ABSTRACT

BAI, RUI. Improving Student Testing Practices through a Lightweight Checklist Intervention. (Under the direction of Dr. Kathryn Stolee).

Software testing is widely practiced in industry for preventing and catching regression faults in software development and maintenance, and hence is an essential skill for both professional software developers and graduating computer science students. Educators have sought to establish and enhance students' testing skills, including integrating testing in introductory level courses and introducing tools to support students learning testing.

Yet, students still face challenges when they are learning testing. For example, students need to learn the new concepts, new syntax, new tools, and new libraries. This often requires students to search and use online resources, in which they also encounter challenges. Additionally, students often run into trouble when they are practicing testing, such as identifying what code or scenarios to check and determining how to test them. Understanding the challenges that students face when learning and performing testing can help inform practices for software testing education.

This dissertation identifies the barriers students face during software development, with a specific focus on software testing, and presents an intervention that provides testing support without a steep learning curve.

First, we study the strategies that students adopt and the barriers that students face when searching and using online resources during program implementation. Next, we explore how students perceive and perform unit testing via a case study, as well as how they interact with given unit tests. Motivated by the observed barriers, we design and assess a lightweight testing checklist that bridges the gap between program requirements and source code. Finally, we integrate the explicit testing strategies to the checklist, and investigate their values in testing education. We demonstrate that the checklist intervention improves student testing practices. Collectively, the research efforts detailed in this dissertation help identify barriers students have during software engineering activities, in particular software testing, and address them with a lightweight checklist intervention.

Improving Student Testing Practices through a Lightweight Checklist Intervention

by
Rui Bai

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2022

APPROVED BY:

Dr. Sarah Heckman

Dr. Thomas Price

Dr. Shiyan Jiang

Dr. Susan Fisk
External Member

Dr. Kathryn Stolee
Chair of Advisory Committee

DEDICATION

To my uncle Lei Mei.

ACKNOWLEDGEMENTS

Throughout my journey of doctoral studies, I have been supported by many phenomenal individuals.

I would like to thank my advisor, Dr. Kathryn Stolee, who introduced me to software engineering, guided me through this long journey, and inspired me to reach higher than my goals. To be honest, it is hard for me to precisely describe how much your patience, encouragement, and support mean to me in English.

I am very grateful to my committee members for their time and constructive feedback on my dissertation. To Dr. Sarah Heckman, for believing in me as an instructor and providing valuable guidance on my teaching and research. To Dr. Thomas Price, for expanding my research knowledge in computer science education. To Dr. Susan Fisk, for sharing your sociology expertise in our interdisciplinary collaboration. To Dr. Shiyan Jiang, for lending the perspectives from STEM education.

None of these works would be possible without my research colleagues and academic friends. Thank you to Kai Presler-Marshall, Nischal Shrestha, Justin Middleton, George Mathew, Justin Smith, Peipei Wang, Dr. Westley Weimer, and Yu Huang for supporting me with your expertise, experience, and most importantly, friendship.

I also want to acknowledge Dr. Laura Bottomley, for piquing my interest in teaching, and Dr. Barbara Adams, for helping me develop towards a professor. I am also very appreciative of other faculty and staff at NC State, including Ms. Leslie Rand-Pickett, CSC Graduate Program Office, and CSC IT Help Desk.

Above all, I would not have been able to reach this moment without the unconditional love and support of my family - Mom, Dad, cousin Liang, uncle Lei, and my grandparents. I am so fortunate to always have you by my side. I love you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1 INTRODUCTION

Software testing is crucial to help developers detect and fix bugs in software systems. With the availability and pervasive adoption of convenient testing frameworks (e.g., JUnit) and agile development methodologies (e.g. Extreme Programming and Scrum), writing unit test cases is an increasingly common practice in industry [70, 136], and hence an emerging topic in computer science education. Educators have sought to establish and enhance students' testing skills in recent years. For instance, educators have integrated testing into CS1 courses [66, 74, 82, 107, 164], proposed postgraduate level courses focused on testing [118], and introduced tools to support students as they learn and practice testing [24, 52, 164]. The topic of software testing in education is also capturing attention from researchers, with the first Software Testing Education Workshop being held in 2020.[1] This demonstrates the growing recognition of the importance of software testing in education.

Despite considerable effort made by educators and researchers, students face challenges when learning testing [28, 88, 135]. To master testing, students need to learn the new concepts, new syntax, new tools, and new libraries. This often requires students to search and use online resources, such as code examples and web-based tools; however, how to

[1] TestEd 2020 - https://testedworkshop.github.io/

search, select, and use these online resources can be challenging as well. Besides, students often run into trouble when they are practicing testing. Educators have observed that, when practicing testing, students make mistakes such as missing boundary testing [4, 20], writing smelly tests [29], testing happy paths only [20], and not testing the program until they are already done with development [135]. Understanding how students test, how they search and use online resources to assist testing, and the challenges they encounter during the processes can motivate future tools and educational practices to better support students.

1.1 Motivating Example

To further understand students' testing practices, we will use Caroline's experiences. Caroline is a hypothetical graduate student who is expected to test a DevOps course project in Java, but was not formally exposed to testing concepts and techniques in her undergraduate program. Caroline then decides to search online for tutorials and example tests. After learning the syntax for JUnit, she implements several unit tests that assert the expected program behaviors. However, one of the tests fails, which raises the following questions: "*Does the failure indicate a bug in the source code? Or in my testing code?*", "*Does the test precisely reflect the intended testing scenario?*", "*Are the program specifications correctly interpreted?*" Caroline then checks the test case along with the program specifications, and finds the test code is buggy. However, she does not know how to fix the test, and hence decides to delete this test instead. Now all tests pass, and Caroline starts to wonder "*I have found the bug, can I stop testing now? Is this program tested enough with these unit tests? What else should I check?*" She searches again for the measurements of test suite quality, and she finds the tool EclEmma that provides code coverage analysis. However, she finds it is challenging to learn a new tool, comprehend the coverage reports, and figure out the tests that could improve the coverage metrics.

This dissertation responds to students' challenges in testing by identifying their needs for information and tool support, as well as the barriers they have when practicing testing. We study students' software engineering activities outside the classroom, including learning a API (e.g., JUnit), searching and using online resources (e.g., code example and web-based tools), and performing closed-box and clear-box testing. These investigations lead us to the thesis statement of this dissertation:

1.2 Thesis Statement

Motivated by the identification of barriers students have during software engineering activities, in particular software testing, we show that a lightweight checklist intervention improves student testing practices.

1.3 Research Questions

To support this thesis statement, we structure this dissertation around the following three high-level research questions:

1. **RQ1: What barriers do students have when searching and using online resources?** Students often search for online resources to facilitate programming tasks and formulate the unit tests. Understanding the strategies that students adopt and the barriers that students encounter when searching and using online resources allows us to better support them. To do so, we conduct two case studies to explore students' search behaviors, exploring questions such as why and how they search, how they use the online resources, how they generate and modify the search queries, as well as the challenges they encountered. Chapter 3 details these two studies.

2. **RQ2: What barriers do students have when learning and practicing testing?** Testing is an essential software development activity and an important topic in CS education. Understanding the barriers students face when learning and practicing testing can help inform educational practices. To answer this research question, we conduct an exploratory study focusing on students' testing performance, including perceptions, practices, and pitfalls, as well as an experiment focusing on students' interactions with provided unit tests. Chapter 4 details these two studies.

3. **RQ3: How to improve students' testing practices with checklists?** From RQ1 and RQ2 we learned the barriers students encounter, as well as the information and the tool support that students need during software engineering activities. We answer this research question by responding to the findings from previous questions. We design and evaluate two novel approaches, specifically, checklist intervention and explicit testing strategies, via empirical studies. Chapter 5 details these two studies.

1.4 Contributions

In this dissertation, we identify barriers that students have during software engineering activities, and introduce novel approaches to improve student testing practices. Our contributions include:

1. Insights on students' needs for tool support and code examples when developing (Sections 3.1 and 3.2) and testing software (Sections 4.1 and 5.1).

2. Better understanding of students' code search activities and strategies, as well as the similarities and differences of search behaviors between students and professionals (Sections 3.1 and 3.2).

3. A comprehensive investigation of students' testing activities, which revealed students' perceptions of and the challenges in unit testing (Section 4.1).

4. An empirical evaluation of the impact of the test smell *Assertion Roulette* from the perspective of testing education (Section 4.2).

5. A lightweight testing checklist containing both tutorial information and testing strategies to assist students with testing practices, which is a novel form of tool support in testing education (Section 5.1).

6. The first exploration of the adoption of explicit strategies in testing education (Section 5.2).

1.5 Outline of Dissertation

The rest of this dissertation begins with the background information on current testing educational practices and the research areas focusing on students' software engineering activities in Chapter 2. Chapter 3 details the empirical investigation of the barriers that students have when searching and using online resources. Chapter 4 discusses students' perceptions of testing and the challenges they encounter when practicing testing. In Chapter 5, we present and assess new approaches to improve testing education. Chapter 6 revisits the thesis statement, and concludes the implications of this dissertation and the future work. Finally, the appendix includes study materials for the research presented in this dissertation.

CHAPTER

2

BACKGROUND AND RELATED WORK

This chapter provides background information and related studies on current testing educational practices (Section 2.1), students' software engineering activities such as code searching and programming (Section 2.2), as well as the application of checklists (Section 2.3), which are key for the research presented in this dissertation.

2.1 Testing Education

Software testing has many benefits in education. It enables students to produce cleaner code "faster than ever before" by catching bugs early in development [182]. Studies also report that when students write their own tests, they write better code [103, 154]. Wick et al. [182] suggest that unit testing helps group projects by demonstrating the correctness of the code to group members.

Educators have experimented with diverse approaches to teaching testing, such as requiring students to turn in tests along with their solutions [51, 66, 74], asking students to perform closed-box testing on a program seeded with errors [79], and instructing students to conduct peer testing [64].

Most testing education work focuses on CS1 and sometimes CS2 (e.g., [50, 64, 74, 138, 154, 185]). Fewer papers have focused on upper-level or SE-style courses (e.g., [23, 30, 89, 138]), and these were focused on specific aspects of testing (e.g. peer review, procrastination). Recent prior work has focused on teaching students about testing concepts by providing tutorials and other instructional materials during testing [4, 38].

This dissertation explores students' testing processes, specifically the upper-level undergraduate or graduate students, and suggests the use of various testing concepts and provide tutorial information related to the use of the JUnit library during testing.

2.1.1 Student-related Challenges in Testing Education

In this section, we discuss four student-related challenges in testing education that are categorized in prior work [63] (i.e., low motivation (Section 2.1.1.1) and increased cognitive load (Section 2.1.1.2)), as well as observed in our own study in Section 4.1 (i.e., misconceptions (Section 2.1.1.3) and learning barriers (Section 2.1.1.4)).

2.1.1.1 Low Motivation

Many students perceive software testing as "uninteresting", "tedious", "unnecessary" and "irrelevant" due to the course-level small-scale projects and exercises [20, 135, 139, 153]. To increase interest, motivation, and confidence in software testing among students, educators also introduced cross-course activities [5, 139], in which students in software testing courses wrote tests for peers who are taking courses like software development, data structures and algorithms.

2.1.1.2 Increased Cognitive Load

To perform (automated) testing, students are expected to learn new concepts, new syntax, new tools and libraries. This is particularly challenging to beginners as they might still struggle with basic programming concepts [28, 88, 172]. The increased cognitive load also contributes to students' negative attitudes towards testing [28, 88, 135, 153].

2.1.1.3 Misconceptions

Many students view testing as debugging [20, 139]. Some students stop testing after designing a test for a presumed bug in the code [20]. Sometimes they might modify the failing tests by simply changing test cases to make the test pass, so the incorrect software appears to be correct [139]. Praphamontripong et al. [139] report that students believe the only necessary skill to perform testing is knowing how to print debugging information, while Heckman et al. [74] found that students "rarely saw how test failures can help find bugs in [an] implementation".

2.1.1.4 Learning Barriers & Code Examples

Learning barriers in programming systems can arise from the environment and accompanying libraries [95]. Ko et al. [95] observed that learners sometimes knew what set of interfaces could achieve the behavior, but did not know how to use and coordinate them. The *use* and *coordination barriers* are hard to overcome without well-written documentation and code examples.

Prior studies [127, 147, 158] point out that it is essential for API documentation to provide developers, especially those who are trying to learn a particular API, with sufficient and adequate code examples. Parnin and Treude [127] observed that 90% of code-related blog posts contain code examples and snippets, with a median of three code snippets per post. McLellan et al. [113] also found that the code examples supported several different learning activities, such as understanding the libraries, protocols, and usage context.

