A Model of How Students Engineer Test Cases With Feedback

AUSTIN M. SHIN*, Ridgeline, Inc., USA
AYAAN M. KAZEROUNI, California Polytechnic State University, USA

Background and Context. Students' programming projects are often assessed on the basis of their tests as well as their implementations, most commonly using test adequacy criteria like branch coverage, or, in some cases, mutation analysis. As a result, students are implicitly encouraged to use these tools during their development process (i.e., so they have awareness of the strength of their own test suites).
Objectives. Little is known about how students choose test cases for their software while being guided by these feedback mechanisms. We aim to explore the interaction between students and commonly used testing feedback mechanisms (in this case, branch coverage and mutation-based feedback).
Method. We use grounded theory to explore this interaction. We conducted 12 think-aloud interviews with students as they were asked to complete a series of software testing tasks, each of which involved a different feedback mechanism. Interviews were recorded and transcripts were analyzed, and we present the overarching themes that emerged from our analysis.
Findings. Our findings are organized into a process model describing how students completed software testing tasks while being guided by a test adequacy criterion. Program comprehension strategies were commonly employed to reason about feedback and devise test cases. Mutation-based feedback tended to be cognitively overwhelming for students, and they resorted to weaker heuristics in order to address this feedback.
Implications. In the presence of testing feedback, students did not appear to consider *problem coverage* as a testing goal so much as *program coverage*. While test adequacy criteria can be useful for *assessment* of software tests, we must consider whether they represent good goals for testing, and if our current methods of practice and assessment are encouraging poor testing habits.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Social and professional topics → Software engineering education**.

Additional Key Words and Phrases: software engineering education, software testing, mutation analysis, branch coverage

ACM Reference Format:
Austin M. Shin and Ayaan M. Kazerouni. 2023. A Model of How Students Engineer Test Cases With Feedback. *ACM Trans. Comput. Educ.* 1, 1, Article 1 (January 2023), 30 pages. https://doi.org/10.1145/3628604

1 INTRODUCTION

Software testing is a core competency in undergraduate computer science and software engineering programs (e.g., [4, 23, 38]). CS educators commonly assess the quality of student-written software tests in addition to solution correctness (e.g., [24, 61, 67]). As such, as students implement solutions

*Work performed while at California Polytechnic State University.

Authors' addresses: Austin M. Shin, amshin1775@gmail.com, Ridgeline, Inc., 1 Grand Ave., San Luis Obispo, California, USA, 93405; Ayaan M. Kazerouni, ayaank@calpoly.edu, California Polytechnic State University, 1 Grand Ave., San Luis Obispo, California, USA, 93405.

1946-6226/2023/1-ART1 $15.00
https://doi.org/10.1145/3628604

and write tests, it is usually desirable for them to engage in a related self-checking behavior—using a *test adequacy criterion* to frequently assess the quality of their own tests.

A test adequacy criterion describes the conditions that need to be met for a set of software tests to be considered "adequate" [31]. The focus of such a criterion is the *tests*, not the software under test. For example, *branch coverage* and *mutation analysis* are two common test adequacy criteria (§2.2). Students are often expected to meet some threshold using one of the above criteria for their tests to be considered "thorough" [1, 23]. That is, they are incentivized to write "strong" (according to the criterion) test suites in their programming projects.

Prior work has compared test adequacy criteria in terms of *defect-detection capability*; mutation analysis is a far stronger and more reliable criterion than branch coverage [24, 35, 46, 53]. Other studies have explored the thought processes of students [20] and software engineers [5, 28] as they compose test suites and of students as they write programs [8]. We add to this body of work by exploring students' interactions with different test adequacy criteria while composing test suites.

This paper addresses the research question: **How do students make use of feedback from test adequacy criteria as they compose software test cases?** We report on qualitative results from 12 one-on-one interviews in which students were asked to think aloud while completing software testing tasks, being guided by different testing feedback mechanisms (test adequacy criteria). Testing tasks were carried out using an application we built for researching and assigning software testing exercises. We used grounded theory to explore how students approach a software testing task and interact with various feedback mechanisms. Grounded theory is appropriate when little is known about the phenomenon being studied, and we are aware of little prior work exploring the interaction between students (or indeed, professional developers) and test adequacy criteria as they create software tests. Where possible, we build on concepts defined in prior work ([5, 28]).

2 BACKGROUND

2.1 Software testing in fundamental CS courses

Testing is taught at various levels in the undergraduate CS curriculum, e.g., in dedicated software testing courses [4, 12], in introductory programming courses [23, 37], or integrated holistically into the curriculum [38]. In this paper we are concerned with software testing instruction in fundamental CS courses (e.g., CS 1 to CS 3). Though there is a large body of relevant work, Scatalon et al. [59] note that most scholarship in the area describes tools and experience reports, and describe a need for theory generation and experimental research about various aspects of testing instruction, e.g., how we should teach testing, how students learn testing, their perspectives on testing, and the impact of testing on their programming abilities.

Research about testing pedagogy tends to focus on students' testing *process* and test *quality*. Practice and assessment of these two aspects are tightly intertwined. Put simply, we want students to write *strong* software tests, and to write them *early* and *often* [19, 43].

Empirical evidence suggests that continuous engagement with software testing is associated with both higher-quality test suites and higher-quality programs [43]. Many have argued for teaching *test-first* styles of development like test-driven development (TDD) [23, 37, 61], while others have generally advocated for continuous engagement with testing, so long as it is done before or soon after writing the code-under-test [42].

Early testing is beneficial because it encourages the student to think systematically about the problem they are solving before large portions of their implementation have taken shape. There is some evidence that students struggle with problem comprehension [56, 66], i.e., they form an incorrect mental model of the problem and make significant progress toward a solution to the wrong problem. Automated assessment tools like Web-CAT [23] incentivize early testing by

requiring students to meet minimum thresholds of test adequacy before receiving feedback about their implementations (that is, Web-CAT will not test a student's code if the student has not tested it themselves). The authors of Marmoset [61] suggest that looking for gaps in students' tests when they seek help could mitigate the effects of their expedient help-seeking behaviors [40]. Wrenn & Krishnamurthi [66] have proposed training early testing by encouraging students to write machine-checkable input-output examples before beginning their implementations, which are then checked against instructor-written faulty implementations.

To encourage students to write strong software tests, instructors often require students' test suites to meet minimum thresholds of test adequacy in order to get full credit on assignments. For example, systems like Web-CAT [23] and Marmoset [61] base a portion of the student's project grade on the adequacy of their tests measured by criteria like branch coverage. Wrenn & Krishnamurthi [66] assess test adequacy by running students' assertions against a suite of intentionally-faulty instructor-written implementations. Finally, Goldwasser [30] proposed assessing the strength of students' tests by running their tests against all other students' implementations, and checking whether tests were able to detect known-faulty implementations.

Since students may be graded based on these criteria, they are implicitly encouraged to incorporate them into their development workflows. That is, as students are encouraged to frequently write and run their tests in order evaluate their program correctness, so they are encouraged to frequently assess the adequacy of their tests using criteria like branch coverage. Little is known about this type of self-checking behavior in students. We explore this gap in this paper by studying how students choose test cases while being guided by different test adequacy criteria (namely branch coverage and mutation analysis).

2.2 Test adequacy criteria

A test adequacy criterion describes the conditions that must be met for a test suite to be considered "adequate" [31]. Numerous test adequacy criteria have been proposed over the years, but *branch coverage* and *mutation analysis* are of primary relevance to this paper. We focus on these criteria because they can be applied *incrementally*, i.e., during the software development process [41], and they are commonly used as assessment criteria in education and industry. Instructor-written faulty implementations [66] also meet the incrementality requirement, but they are less commonly used in educational settings.

Branch coverage is satisfied when all logical branches in the control-flow graph of the program are executed by the test suite [51]. For example, for code involving `if` conditions, the condition must be made to evaluate to `True` at least once and `False` at least once. The "condition" that is checked for coverage may be composed of multiple sub-conditions connected by relational operators. The test suite's adequacy is measured as a percentage of branches that are covered.

The shortcomings of measures like branch coverage are widely known: they depend on the tests simply executing the entire program, and not on the assertions within those tests. As such, they are not a good indicator of a test suite's fault-finding capabilities [1, 24, 63] and do not strongly correlate with software reliability [34–36, 46]. Despite these shortcomings, code coverage measures are commonly used in educational settings (e.g., [12, 23, 33, 61]) since they are fast to compute, easy to reason about, and have good out-of-the-box support in tools like IDEs and continuous integration workflows.

Mutation analysis [17] is a far stronger criterion [1, 24, 41, 53]. It works by systematically generating alternate versions of the original program (*mutants*) and evaluates test suites by running them against these mutants. A mutant is considered detected or "killed" when it causes a test to fail or otherwise behave badly (e.g., time out or crash with an exception). The adequacy of the test suite—its *mutation coverage*—is thus measured as the percentage of mutants it kills.

Mutants are generated by making syntactic changes to the program (e.g., changing `>` to `<=`, or replacing an entire Boolean expression with `true` or `false`). The types of changes that can be made are called *mutation operators*. While there can be infinitely many mutation operators in principle, they have been designed to emulate errors commonly made by programmers [44].

Mutation analysis subsumes branch coverage [53]. This means that if a test suite satisfies mutation analysis (i.e., all killable mutants are killed), then it satisfies branch coverage. The reverse is not true—satisfying branch coverage is not an indication that a test suite also satisfies mutation analysis.

Numerous empirical studies have made a case for the validity and effectiveness of mutation analysis. Mutants generated by common mutation operators have been shown to be usable substitutes for "real" software defects [32, 39]. Moreover, Just et al. report that developers write more and better tests when using mutation analysis for feedback [55].

Educators have considered using mutation analysis to assess student-written software tests [1, 24, 33, 41]. Aaltonen et al. [1] report that it is a more difficult criterion than code coverage for students to satisfy, leading them to write stronger test suites. These findings were echoed by Edwards et al. [24]. Contrary to other findings, Hall et al. [33] report that students wrote *fewer* test methods when they were guided by mutation-based feedback than when they were guided by coverage-based feedback. However, their tests were more complex when using mutation-based feedback: they used more of the assertion types available in the testing library, and each test method contained more assertions on average.

