Exploring Student White-Box Testing Patterns in a Software Quality Assurance Course

UNDERGRADUATE HONOURS THESIS
COMPUTER SCIENCE, FACULTY OF SCIENCE
ONTARIO TECH UNIVERSITY
OSHAWA, ON, CANADA

Daniel Hinbest

SUPERVISORS:
Jeremy S. Bradbury, Michael A. Miljanovic

April 19, 2025

Abstract

Graduates in Computer Science often enter the workforce with inadequate software testing skills, despite software testing being an important part of the software development lifecycle. While topics such as programming and algorithms are a focus of undergraduate studies, testing typically receives less instruction. As a result, students may experience more difficulty developing effective unit tests or to apply testing practices in a real-world scenario. This study aims to achieve a better understanding of how students approach testing by identifying the patterns they form during the development process.

With approval from the Research Ethics Board at Ontario Tech University, this study was conducted during the Winter 2025 semester in CSCI 3060U - Software Quality Assurance. Students were asked to complete a short survey regarding their background and confidence in testing, and industry experience. Out of 59 respondents, 31 participated in a four-part testing activity that was created to evaluate different types of code coverage. The activity was carried out during scheduled lab time and used a plugin called TaskTracker to log students' IDE activity in real time.

Data collected by TaskTracker was analyzed using a custom Python script that categorized testing sessions into specified patterns. These included *Heavy Editing*, *Write Without Running*, *Iterative Write/Compile*, *Copy/Paste + Write + Compile*, among others. Results showed most students relied more heavily on more editing-based approaches, and relatively few were found to use iterative testing processes.

The findings suggest that students may actively engage working with test code, but may omit consistent test execution. These insights can be used to inform further improvements to testing instruction, help identify areas of weakness, and support the development of resources to encourage more effective testing habits.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The motivation for identifying patterns in student-written white-box test suites is to improve our understanding of what students are doing when they are learning to write their white-box unit tests. Previous research has shown that students often have difficulty writing effective test suites, specifically testing valid test cases, and are testing invalid input less frequently [6][13]. Falling into poor testing habits is cause for concern, as it is a crucial step in the software development life cycle.

By identifying patterns in student-written test suites, we gain insight into the specific challenges students face when learning to write effective unit tests. Recognizing these patterns helps educators better understand where students struggle, such as edge cases or failing to test invalid inputs. This knowledge can inform instructional strategies and support development of tools that provide better feedback helping to build stronger habits and improve their overall skills. Bijlsma et al. (2021) found that students often believe they are testing systematically, yet their test cases frequently lack coverage for edge cases and invalid inputs, highlighting the need for targeted educational interventions [2].

1.2 Problem

Students often experience difficulty writing white-box unit tests, a challenge well-documented in software engineering education research. Bai et al. (2021) found that students frequently struggle with designing comprehensive test cases, often omitting edge cases and invalid inputs, which are crucial for effective white-box testing [1]. This study aims to address these challenges by analyzing

the patterns students form while writing unit tests, with the primary objective of gaining a clearer understanding of student testing processes and identifying common trends that emerge during test development.

A deeper understanding of students' testing habits can provide educators with valuable insights into how they can better support education in software testing. In addition, by identifying these testing patterns, students can use these findings to improve their approach when they are writing software tests. Recognizing the gaps in knowledge and understanding may also help highlight common misconceptions or gaps in knowledge and can be addressed more directly in course materials.

By focusing on the actual behaviour when they are writing tests, this study aims to better understand what students are doing when they are developing them. The insights from this study can contribute to improving teaching strategies and give students better guidance on how to build strong, reliable tests that can be added to their software development skillset.

1.3 Research Question

The research question this work is aiming to answer is:

RQ: What testing patterns are students adopting when writing unit tests?

2 Background

A large focus in software development is testing the software to ensure that it works as intended. However, despite the widespread adoption of unit testing in the industry, there is a lack of attention to unit testing in academic programs and has led to an increasing emphasis on testing education in

computer science curricula [5]. Previous research has investigated the effectiveness of test suites by performance and quality [13]. Previous research has also been conducted studying development activities during coding activities and identifying test smells in student-written unit test suites [4][8].