To better help students overcome these challenges, this dissertation investigates how students perceive and perform unit testing.

2.1.2 Measurements of Test Code Quality in Testing Education

Test code quality is usually evaluated by completeness using code coverage metrics [2, 42, 98, 124], effectiveness using mutation coverage [70, 179] and maintainability using test code smells [4, 167]. Requirements coverage, which is concerned with structural coverage on software requirements, is less often used as there is a lack of automated tools to map requirements to code [67].

2.1.2.1 Completeness & Effectiveness

In education, the quality of student-authored test code is usually measured by code coverage, specifically statement coverage and branch coverage [49, 74]. Code coverage analysis tools such as EclEmma [48] can be used with automated testing (e.g., JUnit tests in Java) to generate coverage reports for users during development. However, prior studies on industry code have shown that code coverage has no correlation with test suite effectiveness [78], and it can be gamed by writing poor tests that execute code but lack effective assertions [74]. To measure test suite effectiveness, some educators [49, 156] have adopted mutation testing and all-pairs testing of student tests against other students' code to identify if a test suite has bug-revealing capability.

2.1.2.2 Maintainability

Assessing maintainability has been done through test smells, which may cause inaccurate test results as programs evolve and impact maintainability of unit tests [17, 123, 165, 167]. However, this usually requires manual inspection, and hence is not commonly adopted in testing education at scale.

Test smells have been actively discussed among practitioners and researchers [47, 116, 146]. Various studies have considered smell prevention [83, 188], smell detection [120, 130, 131, 178], and smell correction [21, 71], though none have focused on student-authored test code.

Bavota et al. [17] analyze the distribution of test smells and their impact on software maintenance in industrial and open-sourced systems. Ma'ayan reports that, with JUnit testing, 61% of all test methods contain more than a single assertion; meanwhile, 78% of all `assertTrue` and `assertFalse` assertions do not contain a customized error message [106], which can make traceability more difficult. Garousi and colleague [62] conducted a literature mapping on both the scientific literature and practitioners' grey literature, and they classified the test smells.

In this dissertation, we measure student-authored test quality from three aspects: completeness (i.e., requirements coverage and code coverage), 2) effectiveness (i.e., mutation coverage and the number of identified bugs), and 3) maintainability (i.e., test smells).

2.1.3 Measurements of Students' Testing Activities

Students' testing behaviors are commonly assessed by how many times [8] and how often they test the code [89], as well as when during development students usually test their code [89, 135]. In this dissertation, we investigate students' testing behaviors by studying the number of test suite executions, how early students start testing, and how often they test their code during development.

2.1.4 Professionals' Perceptions and Practices in Software Testing

Beyond surveying students about testing challenges [4], researchers have surveyed professional developers about their testing practices [33, 42, 121, 149] and the level of their related education and training [33, 121].

Ng et al. [121] surveyed 65 organizations and report that 64.6% of them have used at least one structured software testing methodology in the past three years. The three most popular testing methodologies were test case selection, static analysis, and dynamic analysis. Closed-box testing, which focuses on the external behavior of the software and requires no knowledge of internal functioning, was more commonly adopted than clear-box testing, in which the tester checks the internal implementation of the software. They also suggest that the major problem with using testing methodologies was the lack of expertise, and that cost was a major barrier to the use of automated software testing tools. These findings match the results from Chan and colleagues [33] on 34 organizations. Both studies demanded software testing courses in university programs and professional development training.

Daka et al. [42] surveyed 225 software developers on their unit testing practices, covering different programming languages and 29 countries. Their survey has four question types: rating, ranking, selection, and distribution questions. The authors conclude that developers write unit tests to manage requirements, and that unit tests are perceived as helpful. Participants usually spent less time on writing new tests than writing and fixing code. Developers also seek assistance with identifying what to test and how to produce robust tests. Runeson [149] also suggests that developers have issues determining if a module has been sufficiently tested.

In this dissertation, we adapt survey questions from prior work [42]and investigate how the students perceive unit testing, including the goal and the important aspects of unit testing, and the processes that they find challenging.

2.2 Students' Software Engineering Activities

In this section, we focus on related work in both students' and professional developers' code search activities (Section 2.2.1), measurements of students' programming activities (Section 2.2.2), as well as the factors that impact students' self-assessed CS ability and their persistence intentions in CS (Section 2.2.3).

2.2.1 Code Search

Developers frequently turning to code search to find code examples to learn from [150] and improve their productivity during development activities [16, 150, 177]. Various studies focusing specifically on code search have been performed by surveying participants about the reasons why they searched [77, 150, 159, 187], the tools they searched with [160], the popular search sites [169], as well as their selection criteria for code [96]. Prior work has adopted different approaches for studying how developers search for code. Some studies focused on directly observing participants' behaviors [161], while some other studies focused on collecting and analyzing search logs, such as work done by Brandt et al. [25, 26], with participant solving the pre-selected tasks. Sadowski and colleagues [150] combine surveys and logs to analyze developers' code search behavior when working on daily tasks.

Rahman et al. find that code search tasks take more effort than information search tasks [142]. Tsai and colleagues [175] yield similar results that students perceived study-oriented topics as harder to search for than daily life information.

Automatic API recommendation and query reformulations have also been studied. Rahman et al. propose a novel query reformulation technique, which can translate a natural language code search query into a ranked list of relevant Java APIs [143, 144]. Various strategies for reformulation have been adopted by researchers, such as a small edit distance [150] or a potentially large edit distance [81, 142]. In Section 3.2, we explicitly ask participants about whether or not their query is a reformulation in an effort to avoid issues around automated detection of query reformulation.

Researchers specifically explored search sessions to learn the usage of the web search by software engineers [15, 142]. They concluded that the major intentions of web search are finding information to 1) debug an error or an issue, 2) accomplish a specific task, 3) learn about a topic and 4) learn about a specific API element. The researchers also suggested that code related queries are more verbose [142]. Code related queries also lead to higher rates of reformulation, longer dwell time, and fewer clicks [15, 142].

Various code search tools have been developed to facilitate developers with searching for the desired code, such as Koders, Krugle, and Sourcegraph. Despite these efforts, Google remains the most popular general-purpose search engine with developers [160, 169], which is why we target it in this dissertation.

2.2.2 Measurements of Students' Programming Activities

Studies that focus on students' programming performance and behaviors range in size from a few students working on small tasks in controlled, hour-long lab sessions to hundreds of students working on assignments and/or projects in a CS class over the course of a semester or even years.

Researchers and educators often measure the accuracy of student-authored source code via the number of passed test cases/suites [8, 89, 184] while students' productivity is often evaluated using the changes in the number of lines of code (LOC) per hour [7] or per work session [89].

Pettit et al. [134] investigate how students modify their programs given automated assessment tools, using more than 45,000 CS1 student submissions collected over seven semesters. They evaluate students' programming progress using the number of LOC changed between each test execution or submission. Programming processes are also commonly measured using compilation behaviors; for example, the time interval between two compilations [80, 89], and the total number of compilations [180, 184].

In this dissertation, we quantify students' programming performance in terms of: 1) accuracy, such as how many assertions and test cases do student-authored source code pass, and 2) effectiveness, for example, the number of lines of code changed between two consecutive test suite runs, and the time intervals between test suite runs.

2.2.3 Students' Self-assessment of Ability and Persistence Intentions

In CS education, researchers and educators have sought to identify the factors that impact students' self-assessment of ability in CS, especially for introductory level students (e.g., [105, 157, 181]). Bandura [14] posits that, in general, performance attainment is the most important factor that people rely on when making judgments of self-efficacy. Prior work also suggests that self-assessment of ability is a significant predictor of persistence in a field [40] and success in CS [157, 181].

Considerable effort has also been made to improve students' self-assessment of abilities and persistence in CS. Some common approaches include providing research opportunities to students early in their undergraduate education [128], or sending them to technical conferences [186]. More recent approaches have examined the impact of improved feedback on self-assessment, persistence, confidence, and engagement. Fisk and colleagues [56] found that positive performance feedback from instructors via email increased students' self-assessed CS ability, as well as women's intentions to persist in CS. Parkin et al. [126] also found that releasing grades along with feedback from instructors can improve students' confidence in a course, and significantly enhance their engagement. Feedback from tools has also been found to improve students' intentions to persist in CS. Marwan et al. [108] integrate an adaptive immediate feedback system into a block-based programming environment, which provided real-time, personalized, positive, and corrective feedback to students. They observed that the adaptive immediate feedback led to better student engagement, and significantly increased their intentions to persist in CS.

In this dissertation, we examine how CS1 students' intentions to persist in CS were impacted by the granularity of feedback by testing source code through JUnit frameworks, as the presence of *Assertion Roulette* in the provided test suite causes students to receive less granular feedback.

2.3 Checklists in Computer Science

Checklists have been identified as a useful tool in software engineering for code review [36, 148] or encouraging software security [1]. Prior work [110] has found that students appreciate having a checklist of programming sub-goals during implementation. To provide testing support without the steep learning curve of a new tool [86, 87], we conjectured that a checklist containing testing tips could alleviate some students' testing struggles.

2.3.1 Checklists in Software Engineering

Checklists have been perceived as a handy and effective tool for mastering various skills or performing various tasks in software engineering, such as code review [36, 148], software inspection [27, 114, 163], and ensuring software security [1, 65]. For example, McMeekin et al. [114] found developers who completed a checklist-based reading inspection were able to better understand and more systematically modify the code than those who did not.

Educators in software engineering have also adopted checklists to assist teaching and student learning [133, 148, 152]. Rong et al. [148] found a checklist served as a necessary and helpful guideline for inexperienced students to conduct code review. They encouraged students to construct and maintain their own checklists, but cautioned students that use of a checklist solely did not guarantee a high-quality code review. By contrast, Chong and colleagues [36] investigate the mistakes in student-authored code review checklists to inform learning activities of code review. They also found that developing a code review checklist stimulated students in developing their analytical skills for code reviews.

In this dissertation, we present a testing checklist that is deliberately lightweight and provides testing support without a steep learning curve. To the best of our knowledge, the application of checklists to software testing is novel.

2.3.2 Explicit Strategies

LaToza et al. [102] introduce the concept of explicit programming strategies – human-executable procedures for accomplishing programming tasks. The explicit programming strategies emphasize *how* to work in a specific way. The researchers also found that developers who used explicit strategies were objectively more successful at the design and debugging tasks than those who were free to choose their own strategies. The explicit programming strategies also allowed developers to work in a more organized, systematic, and predictable way, although more constrained at the same time [102].

Ko and colleagues [93] introduce the explicit strategies to an introductory programming course. They found that students largely ignored the more systematic strategies, and preferred to engage in rapid cycles of editing and testing, without deeply understanding their code. They point out that the explicit strategies might be too sophisticated for students to learn while also learning basic programming concepts.

In this dissertation, we integrate the explicit strategies into the testing checklist to help students determine *what* and *how* to test. Similar to prior work [93], we evaluate the explicit strategies in a classroom setting.

CHAPTER

3

BARRIERS IN SEARCHING AND USING ONLINE RESOURCES FOR STUDENTS

This chapter presents studies identifying the barriers that students have when working with an unfamiliar language while searching and using online resources. To encourage students' searching behaviors and adoptions of online resources, we conducted two studies that expected students to implement small programs that either require usage of an unfamiliar API in a familiar language (Section 3.1) or in an unfamiliar language (Section 3.2). We summarize the barriers that students have in Section 3.3. Additional details and materials for these studies can be found in Appendix A.

These studies address the bolded parts of my thesis statement:

> Motivated by the **identification of barriers students have during software engineering activities**, in particular software testing, we show that a lightweight checklist intervention improves student testing practices.

3.1 Study 1: Students' Strategies of Searching and Using Online Resources While Using an Unfamiliar API

This study received IRB approval from NCSU (Protocol Number #11878), and was published in ICPC '19 [9].

Gina R. Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T. Stolee. "Exploring Tools and Strategies Used During Regular Expression Composition Tasks". In: *Proceedings of the 27th International Conference on Program Comprehension*. ICPC '19. 2019, pp. 197–208

3.1.1 Study Rationale

In CS education, it is a common practice for instructors to provide students with detailed program specifications and a set of unit tests to verify the functionality of student-authored code. Understanding how students search and use online resources (e.g., code examples and web-based tools) to facilitate programming and satisfying the unit tests allow us to better support them.

We conducted an exploratory case study to reveal the strategies that students adopt and the barriers that students face when searching and using online resources during program implementation. Students were expected to compose regular expressions in Java that pass provided unit tests illustrating the intended behavior. We combined surveys and screen-capture videos to better understand students' overall browsing behaviors and performance on the tasks. This study addresses the general research question RQ1, which is broken into two sub-questions:

RQ1: What barriers do students have when searching and using online resources?

