Towards Improving Students' Software Testing Practices using Modified Mutation Testing

Rifat Sabbir Mansur

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science and Application

Clifford A. Shaffer, Chair

Stephen H. Edwards, Co-chair

Eli Tilevich

Francisco Servant

Thomas W. Price

February 13, 2025

Blacksburg, Virginia

Keywords: CS education, post-CS2, data structures and algorithms, software testing, mutation testing, software engineering education, automated assessment tool

Copyright 2025, Rifat Sabbir Mansur

Towards Improving Students' Software Testing Practices using Modified Mutation Testing

Rifat Sabbir Mansur

(ABSTRACT)

Mutation testing (MT) is a powerful technique for evaluating the quality of software test suites by introducing small faults, called "mutations," into code to assess if tests can detect them. While MT has been extensively applied in the software industry, its use in programming courses faces both computational and pedagogical barriers. My research investigates the successful integration of MT in a post-CS2 Data Structures and Algorithms (DSA) course with 3-4 week long programming projects. Through a comprehensive study across multiple semesters, I investigated three key aspects: the computational demands of MT in an educational auto-grading system, the effect of MT on student test suite quality and coding practices, and the development of a framework for effectively integrating MT in programming courses. Initially, the implementation of standard MT showed mixed results due to inadequate stock feedback. This prompted me to develop a tailored approach that modified MT feedback, while also incorporating additional documentation and training materials. I also observed a noticeable increase (30-50 seconds per submission) in the auto-grader's processing time and feedback turnaround time when using MT, raising concerns about potential server overload. At the same time, the collection of changes made to the environment and requirements as part of this intervention led to an overall reduction in the number of submissions per student needed to complete the projects. My findings suggest that students using modified MT, as a group, demonstrated higher quality test suites and wrote better solution code compared to students whose test suites were graded on code coverage. This

version of MT with modified feedback also showed positive results in student understanding and application of MT principles compared to MT with stock feedback. Analysis of IDE activity data, code submissions, and 38 semi-structured student interviews led me to provide a framework for introducing MT as an effective intervention. Thus, my research provides a framework for effectively integrating MT in programming courses, contributing to improved student test suite development and offering practical guidelines for instructors introducing MT in undergraduate Computer Science courses.

Towards Improving Students' Software Testing Practices using Modified Mutation Testing

Rifat Sabbir Mansur

(GENERAL AUDIENCE ABSTRACT)

Software plays a crucial role in our daily lives, from the apps on our smartphones to the systems that control our cars and homes. Ensuring that software works correctly and reliably is essential to prevent potential harm or inconvenience caused by malfunctions. One way to ensure software quality is through thorough testing, which involves writing test cases that check whether the software behaves as expected under various conditions. One advanced testing technique used in the software industry is mutation testing (MT), which deliberately introduces small changes (or "mutations") into working code to see if existing tests can catch these artificial mistakes. While this technique is powerful, its adoption in programming courses has been challenging due to technical limitations and teaching difficulties. My research focused on successfully introducing MT in a college-level computer programming course. I studied how MT could be effectively taught to students while managing the additional computational demands it places on the course's automated grading system. Over multiple semesters, I developed and refined an approach that made MT more accessible and beneficial for students. Initially, using standard MT tools proved problematic because students found the stock feedback difficult to understand. In response, I modified a more student-friendly version with clear explanations, documentations, and training materials. While this improved version did require more processing time from the grading system, I also applied solutions to alleviate computational overload on the course's automated grading

system, such as allowing students to run MT on their own computers. The results were encouraging: students who used modified MT wrote better tests and produced higher quality code compared to students graded using the previous test suite quality measure (code coverage). Following interviews with students and analysis of their coding patterns, I developed a comprehensive framework for successfully introducing MT in programming courses. Finally, my research provides practical guidelines for instructors who want to incorporate this industry-standard testing technique into their teaching, ultimately helping students become better programmers. My research contributes to the field of Computer Science education by improving training in testing, helping students create more robust software tests, and providing instructors with insights into effectively teaching this influential yet complex topic.

Dedication

To my beautiful Maya,

whose tiny heartbeat gives rhythm to my universe.

Acknowledgments

Alhamdulillahi Rabbil-'alamin. All praise is for Allah — Lord of all worlds. Every accomplishment, every moment of clarity, and every obstacle overcome on this journey has been by His grace and mercy.

A PhD is more than an individual achievement; it is the culmination of collective support, guidance, and love from countless individuals who touch our lives. This dissertation would not have been possible without the village of supporters who believed in me, even when the path seemed uncertain.

First and foremost, I am eternally grateful to my family, my strongest foundation through this journey. To my father, Dr. Abul Mansur, who planted the seeds of curiosity and instilled in me an unquenchable thirst for knowledge. Your own academic pursuits showed me that intellectual growth knows no boundaries. To my mother, Jesmine Aktar, who taught me the power of empathy and persistence in the face of hardship—your strength has been my guiding light through the darkest moments of this journey.

I owe special gratitude to my sister, Dr. Fariha Jasin Mansur—my inspiration and, without question, the brighter sibling. As the first in our family to earn a PhD, you paved the way and showed me it was possible. Thank you for consistently supporting me to grow and become a better version of myself. Your unwavering belief in my abilities pushed me forward when I doubted myself.

Words cannot express my gratitude to my wife, Rahnuma Nurain Maria, whose constant presence transformed every difficult moment into something manageable, even beautiful. Your patience, understanding, and unwavering support created the stable foundation I needed

to pursue this dream. The countless late nights, the missed weekends, and the emotional rollercoaster of research—you weathered it all with grace and love.

To my daughter, Mehrish Nurain (Maya), the newest and brightest light in my life—you have redefined my purpose. Though you arrived near the completion of this journey, your presence has given me renewed motivation and perspective. You are everything that makes me strive to be a better person, scholar, and father. I hope one day you will understand that your tiny hands held my heart through the final stretches of this work.

I am deeply indebted to my advisors, Dr. Cliff Shaffer and Dr. Steve Edwards, who shaped not just my research but my entire approach to scholarship. Dr. Shaffer, thank you for teaching me how to approach teaching with intentionality and purpose, and for showing me how to transform problems into meaningful research opportunities. Dr. Edwards, your meticulous attention to detail and practical wisdom helped me understand the critical importance of effective implementation. Our countless interactions over the years have been invaluable in shaping my academic identity.

My heartfelt appreciation extends to my committee members. Dr. Eli Tilevich (Dr. T), one of the coolest professors I've had the privilege to work with, thank you for demonstrating how to effectively communicate complex technical topics to broader audiences—a skill I will carry forward in all my future endeavors. Dr. Francisco Servant, your Software Engineering class in my first year fundamentally reshaped my approach to research. Our discussions about research papers developed my early research mindset in ways I'm still discovering. Dr. Thomas Price, your insightful suggestions and valuable perspectives consistently pushed my research toward greater completeness and impact. I continue to learn from you the importance of examining my work from multiple angles and identifying opportunities for improvement and extension.

I am blessed to have had exceptional teachers and mentors throughout my life. While it is impossible to name everyone who has guided me, please know that each of you has contributed meaningfully to my development as a scholar and human being. To my students, thank you for constantly inspiring me to learn new ways of teaching. In my role as an educator, I find myself in a continuous state of learning alongside all of you. Our collective pursuit of knowledge represents the beautiful way in which we, as a human species, grow and evolve together through shared understanding.

A PhD journey would be unbearably lonely without friends to share in both struggles and triumphs. I am grateful for all the wonderful people who made my life cheerful during the difficult transition to a new country while pursuing advanced degrees. Special thanks to my bandmates in "Migratory Birds of Blacksburg," the cultural folk band I co-founded. Those performances and our time together represent some of the most joyful moments of my graduate experience, providing much-needed balance and creative expression.

I extend my gratitude to the Computer Science Department at Virginia Tech for creating a supportive environment where research and innovation could flourish. The faculty, staff, and fellow students have collectively created a community that fosters growth and collaboration.

Finally, I am grateful to the National Science Foundation for their financial support under grant DLR-1740765. This funding made it possible for me to focus on my research and academic development without the additional burden of financial concerns.

As I close this chapter and begin the next, I carry with me the lessons, support, and love of everyone mentioned here and many more. Any achievements that may come from this work are shared with all of you who have walked this path alongside me.

Contents

List of Figures **xiv**

List of Tables **xviii**

1 Introduction **1**

2 Review of Literature **6**

2.1 Computational Cost of Running Test Quality Metric on Online Auto-Grader 6

2.2 Impact of Mutation Testing as Test Quality Metric in Programming Courses 8

2.3 Effect of Modified Mutation Testing in Programming Courses 12

3 Computational Cost of Running MT **15**

3.1 Methodology 18

3.1.1 Deploying Mutation Testing 18

3.1.2 The Web-CAT Auto-Grader 20

3.1.3 Data collection 21

3.2 Results 24

3.2.1 RQ1: Impact of MT on Server Processing Time 25

3.2.2 RQ2: Impact of MT on Feedback Turnaround Time (FTT) 29

3.2.3 RQ3: Impact of local MT use on Web-CAT submissions 35

3.3 Start-up Problems 39

3.4 Discussion 41

3.4.1 RQ1: Additional Processing Time due to MT 41

3.4.2 RQ2: Impact of MT on Feedback Turnaround Time 42

3.4.3 RQ3: Impact of Client-Side MT on Student Submission Behavior . . 43

4 Impacts of MT on Student Performance 44

4.1 Research Methodology 46

4.1.1 Participants 47

4.1.2 Study Design 48

4.2 Results and Discussion 49

4.2.1 RQ1: Effects of MT on Student-written Test Cases 49

4.2.2 RQ2: Effects of MT on Project Solution Code 53

4.2.3 RQ3: Effects of MT on Testing Activity 55

5 Improving MT Feedback 59

5.1 Research Approach 61

5.1.1 Programming Assignments and Assessment Rubrics 63

5.1.2 Qualitative Research 64

5.1.3 Quantitative Research 68

5.2 Code Coverage Semester 69

5.2.1 Description of CC Intervention 69

5.2.2 Students' Impression of CC 70

5.2.3 CC Intervention Summary 76

5.3 MT Phase I: Deploying MT with PIT Stock Feedback 77

5.3.1 Description of PIT stock MT Intervention 77

5.3.2 Students' Impression of MT with PIT Stock Feedback 81

5.3.3 PIT stock MT Intervention Summary 84

5.4 MT Phase II: Implementing MT with Modified Feedback and Instructional Materials 85

5.4.1 Description of Modified MT Intervention 85

5.4.2 Students' Impression of MT with Modified Feedback and Instructions 87

5.4.3 Modified MT Intervention Summary 94

5.5 Quantitative Analysis 96

5.6 Discussion And Threats to Validity 103

6 Conclusions and Future Work **110**

Bibliography **113**

Appendices **123**

Appendix A Additional Materials **124**

A.1 Interview Script 124

A.2 Investigation of Null Results 127

List of Figures

3.1 The MT semester exhibits a statistically significant increase (about 44 seconds or 220%) in processing time for DSA submissions using MT (mean 64 seconds), with a median processing time of about 50 seconds and a wider range of processing times (Q1: 30 seconds, Q3: 83 seconds). In contrast, the non-MT semester had a shorter mean (20 seconds) and shorter median (16 seconds) processing time for DSA submissions using CC, and a narrower range of processing times (Q1: 13 seconds, Q3: 19 seconds). 27

3.2 When the Web-CAT server is at load, the processing time of DSA submissions takes significantly longer (about 56 seconds or 187%) to complete using MT (mean 86 seconds) compared to using CC (mean 30 seconds). The median processing time of DSA submissions using MT is 81 seconds with a wider range of processing times (Q1: 41 seconds, Q3: 111 seconds). In contrast, the median processing time of DSA submissions using CC is 23 seconds with a narrower range of processing times (Q1: 17 seconds, Q3: 32 seconds). 28

3.3 The distribution of feedback turnaround times (FTTs) for all courses running on Web-CAT in the non-MT semester (left) and MT semester (right). The MT semester exhibits a significant increase in feedback turnaround time (FTT) (27 seconds for the median time, or 139%), for DSA projects using MT. Note: The y-axis has been limited to 600 seconds for both figures to facilitate direct visual comparison. As a result, some outliers extend beyond the visible range. 33

3.4 The distribution of feedback turnaround times (FTTs) for all courses running on Web-CAT at load condition in the MT semester (left) and non-MT semester (right). There is a statistically significant increase in FTT for DSA course in the MT semester, with an additional delay of approximately 48 seconds (or 56% longer than usual). However, note that only 4% of submissions experienced higher FTT delays due to load. Note: The y-axis has been limited to 600 seconds for both figures to facilitate direct visual comparison. As a result, some outliers extend beyond the visible range. 34

3.5 Frequency of student-initiated assessments of test quality, specifically local execution of code coverage (non-MT semester) and mutation testing (MT semester). Twice as many test coverage checks were made locally using Eclipse during the MT semester. This is significant ($p - value < 0.001$), indicating a clear and substantial shift in student behavior. 35

3.6 Number of submissions made to the Web-CAT by each student during non-MT semester (left box plot) and MT semester (right box plot). There is a statistically significant difference between the two groups, with students in non-MT semester exhibiting a higher tendency to submit to Web-CAT ($p - value << 0.001$). The number of submissions made by students in non-MT semester is approximately 40% greater than by their MT semester counterparts. 36

4.1 Comparing mutation coverage of students' test suites for four different comparable projects from non-MT and MT semesters. We used four different tools: Code Coverage (CC), PIT with selected mutators (PIT-SM), PIT with All mutators (PIT-ALL), and MuJava. 50

4.2 Comparison of correctness scores of students' final submissions for four different comparable projects for non-MT and MT semesters. 54

4.3 Distribution of cyclomatic complexities for student-written solution code (sorted low to high) for four projects across MT and Non-MT semesters. Lower cyclomatic complexity scores are considered better. 55

5.1 CC analysis of student-written test suite in Eclipse IDE demonstrated through EclEmma's color-coded interface and coverage statistics. 70

5.2 MT analysis (using selective mutators) of a student-written test suite in Eclipse IDE with PIT's stock feedback. 78

5.3 MT analysis (using selective mutators and modified feedback) of a student-written test suite on Web-CAT AAT online portal. 80

5.4 MT analysis (using selective mutators and modified feedback) of a student-written test suite shown in the Eclipse IDE using our MT plug-in with modified feedback. 86

5.5 Distribution of incremental test writing scores across CC, MT_{ps} and MT_{mf} semesters, showing significantly lower values in MT semesters compared to CC. Lower values indicate closer alignment between test and solution edits, demonstrating MT's effectiveness in promoting incremental testing practices throughout project development. 95

5.6 Comparison of mutation coverage scores across CC (48%), MT_{ps} (65%), and MT_{mf} (73%) semesters using MuJava, demonstrating significantly higher test suite quality in MT semesters (p-value < 0.001), with modified feedback (MT_{mf}) yielding the most robust test suites. 98

5.7 Comparison of cyclomatic complexity values across semesters, showing significantly lower complexity in MT semesters (MT_{ps} = 152, MT_{mf} = 119) compared to CC semester (mean = 220, p-value < 0.05), indicating improved code testability through MT intervention. 100

5.8 Comparison of Students' Project 1 Correctness Score in CC, MT_{ps} and MT_{mf} Semesters. Here, mean scores for students in CC, MT_{ps}, and MT_{mf} semesters were 70%, 69%, and 82%, respectively. This indicates, a large portion of the students using PIT's stock MT tool had difficulty with project correctness score. 101

5.9 Comparison of Students' Project 4 Correctness Score in CC, MT_{ps} and MT_{mf} Semesters. Here, mean scores for students in CC, MT_{ps}, and MT_{mf} semesters were 69%, 79%, and 86%, respectively. This shows students using MT in both MT_{ps}, and MT_{mf} semesters had higher project correctness scores compared to students using CC in their fourth and final project. 101

A.1 Comparison of OpenDSA final scores across the three semesters: CC, MT_{ps} and MT_{mf}. We found that the introduction of MT concepts did not substantially impact students' performance of solving OpenDSA assignments. . . . 128

List of Tables

3.1 Hardware specification of our online auto-grader, Web-CAT. 19

3.2 Comparison of Web-CAT submission statistics across Non-MT and MT semesters 23

3.3 Feedback Turnaround Time (FTT) metrics for DSA and non-DSA courses during MT and non-MT semesters, comparing the performance of the Web-CAT server when under overall vs. load conditions. 32

4.1 Analyzing differences in students' final solution code: comparison of four comparable projects in MT and non-MT semesters using Kruskal-Wallis with Dunn's post-hoc test with Bonferroni correction. All differences in means are significant, with p-values of 0.001 or lower. 52

5.1 Semi-structured Interview Protocol 66

5.2 Interviews from three groups of students across three semesters. 66

5.3 Demographics of the Student Interviewees. 68

Chapter 1

Introduction

Software testing is a critical component of the software development lifecycle, ensuring the quality and reliability of software systems [33, 49, 74]. In the context of undergraduate Computer Science Education (CSEd), teaching effective software testing practices is crucial for equipping students with the necessary skills for their future careers [34, 72]. Mutation testing (MT) [15], a fault-based testing technique, has emerged as a powerful tool for evaluating the effectiveness of test suites and improving software quality. In the software industry, MT plays a vital role in ensuring software reliability by identifying weaknesses in test suites, guiding developers towards more comprehensive testing strategies.

Despite the importance of software testing, few Computer Science departments offer dedicated courses on this subject [41, 49, 64]. Instead, many programs incorporate testing education within existing programming courses through automated assessment of student-written test cases. Traditionally, code coverage (CC) has been the primary metric for evaluating test effectiveness in these settings. CC measures the percentage of code executed during testing, providing a basic indication of testing thoroughness. However, CC has significant limitations in measuring actual fault detection capabilities and can be manipulated by students to achieve high coverage without meaningful testing [17].

MT offers a more rigorous approach to assessing test suite quality by systematically introducing small faults ("mutations") into the code and evaluating whether the test suite can detect these changes [15]. While MT has gained popularity in industry [6, 56, 60], its

adoption in educational settings faces both computational and pedagogical challenges. The computational cost of MT is significant, as each mutation requires a separate execution of the test suite, potentially leading to increased processing times and server load issues in automated grading systems. Additionally, the complexity of existing MT feedback can be overwhelming for students, requiring careful consideration of how feedback is presented and interpreted.

My dissertation research investigates the comprehensive integration of MT in undergraduate CSEd through a series of three studies, each addressing a key aspect of MT implementation:

1. Practical Integration of MT and its Computational Cost

The first study examines the computational implications of integrating MT into an educational auto-grading system. I investigate the additional processing time required to assess test suites by MT, the impact of MT on feedback turnaround time (FTT) for both DSA and other courses running on the auto-grader, and the potential of client-side MT execution to mitigate server load issues. This study aims to provide concrete guidelines for institutions considering MT implementation, helping them balance the educational benefits with the practical constraints of their computational resources.

Research questions associated with this study:

- **RQ1.** How much additional processing time is required for a typical DSA job that uses MT?
- **RQ2.** What is the impact of using MT in a DSA course on the FTT for DSA projects within the course, and the FTT for projects in other concurrent courses?
- **RQ3.** Does providing client-side MT execution capability affect the number of submissions students make to the AAT?

2. Impacts of MT on Student Performance

The second study focuses on the impact of MT on student performance in programming projects. I compare student performance between a semester where students were graded using CC to assess test suite quality and a semester using MT. I compared test suite quality, solution code correctness and complexity, and testing behavior patterns. This study assesses the pedagogical benefits of MT in fostering better testing practices and overall software development skills.

Research questions associated with this study:

- **RQ1.** How does our introduction of MT impact the quality of student-written test cases?
- **RQ2.** What is the effect of MT on the project solution code written by the students?
- **RQ3.** How does our introduction of MT impact students' behavior in terms of test case development?

3. Effect of Modified MT Feedback in Programming Courses

The third study addresses the pedagogical challenges of integrating MT in CS education. I investigate students' perceptions and behaviors when using CC, their experiences with standard MT feedback provided by industry tools, and the development of modified MT feedback mechanisms tailored for educational contexts. Through multiple semesters of implementation and refinement, supported by student interviews and quantitative analysis, I develop a comprehensive framework for presenting modified MT feedback in educational settings. This framework aims to bridge the gap between students' current understanding and the sophisticated feedback provided by MT, ultimately improving their testing practices and understanding.

Research questions associated with this study:

- **RQ1.** What are students' perceptions of the value of writing test suites and their testing behaviors when using CC in programming projects?
- **RQ2.** What are students' performance and their impressions of using PIT's stock MT feedback for evaluating their test suites?
- **RQ3.** What impact does modified MT feedback and targeted MT instructional resources have on students' testing perception, practices, and outcomes?

The overarching goal of my research is to provide a comprehensive framework for effectively integrating MT into undergraduate CS courses, addressing the computational, performative, and pedagogical challenges associated with its implementation. By investigating these three interconnected aspects, I aim to bridge the gap between industry-standard testing practices and the unique needs of educational settings, ultimately contributing to the improvement of software testing education and better preparing students for professional software development careers.

Through this comprehensive investigation, my dissertation contributes to the field of CSEd by providing practical guidelines for instructors seeking to incorporate MT into their courses, a framework for managing the computational demands of MT in educational settings, and insights into the pedagogical strategies that can maximize the educational benefits of MT.

The outline of my dissertation is as following:

- Chapter 2: I place my work within a broader context by reviewing and linking literature on the computational cost of running MT, impact of MT as a test quality metric, and an effective feedback framework for integrating MT.

- Chapter 3 I explore the technical feasibility of MT integration in educational settings, identifying potential bottlenecks and mitigation strategies.
- Chapter 4: Building upon the foundation from Chapter 3, I assess the pedagogical benefits of MT in terms of student performance outcomes and development of testing skills.
- Chapter 5: I address the challenges using standard MT feedback, proposing a modified MT feedback and training materials to meet the specific needs of undergraduate students.
- Chapter 6: I provide a summary of our findings and discuss potential directions for future research that emerge from the work presented in this dissertation.

Statement of attribution.

Adhering to the "manuscript" dissertation format, the subsequent chapters incorporate material from published or in-progress manuscripts, where I am the primary contributor and author. I have reorganized certain sections to create a cohesive document.

The papers are:

- Exploring the Bug Investigation Techniques of Intermediate Student Programmers [43]
- Mutating Matters: Analyzing the Influence of Mutation Testing in Programming Courses [44]
- The Hidden Cost of Mutation Testing on Online Auto-Grader [45]
- Improving Students' Testing Skill through Mutation Testing with Modified Feedback and Training (in preparation)

Chapter 2

Review of Literature

2.1 Computational Cost of Running Test Quality Metric on Online Auto-Grader

Software testing is a critical part of evaluating the correctness of computer programs. Therefore, many CS educators have incorporated software testing into the curriculum [4, 18, 33, 67]. Even so, Edwards et al. [21] discovered that the majority of student-written test cases tend to only cover mainstream expected behavior of the code, and not to expose more subtle bugs.

A number of techniques are meant to help student programmers get better at software testing. One of the most important is to give measures of test suite quality. Researchers have proposed many approaches to evaluate student-written test cases [1, 18, 25]. One approach is code coverage (CC), which measures the fraction of the project code that is executed when a student's test suite runs. CC is fast to compute and supports incremental feedback.

Incremental feedback is the pedagogical approach of providing students with timely and continuous evaluative feedback throughout their developmental process. This contrasts with proving singular summative feedback after their final submission. Incremental feedback has been found to improve student learning outcomes [7, 18]. Incremental feedback enables students to monitor their progress, recognize areas of improvement, and rectify mistakes as

they emerge. Pedagogically, fast feedback (seconds, not minutes) has been found to improve student learning [5, 68], and so is becoming standard practice in CS education [58]. Many educational institutions also use a centralized Automated Assessment Tool (AAT) [31, 55, 57, 67, 70] to support providing incremental feedback.

However, good code coverage does not necessarily mean good ability to detect bugs, since it is satisfied simply when tests execute the code, whether or not they also provide confirmation that the executed code works correctly [1, 20, 30]. In particular, high CC scores are not always correlated with bug-free software both in educational [62, 66] and non-educational [30] settings. Additionally, students often write "pathological" tests to achieve high coverage when that is what they are graded on [62].

Mutation testing (MT) is an alternate test suite quality metric [15]. It has a stronger test adequacy criterion [1, 52, 62, 73], meaning better correlation with detecting bugs. Aaltonen et al. [1] reported that MT revealed deficiencies in students' testing that were not revealed by CC. MT generated "mutants" in the solution code have been found to be valid substitutes for "real" faults [3, 36].

Unfortunately, MT is computationally expensive [32]. This can significantly delay the incremental feedback process. Speedy program execution might prevent the AAT from getting overloaded. However, students' tests and solutions are typically not completed until near the deadline [39, 40]. As a result, at institutions that offer incremental feedback, each student might make many dozens of submissions per assignment to the institution's AAT, with bursty traffic near due dates [22]. This can cause overload on the AAT servers and hinder the process of providing fast and continuous incremental feedback.

There has been much research to lower the computational cost of MT. Jia and Harman [32] categorized these efforts into "Do fewer," reducing the number of generated mutants, and

"Do faster," reducing the execution cost in other ways (e.g., avoiding I/O). Shams and Edwards [63] conducted a comprehensive investigation into the application of MT in introductory programming courses. Their research extended to a comparative analysis of the cost-effectiveness of diverse selective mutation approaches against other established measures of test quality [20]. The study yielded a significant finding: the statement deletion technique, which involves the straightforward removal of statements from the source code, emerged as the most cost-effective mutation-based test adequacy criterion for developing high-quality tests. Kazerouni et al. [37] found that using two particular mutation operators ("Remove Conditionals" and "Arithmetic Operator Deletion") provides bug detection at 92% of the accuracy of the full set of operators from the open-source MT tool PIT [10] with only one-tenth of the computational cost. The first, "Remove Conditionals," focuses on testing conditional statements, while the second, "Arithmetic Operator Deletion," focuses on testing arithmetic operations. In my research, I aim to effectively deploy a MT tool within a substantial programming course and assess its impact on the resources of the auto-grading system.

2.2 Impact of Mutation Testing as Test Quality Metric in Programming Courses

Software testing is critical to evaluating the correctness of computer programs, and many CS educators have incorporated software testing into the curriculum [4, 18, 33, 67]. Researchers have proposed techniques to help student programmers improve their software testing skills. One approach is to provide feedback on the quality of student-written test suites [1, 18, 25].

A basic metric for a test suite is its number of tests. While studies by Namin and An-

drews [50], Gopinath et al. [26] and Inozemtseva, and Holmes [30] indicate a correlation between test suite size and test set coverage, contradictory conclusions were drawn by Just et al. [36] and Papadakis et al. [54], suggesting a negligible correlation. To provide a more comprehensive evaluation of the test suite, I move beyond simply counting the number of test cases. Instead, I aim to assess the quality and effectiveness of the test suite itself.

One commonly used metric for measuring the quality of a test suite is code coverage, which determines the lines of project code executed when a student's test suite runs. Code coverage is fast to compute and supports incremental feedback (that is, it can be evaluated at any time in the project life cycle). However, good code coverage can be achieved simply by executing ("covering") the code, regardless of whether the tests confirm that code's correct behavior [1, 30]. Therefore, studies have shown that high code coverage scores do not necessarily correlate with bug-free software, both in educational [62, 66] and non-educational [30] settings. Some students write "pathological" tests to achieve high coverage scores for grading purposes, which defeats the purpose of testing [62]. Not only does this tactic run up the code coverage score, but it also hides the fact that those "covered" lines have not yet been adequately tested. Code coverage can be useful as a screen, but should be complemented with other techniques to ensure the efficacy of tests.