Building onto this work, this study aims to explore how students are actually engaging in writing unit tests, with a focus on the patterns and behaviours that emerge during the process. By analyzing in student-written tests, we can gain insights into how students are interpreting testing concepts, apply best practices, and approach test development. This approach could reveal more successful strategies and common pitfalls, and offer useful information for educators to improve their instruction and for students who want to refine their testing skills.

2.1 Student Development Activity Research

Understand how students are writing their software code provides insight into common practices and pitfalls students are falling into during their education and addresses them early to prevent further problems caused by these practices. Previous research by Lyulina et al. [8] has studied development activity using various JetBrains integrated development environments (IntelliJ IDEA, PyCharm and CLion). They found that students would copy and paste code from other sources, take several minutes as a break to consider the steps they need to take, and execute the code at several stages to ensure their code is on the right track as they write it.

2.2 Code Coverage

Code coverage is a metric that is used to determine how much of a project's code has been tested. Many testing frameworks and IDEs offer users an option to run part or all of their test suites with coverage analysis used, and provides the user with the test coverage rate, and in some cases, in the IDE, it can also show specifically which parts of the code have been covered and which parts are not. There are three types of code coverage this study focuses on: statement, branch, and path coverage.

2.2.1 Statement Coverage

Statement coverage is a metric that is used to determine whether each line in a program has been executed at least once during testing. It is used to help identify parts of the code that have not been run by test cases and ensures that all statements have been executed. Statement coverage provides confidence in the code's execution but may not account for all paths or branches in the program. Full statement coverage is met when there is a test case for every statement in the code [7].

2.2.2 Branch Coverage

Branch coverage, also known as decision coverage, extends beyond statement coverage to evaluate all of the possible branch outcomes (e.g., true/false) for conditional statements, such as if-else statements or switch cases. It ensures every decision point is tested at least once for both outcomes and uncovers issues that can arise if only one of the branches is exercised. It is useful in spotting missing logic or unreachable code resulting from control structures. Full branch coverage is met

when there is a test case for each side of a branch [7].

2.2.3 Path Coverage

Path coverage is a metric that is more thorough than statement and branch coverage and is focused on executing all possible paths through a control flow. It considers different sequences of decisions and loops and aims to test every unique route in the program. Since the number of paths grows significantly as a program gets more complex, it can become more difficult to achieve. It is still effective to identify errors and bugs in the code. Path coverage is met when each unique path in the program has been tested [7].

3 Methodology

3.1 CSCI 3060U Course Project

The activities used for this study are from CSCI 3060U, a third year software quality assurance course during the Winter 2025 semester. Students enrolled in the course participate in a course project in groups where they learn to plan, build, and execute software tests. The project requirements are the same for all the groups and are provided by the instructor. They are not expected to implement anything that is not directed by the requirements or their Teaching Assistants.

The course project is divided into six phases, with the focus on requirements, continuous testing, and agile development. This research focuses on phase five. This phase of the project is when students have completed the back-end prototyping and are now building white box unit tests on the code they wrote in phase four.

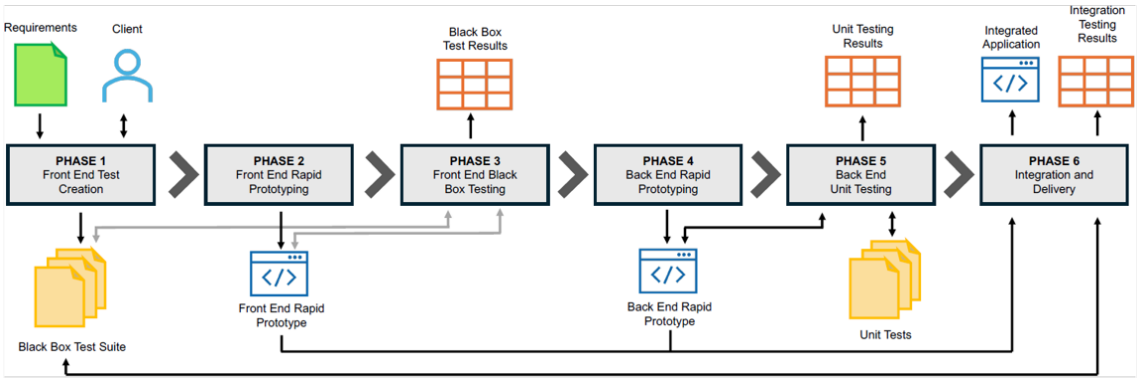

Figure 1: Timeline for the CSCI 3060U Course Project

3.2 TaskTracker-tool

TaskTracker-tool is a plugin developed by the ML4SE Lab @ JetBrains Research, consisting of three main components: the plugin, the server, and the post-processing tool [8]. The plugin is designed for various JetBrains integrated development environments (IDEs), including IntelliJ IDEA, PyCharm, and CLion. It allows students to select and complete tasks while tracking their code and IDE activity. Once a task is completed, the students can submit the code through the plugin, which uploads the task data to the server [9].

