EXAMINING INTRODUCTORY COMPUTER SCIENCE STUDENT COGNITION WHEN TESTING SOFTWARE UNDER DIFFERENT TEST ADEQUACY CRITERIA

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Austin Shin

August 2022

COMMITTEE MEMBERSHIP

TITLE:	Examining Introductory Computer Science Student Cognition When Testing Software Under Different Test Adequacy Criteria
AUTHOR:	Austin Shin
DATE SUBMITTED:	August 2022
COMMITTEE CHAIR:	Ayaan Kazerouni, Ph.D. Assistant Professor of Computer Science
COMMITTEE MEMBER:	Bruno da Silva, Ph.D. Assistant Professor of Computer Science
COMMITTEE MEMBER:	John Clements, Ph.D. Professor of Computer Science

ABSTRACT

Examining Introductory Computer Science Student Cognition When Testing Software Under Different Test Adequacy Criteria

Austin Shin

The ability to test software is invaluable in all areas of computer science, but it is often neglected in computer science curricula. Test adequacy criteria (TAC), tools that measure the effectiveness of a test suite, have been used as aids to improve software testing teaching practices, but little is known about how students respond to them. Studies have examined the cognitive processes of students programming and professional developers writing tests, but none have investigated how student testers test with TAC. If we are to improve how they are used in the classroom, we must start by understanding the different ways that they affect students' thought processes as they write tests.

In this thesis, we take a grounded theory approach to reveal the underlying cognitive processes that students utilize as they test under no feedback, condition coverage, and mutation analysis. We recorded 12 students as they thought aloud while creating test suites under these feedback mechanisms, and then we analyzed these recordings to identify the thought processes they used. We present our findings in the form of the phenomena we identified, which can be further investigated to shed more light on how different TAC affect students as they write tests.

ACKNOWLEDGMENTS

I would like to begin by thanking my advisor, Dr. Ayaan Kazerouni, for his endless guidance and the patience that he had for me throughout my first research experience. From taking his Teaching Computing class to running the Tutoring Center to writing this thesis, he has been my biggest academic inspiration, and I hope to one day be able to give my students as much wisdom and advice as he gave me. Your constant support and all of your answers to my questions got me through this process, and I am truly thankful for everything.

To my committee members Dr. John Clements and Dr. Bruno da Silva, I would like to thank you both for your time and your willingness to attend my defense over Zoom. It was nice flipping the tables and lecturing to both of you for an hour, and the edits and comments you gave me were extremely helpful in ensuring my thesis was as well-rounded and polished as it could be. John, teaching 231 with you was an absolute pleasure and I had so much fun. Teaching was always something I wanted to do and you were the best guide I could have asked for throughout the whole process. Bruno, your edits and feedback during our meetings were invaluable and your perspectives kept me and Ayaan from getting too tunnel-visioned.

To my family, without you all, I would not have been able to get to where I am today. The love you all showed me throughout my life is more than any one person could ever hope for, and as we say, it's probably the reason I'm tall. With the last of the kids going off to college it's been harder to get everyone in the same place at the same time but I'm always thinking of all of you and I can't wait for our next gathering.

When I entered my fifth year at Cal Poly I was worried that I wouldn't have any friends left, but my friends new and old who were there with me for my super-senior year were the best support system I could have asked for. You all kept me on task(s) and kept me from staying in my house all the time, which are both things that I struggle with. The memories I made with you all this past year are the ones that I will cherish for the rest of my life and I love you all.

To my students and WOWies, you were the ones who helped me mature and learn more about myself than I ever could in any class. My time with each and every one of you was precious and I enjoyed every second that we spent together, whether it was kayaking in Morro Bay or learning about NumPy in that ventillationless classroom. Thank you all for providing me with the opportunities to teach and to guide you, I will always refer to you all as my kids.

This work was supported by the Warren J. Baker and Robert D. Koob Endowments.

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Software testing is a vitally important skill that is relevant in all domains in computer science. In industry, establishing code quality can guarantee patient care, safeguard customer data, and keep airplanes in the sky, and in academia, it can ensure that research findings are accurate and unbiased. To measure the efficacy of test suites, test adequacy criteria (TAC) are used, which are standards for judging a software test suite that defines what conditions need to be met for the set of tests to be deemed "adequate" [1].

Quality software testing instruction rarely finds its way into introductory computer science curricula [2, 3, 4] despite its importance to the field, leading to students forming bad habits and not developing the skills that they need [5, 6]. However, as indicated by an upward trend of paper publication and pedagogical approach, course-ware, and tooling development [7], improvements are being made. For instance, TAC are being used as an educational tool for providing students feedback on their test suites.

To further advance this teaching method, the first step that must be taken is understanding student thought processes as they test under various TAC. There have been studies into the cognitive processes of professional software engineers as they write tests [8, 9] and into student thinking as they write code [10], but there has not been any work done targeted specifically toward student testing thought processes. Furthermore, no studies regarding tester cognition under different TAC exist.

The aim of this study is to fill that void, providing insight into students' cognitive processes when testing while being guided by different TAC. Chapter 2 provides background information on the specific TAC that were used in this study as well as relevant findings from the aforementioned studies. Chapter 3 describes the implementation of the data collection tool and related components and Chapter 4 details the data collection and analysis methodology. Finally, Chapter 5 outlines our findings and Chapter 6 concludes this study and presents possible future work.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 Test Adequacy Criteria

The effectiveness of different TAC in all fields of computer science has been studied for decades, but the two main ones related to this thesis are code coverage and mutation analysis (MA). Code coverage is a type of structural testing criterion that evaluates how many lines and/or branches of source code are executed by a given test suite [11] and it is widely used due to its cheap generation cost and easily understandable feedback. On the other hand, MA is a type of fault-based testing criterion that involves artificially inserting bugs into a program and measuring how well a test suite can discover them, but it has limited practical applications due to its high computational cost.

2.1.1 Condition Coverage

The type of code coverage used in this study is condition coverage (CC). CC examines the statements of a program and considers each one "fully covered" if all of its branches are executed (e.g., the conditional for an *if* statement evaluates to both *True* and *False*), "partially covered" if only some of its branches are executed (e.g., the conditional for an *if* statement only evaluates to *True*), and "uncovered" if none of its branches are executed (e.g., the statement is never reached). A coverage score, represented as a percentage of statements that are covered, is commonly used to measure a test suite in terms of code coverage.

Despite how common it is, the shortcomings of code coverage in test suite evaluation in education have been widely acknowledged. Code coverage is not a good indicator of the true fault-finding capabilities of a set of tests [12, 13, 14] and does not strongly correlate to software reliability [15, 16, 17].

2.1.2 Mutation Analysis

In MA, the artificially-created bugs that are created are called "mutations", and the defective copies of the program that result are called "mutants." Generally, mutations are small syntactic changes (e.g., changing a *0* to a *1* or changing $>$ to $>=$) that are categorized into different types called "mutation operators." While there are infinitely many possible mutation operators, they have been designed to emulate commonly-made errors made by programmers [18].

Similar to CC, a mutation score, commonly represented as a percentage of "killed" to total mutants, is used to evaluate the thoroughness of a test suite. A mutant is considered "killed" if at least one test case in a suite fails upon encountering that mutant, and is considered to have "survived" otherwise [18]. However, some mutations result in a mutant that functions identically to the original program, making it impossible to kill and thus skewing the mutation score. Detecting these mutants, known as equivalent mutants, is an important problem in MA, but is undecidable as it is essentially the program equivalence problem. In this study, any equivalent mutants were manually identified by researchers and were not counted in the number of total mutants.

Empirical studies have found that there is a correlation between the mutant-finding capabilities and real fault-finding capabilities of both developer- and automatically-created software test suites [19, 20]. Additionally, developers have been shown to write more and better tests over time while testing under MA [20]. As for students testing with MA as a TAC, correlations have been found between mutation scores and manual instructor-created fault detection rates as well as between mutation scores and all-pairs grading scores (a student's program and test suite are run against their peers', combining the percentage of test suites for which their code passed and the percentage of flawed programs that did not fail their test suite [21]) [22].

Studies directly comparing code coverage and MA as educational tools have suggested MA as the cure for the shortcomings that using only code coverage can bring. One study compared their usefulness as automated test assessment tools and found that MA evaluates the strength of students' test suites, measures the correctness of their source code, and can identify students who try to fool automated coverage systems with meaningless tests [12]. Another study found that, while student tests achieved an average of 95.4 percent code coverage, they only detected 13.6 percent of faults across the entire population of student source code [13]. MA has also been proven to subsume CC, meaning a test set with a 100 percent mutation score will have a 100 percent code coverage score [23].

MA is the most powerful TAC that exists [24] and it has been proven to be an effective educational tool, but it is not without its drawbacks. Certain mutation operators may not be representative of probable faults in some languages [25] and MA feedback can be hard for students to interpret [12]. But perhaps its most glaring deficiencies are its computational expense and long runtime, as test suites must be run against potentially hundreds or thousands of copies of a potentially complex program.