RQ1 (a): What online resources do students adopt while using unfamiliar API?

RQ1 (b): How do students employ these online resources while using unfamiliar API?

3.1.2 Methodology

The study was conducted in a lab setting over two sessions, one hour each. Students attended one lab session only. All desktop computers in the lab had screen capturing software and browsers installed prior to the study. Eclipse was pre-loaded with the study tasks. Students were asked to use the lab computers only; no personal computers were allowed.

3.1.2.1 Participants

In total, 29 students (25 undergraduate and four graduate) were recruited through e-mails sent to students who had passed the introductory object-oriented programming course at Iowa State University. Every student was required to fill out a survey before they started to solve the tasks. Each participant received $20 in cash upon completion of the study.

3.1.2.2 Survey

At the beginning of the study, students were asked to complete a paper-based survey on educational classification and programming expertise.

Students in this study had an average of 4.2 years of programming experience, and an average of 3.3 years of Java experience. The survey responses showed that the majority of students (22/29) considered themselves as having intermediate Java programming knowledge though 20 students had little or no experience with regular expressions. Students either had no experience with JUnit tests at all (19/29) or used them regularly (10/29). All students made use of web searches while programming.

3.1.2.3 Regular Expression Composition Tasks

There were 20 tasks available for students to complete. These tasks were selected for diversity of questions and to cover the most common language features of regular expressions [34]. Table 3.1 shows the complete list of tasks (*Task*), the number of JUnit tests in each task (*#JUnit*) and their sample regular expression solutions (*Sample Regular Expression Solutions*), including escaped slashes. Each student received a list of tasks from Table 3.1 defining the order in which they were to attempt the tasks. The order was randomly generated to avoid potential learning effects.

3.1.3 Data Analysis - Screen-capture Video Transcription

To enable a quantitative analysis of the students' problem solving processes, the screen-capture videos were transcribed into the log format.

Trigger events are used to identify when a log entry should be made. A trigger event is an on-screen event that prompts the transcriber to log the action. Once an event trigger is

Table 3.1: Regular Expression Problems and Sample Solutions.

<table>
<tr><th>Type</th><th>Task</th><th>#JUnit</th><th>Sample Regular Expression Solution</th></tr>
<tr><td>Extraction</td><td>AlternatingParity</td><td>7</td><td><code>(?<!\\d)(([02468][13579])+[02468]?|</code>
<code>([13579][02468])+[13579]?)(?!\\d)</code></td></tr>
<tr><td>Extraction</td><td>GoogleKeywords</td><td>5</td><td><code>.*www.google.com.*q=(.*)</code></td></tr>
<tr><td>Extraction</td><td>OnMinuteEvents</td><td>5</td><td><code>\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:00Z\\s+(.*)</code></td></tr>
<tr><td>Extraction</td><td>PossessedPossessions</td><td>4</td><td><code>[a-zA-Z]{2,}'s([a-zA-Z]{2,})</code></td></tr>
<tr><td>Extraction</td><td>ReceiptScanner</td><td>4</td><td><code>\\A\\s*(.+?)\\s*[\n]|\\\$?(\\d+(\\.\\d{2})?).*\\Z</code></td></tr>
<tr><td>Extraction</td><td>RepeatedWords</td><td>5</td><td><code>(\\w+),?\\1\\b\\</code></td></tr>
<tr><td>Extraction</td><td>TSVParser</td><td>4</td><td><code>([^\t\n]+)\t[^\t\n]+\t([^\t\n]+)</code></td></tr>
<tr><td>Extraction</td><td>VerbPortion</td><td>5</td><td><code>\\b([a-zA-Z]{2,})ing\\b</code></td></tr>
<tr><td>Extraction</td><td>JavaIntDeclaration</td><td>5</td><td><code>\\s*int\\s+\\w+\\s*=\\s*(\\d+);</code></td></tr>
<tr><td>Extraction</td><td>LastDuplicateByte</td><td>6</td><td><code>^.*([0-9ABCDEF]{2}).*?\\1.*$</code></td></tr>
<tr><td>Extraction</td><td>ShortestDNA</td><td>5</td><td><code>(AT(((?!AT).)*?)GC)</code></td></tr>
<tr><td>Extraction</td><td>TrimWhitespace</td><td>6</td><td><code>^\\s*(.*?)\\s*$</code></td></tr>
<tr><td>Extraction</td><td>HasSoyIngredient</td><td>14</td><td><code>\\b(Edamame|Kinnoko|Kyodofu|Miso|Natto|Okara|Shoyu|Soy(a)?|</code>
<code>soybean(s)?|Tamari|Tempeh|Teriyaki|vegetable protein|Tofu|</code>
<code>Yakidofu|Yuba|TSF|TSP|TVP)\\b</code></td></tr>
<tr><td>Extraction</td><td>NoGremlins</td><td>6</td><td><code>\\A[^\\f\\r\b]*\\z</code></td></tr>
<tr><td>Extraction</td><td>NoVowelsWord</td><td>5</td><td><code>\\b[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]{2,}\\b</code></td></tr>
<tr><td>Validation</td><td>OutlineFormat</td><td>10</td><td><code>^\\s*([a-z]|[A-Z]|i+|\\d+)\\.\\s+.*$</code></td></tr>
<tr><td>Validation</td><td>ReverseSentences</td><td>5</td><td><code>\\.(\\S+)*\\S*[A-Z](\\s+\\.(\\S+)*\\S*[A-Z])*</code></td></tr>
<tr><td>Validation</td><td>SpacedWords</td><td>11</td><td><code>^([a-zA-Z0-9]+\\s+)+([a-zA-Z0-9]+)?$</code>
<code>|^(\\s+[a-zA-Z0-9]+)+(\\s+)?$</code></td></tr>
<tr><td>Validation</td><td>ValidEmail</td><td>8</td><td><code>[A-Z0-9a-z.!#$%&'*+-/=?^_`{|}~]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}</code></td></tr>
<tr><td>Validation</td><td>ValidPhoneNumber</td><td>13</td><td><code>^(\\([0-9]{3}\\)|[0-9]{3})[-]?[0-9]{3}[-]?[0-9]{4}$</code></td></tr>
</table>

seen, the event is logged. The logs contain columns to describe the event. When a trigger event occurred, a row was added to the log and its associated column(s) were logged. A detailed description of the columns follows:

Time: Current time in the video when an event occurred.

Task: The name of the task in Eclipse.

Search: String from an online search query.

Website Visited: Current website visited.

Regex String: Regular expression in Eclipse or web tool.

Copy Paste: Type of copy-pasted item: test string or regular expression.

Debugger: True if using the Eclipse Debugger.

Eclipse: True if the event occurred while using Eclipse.

Web Tool: True if the event occurred while using a web tool.

JUnit Tests: True if the student visited the JUnit tests.

Test Passed: The specific test(s) that passed.

Pass Rate: The pass rate of the executed JUnit tests, calculated as: $\frac{\#\ Passed\ JUnit\ Tests}{\#\ Total\ JUnit\ Tests}$

We consider the following as trigger events for the log entries:

- Application switch (logged: *Time*, *Eclipse*, *Web Tool*)
- Switch to browser (logged: *Time*, *Website Visited*)
- Search in browser (logged: *Time*, *Search*)
- Access website in browser (logged: *Time*, *Website Visited*)
- Access debugger or development environment in Eclipse (logged: *Time*, *Debugger*)
- Copy regular expressions or strings (logged: *Time*, *Copy Paste*, *Website Visited* or *Eclipse*)
- Paste regular expressions or strings (logged: *Time*, *Copy Paste*, *Website Visited* or *Eclipse*)
- Compose/Edit regular expressions (logged: *Time*, *Regex String*)
- Run JUnit tests (logged: *Time*, *Test Passed*, *Pass Rate*)
- Switch task in Eclipse (logged: *Time*, *Task*)
- Switch between JUnit tests and code in Eclipse IDE (logged: *Time*, *JUnit Tests*)

For example, consider a student who 1) opens task *ValidEmail* and starts to work on it at 00:10:04, 2) switches to Chrome browser and 3) searches "valid email in regex Java" at 00:10:34, 4) clicks on and visits a Stack Overflow result at 00:10:39, 5) copies an existing sample regular expression, `.+@.+`, at 00:12:15, 6) switches to Eclipse at 00:12:16, and 7) pastes into Eclipse at 00:12:17, then 8) runs the tests. In this scenario, the eight trigger events are: 1) switch task in Eclipse, 2) switch to browser, 3) search in browser, 4) access website in browser, 5) copy regular expressions or strings, 6 application switch, 7) paste regular expressions or strings, 8) run JUnit tests. The total columns being logged are: *Time, Eclipse, Search, Website Visited, Copy Paste, Regex String, Test Passed, Pass Rate.*

Table 3.2: Online Sources & Average Improvement in Pass Rate & Average Pass Rate.

Online Sources	**avgImp**	**avgPass**	**#Attempt**
Q&A sites only	24%	50%	7
D&T sites only	35%	62%	13
Both Q&A and D&T sites	31%	51%	57
None	29%	70%	17
Total			94

3.1.4 Results

In total, the 29 students viewed 121 tasks, but only ran the tests for 94, yielding 94 attempts for analysis.

3.1.4.1 RQ1(a): Students Need Documentation and Tutorials

We classified the information sources online into two categories: Q&A sites (e.g., Stack Overflow, online forums) and Doc&Tutorial sites (official documentation and official/third-party tutorials), similar to prior work [26]. We investigated the percentage of improvement in the passing test cases after consulting the online resource.

Table 3.2 lists the categories (*Online Sources*), their associated average improvement in pass rate (*avgImp*) and average pass rate after consulting each of several web sources (*avgPass*), and the number of attempts (*#Attempt*) for each category that was consulted. Of the 29 participants, 28 used search during their problem solving tasks. The most common resources consulted were Stack Overflow (visited 677 times), OracleDocs (visited 282 times) and Vogella (visited 92 times).

The results revealed that students who only consulted official documentation and tutorials for regular expressions improved their pass rate in JUnit tests by 35% on average, which provided the highest improvement in pass rate. Attempts being solved by consulting only Q&A sites led to the lowest improvement in pass rate; those being solved without consulting any online sources achieved highest pass rate on average. A potential explanation for this phenomenon is that participants who only relied on Q&A sites were not engaged in learning and thinking as actively as those who consulted documentation and tutorials, and those who did not search may have prior experience with regular expressions.

3.1.4.2 RQ1(b): Students Reuse Existing Code

We observed that students sometimes reused the regular expressions from search results, and the most frequent online resources participants copied from were Stack Overflow, OracleDocs, Vogella, TutorialsPoint and Regular-Expressions. We found that 35% (33/94) of the regular expression strings being tested contained the online code. Students tested the copy-pasted regular expression strings directly as solutions in some cases (after a copy and paste, 36.3% of the time the regular expression was tested immediately after pasting), but more often participants modified the regular expression strings before testing (after a copy and paste, 57.7% of the time the regular expression was directly modified and then tested). We also noticed that 29 of 80 copy-and-pasted regular expression strings from websites to Eclipse were modified to simply resolve compile errors.

3.1.5 Takeaways

The combination of surveys and video transcriptions has led to insights regarding the tools and problem solving strategies students used during regular expression composition. In this section, we elaborate on the main findings.

3.1.5.1 Insufficient IDE Support

We found that the developing and testing environments are not always sufficient for regular expressions. For example, the JUnit tests return only pass/fail for regular expression users and the Eclipse built-in debugger does not explain why the matching fails. Students who consulted web-based tools (e.g., [46, 122, 145]) that support dynamic testing, visualize the regular expression behavior, and explain the syntax of the regular expression, passed more tests than those who did not consult web-based tools (67.6% vs. 54.6% in terms of pass rate)). However, only 9 of 29 students were able to find the web-based tools for regular expressions when searching for online resources.

3.1.5.2 Visualization

Visualization, as provided by web-based tools, seemed to help students with composition (Section 3.1.5.1). Beck et al. [19] conclude that regular expression experts agree that visual encoding of the regular expression is beneficial. Other work on visualization also pointed out that visualization can support the task solving process by displaying useful information in a condensed way [101], enhance the developer's understanding of the process of the execution [117], and serve as an external memory aid to help developers tracking runtime execution [94]. Moreover, visualization can significantly help with conquering the design barrier where developers have cognitive difficulties with visualizing solutions to programming problems [95]. Our results support these findings. Therefore, we suggest that features such as visualization and quick search for documentation should be integrated into the IDE to facilitate regular expression comprehension and reduce context switching for users. Tools for different languages that support regular expressions should be developed as well.

