader

- **The Mutant Term** ($n = 306$): offered in Fall 2021, whose tests were scored using the Mutation Grader

Both semesters were offered online due to the ongoing COVID-19 pandemic, and shared the same project specification.

In the Coverage Term, starter code was provided to students as is common practice for course projects. In the Mutant Term, an additional component was added to the Testing phase, which had students creating their own starter code to serve a newly introduced learning outcome. As such, starter code was withheld from students in the Mutant Term.
5.2 The Mutation Oracle

We assess the quality of student test suites by running the them against an additional set of expert-made mutants, called The Mutation Oracle. The Mutation Oracle has 73 mutants, each designed to cause a unique fault in the project. A test suite’s Mutation Oracle score is the number of mutants killed from the Mutation Oracle over the number of mutants in the Mutation Oracle.

The Mutation Oracle was created by an independent educator from a prior semester of the course, who was intimately familiar with the project specification. They were asked to create a set of important failures worth grading from the project specification. We employ an independent researcher because using the same educator (with the same understanding of the specification) to construct both the Mutation Oracle and Mutation Grader could artificially inflate the correlation between the scores they produce.

5.3 RQ1: Is our Mutation Grader an accurate indicator in assessing student test quality?

To evaluate the effectiveness of our Mutation Grader, we compared the scores of student test suites on our Mutation Grader and Mutation Oracle. Correlating the scores of these test suites finds $R^2 = 0.89$, indicating a strong correlation between the two Mutation Analyses.

We repeat this with coverage scores and Mutation Oracle scores to find a similarly strong $R^2 = 0.81$, however weaker than that achieved by the Mutation Grader, indicating that the Mutation Grader is a more accurate indicator of test suite quality than statement coverage scoring.

5.4 RQ2: Does student test quality improve under Mutation Analysis?

This section presents our evaluation of how the quality of student tests are affected by switching from the Coverage Grader to the Mutation Grader as a means of providing feedback to students on their test suite.
5.4.1 Fault Finding Ability

In both the Coverage Term and the Mutant Term, students struggled to kill all (or even most) of the mutants in the Mutation Oracle. Student test suites from the Coverage Term achieved an average score of 63.46%, with a standard deviation of 15.88%. Student test suites from the Mutant Term achieved an average score of 54.08%, with a standard deviation of 20.37%.

Figure 5.1 displays the distributions of the respective terms’ Mutation Oracle scores. The Mutant Term’s distribution peaks considerably lower than the peak of the Coverage Term. This may be a result of the Mutant Term’s grading scheme, which allowed for some mutants to remain alive for a student to achieve the maximum score of 100%. This grading scheme was enacted as the difficulty of the new Mutation Grader was unknown, however it likely introduced a capping-influence, resulting in students stopping the development of their test suites earlier. This capping would potentially explain the somewhat lower Mutation Oracle scores for the Mutation Term cohort.
5.4.2 Asserting on invoked code

We evaluate the correlation between students’ Mutation Oracle scores and coverage scores; a higher correlation may indicate that all code being invoked is asserted upon.

Coverage Term test suites showed a high correlation between coverage scores and Mutation Oracle scores, with $R^2 = 0.8$. Mutant Term test suites showed a similar correlation, with $R^2 = 0.81$.

As students in the Coverage Term were provided with starter code, weaker submissions are all elevated to the same coverage score, artificially boosting the correlation between coverage the fault finding ability. To compare without these weaker submissions, we restrict our analysis to test suites that achieved an Oracle Score above 63% (the average Mutation Oracle Score of the Coverage Term). We now see $R^2$ from the Coverage Term drops to 0.36, while the Mutant Term’s $R^2$ falls to only 0.69.

Figure 5.2 shows the relationship between Mutation Oracle score and coverage scores. Points in the lower right indicate test suites that invoked code without correctly asserting over its behaviour. As we see more of these points in the Coverage Term, this is evidence that students in the Mutant Term were more likely to correctly assert over code that they invoked.
5.4.3 Over Asserting

The Mutation Grader begins by executing students’ tests against an unmodified reference solution. This baseline execution gives us a view of the correctness of student assertions. Tests that fail against a correct solution are over asserting. To over assert is to make assertions that are too strong and assert behaviour outside of the project specification, or make assertions that are entirely incorrect. A test suite’s Over Asserting Score is calculated as the number of over asserting tests over the total number of tests. A lower Over Asserting Score is more desireable.