2.2 Related Work

The related works to this thesis primarily deal with programmer and tester thought processes. The approach of treating software testing as a problem-solving task was only first explored recently, so not many studies exist yet.

One of the inspirations for this thesis was the study by Castro and Fisler in which they examined how novice programmers shifted between thinking at the task level (focusing on task decomposition) and at the code level (focusing on the source code) while coding [10]. They noticed that the way they moved between the two levels of thinking corresponded to their success on a problem, with those alternating between task-level and code-level thinking faring the best, and those starting at the task level and then staying at the code level or solely focusing on code being the farthest from a correct solution. These findings motivated our original research topic of trying to find similar patterns in students while they tested under different TAC, and while we deviated from this path, in our analysis we considered which level(s) of thought students were using throughout the testing process.

The first framework describing the cognitive processes of software testers (see Figure 2.1) was created by Enoiu et. al. [9]. Their model lays out the task of testing as a cyclical problem-solving model, with testers repeatedly identifying and understanding test goals, planning a testing strategy, writing and executing tests, and evaluating test results. TAC are examples of test goals, so one idea we considered was observing how students would move between the stages of the framework differently when writing tests under different ones. While we did not follow this path for this study, we kept this framework in mind during our analysis to better categorize student actions.

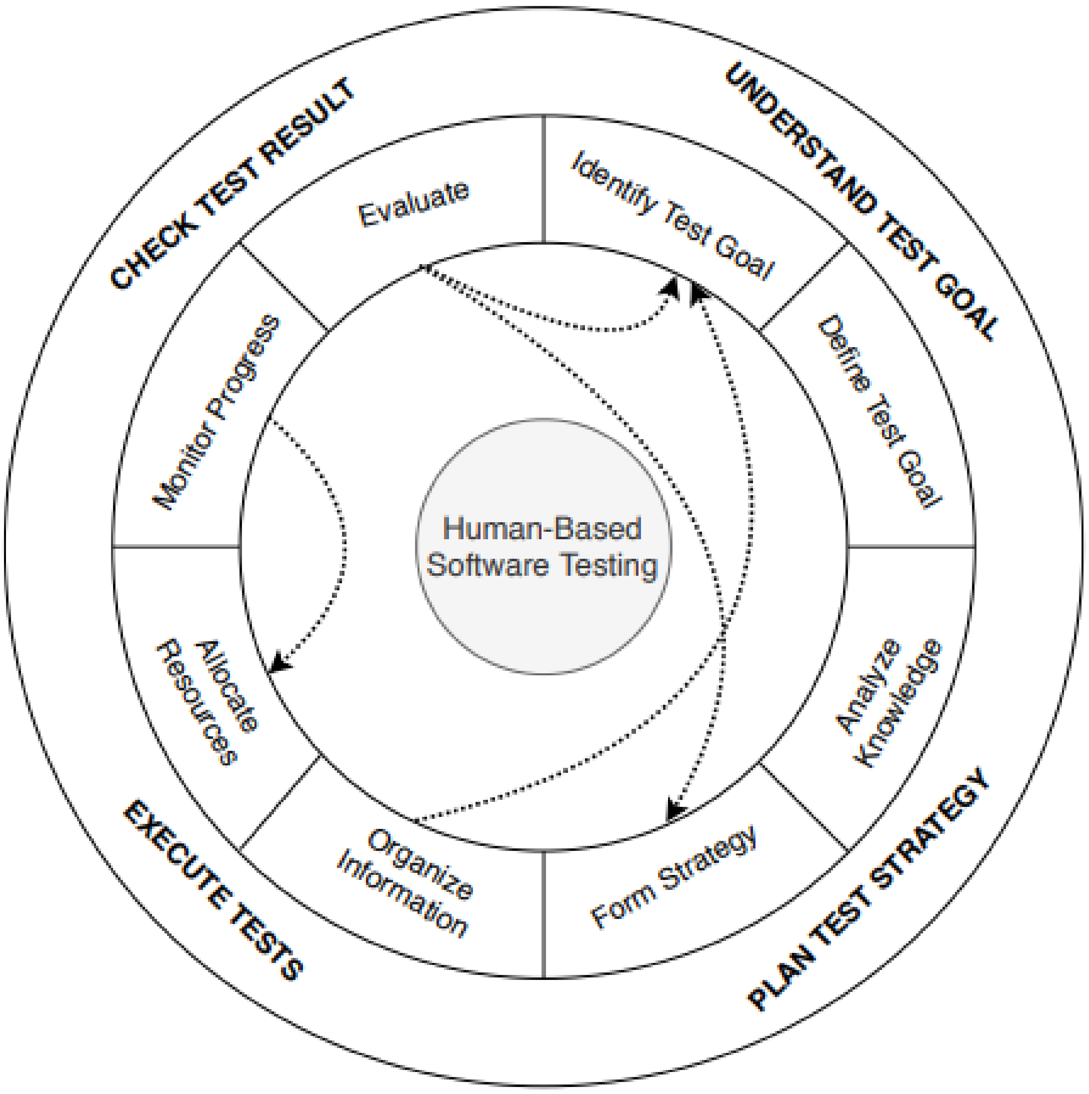


Figure 2.1: Proposed cyclical model by Enoiu et. al. (2020). The cyclical representation of software testing when viewed from a problem-solving perspective.

Enoiu and Feldt followed up this study by proposing the Human-based Automated Test generation framework, creating an all-encompassing architecture that can be used to examine how the human mind writes test cases [26]. The authors mention that identifying cognitive processes just from written tests can be difficult, so they recommend the use of verbal protocol analysis (having participants think aloud) to better identify explanations for behaviors. This technique aligned with our data collection methods (see Section 4.1) and we emphasized it throughout the study. They also discuss how other influences like motivation, creativity, psychological factors, and social factors all affect the test-writing process. By keeping this idea in mind while analyzing our data, we were able to keep a broad perspective while determining the reasons behind student testing decisions and we picked up on cues that we would otherwise have missed.

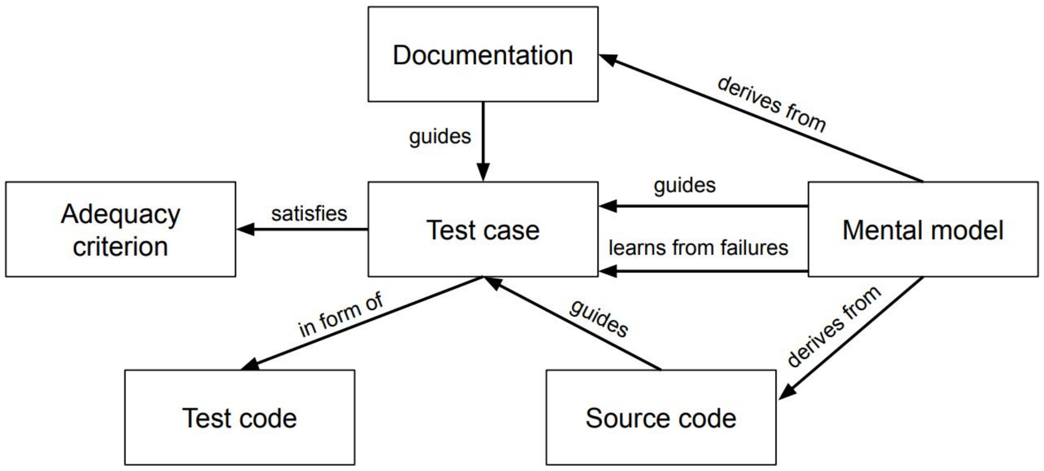


Figure 2.2: Proposed testing framework by Aniche et. al. (2021). The framework of strategies that describe how developers think about testing.

Building upon the framework proposed by Enoiu et. al., Aniche et. al. created their own model of software tester cognition (see Figure 2.2) [8]. By observing developers think aloud as they wrote tests, they identified six main concepts and the relationships between them that the testers would use as they worked. As with the other framework, we considered studying student movements through the model while testing under different TAC, but we used it more as a reference to guide our analysis instead. In particular, we focused on how students built up their mental models of a program differently for each TAC.

While work has been done towards developer cognition while writing and testing code, no research has gone into observing how their thought processes change under different TAC. Additionally, the majority of these studies focus on professionals working in industry, while this thesis centers around introductory computer science students.