3.1.5.3 Example Integration

Students seemed to assume that regular expressions are a *lingua franca* [44], but they are not. We observed that some students faced challenges in migrating regular expressions from web-based tools or websites they visited to Java due to differences in programming language representations of regular expression. As mentioned in Section 3.1.4.2, over 36% of copy-and-pasted regular expression strings from websites to Eclipse were edited to address the compile errors. For example, a single backslash in Python needs to be edited to a double backslashes in Java. This suggests that a language migration tool would be helpful.

3.1.6 Threats to Validity

Conclusion: Editing and testing activities within web-based tools are not part of the IDE flow and thus not captured in our results.

Some tasks can produce a non-zero pass rate when tested against an empty regular expression. This situation might lead to an overestimation of participants' performance. In this study, two participants tested three tasks in total against an empty regular expression and gained 32.7% pass rate in average. This special case is rare, so the influence is small.

Construct: Students knew they were being observed which might have influenced their searching and composition behaviors.

Internal: Most students attempted multiple tasks, yet we did not consider task order in the analysis of the results. It is possible that learning effects influenced pass rates of tasks attempted later in the study. However, as the tasks were assigned in random order, the impact should be distributed across all tasks.

External: Some tasks in this study have solutions that are less complex than regular expressions in the wild[1], and thus may not be representative of tasks for which developers would use regular expressions. Future studies should use tasks whose solutions are reflective of regular expressions found in source code repositories.

[1]We thank Peipei Wang for assistance with the regular expression complexity.

3.2 Study 2: Students' Code Searching Activities While Using an Unfamiliar Language

This study received IRB approval from NCSU (Protocol Number #11719), and was published in ICPC '20 [10].

Gina R. Bai, Joshua Kayani, and Kathryn T. Stolee. "How Graduate Computing Students Search When Using an Unfamiliar Programming Language". In: *Proceedings of the 28th International Conference on Program Comprehension*. ICPC '20. Seoul, Republic of Korea, 2020, 160–171

3.2.1 Study Rationale

Developers and computing students are usually expected to master multiple programming languages [111, 173]. To learn a new language, developers and students often turn to online search to find information and code examples [16, 150, 177]. However, insights on how students perform code search when working with an unfamiliar language are lacking. Understanding how students search and the challenges they encounter when using an unfamiliar language can motivate future tools and techniques to better support students [22, 75, 119, 155].

We conducted an exploratory study on how students search for code in VBA, which is an unfamiliar programming language to the study population. We used *in situ* surveys to explicitly learn about search success and query reformulation to gather reliable data on those metrics. With a combination of search logs and survey responses generated by students, we were able to study how students perform code search, as well as how we may be able to train students to use search better in their workflow, or build a better search tool to facilitate more successful searches. This study addresses the general research question RQ1, which is broken into two additional sub-questions:

RQ1: What barriers do students have when searching and using online resources?

RQ1(c): What do students search for while using an unfamiliar language?

RQ1(d): How do students search while using an unfamiliar language?

3.2.2 Methodology

The study was conducted in a lab setting over two sessions, 90 minutes each. Students attended one lab session only. Students were instructed to install the Google Chrome extension (Section 3.2.2.1) on their personal laptops and keep the extension enabled during the lab session.

3.2.2.1 Tool Implementation - Google Chrome Extension

We developed a Google Chrome extension[2] that records search events and browsing history and deploys surveys (Section 3.2.2.2). The implementation consists of two parts: a client-side Google Chrome browser extension for logging data and a Flask web server for storing data. On the client side, the browser extension does the following:

1. initialize browser local storage that holds the log information, including issuing a unique 10-digit ID number to each participant,
2. track Google searches,
3. track link clicks on all pages,
4. detect possible query reformulations via word-level Levenshtein distance (if distance $\geq$ two words [81]),
5. format and deploy the surveys, and
6. send all collected logs to our server for storage.

All collected data are stored in a password-secured SQLite database on an encrypted server.

3.2.2.2 In situ Surveys

To capture the questions the students are trying to solve, we prompt users with short surveys periodically. To avoid over-taxing the students, a maximum of 10 surveys per hour were deployed with a one-minute minimal interval between surveys.

A survey is triggered when a student performs one of the following four actions:

[2]Chrome web store: `https://chrome.google.com/webstore/detail/chrome-code-search/ljdehfmdnkelbnpogdeobpaldeibecek`

1. **Finishes installing the extension**
   Preliminary Survey (Appendix A.2.2.1). We collect students' technical background such as programming experience, intentions of code search, and demographic information.

2. **Makes a search**
   Context Survey (Appendix A.2.2.2). We ask students about the activities they are doing, and the specific question they are solving. Questions Q2, Q3, and Q5 are from prior work [150]. In addition, we ask what approach(es) they've adopted to solve the tasks.

3. **Reformulates a query**
   Query Reformulation Survey (Appendix A.2.2.3). A query is identified as a potential reformulation if its word-level Levenshtein distance from the prior query is greater than or equal to two. This survey verifies the reformulation by asking in the current query is related to the previous one. It also asks if the prior query solved the problem. If so, how did it help; if not, what information is missing.

4. **Closes the search tab**
   Search Tab Closed Survey (Appendix A.2.2.4). This survey asks if the prior query solved the problem. If so, how did it help; if not, what information is missing.

A survey will not be triggered by any event if the prior survey is deployed within a minute or the number of deployed surveys has reached the hourly limit. We asked students to complete the survey every time it is deployed, although completion is not forced.

3.2.2.3 VBA Programming Tasks

We chose Visual Basic for Applications (VBA) because it is a popular programming language with many online resources [15, 55], but it is not commonly used by the graduate students in Computer Science at North Carolina State University. Each task includes a general learning goal (e.g., conditions and loops, string manipulation), a task description with requirements, and a sample output/result for the task.

3.2.2.4 Participants

We invited students enrolled in a graduate software engineering course at North Carolina State University via email to participate in the study (response rate: 42.6%). A short introduction to this study, including the purpose thereof, and the tool (Google Chrome extension) used in the study, was made available to all potential students. As compensation, students who have the extension enabled and recording for at least 45 minutes were eligible to enter into a drawing to win a $30 Amazon gift card. In total 23 students completed the initial surveys. After discarding data from five students whose interactions were recorded for less than ten minutes, we were left with data from the remaining 18 students for analysis (average: 78.5 minutes, median: 84 minutes), which includes four female students and fourteen male students.

The 18 students self-reported their expertise in general programming skills: one as a novice developer, five as experts, and twelve as intermediates. Two students claimed to have prior working experience in the industry. All students often search on websites for code, and the majority of students sometimes or often ask friends for help (16/18) or search on IDEs (16/18). Students came from a diverse programming language background, including object-oriented programming languages and scripting languages, but not VBA. The top five frequent programming languages used by the students were: Java, JavaScript, C++, C, and Python.

3.2.3 Data Analysis - Search Logs

We grouped the logs by students' IDs and sorted them by log time. Surveys were tabulated and associated with browser events for analysis. Logs were analyzed to collect 1) search queries, 2) ordered list of clicked website URLs associated to each query, and 3) time spent by a student on a clicked result page.

The logs were split into search sessions. Due to our study context, adopting the definition of a search session from the study on Google developers [150] was problematic. When considering a search session as a series of developer activities separated by 6-minutes of inactivity, each student in our study had exactly one search session. Thus, we treat each search query and its associating result clicks as one search session (referred to as "micro-session" in prior work [150]).

Our reformulation surveys were deployed when the Levenshtein distance was two or greater, in line with prior work [81, 142]. If student answers *yes* to $Q1$ (Figure A.2.2.3), this indicates that the current query is not a reformulation and begins a new topic. If the student answers *no* to $Q1$ and *yes* to $Q2-no$, this means that the previous query did not solve the question and the current query is related to the same problem, and thus the current query is a reformulation. If a student answers *no* to $Q1$ and *no* to $Q2-no$, we assume that the problem is the same but the student is trying a new approach, and thus the query is not a reformulation. While prior work makes assumptions about reformulation based on topic analysis or query distances, these survey responses serve as a ground truth for reformulation analysis. While we do not have enough data to build a classifier to predict reformulation, we can analyze the context around known reformulations versus known non-reformulations.

3.2.4 Results

In total, the 18 students generated 3,508 log entries, including 216 survey responses and 229 queries during the lab sessions. Each student made an average of 12.7 queries (median: 13 queries), each query led to an average of three search result clicks (median: two clicks), and the average time interval between each query was 5.8 minutes (median: 4.3 minutes).

As to the surveys, 216 of 389 deployed surveys were completed and collected, including 18 *Preliminary Surveys* (response rate: 100.0%), 35 *Context Surveys* (response rate: 39.3%), 138 *Query Reformulation Surveys* (response rate: 54.5%) and 25 *Search Tab Closed Surveys* (response rate: 86.2%).

3.2.4.1 RQ1(c): Students Search for Example Code

We analyzed the self-reported purposes of why students search during programming. In 18 *Preliminary Surveys*, the most frequently reported goals were to find specific syntax, libraries/classes/APIs, implementations or sample usage of methods/functions/operations during development (Table 3.3).

We then explored the questions students were trying to solve as collected in 35 *Context Surveys* (See Table 3.4). Students were asked to categorize their motivation into exploring, designing or debugging (*Goal*), and then answer the open-ended question, "What question are you trying to answer?" with a brief description of current search activity; example queries can be found in column *Example*. For example, 19 queries were made to explore how to perform an action in Excel or with VBA with queries such as *"How to print in VBA"*.

Students explored code (Table 3.4) and searched for example code (Table 3.3) frequently. They also searched for assisting debugging, which matches their self-reported purposes for searching were example codes, including APIs, syntax and existing solutions, and determine reasons why code did not work correctly (Table 3.3). This observation also matches the search goals of professional developers reported in the prior work [150]. While only 10% of professionals' queries tried to determine why something is failing, 17% of students' queries recorded in *Context Surveys* focused on debugging.

Table 3.3: Answers to 18 Preliminary Survey Question "In what contexts or for what purpose(s) do you search for codes via website during software development activities" (*Category* - Multiple Answers are Allowed).

Category	**#par (%)**
Look for Example Code (How)	
Library / Class / API	9 (50.0%)
Method / Function / Operation	7 (38.9%)
Syntax	9 (50.0%)
Solution	4 (22.2%)
Explore Code (What)	
Concept	3 (16.7%)
Determine Reasons (Why)	
Debug	7 (38.9%)

Table 3.4: Answers to 35 Context Survey Question "What are you doing?" (*Goal* - Multiple Choice) and Question "What are you trying to answer?" (*Example* - Short Answer).

Goal	**Example**	**Num**
Exploring	"How to enable developer tools in Excel" "How to do lookup in excel" "How to print in VBA"	19
Designing	"Basic steps to record an excel macro" "Using and iterating hashtable in VBA" "Find unique values in a column"	8
Debugging	"Loop over a column excel vba" "Create dictionary in excel vba" "Found an implementation but was getting error"	6
Other	"enabling a feature in Microsoft Excel"	2

3.2.4.2 RQ1(d): Students Borrow Terms from Other Languages

We identified that 16/18 students borrowed the terms from languages with which they are familiar. Similar borrowing behavior was also reported in prior work [6, 158], and developers claim that previous knowledge supports retrieval of new information; our results concur. We found 65 queries used terms such as "dictionary", "hashtable", "hashmap", "regex" from languages like Java and C++. However, this term borrowing sometimes introduced term mismatch, that is the cases when vocabulary mismatch between queries and documents (e.g., [41, 72, 189, 191]). For example, borrowing the term "dictionary" from Java. When five students intended to search for functions in VBA that behave like `java.util.Dictionary`,

Table 3.5: Online Sources & Success.

Online Sources	**#Succ**	**#Fail**	**Total**
Q&A sites only	19 (11.7%)	9 (5.5%)	28 (17.2%)
D&T sites only	54 (33.1%)	14 (8.6%)	68 (41.7%)
Both Q&A and D&T	25 (15.3%)	20 (12.3%)	45 (27.6%)
None	9 (5.5%)	13 (8.0%)	22 (13.5%)
Total	107 (65.6%)	56 (34.4%)	163 (100.0%)

which should be the `VLOOKUP` function, they searched for, "*dictionary in excel*", "*add or edit words in a spell check dictionary*" or, "*creating a dictionary with Microsoft Excel*".