Student test suites executed against a correct project implementation in the Coverage Term achieved an average Over Asserting Score of 15.37%, with a standard deviation of 11.3%. In the following Mutant Term, student test suites achieved an Over Asserting Score of 7.42%, with a standard deviation of 13.22%.

This finding is unsurprising, as students in the Mutant Term received a list of failing tests in their Autograder feedback. Students in the Coverage Term did not have this feedback, yet were still able to derive a list of failing tests, as they had access to a model of correct behaviour through the Reference User Interface (described in Section 4.2.1). However, students are more likely to be aware of incorrect assertions if they are informed of mistakes explicitly.

5.4.4 Under Asserting

Student tests were run against a stub implementation of the project. This allows us to see at what rate tests always pass. These ineffective tests are under asserting, as they fail to even identify that the project has not yet been implemented, and require stronger assertions. A test suite’s Under Asserting Score is calculated as the number of under asserting tests over the total number of tests. A lower Under Asserting Score is more desirable.

Test suites from the Coverage Term had a low average Under Asserting Score of 2.07%, with a standard deviation of 4.05%. The Mutant Term’s test suites had a higher Under Asserting Score of 5.22% with a standard deviation of 10%.

We expect this finding, as in the Coverage Term, a list of the test suite’s under asserting tests were directly reported to students in the Coverage Grader’s automated feedback, encouraging them to make the test fail on a stub.
### Table 5.1: Student posts to the forum

<table>
<thead>
<tr>
<th>Category</th>
<th>Coverage Term</th>
<th>Mutant Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification Posts</td>
<td>53</td>
<td>97</td>
</tr>
<tr>
<td>Testing Posts</td>
<td>423</td>
<td>813</td>
</tr>
<tr>
<td>Implementation Posts</td>
<td>1,028</td>
<td>1,238</td>
</tr>
<tr>
<td>Students</td>
<td>290</td>
<td>306</td>
</tr>
</tbody>
</table>

#### 5.4.5 Summary

When comparing the test suites created by the Coverage Term and the Mutant Term, the Mutant Term demonstrated a lower rate of over asserting tests. However the Coverage Term was slightly better in rate of under asserting tests, and fault finding ability, however we believe that this can be attributed to the capping-influence of the bonuses in the Mutant Term. When we look more closely at the stronger test suites in the Mutant Term (those of students who persisted beyond the effect of any bonuses) we saw a stronger correlation between Mutation Oracle scores and coverage scores, indicating that they were correctly asserting over more of the code they invoked.

#### 5.5 RQ3: Do students strive to understand software specification better under Mutation Analysis?

We assess the effort that students put into understanding of the project specification by measuring how many specification clarifying questions students ask on the shared forum. It is reasonable to compare forum questions about the specification since the specification was the same across these semesters. We conduct an investigation into the frequency at which students ask specification clarifying questions on the online forum over the Testing phase. We examine each post on the class forum from the beginning of the semester until the deadline of the Testing phase of the project. If a post asks for more information on the functional requirements of the Implementation phase of the project, it is counted as a specification clarifying question.

A potential threat to our investigation in the frequency at which students ask
specification clarifying questions is the varying depths into the COVID-19 pandemic when the respective semesters were offered. As students in the Mutant Term had more prerequisite courses in online classes due to the COVID-19 pandemic, it is possible that the students in the Mutant Term were merely more ready to post on the forum, instead of as a result of a change in the Testing phase of the project.

To assess students’ baseline post frequency, we count how many posts were made between the beginning of the Implementation phase of the project and the end of the semester.

During the Implementation phase of the project, students in the Coverage Term made 1,028 posts, or an average of 3.54 posts per student, and students in the Mutant Term made 1,238 posts, or an average of 4.05 posts per student. As the post count between terms are similar, this indicates that any significant change in post count during the Testing phase of the project is likely due to the change from the Coverage Grader to the Mutation Grader.

As seen in Table 5.1, the students in the Mutant Term were more active on the forum during the Testing phase of the project. The Coverage Term submitted an average of 0.18 specification clarifying questions per student, and an average of 1.46 posts per student. The Mutant Term submitted an average of 0.32 specification clarifying questions per student, and an average of 2.66 posts per student.