Chapter 3

MUTTLE: DATA COLLECTION TOOL

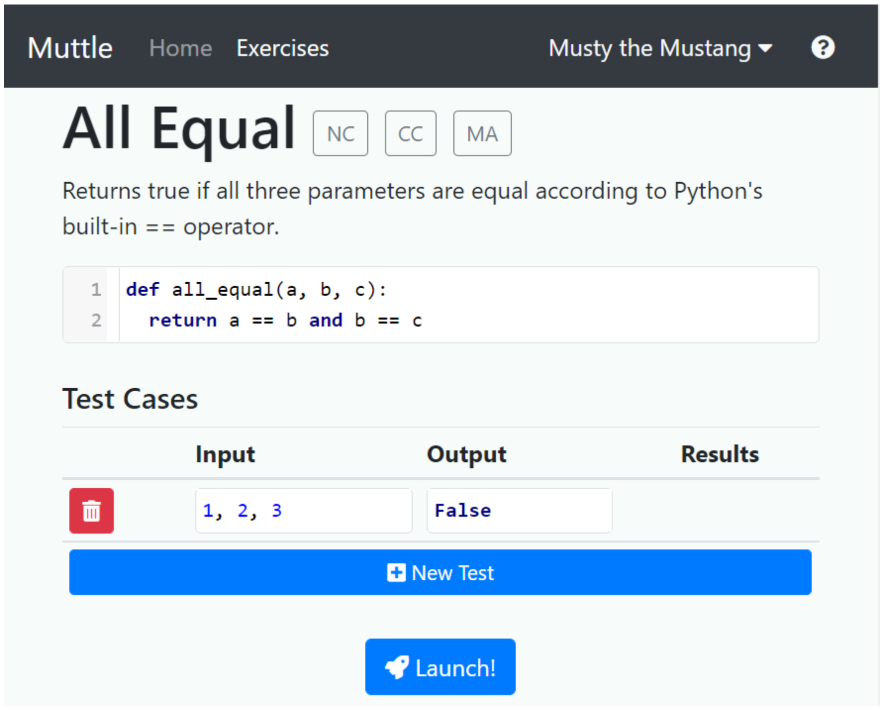
Muttle is a web application that allows users to write unit tests for a given Python exercise, run them, and then instantly receive test adequacy feedback. Muttle's current features and an overview of its architecture are described in this section in order to provide context into the data collection process, but it already existed in a functional state before my use of and contributions to it. More information about my work on Muttle is shown in Section 3.2.

At the core of Muttle are a MySQL Server database, an Express/Node.JS backend, and a React frontend. All of the code was written in TypeScript. While most of Muttle's functionality is built from scratch, some of the notable third-party tools that it uses include CodeMirror [27] for the in-browser code editor, TypeORM [28] for database-backend connections, and MutPy [29] for generating CC and MA feedback.

3.1 Features

The majority of the content that Muttle users interact with are known as "exercises." Exercises are the combination of a title, problem description, and function source code that come together to form a standalone testing activity. Users can freely create their own exercises or write tests for existing ones.

Test cases in Muttle consist of one or more comma-separated input values (corresponding to the exercise's function's parameter(s)) and one expected output value. Native Python expressions are used as input and output, and users can freely add and remove any number of test cases. Figure 3.1 shows all of these features in a sample Muttle exercise. The user can also run their test suite at any time, at which point test adequacy feedback will be generated and displayed.



Returns true if all three parameters are equal according to Python's built-in == operator.

```python
def all_equal(a, b, c):
  return a == b and b == c
```

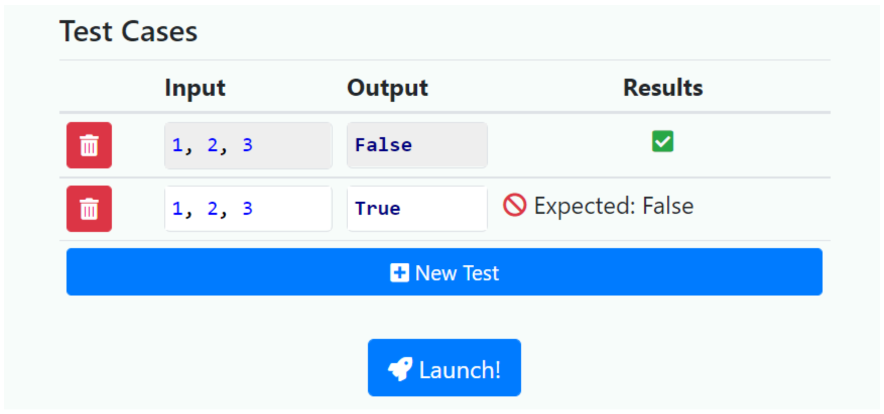
Test Cases

	Input	Output	Results
	1, 2, 3	False	

Figure 3.1: Muttle Testing User Interface. Shows a sample exercise and an input and output for a test case that has not been run yet. The test case deletion and addition buttons, the launch test button, and the three TAC toggle buttons are shown.

Currently, Muttle supports no feedback (NF), CC, and MA as its possible modes of test adequacy feedback. At the lowest level of detail, the NF option simply states if each test case passes or fails when run, as seen in Figure 3.2. Before any additional feedback can be displayed, Muttle expects all tests to first pass, so this indication of test success or failure is always shown.

The next most detailed feedback mechanism is the CC of the test suite, represented by the colored gutter that runs vertically next to the exercise's source code which can be seen in Figure 3.3. After running the test suite, either red,

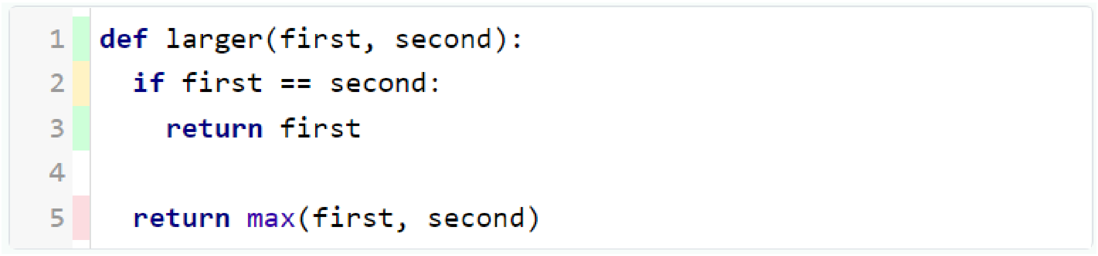


Test Cases

	Input	Output	Results
	`1, 2, 3`	`False`	✓
	`1, 2, 3`	`True`	Expected: False

Figure 3.2: Muttle Pass and Fail Indicators. Results for one successful and one unsuccessful test case are shown.

yellow, or green can appear in the gutter next to each line, indicating that line's coverage status.



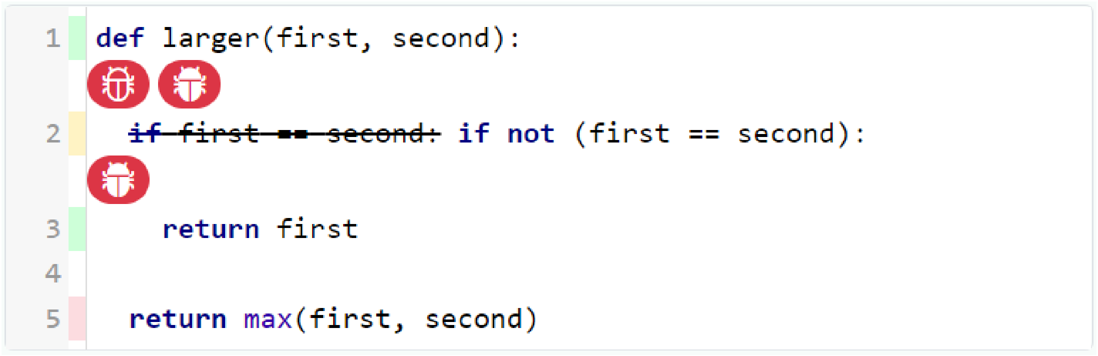
```python
def larger(first, second):
  if first == second:
    return first

  return max(first, second)
```

Figure 3.3: Muttle Code Coverage Gutter. The gutter showing all three levels of coverage is shown next to a sample exercise.

Red indicates no coverage, meaning that a line of code was never run. Yellow indicates partial coverage, meaning that a line of code is a branching statement and that only some of its branches were run. Some examples of partial coverage include the conditional in an *if* statement only being evaluated to *True* but never *False*, or a *for* loop not running to completion. Finally, green indicates full coverage, meaning a non-branching line of code was run, or all branches of a branching line of code were run.