The server handles database management, including storing task-related data, handling files uploaded via the plugin, and providing APIs for data retrieval and analysis, and creating and editing the tasks [11]. Researchers can then use the post-processing tool to analyze this data. After retrieving files from the server, the tool processes the data and generates visualizations of students' coding activities, aiding in further research and insights [10].

3.3 Experimental Design

The experimental design takes place in the lab sections of CSCI 3060U in the winter 2025 term during phase five of the course project, where students implement unit tests in their projects. Students complete a series of four activities in their lab sections with the purpose of practicing unit testing with Pytest. Students were able to complete the exercises regardless of whether they consented to the research or not and would receive a bonus mark towards the project for completing the exercises. However, only students who consented to participate in the study would do so using TaskTracker and have their data collected for the study.

In each lab section, students were shared details about the activities they were invited to complete and the goals of the study. Students interested in participating in the study were then provided a survey to complete that covers basic information on their education and industry experience, such as whether they had previously had to write unit tests as part of their industry experiences. After this, they were provided a zip file of the TaskTracker plugin and instructions for its installation into the PyCharm IDE. Once the installation process is completed, TaskTracker connects to the server hosted within the university and they can begin the tasks.

To complete the tasks, students would select a task within TaskTracker and work through them. There were four tasks for students to complete for the activity:

1. Statement coverage for a simple error output
2. Statement coverage for reading from a text file
3. Branch coverage for writing to a text file
4. Path coverage for reading from a text file

After each task is completed, they can submit them through TaskTracker through the server and move onto the next task. After all the tasks are completed, participants proceeded to remove TaskTracker from their PyCharm installation.

After all the lab sections had the opportunity to complete the activities, the TaskTracker data was retrieved and anonymized to move onto the post-processing stage. This step involved the use of the post-processing tool, which was written to process the data. The data from TaskTracker is stored as two CSV files for each task, the first file is the Activity Tracker item, which collects the IDE activity, and the second is the task data item, which tracks the code changes while the task is in progress. At this point, I could proceed to the analysis stage, which was done by writing Python scripts to prepare and clean the data into CSV formats and providing summaries of some of the key metrics that were being analyzed.

4 Analysis

4.1 Participant Demographics and Background

Table 1: Participant Background Based on Survey Responses

Year of Study			**Confidence in Unit Testing**		
Answer Options	**Count**	**Percentage**	**Answer Options**	**Count**	**Percentage**
Year 3	39	66%	1 - Not confident at all	6	10%
Year 4	17	29%	2	9	15%
Year 5+	3	5%	3 - Moderately confident	33	56%
			4	10	17%
			5 - Extremely confident	1	1%
Industry Experience			**Prior Unit Testing Experience**		
Answer Options	**Count**	**Percentage**	**Answer Options**	**Count**	**Percentage**
Yes	20	34%	Yes	6	30%
No	39	66%	No	14	70%

Note: Percentages for "Prior Unit Testing Experience" are calculated based on the number of participants who indicated they had industry experience.

The first part of the study was to conduct a survey among the students participating in the study. In total, there were 72 total responses to the survey, with 10 incomplete responses, and 3 duplicates. After excluding these from the count, there were 59 total participants in the survey. This was focused on basic information regarding their year of study, their confidence level in the development of software tests, an understanding of their professional experience in Computer Science-related roles, and whether they had been exposed to writing unit tests in their positions.

CSCI 3060U is considered a third-year course, and is also available to other upper year students enrolled in a 4th year or higher, but is not available for students in an earlier year of the program, so the responses were limited to the years of study that would be able to take the course. The responses showed that despite this, the largest portion of students enrolled in the course were third year students, counting for 39 of the survey responses. There were also 17 fourth year students who completed the survey of the students in the study. Since the program is a standard four-year program, and there are not as many students who return for a fifth year, it isn't too surprising to see 95% of the responses being from third and fourth year. However, some students do opt for an extra year and there were three who responded to the survey and made up 5% of the responses.