Despite the count of students and count of Implementation phase posts remaining steady, we see a stark jump in the count of specification clarifying posts, and Testing phase posts overall. This indicates to us that students in the Mutant Term were more active in seeking to understand project specification and testing requirements.

5.5.1 Summary

During the Testing phase, Students in the Mutant Term made more posts on the student forum, and asked more specification clarifying questions than the Coverage Term, despite having a similar number of posts in the remainder of their respective terms.
Table 5.2: Grader scores at different points in development

<table>
<thead>
<tr>
<th>Average Score</th>
<th>Coverage Term</th>
<th>Mutant Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commits until score &gt; 0</td>
<td>9.08</td>
<td>8.68</td>
</tr>
<tr>
<td>Commits until score ≥ 50</td>
<td>28.78</td>
<td>25.73</td>
</tr>
<tr>
<td>Score after 5 commits</td>
<td>30.26%</td>
<td>33.38%</td>
</tr>
<tr>
<td>Rate of helpful commits</td>
<td>15.00%</td>
<td>16.12%</td>
</tr>
</tbody>
</table>

5.6 RQ4: Does the velocity of student implementation improve after Mutation Analysis?

We evaluate whether Mutation Analysis during the Testing phase of the project positively influences the velocity of students in the subsequent Implementation phase. We define velocity as the rate at which a students’ grade improves.

We measure a student implementation’s strength through the use of an instructor written test suite, called the Implementation Grader. An implementation is scored as the number of Implementation Grader tests that pass out of the total number of Implementation Grader tests. We use the Implementation Grader to evaluate student software every commit during the Implementation phase of the students’ project.

We examine the following metrics to understand how students developed during the Implementation phase of the project:

- The number of commits until receiving an Implementation Grader score above 0.

- The number of commits until receiving an Implementation Grader score equal to or above 50.

- The Implementation Grader score after five commits.

- The rate of commits that cause a positive change in score in the first five commits.

We use commits in our analysis instead of time ranges to get comparable submissions, as students in the Coverage Term had two intermediate deadlines during
the Implementation phase that the students in the Mutant Term were free from. We restrict our analysis to the first five commits as we are evaluating students’ understanding of the project specification during and immediately after the Testing phase; students will continue to develop a more intimate understanding of the project specification as development continues.

As described in Table 5.2, across all four metrics, we see little change in the performance of students during the Implementation phase between the Coverage Term and the Mutant Term. The addition of Mutation Analysis did not demonstrate obvious benefit to student implementation velocity.

5.7 RQ5: What do student tests look like in practice under Mutation Analysis?

5.7.1 Evaluation Infrastructure

Student tests are written in TypeScript, supported by the Chai assertion library,1 and the Mocha test framework.2 Using the TypeScript parser,3 we created a static-analysis tool to characterize both the tests and assertions present in student test suites.

5.7.2 Test Density

Figure 5.3 shows that students in the Mutant Term wrote fewer tests overall. Students in the Coverage Term wrote an average of $113.74$ tests with a standard deviation of $87.69$. Students in the Mutant Term wrote an average of $53.11$ tests with a standard deviation of $36$.

To account for the possibility that students in the Mutant Term had weaker acceptance criteria, and thus simply stopped writing unit tests earlier, we compare the test density of suites with a similar effectiveness. We can restrict our analysis to only test suites that achieved a Mutation Oracle score above $63\%$, which is the average Mutation Score achieved by the Coverage Term, as described in Section 5.4.

---

1https://www.chaijs.com/
2https://mochajs.org/
3https://www.typescriptlang.org/
Figure 5.3: Tests per test suite across all students

When restricted to this stronger set of test suites, we see the average size of Coverage Term test suites increase to 130.23 tests with a standard deviation of 91.73. The respective Mutant Term test suites had an average size of 77.98 tests with a standard deviation of 38.59. Students in the Mutant Term used fewer tests to achieve similar rates of effectiveness.

5.7.3 Assertion Density

Students in both terms wrote largely simple tests with few assertions. Students in the Coverage Term wrote an average of 1.05 assertions per test, while students in the Mutant Term wrote an average of 1.63 assertions per test.

Students in the Mutant Term were more likely to have more assertions in each test. As seen in Table 5.3, 37.28% of student tests in the Mutant Term contained two or more assertions, compared to the 7.88% of student tests in the Coverage Term.