MA is the most comprehensive feedback mechanism that Muttle offers, represented by the bug badges that appear above lines of code. The presence of a bug badge indicates that the line below it contains a surviving mutant, and clicking on it displays the original and mutated lines of code next to each other with the original line being struck through. Figure 3.4 demonstrates Muttle's MA feedback mechanism.



```
def larger(first, second):
    ~~if first == second:~~ if not (first == second):
        return first

    return max(first, second)
```

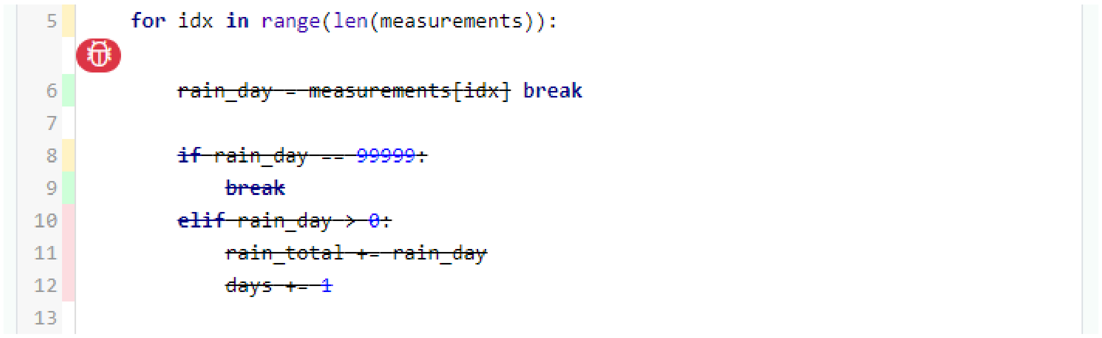
Figure 3.4: Muttle Mutation Analysis Feedback. Red bug badges indicating surviving mutants as well as an example of original and mutated code side-by-side are shown.

A red badge indicates a surviving mutant, a yellow badge indicates a timed-out mutant, and a grey badge indicates an incompetent mutant. As MA subsumes CC [23] by default the colored gutter representing coverage is shown alongside the bug icons, but it can be toggled on and off.

3.2 My Contributions

Before the interviews for this study could be completed, several changes had to be made to Muttle so it was ready for the data collection we had in mind. The majority of its core functionality was completed by Jon Lai and Ayaan Kazerouni before I joined this project, and they have been working on it since January 2021.

My first contribution to this project was adding the ability to display mutations that affect multiple lines of code. At the time, the code for displaying mutations was bugged and assumed that they were all one-line changes, so any multi-line mutations only showed the first line of original code as being struck-through. Some work had already been done towards adding this feature, so I continued what the previous author was doing and got Muttle to properly display all mutations (see Figure 3.5).



```
    for idx in range(len(measurements)):

        ~~rain_day = measurements[idx]~~ break

        ~~if rain_day == 99999:~~
            ~~break~~
        ~~elif rain_day > 0:~~
            ~~rain_total += rain_day~~
            ~~days += 1~~

```

Figure 3.5: Multi-line Mutant. An example of a properly displayed multi-line mutant.

Next, I focused on getting Muttle to a state where it could be used for this study. Again, the main functionality was all in place, but some of the key features were on separate branches or were missing. After merging my multi-line mutation changes into the main Muttle branch, I merged the finished user login and sign-up system that Jon had written into it as well. After resolving the merge conflicts and bugs that arose, I added a set of buttons that toggled which type(s) of TAC were active for the current exercise, which can be seen at the top of Figure 3.1.

3.3 Future Work

The next steps for Muttle should focus on accessibility and bug fixing improvements so that it can be used as a teaching tool. For instance, symbols inside of the CC gutter and in the different MA bug badges would allow colorblind users to differentiate between the different coverage levels and mutant classifications.

Additionally, there are some bugs that we were aware of and could ignore during the study that users unfamiliar with the system would trip over.

After these quality of life changes are rolled out, work to create exercise offerings, or wrappers around exercises, should be done. This feature will, for example, allow teachers to create testing assignments and distribute them to students in a highly-customizable manner. Creating an exercise offering would consist of selecting an exercise and then choosing TAC, and if MA is selected, choosing which mutation operator(s) to generate. Following this change, Muttle can then be deployed to be widely available for in-classroom use. While it was perfect for data collection in this study, its impact can be far wider-reaching if it was adopted by educators as a tool for teaching and practicing good software testing habits.

Chapter 4

METHODOLOGY

For this study, we used an approach based on the grounded theory (GT) method. A series of students were recorded as they thought aloud while writing unit tests for different Python functions, and these interviews were then transcribed and analyzed. This study design was approved by our university's Institutional Review Board (IRB).

GT is a qualitative analysis technique that refers to constructing hypotheses and theories through the collection and analysis of data. It is useful when "little is known about the phenomenon; the aim being to produce or construct an explanatory theory that uncovers a process inherent to the substantive area of inquiry" [30]. The name "grounded theory" comes from the fact that the theories resulting from these types of studies are "grounded" in the data, i.e., based on and arising from the data.

Anselm Strauss and Barney Glaser created one of the core principles of GT while working on their book, *Awareness of Dying* [31], as the traditional scientific method did not fit their topic of study [30]. This principle, constant comparative analysis, refers to the practice of performing data collection and analysis in parallel, with the data collection process changing as new findings appear. The combination of constant comparative analysis and the data analysis technique known as coding (see Section 4.2) are what define GT studies.

4.1 Data Collection

Efforts were made to collect and store data solely on university-provided Microsoft services when possible to maximize participant privacy and data safety. Interview and screen capture were done using a university Zoom account using Zoom's cloud recording feature and only researchers were given access to these recordings. Interview audio was transcribed and analyzed using the GT methodology, and the screen captures were used to fill in any transcription ambiguities (e.g. "this right here"). A third-party tool created by a former student, Azure Zoom Recording Transcription (AZRT) [32], was used to generate transcriptions as Zoom's native transcription feature was lacking. No participant data was ever stored on Azure, it was only used to process the recordings into text transcripts.

4.1.1 Interviews

Interviews were completed in person as Muttle (see Chapter 3) had to be hosted locally on the laptop that participants used. Each interview took 30-45 minutes and began with a short demographic survey (see Table 4.1) followed by a series of testing exercises. The participants were required to sign a consent form (Appendix A) and had the session audio and computer screen recorded.

To advertise this study, emails and presenters were sent out to several sections of the classes that are typically taken after our university's data structures course. These classes were chosen because students at this level have some experience with both reading and testing programs, but are still considered introductory programmers. Students were offered \$25 Amazon gift cards as compensation for their time.

Table 4.1: Summary demographic survey data from study participants.

Student Demographic Data	
Ethnicity	**n = 12**
Asian	4
Hispanic/Latinx	1
Two or More	3
White	4
Gender	
Man	7
Woman	5
Academic Standing	
Freshman	9
Sophomore	2
Junior	1
Major	
Computer Science	6
Computer Engineering	3
Statistics	2
Liberal Arts and Engineering Studies	1

To begin the interview, we asked students to review and sign the aforementioned consent form as an indication of their agreement to participate in the study. The form explains the purpose of the study, potential risks, protections, resources, and relevant contact information. This step was necessary as a part of compliance with IRB protocols.

Following the demographic survey, we introduced Muttle to the student and presented them with a warm-up function to test. The goal of the warm-up phase was to familiarize the student with the Muttle user interface as well as the different types of test adequacy feedback. Next, the student was tasked with thinking aloud as they wrote tests for three or four other functions, each with a different combination of TAC. See Section 4.1.2 for more information on the functions selected for this study and see Section 4.1.3 for more information on the think-aloud testing process.

12 students were selected for this study because, with three Python functions and three TAC that each had to be used once in each interview, six sets of function/feedback pairs exist. This number of participants allowed for each set to be used for two students, reducing any potential bias that the specific pairings of function and feedback would have.

4.1.2 Problem Selection

Functions used in the interview process were chosen based on their type (the primary programming concept used in the code e.g., loops, control statements, sorting, etc.) and the amount of test adequacy feedback generated. Varying the types of programs was done to reduce any cognitive bias that could arise from using only one, and most of the functions were tested in a pilot study to examine the feedback it produced.

Initially, four testing functions plus one warm-up function were used as the problem bank for interviews: SELECTION SORT, TRIANGLE, RAINFALL, CENTERED AVERAGE, and MULTIPLY (warm-up), which can all be seen in Appendix B. The SELECTION SORT problem was chosen to fill in a problem type but it did not provide adequate test adequacy feedback and was removed from the set of functions (see Section 4.1.3 for more detail on why this choice was made).

The second function was the triangle classification problem, referred to as TRIANGLE. This problem was chosen as it is documented in mutation testing literature [33] and has been used in testing and mutation-related studies [34, 35, 36]. Given three side lengths representing a triangle, TRIANGLE returns a different number indicating whether the triangle is invalid (the sum of any two sides is less than or equal to the third side), scalene (valid and all sides are different lengths), isosceles (valid and only two sides are of equal length), or equilateral (valid and all sides are the same length). This problem fills the role of the control structure-centric function, with its heavy use of *if* statements, and it also gave great feedback for both CC and MA in test studies.

RAINFALL was chosen as the third problem in the problem bank. Given a list of numbers representing daily rainfall readings, it calculates the average daily rainfall for rainy days with valid (non-negative) readings for days before a sentinel number (99999) or before the end of the list. This function is a mixture of types, with a *for* loop presenting as its main feature alongside several conditional and control statements. RAINFALL is a classic problem in computer science education and has been used in prior studies [37], including the one done by

Castro and Fisler [10] that partially inspired this thesis. It also gives quality CC and MA feedback.