Shortcomings of mutation analysis include its high computational cost and the possibility of generating *equivalent mutants*.[1] Previous work [41] has explored methods to reduce the computational cost of mutation analysis such that it could reasonably be deployed in automated assessment tools like Web-CAT.

We are not aware of prior work that explores the *cognitive* aspects of mutation-based feedback and code coverage-based feedback in educational settings. We take steps toward this in this paper.

2.3 Models of software tester cognition

Enoiu et al. propose the first framework that we are aware of describing the cognitive process of software testers [28]. Their model describes software 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 results. They also discuss other influences on the test-writing process like motivation, creativity, psychological factors, and social factors.

In follow-up work [27], Enoiu and Feldt note that identifying cognitive processes just from written tests can be difficult. They recommend the use of verbal protocol analysis (having participants think aloud) to better identify explanations for behaviors. In this respect, our work complements the work of Hall et al. [33]—they quantitatively analyzed tests that were written by students guided by coverage- and mutation-based feedback, while we used a think-aloud procedure to explore students' interactions with these feedback mechanisms (§3.2).

Aniche et al. followed the recommended think-aloud procedure and observed developers as they wrote tests for specific pieces of software [5]. Based on their results, they proposed their own model of software tester thought processes. They identified six main concepts and the relationships between them that the testers would use as they worked. Specifically, developers would build up a *mental model* of the program as they worked, guided by the *source code* and the *documentation*.

[1]Equivalent mutants are functionally identical to the original program, and therefore cannot be detected by a test suite. The problem of identifying these mutants automatically is undecidable in general; it is the subject of a large body of research that is out of the scope of this paper [48].

They would select *test cases* based on this mental model and aim to satisfy an *adequacy criterion* (specifically, code coverage in [5]) with their test code.

The models described above inform our data collection, analysis, and interpretations. Following recommendations from Enoiu and Feldt [27], we used a think-aloud protocol to gain insight into students' thought processes while they composed software test suites. Test adequacy criteria are good examples of "test goals" described in Enoiu's model—an important role of test adequacy criteria is to guide the developer to test parts of their software.

Additionally, we draw inspiration from the work of Pennington [54] and Castro & Fisler [8], who emphasize the importance of "plan knowledge" or "task-level thinking". Castro & Fisler observed how novice programmers shifted between thinking about a programming problem at the task level (focusing on task decomposition) and the code level (focusing on the source code). They noticed that students who moved back and forth between task-level and code-level thinking tended to fare the best, while those who stayed at the code level tended to perform the worst. We noticed similar movements while students wrote tests, though we did not note any significant associations with specific test adequacy criteria.

We build on the works described above (particularly those of Enoiu et al. [27, 28] and Aniche et al. [5]) by focusing on how students' test-selection strategies vary based on the test adequacy criterion that is guiding them.

3 METHODOLOGY

We conducted think-aloud interviews in which students were asked to complete software testing tasks while being guided by feedback from a test adequacy criterion. We describe tooling we built for data collection (§3.1) and our interview (§3.2) and analysis (§3.3) procedures.

3.1 Data collection tool

Before carrying out the interviews, we attended to certain confounding factors. First, different levels of familiarity with testing libraries could affect participants' test-writing habits. Second, branch coverage and mutation analysis enjoy differing levels of integration with software development environments. IDEs like PyCharm and VS Code have good out-of-the-box support for displaying branch coverage feedback, but most mutation analysis tools provide feedback simply as lists of killed or surviving mutants, often as textual output printed to the standard output stream (see §7.2). This discrepancy could further affect how participants respond to different feedback mechanisms.

To avoid these confounding effects, we built an application called *Muttle* to help us collect data for this study (depicted in Figure 1). This section briefly describes its key features.

Muttle provides an interface for students to complete testing exercises in which they are presented with a Python function and a problem statement, and are asked to write tests for the function. Muttle uses `pytest` [47] to execute test cases, but this is hidden from the user. Test cases are written as input-output pairs (corresponding to individual *assertions* as opposed to entire test methods). For each test case, the user writes one or more comma-separated input values (corresponding to the parameters accepted by the function) and one expected output value. Any valid Python expressions can be used as inputs and output. This precludes the need for familiarity with any Python testing library. Users can freely add and remove any number of test cases, and can run their test cases and receive feedback about them at any time.

Muttle does not display testing feedback until the user submits a test suite containing one or more tests. It supports three feedback modes. Each successive feedback mode includes the previous one. The first is **no feedback** (NoFeedback). This mode simply tells the user whether each test

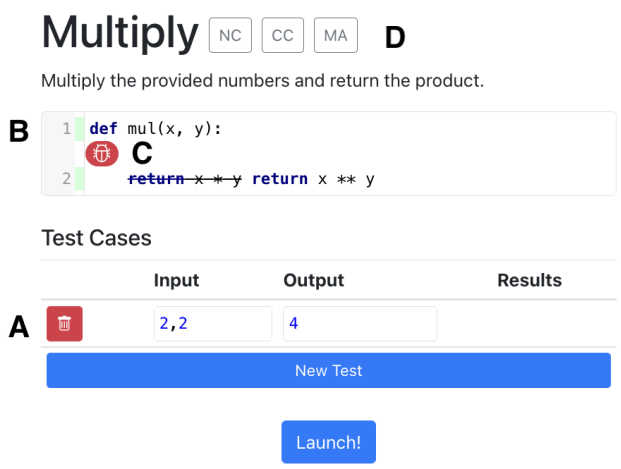

Fig. 1. The *Muttle* interface. Each exercise lets the user create, modify or delete test cases (A). If their test cases pass, they are given BranchCov feedback in the colored gutter next to the line numbers (B) or MutationCov feedback in the form of bug badges above the lines of code where mutations were made (C) In this example, the statement `return x * y` was mutated to `return x ** y`. The interviewer used the toggles at the top of the screen to switch between no coverage, code coverage, or mutation analysis feedback, based on the experimental condition (D).

passed or failed (including if the test failed to compile), but gives no indication of the thoroughness of the test suite. This indication is shown in all feedback modes.

Next, Muttle displays **branch coverage** information (BranchCov). It displays this information using a form that is common in IDEs, i.e., the gutter of the editor is colored red if the line was not executed at all, green if the line was fully covered, or yellow (for branching lines) if the line was partially covered. Branch coverage is computed using `pytest` [47].

Finally, Muttle supports **mutation coverage feedback** (MutationCov), represented by bug badges appearing above lines of code that contain surviving mutants. Clicking on a badge displays the original and mutated code side-by-side. The original code has a line drawn through it. We are not aware of mutation analysis tools that communicate feedback using this form. We discuss this further in §7.2. Since mutation analysis subsumes branch coverage [53], BranchCov feedback is also displayed in the gutter in this mode. Mutation analysis is conducted using `MutPy` [18].

3.2 Think-aloud interviews

3.2.1 Interview procedure. We conducted 12 one-on-one think-aloud interviews in which participants were asked to devise test cases for Python functions that were chosen by the researchers. Our primary data sources were audio and screen-capture recordings of the interviews and the accompanying transcriptions. The following protocol is similar to that of Whalley et al.'s think-aloud study on debugging practices [64]. It was approved by our institutional review board (IRB).

The study was conducted at a public, medium-sized, primarily-undergraduate university in the USA. We recruited students who had *completed* the first and second programming course in our introductory course sequence for computer science majors: an introductory programming course and a Data Structures course taught in Python. Students were recruited by making announcements in sections of the course that follows the Data Structures course, an Object-oriented programming (OOP) and design course taught in Java. Interested students reached out to the researchers over

email and interviews were scheduled with everyone who expressed interest (i.e., we did not choose specific students to interview).

Our university runs on the quarter system (i.e, 10-week academic terms as opposed to 15–16 weeks). Interviews were conducted during the Spring 2022 term. The participant pool was made up of mostly first-year computing (Computer Science or Computer Engineering) majors, who had typically taken the previous two CS courses during the previous two academic terms. Non-computing majors (who are typically achieving a minor in CS) tend to take CS courses later in their post-secondary education—this applies to two students who were in their second year, and one who was in their third year.

Interviewees generally performed well in the first CS course, but there was a larger spread of performance in Data Structures, with letter grades ranging from A to C+. Still, having passed the previous two courses, participants were considered to be "accomplished novices" in terms of their Python programming abilities. In particular, the Data Structures course involved fairly complex data structures (e.g., a Huffman tree) and recursive algorithms. Assignments in the previous courses required unit testing, and students had previously been graded based on the branch coverage achieved by their tests. So they were familiar with that particular adequacy criterion.

Each interview took 30–45 minutes, and students were compensated with a $25 Amazon gift card for their participation.

Interviews were carried out in person by one author of this paper. All interviews began with a short demographic survey, following which the researcher provided the participant with a laptop that was running a locally-hosted version of Muttle (§3.1). Participants used this laptop to create test cases and used the provided feedback to check their progress, while thinking out loud. The laptop was also running a Zoom call with no other participants; this was used to capture voice and screen recordings of the sessions.

The testing session began with the researcher giving the participant an introduction to the Muttle interface, and giving them a "warm-up" function for which they wrote tests. This warm-up activity also allowed participants to get used to "thinking out loud". Following this, participants were given 3 functions to test, one after the other. They were told their job was to "test the function". For each function, the participant was given a different form of feedback (NoFeedback, BranchCov, or MutationCov). Students were not required to stop testing after satisfying a given feedback condition. The interviewer returned to the warm-up problem between each function to demonstrate each new feedback mechanism.

After an initial adjustment (§3.2.3), all participants tested the same 3 functions. The order of functions was rotated from one interview to the next. Each function-feedback pair was attempted by two participants in an effort to minimize effects arising from particular participant-function-feedback combinations.

The order in which feedback conditions were presented to participants remained consistent through the interviews, since each feedback type is subsumed by the subsequent type (in the order NoFeedback → BranchCov → MutationCov).

We imposed a hidden time limit of 20 minutes to test each function, but that time limit was never reached.