Mutation analysis [15] has been shown to provide a more reliable measure for assessing the effectiveness of students' test suites in detecting faults than does code coverage [1, 52, 62, 73]. Aaltonen et al. [1] showed that MT could uncover deficiencies in students' testing efforts that were not identified by code coverage analysis. MT has been extensively studied as a technique for evaluating the quality of test suites and their ability to catch real faults. Daran and Thévenod-Fosse [13] found that injecting mutants produced similar errors and failures as real faults when exercised with generated test suites. However, that study was limited in scope, considering only a single 1,000-line C program and evaluating 1% of the generated

mutants without controlling for code coverage.

Andrews et al. [3] found that mutants are a good substitute for real faults, with no practical difference between the mutation score and real fault detection rate. However, this study also had limitations, considering only one C program called "Space," developed at the European Space Agency [69] with real faults, evaluating 10% of generated mutants, and not controlling for code coverage. Namin and Kakarla [51] replicated this work and found a weak correlation between mutation score and real fault detection for Space, but a stronger correlation for hand-seeded faults in smaller Java programs. Just et al. [36] found that 73% of real faults are coupled to commonly used mutation operators, with an average of two mutants coupled to a single real fault. They also found that mutant detection is positively correlated with real fault detection, independently of code coverage, and more strongly than statement coverage. Papadakis et al. [54] found moderate-to-strong correlations between mutant detection and fault detection when test set size was ignored. However, Chen et al. [9] questioned whether the higher fault detection of mutation-adequate test suites is due to better test goals or simply more tests.

In the context of CS education, Mansur et al. [43] discovered that students who participate in more testing activities typically attain higher overall scores in large programming projects. Hall and Baniassad [27] found that MT encouraged students to write more correct tests, check more code behavior, and actively seek to understand specifications. Our research seeks to examine the impact of introducing MT on students' testing activities, observed in post-CS2 level DSA courses.

MT is facilitated by various tools, with the most popular options for the Java programming language being PIT [10], Major [35], and MuJava [42]. Among these, PIT (Pitest) [10] is widely regarded as the most commonly used MT tool, both in industry and academia, due to its extensive support and comprehensive documentation [60].

Delahaye and du Bousquet [14] identified three distinct profiles that can influence the selection of a MT tool: teaching, research, and industry. They concluded that PIT is a favorable choice for industrial and teaching contexts, where tools should be easy to apply, striking a balance between efficiency and the meaningfulness of results. Papadakis et al. [53] further noted that while MuJava primarily appears in research projects, PIT widely appears in development and teaching repositories and is also the tool with the most repositories dedicated to research. Given the complexity of evaluating the effects of MT on Java testing, I chose to use a diverse range of MT tools (PIT, Major, and Mujava) in my study.

In addition to my investigation into the effects of MT on student testing, I also aim to evaluate its impact on the quality of student-written solution code. To quantify correctness, I used instructor-provided reference tests as a benchmark for measuring the accuracy of student submissions. For measuring complexity, I relied on McCabe's Cyclomatic complexity metric [46]. This metric provides an objective measure of the number of linearly independent paths within a program's control flow graph. It is widely used in software engineering as an indicator of code complexity and testability [23]. Higher values of cyclomatic complexity suggest that the code is more complex, harder to understand, and harder to test. It can help identify overly complex methods that may need to be refactored or broken into smaller, more manageable units. This metric is useful in large codebases, where identifying and addressing complex code can improve maintainability and reduce technical debt [71]. I compare the cyclomatic complexity of student projects developed when graded for test suite code coverage versus those developed when graded for test suite mutation coverage.

2.3 Effect of Modified Mutation Testing in Programming Courses

Mutation testing (MT) is a powerful technique for evaluating the quality of test suites by intentionally introducing faults (mutations) into the code and assessing whether the tests can detect these mutations. Numerous studies have explored the efficacy of MT in detecting real faults, with varying conclusions. Just et al. [36] found that "73% of real faults are coupled to the mutants generated by commonly used mutation operators," and that "mutant detection is positively correlated with real fault detection, independently of code coverage." However, Daran and Thévenod-Fosse [13] cautioned that their findings were "limited in scope, considering only a single 1,000-line C program and evaluating 1% of the generated mutants without controlling for code coverage."

Despite its effectiveness, MT is computationally expensive [32], which can hinder providing timely feedback to students. Jia and Harman [32] have explored strategies to reduce the computational cost, such as "Do fewer" (reducing the number of generated mutants) and "Do faster" (reducing execution cost). Kazerouni et al. [37] found that using two specific mutation operators provided bug detection at 88% accuracy compared to the full set of operators from PIT [10], with only one-tenth of the computational cost.

MT has been shown to be a valuable tool for improving students' testing skills and coding abilities. [43] discovered that "students who participate in more testing activities typically attain higher overall scores in large programming projects." Aaltonen et al. [1] showed that MT could "uncover deficiencies in students' testing efforts that were not identified by code coverage analysis," highlighting the need for better feedback and instructional strategies. Hall and Baniassad [27] found that MT "encouraged students to write more correct tests, check more code behavior, and actively seek to understand specifications." Especially in later

programming-intensive courses (such as our Junior-level Data Structures course), MT offers a unique opportunity to enhance students' understanding and application of software testing principles due to the large scope of the programming assignments.

Effective feedback is crucial for student learning and motivation. However, traditional feedback mechanisms in these courses often fall short in addressing the specific learning needs of students. Hattie and Timperley [28] noted that feedback is most powerful when it addresses faulty interpretations, not a total lack of understanding. As stated by the authors, "Feedback has no effect in a vacuum; to be powerful in its effect, there must be a learning context to which feedback is addressed." Shute [65] defined formative feedback as "information communicated to the learner that is intended to modify his or her thinking or behavior for the purpose of improving learning." She emphasized the importance of feedback complexity, noting that overly complex feedback can be ineffective, while more elaborative information tends to produce increased understanding. Furthermore, Shute highlighted the importance of goal-directed feedback, which "provides learners with information about their progress toward a desired goal (or set of goals) rather than providing feedback on discrete responses." She emphasized the need for scaffolding in feedback, which "motivates the learner's interest related to the task, simplifies the task to make it more manageable and achievable, provides some direction to help the learner focus on achieving the goal, clearly indicates the differences between the learner's work and the standard or desired solution, reduces frustration and risk, and models and clearly defines the expectations (goals) of the activity to be performed."

Dweck and Leggett's [16] social-cognitive approach to motivation and personality provides a framework for understanding how different types of goals influence learning behaviors. They differentiate between performance goals, which focus on demonstrating ability, and learning goals, which emphasize the acquisition of new skills and knowledge. According to the authors, individuals with performance goals are primarily concerned with the adequacy

of their ability, which can lead to a fixed mindset and a fear of failure. In contrast, those with learning goals are more focused on improving their ability and mastering new challenges, fostering a growth mindset.

Chapter 3

Computational Cost of Running MT

The implementation of MT in educational settings presents unique challenges, particularly regarding computational resources and system performance. While MT offers significant pedagogical benefits, its computational demands can potentially impact the effectiveness of automated assessment tools (AAT) that are crucial for providing timely feedback to students. In this research I examine the computational implications of integrating MT into the post-CS2 DSA course at Virginia Tech. The computational demands of MT are significant because each mutation requires a separate execution of the test suite. In our DSA course, where student submissions typically exceed 900 lines of code and include more than 30 test cases, this computational overhead becomes particularly relevant. The processing of these mutations could potentially affect not only the students in the DSA course but also students in other courses sharing the same AAT infrastructure.

In this chapter, I seek to address the following three research questions.

- **RQ1.** How much additional processing time is required for a typical DSA job that uses MT?
- **RQ2.** What is the impact of using MT in a DSA course on the FTT for DSA projects within the course, and the FTT for projects in other concurrent courses?
- **RQ3.** Does providing client-side MT execution capability affect the number of submissions students make to the AAT?

My investigation focuses on three critical aspects of MT implementation: First, I examine the additional processing time required for MT-enabled jobs (RQ1). This analysis is crucial because longer processing times could potentially impact the educational effectiveness of the AAT. I measure this by comparing the processing times of submissions with and without MT, controlling for factors such as code size and test suite complexity.

Second, I investigate the broader impact of MT implementation on feedback turnaround time (FTT) across both DSA and non-DSA courses (RQ2). This question is particularly important because our AAT server, which operates with five concurrent workers, serves multiple courses simultaneously. When the server reaches a load state (processing over 300 submissions per minute), the impact of MT-related processing could potentially cascade across all courses using the system.

Third, I explore whether providing client-side MT capabilities can help mitigate server load issues (RQ3). Building on previous research by Mansur et al. [43], which established a correlation between testing frequency and project performance, I hypothesize that enabling local MT execution could reduce the number of submissions to the AAT while maintaining educational benefits.

Understanding these computational impacts is crucial for several interconnected reasons in educational settings. First and foremost, timely feedback plays a vital role in effective learning, as demonstrated by previous research [5, 68]. When students receive prompt feedback on their work, they can more effectively identify and correct mistakes, leading to better learning outcomes. Additionally, system performance has a direct influence on student engagement and learning outcomes. When students experience significant delays in receiving feedback, their motivation to make frequent submissions and engage in iterative development may decrease, potentially hampering their learning process. Furthermore, institutions must carefully consider resource allocation decisions, striking a delicate balance between

the educational benefits of advanced testing techniques and their associated computational costs. This balance is particularly critical in educational environments where computational resources are shared across multiple courses and students.

My methodology involves comparing server performance metrics across semesters with and without MT implementation (which used CC instead).

I focus on key metrics including:

Processing time per submission: This metric represents the duration required for our AAT to completely evaluate a single student submission. In the context of MT, this includes the time needed to compile the code, generate mutations, execute the test suite against each mutation, and prepare the feedback.

Queue wait time: This metric measures the duration between when a submission enters the system and when it begins processing. When our AAT server experiences high submission volumes, new submissions must wait in a queue until one of the five worker processes becomes available. This metric is particularly important during peak usage periods, such as near project deadlines, when submission rates can exceed 300 per minute. Queue wait time directly impacts students' experience with the system and their ability to receive timely feedback on their work.

Feedback turnaround time (FTT): This metric encompasses the total duration from submission to feedback delivery, combining both queue wait time and processing time. FTT serves as my primary metric for assessing system performance from the student perspective. It represents the actual wait time students experience when submitting their work, excluding network delays and browser load times which vary based on individual circumstances. Understanding FTT is crucial because prolonged wait times can potentially discourage students from making frequent submissions and impact their learning process.

Server load patterns during peak usage periods: These periods describe the temporal distribution of submission volumes and system resource utilization. These patterns typically show increased activity near assignment deadlines, with submission rates often exceeding 300 per minute during peak periods. By analyzing these patterns, we can better understand how MT implementation affects system performance under various load conditions and identify potential bottlenecks in the assessment process. This information is vital for developing strategies to maintain system responsiveness while providing the educational benefits of MT.

This investigation aims to provide concrete guidelines for institutions considering MT implementation in their programming courses, helping them balance the educational benefits of MT with the practical constraints of their computational resources.

3.1 Methodology

3.1.1 Deploying Mutation Testing

In our study, we deployed a modified version of PIT [10], an open-source MT tool. To mitigate the computational overhead associated with running MT on individual student submissions, we adopted the selective set of mutators proposed by Kazerouni et al. [37]. Our selective MT approach incorporates two specific mutators:

- "Remove Conditionals": This mutator systematically replaces conditional expressions in the source code with 'true' and 'false' values.
- "Arithmetic Operator Deletion" (AOD): This mutator iteratively removes one operand at a time from each arithmetic operation present in the source code.

Table 3.1 Hardware specification of our online auto-grader, Web-CAT.

Specification	Details
Operating System	CentOS 7
CPUs	two 16-core 2.60 GHz Intel Xeon Gold 6142 CPUs
Memory	256 GB RAM
Persistent storage	two 600 GB, 10,000 RPM Hitachi HDDs

By employing these targeted mutators, we aim to strike a balance between comprehensive testing and computational efficiency, enabling the practical application of MT in our educational context.

Our system architecture facilitates both local and auto-graded MT environments. For the local setup, we have developed and distributed an Eclipse IDE plug-in that implements our cost-effective selective MT approach. This plug-in enables students to perform incremental MT directly within their development environment during the coding phase, similar to how they might access CC. This approach enhances the convenience of evaluating test suite quality and allows students to quickly identify and rectify code issues based on MT results. To maintain consistency, the plug-in employs the same two mutation operators as the Web-CAT auto-grader. This consistency ensures that students receive comparable feedback regardless of the testing environment.

This local testing capability has the potential to reduce the overall number of submissions to the auto-grader for two reasons. First, consistent use of MT by students should facilitate faster bug detection and resolution, ultimately streamlining the process of debugging and passing reference tests. Second, students won't need to rely on the auto-grader to let them know if their test suite provides adequate mutation coverage to get full credit for this aspect of grading.

In order to measure the quality of students' test suite as part of their project score, we execute MT on each of the submissions made by the students to our Web-CAT server. Web-CAT's

MT environment leverages the Apache Ant framework for initiating the mutation analysis process. Apache Ant [**?**] is a popular build automation tool that can be used to automate the process of compiling, testing, and deploying software projects. Once the MT process is complete, Web-CAT generates and displays the mutation feedback to the student. This feedback provides insights into the correctness of the student's solution code and quality of student's test suite. As part of a seamless experience, students can submit their project code to our Web-CAT auto-grader directly from their Eclipse IDE via our plug-in.

3.1.2 The Web-CAT Auto-Grader

During most of the academic term, our Web-CAT server experiences a consistent load of hundreds of student users on a daily basis, with approximately 5,000+ submissions on peak days. On regular days, these users generate an average of 11 submissions per minute throughout the day. This pattern of student submissions is not uniformly distributed. Rather it is heavily clustered, especially, before assignment deadlines. Usually the server's processing capacity is sufficient that load from submissions does not cause any delay in feedback to students. During peak submission periods close to assignment deadlines, the Web-CAT server experiences a heightened submission rate, averaging 39 submissions per minute and can receive over 2,500 submissions within the hour leading up to a major deadline. The wall time needed for the Web-CAT server to process a submission varies depending on the submission's size and complexity, typically ranging from 15 seconds to 3 minutes. This encompasses the entire process of generating comprehensive submission feedback, including compilation, static analysis, running instructor-written reference tests, and assessment of test adequacy (either using MT or CC).

Table 3.1 shows a hardware specification of our Web-CAT online auto-grader. The server

runs the CentOS 7 operating system and is equipped with two powerful 16-core Intel Xeon Gold 6142 CPUs, each running at a clock speed of 2.60 GHz. This provides a total of 32 CPU cores for efficient parallel processing and handling of multiple concurrent users. The server has a substantial amount of memory, with 256 GB of RAM. This allows Web-CAT to handle large datasets and support multiple users simultaneously without experiencing performance bottlenecks most of the time. For persistent storage, the server utilizes two Hitachi hard disk drives (HDDs), each with a capacity of 600 GB and a spindle speed of 10,000 RPM. These high-performance HDDs ensure fast read and write operations, enabling quick access to stored data and efficient handling of user requests.

Prior to the implementation of MT, the Web-CAT system's hardware configuration had consistently met the computational demands for code analysis, testing, and grading tasks across various courses. This infrastructure demonstrated reliable performance, with an average feedback turnaround time (FTT) of 57 seconds, which falls within acceptable parameters for a large-scale AAT such as Web-CAT.

However, the introduction of computationally intensive MT techniques in a large DSA course necessitates a thorough evaluation of the system's capacity to manage the increased server load. This assessment is crucial to maintain the efficiency and responsiveness of the Web-CAT platform while accommodating the additional computational requirements imposed by MT. The following sections will detail our data collection for measuring and analyzing the impact of MT implementation on system performance and FTT.

3.1.3 Data collection

Our study focuses on two specific semesters. During Fall 2021, students in the DSA course were graded in part on test suite quality as measured by CC. In the intervening semesters,

we had some initial deployment of MT to the course, improved our training procedures, and ironed out some start-up problems with deploying MT (see Section 3.3. By Spring 2024, MT was routinely deployed in the DSA course, and this is the semester used for our study.

The shift from CC to MT introduces additional computational complexity to our auto-grading process. MT, while providing more comprehensive insights into test suite quality, is inherently more resource-intensive than CC analysis. This increased computational demand is due to the need to generate and execute multiple mutated versions of students' solution code against their test suite. In our study, we analyzed server load data from non-MT and MT semesters to evaluate the impact of transitioning from CC to MT in our DSA course. The data collection period encompassed non-MT semester (Fall 2021) during which CC was used, and MT semester (Spring 2024) following the implementation of MT.

To ensure the validity of our comparative analysis, we sought to minimize the influence of differing project complexity and conditions on our calculations. This was another reason for choosing the Fall 2021 and Spring 2024 courses to compare, as they used quite similar project sets. By comparing these two specific semesters, we aim to isolate the effects of CC vs MT on server load and similar relevant metrics.

Another confounding factor in our analysis is the varying student enrollment across non-MT and MT semesters. We observed a general upward trend in student enrollment, with an increase observed in the semesters using MT compared to the earlier CC semesters. This increase in the student population might directly translate to an increased volume of submissions to our Web-CAT server, which could impact its performance and processing time.

DSA student enrollment for Fall 2021 and Spring 2024 semesters was 328 and 381, respectively. Further analysis of server logs revealed similar patterns in overall Web-CAT activity

Table 3.2 Comparison of Web-CAT submission statistics across Non-MT and MT semesters

	Non-MT semester			
	Total # of Submissions	% (of all subs)	# of Submissions under Load	% of Submissions under Load
All courses	223521		9418	5.89% (of all subs)
Other Courses	179155	80.15%	6071	4.74 % (of all other courses)
DSA Courses	44366	19.85%	4681	10.55% (of all DSA)
	MT Semester			
	Total # of Submissions	% (of all subs)	# of Submissions under Load	% of Submissions under Load
All courses	232217		8911	3.84% (of all subs)
Other Courses	188510	81.18%	7583	4.02% (of all other courses)
DSA Courses	43707	18.82%	1327	3.04% (of all DSA)

between the two semesters. As seen in Table 3.2, the total number of Web-CAT submissions (from all courses) were 223,521 and 232,217 for Fall 2021 and Spring 2024, respectively, representing a modest increase of less than 4% between the two semesters. We also observed comparable Web-CAT submission patterns across non-MT and MT semesters, with the DSA course exhibiting only a 1.5% difference (44,366 vs. 43,707 submissions). These similarities in submission volumes support our decision to use these two semesters as appropriate candidates for comparison in our study.

To accurately quantify the additional computational cost attributable to MT, we consider two key aspects:

- Submission Processing Time: A detailed analysis of the average time required to evaluate individual student submissions using CC versus MT, while controlling for other potential influences on processing times.
- Submission Processing Time at Load: An assessment of how CC and MT processes impact Web-CAT's processing time when the server is at load, meaning enough concurrent load to engage all parallel workers resulting in at least one submission being queued (and thus experiencing some wait time).

Our study incorporated data collection from both server-side (Web-CAT) and client-side

(students' local machines) sources. Comprehensive logs from Web-CAT were gathered. These contained essential information such as processing time and queue wait time. The server-side logs were collected not only from the DSA course, where MT was introduced, but also from other courses using the Web-CAT server without MT. To establish a control non-MT semester, we used logs from the Fall 2021 DSA course, which used CC for test quality measure. This provided a baseline for comparison in our subsequent experiments. In Spring 2024, we deployed our MT implementation as described in Section 3.1.1. To facilitate student adaptation to MT in their project development, we provided additional educational resources and tutorials. Submission through the Eclipse plug-in became mandatory for all DSA students during the MT semester. This enhanced MT intervention enabled the collection of high-quality data, which we consider stable and suitable for our analysis.

To collect logs from students' local workspaces, our Eclipse plug-in logs whenever a student performed MT. These logs enable us to track the frequency at which students execute MT on their test suite. Students are encouraged to conduct MT frequently in their local environment to enhance the quality of their tests. One of our hypotheses suggests that students who frequently run MT on their local machine would exert less load on the Web-CAT server, as they possess a faster and more convenient local MT tool to use in preference to Web-CAT submissions. To test this hypothesis, we compare the submission patterns of students who actively run MT locally versus those who do not.

3.2 Results

Our DSA course requires students to complete four sizable programming projects. One concern is that deploying MT might delay feedback to DSA students, since executing MT on the projects takes time. A second concern is that the computational load required to

support this additional processing for MT use in the DSA course might negatively affect server load for all students. From the server logs, we hope to determine the extent to which MT affects processing time and feedback turnaround time (FTT) on Web-CAT. To see the effects on other courses, we collected similar server log data during the same two semesters.

3.2.1 RQ1: Impact of MT on Server Processing Time

This section presents results comparing the server processing time between a semester using CC (non-MT semester) and another semester using MT (MT semester) for project submissions in our post CS2-level DSA course. The data was collected from server logs spanning two semesters to investigate the additional processing time incurred by introducing MT on our Web-CAT AAT server. Processing times were analyzed both during normal server conditions and load conditions. At load condition, all available server workers are engaged in processing. As a result, incoming submissions are temporarily placed in a waiting queue, awaiting their turn for processing.

To measure the impact of introducing MT on the Web-CAT AAT server, we conducted a comparative analysis of average processing times for evaluating jobs across semesters with and without MT implementation. In this context, a job encompasses the comprehensive evaluation process for an individual student's submission for a large DSA project.

The evaluation process comprises multiple stages:

- Compilation of the submitted code
- Execution of student-written test cases
- Application of MT to assess the quality of student-authored tests
- Execution of instructor-provided reference test cases to evaluate code correctness

- Static code analysis
- Generation of detailed feedback
- Presentation of feedback in a human-readable format

To ensure a robust comparison, we analyzed processing times under various server conditions, including periods of load and absent load. Figure 3.1 presents a visual representation of the processing time distributions for all student submissions for a DSA project, contrasting non-MT and MT semesters for all server conditions. From Figure 3.1, we can observe statistically significant differences in processing times between CC and MT approaches. During the non-MT semester, when CC was used, the server demonstrated a median processing time of 16 seconds for student submissions for a particular DSA project. The interquartile range spanned from 13 seconds (Q1) to 19 seconds (Q3), with a mean processing time of 20 seconds.

In contrast, submissions using MT had a notable increase in processing times for comparable DSA projects. The median processing time rose to 50 seconds, with the first and third quartiles at 30 and 83 seconds, respectively. The mean processing time for MT-based student submissions for that DSA project increased to 64 seconds.

These findings indicate a substantial rise in computational demands when transitioning from CC to MT. The median processing time increased by approximately 213%, while the mean processing time saw a 220% increase. These results indicate that incorporating MT considerably increases the processing time on the AAT server under all load conditions. This represents increased load on the server, and ultimately increased FTT for DSA students.

We also want to see if the load conditions affect the processing time. Figure 3.2 presents the comparison of processing time distributions for all student submissions for the same DSA project between the non-MT and MT semesters, but only considering those submissions made during load conditions. Under load conditions, we observed median processing time

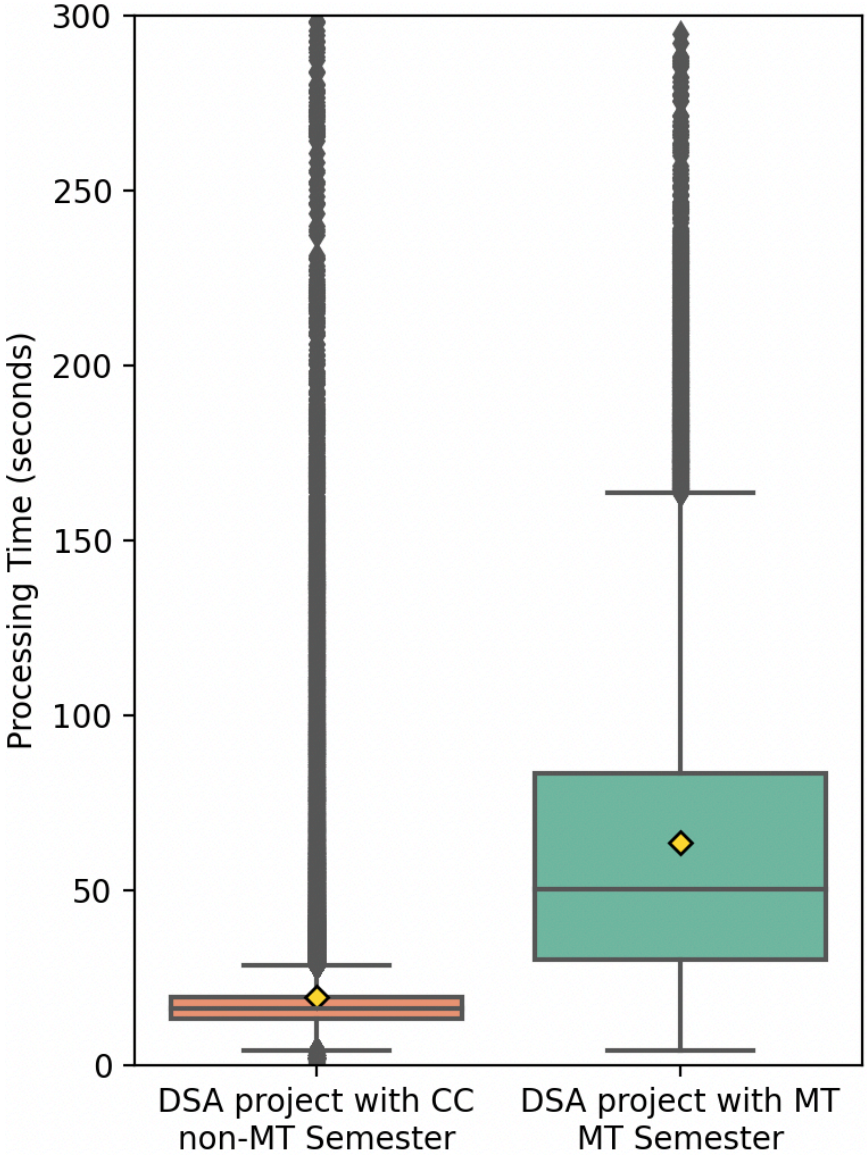

Figure 3.1 The MT semester exhibits a statistically significant increase (about 44 seconds or 220%) in processing time for DSA submissions using MT (mean 64 seconds), with a median processing time of about 50 seconds and a wider range of processing times (Q1: 30 seconds, Q3: 83 seconds). In contrast, the non-MT semester had a shorter mean (20 seconds) and shorter median (16 seconds) processing time for DSA submissions using CC, and a narrower range of processing times (Q1: 13 seconds, Q3: 19 seconds).

