Mutating Matters: Analyzing the Influence of Mutation Testing in Programming Courses

Rifat Sabbir Mansur
rifatsm@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Clifford A. Shaffer
shaffer@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Stephen H. Edwards
edwards@cs.vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Abstract

Mutation testing is used to gauge the quality of software test suites by introducing small faults, called "mutations", into code to assess if a test suite can detect them. Although it has been applied extensively in the software industry, mutation testing's use in programming courses faces both computational and pedagogical barriers. This study examines the impact of mutation testing on student performance in a post-CS2 Data Structures and Algorithms course with 3-4 week life-cycle programming projects. We collected a semester of data with projects using only code coverage (control group) and another semester that used mutation testing (experimental group). We investigated three aspects of mutation testing impact: the quality of student-written test suites, the correctness and complexity of students' solution code, and the degree of incremental test writing. Our findings suggest that students using mutation testing, as a group, demonstrated higher quality test suites and wrote better solution code compared to students using traditional code coverage methods. Students using mutation testing were more likely to exhibit incremental testing practices.

CCS Concepts

• **Social and professional topics** → *Computing education*; • **Software and its engineering** → *Software development process management*; **Software testing and debugging**.

Keywords

software testing, cyclomatic complexity, mutation analysis, code coverage, post-CS2, data structures and algorithms

ACM Reference Format:
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 ACM Virtual Global Computing Education Conference V. 1 (SIGCSE Virtual 2024), December 5–8, 2024, Virtual Event, NC, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649165.3690110

1 Introduction

The process of developing software involves a broad set of skills that go beyond just writing code. This paper focuses on the process of mastering effective software testing. Testing is an integral part of the software development life cycle. It attempts to verify that a piece of software behaves as intended. Professional developers often depend on automated execution of unit tests to ensure that any errors caused by code changes are promptly identified. As a result, confidence in the software system depends in part upon the perceived quality of these tests.

One form of software testing, known as mutation testing [10], has become increasingly popular within the software industry. Mutation testing (MT) involves making small changes, called "mutations", to the source code, with the goal of identifying whether the test suite can catch these changes as bugs. This approach has the potential to offer a measure of the comprehensiveness and effectiveness of the test suite. Of course, a program can potentially be mutated in many ways. Good MT relies on using an appropriate set of rules for making changes to the program (the mutations) that balances effective testing with tolerable computation costs. (Computational costs are a concern because each introduced mutation requires a separate run of the program against the test suite.)

Despite its growing use in industry [4, 31, 32], the use of MT in education remains less explored. While some research has been conducted on software testing education [11, 16, 35], the use of MT in a classroom environment is still largely uncharted territory. Consequently, the potential benefits and drawbacks of this method in fostering students' software testing skills and understanding remain unclear despite its established use in the software industry.

To gain a deeper understanding of MT in an educational setting, we developed a study to assess how incorporating MT into the curriculum affects students' test suites, code quality, and project correctness. Our study group was student programmers in a post-CS2 data structures course that has a major programming component. We compared programming projects implemented during this course from semesters where MT was not used (control group), and semesters where MT was introduced into the course structure (experimental group). The control group was graded on test quality using code coverage (the percentage of lines of code executed when running the test suite), while the study group was graded on test quality based on the fraction of introduced mutations caught by the test suite ("mutation coverage"). We seek to understand the influence of MT by comparing the differences found in non-MT and MT semesters by addressing the following 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?

SIGCSE Virtual 2024, December 5–8, 2024, Virtual Event, NC, USA

ACM ISBN 979-8-4007-0598-4/24/12
https://doi.org/10.1145/3649165.3690110

Our analysis, described in Section 4, found that introducing MT in this course significantly improved the quality of student-written test suites compared to semesters grading on code coverage. But we found that MT's influence extends beyond testing, leading to higher quality code in terms of both correctness and understandability. This improvement is attributed to MT's ability to encourage students not only to write better tests but also to produce more testable code. Additionally, our analysis indicates that MT fosters a proactive and incremental approach to test case development throughout the entire project lifecycle. These findings underscore the potential of MT as a powerful educational tool in computer science education, particularly in teaching software testing and promoting better coding practices.

The paper is organized as follows. Section 2 presents a review of the literature relevant to software testing education and MT. Section 3 describes the research methodology used in order to answer the research questions that guide our study. Section 4 provides a detailed presentation of our findings. Section 5 discusses threats to the validity of the study, and Section 6 concludes with a summary of the main findings and suggestions for future research.