The impact of term mismatch in terms of query success and reformulation is fairly neutral. Queries with borrowed terms were more successful, on average; this observation is further supported by a result in general information search [190] that finds engineering and science students who had higher levels of domain knowledge retrieved more relevant documents for the search question. However, these queries were no more associated with reformulation than any other query. For the 65 queries demonstrating term mismatch, 34 (52%, 26 from *Reformulation Survey*, 8 from *Search Tab Closed Survey*) were associated with a success survey, and only 15 (23%, 13 from *Reformulation Survey*, 2 from *Search Tab Closed Survey*) were reported unsuccessful. Despite the term mismatch, these queries were more successful on average than the typical query (77% success vs. 66% success, per Table 3.5). We conjecture that this is because VBA is so well documented that the current search algorithms are able to handle the term mismatch; this may not be the case with newer or less common programming languages.

There were subsequent query reformulations for 33% (13/39) of the term borrowing queries, where 39 of the term borrowing queries were followed by a reformulation survey, and 13 of the responses explicitly indicated reformulation occurred. This is the same proportion we saw from all reformulation surveys (34.8% are reformulations, 48/138).

3.2.5 Takeaways

In this section, we compare the student search behavior to professionals, as well as provide the suggestions to improve search success.

Table 3.6: Summary on Study Designs of Sadowski's Study [150] and this Study.

	Sadowski's Study	**This Study**
Duration	15 consecutive days (4 weekends and 11 weekdays)	90-minute lab session
# Participants	27 Google developers	18 NCSU graduate students
Compensation	None	Drawing to win a $30 Amazon gift card
Context	Daily work	5 VBA tasks
Survey Types	1 survey, 4 questions	4 surveys, 2-6 questions each (Appendix A.2.2)
Survey Setting	10 surveys per day 10-minute time interval	10 surveys per hour 1-minute time interval
Survey Responses	394 responses	216 responses
Survey Collection	Browser extension, surveys triggered when developers accessed the internal code search site	Browser extension, survey triggered under four conditions (Section 3.2.2.2)
# Total Log Entries	77,632 entries	3,508 entries
# Total Queries	3,870 queries	229 queries

3.2.5.1 Comparison of Search Behaviors between Students and Professionals

Our study can be viewed as a partial replication of the study methodology of prior work [150], in which Sadowski et al. explore the search behavior of Google developers, with a drastically different context. Table 3.6 summarizes the similarities and differences of the context and collected data between the previous study and this study.

In our study, we investigated how students search for code in an unfamiliar programming language, rather than using professional developers conducting their daily work. Like Sadowski's study, we explored the reasons why participants search, the properties of search queries, and search sessions in micro level. To extend Sadowski's study [150], we analyzed the properties of successful queries with the survey responses collected right after either a search activity or a search tab being closed. We also explicitly ask about query reformulation instead of speculating based on query edit distance. Table 3.7 summarizes the similarities and differences of the main findings of Sadowski's study and this study.

For example, search sessions observed in this study were more likely to have at least two clicks after one search activity (55%), while the professionals click no result in the most search sessions (38.8%). While it only took professionals 23 seconds to reformulate or make a new query, students usually spent five minutes before reformulating or starting a new query. students searched more frequently than the professionals did (13 queries/hr vs. 12 queries/day).

Table 3.7: Summary on Findings of Sadowski's Study [150] and This Study.

Professionals	Students
Why do they search?	
Developers use code search to answer questions a wide range of topics, including how to do something, learning what code does, and determining why code is behaving as it is.	Most searches are associated to goals of exploring for example code and designing a new feature. Students also search for debugging.
What are they looking for?	
Developers search for examples more than anything else. Developers navigate code search tools even through code they know well.	Students search for example code more than anything else. All students search for exploring regardless of the familiarity to the code.
Query properties	
The average length of the queries was 1.85 words. Most queries were scoped to a subset of the code repository.	The average length of the queries is 5.6 words. Most queries contain keywords "VBA" and "excel" to restrict the search.
Nearly a third of searches are incrementally performed through query reformulation.	Only 35% (48/138) of the suspected reformulations were actual reformulations. Adopting the reformulation definition from Sadowski's study, we found only 5.2% (12/229) of the queries had an edit distance of one.
Time between queries	
A typical professional developer requires eight seconds (average of 23 seconds) to scan the search results, determine if they meet their needs, and reformulate an initial query.	A typical student requires three minutes (average of five minutes) to scan the search results, determine if they meet their needs, and reformulate an initial query.
Search frequency	
An average professional developer composes 12 search queries per weekday.	An average student compose 12.7 queries in a 90-minute lab session.
Clicks per query	
Each query lead to 1.3 clicks on average. Most micro-sessions contain one search without any clicks.	Each query lead to three clicks on average. Most search sessions contain one search followed by at least two clicks.
Clicks vs. Search goal	
Most searches on code reviews (determine why) lead to multiple clicks. Developers also make multiple searches and clicks when exploring for example code and designing.	Most searches on exploring new content/topics and all debugging search activities led to one search followed by multiple clicks.
Clicks vs. Code familiarity	
Code familiarity does not mean less clicks.	Code familiarity does not mean less clicks.

Continued on next page

Table 3.7 (continued).

Professionals	Students
Search success	
N/A	Students frequently borrow terms from familiar languages, and hence be more successful than the typical query. The most successful searches occur when looking for APIs or implementations.

3.2.5.2 Successful Search Behaviors for Students

Search success in this study was associated with consulting official documentation and tutorials rather than on Q&A sites (e.g., Stack Overflow). Additionally, mapping the context in an unfamiliar languages to previous knowledge in a familiar language, and borrowing the corresponding terms, also led to search success. Lastly, natural language phrases were more associated with success than coding shortcuts. For example, spelling out "equals" instead of "==", and search with "if statement" instead of "if". These behaviors may increase the likelihood of finding relevant results, at least in contexts similar to the one studied here.

3.2.5.3 API Translation

Shrestha et al. [158] point out that developers usually face a selection barrier [95] when moving to a new programming language, and searching for the right terminology and code example is difficult. They also concluded that learning a language is difficult when there is little to no mapping of features to previous languages [158]. While in this study term mismatch was not an issue for developers in that reformulation was not more likely and search success was actually higher, we conjecture that this was due to the study context. VBA was chosen because it is well-documented, and the participants seemed to benefit from this. For languages or APIs that are less well documented, we envision "API translators" that can map an API in one language to a destination language and provide short sample code would assist developers to work and learn more efficiently, especially when migrating languages, or working with new, unfamiliar languages. Some of these features are present in tools like Mica [170], which can assist developers in finding the right API classes and methods given a description of the desired functionality. Chen et al. [35] also present an approach that can recommend analogical libraries for different programming languages or different mobile platforms given a library.

3.2.6 Threats to Validity

Conclusion: This study did not collect the code solutions from participants, which left us unable to evaluate whether a participant completed the tasks in a way that fully met the task requirements. However, as we focused on queries and searching results in this study, the actual completeness of the tasks was not a priority.

Additionally, the conclusions were drawn based on our participants' search behaviors when using VBA, an event-driven programming language used by Excel, which may not generalize to learning other languages.

Construct: Participants knew their searching and browsing activities were being recorded, and were periodically surveyed, which may have influenced their behavior.

Some metrics used in this study may not be consistent across participants, such as #avgClick/Query and #avgQuery, since the search results and available documents may vary due to the personalized search results. In addition, the Chrome extension did not record if a search query was a result of using an auto-complete query suggestion. Future studies should enforce non-personal results and detect auto-complete.

Internal: Participants were required to work on these tasks in a lab environment in a limited time, which could potentially impact their search behaviors. Response bias may be introduced in the participant self-reported surveys, and impacted the validity of surveys.

All participants were graduate students and the majority claimed that their overall programming skills ranged from intermediate to expert (17/18). Therefore, the results may not generalize to other populations. A replication with a more diverse and larger set of students is needed.

We use survey responses on the reformulation survey as ground truth for reformulations. However, participants may have different notions about what is and is not reformulation, which may impact the validity of the ground truth.

3.3 Chapter Summary

Overall, these two studies pointed out **students' needs for code examples**, and revealed **two barriers** that students encountered when searching and using online resources:

1. **Example integration** (Section 3.1). This barrier occurs when students attempt to retrieve and reuse existing code from documentation or Q&A sites that is implemented in different programming languages.

2. **Term mismatch between languages** (Section 3.2). This barrier occurs when students attempt to search for code to support learning an unfamiliar programming language but do not know the exact terminology of the desired library or API to use in search query.

CHAPTER

4

BARRIERS IN LEARNING SOFTWARE TESTING FOR STUDENTS

This chapter presents studies identifying the barriers that students have when learning and practicing testing, specifically, unit testing. To better understand students' testing practices, we conducted two studies to investigate how students perceive and perform unit testing (Section 4.1), as well as how they interact with given unit tests (Section 4.2). We summarize the barriers that students have in Section 4.3. Additional details and materials for these studies can be found in Appendix B.

These studies address the bolded parts of my thesis statement:

> Motivated by the **identification of barriers students have during** software engineering activities, in particular **software testing**, we show that a lightweight checklist intervention improves student testing practices.

4.1 Study 1: Students' Testing Practices and Pitfalls

This study received IRB approval from NCSU (Protocol Number #18025), and was published in ITiCSE '21 [13].

Gina R. Bai, Justin Smith, and Kathryn T. Stolee. "How Students Unit Test: Perceptions, Practices, and Pitfalls". In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1.* ITiCSE '21. Virtual Event, Germany: ACM, 2021, 248–254

4.1.1 Study Rationale

Unit testing is reported as one of the skills that graduating students lack [4, 33, 37, 104, 121], yet it is an essential skill for professional software developers [70, 136]. Understanding the challenges students face during testing can help inform practices for software testing education.

To that end, we conduct an exploratory study to reveal students' perceptions of unit testing and challenges students encounter when practicing unit testing. We surveyed 54 students from two universities and gave them two testing tasks, one involving closed-box test design and one involving clear-box test implementation. For the tasks, we used two software projects from prior work in studying test-first development among software developers. We quantitatively analyzed the survey responses and test code properties, and qualitatively identified the mistakes and smells in the test code. We further report on our experience running this study with students. To our knowledge, this is the first work reporting students' unit testing practices and perceptions outside the classroom. It addresses the general research question RQ2, which is broken into three sub-questions:

RQ2: What barriers do students have when learning and practicing testing?

> **RQ2(a):** What aspects of unit testing do students perceive to be challenging?
>
> **RQ2(b):** What challenges do students encounter when creating or editing tests?
>
> **RQ2(c):** Does student-written test code smell good?

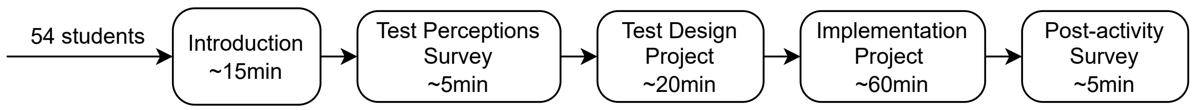

Figure 4.1: Study Procedure.

4.1.2 Methodology

The study was conducted in a lab setting over four sessions, two hours each. Each student attended one lab session only. At the beginning of the study, students received a 15-minute introduction on the procedure of the study and the overview of each testing project, including the expected functionalities and testing goals. Students were guided to a GitHub repository (Appendix C.1) containing links to 1) a preliminary testing perceptions survey (via Qualtrics), 2) a closed-box testing project, 3) a clear-box testing project, and 4) a post-activity survey (via Qualtrics). Students were instructed to complete the tasks in sequence, as shown in Figure 4.1. Students were allowed to use any Eclipse plug-ins and consult any online resources for assistance. Students uploaded their test code via Google Form upon completion of the study. During the study, screens were recorded to facilitate analysis.

Prior to the study, we conducted a pilot (n = 4) to clarify our survey questions and testing projects. As a result, we modified the survey choices to avoid ambiguity; removed questions unsuitable for a student population; and adjusted the suggested times allocated for each task.

4.1.2.1 Surveys

Test Perceptions Survey - We adapted the survey from Daka et al. [42] for the Test Perceptions Survey (preliminary survey, Appendix B.1.1). This survey asks about the techniques they use and the difficulties they have in writing/editing/fixing unit tests. The preliminary survey consists of four types of questions: selection questions, ranking questions, rating questions and distribution questions.

Post-activity Survey - At the end of the study, students completed a Post-activity Survey (Appendix B.1.2) that prompts them for the challenges they encountered during testing and their demographic information such as programming experience, working experience, and prior testing education. The Post-activity Survey consists of open-ended and selection questions.

```java
@Test
public void theRoverMovesForwardOnceLanding() throws Exception {

    /*
    * Assumption/Scenario: A rover is being created/landed and
    * moved forward based on the string command "f"
    * Testing Input: "f"
    * Expected Output: rover position: (0,1,N)
    */

    fail("Not yet implemented");
}
```

Figure 4.2: Example Test for *Design Project - Mars Rover API*.