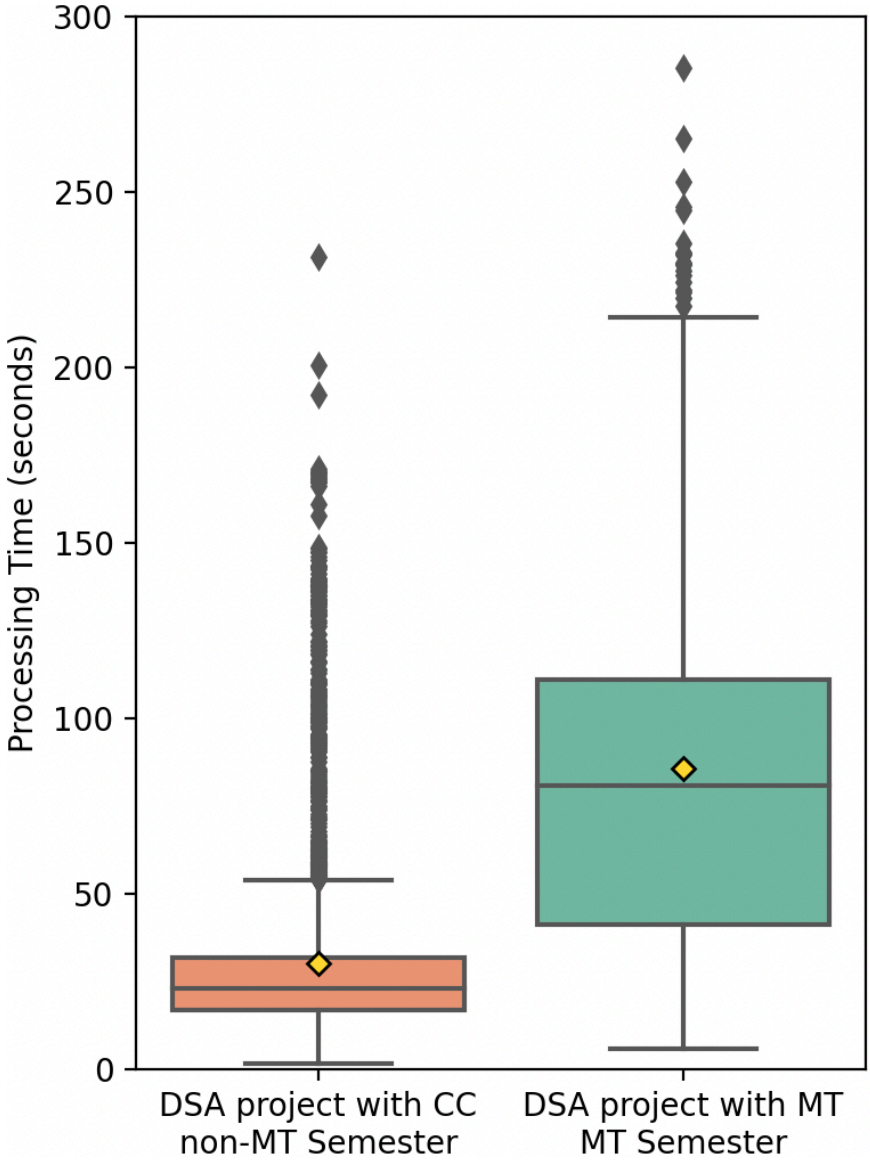

Figure 3.2 When the Web-CAT server is at load, the processing time of DSA submissions takes significantly longer (about 56 seconds or 187%) to complete using MT (mean 86 seconds) compared to using CC (mean 30 seconds). The median processing time of DSA submissions using MT is 81 seconds with a wider range of processing times (Q1: 41 seconds, Q3: 111 seconds). In contrast, the median processing time of DSA submissions using CC is 23 seconds with a narrower range of processing times (Q1: 17 seconds, Q3: 32 seconds).

of 23 seconds, with the first quartile at 17 seconds and the third quartile at 32 seconds in non-MT semester. The mean processing time during load conditions in non-MT semester was 30 seconds. In contrast, we observed significantly higher processing times for student submissions for a comparable DSA project, with a median of 81 seconds, a first quartile of 41 seconds, and a third quartile of 111 seconds for load condition during MT semester. The mean processing time for DSA submissions during the MT semester was 86 seconds.

Comparing the processing time during load conditions, we found a 252% increase in median processing time during MT semester compared to non-MT semester. The mean processing time for load condition during MT semester was 187% higher than that of the loaded CC semester. These findings suggest that the impact of MT on processing time is further amplified when the AAT server is at load. In summary, DSA submissions during non-MT semesters face a median-time "load" penalty of slightly less than 10 seconds, and during MT semesters there is a median-time "load" penalty of about 30 seconds.

In summary: We addressed RQ1 by comparing the processing times between the MT and non-MT semesters. Considering all server conditions, the median processing time for DSA project submissions exhibited a moderate increase of approximately 34 seconds. During load conditions, the median processing time for DSA project submissions had a more substantial increase of about 58 seconds. These differences highlight the significant additional processing overhead incurred by incorporating MT into the DSA course's assessment process.

3.2.2 RQ2: Impact of MT on Feedback Turnaround Time (FTT)

Students do not actually care about processing times. What they care about is the time that it takes to receive feedback on their projects. From the server perspective, we identify two major components to this time. One is the processing time, which we have already analyzed.

The other is wait time due to queuing under load. We call the combination of these two effects Feedback Turnaround Time (FTT).

Figure 3.3 presents a comparison of FTT measured for different courses in non-MT and MT semesters during normal server operation. For DSA submissions, the mean FTT increased from 28 seconds in the non-MT semester to 67 seconds in the MT semester, representing a substantial 139% increase. This suggests that the introduction of MT significantly impacted the FTT for the DSA course during normal server conditions. Additional server resources must be allocated to handle the increased computational requirements of the DSA course using MT, potentially at the expense of submissions from other courses. To test this hypothesis, we compared FTT measured for other courses across non-MT and MT semesters. Interestingly, the mean FTT for other courses decreased by 32%, from 25 seconds in the non-MT semester to 17 seconds in the MT semester. Different courses that run on Web-CAT have projects with varying size and complexity, but generally are considerably faster than DSA projects (even when those DSA projects use CC). As the number of non-DSA jobs varies each semsester, this helps to explain the fluctuation in mean FTT for other courses across two semesters. The important thing to note here is that, when we consider all server conditions, **introducing MT in a DSA course does not appear to negatively impact other courses running on the same AAT**.

The box plots in Figure 3.3 provide further insights into the distribution of FTT values. For the DSA course in the non-MT semester, the median FTT is lower than the mean, indicating a right-skewed distribution with some higher FTT values pulling the mean upward. The box plot for the DSA course in the MT semester shows a higher median and a larger interquartile range (IQR), suggesting a shift towards longer FTT values and increased variability compared to the non-MT semester. The box plots for other courses in both semesters exhibit lower medians and smaller IQRs, indicating consistently lower FTT values and less

variability compared to the DSA course.

Figure 3.4 focuses on the comparison of FTT between the non-MT and MT semesters during loaded times. This means that some jobs must wait to be processed. The mean FTT for the DSA course increased from 86 seconds in the non-MT semester to 134 seconds in the MT semester, an increase of about 50 seconds. This is in line with the increase caused by the introduction of MT seen during non-peak times. For other courses, the mean FTT decreased slightly by 4%, from 84 seconds in the non-MT semester to 81 seconds in the MT semester. This further shows that using MT in a DSA course has no observable effect on other courses, even when the server is at load.

The box plots in Figure 3.4 reveal that the median FTT for the DSA course in the MT semester is noticeably higher than in the non-MT semester, with a larger IQR. This indicates greater variability during load in the MT semester. The box plots for other courses show similar medians and IQRs in both semesters, suggesting that the introduction of MT had a minimal impact on the FTT of non-DSA courses during load.

Comparing the results from Figures 3.3 and 3.4, it is evident that the introduction of MT had a more pronounced impact on FTT during all server conditions than during load condition. The percentage increase in mean FTT for the DSA course was 139% during normal conditions, compared to 56% during load. This difference can be attributed to the fact that during load, the server is (by definition) experiencing a high volume of concurrent jobs, and the additional computational requirements of MT may have a relatively smaller impact on the overall FTT since now wait time plays a role in the calculation.

Our comprehensive analysis of FTT across non-MT and MT semesters reveals significant insights into the impact of introducing MT in place of CC for a DSA course within an AAT environment (Web-CAT). The implementation of MT resulted in a significant increase

Table 3.3 Feedback Turnaround Time (FTT) metrics for DSA and non-DSA courses during MT and non-MT semesters, comparing the performance of the Web-CAT server when under overall vs. load conditions.

	FTT (seconds) at Server, All Conditions		FTT (seconds) at Server, Load Condition	
	mean	median	mean	median
MT DSA course	67	52	134	118
MT other courses	17	15	81	36
non-MT DSA course	28	25	86	37
non-MT other courses	25	17	83	26

in FTT for the DSA course, with a 139% rise during normal server conditions and a 56% increase during load conditions. Contrary to initial concerns, the introduction of MT in the DSA course did not significantly impair the performance of other courses using the same AAT. Only a marginal portion of the submissions from other courses observed some delay in FTT during load conditions. This finding indicates that the AAT's resource allocation mechanisms effectively mitigated potential negative impacts on non-DSA courses.

While MT offers enhanced testing capabilities for DSA courses, it comes at the cost of increased FTT for those specific courses. Based on our finding, the increased mean FTT of 67 seconds (at normal load) and 134 seconds (at load) is within the tolerable level to receive feedback on Web-CAT. Furthermore, the minimal impact on other courses suggests that MT can be implemented in select courses without compromising the overall system performance for the broader student population. Yet instructors should be aware of this impact and consider strategies to mitigate the increased FTT, such as preventing spamming submissions, providing alternative feedback mechanisms, or allocating additional server resources.

There is a third component to actual perceived user feedback times (besides processing time for a job and wait time during load). This is the network time, both for the student's submission to reach the AAT, and later for the feedback to travel over the network and

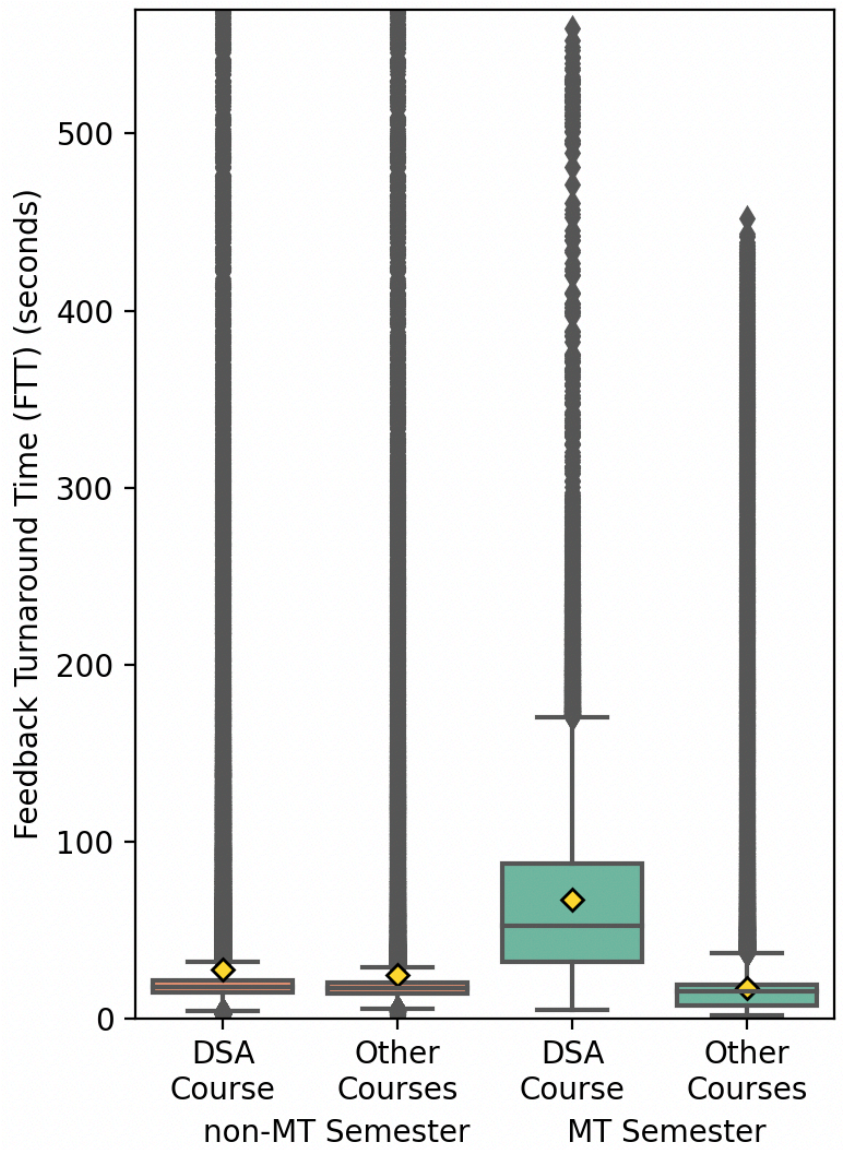

Figure 3.3 The distribution of feedback turnaround times (FTTs) for all courses running on Web-CAT in the non-MT semester (left) and MT semester (right). The MT semester exhibits a significant increase in feedback turnaround time (FTT) (27 seconds for the median time, or 139%), for DSA projects using MT. Note: The y-axis has been limited to 600 seconds for both figures to facilitate direct visual comparison. As a result, some outliers extend beyond the visible range.

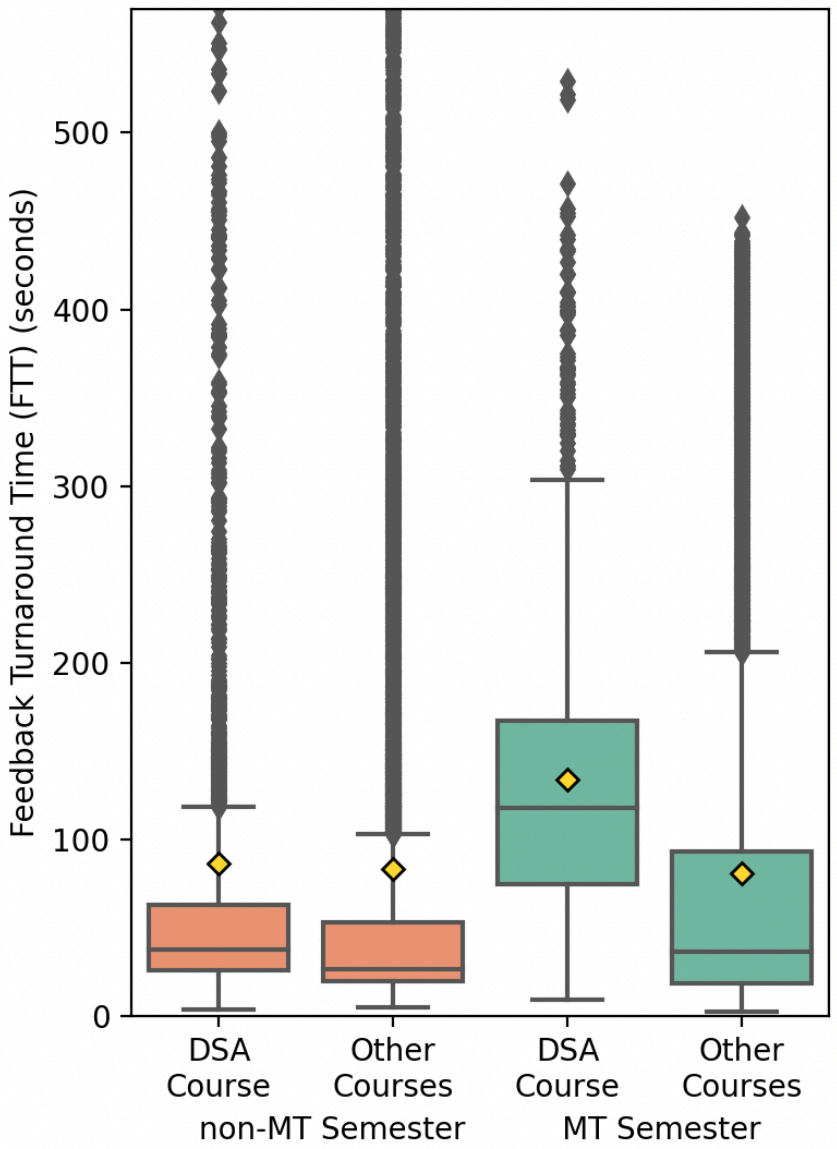

Figure 3.4 The distribution of feedback turnaround times (FTTs) for all courses running on Web-CAT at load condition in the MT semester (left) and non-MT semester (right). There is a statistically significant increase in FTT for DSA course in the MT semester, with an additional delay of approximately 48 seconds (or 56% longer than usual). However, note that only 4% of submissions experienced higher FTT delays due to load. Note: The y-axis has been limited to 600 seconds for both figures to facilitate direct visual comparison. As a result, some outliers extend beyond the visible range.

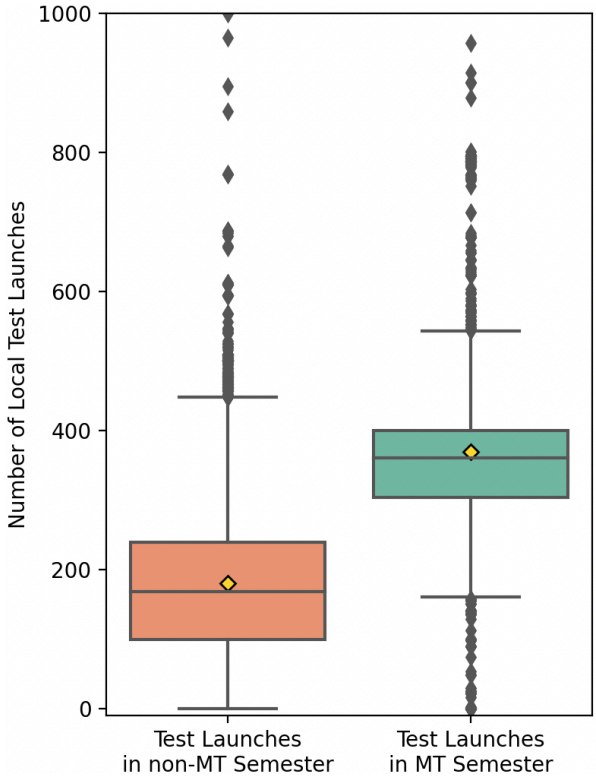

Figure 3.5 Frequency of student-initiated assessments of test quality, specifically local execution of code coverage (non-MT semester) and mutation testing (MT semester). Twice as many test coverage checks were made locally using Eclipse during the MT semester. This is significant ($p - value < 0.001$), indicating a clear and substantial shift in student behavior.

appear on the student's browser page. This varies depending on network speed and loading, but we find that this is typically about 15 seconds. Thus, the overall impact of MT on DSA students can be considered to be an increase of about 45 seconds, growing from a non-MT median time to receive feedback of about 30 seconds to a MT median time of about 75 seconds. Submitting during busy times will typically add about a minute to this.

3.2.3 RQ3: Impact of local MT use on Web-CAT submissions

We have for some years used an Eclipse plug-in to make the student experience within the IDE to be as close as possible to that encountered on Web-CAT [38]. We updated this

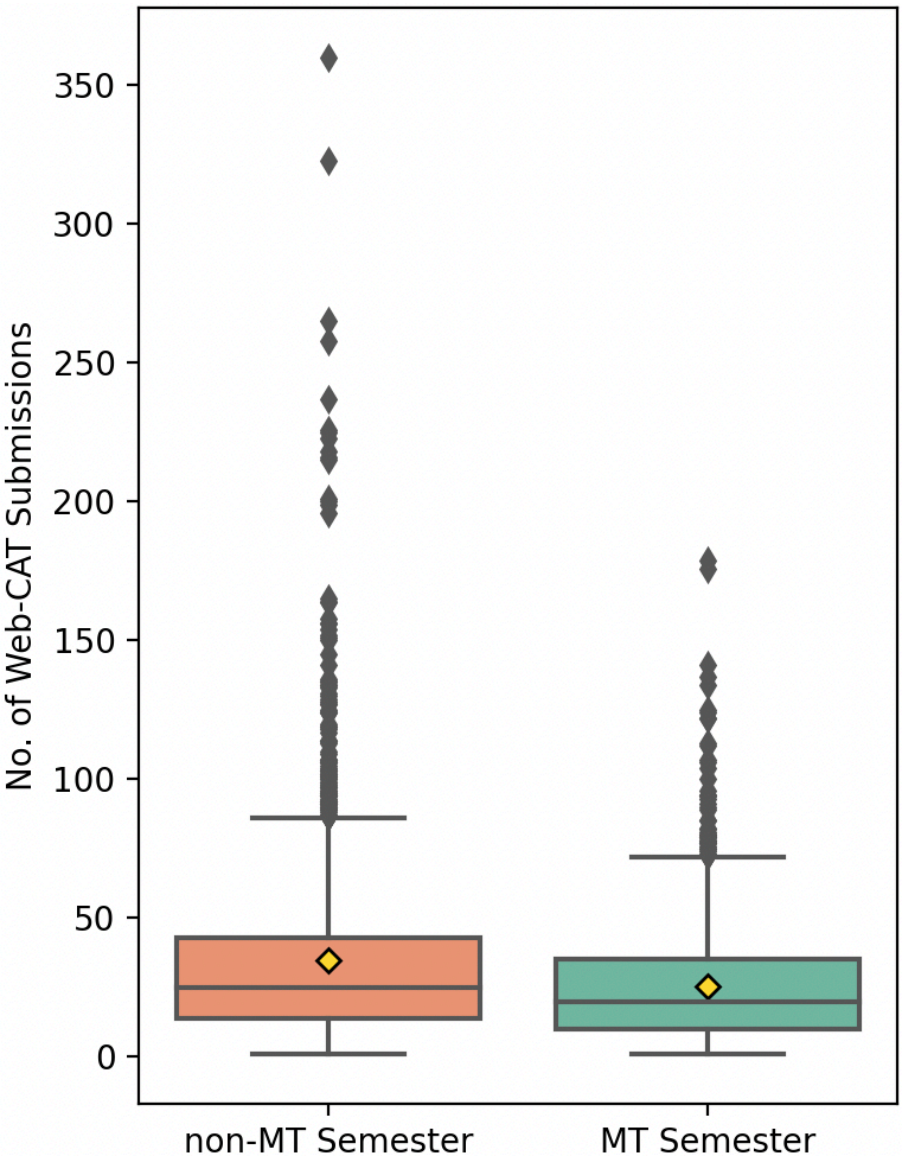

Figure 3.6 Number of submissions made to the Web-CAT by each student during non-MT semester (left box plot) and MT semester (right box plot). There is a statistically significant difference between the two groups, with students in non-MT semester exhibiting a higher tendency to submit to Web-CAT ($p{-}value << 0.001$). The number of submissions made by students in non-MT semester is approximately 40% greater than by their MT semester counterparts.

plug-in to support MT when the MT requirement was introduced. One hoped-for result is that the ability to run MT locally will reduce the number of times that students submit their project to Web-CAT. To further investigate whether providing a client-side MT execution method affects the number of submissions students make to Web-CAT, we analyzed server logs from two semesters, Fall 2021 (non-MT) and Spring 2024 (MT). We focused on two key metrics: the number of local test event launches by students and the number of student submissions made to the Web-CAT server for a single project.

The comparison of local test launches between the non-MT and MT semesters reveals a significant increase in the number of times students executed tests locally when MT was introduced. In the non-MT semester, the mean number of local test launches per student was 181 (SD = 116.41), with a median of 169, as seen in Figure 3.5. The minimum number of launches was 0, and the maximum was 1011. The interquartile range (IQR) was 139.50, with the 25th percentile at 100 and the 75th percentile at 239.

In contrast, during the MT semester, the mean number of local test launches per student increased to 369 (SD = 177.97), representing a 104% increase compared to the non-MT semester. The median value was 361, and the IQR was 96, with the 25th percentile at 303 and the 75th percentile at 400. The minimum number of launches remained at 0, while the maximum increased to 1773.

These results suggest that the introduction of MT in the MT semester had a substantial impact on students' local testing behavior. The higher mean and median values indicate that students were more engaged in running tests locally when MT was available. The increased local testing activity could be attributed to the enhanced feedback provided by MT, which encourages students to develop more robust test cases to detect potential faults in their code.

The analysis of student submissions to the Web-CAT server for a single project reveals a different trend compared to the local test launches. For our data analysis, we discarded student groups who scored zero on the project score. In the non-MT semester, the mean number of submissions per student was 35 (SD = 35.47), with a median of 25, as seen in Figure 3.6. The minimum number of submissions was 3, and the maximum was 360. The IQR was 29, with the 25th percentile at 14 and the 75th percentile at 43.

During the MT semester, the mean number of submissions per student decreased to 24 (SD = 22.56), indicating a reduction in the number of times students submitted their work to the Web-CAT server. The median value was 20, and the IQR was 25, with the 25th percentile at 10 and the 75th percentile at 35. The minimum number of submissions remained at 3, while the maximum decreased to 179.

To determine the statistical significance of the observed differences in Web-CAT submissions between the two semesters, we conducted a Mann-Whitney U test. The test results showed $p-value =<< 0.05$, indicating a significant difference between the non-MT and MT semesters.

The decrease in Web-CAT submissions during the MT semester suggests that students were more confident in their code quality before submitting to the AAT. The availability of MT likely encouraged students to thoroughly test and refine their code locally, reducing the need for multiple submissions to the Web-CAT server. The enhanced local testing process, facilitated by MT, might have helped students identify and resolve issues earlier, leading to fewer iterations and submissions to the AAT.

It is important to note that while the decrease in Web-CAT submissions might initially seem counter-intuitive, it aligns with the goal of promoting better code quality and reducing the reliance on the AAT for debugging purposes. The increased local testing activity, as

evidenced by the higher number of local test launches, indicates that students were more proactive in testing and refining their code before submitting it for automated assessment. The use of MT also encouraged code with lower cyclomatic complexity, as seen in Figure **??**, which in turn might have affected the number of submissions needed to complete the project.

These findings provide valuable insights into the impact of introducing MT on students' testing behavior and submission patterns. The increased local test launches and decreased Web-CAT submissions during the MT semester suggest that MT encourages more thorough local testing and reduces the need for multiple submissions to the AAT. These findings highlight the potential benefits of incorporating MT in post CS2-level courses to promote better testing practices and code quality among students.

3.3 Start-up Problems

The results in Section 3.2.2 are quite positive, in that they show that there was no negative impact on overall server loads or feedback turn-around times due to the additional load imposed by the introduction of MT. However, we must point out that these data were collected in the Spring 2024 semester, which was actually our second major deployment of MT to the DSA course. After some initial testing in earlier semesters, our first major deployment of MT was actually in the Fall 2023 semester.

During this initial deployment, we encountered severe problems with server load. The primary cause of this appears to have been abusive over-use of submissions by a portion of the DSA students, who made many submissions of essentially the same version of the project in a short period of time. There is no pedagogical or technical benefit to doing this, and it indicates a failure to properly test, assess, and debug the program. It is not a novel phenomenon, and anecdotal evidence suggests that there might have been some congestion and

server slowdowns in prior semesters during peak submission times for the DSA course. But with the use of MT, sudden surges of submissions near deadlines did paralyze the server.

We ended up with two solutions to this problem. The first was to invoke use of a policy called *submission energy*. This is a policy that essentially limits students to three submissions of their project within any one-hour time period. Introduction of this policy immediately solved the congestion issues.

The second solution comes from recognition that, even in the face of such a surge of submissions, the server should have been better able to handle the load. The core issue here was that the submission surge exposed weaknesses in the scheduling algorithm used.