The next question in the survey was to get a better understanding of students' confidence in their unit testing abilities. This was structured as a Likert scale that ranged from 1 being *Not confident at all* to 5 being *Extremely confident*. In total, the majority of the responses were moderately confident, with 33 of responses. There was also a moderate number of students with other confidence levels, with 6 students with low confidence and 9 in between low and moderate-level confidence. There was a smaller percentage of students who showed high confidence in their abilities, with just one student saying they were extremely confident, and 10 being in between moderate and extremely confident in their abilities. This shows that students tend to have confidence issues when they are

tasked with writing unit tests.

In this study, we were also interested in having a clearer understanding of the industry experience. In the program, there are opportunities for co-ops and internships, among other opportunities that may arise, so we were interested in how this breaks down across the participants, for industry focused employment but also who has previously had experience with writing unit tests while they were working in a position relevant to the computer science field. In total, there were 20 students who had prior or ongoing industry experience, and 39 who had not gotten any experience yet. For the students who did have industry experience, we also asked them while employed, if they had been asked to develop unit tests as part of their roles. Of the 20 students who had worked in a related position, six of them have previously written unit tests, and 14 have not.

Overall, the results from the survey show that students who are enrolled in the course typically have limited or no prior experience in unit testing, which should not come as too much of a surprise because this is the course that is designed to educate students on software testing and quality assurance, and the importance of it in the software development lifecycle. At the time that this study was conducted, students have done some work on unit testing in their course projects. Despite this, based on the survey responses, we can see that there remain lower levels of confidence in students' abilities. This is shown with 82% of students reporting low to moderate levels of confidence, compared to just 18% that entered a 4 or 5 input in the question.

4.2 Task Participation Overview

When students used TaskTracker for the study, not everyone was able to participate in the study, or if they did, not everyone was able to complete the full study. In total, of the 59 survey respondents,

Table 2: Task Participation Summary

Type	Count
Survey Respondents	59
Participants	31
Task 1	31
Task 2	28
Task 3	24
Task 4	23
Total Tasks Submitted	**106**

31 proceeded to also participate in the study, and all 31 of those were able to complete Task 1. After that, there were 28 submissions for Task 2, 24 for Task 3, and 23 for Task 4, leading to a total of 106 task submissions overall. There are a few possible explanations for the range between the total number of survey respondents compared to the participants in the testing activity, and the decreasing number of submissions as the activity went on.

The first explanation would be technical issues. One notable issue that was known about and shared with the students during the lab was due to compatibility issues of TaskTracker on Apple Silicon-based Macs. During the plugin's setup and development, there were several adjustments that were made and was primarily done so with a Windows computer, and there was an issue where Mac computers would be unsuccessful due to a compatibility issue with a JavaFX dependency for the plugin. Despite this, several students still attempted to use the plugin but were ultimately unable to proceed and their work was not used for the study. Aside from this, other students who were using Windows issues may have come across issues of their own, and many did seek assistance in the lab to resolve the issues, but ultimately some of the students may have decided not to continue with the plugin as a result of these issues, and ended up completing the activity separately as well.

The rate of students who completed all the tasks may have resulted from time constraints itself. The activity was intended to be completed in the duration of an 80-minute lab section, and this

time constraint may have prevented students from completing the full study because they were working to complete the earlier tasks. As a result, later tasks, particularly Task 3 and Task 4, saw a drop in submissions, which may not necessarily reflect disengagement but rather the limited time available. This pattern is important to consider when analyzing the quality and completeness of student submissions, as students may have prioritized earlier tasks or run out of time before reaching the later tasks.

4.3 Patterns in Student Testing

There were a few key patterns that came up in the data from TaskTracker. To guide the analysis of these patterns, several terms were defined to consistently categorize the student behaviour across the four tasks. These definitions helped to establish a shared understanding of what constituted each of the four main testing patterns that were found in the submissions.

- **Copy/Paste + Write + Compile**

  This is a pattern that is characterized using clipboard operations (cut, copy, paste), followed by code writing and executing the code. This behaviour suggests the reuse or adaptation of existing code, and possibly from external sources or previous attempts.