4.1.2.2 Testing Projects

There were two testing projects. The closed-box testing task focuses on designing a test suite based on specifications. The clear-box testing task focuses on writing unit tests based on an implementation and specification. For both tasks, we provide example test cases, either in natural language or JUnit. Students are instructed to, "*test the program as thoroughly as possible based on the specifications.*"

Test Design Project (Mars Rover API) - We adopt this task from a prior study [54], as it is also used in other testing-related studies (e.g. [58, 174]). The objective is to test an API that follows the movement of a planet exploration vehicle (rover) and keeps track of its position and the obstacles that it may have encountered on a tour. The planet is set up as a 100×100 grid.

This testing project provides students with a description of expected behaviors, and asks them to write skeleton test cases that each contain: 1) a test name, 2) a description of testing scenario (in comments), 3) input (in comments), and 4) expected output (in comments). An example test is shown in Figure 4.2. This implementation-free project allows students who have limited experience with Java or JUnit to carry out the testing plan in natural language. Students had approximately 20 minutes to complete the task.

Implementation Project (Bowling Score Keeper) - We adopt this task from a prior study [54]; it is also used in multiple other testing-related studies (e.g. [58, 174]). The objective is to test an application that calculates the score of a single bowling game.

This testing project provides students with: 1) a description with expected behaviors, and 2) a completed program with one malfunctioning method (three classes, LOC = 86).

Students are asked to create JUnit tests to verify the behavior of this implementation against the specification. Students had approximately 60 minutes to complete the task.

4.1.2.3 Participants

We first conducted this study with 36 graduate students from North Carolina State University. To avoid the results being too specific to that context, we replicated this study with 18 undergraduate students from Lafayette College. All study participants are taking or have taken a software engineering course. Students are eligible to receive extra credit upon completion of the study.

Overall, students have an average of 4.1 years of programming experience (4.0 for undergraduates, 4.2 for graduates), and 70% of students believed that they are at least competent in programming and 50% of them were confident in Java, while only 26% of students considered themselves competent at unit testing.

All undergraduate students are taking or have taken a Computer Science course that involves unit testing (18/18), and the majority of them gained experience with creating/editing/maintaining unit tests (17/18). There were 23 (63.9%) graduate students who claimed that they are taking or have taken a Software Testing course in Computer Science education program; however, only 21 of them claimed that they are taking or have taken a course involves unit testing, and only 20 of them have experience with creating/editing/maintaining unit tests. A potential explanation is that some graduate students were given unit tests in programming assignments but not required to create new tests.

4.1.3 Data Analysis

In total, 54 students completed 108 surveys, designed 361 tests for the Test Design Project, and wrote 433 unit tests for the Implementation Project.

4.1.3.1 Surveys

For rating questions, we converted text Likert scale to numbers, where 0 maps to the lowest score, such as "Strongly disagree" and "Not at all important", and 1 maps to "Disagree" and "Slightly important" and so on. We treated them as interval-scaled data [73], and reported the average numeric value of each option. We used Borda count for ranking questions: among `n` candidates, a candidate receives `n-1` points for a first rank, `n-2` points for a second rank, and so on, with zero points for being ranked last.

4.1.3.2 Testing Projects

Test Design Project (Mars Rover API) - Since students are not expected to implement test code in the Design Project, we evaluated students' performance by manually measuring the **requirement coverage** with the following specifications:

- **Req 1**: The rover moves one position on the grid towards the direction it is facing given command "`f`".
- **Req 2**: The rover moves one position on the grid away from the direction it is facing given command "`b`".
- **Req 3**: The rover changes its direction counter-clockwise given command "`l`".
- **Req 4**: The rover changes its direction clockwise given command "`r`".
- **Req 5**: The rover returns its position and facing direction in the format `(pos_x, pos_y, direction)`.
- **Req 6**: If obstacles are found, the rover returns output in the format `(pos_x, pos_y, direction)(obs1_x, obs1_y)... (obsN_x, obsN_y)`. Obstacles are reported in the order they are found. The same obstacle shall be reported only once.
- **Req 7**: When the rover moves over the edges of the planet from North, it spawns on the opposite edges.
- **Req 8**: When the rover moves over the edges of the planet from South, it spawns on the opposite edges.
- **Req 9**: When the rover moves over the edges of the planet from West, it spawns on the opposite edges.
- **Req 10**: When the rover moves over the edges of the planet from East, it spawns on the opposite edges.

The example test shown in Figure 4.2 covers Req 1, introducing a *baseline requirements coverage of 10%.*

Implementation Project (Bowling Score Keeper) - We first manually measured the **requirement coverage** with the following specifications:

- **Req 1**: A frame consists of two throws.
- **Req 2**: The possible points for the first throw range from 0 to 10.
- **Req 3**: The possible points for the second throw range from 0 to 10.
- **Req 4**: The score of a frame is the sum of two throws and it ranges from 0 to 10.
- **Req 5**: A frame is a "strike" if the first throw is 10 points.
- **Req 6**: A strike frame receives bonus score of the value of the next two throws.
- **Req 7**: A frame is a "spare" if its score is 10 and it is not a strike.
- **Req 8**: A spare frame receives bonus score of the value of the next throw.
- **Req 9**: A bowling game consists of ten frames.
- **Req 10**: The score of a game is the sum of all frames.
- **Req 11**: If the tenth frame is a spare , there is one bonus throw.
- **Req 12**: If the tenth frame is a strike, there are two bonus throws.
- **Req 13**: The bonus throw(s) is added to the game score.

An example test is provided that covers Req 4 (Figure 4.3), introducing a baseline requirements coverage of 7.7%.

Next, we analyzed the quality of student-authored test code. We adopt EclEmma [48] to measure the **completeness** via instruction coverage (baseline = 20.3%) and branch coverage (baseline = 13.6%).

We measured the test suite **effectiveness** via mutation score (baseline = 11.5%), which is the percentage of killed mutants with the total number of mutants, supported by PITest [137].

```
@Test
public void testFrameScore() throws Exception {
    Frame f = new Frame(2, 4);
    assertEquals(6, f.score());
}
```

Figure 4.3: Example Test for *Implementation Project - Bowling Score Keeper*.

Table 4.1: "*How important are the following aspects for you when you write new unit tests?*" (5-point Likert Scale (0-4 in numeric), Preliminary Survey, 37 respondents).

Aspect	**Overall Rating**	**UGrad Rating**	**Grad Rating**
How easily faults can be localised or debugged if the test fails	3.0	3.0	3.1
Code coverage	3.0	2.9	3.0
Robustness against code changes	3.0	2.6	3.3
Sensitivity against code changes	2.9	2.7	3.1
How realistic the test scenario is	2.9	2.8	3.0
How easily the test can be updated when the underlying code changes	2.9	2.6	3.1
Execution speed	2.4	1.8	3.0

For **maintainability**, we identified potential smells in student-authored unit tests via test smell detector *tsDect* [130] and then manually validate the smells using their definitions [47, 62, 116, 146] as well as classifications from prior work [62]. No inter-rater reliability was considered in this process as only one author coded the smells, and the process required little personal interpretation [112].

4.1.4 Results

We report students' perceptions of unit testing by surveying 37 students who have experience with creating/editing/maintaining unit tests. We analyze and discuss the quality of student-authored test code generated by all 54 students.

4.1.4.1 RQ2(a): Challenges in Testing - Self-reported by Students

Challenge 1: Students did not have a consensus on good unit tests.

We explored the aspects that students perceived to be important when writing new unit tests (Table 4.1). These perceptions are based on their experiences before the study took place.

Both undergraduate students (*UGrad Rating*) and graduate students (*Grad Rating*) rated the ease of bug localization and code coverage, a common indication of test completeness, as the most important aspects of good unit tests. Graduate students also ranked highly the robustness of test cases against code changes, but undergraduate students believed it is less important. Additionally, undergraduate students overwhelmingly rated *execution speed* as the least important aspect. Overall, responses were distributed uniformly among the six options, which indicates most aspects are equally important to students, especially graduate

Table 4.2: "*Please rank the following aspects of writing a new unit test according to their difficulty*" (Borda Count in parenthesis, Preliminary Survey, 37 respondents).

Aspect	**Overall Rank**	**UGrad Rank**	**Grad Rank**
Determining what to check	1 (98)	2 (44)	1 (54)
Identifying which code/scenarios to test	2 (95)	1 (43)	2 (52)
Finding a sequence of calls to bring the unit under test into the target state	3 (64)	3 (28)	3 (36)
Finding and creating relevant input values	4 (58)	3 (28)	4 (30)
Isolating the unit under test	5 (55)	5 (27)	5 (28)

Table 4.3: "*Please indicate your level of agreement with the following statements*" (7-point Likert Scale (0-6 in numeric), Preliminary Survey, 37 respondents).

Agreement on statements...	**Overall Rating**	**UGrad Rating**	**Grad Rating**
I would like to have more tool support when writing unit tests	4.5	4.4	4.5
I would like to have more unit tests	4.1	3.4	4.6
Maintaining unit tests is difficult	3.3	3.5	3.2
Writing unit tests is difficult	3.2	3.7	2.9
I usually have sufficiently many unit tests	2.9	2.6	3.2
I enjoy writing unit tests	2.8	2.2	3.4

students, in this study. That is, students did not have a clear standard of high-quality test code in terms of effectiveness and maintainability.

> Challenge 2: Students considered determining what to check and identifying which code/scenarios to test are the most challenging aspect when writing new unit tests.

Table 4.2 summarizes the difficulties students faced (*Aspect*) when writing new unit tests. Both undergraduate and graduate students reported that the most challenging aspect is to determine what to check and identify which code/scenarios to test (*UGrad Rank* & *Grad Rank*). Meanwhile, students consider isolating the unit under test was the least difficult aspect of writing a new unit test.

One possible explanation for these challenges is that students feel like they are lacking tool support while writing tests. We found that 76% of undergraduate students (Rating = 3.4, *UGrad Rating* in Table 4.3) and 90% of graduate students (Rating = 4.6, *Grad Rating*) agreed that they would like to be better supported by tools during unit tests composition.

Overall, 85% of graduate students want more unit tests, 55% of them claimed that they enjoyed writing unit tests, and only 30% of them thought writing and maintaining unit tests is difficult. However, 76% of undergraduate students found writing and maintaining unit tests difficult and only 6% of them enjoyed writing unit tests. This demonstrates that students see value in unit testing, but also report that writing (19/37) and maintaining tests is difficult (21/37).

Challenge 3: Students found it challenging to determine when to stop testing.

After the study, the most common challenges students reported were understanding the source code implementation (17/54), and determining when to stop testing (14/54). Similarly, Aniche et al. [4] found that over 40% of students and teaching assistants considered “deciding how much testing is enough” a hard topic to learn.

Challenge 4: Students might not be sufficiently aware of the importance and difficulty of some testing activities.

As our surveys were adapted from industry, we noticed several disagreements on perceptions of unit testing among students and professional testers/developers. For example, students believed that the code coverage is one of the most important outcomes of unit tests, while prior work reports that “realistic test scenarios” are most important to professionals [42]—this aspect ranked last among graduate students. Moreover, while professionals consider isolating the unit under test somewhat challenging [42], students thought it was the least difficult aspect of writing a new unit test. These observations may suggest that students are not sufficiently aware of the importance and difficulty of some testing activities, and should be exposed to more complex testing scenarios.

4.1.4.2 RQ2(b): Challenges in Testing - Revealed by Student-authored Tests

Challenge 5: Students did not perform particularly well when it comes to discovery of the seeded faults.

For Implementation Project, students were informed that there was at least one bug in the source code for the Implementation Project (three in total), yet only half of the students (27/54) were able to detect one or more bugs. This low level of fault discovery may mean that the projects—which were designed for use in experiments involving professionals [54]–were too difficult for the students.

Table 4.4: Coverage achieved by students in the Design Project and the Implementation Project.

Coverage		**Overall (%)**		**UGrad (%)**		**Grad (%)**	
		avg	med	avg	med	avg	med
Design	Requirement	62.8	66.7	71.2	66.7	59.7	55.6
Implementation	Requirement	44.9	43.3	61.6	60.0	35.1	33.3
	Instruction	69.6	81.6	81.7	89.4	61.2	70.0
	Branch	49.9	55.1	61.1	66.7	41.9	47.7
	Mutation	43.6	47.4	56.6	60.3	33.9	29.5

Challenge 6: Students were better able to figure out how they want to test the code than actually test the code.