We have since implemented (in Fall 2024) the Highest Response Ratio Next (HRRN) scheduling algorithm in place of the previously used First-In-First-Out (FIFO) approach. HRRN is a non-preemptive, optimal algorithm that calculates a response ratio for each task, considering both waiting time and burst time. The response ratio is computed as follows:

$$\text{Response Ratio} = \frac{\text{Waiting Time} + \text{Burst Time}}{\text{Burst Time}} \tag{3.1}$$

By prioritizing tasks with higher response ratios, HRRN ensures fairer scheduling, reduces delays for shorter tasks, and minimizes the convoy effect. In the context of Web-CAT, this means that student submissions with shorter estimated processing times and those that have been waiting longer will be given higher priority, effectively reducing the overall FTT.

While deploying HRRN has likely improved scheduling for the server, we again note that this deployment was **not yet** in place during Spring 2024, the semester when our MT data were collected. In other words, submission energy alone turned out to be sufficient to solve the congestion problems. However, we find it instructive to note that introducing MT did expose various weaknesses in both technical and policy aspects of server administration. In

our view, MT did not **cause** negative impacts on the server. But it **exposed** problematic situations that were previously in place (one a policy issue, one a technical deficiency). Fixing those improved server performance as compared to prior semesters for all students, even with MT in effect. Again, we note that non-DSA FTT was reduced in the MT semester, which indicates that introduction of submission energy in the DSA course had a greater positive impact than the introduction of MT had a negative impact on load.

3.4 Discussion

The results of this study provide valuable insights into the impact of introducing MT into a post CS2-level large DSA course. The findings shed light on the additional processing time incurred by MT on the Web-CAT AAT server, the effect of MT on feedback turnaround time (FTT), and the influence of providing a client-side MT execution method on students' submission behavior.

3.4.1 RQ1: Additional Processing Time due to MT

The analysis of server logs from the non-MT and MT semesters reveals that the introduction of MT significantly increases the processing time of DSA project submissions on the Web-CAT AAT server. Under normal load conditions, the median processing time increased by 34 seconds, and the mean processing time increased by 44 seconds in the MT semester compared to the non-MT semester. This substantial increase in processing time can be attributed to the computational complexity of generating and evaluating mutants, which requires additional server resources.

The impact of MT on processing time of a DSA project submission is even more pronounced

when Web-CAT server is at load conditions. During load conditions, the median processing time increased by 58 seconds, and the mean processing time increased by 55 seconds in the MT semester compared to the non-MT semester. This amplified effect suggests that the server's performance is more sensitive to the computational demands of MT when it is already handling a high volume of concurrent jobs.

3.4.2 RQ2: Impact of MT on Feedback Turnaround Time

An important consideration in our analysis is the potential cascading effect of increased processing times of DSA project submissions on the overall efficiency and scalability of the Web-CAT server. Longer processing times from one course might engage the server workers, resulting in delayed feedback cycles, compromising the learning experience for students.

The analysis of feedback turnaround time (FTT) reveals that the introduction of MT has a significant impact on the time it takes for students in the DSA course to receive feedback on their submissions. During normal server conditions, the mean FTT for the DSA course increased up to 134 seconds in the MT semester compared to 86 seconds in the non-MT semester. This substantial increase in FTT can be attributed to the additional processing time required for MT, as discussed in RQ1. Although this increase is significant, this new FTT in the presence of running MT falls in the tolerable range with the added benefit of getting higher quality feedback. When considering the time required to deliver actual web pages to the user, the median time goes from about 100 seconds to about 150 seconds, which is noticeable but unlikely to have a major pedagogical impact, especially since students can run MT locally for quicker feedback.

Interestingly, the impact of MT on FTT varies between the DSA course and other courses. While the DSA course experienced significant increases in FTT during both normal and load

conditions, other courses did not experience any increase in FTT, particularly during normal server operation. We observe a small portion (4%) of the student submissions from other courses experience longer FTT in the MT semester.

3.4.3 RQ3: Impact of Client-Side MT on Student Submission Behavior

The analysis of local MT usage and Web-CAT submissions shows that with the presence of the MT plug-in, students run their test cases more frequently using MT compared to using only CC. Students not only improved their test suites, but also decreased the complexity of their code. This finding suggests that the availability of a client-side MT execution method can influence students' submission behavior.

By providing students with the ability to run MT locally, they can gain immediate feedback on the quality of their test cases and the robustness of their code. This local feedback loop encourages students to refine their code and test cases before submitting to the AAT, potentially reducing the number of iterations needed to achieve a satisfactory solution.

The comparison of student behavior across MT and non-MT projects further supports this notion. Students who consistently used the MT plug-in rely less on the feedback from Web-CAT. In contrast, students who did not make use of the local MT plug-in made significantly more submissions to Web-CAT for similar DSA projects. This suggests that the lack of a client-side MT execution method may lead to more frequent submissions to the AAT, as students rely on the server feedback to gauge the quality and correctness of their code.

Chapter 4

Impacts of MT on Student Performance

The integration of MT into CSEd presents a unique opportunity to examine its effects on student performance on programming project development. While CC has traditionally been the primary metric for evaluating test effectiveness in educational settings, its limitations in measuring actual fault detection capabilities have prompted the exploration of more sophisticated approaches [17]. MT offers a more rigorous method for assessing test suite quality by systematically introducing small faults into the code and evaluating whether student-written tests can detect these changes [15].

In this chapter, I seek to understand the influence of MT by comparing the differences found in non-MT and MT semesters by addressing the following three research questions:

- **RQ1.** How does our introduction of MT impact the quality of student-written test cases?
- **RQ2.** What is the effect of MT on the project solution code written by the students?
- **RQ3.** How does our introduction of MT impact students' behavior in terms of test case development?

My investigation into MT's impact on student performance centers around three fundamental

research questions that examine different aspects of student learning and development. The first research question explores how MT affects the quality of student-written test cases (RQ1). This investigation is crucial because previous research has shown that students often struggle with writing effective tests, particularly in more complex programming projects [18]. By comparing test suites written by students using MT versus those using traditional CC, I assess whether MT leads to more comprehensive and effective testing strategies.

The second research question examines MT's influence on students' solution code quality (RQ2). This aspect is particularly interesting because it goes beyond just testing practices to understand how the use of MT might influence students' overall programming approach. Prior studies have suggested that better testing practices can lead to improved code quality [67], but the specific impact of MT on student code has not been thoroughly explored. I analyze both the correctness and complexity of student solutions to understand this relationship.

The third research question investigates how MT affects students' test development behavior throughout the project life-cycle (RQ3). This behavioral aspect is critical because successful software development requires not just the ability to write tests, but also the discipline to write them consistently and systematically. Previous research in software testing education [27] has highlighted the importance of developing good testing habits early in students' programming careers.

In my study, conducted in the post-CS2 DSA course at Virginia Tech, I compared student performance across semesters with MT vs. with CC. The course structure, with its multi-week programming projects, provided an ideal environment for studying the long-term effects of MT on student development. The control group used traditional CC metrics for test assessment, while the experimental group used MT-based evaluation.

The analysis reveals several significant findings. Students using MT demonstrated improved test suite quality compared to those using CC alone. More importantly, MT's influence extended beyond just testing practices - students wrote more maintainable and correct code when using MT. This improvement appears to stem from MT's ability to encourage students to think more critically about potential edge cases and error conditions in their code. Furthermore, I observed that students using MT were more likely to adopt an incremental approach to testing, developing and refining their test cases throughout the project life-cycle rather than leaving testing until the end.

These findings have important implications for CSEd, particularly in how we teach software testing in undergraduate courses. The results suggest that MT, despite its computational overhead, offers significant pedagogical benefits that extend beyond just improving testing skills. The approach appears to foster better overall software development practices, preparing students more effectively for professional software development where testing is an integral part of the development process [56, 60].

4.1 Research Methodology

We have implemented a modified version of PIT with a selected set of mutators [37], 1) remove conditional statements and 2) arithmetic operator deletion, for our use in post-CS2 level courses. To evaluate the effectiveness of our MT tool for assessing the quality of student-written test cases, we compared it to additional metrics, including PIT with the full set of mutators, MuJava, and Major. This allows us to confirm that our limited set of mutation operators gives students feedback comparable to that given by state-of-the-art test quality metrics. We investigate the effects of MT on various aspects of student projects. The following sections outline the details of the participants, study design, data collection,

and analysis approaches.

4.1.1 Participants

Our research study involved students enrolled in a post-CS2 undergraduate Data Structures and Algorithms (DSA) course at our university, Virginia Tech. As part of the DSA course curriculum, students were tasked with completing four 3-4 week long programming projects in Java. They were required to develop their own test suites to confirm the functionality of their solution code. Two metrics were used to evaluate students' programming proficiency: 1) "project correctness," which measured the accuracy of students' solution code against hidden instructor reference test cases, and 2) "test coverage," which gauged the quality of students' test suites. We provided additional feedback on test quality by employing an Eclipse plugin that students could use on their local machines.

Data was collected from three semesters (Fall 2019, Fall 2020, and Fall 2021) that used code coverage (CC) as the graded test suite quality metric, and two semesters (Fall 2023 and Spring 2024) where mutation testing (MT) was used instead as the graded test suite quality metric. The students using MT were also informed of their code coverage metric. Students were assigned a set of projects each semester that involved implementing data structures and algorithms such as memory managers, buffer pools, various trees, external sorting, hash tables, and graphs. The set of projects assigned each semester aimed for similar collective difficulty.

To evaluate MT's impact on different project aspects, we chose four projects that were similar in both non-MT and MT semesters. We used four different comparable programming projects (which we identify as "Memory Manager," "Tree," "External Sorting," and "Graph") to compare the test suites in the final version of students' project submissions in non-MT

vs. MT semesters. To ensure a fair comparison, we verified that the cyclomatic complexity between the two sets of projects are similar. This ensured that these projects are equivalent in terms of complexity and difficulty level.

All student submissions from these projects across multiple semesters were gathered for both the experimental group (those using MT), and the control group (those using only code coverage). Students developed their projects within their local IDEs (such as Eclipse) and could submit to our online auto-grader, Web-CAT [19], multiple times to receive feedback on the correctness of their solution code and the coverage score of their test suite. To mitigate the computational cost, selective MT [37] was used to measure the coverage score in the semesters using MT. Each semester typically consisted of approximately 400 to 500 students, with over 10,000 student submissions for each project.

4.1.2 Study Design

Test Suite Quality: To evaluate the impact of MT on test suite quality, we conducted a retrospective assessment of mutation coverage – indicating the percentage of code effectively tested – across the control and experimental groups. We used both selective MT (identical to that used by the experimental group), and also a comprehensive set of all mutators from the PIT mutation testing tool to ensure thorough evaluation.

Additionally, to reduce potential bias associated with a single MT tool, we assessed test suite quality with two other widely used tools: Major [35] and MuJava [42]. These tools employ distinct algorithms and mutation operators, offering a more comprehensive assessment of the test suite's ability to detect various types of faults. We also measured and compared the number of test cases and asserts to assess the size of students' test suite.

Solution Code Quality: We investigated the effect of MT on project correctness by

analyzing mutation coverage metrics between the control and experimental groups. Higher mutation coverage can indicate a more robust and correct implementation, as the test suite is better able to catch faults.

Additionally, we used PMD [11], a source code analyzer that examines code quality metrics, such as cyclomatic complexity, to determine if MT had any impact on the overall quality of the students' code. Note that MT indirectly gives feedback on complex code (since mutants in such code cannot be killed), which might encourage writing simpler solution code as a way of improving the mutation coverage score.

Testing Activity: To investigate the effect of MT on students' testing activity, we collected various interactions within the IDE, such as code edits, test suite edits, local execution of code, test suite launch, etc. Using these IDE event-streams, we quantified incremental activity following the methodology outlined in Kazerouni, et al. [38]. We devised a new metric to differentiate between authentic and credit-seeking testing behavior among students, discussed in Section 4.2.3.

4.2 Results and Discussion

4.2.1 RQ1: Effects of MT on Student-written Test Cases

We analyzed the effect of introducing MT by comparing the quality of student-written test cases between semesters without MT (control) and those with MT (experimental). We used PIT with selective mutations (PIT-SM) in the MT semesters, whereas code coverage (CC) was used in the non-MT semesters. It is conceivable that our chosen metric, PIT-SM, is idiosyncratic and might be inadvertently steering students to optimize their test suites specifically for PIT-SM's mutation coverage, without genuinely enhancing the overall

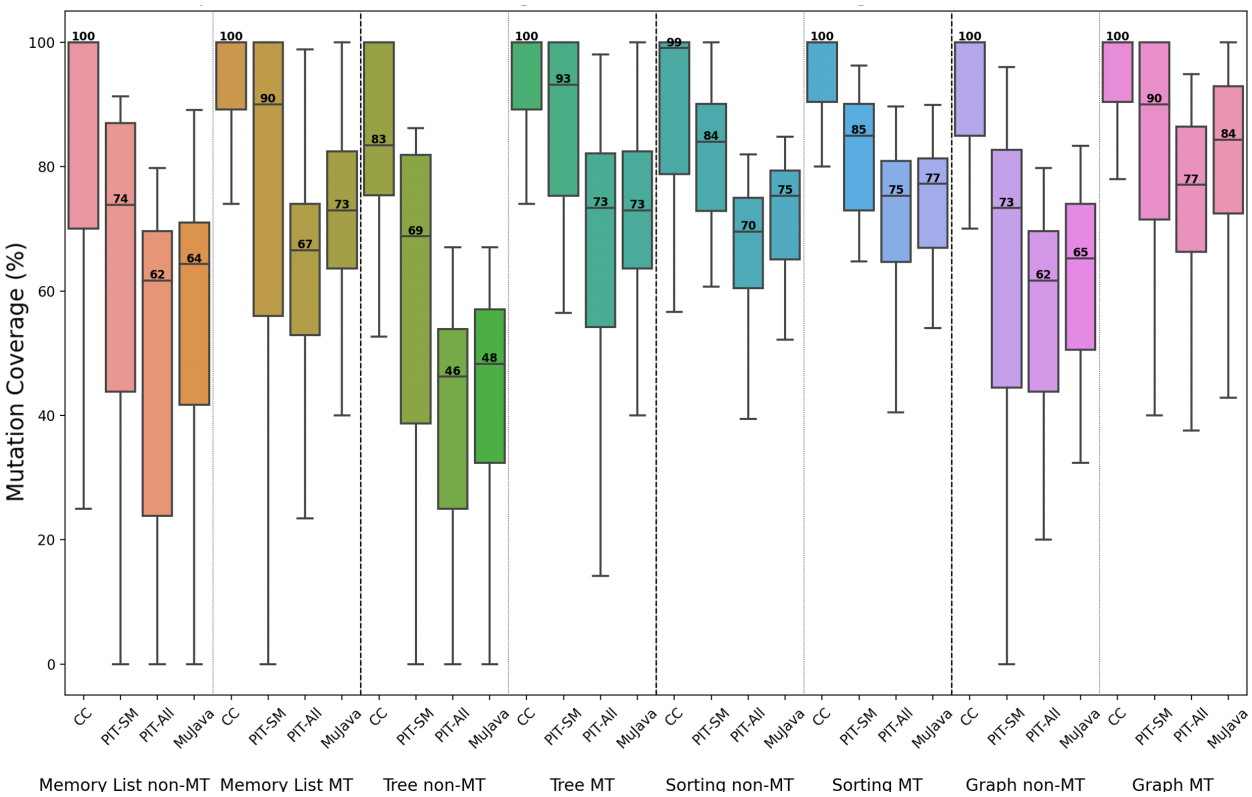

Figure 4.1 Comparing mutation coverage of students' test suites for four different comparable projects from non-MT and MT semesters. We used four different tools: Code Coverage (CC), PIT with selected mutators (PIT-SM), PIT with All mutators (PIT-ALL), and MuJava.

quality of their testing. So, to check for bias from our in-class metric, we compared five different metrics to assess the quality of students' test suites: code coverage (CC), PIT with selective mutations (PIT-SM)—the experimental condition, PIT with all mutations (PIT-ALL), Major, and MuJava. Based on our observation, we found that Major and MuJava returned similar coverage scores for our data. Therefore, we exclude Major from our results for brevity. We plotted our findings in the form of boxplots in Figure 4.1.

Our analysis revealed that across all four projects the test suite coverage, as measured by CC, PIT-SM, PIT-ALL, and MuJava, is consistently higher in MT semesters compared to non-MT semesters. This pattern holds true for both median and mean values. Among these, MuJava is the most harsh and discriminating metric. It is also recognized as the standard metric within the MT community (after PIT, which we exclude since we want an independent metric from that used in our experimental condition). Consequently, we opted to use MuJava for comparing test suite coverage between MT and non-MT semesters, as presented in Table 4.1. Figure 4.1 uses boxplots to visualize the median test coverage values, while Table 4.1 displays the mean test coverage between non-MT semesters and MT semesters, as quantified by MuJava for each project.

Before selecting the statistical tests to use, we checked the assumptions of normality and equal variance in our data. The Shapiro-Wilk test was used to assess the normality of four measures in students' final submissions: 1) test coverage, 2) project correctness, 3) code complexity, and 4) incremental testing score. All four data sets violate the normality assumption (p-values < 0.05). Additionally, Levene's test was applied to examine the assumption of equal variances for these four metrics across non-MT and MT semesters. The test results revealed significant differences in variances (p-values < 0.05), indicating a violation of the equal variance assumption. Consequently, the use of parametric tests like Student's t-test and ANCOVA, which assume normality and equal variances, was deemed inappropri-

Table 4.1 Analyzing differences in students' final solution code: comparison of four comparable projects in MT and non-MT semesters using Kruskal-Wallis with Dunn's post-hoc test with Bonferroni correction. All differences in means are significant, with p-values of 0.001 or lower.

Program--ming Projects	MuJava Test Set Quality (%)		Solution Code Correctness (%)		Solution Code Complexity		Incremental Test Writing	
	non-MT mean	MT mean	non-MT mean	MT mean	non-MT mean	MT mean	non-MT mean	MT mean
Memory Manager	60	74	70	82	130	102	4.32	1.22
Tree	46	73	66	85	220	119	4.81	1.12
External Sorting	72	79	69	86	121	77	3.81	1.80
Graph	62	81	68	83	197	152	4.45	1.23

ate for our dataset. Instead, we opted for non-parametric tests. To assess the statistical significance of the coverage difference between MT and non-MT semesters, we conducted the Kruskal-Wallis test, followed by Dunn's post-hoc analysis. We applied the Bonferroni correction to adjust the post-hoc test's p-values, ensuring strict control over Type I errors in non-MT versus MT comparisons. The Kruskal-Wallis test results, as shown in Table 4.1, indicate that the difference in test suite coverage between MT and non-MT semesters is statistically significant across all projects. These findings strongly suggest that students using MT achieve higher test suite coverage compared to students not using MT.

We also observed a notable increase in the size of the test suite during MT semesters compared to non-MT semesters (not shown in the table), including an increase in the number of test cases, assert statements, and lines of test code among students, with statistical significance (p-value $= 0.0001 < 0.05$).

We conclude that **introducing mutation testing (MT) in programming courses significantly improves student test suite quality compared to code coverage**.

4.2.2 RQ2: Effects of MT on Project Solution Code

Correctness Scores

We investigated the impact of using MT versus code coverage on the correctness of student programming projects. Project correctness was determined by assessing the percentage of the weighted value of the instructor's hidden test cases that were successfully passed by the students' solution code. Figure 4.2 reveals a distinct improvement in the distribution of correctness scores for the final submitted version of students' solution code for the experimental group across all projects. In Figure 4.2, we plotted the normalized correctness scores on students' final submission code for the MT and non-MT semesters for four comparable projects. The x-axis represents the normalized student final submission sorted by correctness scores, and the y-axis represents the correctness scores. The plot shows that the overall correctness scores for the MT semester are higher than those for the non-MT semester. Applying Kruskal-Wallis with Dunn's test with Bonferroni correction, we observed a significant difference in the correctness score of students' final solution code between MT semesters and non-MT semesters, as shown in Table 4.1. For the Memory Manager project, the MT group had a correctness score median of 95%, mean 82%, while the non-MT group's median score was 79%, mean 70%. For the Tree project, the MT group's correctness median score was 92%, mean 85%, while the non-MT group's median score was 75%, mean 66%. In the External Sorting project, we observed the most improvement, from a median of 81%, mean 69% in non-MT semesters to a median of 95%, mean 86% in MT semesters. For the Graph project, the MT group (median 88%, mean 83%) outperformed the non-MT group (median 78%, mean 68%). This consistent pattern across all four projects reinforces the finding that **MT leads to improved correctness in student solution code.**

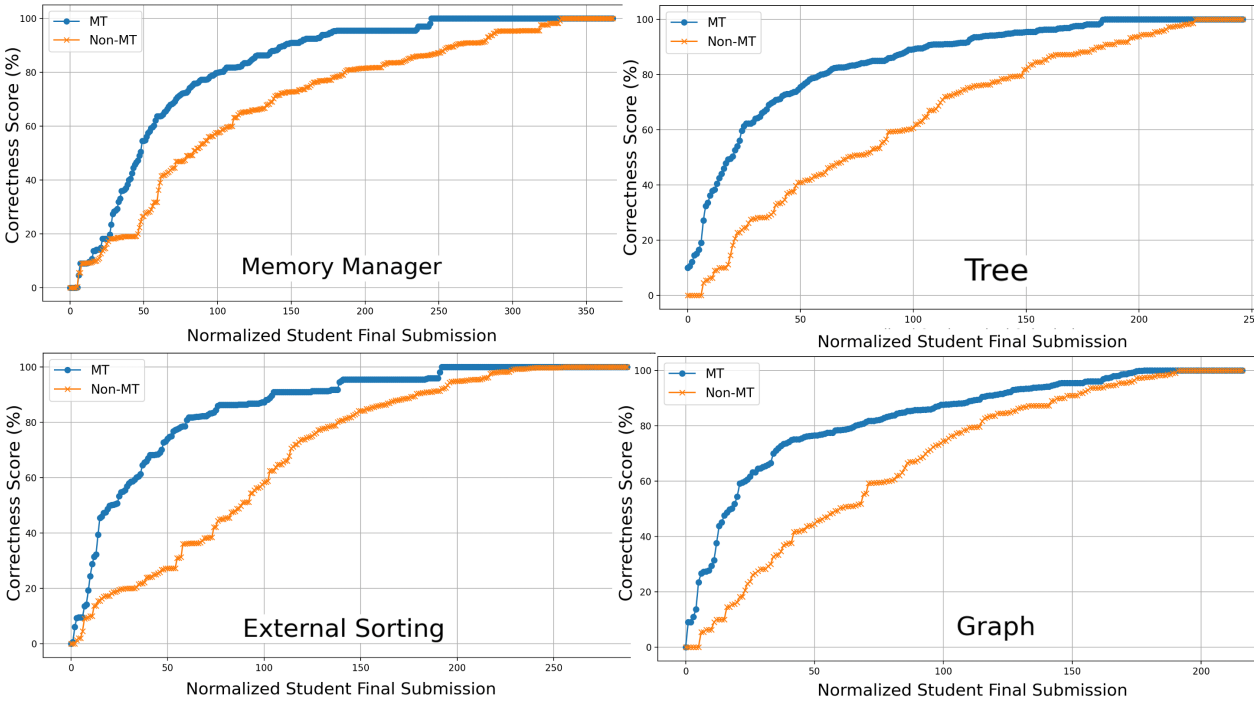

Figure 4.2 Comparison of correctness scores of students' final submissions for four different comparable projects for non-MT and MT semesters.

Cyclomatic Complexity of Solution Code

To evaluate the quality of students' solution code, we used cyclomatic complexity as a metric at the Java class level. Cyclomatic complexity [46] is used to assess program stability and confidence by quantifying the number of independent paths through its source code. Figure 4.3 depicts the normalized cyclomatic complexity values for the MT and non-MT semesters across four comparable projects. The plot illustrates that the cyclomatic complexity values are lower in the MT semester compared to the non-MT semester. We hypothesize that to improve mutation coverage, students not only write improved test cases but also develop more testable code, thereby resulting in less complexity and higher-quality solution code. We found that for all four projects, the difference in cyclomatic complexity between MT and non-MT semesters is statistically significant, see Table 4.1. We saw a notable disparity in total (Java class-level) mean cyclomatic complexity values between non-MT and

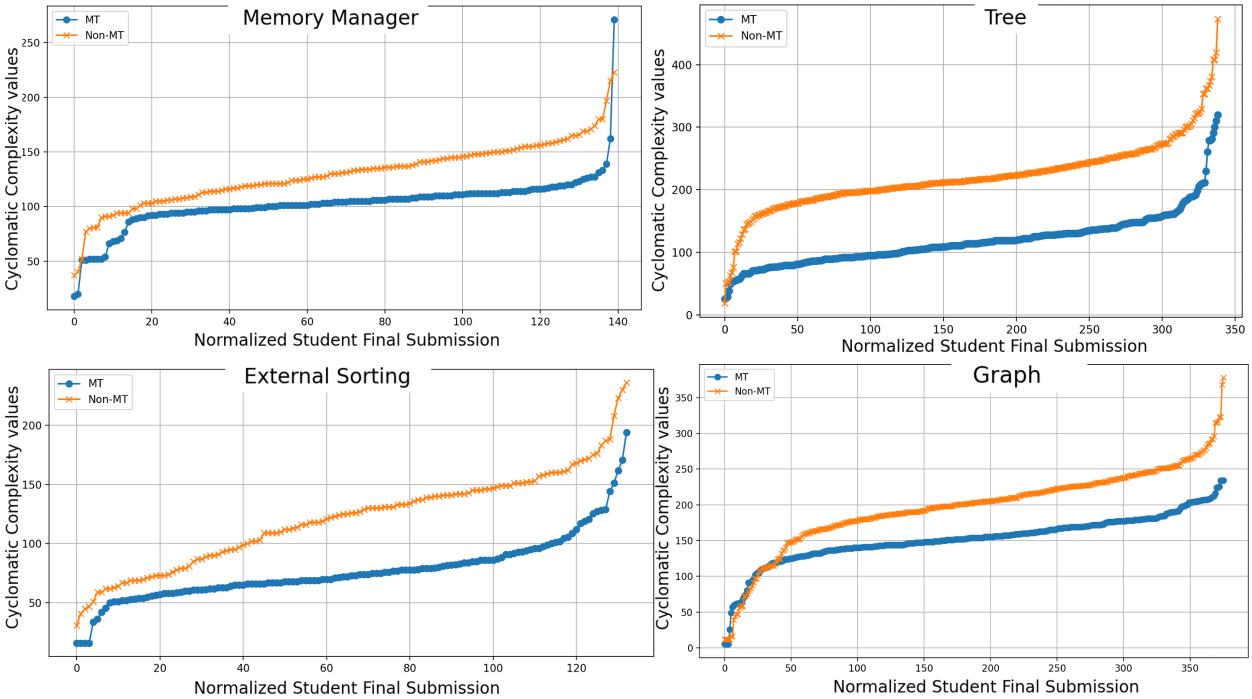

Figure 4.3 Distribution of cyclomatic complexities for student-written solution code (sorted low to high) for four projects across MT and Non-MT semesters. Lower cyclomatic complexity scores are considered better.

MT semesters across four projects: Memory Manager (130 vs. 102), Tree (220 vs. 119), External Sorting (121 vs. 77), and Graph (197 vs. 152).

Our results suggest that **MT leads to higher quality code by encouraging students not only to write better tests but also to write more testable code**.

4.2.3 RQ3: Effects of MT on Testing Activity

Incremental Test Writing

We seek to know if students using MT are more likely to write tests incrementally while developing their solution. To quantify this behavior, we applied the incremental test writing metric as defined by Kazerouni, et al. [38].