In sum, interviews proceeded as follows:

(1) Warm-up problem with NoFeedback
(2) Problem 1 with NoFeedback
(3) Warm-up problem with BranchCov
(4) Problem 2 with BranchCov
(5) Warm-up problem with MutationCov

(6) Problem 3 with MutationCov

3.2.2 Interview procedure updates. Following the principle of constant comparative analysis [29], we analyzed data after each interview and adjusted our data collection procedures accordingly. In particular, some changes were made to the interview procedure as interviews progressed.

"Is this program correct?" Occasionally the participant asked the interviewer if the program they were testing was correct. As a reflex reaction the first time this happened, the interviewer said "yes". We gave non-committal answers to this question in future interviews.

Explaining MutationCov to novices. We struggled to find a novice-friendly way of explaining MutationCov feedback to students. Eventually, we settled on "if this bug existed in your code, none of your tests would have caught it and failed".

Stable ordering of feedback mechanisms. We originally rotated the order of feedback conditions from one interview to the next. Since feedback mechanisms built on one another, we found that it made the most sense to consistently present students with feedback in the order NoFeedback → BranchCov → MutationCov. This change was made after the first 3 interviews.

More detailed program descriptions. We noticed that participants asked more clarifying questions for Triangle and Centered Average than for Rainfall, perhaps due to the longer description for Rainfall. We added more clarifying detail to the descriptions for Triangle and Centered Average. We did not receive noticeably differing levels of clarifying questions after that.

Updates to selected problems. In early interviews we noticed that some functions did not give students an opportunity to engage much with the testing feedback. These problems were replaced with others. See §3.2.3 for details.

3.2.3 Problem selection. We chose a set of functions that had good coverage of programming concepts that the students would be familiar with (e.g., loops, conditional control flow, arithmetic). We arrived at these problems after conducting pilot interviews before beginning the research study with the intended participants. The descriptions and code used for each problem are available in Appendix A.

Multiply. This warm-up function took two inputs and returned their product. Due to its simplicity, it did not provide opportunities to demonstrate the different feedback mechanisms, so it was replaced with Larger.

Larger. This warm-up function returned the larger of two numbers. It was written relatively verbosely since it was only used to demonstrate branch coverage and mutation analysis, which need more program constructs to generate feedback.

Triangle. To include conditional branching logic, the next function was the triangle classification problem, which is commonly studied in software testing textbooks [2] and research papers [41, 45]. The function takes in three side lengths and returns a number indicating the kind of triangle they form (1 if equilateral, 2 if isosceles, 3 if scalene), or 0 if they do not form a valid triangle.

Rainfall. We included the rainfall problem, which is commonly used in computing education research as a measure of novice students' programming abilities (e.g., [8, 49]). The function takes in a list of numbers (daily rainfall) and computes the average rainfall. It stops counting rainfall when it comes across a sentinel value (99999), and must discard negative values as they are invalid.

Selection Sort. The function performs a selection sort on a given input list. Participants did not receive useful testing feedback on this function. The first participant reached 100% branch coverage with a single test case, and the second participant received only equivalent mutants. These did not allow us to study how the participant responded to the feedback, so we replaced selection sort with Centered Average.

Centered Average. The function computes the mean of the given list of numbers but excludes the minimum and maximum values from the computation. If there are multiple instances of the minimum and maximum, it excludes only one of each. The function assumes that there are at least two items in the list.

3.3 Analysis

We employed a grounded theoretic approach [29] to analyze participants' thought processes as they wrote tests for a series of Python functions. We aimed to characterize their test input selection strategies when they are given feedback based on different test adequacy criteria (NoFeedback, BranchCov, or MutationCov).

Our goal in this work is exploration of students' interactions with mechanisms for feedback about their software tests. As such, we did not approach the analysis with specific research questions or overarching hypotheses in mind. Grounded theoretic approaches are appropriate when little is known about the phenomena being studied [9]. Since there is a scarcity of prior work on software developers' thought processes while they test, and none we are aware of that take the adequacy criterion into consideration, we believe this is an appropriate methodology for this work.

Two researchers conducted open coding on the interview transcripts, preparing memos noting individual moments of interest. We consulted the screen and audio recordings to resolve ambiguities in the transcript, e.g., if the participant said something like "let me try *this*" and added a new test case, or hovered their mouse to indicate portions of the code without reading the code out loud (as an example, see code 31 in Table 2). As an example of an incident of interest, one participant mentioned that they were writing a simple-to-understand test to start with. This snippet was tagged with the code "writing a basic, easy-to-reason-about test". Another participant mentioned that, due to a programming course they have taken, they tend to think in terms of BranchCov even when writing tests without feedback. This was tagged as "prior experience with BranchCov leads student to think in terms of BranchCov on their own".

The open coding process led to a codebook of 39 codes. Two researchers individually coded the first six interviews and then met to combine these initial codes into a codebook. Any matching or closely related codes identified by both researchers were added to the codebook. Where only one researcher identified a code, the incident was examined by both researchers together (using the Zoom screen and audio recordings) before a code was agreed upon and added to the codebook. After the codebook was created, both researchers used it as a guide to gather additional snippets from the remaining interview transcripts and recordings. New codes did not emerge from this process.

These initial codes were further analyzed in an axial coding process. The goal of the axial coding step is to take the concrete initial codes and abstract them into higher-level themes or categories. Higher-level categories were developed inductively by examining the initial codes and, where appropriate, referring to extant literature. Codes were organized into categories based on the following emergent factors.

- The stage of the test writing process (E.g., Had the student received any feedback yet? Had they read and understood the program or description?)
- The test adequacy criterion used for feedback,
- Concepts in the cognitive models proposed by Enoiu et al. [28] and Aniche et al. [5], and
- Whether the participant appeared to be thinking terms of the problem or program [8, 54]

For instance, the two example initial codes described above were both included in the higher-level category "testing with no feedback" (i.e., the first set of tests written before any feedback

is presented). One researcher carried out the intermediate coding process following which both researchers discussed the proposed categories and arrived at a shared understanding of them.

3.4 Acknowledging researchers' positionality

Qualitative coding of interview transcripts and recordings involves researchers' interpretations of participants' words and actions. The process is therefore inevitably influenced by the researchers' perspectives, which we acknowledge in this section. Both researchers were men. The first researcher was relatively new to computing education and related research—they had taught one introductory programming course and ran the departmental peer tutoring center. They also had taken the same CS courses as the interviewees a few years prior, when they were a student at the same institution. All interviews were carried out by this researcher. The other researcher was more experienced in computing education and related research, having taught and researched in the field for around 6 years. As a result, the research team brought mixed perspectives to the analysis: one with experience in computing education research—particularly with research on student software testing—and one with a perspective that was far closer to that of the students we interviewed.

4 RESULTS

We present our results as compelling phenomena that ought to be further investigated. The emergent themes are organized into a process model of students' test writing in §5. The complete set of codes accompanied by illustrative quotes or incidents is provided in Appendix B (Table 2). Codes in Table 2 are grouped based on their parent categories.

Where applicable, we include quantitative observations from data collected in Muttle along with our qualitative results (Table 1). Since participants were asked to test functions at least until they satisfied the provided feedback criterion, their coverage *of that criterion* reached 100% in all cases. That is, in the BranchCov condition, there is little to learn from branch coverage scores, since it was satisfied in all cases. In the MutationCov condition, there is little to learn from participants' mutation coverage scores *or* branch coverage scores, since both were satisfied in all cases.

However, two measures are of interest. First, the *number of test cases* the participant wrote under a given feedback mechanism. Second, when students are given feedback based on less stringent criteria (like NoFeedback and BranchCov), it is instructive to study their test suite's performance according to the more stringent criteria. For example, it is instructive to study the number and types of mutants that survive a branch coverage-adequate test suite.

4.1 Problem and program comprehension

This theme is concerned with how students formed and developed their understanding of the functions they were asked to test, corresponding to the "mental model" described in the framework of Aniche et al. [5]. We are concerned with the initial formation of and subsequent updates to the participant's mental model of the problem and program under test.

All participants started by either reading the problem statement or by reading the code line-by-line in order to understand the function they were asked to test. Similarly, 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" [5]. Some participants started by reading the function implementation (P2, P3, P6, P10, P11) and only referred to the problem description when further clarification about the function's requirements were needed.

> [In Rainfall, on reading the description] *Because that way it's easier to understand what the code is doing, especially since it's not commented.* (P6)

Other than one participant (P5), all participants who started by reading the description followed this up by reading at least part of the program before testing.

Participants employed a number of strategies to update their mental models as their understanding of a program evolved. Most participants (P1, P2, P3, P5, P6, P8, P9, P11, P12) read or re-read the problem description, but some would (re)turn to the code (P1, P3, P4, P5, P6, P10).

In all three feedback conditions, a common code comprehension strategy was to rely on recognition of *variable roles* [57] (P1, P2, P3, P4, P6, P9, P11). For example, participants recognized variables as being temporary holders of minimum or maximum values (in Selection Sort and Centered Average). This understanding helped them to quickly process the source code at hand ([58]), and also to quickly reason about test cases that would execute particular branches or distinguish a mutant from the program's original behavior.

> [In Centered Average] `min_index = 1` , *I guess we just write a test where everything is less than one.* (Participant actually misunderstood the role of `min_index` here.) (P1)

Only one participant (P3) used a test as a meta-cognitive scaffold to confirm their understanding of the problem prompt and program before testing ([56, 66]), though others used feedback from failing tests to help them correct their misunderstandings (P5, P7). For example,

> [In Rainfall, runs a test and that the output did not match the expected value] *Expected 2...Wait. Am I missing something? It's 6...Oh, but is 0 a positive number?* (P5)

4.2 Responding to testing feedback

The three feedback conditions we employed were NoFeedback, BranchCov, and MutationCov. Participants tended to write the most test cases when they were given MutationCov feedback (median 6.5). Interestingly, they tended to write more tests when they were given NoFeedback (median 5) than when they were given BranchCov feedback (median 4). Perhaps relatedly, two participants (P2, P3) mentioned that they were putting in more testing effort since they were participating in a study about testing.

Table 1 depicts the median number of tests written in each feedback condition and the test adequacy achieved according to different criteria.

4.2.1 Testing with No Feedback. Some participants tested the code while reading it, either going over the entire function or just part of it, formulating tests as they went. Of the students who tested the code line-by-line in its entirety, two (P2, P3) jumped right into it without reading the function description at all, while another two (P1, P6) read the description first. Four students (P8, P9, P10, P11) wrote tests while reading the description.