- **Iterative Write/Compile**

  Iterative write/compile is where students make code changes that are followed by more frequent testing. This suggests there is a more test-driven focus or an incremental approach, and students compile and run their code several times while making changes.

- **Heavy Editing**

A pattern with heavy editing involves significant code modification or cleanup and could include more frequent write events or a higher number of delete events. This could suggest there were more refactors, rewrites, or troubleshooting scenarios.

- **Write Without Running**

  This pattern occurs when students are writing code but are not testing it throughout the lab session. Some cases that could play a factor in this are time constraints, reasoning, or executing the changes in a command line format instead of using the PyCharm test execution features.

There were three smaller testing patterns that were analyzed: running with minimal changes, writing the entire test, then executing, and read only - where there were no changes, and are likely caused by no testing code written during the study. These patterns made up a combined total that was less than 5% of the overall results and were therefore combined into a miscellaneous category.

4.3.1 Distribution of Testing Patterns

Overall, there were 106 total submissions through TaskTracker over the course of the study, and one of the things we wanted to look at was what was the overall most distribution of the testing patterns that were observed during the activity. After running a Python script that processed the data built a plot of the data. In Figure 2, we can see a graph that shows the overall percentage distribution of patterns that came up across all four testing activities. This visualization helped to reveal broader trends in student testing behaviours.

We can see that many of the students were writing their unit tests with a *Heavy Editing* approach, making up nearly half of the task submissions, and by far the most common approach

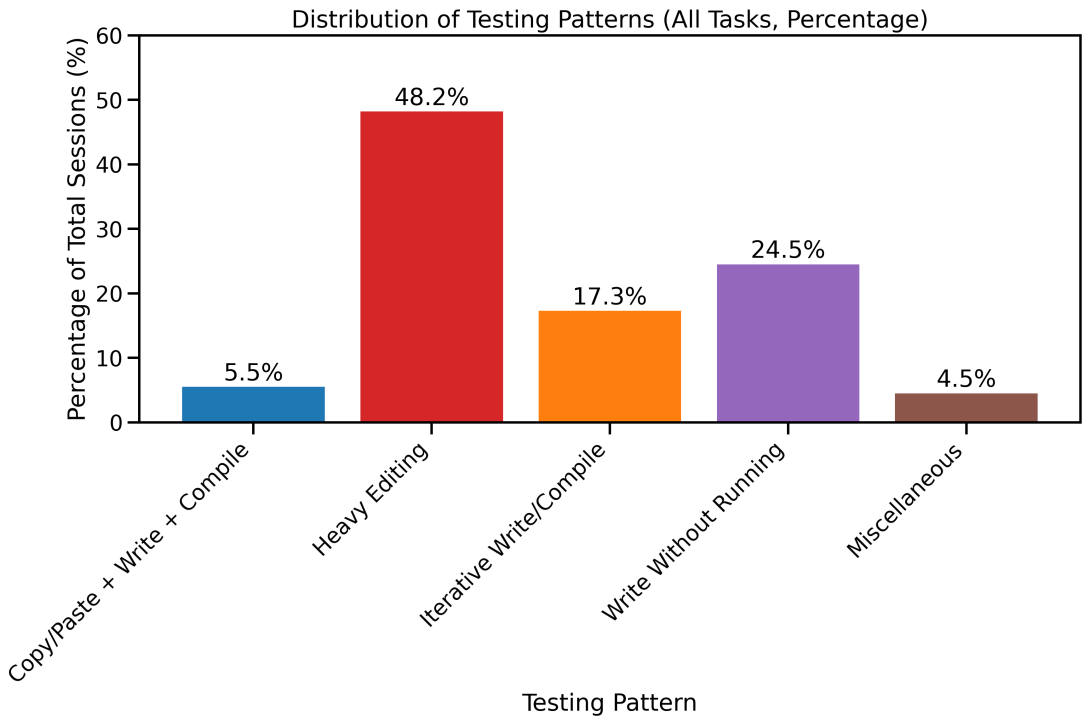

Figure 2: Distribution of testing patterns observed across all tasks.

to the activity. The next largest pattern was *Write Without Running*, which makes up 24.5% of the tasks that were written using this approach. After that, the next pattern was *Iterative Write/Compile*, which was made up of 17.3% of the task submissions. The smallest of the tasks that this study focuses on was *Copy/Paste + Write + Compile*. There were much fewer instances of this, making up just 5.5% of the overall cases. The last portion of the cases were smaller, miscellaneous cases that include a combination of writing the tests fully before executing, minimal changes, or read-only that are the remaining 4.5%. These on their own were not large enough to call for an individual analysis.