The fourth function used was CENTERED AVERAGE. Given a list of at least three numbers, CENTERED AVERAGE calculates the average of the list excluding one instance each of the highest and lowest numbers. Similar to RAINFALL, this program is a combination of types, featuring a loop as well as conditionals. It gives decent CC feedback and while several equivalent mutants were generated, the killable mutants provide a sufficient challenge for testers. This problem was initially added as a bonus question to be used if there was still time remaining after the other problems were tested, but it was eventually used to replace SELECTION SORT.

MULTIPLY was the first warm-up problem used in the first set of interviews but it was replaced with LARGER (see Appendix B due to its poor test adequacy feedback and a change in interview procedure (see Section 4.1.3 for more information about the think-aloud testing process). Given two numbers, MULTIPLY returns the result of multiplying them together, and as the code was only one line long, the code is always fully covered under CC.

On the other hand, Large contains an *if* statement, so it gives some CC feedback and also generates simple mutants. Given two numbers, LARGER returns the greater of the two. While it is still short, it generated more feedback than MULTIPLY and thus provided a better introduction to Muttle and to each TAC.

4.1.3 Think-Aloud Testing

The bulk of the data for this study were generated by recording participants thinking aloud while writing unit tests under different TAC. Students were instructed to voice any thoughts they had while reading code and writing tests, and we would ask follow-up questions to extract more information about their thought processes behind different actions.

For the first two interviews, students were given MULTIPLY as a warm-up exercise and were instructed to write a simple test for it. The first student was then presented with TRIANGLE with NF and instructed to test it to their satisfaction. The student was told that the code was fully functional and that they were to test to the best of their abilities, and a hidden 20-minute time limit was set. While this time limit was used for each problem for each participant in the study, it was never reached.

Next, that student was introduced to CC and tasked with testing SELECTION SORT to full CC. After running only a single test 100 percent CC was reached, so to try to gather some data we manually entered a test that resulted in partial coverage. We instructed the student to finish testing the problem to 100 percent CC, but not much data was gathered due to the ease of which the problem could be fully covered.

The student was then familiarized with MA and presented with RAINFALL with the goal of reaching a full mutation score. While there was some initial confusion about MA, after writing some simple tests to generate feedback and asking questions, the student was able to able to figure out how to kill all of the surviving mutants.

Next, the student was presented with CENTERED AVERAGE with NF and again instructed to test it to their satisfaction. After they had finished, we then enabled CC feedback and had the student continue to write tests until 100 percent coverage was reached. Finally, we enabled MA feedback and had the student test until no mutants remained. This stacking of feedback mechanisms was done to gain insight into how the student's cognition changed directly because of a TAC, but this practice was abandoned in later studies due to a lack of problems to use.

The second interview was performed directly after the first, with the same warm-up and first problem and the same TAC used for each problem. This student was supposed to be given MA as the feedback mechanism for SELECTION SORT, but after one test was run only equivalent mutants were generated. To salvage some data from the session, we switched to CC and used the same manual test trick to force a gap in coverage and then had the student test the problem to full coverage. The TAC and procedures used for the third and extra problems were the same ones used for the first student.

After these two sessions were completed, the interview notes and recordings were analyzed to see how we could improve our methods to increase the richness of the data we gathered. The most glaring flaw was the uselessness of SELECTION SORT in terms of test adequacy feedback, so it was replaced with CENTERED AVERAGE. In the interest of time, no extra problem was found as a replacement, so the practice of layering TAC onto each other was abandoned. It was also determined that MULTIPLY was too simple and did not give enough feedback to act as a sufficient warm-up, and was replaced with LARGER.

While our interviewing techniques started rough, they improved over time with practice and the knowledge of when getting a student to vocalize their thoughts would reveal useful information. When students were quiet or not thinking aloud, we asked clarifying questions to encourage them to speak. We also asked more probing questions about students' actions to gather more detailed information, and we started asking the student directly to reflect on how different TAC affected their approach to testing.

After implementing these changes better data was generated, but we soon found that using the same ordering of functions for every student irrespective of feedback mechanism could be biasing their cognitive processes. The third student interviewed was given CC then NF then MA, and when asked about their approach to testing for the NF problem, mentioned that they were thinking in terms of CC even when told that no TAC would be given. The fourth student was given CC then MA then NF, and when similarly questioned for their NF problem, asked if they should think in terms of CC or MA.

As the goal of having problems with NF was to provide a baseline cognitive model of each student's testing process, we decided to present the problems in this order: warm-up with NF, NF problem, warm-up with CC, CC problem, warm-up with MA, and MA problem. With the functions being presented in order of increasing TAC complexity, the effect of each feedback mechanism on the next was lessened. Additionally, providing the same warm-up function for each TAC helped students ease into each new feedback type.

Basing the order of functions tested on the feedback was a great improvement, but there were still more changes that needed to be made to eliminate bias and to improve the quality of our data:

- **Bias:** Students were told that the programs were correct.

  Solution: We no longer mentioned this fact, and we gave non-committal answers if asked about it.

- **Bias:** Students mentioned trying harder to write tests because they were taking part in a study.
  Solution: We told students to treat writing these tests as a lab exercise or homework assignment.

- **Bias:** Students turned to the interviewer or the code less for RAINFALL because of its longer description.
  Solution: More detail and clarifications were added to the descriptions for TRIANGLE and CENTERED AVERAGE.

- **Problem:** We struggled to find a beginner-friendly way of explaining MA to students.
  Solution: We came up with the phrase, "if this bug existed in your code, none of your tests would have caught it and failed."

- **Data:** Students were asked about how and when they learned how to test in order to form a background about their NF testing habits.

By the end of this study, the data collection process was fairly streamlined and we were able to extract much more information from students. However, it should be noted that, due to the fact that SELECTION SORT was only given with CC feedback and eventually replaced, CENTERED AVERAGE only had three students testing under MA and three testing under CC, and RAINFALL had five students testing under MA and three testing under CC. Each problem should have had four students testing under each TAC.

4.2 Data Analysis

For the first six interviews, initial coding was used to break participant interviews down into individual, coded incidents. Initial coding is the most concrete type of coding, where actions and behaviors, as well as their underlying psychological processes, are identified and labeled as a snippet-code pair. For example, one student mentioned that they were writing a very simple-to-understand test to start with, and this snippet was tagged with the code, “writing a ‘basic’, easy-to-reason-about test.” Another student mentioned that, due to a previous class, they think in terms of code coverage when writing tests on their own. This incident was tagged with the code, “prior experience with CC leads students to think in terms of CC on their own.”

Two researchers separately coded the first half of the interviews and then met to combine these initial codes into a codebook. Any matching or closely related codes created by both of the researchers were discussed and added to the codebook, while codes found by only one person required further examination. The goal of this divided approach was to ensure that our findings were truly grounded in the data and not haphazard guesses, as we believed that if two people independently identified a cognitive pattern then it was likely to exist. However, if a researcher strongly believed in a code that the other did not identify, they could present the behaviors and quotes that they thought were indicative of it to try to convince them of its validity. After the codebook was formed, it was used as a guide to gather more snippets from the remaining six interviews.

Our initial codes were then further analyzed in a process known as intermediate coding. Intermediate coding typically follows initial coding, and the goal of this step is to take the very concrete initial codes and abstract them into higher-level categories. Connections between codes were examined and then a code was either abstracted into an entire category, or a new concept was created to act as one. For example, the initial codes from the example above could both be grouped under the category of "writing tests to get started."

Only one researcher performed the intermediate coding step, but both researchers discussed the proposed categories at length before any final conclusions were made. After grouping the initial codes by factors such as TAC, stage of the test writing process, concepts in cognitive models (Figure 2.1 and Figure 2.2 in Section 2.2), and level of thought (the study done by Castro and Fisler [10] detailed in Section 2.2), the resulting categories and their validity were reviewed. Our discoveries from our analysis through both coding steps are discussed in the next chapter (see Chapter 5).

Chapter 5

RESULTS

The first section of this chapter describes our findings related to how students understand programs when first encountering them with the intention of testing them. The next three sections are based on the TAC that were used in this study (NF, CC, and MA), and for each their associated patterns of cognition with corresponding actions and interview snippets are presented. Finally, the last section details miscellaneous findings that are not tied to a specific TAC.

The results of this study are presented as compelling phenomena that can be further investigated to better understand student thought processes while testing.

5.1 Problem Comprehension

This section was derived from the categories and codes related to how students formed and developed their mental model of programs, with "mental model" referring to the mental model portion of the framework created by Aniche et. al. (see Figure 2.2 and Section 2.2). Our findings here relate to both the initial formation and updating of said models.