> [In Triangle] *But like with this, especially like with this setup, I can look at my code and write tests at the same time so I can just like go line-by-line and like look at each part like each line and make sure that my code is like hitting every section.* (P6)

Intuitions about "edge cases". Every participant's testing efforts were at least partially based on their own intuitions about "edge cases". When asked the reason behind this, participants pointed to the Data Structures course that they had recently completed. That course included an automated grading system in which instructor-written tests tended to include inputs of different data types (integers/floating point values) and boundary values (positive/negative numbers, zero, empty lists). Participants prioritized edge cases mirrored by these types of boundary values.

> [In Rainfall] *I think there would be something suspicious that would happen if it was just a day with zero rainfall.* (P1)

Table 1. The median number of tests in the final test suite for each problem under each feedback condition, and the test suite's adequacy according to different criteria. Each row labeled "Median" depicts the median values across all problems within that condition. The table does not include problems that were removed from the interviews (§3.2.3).

Feedback	Problem	# Tests	BranchCov	MutationCov
NoFeedback	Triangle	7.5	100%	89%
	Centered Average	6	100%	80%
	Rainfall	6	100%	96%
	Median	6	100%	89%
BranchCov	Triangle	5.5	100%	91%
	Centered Average	1	100%	77%
	Rainfall	4	100%	96%
	Median	4	100%	89%
MutationCov	Triangle	10.5	100%	100%
	Centered Average	5.5	100%	87%
	Rainfall	6	100%	100%
	Median	6.5	100%	100%

Mentally simulating BranchCov. Some participants used a mentally-approximated form of branch coverage even when they weren't given that feedback (P3, P4, P11, P12). They mentioned that their tests had been assessed using the criterion in previous programming assignments. Additionally, before we settled on a consistent ordering of feedback mechanisms, one participant (P3) had received feedback using BranchCov and MutationCov before reaching the NoFeedback condition—this may have induced them to think in terms of one of those criteria.

> [In Triangle, before any feedback has been displayed] *So I guess I'll just write tests for all these three* [Participant hovers mouse over lines 2–4 in Triangle] *and...OK, so just going off like the* `if` *statements or...yeah* `if` *statement.* (P4)
> [In Rainfall] *Like I feel like ever since [recent CS course], like we have to have full coverage all the time, I kind just feel like I have to look at everything so everything runs.* (P3)

Testing "beacons". While testing with NoFeedback (or while writing initial tests before feedback has been generated) some participants appeared to write tests to target specific features or "beacons" in the code or description that caught their attention. Beacons are parts of a program that help a reader understand what the code does (comments, variable or function names, etc.) [15]. In this case, they appeared to drive participants' testing efforts to an extent. The `if days == 0:` conditional check in Rainfall and the final standalone `return 3` in Triangle are examples of code that "stood out" enough that students (P4, P9, P10, P11, P12) were drawn to writing a test for those lines in particular, with some (P1, P2, P6, P10) going a step farther and writing multiple tests in order to pre-emptively reach full (mentally-approximated) branch coverage. Similarly, most participants were drawn to testing the sentinel number requirement in the Rainfall description early in their testing process (P1, P3, P5, P7, P8, P11, P12), and one participant (P11) stated that they wanted to first trigger every `return` statement in Triangle.

> [In Rainfall] *I...first, I guess, for why-notsies I'll...I'm first gonna write something just that breaks this* [Hovers mouse over `rain_day == 99999`] *just because I want to make sure that this runs.* (P3)

On the whole, test suites written with NoFeedback were quite thorough. They scored a median 100% BranchCov and median 89% MutationCov. Surprisingly, test suites produced in the NoFeedback condition tended to be as strong as tests produced in the BranchCov condition, as measured by MutationCov (see Table 1).

4.2.2 Testing with Branch Coverage. We expected that students would engage in more *code-level thinking* (as described by Castro & Fisler [8]) when driven by BranchCov, a structural adequacy criterion. And they would engage in more *task-level thinking* when driven by MutationCov, a fault-based adequacy criterion.

We did not find qualitative support for this expectation. Of the students who had any significant BranchCov feedback (P1, P2, P4, P7, P9, P12), only one (P4) appeared to rely on significant task-level thinking (i.e., they verbalized a test plan based primarily on the high-level problem description). The rest focused on mentally executing the program with their test case inputs, using the coverage gutter to direct them to uncovered or partially covered statements. It should be noted that in the study by Castro & Fisler, the students were taught using a pedagogy that emphasized programming plans and testing.

As described in §4.1, program comprehension was an important step in the testing process. Participants developed strategies to manage the cognitive load associated with program comprehension for the purpose of satisfying a test adequacy criterion.

Tracing basic blocks. While considering test case inputs, some participants traced *individual basic blocks* in isolation as opposed to tracing the entire program (P2, P3, P7, P9, P12). A *basic block* in a program is a straight-line sequence of statements such that "if the first statement is executed, all statements in the block will be executed" [53]. For example, a function with cyclomatic complexity of 1, or the `then` clause of an `if` condition (provided it doesn't branch further).

Variable roles. Additionally, participants (P1, P2, P3, P4, P6) again employed their knowledge of variable roles [57]. For example, consider the following code in Selection Sort:

```
if input_list[j] < input_list[min_idx]
```

When faced with partial BranchCov for this condition, one student (P1) used their knowledge of the roles of the current and minimum index variables to deduce that they needed a test case where a new minimum index was never found.

Ignoring feedback. Some participants (P2, P3, P9) chose to ignore BranchCov feedback in favor of testing boundary values. When asked why, they said they were unsure of how to reach full coverage for a statement, and that they would return to it once they had addressed boundary values. Others expressed a desire to base their tests on the problem description and not the code (P7, P12).

> [In Centered Average] *Definitely code coverage is like something I try to catch after I've written test cases [to my own satisfaction].* (P7)

Finally, the median participant wrote the fewest tests in this feedback condition (Table 1), meeting expectations that it was easier for participants to satisfy BranchCov than MutationCov.

4.2.3 Testing with Mutation Coverage. Reasoning about MutationCov feedback appeared to be a high-cognitive-load activity. Even after demonstrating an understanding of the idea behind mutation analysis (by successfully killing other mutants), many participants struggled to devise test cases to kill certain killable mutants (P2, P3, P4, P6, P8). In particular, difficulties seemed to arise when mutants appeared in conditional statements (e.g., conditional operator replacements).

To devise test cases to kill a mutant, participants needed to develop and maintain an understanding of the mutated program in parallel to their understanding of the original program. They found this parallel tracing of two programs to be fairly demanding. Sometimes this led to mistakes. For example, the effort of comprehending a mutant seemed to distract P4 from their task to the extent that when they wrote a test case, they wrote it so that it would *pass the mutant program* and *fail the original program.*

> [In Rainfall] *So like given this [for loop], so it's running through, everything's running through 1, it's running through 2, running through 2, and so on and so forth, and then, then it's going to...* [trails off] *You know, I'm not entirely sure how to squash this bug.* (P10)

When this dual code comprehension task overloaded participants' working memory, they developed ad-hoc coping strategies.

Tracing mutants. Participants traced the mutant program line-by-line with a specific input value, similar to how they traced the original program while forming their initial mental model of the function. They would focus on identifying the point during execution where the mutated program would deviate from the original program (P3, P4, P6, P7, P8, P11, P12), if such a point exists.[2]

Our observations echo findings from Middleton & Stolee [50]. In their study, when developers were asked to tell the difference between two code snippets, they either traced one entire code snippet before moving onto the second one, or moved frequently back-and-forth between the snippets, comprehending smaller chunks of each in tandem. Middleton & Stolee refer to this latter strategy as *structural comparison*, i.e., the developer comprehends a chunk of discrete behavior in one snippet before moving on to consider the next snippet. This is similar to the way in which our participants traced entire mutant programs (P5, P8) or, like students in the BranchCov condition, limited their tracing to the basic block in which the mutant appeared (P2, P11, P12).

> [In Rainfall] *So if it was...* `rain_day` *is the thing, the element of the list. So if* `rain_day` *is greater than 0, it should run. But if it's greater than 1, it wouldn't. Yeah, I'm trying to like run through what the code is doing I guess.* (P2)

Inputs for tracing tended to be chosen based on boundary values (commonly in terms of the input data type, but sometimes in terms of the input domain). For example, in Rainfall, the line `if rain_day >= 0` is mutated to `if rain_day >= 1` . One participant (P3) guessed that the mutant would be killed by a test case that included the value `1` , but struggled to verbalize why in terms of the problem description.

Satisfying BranchCov. In another common strategy, participants resorted to simply satisfying BranchCov for conditional statements where mutants appeared. They reported a "feeling" that executing all branches of the original program would likely kill surviving mutants appearing in conditional statements (P2, P3, P4, P6, P7, P8, P9, P10, P11).

> [In Rainfall] *So I was wondering [about] this bug and I realized that I hadn't tested 1, even though I said it was a special number you know, in my opinion. So I was like, oh yeah, I should just try it.* (P2)

Since mutation analysis subsumes branch coverage [53], this was not expected to be a sound strategy. However, the heuristic appeared to be fruitful. Half of the surviving mutants in NoFeedback and BranchCov sessions were relational operator replacements (e.g., `>` is replaced with `>=`). Other types of mutants targeting control flow (like Logical Connector Replacement[3], Conditional

[2]The interviewer manually pointed out equivalent mutants and asked the participants to ignore them.

[3]E.g., `and` is replaced with `or` .

Operator Insertion[4] and One Iteration Loop[5]) were killed in the BranchCov sessions, but not in the NoFeedback sessions. It appears that satisfying BranchCov in general may lead to the killing of a substantial number of mutants that target control flow constructs. This has implications for reducing the computational and cognitive cost of mutation-based testing feedback.

Ignoring mutants. Others implicitly relied on a similar strategy (P5, P7). They did not click to see the mutated source code, but instead relied on the *presence* of mutants (i.e., bug badges in Figure 1) to guide their intuition-based testing efforts.