This shows that in many cases students make frequent changes to their tests and copy and paste less frequently. There are also a decent number of students who may not have had as many changes and adjustments to their test suites, but were still adjusting or writing it, but were not found to

have run it as often. There were fewer than expected students who were found to be following the *Iterative Write/Compile* pattern, which suggests there is less of a test-driven development focus at this phase of the software development lifecycle.

4.3.2 Testing Pattern Distribution by Task

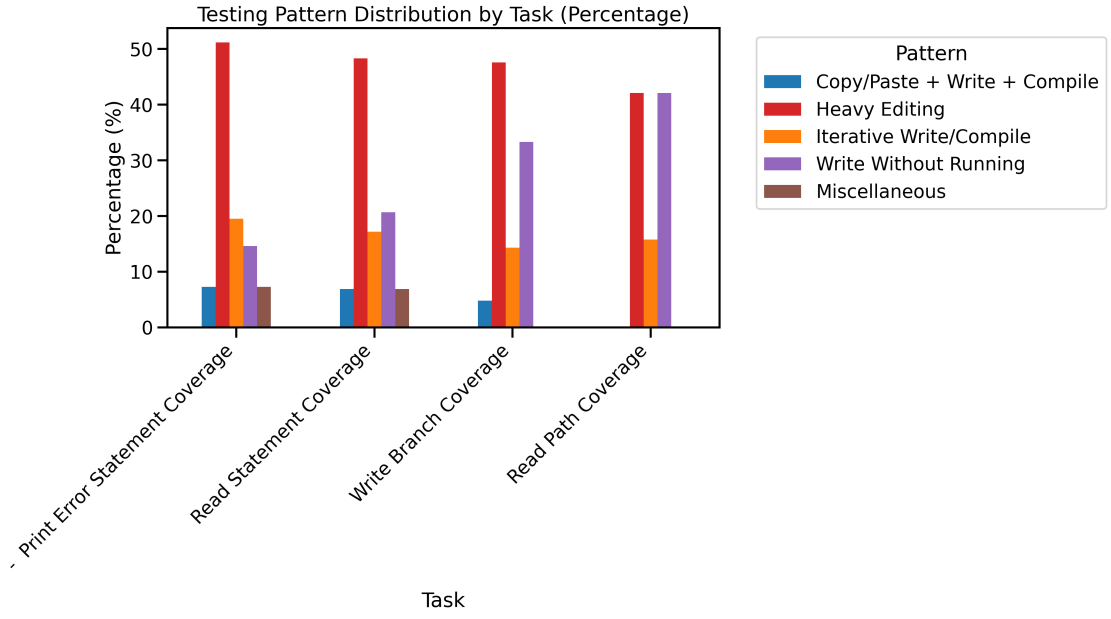

Figure 3: Testing pattern distribution by task, shown as percentage of sessions.

In Figure 3, the five types of test patterns we found were split across the four tasks. Each task was designed to focus on a different aspect of code coverage, and the distribution of patterns reflects subtle shifts in how students approached them. The first task, which was the print error statement coverage, showed a significant number of students following the *Heavy Testing* pattern, while *Iterative Write/Compile*, *Write Without Running*, *Copy/Paste + Write + Compile*, and *Miscellaneous* have been more evenly distributed. This pattern continues in the second task, which was read

statement coverage as well, although the key differences were a slight increase in *Write Without Running* instances, and a small decrease in *Heavy Editing* cases.

In the third task, write branch coverage, the gap between *Heavy Editing* and the other tasks started to close a bit more. While we still saw similar reporting for *Heavy Editing* compared to the second task, the rate of input that included writing without running started to climb. This also saw a decline in instances of *Copy/Paste + Write + Compile* and *Iterative Write/Compile*. However, in the final task, read path coverage, we saw *Write Without Running* grow to equal *Heavy Editing*. There was also a slight increase in *Iterative Write/Compile*, with no instances of *Copy/Paste + Write + Compile* or *Miscellaneous* patterns.