When presented with a new problem, students would start by either reading the problem statement or by reading the code line-by-line. Aniche et. al. "[observed] developers using the documentation as a way to build an initial *mental model* of the program under test, which [was] then leveraged as the main source of inspiration for testing during the rest of task" [8], and all students exhibited this behavior at least once. Additionally, apart from one student (Stu5), participants would often follow reading the description by reading at least part of the code before testing.

The students who started by reading the code (Stu2, Stu3, Stu6, Stu10, Stu11) would completely ignore the problem description and would only refer to it when further clarification was required. However, students could have been influenced by the length of the problem statements, as RAINFALL had the longest description by far. Even when adding more detail to the other problems to match, the additional requirements and relative complexity of RAINFALL meant its description was still the longest.

When it came to updating their mental models when struggling to understand a program, students employed a variety of strategies. The most common practice (Stu1, Stu2, Stu3, Stu5, Stu6, Stu8, Stu9, Stu11, Stu12) was reading the problem description, but some students (Stu1, Stu3, Stu4, Stu5, Stu6, Stu10) sometimes opted to read the code directly. Some participants (Stu1, Stu2, Stu9, Stu11) used their knowledge of variable roles (e.g. index holders or temporary minimum and maximum variables) [38] to fill in the gaps in their understanding, and one student (Stu3) actually wrote an experimental test to see what the expected behavior of the program was, hearkening back to work that uses test cases as meta-cognitive scaffolding to ensure understanding of problem prompts [39, 40].

5.2 Testing With No Feedback

Our findings in this section have to do with how students wrote tests when they were told to test to their own understanding of the specification, given no TAC. For the most part, students exhibited the same overall pattern of behavior, but there were some varying strategies (some of which wildly deviated from the norm) and even "internal TAC" that influenced testing methods.

5.2.1 Student Testing Strategies

We found that every student's (Stu1-Stu12) testing efforts were at least partially based on their own intuitions about edge cases, and when asked about where this habit formed, they pointed out the prerequisite data structures class and its automatic grading system. Grading for lab and project assignments in this class is based on a percentage of tests passed from the instructor's test suite, which is full of cases designed to see if programs can handle inputs with integers/decimals, negative/positive numbers, zeroes, list of varying lengths, etc., which mirrored the boundary values that students were trying to test.

While students were most heavily influenced by their own testing intuition, some used a mentally-approximated form of CC even when not explicitly told to. One participant (Stu4) mentioned that they usually trace their code to see if it is fully covered by their test suite, two others (Stu11, Stu12) used coverage to guide only a part of their testing efforts, and another (Stu3) was biased to thinking about coverage because they tested under CC for the problem prior to their NF one. Other students in the first several interviews whose NF problem followed their CC or MA problem asked if they should think in terms of these TAC but were told to use their own methods instead.

Some participants tested the code directly while reading it, either going over the whole function or just part of it, formulating tests based on edge cases or typical inputs as they went. Of the students who tested the code line-by-line in its entirety, two (Stu2, Stu3) jumped straight into it without reading the function description at all, while another two (Stu1, Stu6) read the description first. Four students (Stu8, Stu9, Stu10, Stu11) partially read and tested their code after reading the description, with one of them (Stu11) stating that they wanted to first trigger every *return* statement.

Our sample size of participants is small but there were some interesting quantitative trends that we observed. On the whole, student approaches to testing scored well in terms of CC and MA (averages of 98 percent and 92 percent, respectively) but had some deficiencies in catching conditional statement-related bugs. Half of the surviving mutants were Relational Operator Replacement bugs, which replaced operators like $>$ with $>=$. Additionally, Logical Connector Replacement (LCR), Conditional Operator Insertion (COI), and One Iteration Loop (OIL) mutants survived in student test suites written under NF but not in student test suites written under CC.

5.2.2 Internal TAC

We also identified several "internal" or self-imposed TAC that students would use while writing their test suites. For example, students in earlier interviews (Stu2, Stu3) mentioned that they were putting in more effort when testing since they were participating in a study, and even when later students were explicitly told to treat testing as an assignment for class, there is still a possibility that they were unconsciously being more thorough. Some students (Stu1, Stu3, Stu5, Stu11) also mentioned wanting to "put the code through to most paces" in some cases, and this sentiment took on different meanings in different contexts. Sometimes it was meant in terms of coverage (reaching the most lines) and other times it was meant in terms of complexity (trying out a wide variety of inputs).

Some students exhibited the tendency to write tests to target a specific feature in the code or description that caught their attention. The *if days == 0:* statement in RAINFALL and the final standalone *return 3* in TRIANGLE are examples of code that stood out enough that students (Stu4, Stu9, Stu10, Stu11, Stu12) were drawn to writing a test for them, with some (Stu1, Stu2, Stu6, Stu10) going a step farther and writing multiple tests in order to reach full CC. The sentinel number (99999) requirement in RAINFALL is a prime example of a part of a description that students (Stu1, Stu3, Stu5, Stu7, Stu8, Stu11, Stu12) were drawn to testing.

Even after the given TAC was met, some students felt the need to add more tests to cover edge cases. For both CC (Stu1, Stu2, Stu7, Stu9) and MA (Stu1, Stu7), participants added more test cases to check for decimal, negative, zero, etc. inputs, but it is unclear whether this was done due to the external pressure of participating in a study or due to a lack of trust in TAC.

5.3 Testing Under CC

On the whole, of the students who had any significant CC feedback to respond to (Stu1, Stu2, Stu4, Stu7, Stu9, Stu12), only one (Stu4) relied on any task-level thinking. The rest (Stu1, Stu2, Stu7, Stu9, Stu12) focused on mentally tracing their test cases through the code. They would use the coverage gutter to direct them to uncovered or partially covered statements and then run these lines and any surrounding relevant ones (i.e., in the same block of code or directly connecting to the line in question) in their head in order to see what coverage gaps existed in their test suite. This phenomenon of code tracing occurring in isolated blocks was not unique to the students testing under CC (Stu2, Stu3, Stu7, Stu9, Stu12), as it also surfaced when students were comprehending MA feedback (see Section 5.4).

Besides just tracing code to understand why a coverage gap existed, students (Stu1, Stu2, Stu3, Stu4, Stu6) employed a strategy similar to the one mentioned above in Section 5.1 of using variable roles. For example in SELECTION SORT, one student (Stu1) had partial coverage for *if input_list[j] < input_list[min_idx]:* when they said that they were using their knowledge of the function of current and minimum index variables to deduce that they needed to add a test where a new minimum index was never found.

Some participants (Stu2, Stu3, Stu9) chose to ignore CC feedback in favor of testing edge cases. Reasons included being unsure of how to reach full coverage for a statement and saying they would come back to it once they had written more tests, wanting to check boundary values, and just an overall desire to test only using the problem description and not the code (Stu7, Stu12).

5.4 Testing Under MA

As with CC, only one student (Stu1) demonstrated any significant problem-space thinking when understanding and responding to MA feedback. Methods for comprehending bugs mainly focused on understanding the differences between the original and mutated programs (Stu3, Stu4, Stu6, Stu7, Stu8, Stu11, Stu12) by tracing the mutant code (Stu2, Stu5, Stu8, Stu11, Stu12), which occurred in isolated blocks (Stu2, Stu11, Stu12).

We noticed that, for each mutant, it appeared that students would develop a parallel mental model of the program that included the mutated line(s), which they would compare to their mental model of the unmutated program. This feat requires a high cognitive load, which we believe is why some students (Stu2, Stu3, Stu4, Stu6, Stu8) had difficulty understanding Muttle's MA feedback. While unfamiliarity with mutation testing methods is partly to blame, even after students understood the concept and were able to successfully kill a different mutant, they were still unable to kill others.

Even when students did not completely understand the effects of a mutant, they would sometimes have a vague idea of how to kill it that would often succeed. This phenomenon was common for mutants that changed the outcome of conditional statements and operators, with students mentioning that they had the feeling that testing all branches of a mutated conditional would kill a bug (Stu2, Stu3, Stu4, Stu6, Stu7, Stu8, Stu9, Stu10, Stu11). Another example is when the *if rain_day > 0:* statement in RAINFALL is mutated to *if rain_day > 1:*, one student (Stu3) could not verbalize what this bug actually changed in terms of the problem and its requirements but correctly deduced that writing a test case with a list that included a 1 would kill it.

Some participants used the presence of mutations to guide their intuition-based testing efforts (Stu5, Stu7) almost like a form of code coverage, and others took this dismissal of MA to the next level and just ignored it completely until their testing methods did not kill any more mutants (Stu9).

5.5 Other Testing Patterns

Apart from the cognitive processes we identified that were specific to each TAC, we also noticed some more general trends that were not exclusive to any single one. For example, every student wrote at least one basic, easy-to-reason-about test. While some of these tests were written to fulfill TAC goals or to better understand the program, often these test cases were "happy path" tests that only simulated typical program inputs. It was also common for students (Stu1, Stu2, Stu3, Stu5, Stu6, Stu7, Stu8, Stu9, Stu10, Stu11) to copy and edit an existing test instead of coming up with an entirely new one.