Table 4.4 reveals students' overall performance on designing tests (Row *Design*) and implementing tests (Rows *Implementation*). Students generally achieved higher requirement coverage in Test Design Project than Implementation Project given the higher average (62.8% vs. 44.9%) and median (66.7% vs. 43.3%). This points to a potential implementation barrier where students were better able to figure out how they want to test the code (design) than actually test the code (implementation).

While undergraduate students (*UGrad (%)*) performed consistently in requirements coverage across the two projects, graduate students (*Grad (%)*) did considerably better with the Design Project. We infer that graduate students may not be as comfortable coding compared to undergraduate students, an observation that is amplified by the other code-related metrics in the Implementation Project where the undergraduate students consistently outperformed the graduate students. These results suggest that unit testing experience and java experience (of which the undergraduate students had more, 3.3 years vs 2.0 years on average) correspond with higher coverage metrics.

Challenge 7: While students do not admit to having trouble with understanding specifications, their code suggests otherwise. Students frequently created test cases that mismatched the program specifications.

Relatively few students reported that they had trouble with actually creating unit tests that can accurately reflect their intended testing scenarios (4/54) or creating syntactically correct unit tests (3/54). However, their code and test designs tell a different story. We found eight participants (14.8%) created tests cases with incorrect syntax and were unable to fix the errors. Among 54 participants, 46 (85.2%) generated at least one test case whose testing scenario did not match the program specifications. As a result, 18.3% (145/794) of total tests contained a mismatch issue (e.g., not returning the position of encountered obstacles in Design Project per Req 6 in Section 4.1.3.2). We also observed 21 participants (38.9%) could not correctly create test cases for some intended scenarios. For example, asserting a game score without first adding the declared frames to the game. Similar student behaviors that misunderstanding the problem leads to flawed implementations were also observed in prior work [185].

4.1.4.3 RQ2(c): Pitfalls in Student-authored Tests

> Challenge 8: Students tend to only test the happy path, which represents a common misconception about the testing process, where the purpose is to verify expected behavior in addition to uncover faults.

We ran the student-authored tests against the test smell detector *tsDetect*[130] to identify potential smells in the test code, and then manually validate the smells using their definitions [47, 62, 116, 146] as well as classifications from prior work [62]. We identified three code-related smells (Smell 1 - 3); three test semantic/logic related smells (Smell 4 - 6); and three smells in test steps (Smell 7 - 9). Table 4.5 introduces these smells (*Test Smell*), their classifications (*Classification*), and number of participants whose unit tests had each smell and the number of corresponding smelly tests (*Occurrence*).

We found that participants were most likely to introduce smells related to test steps: there were 37 participants who ignored the setups which are provided in the test class. Twenty-two participants set assertions that mismatched the intended scenario. These findings suggest that students might not fully understand how each component in test class functions, such as the usage of "`@Before`" tests. In addition, students may have difficulties in correctly understanding the program specifications and embedding them in tests as some tests cannot accurately represent the testing scenarios. Furthermore, over 40% of participants tested the happy path only. This matches the observation in prior work [50] that students tend to write basic test cases to cover expected behaviors rather than detect hidden bugs. None of the `assertTrue` and `assertFalse` assertions contained a customized error message in this study.

The following introduces each test smell with a description and an example extracted from participants' test code:

Table 4.5: Test Smells Listed in Decreasing Order of Frequencies of Occurrence. Occurrence is listed in terms of participants (*par*) and tests impacted (*tests*).

Test Smell	**Symptom**	**Classification**	**Occurrence**
S7: Refused Bequest	Ignore fixtures/setups	Test Steps	37 par / 96 tests
S6: Happy Path Only	Test happy path only	Test Semantic/Logic	24 par / N/A
S9: Assertion Mismatch Scenario	The assertion mismatch the intended testing scenario	Test Steps	22 par / 37 tests
S2: Bad Naming	Test case name that is not reflect the testing intention	Code Related	20 par / 48 tests
S5: Assertion Roulette	Having several assertions in a test method that have no explanation	Test Semantic/Logic	13 par / 43 tests
S1: Test Redundancy	Having redundant unit tests	Code Related	13 par / 32 tests
S3: Lack Comments	There is no comment to help understanding the test case	Code Related	10 par / 43 tests
S8: No Assertions	A test contains no assertion	Test Steps	5 par / 6 tests
S4: Conditional Test Logic	Assertion(s) in conditional structures	Test Semantic/Logic	1 par / 1 test

Smell 1: Test Redundancy [97, 188]

Test cases that semantically or syntactically overlap and test the same thing with respect to other tests. Figure 4.4 has two tests that cover the happy path of the `BowlingGame.score()` and `BowlingGame.addFrame(Frame frame)` methods; when a frame is added to the game, the score is incremented. The participant wrote seven total assertions for this happy path scenario. However, this participant did not test 1) invalid frames, such as `(-1,11)`, 2) strike or spare frames that leads to bonus score, such as `(2,8)`, or 3) invalid number of frames of a game, such as 11 frames. This example test code also introduces *Test Code Duplication* smell [47] by containing code clones (lines 3-9 & lines 14-20). While the impact of duplicate code may not be felt in the moment, future maintainability would be more difficult.

Smell 2: Bad Naming [43, 146]

This smell comes from having a test case name that does not reflect the intention of the testing scenario. As suggested by Daka et al. [43], descriptive test case names allow testers/developers to easily find relevant unit tests, judge the importance of test failures without reading the source code, and easily understand the testing intention of a unit test.

Smell 3: Lack Comments [99]

This smell comes from the lack of comments in the test code, which makes a test case hard to understand. For example, there are no comment describing the testing scenario or pass/fail conditions in Figure 4.4 and Figure 4.5.

```java
@Test
public void testScore1() throws Exception {

    BowlingGame g = new BowlingGame();

    g.addFrame(new Frame(2, 7));
    assertTrue(g.score() == 9);
    g.addFrame(new Frame(7, 1));
    assertTrue(g.score() == 17);
    g.addFrame(new Frame(9, 0));
    assertTrue(g.score() == (17+9));
}

@Test
public void testScore2() throws Exception {

    BowlingGame g = new BowlingGame();

    g.addFrame(new Frame(2, 7));
    assertTrue(g.score() == 9);
    g.addFrame(new Frame(7, 1));
    assertTrue(g.score() == 17);
    g.addFrame(new Frame(9, 0));
    assertTrue(g.score() == (17+9));
    g.addFrame(new Frame(9, 0));
    assertTrue(g.score() == (17+9+9));
}
```

Figure 4.4: Test Redundancy & Bad Naming.

```java
@Test
public void testIsNextFrameBonus() throws Exception {

    BowlingGame bg = new BowlingGame();

    for(int k = 0; k < 11; k++) {
        assertFalse(bg.isNextFrameBonus());
        bg.addFrame(new Frame(1, 0));
    }

    assertTrue(bg.isNextFrameBonus());
}
```

Figure 4.5: Conditional Test Logic & Assertion Roulette & Assertion Mismatch Scenario.

Smell 4: Conditional Test Logic (Traceability) [116, 146]

This smell comes from having assertions in conditional structures. For example, a test case that uses loops to verify the contents of collections of objects, or `if` statements to steer execution to a fail statement or to avoid executing certain pieces of test code when the system under test fails to return valid data. The test `testIsNextFrameBonus()` in Figure 4.5 contains an `assertFalse` in the `for` loop to verify if the next frame being added to this game is a bonus frame, which makes the test hard to trace when the assertion fails.

Smell 5: Assertion Roulette [47, 116]

This smell is caused by: 1) verifying too much functionality in one test (also known as *Eager Test*), and/or 2) containing no assertion messages when there are more than one call to the same kind of `assert` method (also known as *Missing Assertion Message*) [47, 116]. The test `testIsNextFrameBonus()` in Figure 4.5 also exhibited this smell. This smell can be addressed by breaking up the test into a suite of single assertion tests (single condition test) or adding a unique message to each assertion.

Smell 6: Happy Path Only [62]

Test suites that only validate the expected behavior without considering the boundaries and exceptions. For example, in Figure 4.4, this participant only created these two unit tests, and both tested happy paths (i.e. expected results).

Smell 7: Refused Bequest [17, 47, 146]

This smell comes from ignoring the setups when creating a new test case. We only observed this smell in the Design Project since there is no provided setup in Implementation Project. For example, in Figure 4.6, the `void setUp()` method (lines 3-5, given as example test) sets up a planet with the size of 100×100 and has no obstacles. However, the test generated by a participant expects the rover to move over the edge of the planet from West and spawns on the opposite edge at position `(3,0)` and facing West (lines 11-13). That is, this participant ignored the size of the planet and created the testing scenario by tacitly defining the width of the planet as 4.

Smell 8: No Assertions [146]

This smell comes from having no assertions in a test case.

Smell 9: Assertion Mismatch Scenario (New)

Test cases that do not correctly assert the expected behavior given program specifications. When these tests run, they fail and create a false positive in that the code is correct and the test is wrong. For example, consider `testIsNextFrameBonus()` in Figure 4.5. As stated in

```java
// Create a 100x100 grid with no obstacles
MarsRover rover;

@Before
public void setUp() throws Exception {
    rover = new MarsRover(100, 100, "");
}

@Test
public void theRoverWrapsAroundLeft() throws Exception {

    /*
    * Scenario: A rover will wrap around at the left edge
    * Testing Input: "lf"
    * Expected Output: rover position: (3,0,W)
    */

    fail("Not yet implemented");
}
```

Figure 4.6: Refused Bequest.

the program specifications, a bowling game consists of ten frames, each with two throws, and up to two bonus throws if the tenth frame is a spare or strike. However, this test added 11 frames, which violates the rule of a bowling game. In addition, a frame `(1,0)` can never lead to a bonus throw since it is neither a spare nor a strike.

4.1.5 Takeaways

We summarize the challenges students encounter during unit testing, and share insights as to how we can better help students overcome these challenges:

Insufficient understanding of testing:

- **Challenge 1**: Students have no consensus on good unit tests.

  While we encourage students to evaluate the test code quality with coverage metrics, we should also inform students that coverage metrics are not always strongly correlated with fault detection [78]. The quality of the tests should also be adjudged against the original specification of the program, possibly by running the tests against a reference implementation [51].

- **Challenge 8**: Students tend to only test the happy paths, and hence

  Challenge 5: Students do not perform well regarding the discovery of faults.

  Emphasize the goal of testing. Being able to detect hidden bugs is as important as verify expected code behaviors in testing, and hence students should avoid only testing the happy paths.

Insufficient awareness of some challenges in testing:

- **Challenge 4**: Students are not sufficiently aware of the importance and difficulty of some testing activities.

  Instructors should present students with case studies and other evidence illustrating the importance of various aspects of testing. This may challenge the students' views and help them think more critically about testing. For example, while test execution speed was not seen as important by students, research at Google indicates it is of the utmost importance [53].

- **Challenge 3**: Students find it challenging to determine when to stop testing, and **Challenge 7**: Students do not realize the challenges in understanding program specifications, and create test cases that mismatch the program specifications.

  Instructors should emphasize the importance of understanding test oracles (the specifications). As for understanding when to stop testing, introducing a principled approach such as coverage metrics is a start, but coverage metrics are not always strongly correlated with fault detection [78]. Practicing testing techniques such as function-based input domain modeling (IDM) could help students understand and transfer the semantic information from project specifications to the IDM, and hence generate tests that reflect the expected code behavior.

Implementation barrier:

- **Challenge 2**: Students find it challenging to determine what and how to test.

- **Challenge 6**: Students are better able to design a test than implement the test.

  Instructors should encourage hands-on testing experience. Teaching students how to design tests to achieve various objectives (e.g., coverage) or use various techniques (e.g., equivalence class partitioning and boundary value analysis) is important to build confidence in testing techniques. It also demonstrates how to decide what to test. However, it should not replace the task of actually writing the test code.

4.1.6 Threats to Validity

Conclusion: Participants were required to work on these testing projects in a lab environment in a limited time, which could potentially impact their testing behaviors. For example, students may be inclined to take shortcuts, such as choosing test names out of convenience (e.g., `test1()`) or neglecting to write code comments. This may have artificially inflated the frequency of some smells such as *Bad Naming* and *Lack Comments*.

Moreover, the potential ambiguity in requirements may have led to the observation of the new smell *Assertion Mismatch Scenario*, and this smell might be specific to student code.

Construct: Sampling bias could be introduced in participant recruitment. Participants were self-selected into this study. All graduate students were from same university, same with the undergraduate students, which made the sample relatively homogeneous. In addition, graduate students came from different undergraduate programs in various countries, the objectives of courses that involves unit testing and software testing course in each program could vary greatly. Hence, our findings may not generalize to the rest of the population. A replication with a more diverse and larger set of students is needed.