> [In Triangle] *OK, I think they just want me to check for each side. This* [hovers over bug badges appearing over an `if` statement] *basically just means I think that like it doesn't really matter like what's coming out of here, so I should probably like actually test something that will check that [condition].* (P6)

Almost like a form of structural testing like code coverage, the presence of mutants indicated to these participants that those parts of the program were not "tested enough". This is similar in spirit to the "heat maps" proposed by Edmison & Edwards [21, 22] which use a combination of instructor-written tests and branch coverage from the student's own tests to highlight "suspicious" (untested) portions of the student's code.

Another participant took this dismissal of MutationCov feedback further and simply ignored it until their own testing methods did not kill any more mutants (P9). As with BranchCov, only one student (P1) demonstrated any significant task-level thinking when understanding and responding to MutationCov feedback.

4.3 Other patterns

We noticed some general trends that were not exclusive to any single feedback mechanism.

Every participant wrote at least one basic, easy-to-reason-about test, often as the first test in their suites. While some of these tests were written to meet an adequacy criterion or to better understand the program, most often they were "happy path" tests that only simulated typical program inputs.

Since Muttle does not display any testing feedback at first, some participants (P1, P2, P6, P7, P9) would start by generating initial test adequacy feedback by writing one or two simple tests. After receiving initial feedback, participants would proceed to focus on achieving a complete test adequacy score and would not rely on their own test-writing habits until afterward (P1, P2, P7, P9).

Some students (P1, P3, P5, P11) mentioned wanting to "put the code through the most paces". This sentiment took on different meanings in different contexts. Sometimes the participant wanted to assure themselves that a test suite would reach all lines in the program before executing their tests and receiving feedback.

Also common was the practice of copying and editing an existing test case instead of writing new ones, allowing the participant to incrementally build up their test suite's (branch or mutation) coverage (P1, P2, P3, P5, P6, P7, P8, P9, P10, P11). This was also observed by Aniche et al. [5].

Some participants (P2, P5, P6, P7, P12) mentioned that they preferred to write tests based on the problem descriptions rather than source code, but only 3 actually put this into practice (P5, P7, P12). P5 in particular mostly ignored the source code and all forms of testing feedback during their session, and achieved high coverage in both the BranchCov and MutationCov conditions based solely on their understanding of the problem description.

Finally, virtually no students returned to considering the problem description or requirements after any testing feedback was generated. That is, once initial testing feedback appeared, meeting the criterion became the primary "testing goal".

[4]A conditional expression is negated using the `not` operator.

[5]A `break` is added at the end of a `for` loop, causing it to terminate after one iteration.

5 TEST-WRITING PROCESS MODEL

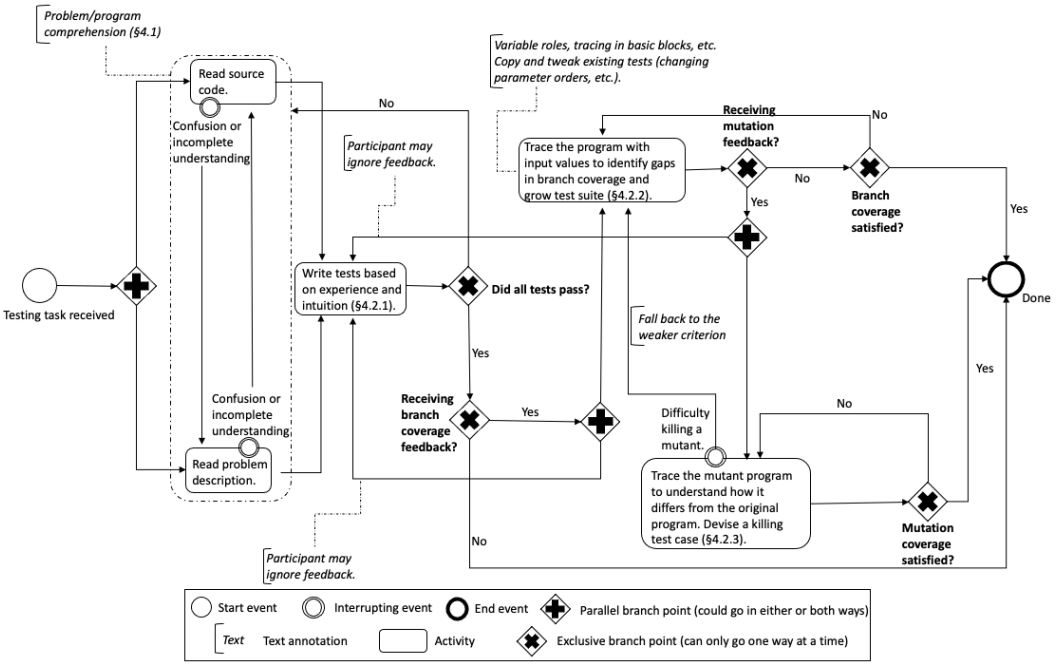

Fig. 2. A process model (§5) for how novices write software tests while being guided by a test adequacy criterion.

We synthesize our results into a process model describing how relative novice programmers create test suites for a given piece of code while being guided by a test adequacy criterion. Note that this is a *descriptive* model rather than a *prescriptive* one. Some aspects may represent imperfect testing strategies, or there may be steps *missing* in the model that we want to see in students' testing strategies. Identifying these aspects and addressing them through targeted instruction is therefore a worthwhile goal. For example, feedback from mutation analysis and code coverage did not appear to influence or be influenced by problem comprehension.

The model draws heavily from concepts defined by Aniche et al. [5] (summarized in §2.3), with the following changes:

- Where the model in [5] refers to a "mental model", we refer to *program and problem comprehension*, to better contextualize the model with existing computing education literature.
- We do not distinguish between the act of choosing test cases and writing test code, since tests in the Muttle interface are simply denoted as input-output pairs (§3.1).
- We include an additional focus on the specific adequacy criterion used by the tester, drawing parallels and differences between testing with BranchCov and MutationCov.

The model is summarized in Figure 2 and described below.

Problem and program comprehension strategies and abilities appear to be intricately tied to students' abilities to devise useful test cases (§4.1). Students would begin their test writing process by forming an initial understanding of the problem and program. They often began by studying the program, using the problem description to address confusions if they arose. In some cases, the student's background knowledge also influenced their understanding of the problem.

Difficulties with comprehension typically led to difficulties with testing, echoing findings from Aniche et al. [5] and Bai et al. [7].

After achieving an initial understanding of the problem and program, the student writes an initial set of tests before any feedback is produced (§4.1, §4.2.1). These test cases were chosen based on the students' intuitions and experiences. They chose inputs representing boundary values, citing past experiences with automated assessment tools that tested their submissions using similar inputs (empty lists, zero, negative numbers). Initial tests also tended to be based on problem and program "beacons". With no hints about what to test first, students were drawn to testing prominent portions of the problem description or program. Occasionally, students would mentally simulate branch-coverage while considering inputs, citing prior experience with the criterion. If any of these initial tests failed, the student would re-evaluate their understanding of the problem and program.

With an initial set of tests in hand, the student receives feedback based on a test adequacy criterion, and may attempt to devise test cases to satisfy the rest of the criterion (§4.2.2, §4.2.3). In both the BranchCov and MutationCov conditions, they turned to well-known code comprehension strategies (like variable roles [57] or identifying beacons [15]) while selecting test cases. They would manage the cognitive load associated with program comprehension (particularly in the context of achieving branch or mutation coverage) by focusing on slices of the program, i.e., basic blocks. Occasionally, the student chooses to ignore the provided feedback in favor of testing based on what they perceive to be "edge cases".

The student "falls back" to a weaker criterion when satisfying a strong criterion is difficult (§4.2.3). Achieving 100% mutation coverage proved to be a difficult task for many students. Identifying an input to kill a mutant requires maintaining a parallel understanding of the original program and the mutant program, and identifying where their executions diverge. This is a high-cognitive-load activity that at times appeared to overwhelm students' working memory. When this happened, the students employed the code comprehension strategies described above. When those did not help, they resorted to heuristically addressing a weaker test adequacy criterion in the hope that the resulting test suite would also address the stronger criterion. This strategy was sometimes useful.

Occasionally, students continued to add to their test suites even after the test adequacy criterion was met, writing tests that had no effect on the test suite's (branch or mutation) coverage (§4.3). They would use (what they believed to be) pathological values for the input data type, even if the values did not represent unexplored equivalence partitions in the problem space. That is, it appeared that they were testing with these values *as a reflex*, and not due to any reasoning about the properties of the program or problem. This occurred more often in the BranchCov condition than the MutationCov condition.

6 THREATS TO VALIDITY

We discuss possible threats to validity [13] and their mitigations where appropriate.

Internal validity. For all subjects, the interview session was their first encounter with mutation-based feedback. This possibly contributed to their challenges with designing test cases to kill certain mutants (§4.2.3). To manage this, each interview included an explanation of mutation analysis feedback followed by a "warm-up" problem on which participants demonstrated their understanding of the general idea. Students generally grasped the idea behind mutation analysis, but tended to be challenged by specific mutants.

Additionally, students were presented with testing feedback using a custom interface that was purpose-built for this research (Figure 1). It is unlikely that students' responses to feedback were affected by this interface. Branch coverage was presented in a form common to many IDEs (§3.1),

and since students were familiar with the criterion and had used it in their IDEs previously, we do not believe this to be a significant threat to the validity of our findings. On the other hand, we used a novel method for presenting mutation-based feedback. Most tools for mutation analysis provide primarily command-line interfaces, with little support for in-editor feedback. Feedback presentation in Muttle represents a step toward improved graphical representations for mutation-based test quality feedback.

It is improbable that students' challenges with mutation-based feedback were induced by the interface, for three reasons. First, we conducted pilot interviews to get initial feedback on the interface and to test-run our chosen testing tasks. No issues with the interface arose. Second, during the 12 interviews conducted for this study (not including the pilot interviews), participants were given warm-up problems to learn the interface. Further, they were able to detect some mutants while facing challenges with others, suggesting that challenges had more to do with the mutants themselves than the interface. Finally, since mutation-feedback in Muttle is presented as simple line-level diffs, we do not believe a user study would have been necessary or informative.