Since Muttle does not generate any test adequacy feedback until all of the tests in a suite pass (i.e. for every test case the actual output matches the expected output), some students (Stu1, Stu2, Stu6, Stu7, Stu9) would start by generating test adequacy feedback by writing smaller, simpler tests rather than a full set of tests based on their testing intuition. After receiving initial feedback, students would proceed to focus on achieving a complete test adequacy score and would not rely on their own test-writing habits until afterward (Stu1, Stu2, Stu7, Stu9).

While some students (Stu2, Stu6) mentioned that they preferred to test based on the problem description rather than source code, others actually put this habit into practice (Stu5, Stu7, Stu12) when writing their test suites under NF, with one student (Stu5) not referring to the code at all. That same student ignored the CC gutter and just wrote tests to catch edge cases, using the partial coverage indicators to point out *if* statements to think of more edge cases for. This was the student who used MA feedback as a form of code coverage (see Section 5.4).

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

To find possible improvements to the way software testing is taught, we recorded and interviewed introductory computer science students as they wrote unit test suites under different test adequacy criteria. Through our analysis, we uncovered some of the different behaviors and cognitive processes students took when testing on their own with no test adequacy feedback, with a CC gutter as feedback, and mutant bug badges as MA feedback.

6.2 Future Work

The results portion of this paper (see Section 5) explains our findings from our analyses, but we did not dive deeply into any of the phenomena we discovered. This section will highlight some of these ideas in the form of research questions that can be explored in future studies.

Which mutation operators commonly survive student test suites when they are written with no test adequacy feedback? Which are commonly killed? These two questions have to do with the findings of our quantitative analysis on student test cases written under NF, where we found that half of the surviving mutants for student test suites written under NF were ROR bugs. Understanding which operators slip through student test cases would reduce the cost of using MA in a classroom setting if the ones that are commonly killed are excluded from generation. Additionally, knowing which operators survive would help educators direct students' testing efforts in order to better eliminate these kinds of faults. Our sample size of 12 students is too small for us to make any definitive conclusions, but a quantitative study done with more data (once Muttle is fully deployed, with existing student test cases from a class, etc.) can be carried out to discover statistically valid findings.

How do students form software testing habits? Are these habits helpful? Are they harmful? Software testing is not widely covered in computer science curricula, with students forming good and bad habits on their own in different ways. Participants in this study mostly tried to test edge cases that focused on data types (integers/decimals, negative/positive/zero numbers, etc.) rather than any problem-specific factors, and some of these tests may have been redundant and may not have added any value to the test suite. They pointed out a prerequisite data structures course and its automatic lab and project grader as the reason for this thought process, and findings about how students develop their personal testing strategies can show us how to stop bad habits before they form. And if these strategies are shown to be helpful, they can be encouraged.

Can teaching using code comprehension for testing strategies also improve testing practices? This question arises from the fact that students used variable roles as both a way to better understand a problem for testing and a way to fill in gaps in CC. Investigating this habit as well as finding other such parallels can help us find new ways to teach software testing.

Do patterns of code-level and task-level thinking correlate to software testing success? While not directly related to any of our findings, the fact that students spent much more time at the code-level, as referred to by Castro and Fisler [10], when responding to CC feedback inspired this question. As with their study, the uses of these two levels can be examined and correlated with how well students write test suites, but another interesting approach would be to use the actions that Aniche et. al. [8] identified instead. They identified several strategies revolving around the uses of documentation, source code, and testers' mental models, and these three factors could be used in place of code-level and task-level thinking.

How do students form different mental models of mutated programs when testing under mutation analysis? We observed that student efforts when testing under MA feedback centered around comparing the original program to the mutated copy. Students seemed to form parallel mental models of each version, which led to a high cognitive load and difficulty in understanding a mutant. A more detailed understanding of how students think about this feedback mechanism will improve how we help students reason about possible defects in code while they test it.

Do certain mutation operators require a higher cognitive load to process than others? A better understanding of the mental processing power required to comprehend different mutation operators will help us find an order in which to present them to students that would provide a gradual introduction to MA. It could also reveal that certain ones are too hard to understand, making them more detrimental than useful as teaching tools.

How do some students kill mutants without fully understanding them? In several instances, a student was not able to fully understand the effects of a mutant on a program but was able to kill it with a test. When asked about their thought processes, students mentioned that they had a "feeling" that a certain input would kill a mutant (fully covering all the branches of a conditional or using an input containing a number that was mutated). Uncovering how and why this intuition is formed will also help understand how to better guide student thought processes revolving around potential faults while they test software.

Does the presentation of mutants matter when it comes to understanding them? Muttle presents mutants as inline bug badges that, when clicked, show the original struck-through line(s) of code next to the mutated line(s). It is not known if this is the best, or even a useful representation of a mutant, so experimentation with different presentations would be very insightful. The full program with the mutant side-by-side with a diff and replacing the line(s) in question when the badge is clicked are two possibilities.

Does the way problem specifications are laid out affect software testing strategies? RAINFALL had a much longer problem description than any of the other functions, which may have drawn students to use it more. While it is unclear if this is true based on the findings of this study, investigating how much the format and content of program specifications affect how students use the documentation versus their mental model or the source code to write test cases could reveal interesting patterns.

BIBLIOGRAPHY

[1] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 156–173, 1975.

[2] T. Astigarraga, E. M. Dow, C. Lara, R. Prewitt, and M. R. Ward, "The emerging role of software testing in curricula," in *2010 IEEE Transforming Engineering Education: Creating Interdisciplinary Skills for Complex Global Environments*, pp. 1–26, 2010.

[3] V. Garousi and A. Mathur, "Current state of the software testing education in north american academia and some recommendations for the new educators," in *2010 23rd IEEE Conference on Software Engineering Education and Training*, pp. 89–96, 2010.

[4] D. Hörnmark and P. Hamfelt, "Shortcomings of developers early in their careers in regards to software testing," bachelor thesis, Blekinge Institute of Technology, Valhallavägen 1, 371 41 Karlskrona, Sweden, August 2020.

[5] A. Radermacher and G. Walia, "Gaps between industry expectations and the abilities of graduates," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, (New York, NY, USA), p. 525–530, Association for Computing Machinery, 2013.

[6] S. Valstar, "Closing the academia-industry gap in undergraduate cs," in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, (New York, NY, USA), p. 357–358, Association for Computing Machinery, 2019.

[7] V. Garousi, A. Rainer, P. Lauvås, and A. Arcuri, "Software-testing education: A systematic literature mapping," *Journal of Systems and Software*, vol. 165, p. 110570, 2020.

[8] M. Aniche, C. Treude, and A. Zaidman, "How developers engineer test cases: An observational study," *IEEE Transactions on Software Engineering*, 2021.

[9] E. Enoiu, G. Tukseferi, and R. Feldt, "Towards a model of testers' cognitive processes: Software testing as a problem solving approach," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 272–279, 2020.

[10] F. E. V. Castro and K. Fisler, "Qualitative analyses of movements between task-level and code-level thinking of novice programmers," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, (New York, NY, USA), p. 487–493, Association for Computing Machinery, February 2020.

[11] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing.* Hoboken and N.J: John Wiley & Sons, 3rd ed ed., 2012.

[12] K. Aaltonen, P. Ihantola, and O. Seppälä, "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*, OOPSLA '10, (New York, NY, USA), p. 153–160, Association for Computing Machinery, 2010.

[13] S. H. Edwards and Z. 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, (New York, NY, USA), p. 171–176, Association for Computing Machinery, 2014.

[14] D. Tengeri, L. Vidács, A. Beszédes, J. Jász, G. Balogh, B. Vancsics, and T. Gyimóthy, “Relating code coverage, mutation score and test suite reducibility to defect density,” in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 174–179, 2016.

[15] H. Hemmati, “How effective are code coverage criteria?,” in *2015 IEEE International Conference on Software Quality, Reliability and Security*, pp. 151–156, 2015.

[16] J. W. Hollén and P. S. Zacarias, *Exploring Code Coverage in Software Testing and its Correlation with Software Quality; A Systematic Literature Review*. Bachelor’s thesis, University of Gothenburg, 405 30 Gothenburg, Sweden, August 2013.

[17] P. S. Kochhar, F. Thung, and D. Lo, “Code coverage and test suite effectiveness: Empirical study with real bugs in large systems,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 560–564, 2015.

[18] K. King and A. J. Offutt, “A fortran language system for mutation-based software testing,” *Journal of Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[19] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "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*, FSE 2014, (New York, NY, USA), p. 654–665, Association for Computing Machinery, 2014.

[20] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Does mutation testing improve testing practices?," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 910–921, 2021.

[21] M. H. Goldwasser, "A gimmick to integrate software testing throughout the curriculum," *SIGCSE Bull.*, vol. 34, p. 271–275, feb 2002.