This data tells us that early on, the students focused heavily on adjusting the code and may have had a lower confidence in their testing abilities at this point. However, over time, there was a gradual decrease of *Heavy Editing* cases and there were more students who were writing their cases but maybe not executing them as often. In addition, there was a small decrease in the *Iterative Write/Compile* cases in the first three tasks but rebounded slightly in the fourth task. This suggests that there was growing confidence as they were not copying and pasting their code as often and were writing them on their own more. However, the lack of execution is concerning, but it is possible that in some of these cases, they may not have been running their tests in PyCharm and instead running them in the command line or after the session.

4.3.3 *Pattern Duration Distribution*

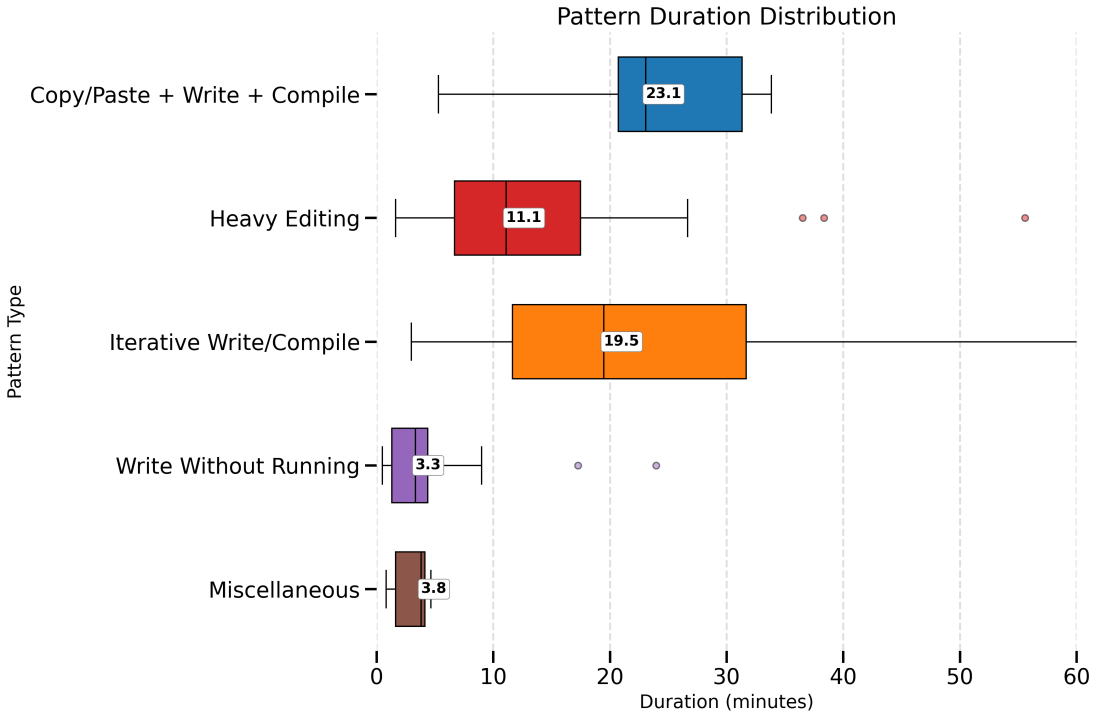

Figure 4: Distribution of session durations by testing pattern, in minutes.

Another area that was worth exploring was the average duration of each of the testing patterns. Earlier analysis focused on how often each of the testing patterns occurred, but understanding how long students were spending in each pattern helps us to offer further insights into their level of engagement. As shown in Figure 4, patterns such as *Copy/Paste + Write + Compile* and *Iterative Write/Compile* had noticeably longer durations on average, with the mean length of those sessions being 23.1 and 19.5 minutes, respectively.

This suggests that these approaches required a more sustained effort or more attention, and possibly involved more complex debugging, alterations, or iterations in the development process. However, *Heavy Editing* sessions had a shorter average duration of 11.1 minutes, which could re-

flect quicker trial-and-error adjustments. The shortest sessions in the study were the *Write Without Running* and *Miscellaneous* categories, both of which averaged under five minutes. These shorter durations may reflect a lower level of engagement or faced constraints such as time pressure or technical issues.

Since task complexities increased as the study went on, it is possible that there were some variations that were influenced by the task itself. However, because the chart is based on all tasks combined, the comparison highlights relative differences in how much time each behaviour tends to take on average, regardless of the task. This helps to build a clearer picture of which testing patterns required more involvement and which needed less effort in the testing process.