2 Literature Review

Software testing is critical to evaluating the correctness of computer programs, and many CS educators have incorporated software testing into the curriculum [3, 11, 18, 35]. 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, 11, 14].

A basic metric for a test suite is its number of tests. While studies by Namin and Andrews [26], Gopinath et al. [15] and Inozemtseva, and Holmes [17] indicate a correlation between test suite size and test set coverage, contradictory conclusions were drawn by Just et al. [20] and Papadakis et al. [30], suggesting a negligible correlation. To provide a more comprehensive evaluation of the test suite, we move beyond simply counting the number of test cases. Instead, we 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, 17]. Therefore, studies have shown that high code coverage scores do not necessarily correlate with bug-free software, both in educational [33, 34] and non-educational [17] settings. Some students write "pathological" tests to achieve high coverage scores for grading purposes, which defeats the purpose of testing [33]. 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 [10] 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, 28, 33, 38]. 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 [8] 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. [2] 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 [36] with real faults, evaluating 10% of generated mutants, and not controlling for code coverage. Namin and Kakarla [27] 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. [20] 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. [30] found moderate-to-strong correlations between mutant detection and fault detection when test set size was ignored. However, Chen et al. [5] 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. [24] discovered that students who participate in more testing activities typically attain higher overall scores in large programming projects. Hall and Baniassad [16] 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 [6], Major [19], and MuJava [23]. Among these, PIT (Pitest) [6] is widely regarded as the most commonly used MT tool, both in industry and academia, due to its extensive support and comprehensive documentation [32].

Delahaye and du Bousquet [9] 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. [29] 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, we chose to use a diverse range of MT tools (PIT, Major, and Mujava) in our study.

In addition to our investigation into the effects of MT on student testing, we also aim to evaluate its impact on the quality of student-written solution code. To quantify correctness, we used instructor-provided reference tests as a benchmark for measuring the accuracy of student submissions. For measuring complexity, we relied on McCabe's Cyclomatic complexity metric [25]. 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 [13]. 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 [37]. We 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.

3 Research Methodology

We have implemented a curated version of PIT with a selected set of mutators [21], 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.

3.1 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 [12], 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 [21] 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.

3.2 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 [19] and MuJava [23]. 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 [7], 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. [22]. We devised a new metric to differentiate between authentic and credit-seeking testing behavior among students, discussed in Section 4.3.2.

4 Results and Discussion

4.1 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

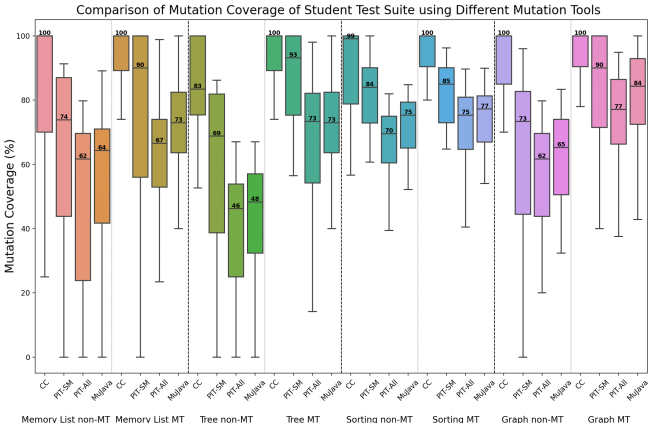

Figure 1: Comparing mutation coverage of students' test suites for four different comparable projects from non-MT and MT semesters. We used four different tools: Code Coverage (CC), PIT with selected mutators (PIT-SM), PIT with All mutators (PIT-ALL), and MuJava.

(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 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 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 1. Figure 1 uses boxplots to visualize the median test coverage values, while Table 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 inappropriate 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 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.

Table 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

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 Effects of MT on Project Solution Code

4.2.1 ***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 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 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

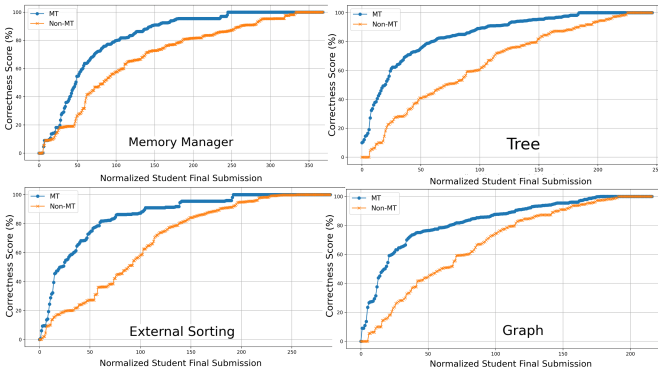

Figure 2: Comparison of correctness scores of students' final submissions for four different comparable projects for non-MT and MT semesters.