5.7.4 Assertion Complexity

To understand how the shift to the Mutation Grader may have affected the verbosity of the assertions written by students, we evaluate the complexity of assertions created by the Coverage Term and the Mutant Term. The Chai assertion library allows users to chain keywords to build complex requirements in a single assertion. This
Table 5.3: Proportion of student test cases containing 0, 1 or more assertions in Coverage Term and Mutant Term

<table>
<thead>
<tr>
<th>Student tests with</th>
<th>Coverage Term</th>
<th>Mutant Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 assertions</td>
<td>7.08%</td>
<td>4.87%</td>
</tr>
<tr>
<td>1 assertion</td>
<td>85.04%</td>
<td>57.84%</td>
</tr>
<tr>
<td>Multiple assertions</td>
<td>7.88%</td>
<td>37.28%</td>
</tr>
</tbody>
</table>

Figure 5.4: Distribution of the number of keywords in assertions in both the Coverage Term and Mutant Term.

allows us to count the number of keywords in an assertion to approximate its complexity.

As seen in Figure 5.4, students tend to write assertions of similar complexity regardless of term, however students in the Coverage Term were more likely to write assertions with more than 4 keywords. We believe this may be a result of the starter code provided to the students in the Coverage Term, which was withheld from the students in the Mutant Term. The starter code included sample assertions using the chai-as-promised plugin library\(^4\) for asynchronous assertions. While not necessary, the library can be helpful for students as all API methods in their project are asynchronous. As seen in Listing 5.1, the style encouraged by the plugin encourages the use of more keywords per assertion.

\(^4\)https://www.chaijs.com/chai-as-promised/
Table 5.4: Categories of assertions encountered in student tests. Categorized from 9,846 assertions from the Coverage Term, and 13,176 assertions from the Mutant Term. An assertion may appear in multiple categories, and categories with less than 1% representation in both terms have been excluded.

<table>
<thead>
<tr>
<th>Category</th>
<th>Coverage</th>
<th>Mutation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>51%</td>
<td>43%</td>
<td>Objects are of a class or primitive type</td>
</tr>
<tr>
<td>Error</td>
<td>48%</td>
<td>19%</td>
<td>Method throws or rejects</td>
</tr>
<tr>
<td>Equality</td>
<td>39%</td>
<td>24%</td>
<td>Expecting exact values</td>
</tr>
<tr>
<td>Failure</td>
<td>8%</td>
<td>13%</td>
<td>Automatic test failure</td>
</tr>
<tr>
<td>Length</td>
<td>1%</td>
<td>11%</td>
<td>Array or string length</td>
</tr>
<tr>
<td>Inclusion</td>
<td>1%</td>
<td>6%</td>
<td>Element presence in a string or array</td>
</tr>
<tr>
<td>Existence</td>
<td>0%</td>
<td>2%</td>
<td>Value is null or undefined</td>
</tr>
</tbody>
</table>

Listing 5.1: An example test case written with and without the chai-as-promised plugin library, demonstrating how the library may encourage more verbose assertions

```
// Common practice in the Coverage Term
const futureResult = api.add(foo);
return expect(futureResult).to.eventually.deep.equal(bar); // 5 Keywords

// Common practice in the Mutant Term
const result = await api.add(foo);
expect(result).to.deep.equal(bar); // 4 Keywords
```

5.7.5 Assertion Categories

To understand the kinds of assertions that students were writing before and after the introduction of the Mutation Grader, we categorize all 9,846 Coverage Term assertions and all 13,176 Mutant Term assertions. Each keyword in each assertion is categorized based on the semantic modifications it induces upon the assertion. Each assertion belongs to all the categories represented by its keywords. We use the semantic assertion categories defined by Zamprogno, and listed in Table 5.4 [11].

As seen in Table 5.4, the three most represented categories in the Coverage Term remained the three most represented in the Mutant Term, however the Mutant
Term also sees a more even distribution of category representation.

While students regardless of term largely use the same kinds of assertions, under Mutation Testing, a wider variety of semantic assertion categories were used by the cohort. This may be indicative of students’ increased comprehension of the project specification, as they were using more precise assertions, though further study is required.

5.7.6 Summary

Under Mutation Analysis, students write fewer tests, even when comparing test suites with similar fault finding ability. Students receiving Mutation Analysis feedback also write more assertions per test, using a wider variety of assertion semantics. Assertions in both terms are of a similar, low complexity.
Chapter 6

Discussion

In this chapter, we discuss our findings in terms of our stated learning outcomes.