5 Conclusions

5.1 Summary & Conclusions

Testing is a highly valuable component towards the development of high quality software, and often students lack the adequate knowledge of testing concepts. This thesis explored student-written white box tests written by upper-year university students to reach statement coverage, branch coverage, and path coverage. The goal was to identify the patterns students are forming when they are writing their unit tests. This study identified four main types of testing patterns that appeared in the testing activity: *Copy/Paste + Write + Compile*, *Heavy Editing*, *Iterative Write/Compile*, and *Write Without Running*. The results tell us that students are often writing their tests with frequent edits and in many cases were not running their test cases in PyCharm. The submissions also shows there tends to be fewer instances of test execution as the testing goes on, which suggests students

may not be running the tests frequently in the development process.

5.2 Limitations

The limitations of the research in this study are:

- **Technical difficulties**

  Students who were using Mac computers experienced compatibility issues and were therefore unable to complete the study. Other students who were using Windows computers experienced some technical difficulties of their own that led to fewer submissions from some of them.

- **Use of command-line vs. PyCharm actions**

  TaskTracker only tracks code executions and other IDE activity if they are done using PyCharm. While it does support many Python test frameworks, such as Pytest, students may instead have opted to use the command line to run their test suites and were therefore not detected to have run their tests during the activity.

- **Time constraints**

  The testing activity was conducted during an 80-minute lab session, and they were intended to complete all four tasks. Time might have played a factor in a drop-off of submissions across the tasks.

5.3 Future Work

- **Analysis of Test Smells**

  What test smells are most common in student-written unit tests? was originally planned as the second research question in this thesis. However, due to time constraints stemming from pending REB approval, and the timing of the lab sessions to conduct the study, this was not able to be conducted, but can be done by extracting the code from the CSV files into Python files and running Pytest-Smell to detect test smells [3].

- **Deeper analysis of task data**

  There were large amounts of data available from the task submissions, but due to time constraints, there was limited opportunity to go into more detail on the trends found during the study. Future work could involve a more detailed breakdown of specific coding behaviors, such as how assertion use, test structure, or error handling evolved across different tasks and participants.

- **Improved tool stability and updated iteration of TaskTracker**

  During this study, students experienced technical issues related to the plugin setup, most notably with operating system compatibility. More recently, a newer version of TaskTracker has been released and may address some of the complications this research faced. Future work could involve using the updated Kotlin-based tool to ensure a smoother experience for setup, usage, and data collection across all platforms [12].

References

[1] Gina R. Bai, Justin Smith, and Kathryn T. Stolee. How students unit test: Perceptions, practices, and pitfalls. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, page 248–254, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Lex Bijlsma, Niels Doorn, Harrie Passier, Harold Pootjes, and Sylvia Stuurman. How do students test software units? In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 189–198, 2021.

[3] Alexandru Bodea. Pytest-smell: a smell detection tool for python unit tests. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 793–796, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Kevin Buffardi and Juan Aguirre-Ayala. Unit test smells and accuracy of software engineering student test suites. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, page 234–240, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Kevin Buffardi, Pedro Valdivia, and Destiny Rogers. Measuring unit test accuracy. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 578–584, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Stephen H. Edwards and Zalia Shams. Do student programmers all tend to write the same software tests? In *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*, pages 171–176. ACM Press.

[7] Cem Kaner. Software negligence and testing coverage. *Proceedings of STAR*, 96:313, 1996.

[8] Elena Lyulina, Anastasiia Birillo, Vladimir Kovalenko, and Timofey Bryksin. Tasktracker-tool: A toolkit for tracking of code snapshots and activity data during solution of programming tasks. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 495–501, New York, NY, USA, 2021. Association for Computing Machinery.

[9] JetBrains Research. Tasktracker plugin, 2020. Accessed: 2025-03-06.

[10] JetBrains Research. Tasktracker post-processing, 2020. Accessed: 2025-03-06.

[11] JetBrains Research. Tasktracker server, 2020. Accessed: 2025-03-06.

[12] JetBrains Research. Tasktracker 3, 2025. Accessed: 2025-04-19.

[13] Amanda Showler, Michael A. Miljanovic, and Jeremy S. Bradbury. How effective and efficient are student-written software tests? In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSETS 2025, page 1057–1063, New York, NY, USA, 2025. Association for Computing Machinery.