Ecological validity. In the NoFeedback condition, two participants (P3 and P4) mentioned that they were putting in more effort while testing because they were participating in a study about testing (e.g., P4 said "Honestly, if I were doing my homework, I would not do this, it's just for this"). While the number of tests produced may have been influenced by the presence of the interviewer, the participant still verbalized their thought process and provided data about how they ended up choosing those test cases, which was ultimately our objective in this study.

External validity. Like any study involving human subjects, our study suffers from threats to external validity. Our study involved participants in a specific context—students in their third programming course testing Python programs. Moreover, students self-selected into this study after announcements were made in person in classrooms. As mentioned in §3.2, students came from a range of expertise levels (based on their performance in the previous CS course). Additional studies involving students with different backgrounds using different programming languages would help test the generalizability of these results.

Content validity. Though we have a relatively small sample size, we have reason to believe that additional interviews would not have revealed new information (at least, not without other changes to the study like the student population or the problems studied). As can be seen in §4, all themes occurred for a number of participants, suggesting that our analyses were not revealing one-off occurrences. Incidents in §4 pertaining to a single participant occurred early in the interview process. That is, after a point, analysis of subsequent interviews only revealed themes we had already seen in previous interviews.

Face validity. We have provided illustrative quotes and incidents for the themes revealed in our analysis.

Conclusion validity. As discussed in Section 3.4, our interpretations of students' words and actions are inevitably entangled with our experiences and perspectives. We have attempted to mitigate this threat by grounding our analyses and interpretations in existing literature where possible, consistent with the Straussian strand of grounded theory [62].

7 DISCUSSION

7.1 Implications for pedagogy of software testing

There are numerous purposes for writing automated software tests, but two are prominent. First, to verify that a given program (or soon-to-be-written program) meets its requirements. Second, to capture the functionality of a program so that future regressions may be detected and avoided.

In educational contexts other than immersive software engineering education experiences, there is little opportunity to demonstrate the second purpose due to the short development timelines and unchanging requirements that are typical of pedagogic programming assignments. Therefore, in fundamental programming courses that teach testing, we tend to emphasize the first purpose.

In this context, we ought to consider the role that we want test adequacy criteria to play in testing instruction. In our interviews, we did not observe any students returning to the problem requirements after they started trying to address the test adequacy criterion (though 3 out of 12 students had already produced strong test suites based on the problem requirements; §4.3). That is, once any testing feedback appeared, satisfying the criterion became the student's "testing goal". They focused on maximizing execution coverage of the program (in the case of branch coverage) or distinguishing the program semantically from other programs (in the case of mutation analysis).

Indeed, this is how students are often encouraged to judge the quality of their own tests in many programming courses. They are incentivized to satisfy test adequacy criteria by having part of their assignment grade depend on the strength of their tests (e.g., [1, 11, 43, 61]). As such, they are implicitly encouraged to incorporate criteria like branch coverage or mutation analysis into their workflows. Importantly, these criteria are based on the *implementation at hand*, and not on the specification it purports to implement. This can perhaps lead to skewed motivations when composing test suites or test plans.

For example, in a study of students' perceptions of test suites, Bai et al. [7] found that code coverage was commonly rated as an important characteristic to be aware of when writing new unit tests. Aniche et al. [5] report that developers did not tend to consider the program description (documentation) while reasoning about test adequacy, preferring to reason about code coverage and their own source and test code. In our own observations, multiple students reported designing their tests based on predicted branch coverage, even when no such feedback was available. This tendency to think from an "implementation-first" perspective can appear even in the absence of an implementation. Doorn et al. studied the *test models* created by graduate students who were tasked with testing a single problem [20] (without an available implementation). They reported a common "development approach" in which students thought about their test plan in terms of how they would go about implementing their solutions. As a result, they applied common programming restrictions to their test models (e.g., reducing duplication), but most still failed to adequately cover the problem space.

If the goal of writing tests is to prevent future regressions, perhaps this focus on a given implementation is acceptable. But if the goal is to verify that a program (or a soon-to-be-written program) meets its requirements, then we would rather that students reason about test adequacy in terms of *problem coverage* as opposed to *program coverage*. This is a key idea behind property-based testing [10], which Wrenn et al. describe as a potential "Trojan horse" for introducing students to writing specifications [68].

Others have used feedback mechanisms that are not primarily driven by implementation-based test adequacy criteria like branch coverage or mutation analysis. Cordova et al. [14] explored *conceptual feedback* that tells the student about instructor-defined "fundamental testing concepts" (e.g., boundary values [14]) that their test suite fails to cover. In a quasi-experiment, they found that, students who were trained with conceptual feedback performed better at testing tasks than those who were trained with feedback based on line and branch coverage.

As another example, consider the *all-pairs approach* described in §2.2 [26, 30, 67], found to be a reliable measure of a test suite's fault-finding capability [25]. A key strength of the approach is that the test suite's thoroughness is evaluated against faulty implementations ("mutants") that represent *real bugs*, i.e., mistakes that students have actually made, either through a mis- or incomplete understanding of the problem or through programming errors. This is in contrast to the mutants

generated by common mutation operators, which may not be faulty in similarly interesting ways. Wrenn et al. [66] have explored this observation further and developed Examplar, a tool that allows students to explore a problem space through test cases, using "mutants" that have been created by course staff to represent "conceptually interesting corners" of problems [65].

We do not argue that we should do away entirely with testing feedback based on adequacy criteria. We must, however, make it much more explicit to students *where in their testing process* these adequacy criteria ought to be incorporated. Specifically, we should encourage a "requirements-first" testing approach, as opposed to an "implementation-first" approach. That is, we want to teach students how to decompose a problem into the requirements that a correct solution would satisfy, and then to write tests that target those requirements. After this process, gaps noted by implementation-based test adequacy criteria (i.e., gaps in branch or mutation coverage) can help further strengthen the test suite [3].

We explicitly delineate this argument from calls to teach TDD. An "implementation-first" testing mindset can still occur during TDD; the study by Doorn et al. [20] suggests that students are still likely to design tests with a particular implementation in mind, whether or not such an implementation exists yet.

7.2 Future Work

This study raises the following directions for future work.

Are some mutation operators (or subsets of mutation operators) less cognitively taxing than others? What influences this load? We observed that students found it particularly difficult to kill mutants that appeared on lines with branching logic (e.g., `if` conditions or conditionals in `for` loops). Reasoning about logical branches and mutants simultaneously seemed to overwhelm their working memory. It seems likely that some mutation operators are less cognitively taxing than others. For example, others have suggested that *deletion mutation operators* (operators that work by simply deleting program constructs instead of making smaller, more subtle changes) may be easier to reason about than comprehensive mutation analysis [16, 41]. As far as we are aware, such claims have not been empirically tested.

How does feedback presentation affect the utility of mutation analysis in educational or professional settings? While these can be useful, mutation analysis has not benefited from the kinds of out-of-the-box IDE support and interfaces that code coverage tools enjoy. Current tools for mutation analysis provide feedback as a written list of killed or surviving mutants, often printed to the standard output stream. Feedback in Muttle (Figure 1) just one possible improvement to this landscape. Further usability studies into different styles of presenting mutation-based feedback would be beneficial for both academics and practitioners.

Are patterns of code-level and task-level thinking associated with software testing success? We originally expected that when students were driven by mutation analysis, they would engage in more *task-level thinking* [8] than *code-level thinking*. We did not find support for this hypothesis. However, we observed that much of students' success with identifying mutant-killing inputs occurred when they were able to verbalize using a high-level description the difference between the mutant and the original program (i.e., a *task-level* plan).

What types of mutants tend to survive test suites that satisfy weaker test adequacy criteria? Much has been written about reducing the computational cost of mutation analysis. A popular approach is *selective mutation analysis*, which limits the number of mutants that are produced by strategically choosing mutation operators (e.g., [16, 41, 52, 60]). Qualitatively, we noticed that simply satisfying BranchCov resulted in also satisfying mutation operators that target control flow constructs. If we can empirically identify the types of mutants that tend to slip

through weaker test adequacy criteria, we can reduce both the computational and cognitive cost of mutation analysis by only generating those mutants. As educators, we can also use this information to better direct students' testing efforts.

ACKNOWLEDGMENTS

This work was supported in part by the Baker/Koob endowments at California Polytechnic State University. The authors are grateful to the anonymous peer reviewers at TOCE and ICER 2023, as well as Dr. J. Davis, Dr. A. Keen, and Dr. S. Beard for critiquing early drafts of this paper; and to Jon Lai for his initial engineering contributions.

REFERENCES

[1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. 2010. Mutation Analysis vs. Code Coverage in Automated Assessment of Students' Testing Skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 153–160. https://doi.org/10.1145/1869542.1869567

[2] P. Ammann and J. Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press. https://books.google.com/books?id=BMbaAAAAMAAJ

[3] Maurício Aniche. 2022. *Effective Software Testing: A Developer's Guide*. Manning, Shelter Island, NY.

[4] Maurício Aniche, Felienne Hermans, and Arie van Deursen. 2019. Pragmatic Software Testing Education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 414–420. https://doi.org/10.1145/3287324.3287461

[5] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2021. How Developers Engineer Test Cases: An Observational Study. *IEEE Transactions on Software Engineering* (2021). https://doi.org/10.1109/TSE.2021.3129889

[6] CS Learning 4 U Group at The University of Michigan. 2015. *16.9 — Rainfall Problem — AP CS Principles — Student Edition*. https://runestone.academy/ns/books/published/StudentCSP/CSPIntroData/rainfall.html Accessed on September 30, 2023.

[7] Gina R. Bai, Justin Smith, and Kathryn T. Stolee. 2021. 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* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 248–254. https://doi.org/10.1145/3430665.3456368

[8] F. E. V. Castro and K. Fisler. 2020. 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*. Association for Computing Machinery, New York, NY, USA, 487–493. https://doi.org/10.1145/3328778.3366847

[9] Y. Chun Tie, M. Birks, and K. Francis. 2019. Grounded theory research: A design framework for novice researchers. *SAGE Open Medicine* 7 (2019), 2050312118822927. https://doi.org/10.1177/2050312118822927 arXiv:https://doi.org/10.1177/2050312118822927 PMID: 30637106.