[22] S. H. Edwards and Z. Shams, "Comparing test quality measures for assessing student-written tests," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, (New York, NY, USA), p. 354–363, Association for Computing Machinery, 2014.

[23] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," 1996.

[24] A. M. Kazerouni, *Measuring the Software Development Process to Enable Formative Feedback*. PhD thesis, Virginia Tech, March 2020.

[25] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pp. 189–200, 2014.

[26] E. Enoiu and R. Feldt, “Towards human-like automated test generation: Perspectives from cognition and problem solving,” in *2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 123–124, 2021.

[27] M. Haverbeke, “Codemirror.” https://codemirror.net/, 2018.

[28] Yakdu, “Typeorm.” https://typeorm.io/, 2022.

[29] A. Derezińska and K. Hałas, “Analysis of mutation operators for the python language,” in *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, 2014, Brunów, Poland* (W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, and J. Kacprzyk, eds.), (Cham), pp. 155–164, Springer International Publishing, 2014.

[30] Y. Chun Tie, M. Birks, and K. Francis, “Grounded theory research: A design framework for novice researchers,” *SAGE Open Medicine*, vol. 7, p. 2050312118822927, 2019. PMID: 30637106.

[31] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine, 1967.

[32] A. Doebling, “Azure zoom recording transcription.” https://github.com/AugieDoebling/azure_zoom_recording_transcription, 2021.

[33] P. Ammann and J. Offutt, *Introduction to Software Testing.* Cambridge University Press, 2008.

[34] A. Arcuri, “Full theoretical runtime analysis of alternating variable method on the triangle classification problem,” in *2009 1st International Symposium on Search Based Software Engineering*, pp. 113–121, 2009.

[35] A. Arcuri, P. K. Lehre, and X. Yao, "Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem," in *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pp. 161–169, 2008.

[36] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris, "Analysing and comparing the effectiveness of mutation testing tools: A manual study," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 147–156, 2016.

[37] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year cs students," in *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '01, (New York, NY, USA), p. 125–180, Association for Computing Machinery, 2001.

[38] J. Sajaniemi, "An empirical analysis of roles of variables in novice-level procedural programs," in *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pp. 37–39, 2002.

[39] J. Prather, R. Pettit, B. A. Becker, P. Denny, D. Loksa, A. Peters, Z. Albrecht, and K. Masci, "First things first: Providing metacognitive scaffolding for interpreting problem prompts," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, (New York, NY, USA), p. 531–537, Association for Computing Machinery, 2019.

[40] J. Wrenn and S. Krishnamurthi, “Executable examples for programming problem comprehension,” in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER ’19, (New York, NY, USA), p. 131–139, Association for Computing Machinery, 2019.

Appendix A

INTERVIEW CONSENT FORM

INFORMED CONSENT TO PARTICIPATE IN A RESEARCH PROJECT:

"Software Testing Interview"

INTRODUCTION

This form asks for your agreement to participate in a research project about the cognitive processes underlying software testing. Your participation involves taking part in a software testing interview, allowing the use of your answers for research and analysis. It is expected that your participation will take approximately 1-1.5 hours. There are some minimal risks anticipated with your participation. You may personally benefit from this study and others may benefit from your participation. If you are interested in participating, please review the following information. Upon completion of your interview, you will be given a $25 Amazon gift card as compensation for your time.

PURPOSE OF THE STUDY AND PROPOSED BENEFITS

- The purpose of the study is to explore the cognitive processes employed while software developers engineer software test suites.
- Potential benefits associated with the study include a better understanding of how developers compose software tests and improved software testing education based on these insights.

YOUR PARTICIPATION

- If you agree to participate, you will be asked to take part in a short survey about your demographic background.
- Following the survey, you will be asked a set of 3–4 software testing problems. This interview will be conducted in person, but audio and the computer screen will be recorded on Zoom.
- Your participation will take approximately 1-1.5 hours.

PROTECTIONS AND POTENTIAL RISKS

- Please be aware that you are not required to participate in this research, refusal to participate will not involve any penalty or loss of benefits to which you are otherwise entitled, and you may discontinue your participation at any time. You may choose to stop participating in this interview at any point during the interview.
- There is a minimal risk to your reputation or status should your data be disclosed along with your identity. There also is a minimal possibility of emotional distress should any of the questions trigger unpleasant thoughts or feelings.
- Your responses will never be shared with outside researchers to protect your privacy. However, your responses can only be protected to the extent allowed by Microsoft Forms which is not a secure survey platform. Data will be stored in Cal Poly OneDrive storage and deleted after a period of 3 years or at your request.
- Interview recordings will be stored on the Zoom cloud, accessible only to the researchers. Zoom's transcription service will be used, so no additional party will gain access to the data. Data will be deleted after a period of 3 years or at the participant's request.

RESOURCES AND CONTACT INFORMATION

- If you should experience any negative outcomes from this research, please be aware that you may contact C*ampus Psychological Services* at **805.756.2511**, for assistance.
- This research is being conducted by Ayaan M. Kazerouni, PhD, and Bruno C. da Silva, PhD (both Assistant Professors) and Austin Shin (MS student) in the Department of Computer

Science and Software Engineering at Cal Poly, San Luis Obispo. If you have questions regarding this study or would like to be informed of the results when the study is completed, please contact the researcher(s) at *Ayaan M. Kazerouni* at ayaank@calpoly.edu or *Bruno C. da Silva* at bcdasilv@calpoly.edu.

- If you have any concerns about the conduct of the research project or your rights as a research participant, you may contact Dr. Michael Black, Chair of the Cal Poly Institutional Review Board, at (805) 756-2894, mblack@calpoly.edu, or Ms. Trish Brock, Director of Research Compliance, at (805) 756-1450 or pbrock@calpoly.edu.

AGREEMENT TO PARTICIPATE

If you are 18 years of age or older and agree to voluntarily participate in this research project as described, please indicate your agreement by signing below. Please retain a copy of this form for your reference. Thank you for your participation in this research.

Participant name:	Participant signature:
________________________	________________________

Appendix B

INTERVIEW CODING PROBLEMS

B.A Multiply

A function that multiplies two provided numbers.

```
def multiply(a, b):
    return a * b
```

B.B Larger

Given two numbers, return the larger of the two. Otherwise return the first number.

```
def larger(first, second):
  if first == second:
    return first

  return max(first, second)
```

B.C Triangle

Given 3 numbers representing side lengths, determine whether the sides form a valid triangle, and if so, what kind of triangle it forms. Return 0 if they form an invalid triangle (the sum of any two sides is less than or equal to the third side), 1 if they form an equilateral triangle (all sides are the same length), 2 if they form an isosceles triangle (two sides are the same length), and 3 if they form a scalene triangle (all sides are different lengths).

```python
def is_triangle(side1, side2, side3):
    if side1 >= side2 + side3 or \
       side2 >= side1 + side3 or \
       side3 >= side1 + side2:

        return 0 # this is not a valid triangle

    if side1 == side2 and side2 == side3:
        return 1

    if side1 == side2 or side1 == side3 or side2 == side3:
        return 2

    return 3
```

B.D Selection Sort

Sort the given list of numbers using the Selection Sort algorithm.

```
def selection_sort(input_list):
    for i in range(len(input_list) - 1):

        min_idx = i
        for j in range(min_idx, len(input_list)):
            if input_list[j] < input_list[min_idx]:
                min_idx = j

        temp = input_list[i]
        input_list[i] = input_list[min_idx]
        input_list[min_idx] = temp

    return input_list
```

B.E Rainfall

Let's imagine that you have a list that contains amounts of rainfall for each day, collected by a meteorologist. Her rain gathering equipment occasionally makes a mistake and reports a negative amount for that day. We have to ignore those. We need to write a program to (a) calculate the total rainfall by adding up all the positive numbers (and only the positive numbers) and (b) return the average rainfall at the end. Additionally, there is a "sentinel" number of 99999—when this number is encountered, stop counting and return the average so far.

```
def rainfall(measurements):
    rain_total = 0
    days = 0
    for idx in range(len(measurements)):
        rain_day = measurements[idx]

        if rain_day == 99999:
            break
        elif rain_day > 0:
            rain_total += rain_day
            days += 1

    if days == 0:
        return 0

    return rain_total / days
```

B.F Centered Average

Return the average of the given list without the highest and lowest values. You may assume there are at least three items in the list and that every item in the list is a number. If there are multiple highest or lowest numbers, only exclude one instance of each.

```
def centered_average(nums):
    min_idx = 0
    max_idx = 0

    for idx in range(len(nums)):
```

```
    if nums[idx] <= nums[min_idx]:
        min_idx = idx
    elif nums[idx] >= nums[max_idx]:
        max_idx = idx

nums[min_idx] = 0
nums[max_idx] = 0

sum = 0
for num in nums:
    sum += num

return sum / (len(nums) - 2)
```