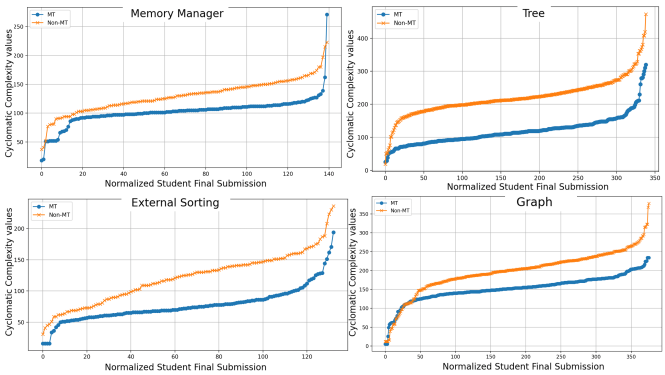

Figure 3: Distribution of cyclomatic complexities for student-written solution code (sorted low to high) for four projects across MT and Non-MT semesters. Lower cyclomatic complexity scores are considered better.

non-MT semesters, as shown in Table 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.**

4.2.2 ***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 [25] is used to assess program stability and confidence by quantifying the number of independent paths through its source code. Figure 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 1. We saw a notable disparity in total (Java class-level) mean cyclomatic complexity values between non-MT and 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**.

4.3 Effects of MT on Testing Activity

4.3.1 ***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. [22].

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 1.

4.3.2 ***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**.

5 Threats to Validity

Our study has several potential threats to validity that should be considered when interpreting the results.

Internal Validity: Students in MT semesters were graded on their mutation coverage, which might have influenced their behavior and level of effort towards improving the quality of their test suites as measured by MT metrics. This grading policy itself could have caused the observed differences between the control and experimental groups, rather than the use of MT alone. In the MT semesters, additional instructional emphasis was placed on the principles and techniques of mutation testing. This increased focus on testing could have contributed to improved student performance, rather than the use of MT itself. We do note that the improvement in cyclomatic complexity was not planned for nor were students (directly) given instructions on this topic, aside from discussions on how "dead code" will lead to unkillable mutants.

External Validity: The study was conducted within a specific educational context (a post-CS2 Data Structures and Algorithms course) at a single institution. The results may not generalize to other educational settings or programming courses with different characteristics. The observed effects of MT may also differ for projects with different complexity, problem domain, or language.

Construct Validity: While multiple metrics were used to evaluate test suite quality (code coverage, PIT, MuJava, Major), the choice of these specific metrics may not fully capture all aspects of test suite effectiveness. The measurement of student testing behaviors (incremental testing, authentic vs. credit-seeking) was based on heuristic metrics derived from IDE activity data. These metrics may not accurately reflect the true intentions or motivations behind students' testing practices.

Despite these threats, the consistent findings across multiple semesters, projects, and testing metrics provide evidence supporting the potential benefits of incorporating mutation testing into programming course curricula.

6 Conclusion and Future Work

Our study provides empirical evidence that incorporating mutation testing (MT) into programming course curricula can offer significant benefits for student learning. Our findings suggest MT leads to higher quality student-written test suites, more proactive incremental testing practices, improved correctness of solution code, and better coding practices that produce solutions with lower complexity. Overall, the adoption of MT appears to enhance students' software testing skills and project outcomes, better preparing them for industry software engineering roles.

We note that we studied relatively advanced students who have taken multiple programming courses prior to this course. The projects are of significant difficulty, typically requiring tens of hours over 3-4 weeks to complete. This tells us little about the applicability of introducing MT into earlier courses in the curriculum.

Further research is needed to evaluate the long-term impacts. While we are confident that these students wrote better projects because of the introduction of MT, we can't tell whether this will change their behavior in future courses or when they become software professionals. In the future, we intend to focus on developing more granular methods to identify credit-seeking testing behavior by analyzing the patterns in which students write test cases to meet specific coverage thresholds. Our plans include gathering information on student perceptions regarding their experience. Other potential future work includes longitudinal studies, exploring MT in different educational settings, and developing adaptive educational MT tools.

7 Acknowledgments

We are grateful to the National Science Foundation, United States of America for their support under the grant DLR-1740765.

References

[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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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).

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.