The "earlyOften" metric generates a weighted average considering the frequency of student

activity and the proximity to project deadline. Subsequently, the "incTestWriting" metric combines earlyOften indices for both test code and solution code. It represents the disparity between the average time spent on solution edits and test edits. This metric assesses the degree of incremental testing, where smaller values suggest greater alignment between test editing and solution editing, while a larger value indicates that test editing typically occurs later in the development process, after the majority of solution code has been written. This metric focuses solely on evaluating the regularity of test writing throughout the project life-cycle.

For non-MT semesters, the incremental test-writing metric gave no compelling evidence of a relationship with project correctness (p-value $= 0.1 > 0.05$). In contrast, during MT semesters, we found significant positive correlation between project correctness and incremental testing (p-value $= 0.003 < 0.05$). We observed that students' test suites in MT semesters exhibited smaller incremental test metric values (meaning more incremental test writing) compared to those in non-MT semesters, indicating that students using MT tended to update their test cases for newly developed solution code. We observed significant differences (p-value < 0.05) in the incremental test metric for students' test suites between MT and non-MT semesters across all four projects, as illustrated in Table 4.1.

Authentic vs. Credit-Seeking Testing

To determine if MT encourages students to engage in authentic testing practices during project development, we first need to establish an empirical method to characterize and identify differences in students' testing behaviors. This characterization is partly influenced by the specific grading criteria used in the course. In our case, our automated grading system withheld feedback on the number of reference tests passed or failed until a student's submission met a minimum test coverage threshold. For both non-MT and MT semesters,

this threshold was set at 75% test coverage. Subsequently, to receive full credit on their project score, students needed to exceed a 90% mutation coverage threshold. This grading structure raises an intriguing question: *Do students merely do testing to meet the thresholds for obtaining feedback or full credit, or do they appear to be genuinely using testing to aid their development process?*

We refer to the former as "credit-seeking testing"—where students test primarily to get the test suite grade. In contrast, we define "authentic testing" as writing tests with the intent to assist in debugging. With this background, we can empirically define the behaviors as follows.

Post-correctness Test Case Writing: Ideally, a student will strike a balance between writing effective solution code (correctness) and a comprehensive set of test cases (test coverage), the two grading criteria. When a student achieves a perfect score for code correctness, it is reasonable to assume that their solution code is complete. Writing additional test cases then becomes about boosting test coverage credit, not testing the accuracy or completeness of their solution. Hence, we consider this behavior as credit-seeking testing. We formulated a metric named Post-Correctness Test Case (PCTC) to detect such behavior in students' testing.

$$PCTC = \frac{L_p}{L_{total}}$$

Here, L_p is lines of test cases written after achieving perfect correctness, L_{total} is total lines of test cases written.

We examined each incremental submission a student made to the auto-grader and calculated the number of test case lines added after they achieved a perfect correctness score. Since these students have already attained perfect correctness, any additional test cases serve

primarily to increase their credit for test case writing rather than for authentic testing aimed at debugging. A higher PCTC value indicates a tendency towards credit-seeking behavior, as it shows that a significant proportion of test cases are written after perfect correctness is achieved.

To identify instances of credit-seeking testing, we employed our PCTC metric to compare non-MT and MT semesters. We found that in non-MT semester, 10.1% of students exhibited credit-seeking testing behavior. This proportion is significantly higher (p-value $= 0.000132 < 0.05$) than in MT semesters, where only 2.8% of students displayed such behavior. This suggests that there may be factors or conditions specific to MT semesters that lead to a lesser prevalence of credit-seeking testing behavior among students.

Our analysis suggests that **MT encourages students to proactively and incrementally write test cases throughout the entirety of their project development process**.

Chapter 5

Improving MT Feedback

The effective integration of MT in CSEd requires not just the implementation of the testing framework itself, but also the development of appropriate feedback mechanisms that students can understand and effectively utilize. While MT has proven valuable in industry settings [56, 60], its direct application in educational contexts presents unique challenges that demand careful consideration of how feedback is presented and interpreted by students.

In this chapter, I present my research into MT feedback development through three distinct phases, each addressing specific aspects of student interaction with testing methodologies. The first phase examined students' existing testing practices and perceptions when using CC. Understanding this baseline was crucial for developing effective MT feedback mechanisms, as it revealed how students traditionally approached testing and what they valued in feedback systems. Previous research has shown that students often struggle with interpreting testing feedback [18], making this foundational understanding essential for developing more effective feedback mechanisms.

The second phase of my research investigated students' experiences with standard MT feedback provided by PIT [10], a widely-used Java MT tool. This investigation was critical because industry-standard tools, while powerful, are typically designed for professional developers rather than students. The feedback from these tools often assumes a level of testing knowledge and experience that undergraduate students may not possess. As highlighted by Jia and Harman [32], the complexity of MT output can be overwhelming, particularly for

novice programmers.

The third phase of my research focused on the development and evaluation of modified MT feedback mechanisms. This work built upon the insights gained from the previous phases, addressing the specific challenges and needs identified in student interactions with both CC and standard MT feedback. I developed specialized documentation, training materials, and feedback formats designed to bridge the gap between students' current understanding and the sophisticated feedback provided by MT.

My investigation was guided by three research questions that traced the evolution of my MT feedback development. I seek to address the following research questions.

- **RQ1.** What are students' perceptions of the value of writing test suites and their testing behaviors when using CC in programming projects?
- **RQ2.** What are students' performance and their impressions of using PIT's stock MT feedback for evaluating their test suites?
- **RQ3.** What impact does modified MT feedback and targeted MT instructional resources have on students' testing perception, practices, and outcomes?

The first research question examined how students perceived and utilized testing feedback in their traditional CC-based environment (RQ1). This understanding was crucial for identifying the specific areas where MT feedback needed to be adapted for educational use. The second research question evaluated students' responses to standard MT feedback, helping us understand the specific challenges they faced when interpreting industry-standard MT output (RQ2). The final research question assessed the effectiveness of my modified MT feedback approach, examining how my specialized feedback mechanisms and supporting materials improved students' testing practices and understanding (RQ3).

Through multiple semesters of implementation and refinement, supported by 38 semi-structured student interviews and extensive quantitative analysis of student submissions, I developed a comprehensive framework for presenting MT feedback in educational settings. This framework not only addresses the immediate needs of students learning to test their code but also helps prepare them for the more sophisticated testing practices they will encounter in professional settings [45].

My findings demonstrate that carefully modified MT feedback, combined with appropriate supporting materials and training, can significantly improve students' understanding and application of testing principles. This improvement extends beyond just test suite quality to influence overall code development practices, suggesting that appropriate feedback mechanisms play a crucial role in realizing the full educational potential of MT.

5.1 Research Approach

Our research focused on students enrolled in a post-CS2 undergraduate Data Structures and Algorithms (DSA) course at [ANONYMOUS] University, an R1 institution in the US. As part of the DSA course curriculum, students complete four 3-4 week long programming projects in Java. This remained true throughout the various iterations of the course reported on here.

Our effort to replace CC with MT for test suite evaluation evolved organically over three semesters, each implementing different approaches to software testing. During our control semester we used code coverage (CC) metrics as the primary method for evaluating students' test suites. During this semester we also interviewed students about their testing practices and solicited their feedback about the prospect of using MT instead of CC.

For the first MT semester, we transitioned to MT using PIT, an open-source tool [10], which we modified to use a selective set of two mutators. During the course of this semester we received anecdotal feedback from students that they were struggling to understand MT, and their overall performance on the earlier projects did not improve over that of CC semester.

Based on student feedback (including interviews that we discuss in this paper), we refined and customized PIT's standard feedback to better align with student learning needs. In the second MT semester, we implemented this modified feedback along with additional instructional materials. We then collected both qualitative and quantitative data to evaluate students' experiences with our improved feedback system.

We use identifiers for the semesters as follows:

- **Code coverage (CC) semester:** Students used CC, a familiar tool from their CS1 and CS2-level courses, to assess test suite quality. This serves as our control (it is a continuation of many years of practice).
- **Mutation testing (MT_{ps}) semester (with PIT's stock feedback):** We introduced MT using a sub-set of mutation operators along with stock PIT feedback. While there was some instruction on the use of MT, it appears to have been inadequate for proper understanding.
- **Mutation testing semester (MT_{mf}) (with modified feedback and additional instructional materials):** We replaced the stock PIT feedback with improved MT feedback along with additional instructional resources to help students better understand and use MT in their project development.

Our study of the student experience and project results employs a mixed-methods approach combining qualitative and quantitative data collection and analysis, following [12]'s method-

ology. This approach provides both a broad population view through quantitative analysis and detailed individual insights through qualitative analysis. The study uses a convergent parallel mixed-methods design with three phases, collecting and analyzing three independent sets of qualitative and quantitative data.

5.1.1 Programming Assignments and Assessment Rubrics

Students in our DSA course are assigned a set of projects each semester that involve implementing data structures and algorithms such as memory managers, buffer pools, various trees, external sorting, hash tables, and graphs. To evaluate MT's impact on different project aspects, we chose collections of projects that were similar in across all three semesters. To ensure a fair comparison, we verified that the cyclomatic complexity of the solutions across the three sets of projects are similar. The goal was to make these projects equivalent in terms of complexity and difficulty level.

For many years we have evaluated students' programming proficiency using two metrics: 1) "project correctness," which measured the accuracy of students' solution code against hidden instructor reference test cases, and 2) "test coverage," which gauged the quality of students' test suites. Students receive feedback on test quality both from our automated assessment tool (AAT) when they submit their project for evaluation, and locally through an Eclipse plug-in.

In both the MT_{ps} and MT_{mf} semesters, we set a target mutation coverage of 90%, rather than aiming for the full 100%. This decision was consistent with our approach during previous CC semesters, where we adopted a similar target for the same reasons. Specifically, we wanted to maintain a balanced focus between testing and other key aspects of programming, such as implementation and problem-solving. Achieving near-complete mutation coverage—

particularly the last few percent—often requires a disproportionate amount of effort, which could divert attention from other important learning objectives. By setting a more achievable target, we ensured that students could focus on a broader range of skills without getting overly fixated on reaching the maximum mutation coverage.

We also implemented a threshold-based feedback system where instructor reference test cases remained hidden until students achieved 75% mutation coverage on their own test suites (which we consider fairly easy to achieve with any reasonable effort to write tests). This mechanism encouraged students to test their code before accessing additional guidance from the AAT "oracle." Once surpassing the 75% threshold, students received feedback on which of the instructor's test cases their code passed or failed, along with hints on why some of the test cases were failing. This approach, consistent with practices from previous years and across CC semesters, helped promote independent testing skills while offering structured support for improvement.

Students developed their projects within their local IDE (Eclipse) and could submit to the online auto-grader, Web-CAT [19], multiple times to receive feedback on the correctness of their solution code and the coverage score of their test suite. To address the computational demands of MT on student submissions to the AAT, we implemented a selective set of two mutators as recommended by [37]. Each semester typically consisted of approximately 300 to 400 students, with over 10,000 student submissions to the AAT for each project.

5.1.2 Qualitative Research

Our qualitative analysis involves semi-structured interviews with students across all three semester groups, following the guidelines outlined by [2]. All interviews were conducted by the first author of this research to ensure consistency in the interview process and questioning

techniques following the protocol listed in Table 5.1. The semi-structured format allowed for flexible probing to better understand and clarify interviewees' experiences and observations. Interviewer could ask follow-up questions when needed for additional clarity beyond the main protocol questions. This approach, as [2] notes, ensures data quality and reliability by combining standardized main questions with flexible follow-up inquiries. All interviews took place in private offices to maintain confidentiality and ensure clear audio recordings.

Our interview protocols focused on:

- **Student experience:** Students were asked about their overall experience with the programming testing approach as taught in that semester.
- **Perceived difficulty:** Questions explored students' perceived difficulty levels when working with CC and MT.
- **Understanding of concepts:** Interviews aimed to assess students' understanding of key testing concepts, such as test suite effectiveness, test case writing strategies, and feedback mechanisms.
- **Testing Strategy:** Students were asked to explain their testing strategies to understand how they used different tools, what features worked for them, the challenges they faced, and their overall perception of the tools.

The semi-structured interview protocol was designed to be flexible, allowing researchers to adapt the conversation based on students' responses and interests while maintaining a consistent framework for data collection.

We applied the principles of Grounded Theory, as outlined by [24], to systematically generate categories and themes, ultimately deriving multiple insights from the interview transcripts. The transcripts were analyzed through open coding, with themes and categories refined and

Table 5.1 Semi-structured Interview Protocol

Section	Topics Covered	Duration
Introduction	◇ Study objectives overview ◇ Informed consent and confidentiality agreements	5 minutes
Background Information	◇ Student programming experience ◇ Prior testing concepts knowledge	5 minutes
Testing Strategy	◇ Strategies used for testing ◇ Perceived difficulties with testing tools	15 minutes
Understanding Key Concepts	◇ Improving Test Suite Coverage ◇ Role of feedback in testing tools	15 minutes
MT Experience	◇ Understanding of MT principles ◇ Perceived challenges with MT	15 minutes
Closing Reflections	◇ Final thoughts and additional comments	5 minutes

Table 5.2 Interviews from three groups of students across three semesters.

Semester	Total Students	Students Interviewed	High Achieving	Average	Struggling
CC: Code Coverage Semester	328	12	4 ID: CCH(1-4)	5 ID: CCA(1-5)	3 ID: CCS(1-3)
MT_{ps}: Mutation Testing Semester (PTI's Stock Feedback)	346	15	5 ID: MPH(1-5)	6 ID: MPA(1-6)	4 ID: MPS(1-4)
MT_{mf}: Mutation Testing Semester (modified Feedback and Additional Instructions)	381	11	3 ID: MMH(1-3)	5 ID: MMA(1-5)	3 ID: MMS(1-3)

validated through multiple rounds of review. We retained only those connections between categories that were consistently supported by the data. Grounded Theory served as the analytical framework for interpreting the results, complementing the principles of design science. This combined approach provided a holistic method for synthesizing our findings across three semesters of data collection.

We conducted interviews with a total of 38 students: 12 students from the CC semester, 15 from the MT_{ps} semester, and 11 from the MT_{mf} semester, as shown in Table 5.2. Using stratified random sampling, we selected participants based on their programming project implementation and midterm exam performance, dividing them into three groups: high

achieving, average, and struggling. The high-achieving group (top 15% of the class) typically demonstrated project scores of 100% and midterm exam scores of 95% or higher. When analyzing their testing practices, we observed their mutation coverage typically exceeded 90%. The average performance group (next 50% of the class) maintained project scores above 85% and midterm scores ranging from 80% to 95%. This group's mutation coverage generally fell between 85-90%. The struggling group (bottom 35% of the class) showed project scores between 60-85% and midterm performance between 40-80%. Their mutation coverage typically ranged from 75-85%. This ensures representation from all performance levels, in an attempt to avoid skewing of data that could occur if random sampling disproportionately selected students from any one achievement level. This is particularly important because students at different performance levels may experience and interact with our testing tools differently. We wanted to collect distinct insights and experiences across different achievement levels.

In this paper, we use a structured system of anonymized identifiers. This is to ensure confidentiality and maintain anonymity while also preserving meaningful information about each student's academic performance and the semester in which they participated. The first part of the identifier represents the semester in which the participant was enrolled. Each group is denoted by a two-letter abbreviation: 'CC' for students in the CC semester, 'MP' for students in the MT_{ps} semester, and 'MM' for students in the MT_{mf} semester. The second part of the identifier indicates the participant's academic performance level, which is categorized into three levels: H for high-achieving students, A for average-performing students, and S for struggling students. The final part of the identifier is a number that differentiates between individual participants within each group. For example, CCH1 is a high-achieving student from the CC semester, whereas MPA2 is an average student from the MT_{ps} semester, and MMS3 is a struggling student from the MT_{mf} semester. Table 5.3 presents demographic information of student interviewees across the three semesters.

Table 5.3 Demographics of the Student Interviewees.

Semester	Participants	Male	Female	CS Major	CS Minor
CC	12	6	4	10	2
MT_{ps}	15	8	7	13	2
MT_{mf}	11	7	4	8	3

5.1.3 Quantitative Research

Collection of Student Submissions: Along with qualitative analysis, which provided insights into student experiences and behaviors, we also performed quantitative assessment to empirically evaluate the impact of MT on test suite quality. Specifically, we conducted a retrospective assessment of mutation coverage – indicating the percentage of code effectively tested – across CC, MT_{ps}, and MT_{mf} semesters. To reduce potential bias associated with retrospectively assessing test quality with the same tool that students used to develop their tests, we here assess test suite quality with MuJava by [42], another widely used MT tool.

We investigated the effect of MT on project correctness by analyzing mutation coverage metrics across all three semesters. We used PMD [11], a source code analyzer that examines code quality metrics such as cyclomatic complexity to determine if MT had any impact on the overall quality of the students' code. Cyclomatic complexity measures the number of linearly independent paths through a program's source code, indicating its complexity based on control flow [23]. We investigated whether MT improves students' code quality by ensuring that edge cases and complex paths (as indicated by cyclomatic complexity) are thoroughly tested, identifying weak spots in their code.

Collecting student activity logs: To investigate the effect of MT on students' testing activity, we collected various interactions within the IDE including code edits, test suite edits, local execution of code, and test suite launch. Using these IDE event streams, we quantified incremental activity following the methodology outlined by [38]. We applied the

testing behavior classification framework developed by [44] to distinguish between students who wrote tests to verify program correctness (authentic testing) and those who primarily aimed to achieve required coverage scores (credit-seeking testing).

Through qualitative analysis of student interviews, we identified several valuable insights on how students approached and understood different testing methods. We then validated these insights through quantitative analysis of student submissions and performance data. This allowed us to cross-reference our findings across both data sources and strengthen our understanding of testing approaches in programming education.

5.2 Code Coverage Semester

5.2.1 Description of CC Intervention

CC is one of the first methods invented for measuring test suite quality [47]. It measures the extent to which the source code of a program has been executed during testing. For many years we have used CC to assess our students' test suites (and to provide feedback about their test suite quality to students). Students' grades partially depended on achieving a specified CC threshold (typically in the 90-97% range). The CC of student test suites is assessed both at our AAT (for grading purposes) and also by the student's IDE (Eclipse) using the EclEmma Eclipse plug-in [29] so that students can know their test suite CC without needing to make a submission to the AAT. Both systems measured the percentage of code executed by test cases, highlighting untested sections through visual indicators. This dual-feedback approach enabled students to continuously assess and improve their test suite quality before final submission. Figure 5.1 demonstrates how students can visually assess which lines of their code are being executed by their test suite.

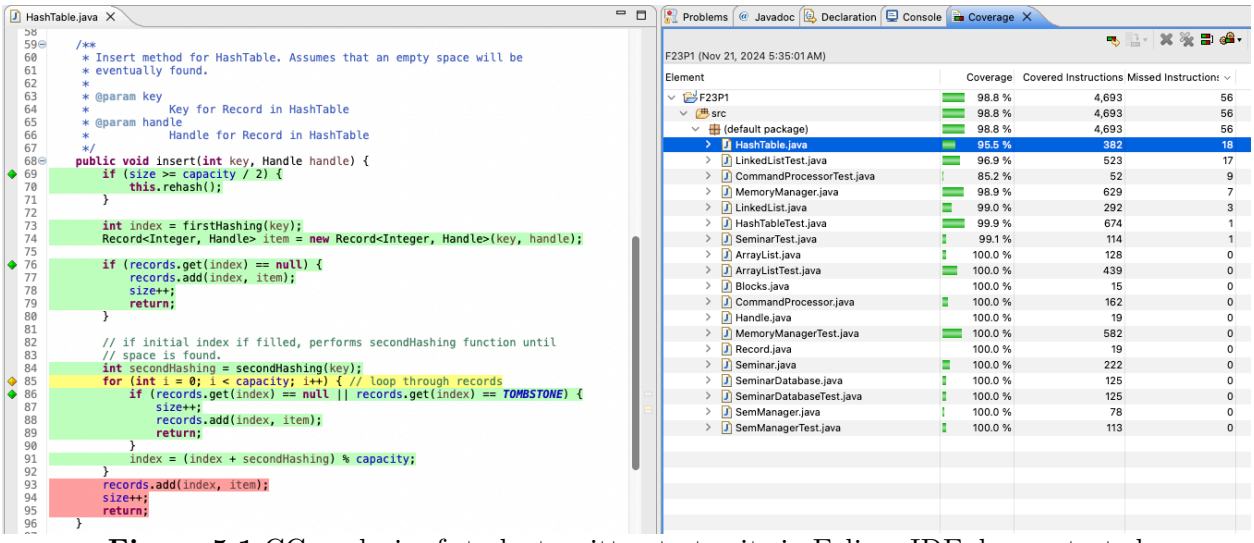

Figure 5.1 CC analysis of student-written test suite in Eclipse IDE demonstrated through EclEmma's color-coded interface and coverage statistics.

One problem with using CC in the context of grading for test suite quality is that it is too easy to abuse CC to gain coverage. Students may focus on increasing coverage to earn credit without properly verifying that the covered code is correct. This practice can influence students' perceptions of the value of writing test suites and achieving meaningful code coverage, as evidenced by the interview results discussed below.

5.2.2 Students' Impression of CC

To gain a deeper understanding of students' experiences with CC-based testing, we conducted semi-structured interviews with 12 participants following the CC semester. We also asked questions to get their perceptions related to a potential switch from CC to MT. Our qualitative analysis, using the Grounded Theory approach, revealed three key themes related to students' perceptions, testing behaviors, and the limitations of CC metrics.

Value Perception and Testing Behavior

Through our qualitative analysis, we identified a significant theme related to students' perceptions of value and their testing behavior in the context of CC-based testing. The categories that emerged within this theme suggest that students found limited value in CC-based testing, and their use of JUnit tests was primarily driven by the desire to obtain course credit rather than engaging in effective testing practices. 10 out of 12 students in the CC semester expressed that CC metrics did not provide substantial insights into the quality or effectiveness of their tests (Extract 1, Extract 2).

> *Extract* 1 (*CCA3*): "The test cases don't reveal too much. Most of the time passes. Rather print statements reveal the problem."

> *Extract* 2 (*CCH2*): "My tests passed most of the time so it wasn't that useful as a method of finding bugs."

Students' testing behavior was largely influenced by the course credit associated with achieving certain coverage metrics (Extract 3, Extract 4, Extract 5). This was reported by all 12 of our students participants.

> *Extract* 3 (*CCH4*): "JUnit is mostly used when its required, not really for debugging."

> *Extract* 4 (*CCH1*): "We made sure we had working code (according to Web-CAT) and not perfectly tested code."

> *Extract* 5 (*CCA2*): "It was hard to keep track of both of our progress. My partner worked on something and didn't actually write test code for it. I was writing

> test code for my parts. When we both finished, we had necessary coverage. So, we didn't write test code for his parts."

All 12 of our students mentioned that they prioritized meeting coverage requirements set by the automated grading system rather than focusing on developing comprehensive test cases. The focus on achieving coverage metrics over effective testing practices was a recurring theme in the students using CC (Extract 6, Extract 7).

> *Extract* 6 (*CCA1*): "Arbitrarily CC feedback is not that helpful."

> *Extract* 7 (*CCA5*): "For Project 2 we had more bugs; and we would realize that there were bugs when we submitted to Web-CAT, so we would realize that even though our tests were passing, we were clearly missing some edge cases."

Furthermore, one third of our students using CC expressed a general aversion towards writing test cases (Extract 8, Extract 9).

> *Extract* 8 (*CCS1*): "I hate writing tests to be honest."

> *Extract* 9 (*CCS3*): "Writing a test case for every single thing is a bit of a pain."

Based on our analysis, we derived this summary Insight 1:

Students using CC find little value in writing test cases for their code. Nor does requiring CC for a grade help students debug their programs.

Impact of Delayed Testing Practices

Our qualitative analysis revealed that delayed testing practices, particularly post-implementation testing, had a significant impact on students' debugging experiences and overall performance outcomes.

8 out of 12 students using CC only engaged in post-implementation system-level testing. These students reported a substantial increase in debugging effort (Extract 10, Extract 11).

> *Extract* 10 (*CCA2*): "After code ran for sample input and output we submitted to Web-CAT; that was when we spent the most time debugging."

> *Extract* 11 (*CCA4*): "No bugs in my local machine; when i submitted to Web-CAT, that's when I started having bugs."

The increased debugging effort associated with post-implementation testing often led to heightened frustration among students during the testing phase (Extract 12, Extract 13).

> *Extract* 12 (*CCH2*): "Trying to figure out what exact test case was breaking our code...we only thought of so many cases, but Web-CAT would give us an ambiguous hint and it was hard to figure out what test case was breaking."

> *Extract* 13 (*CCA1*): "...very time consuming to go to the TA for every new error and to get this information from Web-CAT."

The delayed testing approach also resulted in suboptimal performance outcomes for students. Several students (7 out of 8) acknowledged the benefits of incremental testing practices, even if they had not fully adhered to them (Extract 14, Extract 15).

> *Extract* 14 (*CCH1*): "Definitely write all the code and test it later….. gonna attempt to try to do incremental testing this time around, mainly for sanity... also for confidence knowing that something is working and i don't have to worry about it. This would help with keeping track of project progress."

> *Extract* 15 (*CCA3*): "It is very important for making every part work properly. I believe it is important to focus all the part of programming importantly otherwise it would take more."

In contrast, 4 out of 12 students who employed incremental testing practices reported a more positive experience (Extract 16).

> *Extract* 16 (*CCH3*): "I test things pretty much as I write it. Otherwise you run into situations where bugs manifest in one place, but the root cause is somewhere else. Writing tests and debugging before moving on helps to know that things are working before moving on."

Based on our analysis, we derive the following Insight 2:

Students reported that delaying testing and relying solely on post-implementation system-level testing results in suboptimal performance outcomes.

Limitations of CC Metrics

We found several limitations with using CC metrics in assessing the quality of test suites and guiding students' testing practices. 10 out of 12 students who used CC metrics expressed concerns about the effectiveness of this approach (Extract 17, Extract 18).

> *Extract* 17 (*CCA3*): "CC doesn't help you figure out WHERE the bug is, it would just tell you something is covered."

> *Extract* 18 (*CCS3*): "Part of the requirement was omitted from the sample input and output, so we were stuck on this for like a week before getting help from the TA."

The limitations of CC metrics led to a heavy reliance on the AAT for testing and debugging (Extract 19, Extract 20).

> *Extract* 19 (*CCA2*): "We did submit to Web-CAT multiple times to help find the bugs….definitely took advantage of unlimited submissions sometimes….and there were some times we would fix something and submit again to see if our fix 'worked'."

> *Extract* 20 (*CCS2*): "Web-CAT submissions were the 'ultimate' tool."

The inadequacy of CC metrics also resulted in an increased dependency on instructional support for testing guidance (Extract 21, Extract 22).

> *Extract* 21 (*CCS1*): "No bugs in my local machine; when i submitted to Web-CAT, that's when I started having bugs; so the TA helped me figure out the bug."

> *Extract* 22 (*CCA1*): "TA helped us think about other edge cases that Web-CAT was testing; they referred to the output that Web-CAT expected and where our code was different…..helped us know which edge cases we were not accounting for."

In our interviews with the CC group, we explained to students how MT works and compared it to CC. 11 out of 12 interviewees indicated that what they were told sounds better to them than CC in providing more comprehensive feedback (Extract 23, Extract 24).

> *Extract* 23 (*CCH2*): "MT + reference test feedback would be multiple layers of feedback, that would help triangulate where bugs are in the code."

> *Extract* 24 (*CCA2*): "It (MT) would be better than CC."