[10] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (sep 2000), 268–279. https://doi.org/10.1145/357766.351266

[11] Peter J. Clarke, Debra Davis, Tariq M. King, Jairo Pava, and Edward L. Jones. 2014. Integrating Testing into Software Engineering Courses Supported by a Collaborative Learning Environment. *ACM Trans. Comput. Educ.* 14, 3, Article 18 (oct 2014), 33 pages. https://doi.org/10.1145/2648787

[12] Peter J. Clarke, Debra L. Davis, Raymond Chang-Lau, and Tariq M. King. 2017. Impact of Using Tools in an Undergraduate Software Testing Course Supported by WReSTT. *ACM Trans. Comput. Educ.* 17, 4, Article 18 (aug 2017), 28 pages. https://doi.org/10.1145/3068324

[13] Thomas D Cook, Donald Thomas Campbell, and Arles Day. 1979. *Quasi-experimentation: Design & analysis issues for field settings*. Vol. 351. Houghton Mifflin Boston.

[14] Lucas Cordova, Jeffrey Carver, Noah Gershmel, and Gursimran Walia. 2021. A Comparison of Inquiry-Based Conceptual Feedback vs. Traditional Detailed Feedback Mechanisms in Software Testing Education: An Empirical Investigation. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 87–93. https://doi.org/10.1145/3408877.3432417

[15] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. 2002. The Roles Beacons Play in Comprehension for Novice and Expert Programmers.. In *PPIG*. 5.

[16] M. E. Delamaro, J. Offutt, and P. Ammann. 2014. Designing Deletion Mutation Operators. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 11–20. https://doi.org/10.1109/ICST.2014.12

[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[18] A. Derezińska and K. Hałas. 2014. 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.). Springer International Publishing, Cham, 155–164.

[19] Niels Doorn, Tanja Vos, Beatriz Marín, and Erik Barendsen. 2023. Set the right example when teaching programming: Test Informed Learning with Examples (TILE). In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 269–280. https://doi.org/10.1109/ICST57152.2023.00033

[20] Niels Doorn, Tanja E. J. Vos, Beatriz Marín, Harrie Passier, Lex Bijlsma, and Silvio Cacace. 2021. Exploring students' sensemaking of test case design. An initial study. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 1069–1078. https://doi.org/10.1109/QRS-C55045.2021.00161

[21] Bob Edmison and Stephen H. Edwards. 2019. Experiences Using Heat Maps to Help Students Find Their Bugs: Problems and Solutions. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 260–266. https://doi.org/10.1145/3287324.3287474

[22] Bob Edmison and Stephen H. Edwards. 2020. Turn up the Heat! Using Heat Maps to Visualize Suspicious Code to Help Students Successfully Complete Programming Problems Faster. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* (Seoul, South Korea) *(ICSE-SEET '20)*. Association for Computing Machinery, New York, NY, USA, 34–44. https://doi.org/10.1145/3377814.3381707

[23] Stephen H. Edwards. 2004. Using software testing to move students from trial-and-error to reflection-in-action. *ACM SIGCSE Bulletin* 36, 1 (March 2004), 26. https://doi.org/10.1145/1028174.971312

[24] Stephen H. Edwards and Zalia Shams. 2014. Comparing Test Quality Measures for Assessing Student-written Tests. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 354–363. https://doi.org/10.1145/2591062.2591164

[25] Stephen H. Edwards and Zalia Shams. 2014. Comparing Test Quality Measures for Assessing Student-Written Tests. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 354–363. https://doi.org/10.1145/2591062.2591164

[26] Stephen H. Edwards, Zalia Shams, Michael Cogswell, and Robert C. Senkbeil. 2012. Running Students' Software Tests against Each Others' Code: New Life for an Old "Gimmick". In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 221–226. https://doi.org/10.1145/2157136.2157202

[27] Eduard Enoiu and Robert Feldt. 2021. 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)*. 123–124. https://doi.org/10.1109/CHASE52884.2021.00026

[28] Eduard Enoiu, Gerald Tukseferi, and Robert Feldt. 2020. 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)*. 272–279. https://doi.org/10.1109/QRS-C51114.2020.00053

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

[30] Michael H. Goldwasser. 2002. A Gimmick to Integrate Software Testing throughout the Curriculum. *SIGCSE Bull.* 34, 1 (Feb 2002), 271–275. https://doi.org/10.1145/563517.563446

[31] John B. Goodenough and Susan L. Gerhart. 1975. Toward a theory of test data selection. *IEEE Transactions on Software Engineering* SE-1, 2 (1975), 156–173. https://doi.org/10.1109/TSE.1975.6312836

[32] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How Close are they to Real Faults?. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 189–200. https://doi.org/10.1109/ISSRE.2014.40

[33] Braxton Hall and Elisa Baniassad. 2022. Evaluating the Quality of Student-Written Software Tests with Curated Mutation Analysis. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E* (Auckland, New Zealand) *(SPLASH-E 2022)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/3563767.3568132

[34] Hadi Hemmati. 2015. How Effective Are Code Coverage Criteria?. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. 151–156. https://doi.org/10.1109/QRS.2015.30

[35] Joy W. Hollén and Patrick S. Zacarias. 2013. *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.

[36] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

[37] David Janzen and Hossein Saiedian. 2008. Test-Driven Learning in Early Programming Courses. *SIGCSE Bull.* 40, 1 (mar 2008), 532–536. https://doi.org/10.1145/1352322.1352315

[38] Edward L. Jones. 2000. Software Testing in the Computer Science Curriculum – a Holistic Approach. In *Proceedings of the Australasian Conference on Computing Education (ACSE '00)*. ACM, New York, NY, USA, 153–157. https://doi.org/10.1145/359369.359392

[39] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[40] Stuart A. Karabenick. 2004. Perceived Achievement Goal Structure and College Student Help Seeking. *Journal of Educational Psychology* 96, 3 (2004), 569–581. https://doi.org/10.1037/0022-0663.96.3.569

[41] Ayaan M. Kazerouni, James C. Davis, Arinjoy Basak, Clifford A. Shaffer, Francisco Servant, and Stephen H. Edwards. 2021. Fast and accurate incremental feedback for students' software tests using selective mutation analysis. *Journal of Systems and Software* 175 (2021), 110905. https://doi.org/10.1016/j.jss.2021.110905

[42] Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. 2017. Quantifying Incremental Development Practices and Their Relationship to Procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 191–199. https://doi.org/10.1145/3105726.3106180

[43] Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. 2019. Assessing Incremental Testing Practices and Their Impact on Project Outcomes. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 407–413. https://doi.org/10.1145/3287324.3287366

[44] K. N. King and A. Jefferson Offutt. 1991. A fortran language system for mutation-based software testing. *Journal of Software: Practice and Experience* 21, 7 (1991), 685–718. https://doi.org/10.1002/spe.4380210704 arXiv:https://doi.org/10.1002/spe.4380210704

[45] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, and Nicos Malevris. 2016. 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)*. 147–156. https://doi.org/10.1109/SCAM.2016.28

[46] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. 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)*. 560–564. https://doi.org/10.1109/SANER.2015.7081877

[47] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. 2004. pytest 6.2.2. https://github.com/pytest-dev/pytest

[48] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* 40, 1 (2014), 23–42. https://doi.org/10.1109/TSE.2013.44

[49] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. 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* (Canterbury, UK) *(ITiCSE-WGR '01)*. Association for Computing Machinery, New York, NY, USA, 125–180. https://doi.org/10.1145/572133.572137

[50] Justin Middleton and Kathryn T. Stolee. 2022. Understanding Similar Code through Comparative Comprehension. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–11. https://doi.org/10.1109/VL/HCC53370.2022.9833117

[51] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2012. *The art of software testing* (3rd ed ed.). John Wiley & Sons, Hoboken and N.J.

[52] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (April 1996), 99–118. https://doi.org/10.1145/227607.227610

[53] A. J. Offutt and J. M. Voas. 1996. Subsumption of Condition Coverage Techniques by Mutation Testing.

[54] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295–341. https://doi.org/10.1016/0010-0285(87)90007-7

[55] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does Mutation Testing Improve Testing Practices?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 910–921. https://doi.org/10.1109/ICSE43902.2021.00087

[56] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. In *Proceedings*

of the 50th ACM Technical Symposium on Computer Science Education (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 531–537. https://doi.org/10.1145/3287324.3287374

[57] J. Sajaniemi. 2002. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. 37–39. https://doi.org/10.1109/HCC.2002.1046340

[58] Jorma Sajaniemi and Marja Kuittinen. 2005. An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education* 15, 1 (2005), 59–82. https://doi.org/10.1080/08993400500056563 arXiv:https://doi.org/10.1080/08993400500056563

[59] Lilian Passos Scatalon, Jeffrey C. Carver, Rogério Eduardo Garcia, and Ellen Francine Barbosa. 2019. Software Testing in Introductory Programming Courses: A Systematic Mapping Study. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 421–427. https://doi.org/10.1145/3287324.3287384

[60] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. 2008. Sufficient Mutation Operators for Measuring Test Effectiveness. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 351–360. https://doi.org/10.1145/1368088.1368136

[61] Jaime Spacco and William Pugh. 2006. Helping Students Appreciate Test-Driven Development (TDD). In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 907–913. https://doi.org/10.1145/1176617.1176743

[62] Anselm Strauss and Juliet M Corbin. 1997. *Grounded theory in practice*. Sage.

[63] Dávid Tengeri, László Vidács, Árpád Beszédes, Judit Jász, Gergõ Balogh, Béla Vancsics, and Tibor Gyimóthy. 2016. 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)*. 174–179. https://doi.org/10.1109/ICSTW.2016.25

[64] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2023. A Think-Aloud Study of Novice Debugging. *ACM Trans. Comput. Educ.* 23, 2, Article 28 (jun 2023), 38 pages. https://doi.org/10.1145/3589004

[65] John Wrenn. 2022. *Executable Examples: Empowering Students to Hone Their Problem Comprehension*. Ph. D. Dissertation. Brown University.