**LO1**: Learning to write an effective test suite that finds faults in a target application

Our analysis of RQ1 in Section 5.3 indicates that our Mutation Grader is a strong indicator of test suite effectiveness, and in turn an indicator in a student’s achievement of **LO1**. This means that we can confidently move away from coverage as a primary method of scoring student test suites, as it is known to be a problematic metric [1] [4] [7].

**LO2**: Learning to check the behaviour of all code that is invoked

Within test suites with above average fault finding ability, students in the Mutant Term were more likely to have a high correlation between fault finding ability and code coverage score. This indicates that in this range of student achievement, students graded by the Mutation Grader were more likely to assert over invoked code.

While not direct evidence of test suite effectiveness, it is promising that results of our analysis in RQ5 complement this finding, as students in the Mutant Term were generally more likely to have assertions in their tests, and more assertions per test.

**LO[3-4]**: Learning to write tests that do not over assert or under assert behaviour

Our results showed a change in both the rate that students under assert and the rate
that they over assert after the move to the Mutation Grader. However it is likely that
this is a mere result in the change in feedback, not the change in scoring. Under
asserting tests were reported to the term that had fewer under asserting tests (the
Coverage Term), and over asserting tests were reported to students in the term that
had fewer over asserting tests (the Mutant Term).

Under asserting tests in grading feedback were withheld from the Mutation
Grader as the computational cost of the Mutation Grader was high and the report
deemed redundant. The students could easily run their tests on their own machine
against their own stub to generate the same results. This secondary finding supports
our understanding that students often forgo evaluating their software on their own
machines when automated grading feedback is available.

**LO5:** Learning to carefully read and interpret a specification to then refine it into
a test suite  During the Mutant Term, we anecdotally witnessed a stronger en-
gagement and understanding of the project specification in the earlier weeks of
the semester. Despite the Reference User Interface being publicly available (Sec-
tion 4.2.1), and it remaining, along with the project specification, unchanged for
several semesters, the students in the Mutant Term successfully identified four mi-
nor bugs in the reference implementation!

It is promising that our findings in the increased rate of specification clarifying
questions on the student forum support our intuition that students more attentively
studied the project specification, to afford future refinement into implementation,
as a result of the Mutation Grader.

### 6.1 Threats to validity

In this section, we discuss the threats to internal validity, construct validity, and
external validity.

**Internal validity**  The students in the Coverage Term were more likely to have
taken prerequisite software engineering courses before the COVID-19 pandemic
and thus before the change to online instruction. This change in instruction may
have affected the cohort’s foundational skills in both testing and software develop-
ment.

The Mutant Term were assigned an additional component during their Testing phase, described in Section 5.1, which burdens the time that they can commit to working on the project, and potentially confounds our analysis.

Construct validity The fault finding ability of a test suite is difficult to quantify, and we used The Mutation Oracle as our indicator. The Mutation Oracle may have been a poor representation of the real faults that can occur in the project, making it a weak indicator of the strength of a students’ software suite, and in turn a weak indicator of the predictive strength of the Mutation Grader. We analyzed, but did not edit, the Mutation Oracle, and we believe that it is correct, and proportionally representative of our project specification, though in its nature it is incomplete.

External validity The generalizability of our results on the feasibility of implementing mutation analysis in a pedagogical setting is limited based on the specificity of how it was applied within our pedagogical context. However the finding that a small hand-curated number of mutants can act as a representative and tractable proxy for a larger suite can apply directly to most educational settings in which Mutation Analysis is being considered.

Our results on student response to Mutation Analysis are limited in generalizability by the specificity of the student experience within our course setting.
Chapter 7

Conclusion

Code coverage is widely used to evaluate student test suites despite its well documented shortcomings in estimating test suite effectiveness [7]. Mutation Analysis has been often proposed as an alternative, yet educators have faced barriers to adoption of Mutation Analysis because of the computational demands.

In this paper, we have presented our experience using Mutation Analysis with a small set of handwritten mutants as a means of providing live, incremental Autograder feedback. Results did not show that the fault finding abilities of the test suites written by the Mutant Term were stronger. However, our data suggests that the Mutation Grader is a strong indicator of fault finding ability. We also saw that students graded by the Mutation Grader wrote more assertions, wrote more varied assertions, and made a stronger attempt to understand the project specification.
Bibliography