Based on our analysis, we derive the following Insight 3:

Students found CC metrics inadequate for test assessment, causing reliance on the AAT and staffs for testing guidance.

5.2.3 CC Intervention Summary

Therefore, to address RQ1 we can say that based on our qualitative findings students using CC placed little inherent value in writing test cases (Insight 1), often delayed testing until after implementation (Insight 2), and heavily relied on automated assessment tools for testing guidance (Insight 3). This led to suboptimal testing practices and project outcomes.

5.3 MT Phase I: Deploying MT with PIT Stock Feedback

5.3.1 Description of PIT stock MT Intervention

Our qualitative findings reaffirm the need for an alternative metric to CC. The insights derived from the CC semester interviews underscore the limitations of CC in promoting effective testing practices. In the MT_{ps} semester, we introduced MT as a modified version of PIT, using only the logical test and arithmetic mutation operators, as recommended by [37] in order to reduce load on the AAT without significant reduction in effectiveness. Students' MT scores were assessed as part of project submission grading and feedback by our AAT. Additionally, we implemented PIT Eclipse plug-in that students could use to monitor their MT coverage locally. Both the AAT and the local plugin reported three key metrics: line coverage, mutation coverage, and test strength, along with detailed breakdowns by class. While the local PIT plug-in and the AAT occasionally showed minor variations in their feedback, both tools helped students track their testing progress throughout development.

The PIT plug-in interface shown in Figure 5.2 provides a comprehensive yet potentially overwhelming set of feedback mechanisms. The main summary panel displays three key metrics: Line Coverage (98%), Mutation Coverage (89%), and Test Strength (90%). These stock PIT metrics are accompanied by detailed fractions showing the exact numbers, such as 466/475 lines covered.

The "Breakdown by Class" section provides a granular view of these metrics for each Java class in the project. For instance, the HashTable.java class shows 93% line coverage (79/85), 83% mutation coverage (66/80), and 85% test strength (66/78). Test strength is defined as the ratio of the number of mutations killed by the test suite to the number of mutations

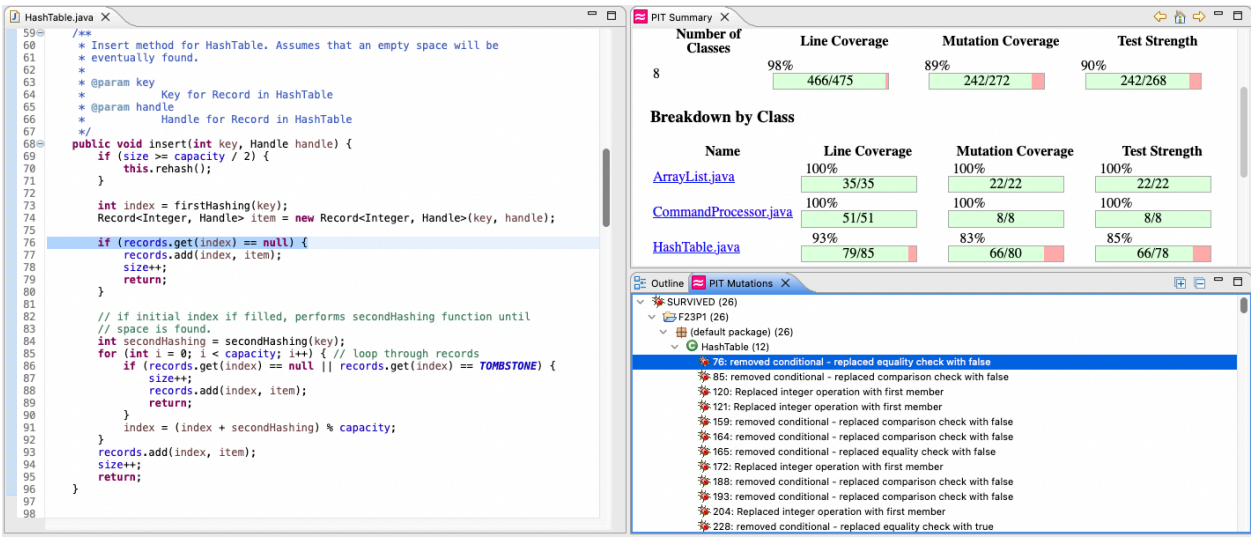

Figure 5.2 MT analysis (using selective mutators) of a student-written test suite in Eclipse IDE with PIT's stock feedback.

covered by the tests. Mutation coverage, on the other hand, is the ratio of mutations killed by the tests to the total number of possible mutations, irrespective of their coverage by the tests. Line coverage refers to the ratio of line statements executed by the tests to the total number of line statements potentially executable within the class under examination.

The plug-in also includes a mutation detail panel that lists "SURVIVED" mutants, which are mutations that weren't detected by the test suite. Meaning the tests still passed even when the code was mutated. In the example shown, line 76 in HashTable.java has a surviving mutant when a particular conditional check was replaced with 'false'. Similar mutations are listed for other lines, such as line 85 where a comparison check was modified. The "KILLED" category refers to mutations that were successfully detected by the test suite — meaning the tests failed when they should have failed. The "TIMED_OUT" group represents mutations that caused the tests to exceed their allocated execution time. As a result, the mutants "survived" or weren't detected by the test suite. Finally, the "NO_COVERAGE" category indicates code segments that weren't executed by any test cases. The code view panel highlights the relevant lines of code where mutations were applied, allowing students to see

exactly where their test cases need improvement. This visual feedback helps students locate specific areas requiring attention.

Through anecdotal evidence from students and poorer project outcomes, we determined that there were problems with the deployment of MT in our courses. One of the issues we identified is that PIT's stock feedback presents several challenges for students. The groupings of mutants and jargon used in the feedback can create an additional layer of complexity, making it difficult for students to understand the results while simultaneously learning software testing concepts. The abstract nature of MT, combined with the specialized vocabulary, appears to lead to knowledge overload and requires additional explanation and context to make the feedback more accessible to student programmers. The volume and complexity of PIT's stock feedback, which is not tailored to students' needs, can create a barrier to learning for those who are still developing their testing skills.

When implementing PIT in Web-CAT, we began augmenting the stock feedback with additional clarifications. Web-CAT presents MT feedback through a visual interface that highlights specific lines of code where mutations have "survived" — meaning the existing test cases failed to detect the introduced changes. As shown in the attached image of HashTable.java, lines that contain surviving mutants are highlighted with a light red background. This visual indication immediately draws students' attention to areas where their test cases need improvement. When students hover over these highlighted lines, Web-CAT displays informative pop-up messages that provide specific guidance about the surviving mutants. For instance, in Figure 5.3, hovering over line 76 reveals a hint stating "Uncovered mutation(s): Test case assertions do not confirm logical condition(s) to produce intended behavior in all cases. HINT: Test 'equality' condition for 'false' cases." This augmented feedback helps students understand exactly what scenarios their tests are missing and how to strengthen them.

src/HashTable.java

Select from previous comments | Error | To Everyone | Show all comments

```
59      /**
60       * Insert method for HashTable. Assumes that an empty space will be
61       * eventually found.
62       *
63       * @param key
64       *            Key for Record in HashTable
65       * @param handle
66       *            Handle for Record in HashTable
67       */
68      public void insert(int key, Handle handle) {
69          if (size >= capacity / 2) {
70              this.rehash();
71          }
72
73          int index = firstHashing(key);
74          Record<Integer, Handle> item = new Record<Integer, Handle>(key, handle);
75
76          if (records.get(index) == null) {
77              records.add(index, item);
78              size++;
79              return;
80          }
81
82          // if initial index if filled, performs secondHashing function until
83          // space is found.
84          int secondHashing = secondHashing(key);
85          for (int i = 0; i < capacity; i++) { // loop through records
86              if (records.get(index) == null || records.get(index) == TOMBSTONE) {
87                  size++;
88                  records.add(index, item);
89                  return;
90              }
91              index = (index + secondHashing) % capacity;
92          }
93          records.add(index, item);
94          size++;
95          return;
96      }
```

Uncovered mutation(s): Test case assertions do not confirm logical condition(s) to produce intended behavior in all cases. HINT: Test 'equality' condition for 'false' cases.

Figure 5.3 MT analysis (using selective mutators and modified feedback) of a student-written test suite on Web-CAT AAT online portal.

Beyond the specific code-level feedback shown in the image, Web-CAT provides a comprehensive breakdown of mutation coverage statistics for the entire project. The platform generates detailed reports showing the mutation coverage percentage for each Java class and the project as a whole. These statistics indicate what percentage of introduced mutations were successfully "killed" (covered) by the test suite. Higher percentages indicate more robust test suites, while lower percentages suggest areas needing additional test cases.

5.3.2 Students' Impression of MT with PIT Stock Feedback

In the CC semester, students typically deferred testing until after completing their code implementation, viewing test coverage as a secondary requirement like code style rather than an integral part of development. This behavior suggests that students did not recognize how CC contributed to effective testing and program correctness, instead treating it as a separate grading criterion to be addressed after ensuring their code worked. Limited guidance in CC also led students to rely on AAT feedback and instruction. To address this, we introduced PIT's MT in the MT_{ps} semester. However, its stock feedback was difficult for students to understand, especially without prior experience, causing confusion and struggles in applying it.

To better understand the challenges our students experienced with PIT's stock MT feedback, we conducted 15 interviews in the MT_{ps} semester. The interviewees represented a diverse range of academic performance: five high-achieving students, six average-performing students, and four students who were struggling with the course material. The qualitative analysis of these interviews revealed key challenges and misconceptions in how students interpreted the feedback. One insight revealed by the interviews is that students perceive PIT's feedback as verbose and overwhelming. 11 out of our 15 student participants ex-

pressed frustration with the nature and verbosity of the feedback, which created barriers to understanding and implementing the suggested improvements (Extract 25, Extract 26).

> *Extract* 25 (*MPS1*): "Sometimes the feedback was redundant. At times, confusing."

> *Extract* 26 (*MPS2*): "[PIT's feedback is like] a black box which takes away points. I don't [know] why and that makes me feel like a poor computer scientist. Because there is this thing that I should understand and I don't."

The volume and complexity of information presented made it challenging for students to identify the most relevant aspects of the feedback to address (Extract 27, Extract 28).

> *Extract* 27 (*MPA6*): "It was confusing where it wanted us to focus. Because we already did what it was telling us to do."

> *Extract* 28 (*MPA2*): "In some cases, I was just lost. Because of some mutants. I tried like removing conditions like, if head equals to null or suppose, if not head. But the mutant was still there."

Another significant challenge was difficulty in extracting actionable insights from the feedback. Some students (6 out of 15) often found themselves unable to translate the feedback into concrete testing improvements (Extract 29, Extract 30). The gap between receiving feedback and understanding how to act upon it effectively was a recurring theme in students' comments.

> *Extract* 29 (*MPA1*): "I can't quote it but it said something like make sure to check the 'true' cases or 'false' cases. And that doesn't mean too much to me."

> *Extract* 30 (*MPA5*): "MT wanted me to test either my insert or my remove if size was changed. And I thought I was testing that. It said did the size changed. I was very confused on how to catch that."

Misconceptions about MT feedback also emerged as a significant barrier to improving test suite coverage. The impact of these misconceptions was particularly pronounced during students' initial encounters with MT (Extract 31). The complex structure of the feedback and the nature of the mutation operations contributed to these misconceptions (Extract 32).

> *Extract* 31 (*MPH2*): "I figured it out on my own. But some other people might have struggled. That might have been the big thing that led to some of confusion. Especially, Project 1 with mutation feedback."

> *Extract* 32 (*MPS4*): "The way the feedback told us to test it didn't make sense to me at all. Because we were already testing it the way it was telling us to test it. But it was basically repeating what we did."

8 out of 15 students noted that the confusion and difficulty in understanding the feedback were most prevalent at the beginning of their experience with MT (Extract 33). The nature of the feedback also created challenges in understanding specific mutation operations (Extract 34, Extract 35, Extract 36). This suggests that proper introduction, guidance, and support could potentially mitigate these challenges.

> *Extract* 33 (*MPH1*): "[I struggled with MT] only at the beginning when I didn't know anything about MT."

> *Extract* 34 (*MPS3*): "It is just the wordage that was confusing. I had to get used to that. I think if there were some sort of example of mutation feedback then that would be helpful."

> *Extract* 35 (*MPH3*): "I am like, give more description of what variable is replaced rather than a broad overview."

> *Extract* 36 (*MPA3*): "Maybe it was getting messed up with the lines or we were misinterpreting them. Even if it had the line of code, it was confusing on what to do."

The challenges and misconceptions also led to concerns about the influence of MT feedback on testing practices. Some students (5 out of 15) expressed worries that focusing too heavily on MT feedback might lead to misaligned testing practices, where they prioritize testing for mutation coverage rather than testing their solution to the problem (Extract 37, Extract 38).

> *Extract* 37 (*MPA4*): "Because I feel like I would be trying to test for MT instead of just testing normally and that would be very confusing because that's not how we usually test. Because you should be testing for your solution to the problem."

> *Extract* 38 (*MPH5*): "I could not identify which test are purposeful and which tests are testing the same things."

Based on our analysis of student comments, we derive the following Insight 4:

PIT's standard MT feedback leads to misconceptions and challenges with interpreting feedback.

5.3.3 PIT stock MT Intervention Summary

This insight addresses RQ2 and underscores the need for clearer, more accessible feedback in MT tools, especially in educational settings.

5.4 MT Phase II: Implementing MT with Modified Feedback and Instructional Materials

5.4.1 Description of Modified MT Intervention

Based on student feedback from the MT_{ps} semester, we enhanced our MT implementation with targeted training modules containing clearer documentation and additional examples. These improvements focused on making MT concepts more accessible and actionable for students. We also heavily modified the PIT feedback provided on both Web-CAT and via the MT plug-in. The design decisions behind the new modified feedback were informed by research on effective feedback practices in education [8, 16, 28, 59, 65]. As [59] noted, feedback needs to be easily understandable and help improve students' learning process. The modified plug-in addresses this by presenting information in a more digestible and actionable format.

The mutation summary panel in the modified version shows the overall mutation coverage for the entire project and a "Breakdown by Class" with individual mutation coverage percentages. The hierarchical organization by class and package makes it easier for students to navigate through their project structure and systematically address testing gaps, and identify classes that have lower mutation coverage percentages.

Mutation results are organized into two clear categories: "LINES_NOT_TESTED" and "LINES_NEEDING_BETTER_TESTING", as seen in Figure 5.4. The new terminology directly communicates what actions users need to take – either writing new tests for untested code or improving existing tests for better coverage. This aligns with [8]'s idea of providing feedback incrementally as "one new idea at a time" to ensure a more graduated learning path. This classification system replaces PIT's jargon of "SURVIVED," "KILLED,"

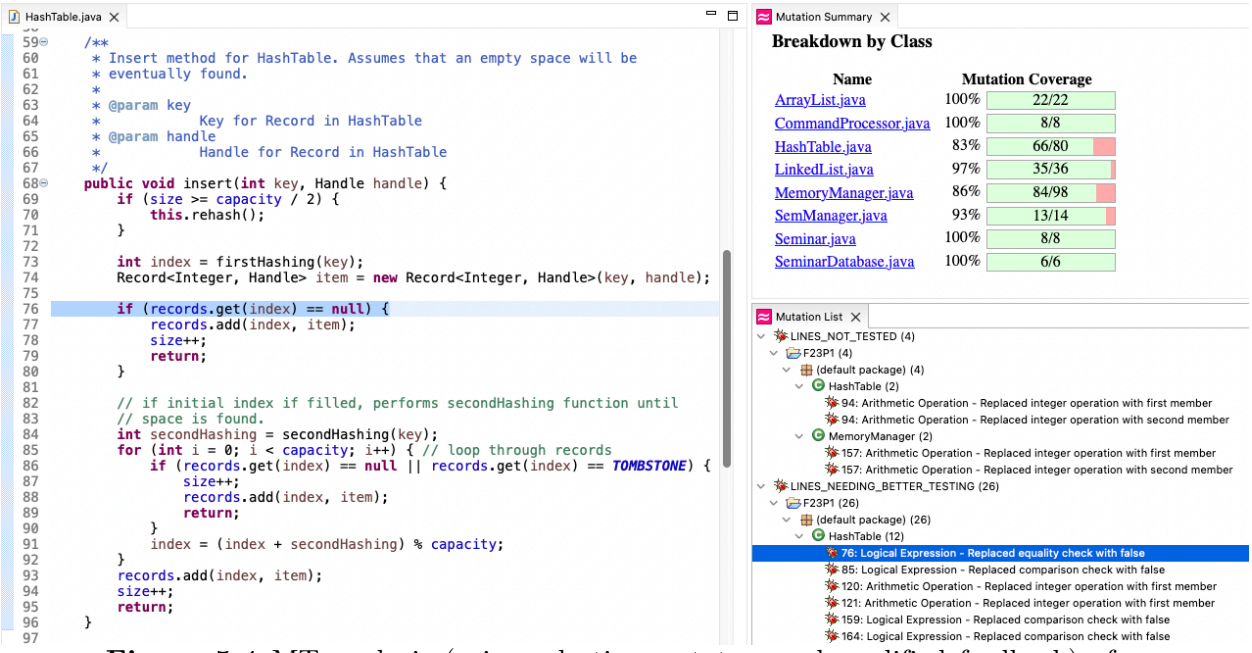

Figure 5.4 MT analysis (using selective mutators and modified feedback) of a student-written test suite shown in the Eclipse IDE using our MT plug-in with modified feedback.

"TIMED_OUT," and "NO_COVERAGE," which could be confusing for users.

The mutation list panel provides precise, actionable feedback about each mutation. For instance, instead of merely indicating a survived mutant, the feedback clearly states the type of mutation applied, such as "Logical Expression - Replaced equality check with false" or "Arithmetic Operation - Replaced integer operation with first member." This explicit description helps students understand exactly what changes were made to their code, which hopefully guides them in writing appropriate test cases. By providing this level of detail, the modified feedback presents deficiencies as faulty interpretations rather than as a total lack of understanding, which [28] noted as being more powerful for learning.

Unlike PIT's stock feedback, which can overwhelm students with details, our intervention includes both a modified version of MT feedback and comprehensive documentation and educational resources specifically designed for student learning. These resources explain MT

concepts, provide examples of how to interpret the feedback, and offer guidance on writing effective test cases. This educational support structure helps students better understand not just what to do, but why they're doing it, fostering a growth mindset and learning goals [16]. By removing jargon and providing clear, actionable feedback with supporting training resources, the tool better helps students understand mutation testing results. The modified feedback and additional documentation provide the necessary scaffolding to motivate students' interest, simplify the task, provide direction, and clearly define expectations [65], ultimately leading to more effective learning and development of strong testing skills.

5.4.2 Students' Impression of MT with Modified Feedback and Instructions

To assess the impact of our modified MT approach, we conducted semi-structured interviews with 11 participants in the MT_{mf} semester. Using the Grounded Theory approach, we derived three main themes related to students' testing outcomes, testing strategy evolution, and debugging practices.

Impact on Testing Perception

In the MT_{mf} semester, students showed a notable difference in their perception and approach to testing compared to those interviewed in the CC semester. 10 out of 11 students using modified MT feedback reported recognizing the importance of quality testing, understanding testable code development, and having a positive attitude toward testing, showing improvement compared to students who used CC or stock MT feedback. Through their reflections on the MT experience, students gained a deeper understanding of the value of comprehensive testing beyond the CC metric that they had used in earlier courses on simpler

programs (Extract 39, Extract 40).

> *Extract* 39 (*MMH1*): "MT was different than CC. MT would get CC but CC would not always get all the MT. It's actually a better form of indicating code maintenance."

> *Extract* 40 (*MMH2*): "Sometimes CC would just cover line. But for MT, covering a line is more meaningful. When a test fails, it tells you something is wrong here."

10 out of 11 students using MT with modified feedback developed an appreciation for incremental development and testing practices, recognizing the benefits of early testing and unit testing (Extract 41).

> *Extract* 41 (*MMH3*): "I think if we waited and wrote the entire project and then tested we'd have no clue where to start."

The accessibility and immediacy of feedback provided by the MT plugin further contributed to students' growing comfort with testing practices (Extract 42, Extract 43). Moreover, students (9 out of 11) reported that MT helped them discover previously unconsidered test scenarios, leading to a more comprehensive approach to test case design (Extract 44).

> *Extract* 42 (*MMS3*): "The plugin one was very helpful, it showed the overall coverage, and also showed what classes had the lowest percentage coverage. You could use it locally and run MT to see locally. So you don't have to like submit to Web-CAT every single time."

> *Extract* 43 (*MMA5*): "After we figured out (MT), it does got a lot easier. (Later, on Project 3) We were experts."

> *Extract* 44 (*MMA4*): "It helped me see all the different possibilities where my code is actually failing at where I would never actually realize in the first place."

The modified MT feedback also contributed positively towards students' testing attitudes. They acknowledged that while MT required more effort, it helped them write more in-depth test cases and potentially reduce issues and bugs (Extract 45).

> *Extract* 45 (*MMS2*): "When you're first starting the program you kind of hate testing... But MT helps you think of more tests and like how rigorous your code actually has to be tested. So I could test like that (rigorously) by myself."

Based on our analysis, we derive the following Insight 5:

Students using modified MT feedback express greater appreciation for quality test writing and testable code development than those who use CC or stock PIT MT feedback.

Improved Testing Outcomes

Our qualitative analysis revealed significant improvements in students' testing outcomes when working with modified MT feedback. 10 out of 11 students demonstrated a systematic approach to improving test suite coverage by strategically prioritizing classes with lower coverage scores before addressing more complex cases (Extract 46).

> *Extract* 46 (*MMH3*): "I'd look at like the least number one, or like the least percentage that's covered. I'd go in that class, and I'd kill all the mutants in that class first, and I'd move on to the other ones, because I know the ones with the higher coverage would be harder to test overall."

They exhibited a better understanding of the purpose and value of their tests, transitioning from superficial coverage to meaningful test cases (Extract 47).

> *Extract* 47 (*MMH2*): "And so some test cases would be kind of pointless in a sense, just to achieve line coverage but now tests are more so purposeful as they're designed to test and fail in certain cases."

Students (8 out of 11) also showed enhanced ability to address testing requirements through improved testing strategies (Extract 48).

> *Extract* 48 (*MMA1*): "...That was test that were tested (by MT) and killed specifically and say, Okay. Line 300 had a lot of mutants and those them were killed. We're clearly not testing this part of the function well enough."

As they gained experience with MT, students (9 out of 11) grew more confident and demonstrated sophisticated decision-making in their testing approach (Extract 43, Extract 49).

> *Extract* 49 (*MMA2*): "We already had, like, previously written linked lists from like CS2 course. But they would be missing like mutation coverage... If it wasn't needed in the project, and mutation was giving me issues. I just fix it or delete it (redundant code)."

Despite the increased effort required, students (8 out of 11) recognized that MT led to better outcomes, helping them discover edge cases and write more in-depth test cases (Extract 50, Extract 44).

> *Extract* 50 (*MMA5*): "From a students' perspective, it was more annoying because it was more work. Potentially have less issues and bugs. It does help us like write more in depth test cases which is the goal for our projects."

Based on our analysis, we derive the following Insight 6:

Students expressed that modified MT feedback leads to more successful outcomes in test suite coverage improvement compared to those who used CC or stock MT feedback.

Testing Strategy Evolution

Our qualitative analysis revealed a significant evolution in students' testing strategies throughout their experience with MT. As students gained experience with MT, they increasingly adopted unit testing practices, moving from superficial testing to more meaningful and methodical approaches (Extract 51, Extract 47).

> *Extract* 51 (*MMA4*): "I use the third option (unit testing) where I rewrite each method and then test it. So if a problem comes up in the method, it doesn't cascade into like mixing that method within another method."

They showed a clear adoption of incremental testing, emphasizing the importance of early testing and improved efficiency as reported by all 11 students (Extract 41, Extract 52).

> *Extract* 52 (*MMA1*): "I would say, the MT strategy definitely allowed us to get higher problem coverage with fewer submissions. If you look at [Web-CAT] okay, you can definitely see the submissions went down with each project considerably. I'll say our first project was like over 112, and our final project was 47."

Students also demonstrated a more systematic integration of testing throughout the development process, recognizing the value of testing even when not explicitly required (Extract 53, Extract 54).

> *Extract* 53 (*MMA2*): "I would say, the best method is writing the test first to make sure your output looks how it should be."

> *Extract* 54 (*MMH1*): "In Project 3, writing tests was not required. But testing was definitely helpful. Which helped us figure out that the algorithm was little bit flawed."

MT influenced not only testing practices but also code design decisions, as students (5 out of 11) learned to optimize their code for better maintainability (Extract 55).

> *Extract* 55 (*MMS2*): "Towards the end (after Project 2 or Project 3), I figured how to write code to reduce the number of mutants. I would optimize it to be better maintained. It was such a light bulb got on like, Oh, you could've reduce the number of branches and saved a lot of error."

The progression in testing strategy was evident in 4 out of 11 students' recognition of best practices, such as test-driven development, even when faced with constraints (Extract 56).

> *Extract* 56 (*MMH2*): "We didn't use TDD (test-driven development) because not all requirements were not known by us. We did not know all the functions and parameter. But TDD would be the best."

10 out of 11 students expressed strategic thinking in their testing approach, systematically handling mutation feedback to identify and address weaknesses in their test cases (Extract 57).

> *Extract* 57 (*MMS3*): "In the Mutation List, each class and the number of branches, I would look at the mutants... We're clearly not testing this part

> of the function well enough, and I'll go to the function. I'll read it. And think of how think of edge cases and how we can better test that function."

Based on our analysis, we derive the following Insight 7:

Students using MT show better adoption to unit testing practice in their project development.

Test-Driven Debugging Practices

Our qualitative analysis revealed distinct differences in debugging practices between students using MT and those relying solely on CC. 91% of MT users developed more systematic debugging approaches compared to the ad-hoc methods employed by 83% CC users (Extract 58).

> *Extract* 58 (*CCA1*): "Debugger is usually used when I have no idea' – when I don't know what's going wrong, but I know 'something' is wrong."

Test-driven bug verification practices were more thorough among 9 out of 11 MT users, while CC users (all 12 out of 12) relied on AAT without proper verification (Extract 59).

> *Extract* 59 (*CCS1*): "Test the fix? We had a time crunch so we just submitted to [Web-CAT] and once we saw it was working we didn't bother with regression testing the bug."

MT users (10 out of 11) demonstrated more systematic approach to bug replication, leveraging MT feedback to identify edge cases (Extract 60). They also exhibited more mindful and thorough testing compared to CC users (Extract 61).

> *Extract* 60 (*MMH2*): "I would say, edge cases mostly come from specs and for MT, it would kind of tell us, okay, this branch is not killed, when this is 'true'."

> *Extract* 61 (*MMS2*): "CC was easier to cover but MT was harder. For MT your tests need to be more thorough."

Post-fix validation methods were more comprehensive among MT users (6 out of 11), while CC users often relied on minimal validation (Extract 4, Extract 44). The evolution of debugging practices was particularly evident in MT users, who reported a change from merely fixing bugs to preventing them through better code design and showed increased appreciation for thorough testing (Extract 55).

Based on our analysis, we derive the following Insight 8:

Students using MT express more mindful testing practices.

5.4.3 Modified MT Intervention Summary

Insights (5-8) address our RQ3. Students using modified MT feedback demonstrated increased appreciation for quality testing and testable code development (Insight 5), believed leads to better test suite coverage (Insight 6), showed improved adoption of unit testing practices (Insight 7), and exhibited more mindful testing practices (Insight 8). These outcomes suggest that modified MT feedback, combined with appropriate instructional resources, significantly enhances students' testing practices and learning outcomes compared to stock MT feedback.