[66] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 131–139. https://doi.org/10.1145/3291279.3339416

[67] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*. ACM, New York, NY, USA, 51–59. https://doi.org/10.1145/3230977.3230999

[68] John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2020. Using Relational Problems to Teach Property-Based Testing. *The Art, Science, and Engineering of Programming* 5, 2 (oct 2020). https://doi.org/10.22152/programming-journal.org/2021/5/9

A PROBLEMS USED IN TESTING TASKS

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

Listing 1. **Multiply**. A function that multiplies two provided numbers.

```
1  def larger(first, second):
2    if first == second:
3      return first
4
5    return max(first, second)
```

Listing 2. **Larger**. Given two numbers, return the larger of the two. If the numbers are equal, return the first number.

```
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
```

Listing 3. **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).

```
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
```

Listing 4. **Selection Sort**. Sort the given list of numbers using the Selection Sort algorithm.

```
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

```

```
17  return rain_total / days
```

Listing 5. **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 integers (and only the positive integers), (b) count the number of positive integers (we will count with "1.0" so that our average can have a decimal point), and (c) 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 [6].

```
1  def centered_average(nums):
2    min_idx = 0
3    max_idx = 0
4
5    for idx in range(len(nums)):
6      if nums[idx] <= nums[min_idx]:
7        min_idx = idx
8      elif nums[idx] >= nums[max_idx]:
9        max_idx = idx
10
11   nums[min_idx] = 0
12   nums[max_idx] = 0
13
14   sum = 0
15   for num in nums:
16     sum += num
17
18   return sum / (len(nums) - 2)
```

Listing 6. **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.

B QUALITATIVE CODING

Table 2. Codes from the open coding process, grouped by the category under which they appeared.

#	Code/description	Frequency	Illustrative quote or incident
	§4.1 Problem and program comprehension		
1	Start by reading the problem description	4	[In Triangle] *First thing I would test is just kind of putting in three random numbers. Say how about...3, 4, 5, that should be a nice scalene, which is a 3...I'm currently just testing based off of the description. Thinking about this is what the code ideally should be doing.* (P7)
2	Start by reading the provided program	2	[In Rainfall] *Here I was just kind of walking through and seeing the different if statements.* (P9)

3	Difficulty with code comprehension complicates testing	1	[In Rainfall] `continue` *, so I don't think I've ever used this, but does this mean it just continues to the next line of code?* (The student does not recognize the `continue` keyword.) (P2)
4	Refer to code to address gap in understanding from description	6	[In Triangle] *And then returning 3. [3 2 1]. Ohh,* [the system] *expected 0. I guess that's not a thing, that must be there* [indicates first `if` condition]. *OK, well, that is good to know. [3 2 1], cool.* (P3)
5	Refer to description to address gap in understanding from code	9	[In Rainfall, on reading the description] *Because that way it's easier to understand what the code is doing, especially since it's not commented.* (P6)
6	Code tracing in basic blocks	7	[In Centered Average] *I need the current element in the list to be greater than the current maximum index.* [Participant is conducting tracing assuming they have slipped into the true clause of an `if` condition.] *With code coverage just tried to hit it like just hit the line, get out of the way, but then for bugs I kind of have to think about it. I probably like run it through and then try and see what's wrong and then yeah base my code off that.* (P2)
7	Hesitation to begin testing with shaky understanding of the problem	1	[In Triangle] Participant did not start testing till the problem was explained; sketched out potential test cases on paper to help with understanding. (P3)
8	Test result aids in problem understanding	3	[In Rainfall, runs the test and sees that the output did not match the expected value] *Expected 2...Wait. Am I missing something? It's 6...Oh, but is 0 positive number?* (P5)
	§4.2.1 Testing with No Feedback		
9	Write a basic, easy-to-reason about test	12	[In Centered Average] *So I'll just do like a simple list first, 1...2...3* (P6)
10	Run tests to get initial feedback	5	—
11	Write a test based on intuition about edge cases	12	[In Triangle] *Let's do a list of only negative numbers* (P5)
12	Want to put the code through the most paces	4	[Giving Selection Sort a reverse-sorted list of numbers.] *That puts the code through the most paces, I think.* (P1)
13	Write a test based on a prominent requirement in the description	7	[In Rainfall] *[The equipment] makes mistakes, reports negative amounts. So I'll try doing negatives, so I'll pass by that.* (P3)
14	Write a test based on a prominent feature in the code	5	[In Rainfall] *I first, I guess, for why-notsies I'll...I'm first gonna write something just that breaks this* [`rain_day == 99999`] *just because I want to make sure that this runs* (P3)

15	Detailed program description induced student to plan tests based on description	2	[In Rainfall] *Also...this was a slightly longer and more complex program...like my first thought was that's a wall of text...although it's really not all that complicated in terms of what it does, it lists out all the things that it should do pretty nicely, which that's helpful. Because that gives me an idea of what should the program be doing.* (P7)
16	Prefer to test based on problem understanding over source code	3	[In Rainfall] *Then when I write the tests I read the prompt and again and I see different things.* (Participant went as far as to scroll the source code off-screen while writing tests.) (P5)
17	Test plan formed around predicted code coverage	5	[In Triangle, before any feedback has been displayed] *So I guess I'll just write tests for all these three* [Participant hovers mouse over lines 2–4 in Triangle] *and...OK, so just going off like the if statements or yeah if statement.* (P4)
18	Prior experience with BranchCov leads participant to think in terms of BranchCov with NoFeedback	5	[In Rainfall] *Like I feel like ever since [recent CS course], like we have to have full coverage all the time, I kind of just feel like I have to look everything so everything runs.* (P3)
19	Read program line-by-line and write tests for each statement	8	[In Triangle] *But like with this, especially like with this setup, I can look at my code and write tests at the same time so I can just like go line by line and like look at each part of like each line and make sure that my code is like hitting every section* (P6)
	§4.2.2 Testing with Branch Coverage		
20	Write a test targeting a particular logical branch	4	[In Centered Average] *So if this current index is greater than equal to the minimum index. Then reset the minimum index to the...that index.* (P2)
21	Domain knowledge plays a role when designing tests	6	[In Triangle] *Ooh, that would be weird though, because right if it's this is a big big bigger side and they have to say a second biggest side or the medium side, then technically they're only a triangle. Is it technically a triangle when they're equaling or when it's an angle of zero degrees, right? Because that triangle is saying it, so I would probably say I mean I would say no, because I think you're right, so I'm just going to go with that.* (P5)
22	Ignore code coverage and test based on problem description	2	[In Triangle] *Trying to remember what isoceles is. Also, based on the definition. I'm looking at the code, but I'm trying to mostly do it based on like what this [problem description] means.* (P3)
23	Ignore code coverage and test based on edge cases	3	[In Centered Average] *Definitely code coverage is like something I try to catch after I've written test cases.* (P7)

24	Trace code to understand coverage gaps	4	[In Centered Average] *So if this current index is greater than equal to the minimum index. Then reset the minimum index to the, that index.* (P2)
25	Use a variable role to understand a gap in coverage	7	[In Centered Average] `min_index = 1` *I guess we just write a test where everything is less than one.* (Note that the variable's role was misunderstood here: `min_index` was holding the index of the minimum value, not the minimum value itself.) (P1)
26	Test further after code coverage is satisfied	4	[In Selection Sort] *OK, cool also, I just thought of this, maybe like it's just a list of more than one of the same number.* (P2)
	§4.2.3 Testing with Mutation Coverage		
27	Create a test to kill a mutant without tracing	4	[In Rainfall]*OK, so I guess that has to do with...1 1 1 99999 1 1. So that should return 1. I guess in both cases.* [changes input to `[1 1 1 99999 2 1]`] *That should kill that bug.* (P1)
28	Trace the mutant program to understand it	7	[In Rainfall, reading a displayed mutant] *So if it was...* `rain_day` *is the thing, the element of the list. So if* `rain_day` *is greater than 0, it should run. But if it's greater than 1, it wouldn't. Yeah, I'm trying to like run through what the code is doing I guess.* (P2)
29	Ignore mutants and test based on edge cases	2	[In Rainfall, ignoring displayed mutation feedback] *I think there would be something suspicious that would happen if it was just a day with zero rainfall.* (P1)
30	Hunch that achieving code coverage for a line will also kill the mutants on that line	9	[In Rainfall] *So I was wondering [about] this bug and I realized that I hadn't tested 1, even though I said it was a special number you know, in my opinion. So I was like, oh yeah, I should just try it.* (P2)
31	Direct testing efforts based on presence of mutants	2	[In Triangle] *OK, I think they just want me to check for each side. This* [Hovers over bug badges appearing over an `if` statement] *basically just means I think that like it doesn't really matter like what's coming out of here, so I should probably like actually test something that will check [that condition].* (P6)
32	Difficulty understanding how a killable mutant differs from the original program	5	[In Rainfall] *So like given this [for loop], so it's running through, everything's running through one, it's running through 2, running through 3, so on and so forth, and then it, then it's going to...You know, I'm not entirely sure how to squash this bug.* [Interviewer proceeds to ask leading questions to help the participant progress.] (P10)

33	Test further after mutation coverage is satisfied	2	[In Rainfall] *I think I'd also probably want to test with some negative numbers in there.* [Interviewer: Why is that?] *Just because it was part of the the original prompt. So I wanna make sure I guess.* (P1)
		§4.3 Other patterns	
34	Write more tests because of a "feeling" that they are needed	2	[In Centered Average] *Ok, I think I'm not doing like, like a bigger list. I should do a bigger list....Let's see, 1, 0, 1, same numbers here. Just in case, I'm gonna put like big numbers.* (P4)
35	Testing will not be typical if the participant is told the program is correct beforehand	1	[In Centered Average] *Well, I know that your code works because you told me that, so I'm just writing whatever test I can think of.* (P4)
36	Reflect on completeness of tests	1	[In Centered Average] *OK, so now I want to do some of my weird checks. For no reason but just to make sure.* (P1)
37	Write more tests because this is a study about testing	2	[In Centered Average] *Honestly, if I were doing my homework, I would not do this, it's just for this. I'm just going to be extra cautious and do that.* [Adds more tests in the NoFeedback condition] (P4)
38	Write a test by changing an existing one	10	[In Triangle] The student used `[10, 1, 2]` to trigger the condition `side1 >= side2 + side3`, then re-ordered those values for the two other conditions. (P4)
39	Design an input for a particular statement in the code	2	[In Centered Average] *So if this current index is greater than equal to the minimum index. Then reset the minimum index to the, that index.* (P2)