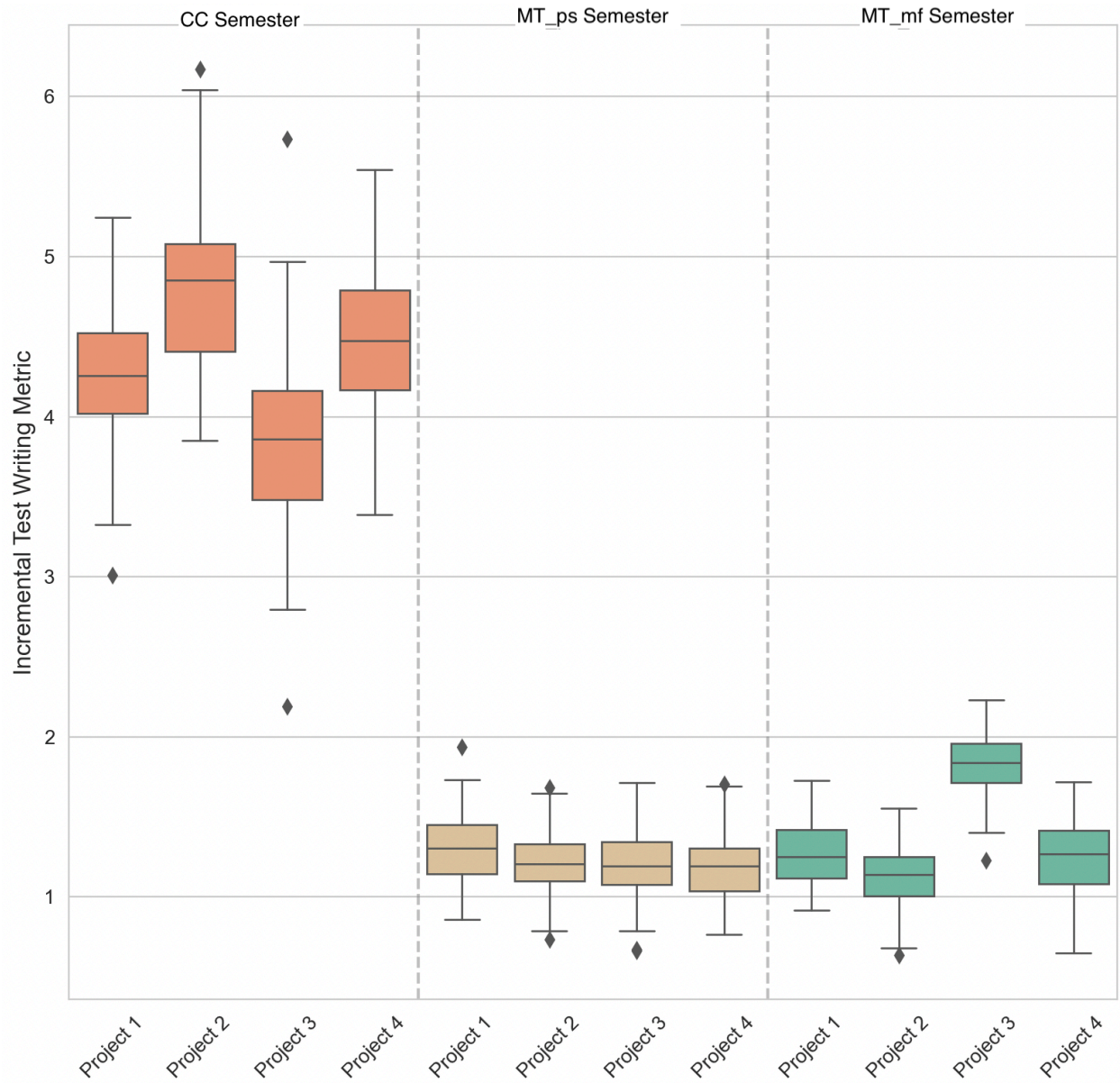

Figure 5.5 Distribution of incremental test writing scores across CC, MT_{ps} and MT_{mf} semesters, showing significantly lower values in MT semesters compared to CC. Lower values indicate closer alignment between test and solution edits, demonstrating MT's effectiveness in promoting incremental testing practices throughout project development.

5.5 Quantitative Analysis

To validate the insight that ***Students using MT show better adoption of unit testing practices in their project development***, we conducted a quantitative analysis of students' test writing behavior using two metrics developed by [38]. The first metric, "incTestWriting," measures the difference in average time spent editing the solution versus the test code, gauging the degree of incremental testing. Smaller values indicate closer alignment between test and solution edits, while larger values suggest that test editing occurs later, after most of the solution code is written. This metric is calculated using the second metric, "earlyOften," which produces a weighted average based on the frequency of student activity and its proximity to the project deadline. Together, these metrics help assess the consistency of test writing throughout the project lifecycle.

Our analysis revealed significant differences in testing practices between the CC and MT semesters. Students in both the MT_{ps} and MT_{mf} semesters demonstrated significantly lower (meaning, better) incremental test metric values compared to those in the CC semester (p-value < 0.05) across all four projects. As illustrated in Figure 5.5, the box plots show that CC semester students typically exhibited metric values between 3.0 and 5.0, while students in MT semesters consistently maintained values between 1.0 and 2.0. This indicates substantially more incremental test writing behavior among MT students.

The quantitative evidence strongly supports our qualitative observation that MT use promotes better integration of unit testing practices during project development. MT students to develop and maintain their test suites alongside their solution code, instead of postponing testing until after completing the implementation. The consistent pattern across all four projects further suggests that this improved testing behavior persists throughout the semester rather than being either a temporary or a slowly learned effect. The adoption of

incremental testing practices not only validates our qualitative findings but also highlights the positive impact of MT on students' overall testing approach.

Next we cross-check our insight that ***Students expressed that modified MT feedback leads to more successful outcomes in test suite coverage improvement compared to those who used CC or stock MT feedback***. We analyzed mutation coverage scores using MuJava, an independent mutation testing tool distinct from PIT, to minimize tool-specific bias.

Students in MT semesters achieved significantly higher mutation coverage compared to those in the CC semester. As shown in Figure 5.6, the median mutation coverage in the CC semester was approximately 48%, while both MT_{ps} and MT_{mf} semesters demonstrated notably higher coverage at 65% and 73% respectively. The MT_{mf} semester, where students used our modified MT feedback, showed the highest overall coverage scores. Due to violations of normality (p-value < 0.05) and homogeneity of variance assumptions (Levene's test, p-value < 0.05), we used non-parametric statistical analysis. The Kruskal-Wallis test, followed by Dunn's post-hoc analysis with Bonferroni correction, confirmed statistically significant differences between all semester pairs (p-value < 0.001). This statistical evidence strongly supports our qualitative finding that MT feedback enhances test suite coverage, with modified feedback showing even greater improvements over stock MT feedback. This quantitative evidence, combined with our interview findings where 70% of students reported successfully improving their test coverage using MT feedback, demonstrates that modified MT feedback effectively guides students toward developing more robust test suites.

Having established the positive impact of MT feedback on test suite quality, we next examined whether this improved testing practice translated into better overall code quality and project outcomes. This analysis focused on two key aspects: the testability of students' code and their project correctness scores. To evaluate code testability, we analyzed

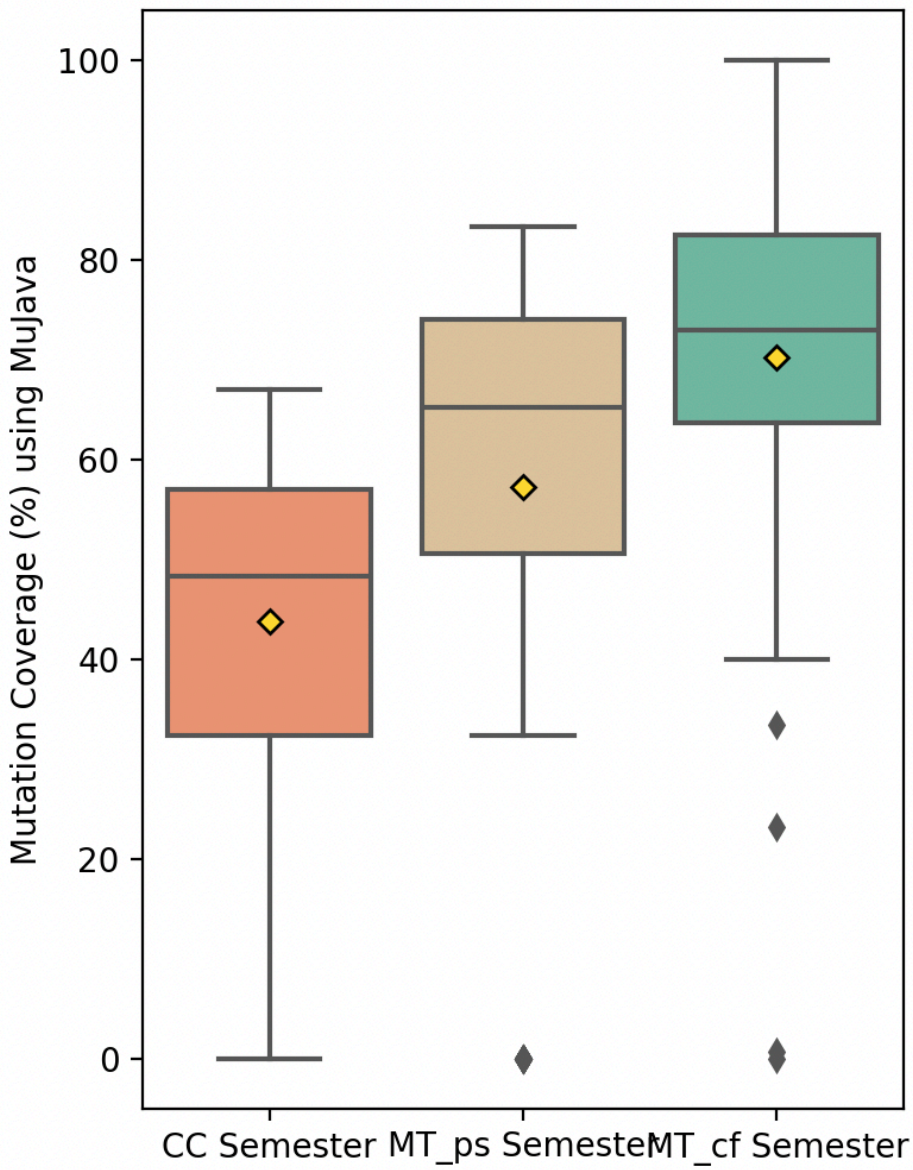

Figure 5.6 Comparison of mutation coverage scores across CC (48%), MT_{ps} (65%), and MT_{mf} (73%) semesters using MuJava, demonstrating significantly higher test suite quality in MT semesters (p-value < 0.001), with modified feedback (MT_{mf}) yielding the most robust test suites.

students' source code using PMD, focusing particularly on cyclomatic complexity metrics. The analysis revealed significantly lower complexity values in MT semesters (mean values: $MT_{ps} = 152$, $MT_{mf} = 119$) compared to the CC semester (mean = 220), with statistical significance confirmed through Kruskal-Wallis and Dunn's post-hoc tests (p-value < 0.05). As illustrated in Figure 5.7, the complexity distribution curves show consistently lower values across student submissions in both MT semesters, with MT_{mf} demonstrating the most favorable complexity profile. This improvement in code testability was accompanied by enhanced project outcomes. Analysis of project correctness scores showed that students using modified MT feedback achieved significantly higher scores (p-value < 0.05) compared to both CC and MT_{ps} semesters. While students using stock MT feedback initially struggled in Project 1, they eventually achieved comparable scores by the semester's end, as seen in Figure 5.8 and Figure 5.9. This pattern suggests that our modified feedback helped students overcome initial barriers in interpreting mutation testing results, enabling them to realize their full potential earlier in the learning process. These findings quantitatively support our insight that ***Students express increased appreciation for quality test writing and testable code development when using modified MT feedback***, as reflected in both their code complexity metrics and project outcomes.

Finally, we examined how MT influenced students' testing behavior and their reliance on Web-CAT AAT. Our qualitative observations suggested more mindful testing practices with MT and reduced dependence on Web-CAT feedback, warranting quantitative verification of these patterns.

Analysis of local testing behavior revealed that students using MT conducted significantly more local test coverage checks compared to those using CC (p-value < 0.001). The mean number of local test launches increased from 181 per student in the CC semester to 382 and 303 in MT_{ps} and MT_{mf} semesters respectively, indicating more frequent local testing

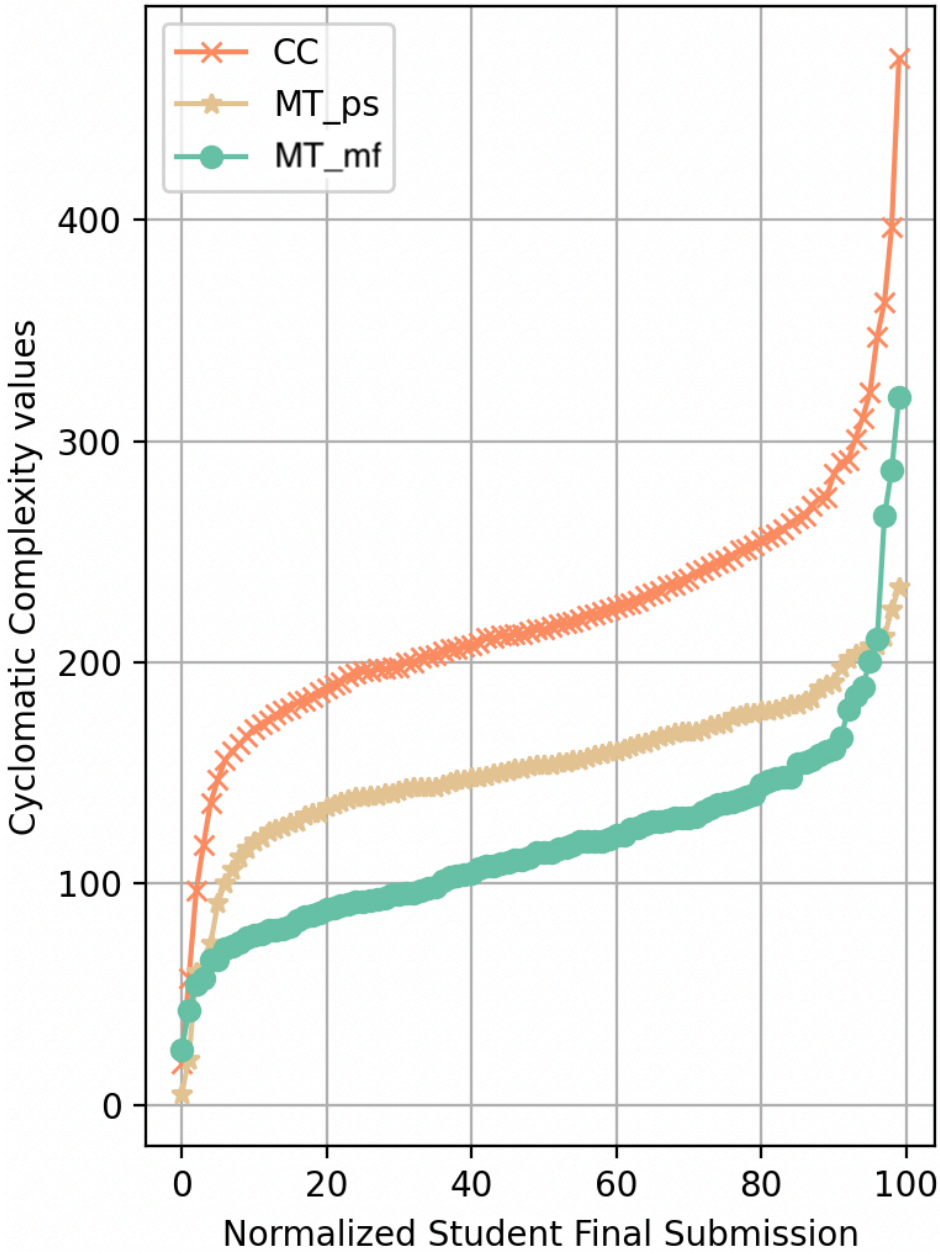

Figure 5.7 Comparison of cyclomatic complexity values across semesters, showing significantly lower complexity in MT semesters ($MT_{ps} = 152$, $MT_{mf} = 119$) compared to CC semester (mean = 220, p-value < 0.05), indicating improved code testability through MT intervention.

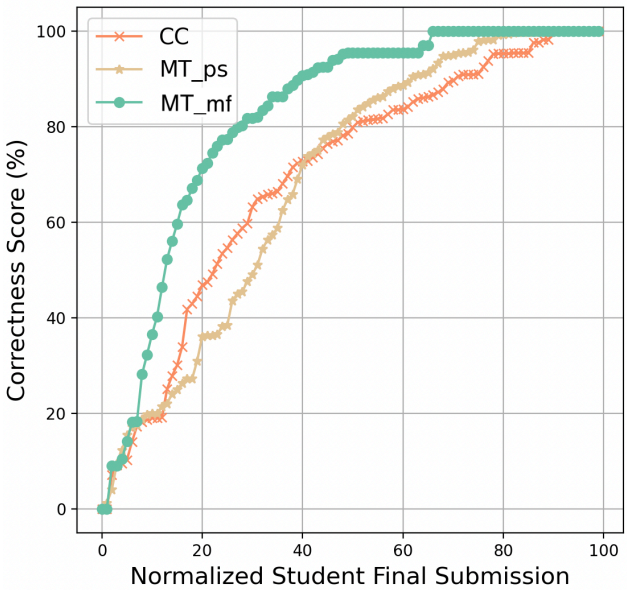

Figure 5.8 Comparison of Students' Project 1 Correctness Score in CC, MT_{ps} and MT_{mf} Semesters. Here, mean scores for students in CC, MT_{ps}, and MT_{mf} semesters were 70%, 69%, and 82%, respectively. This indicates, a large portion of the students using PIT's stock MT tool had difficulty with project correctness score.

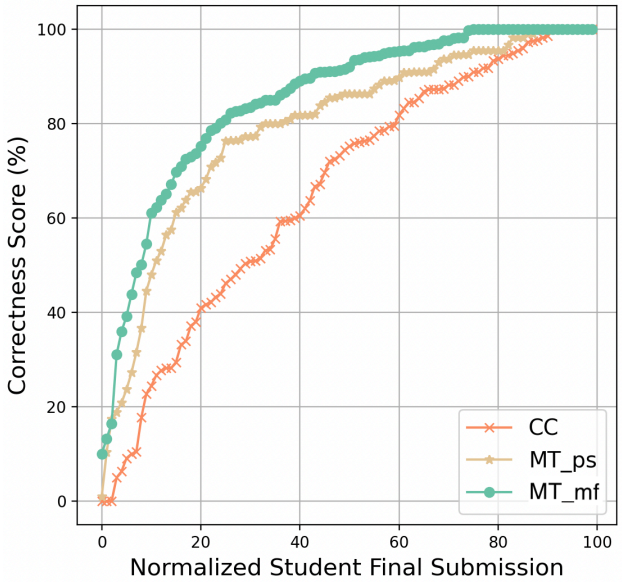

Figure 5.9 Comparison of Students' Project 4 Correctness Score in CC, MT_{ps} and MT_{mf} Semesters. Here, mean scores for students in CC, MT_{ps}, and MT_{mf} semesters were 69%, 79%, and 86%, respectively. This shows students using MT in both MT_{ps}, and MT_{mf} semesters had higher project correctness scores compared to students using CC in their fourth and final project.

activity. Simultaneously, we observed a significant reduction in Web-CAT submissions during MT semesters, which was influenced by both the introduction of MT and the implementation of submission energy (SE) - a policy limiting students to three submissions per hour. The SE policy was implemented in response to server congestion issues that emerged during the initial deployment of MT in MT_{ps} semester. The mean number of submissions decreased from 35 (SD = 35.47) in CC semester to 24 (SD = 22.56) in MT semesters, with the maximum number of submissions dropping from 360 to 179. Statistical analysis using the Mann-Whitney U test confirmed this difference as significant (p-value < 0.05). More importantly, the quality of Web-CAT submissions showed improvement during MT semesters. The percentage of submissions containing testing activity (adding, updating, or launching tests) increased substantially from 20% in the CC semester to 53% and 67% in the MT_{ps} and MT_{mf} semesters respectively. This indicates that students using MT engaged in more meaningful testing activities during their development process.

Through our mixed-methods analysis, we have systematically validated our qualitative insights with quantitative evidence. The data consistently supports our observations across multiple dimensions: from improved test coverage and code quality to more mindful testing practices and reduced dependence on automated assessment. These findings demonstrate that MT, particularly with our modified MT feedback, effectively promotes better coding and testing practices among students in academic context.

While our findings provide valuable insights into the impact of MT on students' testing behavior, it is important to acknowledge that testing and debugging involve a complex interplay of various techniques. As [43] discovered, students often use a combination of strategies, such as running test cases, using diagnostic print statements, and tracing through IDE source-level debuggers, to identify and resolve issues in their code. To gain a more comprehensive understanding of how MT influences students' problem-solving approaches,

in future research we plan to collect and analyze event logs from these different techniques. By examining the ways in which students use various tools and strategies in conjunction with MT, we can develop a more nuanced picture of how the introduction of MT reshapes students' debugging processes and behaviors. This holistic investigation will enable us to refine our interventions and develop more targeted support for students as they navigate the complex landscape of software testing and debugging in an educational setting.

5.6 Discussion And Threats to Validity

The primary goal of our research was to successfully implement MT as a replacement for CC in a large educational setting. In this mixed-methods research study investigating the impact of different feedback mechanisms on student testing behavior, we employed Grounded Theory for the qualitative analysis of semi-structured interviews conducted with students from three distinct semesters. We interviewed 38 students across three semesters: 12 from the CC semester, 11 from the MT_{ps} semester, and 15 from the MT_{mf} semester. Using Grounded Theory, we derived eight valuable insights, which we then cross-checked using quantitative analysis.

To ensure the reliability and validity of our qualitative analysis, we followed best practices in coding and inter-coder agreement [12]. Two researchers independently coded a subset of interviews from all three semesters, allowing for a comprehensive comparison across the different conditions. Initially, the inter-coder reliability was found to be low, with a Cohen's kappa coefficient of 0.52, indicating only moderate agreement. To address this issue, the researchers engaged in iterative discussions to clarify coding definitions, resolve discrepancies, and refine the coding scheme. After several iterations of coding and discussion, the inter-coder reliability significantly improved, reaching a Cohen's kappa coefficient of 0.85, in-

dicating strong agreement between the two researchers. Through this process, the researchers developed a comprehensive codebook that defined each code and provided examples to ensure consistency in the coding process. Using the established codebook, the researchers independently coded the remaining interviews from all three semesters. We also used qualitative data analysis software, such as Atlas.ti [48], to efficiently manage and organize the coded data, enabling us to identify emerging themes and patterns across the interviews.

On the other hand, we conducted the quantitative analysis in our study to provide compelling evidences that support our qualitative findings and insights regarding the impact of MT on students' testing practices and project outcomes. The significantly higher mutation coverage scores achieved by students in MT semesters, particularly those using modified MT feedback, demonstrate the effectiveness of this approach in improving test suite quality. This finding aligns with previous research highlighting the benefits of MT in educational settings [27, 44] and extends our understanding of the role that modified MT feedback plays in enhancing these outcomes.

Moreover, we observed improvements in code testability and project correctness scores among students using modified MT feedback that suggest that the benefits of this approach extend beyond test suite quality. By encouraging students to write more testable code and achieve higher project scores, modified MT feedback appears to foster a deeper appreciation for quality test writing and testable code development. This finding is particularly valuable, as it indicates that the use of MT not only enhances testing practices but also positively influences overall coding habits and project success.

Finally, the changes in students' testing behavior and reduced reliance on AAT during MT semesters provide further evidence of the transformative impact of this approach. The increased frequency of local test coverage checks and the higher percentage of submissions containing testing activity suggest that MT promotes more mindful and proactive testing

practices. Simultaneously, the reduction in AAT submissions implies that students become less dependent on external feedback and more self-reliant in their testing efforts.

It's important to note that submission energy (SE) was introduced in the MT semesters as a policy measure to prevent AAT server overload from rapid successive submissions. It limits students to three submissions per hour, encouraging more thoughtful and well-tested submissions rather than rapid trial-and-error approaches. When we analyzed the CC semester data, only approximately 4% of total student submissions would have been blocked by this policy, suggesting that SE primarily targets extreme cases of rapid submissions without significantly impacting normal student workflow. This low percentage also indicates that the reduction in submission numbers during MT semesters cannot be attributed solely to SE. Instead, the combination of SE and MT appears to have encouraged students to conduct more thorough local testing before submitting their work, leading to fewer but higher-quality submissions.

We addressed three key research questions about student testing behaviors with CC, challenges with MT, and the effectiveness of modified MT feedback. Regarding RQ1, our analysis revealed concerning patterns in students' testing behaviors when using CC. Students using CC demonstrated little intrinsic value for writing test cases, viewing testing primarily as a grading requirement rather than a valuable development practice. This aligns with previous research highlighting the limitations of CC metrics in promoting meaningful testing practices. For example, [1, 30] has highlighted the limitations of CC, noting that high coverage scores do not necessarily correlate with bug-free software. [62] even found some students write "pathological" tests to achieve high coverage scores for grading purposes, defeating the purpose of testing. We further found that the prevalent pattern of delaying testing, relying solely on post-implementation system-level testing, and heavy depending on AAT feedback, suggests that CC fails to promote effective testing habits, potentially leading to suboptimal

learning outcomes.

In contrast, MT has been found to be a better alternative to CC in simulating real faults [3, 36, 54, 69]. In the context of computer science education, [44] discovered that students who engage in more testing activities using MT typically achieve higher overall scores in large programming projects. [27] found that MT encouraged students to write more correct tests, check more code behavior, and actively seek to understand specifications. Such research motivated us to apply the state-of-the-art PIT's stock MT tool in our course setting. In addressing RQ2, we uncovered significant challenges students face when transitioning to MT using PIT's stock feedback. Despite MT's theoretical advantages over CC, during our initial implementation students encountered difficulties interpreting feedback and applying MT concepts effectively. This observation extends previous work by [27], who identified MT's potential benefits in education while acknowledging implementation challenges. Our research led us to understand the difficulties students experienced using the PIT MT tool and how these could be addressed. We used both literature knowledge [8, 16, 28, 59, 65] and insights from our qualitative research to inform our design decisions and develop modified MT feedback.

Our investigation of RQ3 yielded particularly promising results regarding the impact of modified MT feedback and supporting instructional resources. Students using our modified MT feedback demonstrated significant improvements across multiple dimensions: enhanced appreciation for quality testing, better test suite coverage, improved unit testing practices, and more thoughtful testing behaviors. These outcomes suggest that carefully designed feedback mechanisms, combined with appropriate instructional support, can effectively bridge the gap between MT's theoretical benefits and practical implementation in educational settings.

These findings collectively paint a picture of MT as a powerful tool for improving testing practices, code quality, and project outcomes in an educational setting. Our study high-

lights the importance of modified feedback in maximizing these benefits and underscores the potential for MT to fundamentally encourage students to adopt better attitudes and behaviors surrounding testing and code development. Our mixed-methods analysis confirmed our qualitative findings with quantitative data, showing that MT, especially with modified feedback, improves coding and testing practices among students in an academic setting. This study adds to the growing literature on MT in computer science education and offers useful insights for designing effective MT feedback systems. By combining qualitative analysis with quantitative data from student code snapshots and IDE activity logs, we gain a clear understanding of how feedback affects student testing behavior in an intermediate programming course.

We have considered, and where possible made efforts to mitigate, various potential threats to the validity of our research.

1. Selection Bias and Sample Representativeness: To mitigate potential selection bias and ensure representativeness, we employed stratified random sampling to select interview participants based on their programming project and exam performance. We divided the students into three groups: high achieving (top 15%), average (middle 50%), and struggling (bottom 35%), considering their project implementation scores, mutation coverage, and midterm exam scores. This approach ensured representation from all performance levels, as students at different achievement levels may experience and interact with our testing tools differently. By interviewing a total of 38 students (12 from CC, 15 from MT_{ps}, and 11 from MT_{mf} semesters) across various performance groups, we aimed to capture distinct insights and experiences, minimizing the potential skewing of data that could occur if random sampling disproportionately selected students from any one achievement level.

2. Researcher Bias and Reliability: The involvement of researchers in designing and conducting interviews, as well as analyzing the data, could have inadvertently introduced

some level of bias in the interpretation of the results. To mitigate this threat, we employed best practices in coding and inter-coder agreement, such as having the same researcher to conduct all 38 interviews to ensure consistency. We had two researchers independently code a subset of interviews and engage in iterative discussions to clarify coding definitions and resolve discrepancies. We also used qualitative data analysis software to efficiently manage and organize the coded data, reducing the potential for human error.

3. Confounding Variables: While our study aimed to investigate the impact of different testing approaches (CC, MT_{ps}, and MT_{mf}) on student testing behaviors and outcomes, other factors beyond the testing approach might have influenced the results. These confounding variables could include individual student characteristics, prior programming experience, AAT serverload, submission energy, and the influence of specific instructors or teaching assistants. Additionally, we acknowledge that the COVID-19 pandemic significantly altered educational practices and student behaviors. To mitigate potential effects of pre and post-COVID differences, we exclusively analyzed data from Fall 2021 onwards, ensuring consistency in the post-COVID learning environment. We further attempted to mitigate these threats by collecting data across multiple semesters and triangulating our findings through both qualitative and quantitative analyses.

4. Hawthorne Effect: The awareness of being observed and studied could have altered students' behavior and responses during the interviews, potentially influencing the study results. To minimize this effect, we assured participants of the confidentiality of their responses and encouraged them to share their honest opinions and experiences.

5. Limited Generalizability: Our study was conducted within a single institution and focused on a specific course (post-CS2 DSA). The findings may not be directly generalizable to other institutions, student populations, or courses with different characteristics. However, we provided detailed descriptions of our study context, participants, and interventions to

allow readers to assess the transferability of our findings to other settings.

6. Construct Validity: The effectiveness of our interventions was assessed through student performance metrics and interview data. The chosen metrics and interview questions may not have fully captured all aspects of students' testing skills, perceptions, and behaviors. To mitigate this threat, we employed a mixed-methods approach, combining qualitative insights with quantitative data from student code snapshots and IDE activity logs to gain a more comprehensive understanding of the impact of different testing approaches.

7. Novelty of MT: The introduction of MT as a new testing approach in the course might have influenced student behavior and engagement due to its novelty. As students become more familiar with MT over time, their perceptions and behaviors may evolve. To account for this, we analyzed data from multiple semesters and conducted interviews at different stages of MT implementation to capture any potential changes in student experiences.

Despite our efforts to mitigate these threats, it is important to interpret our findings with an awareness of the potential limitations. Future research could build upon our work by replicating the study in different educational contexts, employing more diverse data collection methods, and controlling for potential confounding variables to further strengthen the validity of the findings.

Chapter 6

Conclusions and Future Work

Through my comprehensive research spanning three interconnected studies, I have investigated the integration and impact of mutation testing (MT) in undergraduate computer science education. This work represents a significant contribution to the field, providing valuable insights into the computational demands, educational effectiveness, and pedagogical strategies associated with MT implementation in a post-CS2 Data Structures and Algorithms (DSA) course.

In the first study, I examined the computational impact of integrating MT into an educational auto-grading system. I found that while MT increased processing times by 30-50 seconds per submission, these effects were largely confined to the students using MT in their projects. I also demonstrated the effectiveness of mitigation strategies, such as client-side MT execution, in reducing submission volumes and server load. These findings provide a practical framework for managing the computational demands of MT in educational settings.

The second study focused on the educational effectiveness of MT in comparison to traditional code coverage (CC) metrics. Through a rigorous analysis of student submissions and interviews, I found that students using MT produced higher quality test suites and more maintainable, correct code. I also observed that MT encouraged students to adopt a more proactive, incremental approach to testing throughout the project lifecycle. These results underscore the significant pedagogical benefits of MT in fostering better software development practices.

In the third study, I investigated the development of effective pedagogical strategies for teaching MT. I found that the direct application of industry-standard MT tools, such as PIT, presented challenges for students due to the complexity of the feedback provided. In response, I developed a curated MT feedback approach, incorporating specialized documentation, training materials, and tailored feedback mechanisms. The success of this approach, as confirmed through student interviews and quantitative analysis, demonstrates the importance of adapting MT for educational contexts. Collectively, these studies provide a comprehensive framework for effectively integrating MT into undergraduate computer science curricula. By addressing the computational, educational, and pedagogical aspects of MT implementation, my research bridges the gap between industry-standard testing practices and the unique needs of computer science education. The findings offer practical guidelines for instructors seeking to incorporate MT into their courses, while also contributing to the broader goal of improving software testing education and preparing students for the challenges of professional software development.

Building upon the foundation established by my research, there are numerous opportunities for future work in this area. One promising direction is the exploration of educational strategies for teaching MT to students at different levels of programming experience. While my studies focused on advanced students in a post-CS2 DSA course, investigating the impact of MT on novice programmers could yield valuable insights into how testing practices can be effectively taught throughout the computer science curriculum.

Another potential avenue for future research is the development of more sophisticated metrics for quantifying students' use of different testing and debugging techniques. By creating a more nuanced understanding of how students approach problem-solving in their code, instructors could provide more targeted feedback and interventions to support student learning. This work could also contribute to the identification of ineffective testing and debugging

practices, allowing for the development of proactive strategies to discourage these habits.

Integrating the curated MT feedback approach developed in my research into broader programming curricula is another exciting opportunity for future work. By adapting these techniques to other courses and programming languages, researchers could assess the long-term impact of MT on students' software development skills. This work could also explore the potential for hybrid approaches that combine the strengths of CC and MT, providing students with a more comprehensive understanding of testing methodologies.

Finally, future research could investigate the role of MT in providing students with concrete feedback on code complexity and maintainability. By leveraging the insights gained from MT analysis, instructors could help students develop a deeper understanding of software design principles and best practices. This work could contribute to the development of more effective pedagogical strategies for teaching software engineering concepts in undergraduate computer science education.

In conclusion, my research has demonstrated the significant potential of MT as a valuable tool for improving software testing education in undergraduate computer science curricula. By providing a comprehensive framework for MT integration that addresses computational, educational, and pedagogical challenges, this work lays the foundation for future research and innovation in this area. As the demand for high-quality software continues to grow, equipping students with the skills and knowledge needed to develop robust, reliable code will be increasingly critical. Through continued research and collaboration, the computer science education community can work towards this goal, preparing the next generation of software developers to meet the challenges of an ever-evolving technological landscape.

Bibliography

[1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. 2010. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 153–160.

[2] William C Adams. 2015. Conducting semi-structured interviews. *Handbook of practical program evaluation* (2015), 492–505.

[3] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th international conference on Software engineering*. 402–411.

[4] Maurício Aniche, Felienne Hermans, and Arie Van Deursen. 2019. Pragmatic software testing education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 414–420.

[5] Roger Azevedo and Robert M Bernard. 1995. A meta-analysis of the effects of feedback in computer-based instruction. *Journal of Educational Computing Research* 13, 2 (1995), 111–127.

[6] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 268–277.

[7] Paul Black and Dylan Wiliam. 1998. Assessment and classroom learning. *Assessment in Education: principles, policy & practice* 5, 1 (1998), 7–74.

[8] Duane Buck and David J Stucki. 2000. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. *ACM SIGCSE Bulletin* 32, 1 (2000), 75–79.

[9] Yiqun T Chen, Rahul Gopinath, Anita Tadakamalla, Michael D Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2020. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 237–249.

[10] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis*. 449–452.

[11] Tom Copeland. 2005. *PMD applied*. Vol. 10. Centennial Books San Francisco.

[12] John W Creswell and J David Creswell. 2017. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.

[13] Murial Daran and Pascale Thévenod-Fosse. 1996. Software error analysis: A real case study involving real faults and mutations. *ACM SIGSOFT Software Engineering Notes* 21, 3 (1996), 158–171.

[14] Mickaël Delahaye and Lydie Du Bousquet. 2015. Selecting a software engineering tool: lessons learnt from mutation analysis. *Software: Practice and Experience* 45, 7 (2015), 875–891.

[15] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[16] Carol S Dweck and Ellen L Leggett. 1988. A social-cognitive approach to motivation and personality. *Psychological review* 95, 2 (1988), 256.

[17] Stephen H Edwards. 2003. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing (JERIC)* 3, 3 (2003), 1–es.

[18] Stephen H Edwards. 2004. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. 26–30.

[19] Stephen H Edwards and Manuel A Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*. 328–328.

[20] Stephen H Edwards and Zalia Shams. 2014. Comparing test quality measures for assessing student-written tests. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 354–363.

[21] Stephen H Edwards and Zalia Shams. 2014. Do student programmers all tend to write the same software tests?. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 171–176.

[22] Stephen H Edwards, Jason Snyder, Manuel A Pérez-Quiñones, Anthony Allevato, Dongkwan Kim, and Betsy Tretola. 2009. Comparing effective and ineffective behaviors of student programmers. In *Proceedings of the fifth international workshop on Computing education research workshop*. 3–14.

[23] Geoffrey K Gill and Chris F Kemerer. 1991. Cyclomatic complexity density and software

maintenance productivity. *IEEE transactions on software engineering* 17, 12 (1991), 1284–1288.

[24] Barney G Glaser. 1978. *Theoretical sensitivity.* University of California,.

[25] Michael H Goldwasser. 2002. A gimmick to integrate software testing throughout the curriculum. *ACM SIGCSE Bulletin* 34, 1 (2002), 271–275.

[26] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th international conference on software engineering.* 72–82.

[27] Braxton Hall and Elisa Baniassad. 2022. Evaluating the quality of student-written software tests with curated mutation analysis. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E.* 24–34.

[28] John Hattie and Helen Timperley. 2007. The power of feedback. *Review of educational research* 77, 1 (2007), 81–112.

[29] Marc R Hoffmann, J Brock, and E Mandrikov. 2006. Eclemma-java code coverage for eclipse. *Software, Aug* (2006).

[30] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering.* 435–445.

[31] David Jackson and Michelle Usher. 1997. Grading student programs using ASSYST. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education.* 335–339.

[32] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[33] Edward L Jones. 2000. Software testing in the computer science curriculum–a holistic approach. In *Proceedings of the Australasian conference on Computing education*. 153–157.

[34] Edward L Jones and Christy L Chatmon. 2001. A perspective on teaching software testing. *Journal of Computing Sciences in Colleges* 16, 3 (2001), 92–100.

[35] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 international symposium on software testing and analysis*. 433–436.

[36] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.

[37] Ayaan M Kazerouni, James C Davis, Arinjoy Basak, Clifford A Shaffer, Francisco Servant, and Stephen H Edwards. 2021. Fast and accurate incremental feedback for students' software tests using selective mutation analysis. *Journal of Systems and Software* 175 (2021), 110905.

[38] Ayaan M Kazerouni, Stephen H Edwards, T Simin Hall, and Clifford A Shaffer. 2017. DevEventTracker: Tracking development events to assess incremental development and procrastination. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 104–109.

[39] Ayaan M Kazerouni, Stephen H Edwards, and Clifford A Shaffer. 2017. Quantifying incremental development practices and their relationship to procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 191–199.

[40] Ayaan M Kazerouni, Clifford A Shaffer, Stephen H Edwards, and Francisco Servant. 2019. Assessing incremental testing practices and their impact on project outcomes. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 407–413.

[41] Daniel E Krutz and Michael Lutz. 2013. Bug of the day: Reinforcing the importance of testing. In *2013 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1795–1799.

[42] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: a mutation system for Java. In *Proceedings of the 28th international conference on Software engineering*. 827–830.

[43] Rifat Sabbir Mansur, Ayaan M Kazerouni, Stephen H Edwards, and Clifford A Shaffer. 2020. Exploring the Bug Investigation Techniques of Intermediate Student Programmers. In *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. 1–10.

[44] Rifat Sabbir Mansur, Clifford A Shaffer, and Stephen H Edwards. 2024. Mutating Matters: Analyzing the Influence of Mutation Testing in Programming Courses. In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1*. 151–157.

[45] Rifat Sabbir Mansur, Clifford A Shaffer, and Stephen H Edwards. 2025. The Hidden Cost of Enhancing Test Quality: An Investigation into the Impact of Mutation Testing on an Online Auto-Grader. *TBD* (2025), TBD.

[46] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[47] Joan C Miller and Clifford J Maloney. 1963. Systematic mistake analysis of digital computer programs. *Commun. ACM* 6, 2 (1963), 58–63.

[48] Thomas Muhr. 1991. ATLAS/ti—A prototype for the support of text interpretation. *Qualitative sociology* 14, 4 (1991), 349–371.

[49] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2. Wiley Online Library.

[50] Akbar Siami Namin and James H Andrews. 2009. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 57–68.

[51] Akbar Siami Namin and Sahitya Kakarla. 2011. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 342–352.

[52] A Jefferson Offutt and Jeffrey M Voas. 1996. Subsumption of condition coverage techniques by mutation testing. *Department of Information and Software Systems Engineering, George Mason University, Tech. Rep. ISSE-TR-96-100* (1996).

[53] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, YL Traon, and Mark Harman. 2019. Chapter six-mutation testing advances: An analysis and survey. volume 112 of Advances in Computers.

[54] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th international conference on software engineering*. 537–548.

[55] Andrei Papancea, Jaime Spacco, and David Hovemeyer. 2013. An open platform for managing short programming exercises. In *Proceedings of the ninth annual international ACM conference on International computing education research*. 47–52.

[56] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 910–921.

[57] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An empirical study of iterative improvement in programming assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 410–415.

[58] Raymond Pettit and James Prather. 2017. Automated assessment tools: Too many cooks, not enough collaboration. *Journal of Computing Sciences in Colleges* 32, 4 (2017), 113–121.

[59] D Royce Sadler. 1989. Formative assessment and the design of instructional systems. *Instructional science* 18, 2 (1989), 119–144.

[60] Ana B Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. 2022. Mutation testing in the wild: findings from GitHub. *Empirical Software Engineering* 27, 6 (2022), 132.

[61] Clifford A Shaffer, Ville Karavirta, Ari Korhonen, and Thomas L Naps. 2011. OpenDSA: beginning a community active-ebook project. In *Proceedings of the 11th Koli Calling International Conference on computing education research*. 112–117.

[62] Zalia Shams. 2015. *Automated Assessment of Student-written Tests Based on Defect-detection Capability*. Ph. D. Dissertation. Virginia Tech.

[63] Zalia Shams and Stephen H Edwards. 2013. Toward practical mutation analysis for evaluating the quality of student-written software tests. In *Proceedings of the ninth annual international ACM conference on International computing education research*. 53–58.

[64] Terry Shepard, Margaret Lamb, and Diane Kelly. 2001. More testing should be taught. *Commun. ACM* 44, 6 (2001), 103–108.

[65] Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189.

[66] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM Sigcse Bulletin* 38, 3 (2006), 13–17.

[67] Jaime Spacco and William Pugh. 2006. Helping students appreciate test-driven development (TDD). In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 907–913.

[68] Fabienne M Van der Kleij, Remco CW Feskens, and Theo JHM Eggen. 2015. Effects of feedback in a computer-based learning environment on students' learning outcomes: A meta-analysis. *Review of educational research* 85, 4 (2015), 475–511.

[69] Filippos I Vokolos and Phyllis G Frankl. 1998. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 44–53.

[70] Tiantian Wang, Xiaohong Su, Peijun Ma, Yuying Wang, and Kuanquan Wang. 2011.

Ability-training-oriented automated assessment in introductory programming course. *Computers & Education* 56, 1 (2011), 220–226.

[71] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. 1996. *Structured testing: A testing methodology using the cyclomatic complexity metric*. Vol. 500. US Department of Commerce, Technology Administration, National Institute of Standards and Technology.

[72] James A Whittaker. 2000. What is software testing? And why is it so hard? *IEEE software* 17, 1 (2000), 70–79.

[73] W Eric Wong and Aditya P Mathur. 1995. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal* 4, 1 (1995), 69–83.

[74] Min Xie and Bo Yang. 2003. A study of the effect of imperfect debugging on software development cost. *IEEE Transactions on Software Engineering* 29, 5 (2003), 471–473.

Appendices

Appendix A

Additional Materials

This appendix contains the following materials:

1. The semi-structured script that I followed during the interviews described in Sections 5.3 and 5.4.

2. Null results from different metrics, such as OpenDSA eTextbook assignments, midterm and final exams, milestones, etc.

A.1 Interview Script

Mutation Test Feedback Reflection Interview Script

Welcome

The interviewers introduce themselves.

Purpose

We are conducting an educational research project on software testing skills in intermediate programming courses (CS 3114/CS 5040).

Participation in this interview is purely voluntary—you will not receive any course credit, and your comments will not affect any course grade in any way. For completion of the interview, you will be compensated with a $20 Amazon gift card.

While we hold our discussion, I'll take notes on the comments that are made. However, I will not use any names or write down any personally identifying information, so everything is kept anonymous.

Please read the consent form completely and decide if you wish to participate.

If you don't wish to participate, that is no problem—you are free to leave for any reason you choose, at any time. Also, if you are a minor, we cannot include you in our research, so please decline to participate now.

[Give time to read/sign consent forms, or for participants to elect to leave.]

[Prior experience]

1. What prior experience did you have before taking the course? [Explicitly ask for CS major/minor, the experience of taking CS 1114/CS 2114, and experience of coding in Java.]
2. Did you have any prior experience of using writing test cases?
3. Did you have any prior experience in using test quality metrics, for example, code coverage, mutation testing, etc.?

[*Time-management*]

4. How much time did you **expect** to spend on CS 3114?
5. How much time did you **actually** spend on CS 3114?
    - Did it differ from your expectation? If so, how?
6. In particular, how much time did you spend on the first project in total?
7. How much time did you spend on writing the **solution** for the project?
8. How much time did you spend on writing **test cases** for the project?

[*Project Progress*]

9. How far did you get on that project?

10. How much mutation test coverage did you have in your final submission to project 1?

[*Experience with Writing Test Cases*]

11. When did you write your first test case for projects 1 and 2?
12. When do you usually write your test cases during your project development? (For example,
    - Did you write test cases first and then write the solution?
    - Did you write test cases after completing the whole project?
    - Did you write test cases after writing some functionalities of the project?

[*Mutation Testing*]

13. Did you use the Mutation Test analysis tool in your Eclipse IDE?
    - How often did you run Mutation Test analysis on your Eclipse IDE?
14. Did you use the Mutation Test analysis tool on the Web-CAT submission web portal?
    - How often did you run Mutation Test analysis on Web-CAT?
15. Which tool did you primarily use to read feedback on the mutation test?
16. Did you use any other test quality metric besides mutation testing? For example
    - Code coverage?

[*Mutation Testing Feedback*]

17. How often did you use the mutation feedback to improve your test coverage?
18. Was the hint in the feedback actionable?
19. Could you describe your experience of using the mutation test feedback to improve a test case from start to finish?
20. What parts of the mutation test feedback helped you the most?
21. What parts of the mutation test feedback helped you the least?
22. Did you have any difficulties using the mutation test feedback?
    - Please elaborate.
    - How did you overcome the difficulties?
23. If you had to improve the mutation test feedback, what aspects of the feedback would you change?
24. Was the text in the feedback difficult to understand?
    - Which parts?
    - Did you need additional help understanding the feedback?
    - How could they be improved?
25. Did mutation test feedback help you learn how to write better test cases?
    - If so, how did it help?
    - If not, what were the challenges?

[Concluding thoughts - (optional)]

26. Of all the points that have been made during our discussion, which do you think is most important?
27. Have we missed anything important about your testing process?

Thank you for participating!

A.2 Investigation of Null Results

In our comprehensive analysis of the MT implementation's impact, we investigated several metrics that ultimately showed no significant educational impact. One notable investigation focused on students' engagement with OpenDSA, an interactive eTextbook platform used in our course [61]. OpenDSA assignments, which constitute 20% of the overall course grade, provide self-paced learning modules for software testing concepts.

As shown in Figure A.1, the box plots represent the distribution of OpenDSA final scores across CC, MT_{ps} and MT_{mf} semesters. Statistical analysis of OpenDSA final scores across the three semesters - revealed minimal practical differences despite statistical significance. While the Kruskal-Wallis test showed significant differences (H-statistic = 27.8114, $p < 0.0001$), the effect sizes were notably small (Cohen's d ranging from -0.0322 to -0.1022). Post-hoc analyses found no significant difference between the traditional approach and initial MT implementation (p = 0.5422), suggesting that the introduction of MT concepts did not substantially impact students' interaction with foundational course materials.

The lack of meaningful differences in OpenDSA performance indicates that while teaching approaches varied, students' engagement with basic course concepts remained relatively consistent. This null result is particularly interesting as it suggests that the integration of more complex testing concepts (MT) did not negatively impact students' ability to grasp fundamental software testing principles through self-paced learning. However, it also indicates that improvements in advanced testing concepts did not necessarily translate to enhanced performance in basic course materials.

These findings help refine our understanding of where MT implementation most effectively impacts student learning. Several additional metrics were investigated but showed no significant differences or were deemed unsuitable for comparison:

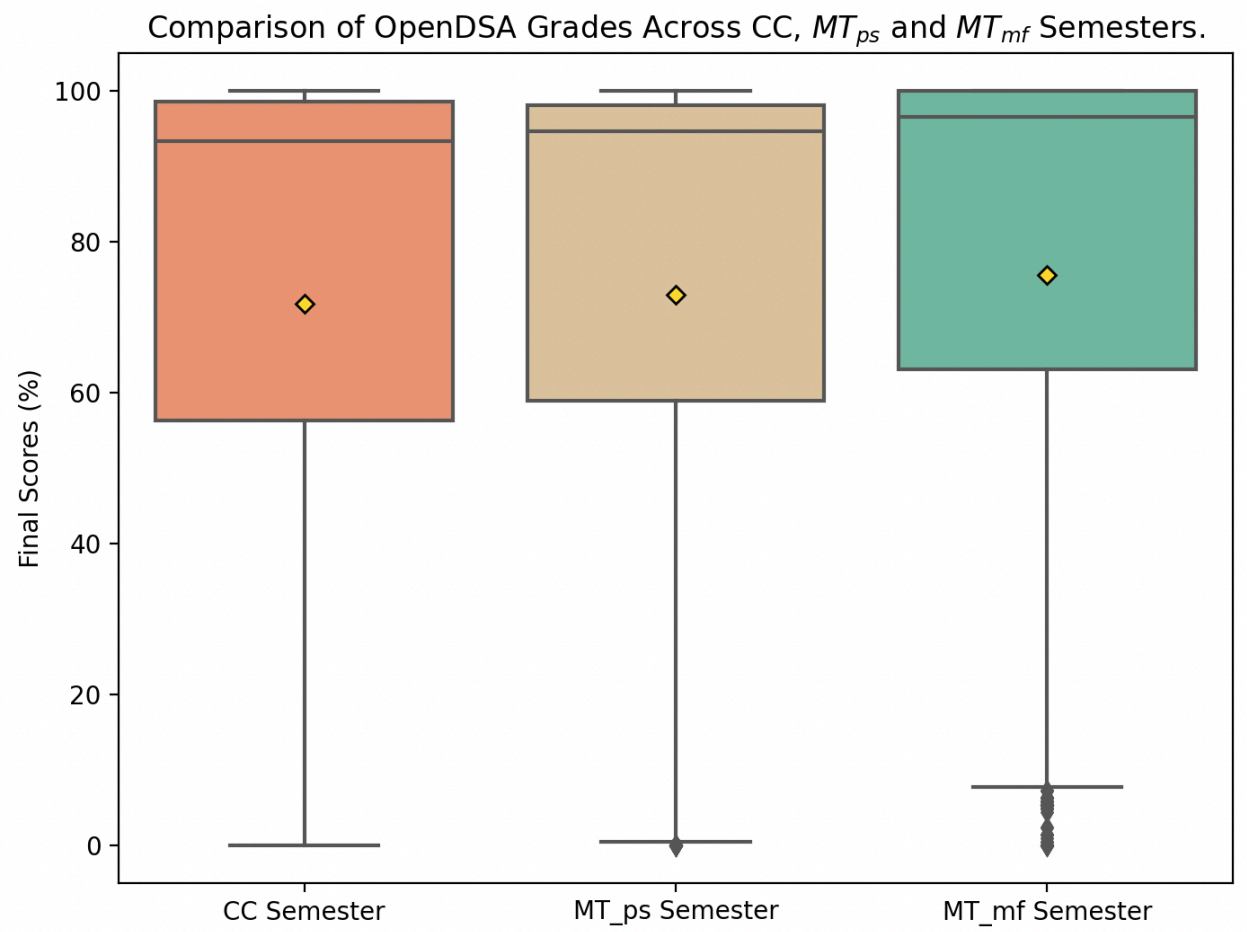

Figure A.1 Comparison of OpenDSA final scores across the three semesters: CC, MT_{ps} and MT_{mf}. We found that the introduction of MT concepts did not substantially impact students' performance of solving OpenDSA assignments.

First, we examined traditional assessment methods such as midterm and final exams. However, these evaluations varied significantly across semesters and instructors, ranging from closed-book, handwritten in-person exams to online, open-book multiple-choice questionnaires. Due to this lack of standardization, we excluded these assessments from our research to maintain analytical rigor.

Second, we analyzed project milestone completion patterns. Each project typically included 2-3 milestones and an early bonus deadline (24 hours before the final deadline). Our investigation revealed no significant correlation between MT implementation and students' milestone completion patterns or their tendency to meet early bonus deadlines.

Third, our initial approach examined test quality, project correctness, code complexity, and Web-CAT reliance on a per-semester basis. The Kruskal-Wallis tests showed minimal within-condition variations (H-statistic = 3.214, $p = 0.475$ for test quality; H-statistic = 2.891, $p = 0.512$ for project correctness). These null results led us to restructure our analysis, comparing aggregated data across CC, MT_{ps} and MT_{mf} semesters instead.

Fourth, we explored various code quality metrics including CouplingBetweenObjects and CognitiveComplexity. These measures showed strong correlation with cyclomatic complexity ($r > 0.85$, $p < 0.001$), leading us to use cyclomatic complexity as a representative metric for overall code quality.

Lastly, while we identified potential value in analyzing students' interim activities between Web-CAT AAT submissions, this remains an area for future investigation. We believe a detailed examination of these intermediate steps could reveal valuable insights into students' testing strategies and learning patterns.

These null results helped focus our research on meaningful metrics and guided our analytical approach toward areas where MT implementation showed significant pedagogical impact.