Internal: In this study, we only observed students' interactions with two unfamiliar codebases. Students may perform differently on their own or peers' programs that they are more familiar with.

The design of surveys could possibly introduce implicit bias to students' responses. To minimize the potential influence, we adapt questions from prior work [42] to form our preliminary survey, and collect participants' demographic information only in post survey. Students' interpretation on survey questions and options may vary and potentially imprecise, which could bias the results as well.

External: The testing projects were relatively small, so the smells observed may not be representative of the smells that would occur when testing a larger codebase.

4.2 Study 2: The Impact of Test Case Design

This study received IRB approval from NCSU (Protocol Number #16607), and was published in VL/HCC '22 [11].

Gina R. Bai, Kai Presler-Marshall, Susan R. Fisk, and Kathryn T. Stolee. "Does Assertion Roulette Smell Bad? An Experiment from the Perspective of Testing Education". In: *2022 IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC '22. 2022

4.2.1 Study Rationale

Test smells are commonly perceived as having a negative impact on software maintainability and correctness. Research has shown that *Assertion Roulette* is the most pervasive smell in industrial and open-source systems. This smell is also observed in student-authored tests (Section 4.1). However, some recent studies argue that the impact of *Assertion Roulette* is not as severe as previously believed, and developers usually consider it acceptable (Section 4.2.2).

Yet, test smells can be frustrating, and the presence of the *Assertion Roulette* test smell may still have a negative impact on students in an educational setting. To assess the impact, we conducted a controlled empirical study with 42 CS students. Participants came from two populations: CS1 and graduate students, to see what role experience may have in terms of test smell's impact.

Students were tasked with implementing source code in Java that passes provided JUnit tests. Through analysis of student-authored source code and survey responses, we measured the impact of *Assertion Roulette* in three ways: first using code quality measures, second using testing behavior measures, and third using sociological measures of persistence intentions and self-assessed CS ability. To our knowledge, this the first study on the impact of *Assertion Roulette* on students' programming performance, testing behaviors, and self-assessed ability and persistence intentions in CS. It addresses the general research question RQ2, which is broken into three additional sub-questions:

RQ2: What barriers do students have when learning and practicing testing?

> **RQ2(d):** How does *Assertion Roulette* in the provided test suite impact students' programming performance?

RQ2(e): How does *Assertion Roulette* in the provided test suite impact students' testing behaviors?

RQ2(f): How does *Assertion Roulette* in the provided test suite impact students' self-assessed abilities and persistence intentions in CS?

4.2.2 Developer Perceptions of Assertion Roulette Smell

Table 4.6 summarizes chronologically the existing work related to *Assertion Roulette*, the most pervasive test smell in both human-written test code (e.g., [18, 100, 176]) and machine-generated test cases (e.g., [69, 123]). These papers were gathered from Google Scholar search results with the keyword "*Assertion Roulette*" and selected based on the relevance to this smell. The expanse of related work shows how researcher understanding of the smell has evolved over time. For example, researchers investigated why it occurs (e.g., [123, 176] from 2016), and how it negatively impact the system (e.g., [166] from 2018 and [32] from 2019). Researchers then explored the approaches to detect (e.g., [132] from 2020) and refactor (e.g., [151] from 2020) test smells with tools. They also explored how test smells were perceived (e.g., [32, 45] from 2019) and handled (e.g., [168] from 2020 and [91] from 2021) by developers.

While new approaches to manually [162] or automatically [151] detect and refactor *Assertion Roulette* have been recently introduced, developers commonly consider it as a low severity test smell [168], and they barely recognize [45] and refactor this smell [90, 91]. We also see that *Assertion Roulette* may have become outdated [125] and having *Assertion Roulette* may actually be an indicator of test robustness [129].

Most similar to our work in Section 4.2 is that of Buffardi et al. [29], which focuses on testing education by exploring the relationship between the existence of test smells in a student-authored test suite and test suite accuracy. By contrast, we provide students with a test suite and focus on the potential impact of the existence of test smells on students' programming and testing activities.

Table 4.6: Brief summaries on research findings related to *Assertion Roulette* in the selected test smell related papers from 2001 to 2021.

Paper title and reference	Venue	Conclusions related to Assertion Roulette
Refactoring Test Code [47]	XP '01	**Formally defined and introduced *Assertion Roulette***: "*This smell comes from having a number of assertions in a test method that have no explanation.*"

Continued on next page

Table 4.6 (continued).

Paper title and reference	Venue	Conclusions related to Assertion Roulette
Inspecting Automated Test Code: A Preliminary Study [100]	XP '07	*Assertion Roulette* is a common smell and test case overloading "**hampered the clarity and maintainability of tests**".
Scotch: Test-to-code traceability using slicing and conceptual coupling [141]	ICSM '11	The occurrence of *Assertion Roulette* is **common** since developers use assert statements to verify the testing environment before verifying the behavior of the tested classes.
Are test smells really harmful? An empirical study [18]	EMSE '15	*Assertion Roulette* is the **most frequent test smell** in both industrial and open source systems, and most of the test smells **frequently co-occur** with it. *Assertion Roulette* is one of the test smells that **more strongly impacts program maintenance**.
On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study [**TestSmells_AutoGenerated_2**]	SBST '16	*Assertion Roulette* frequently occur in test classes automatically generated by EvoSuite, and all test smells frequently co-occurs with Assertion Roulette, and it **has a strong correlation with the size of the system**.
An Empirical Investigation into the Nature of Test Smells [176]	ASE '16	"When testing **large and complex classes**, developers tend to create more complex test methods, exercising multiple methods of the tested class."
On The Relation of Test Smells to Software Code Quality [166]	ICSME '18	*Assertion Roulette* is associated with **highest change-proneness**.
Scented Since the Beginning: On the Diffuseness of Test Smells in Automatically Generated Test Code [69]	JSS '19	All automatic test case generation tools studied in this paper naturally **output a large amount** of *Assertion Roulette* smell.
Test-related factors and post-release defects: an empirical study [129]	ESEC/FSE '19	A high number of assertions (documented or not) **makes tests more robust**.
How the Experience of Development Teams Relates to Assertion Density of Test Classes [32]	ICSME '19	*Assertion Roulette* **impacts the readability/understandability/ maintainability** of test suites. Assertion density is important but is not the only aspect to consider for test code effectiveness.
Assessing Diffusion and Perception of Test Smells in Scala Projects [45]	MSR '19	**Few** developers were able to **identify** *Assertion Roulette*
Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities [125]	ICSME '20	Advances in the JUnit framework have removed the traceability confound entirely, and hence *Assertion Roulette* **has become obsolete**, which is further reinforced by its high degree of overlap with Eager Test."
Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality [68]	ICSME '20	"The presence of the Assertion Roulette smell **negatively impacts test quality perception**."
An Empirical Study on the Evolution of Test Smell [90]	ICSE-Companion '20	*Assertion Roulette* leads to a **high rate of churns**, and it **persists** in refactoring.
tsDetect: an open source test smells detection tool [90]	ESEC/FSE '20	This paper introduces a **test smell detector** that identify test smells including *Assertion Roulette*, with a precision of 94.74%.
Investigating Severity Thresholds for Test Smells [168]	MSR '20	Developers perceive *Assertion Roulette* as a "**low to medium impact on test suite severity**," and an "**acceptable trade-off** between maintainability and ease of writing."
Refactoring Test Smells: A Perspective from Open-Source Developers [162]	SAST '20	To refactor *Assertion Roulette*, most developers prefer to **separate assertions into single test cases**.
RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring [151]	SBES '20	This paper introduces a tool **specifically** for Assertion Roulette identification and refactoring.

Continued on next page

Table 4.6 (continued).

Paper title and reference	Venue	Conclusions related to Assertion Roulette
The secret life of test smells: an empirical study on test smell evolution and maintenance [91]	EMSE '21	Developers **did not remove** *Assertion Roulette* during test code refactoring. *Assertion Roulette* may have a higher correlation with the defect-proneness of source code files.

4.2.3 Methodology

We used an experimental, controlled study design, wherein we randomly assigned students, stratifying based on education, to a control group (AR-free group) or an experimental group (AR group). While the students in the AR-free group worked on a Java-based project without *Assertion Roulette* in the test suite, the students in the AR group worked on a Java-based project with a test suite that exhibited *Assertion Roulette*. Students were also instructed to complete two surveys (one preliminary survey and one post-activity survey) via Qualtrics, which included questions on students' self-assessed CS abilities and CS persistence intentions.

This study was conducted over six two-hour lab sessions that were held synchronously online via Zoom. Each student attended one lab session only. Students were instructed to complete the preliminary survey at least one day prior to the lab session. We cooperated with the instructors of the CS courses that we recruited students from to guarantee that there was no homework assignment or examination between the preliminary survey and study lab sessions. At the beginning of the lab session, students received a 20-minute introduction on the procedure of the study, the overview of the project, and a step-by-step live demo on project setup. Students were guided to a GitHub repository containing links to 1) the video version of the setup instructions, 2) the project, and 3) the post-activity survey. Students were allowed to consult any online resources for assistance.

Prior to the study, we conducted a pilot study (n = 5, including undergraduate and graduate students in CS) to clarify our survey questions and programming project, and to ascertain the potential duration of the lab session. As a result, we modified the survey questions to avoid ambiguity, and adjusted the project for a student population.

4.2.3.1 Surveys

We adapted the preliminary and post-activity surveys (Appendix B.2.1) from prior work that explore the impact of gender on career aspirations [39] and the impact of feedback on novice programmer engagement [109]. The survey consists of 7-point Likert scale questions on students' self-assessed abilities in CS (*Q1*, ranging from "Considerably below average" to "Considerably above average"), and their persistence intentions in CS: the likelihood of taking another course in CS (*Q2*, ranging from "Highly unlikely" to "Highly likely"), and the likelihood of minoring (*Q3*) or majoring (*Q4*) in CS. Demographics (i.e., education level, gender identity) were collected in the post-activity survey.

4.2.3.2 Implementation Task

We adapted the Java-based project *Bowling Score Keeper* task from a prior study [54] as it has been used in numerous other testing-related studies (e.g., [58, 59, 60, 174, 183]). Students were asked to implement a Java program that calculates the score of a single bowling game given a provided set of specifications and JUnit tests.

The original Bowling Score Keeper project consists of thirteen user stories; we shortened it down to the first seven so it could be completed in 90 minutes. Our shortened version of the Bowling Score Keeper project contains 37 lines of Java source code (skeletons for the three necessary classes) and 558 lines of Java test code (JUnit tests). Each user story had a separate test class, with three to sixteen test cases per user story. The *Assertion Roulette* test smell was intentionally seeded into the test code for the experimental group (AR group) by combining all test methods for each user story into a single test method (thus, giving a total of seven test methods). For example, the intended behavior in User Story 1 was:

> **User Story 1 - Frame:**
> Each turn of a bowling game is called a frame. Ten pins are arranged in each frame. The goal of the player is to knock down as many pins as possible in each frame. The player has two chances, or throws, to do so. The value of a throw is given by the number of pins knocked down in that throw.

The AR-free (control) version of the test class (Figure 4.7) contained test cases that each had a descriptive name that reflected the expected program behavior [168] and a single assertion [162]. For example, the test case `testFrameWithScoreIsCreated()` verifies that a frame consists of two chances/throws, and the test case `testExceptionMoreThan10PinsPerFrame()` verifies that an exception is thrown (indicated by the `expected` attribute of the `@Test` annotation) when trying to create a Frame with more than ten pins knocked down. Figure 4.8 shows the *Assertion Roulette* version of the test suite for User Story 1, with all assertions in one test case (lines 5, 8, and 19). We rewrote the test `testExceptionMoreThan10PinsPerFrame()` (lines 14-18 in Figure 4.7) with `try-catch` block (lines 10-16 in Figure 4.8), which served as an alternative to the `expected=Exception.class` to ensure that an `Exception` was thrown in the right place in the test case `test()`.

To verify the equivalence of the original and modified test suites, we used mutation testing, which serves as an approach to compare the bug-revealing capabilities of two different test suites (AR vs. AR-free versions). In particular, we confirmed there were no differences in the bug-revealing capability of both test suites via PITest [137], a commonly-adopted Java mutation testing tool. We considered both versions of the test suites equivalent as they killed the same set of mutants.

4.2.3.3 Development Environment

In order to track how student-authored source code evolved given test suite with or without the *Assertion Roulette* smell, we assigned each student a virtual machine image (VM, supported and managed by NCSU's virtual computing lab) with a standard Linux development environment: Ubuntu 18.04 and Eclipse Java 2020-06, Nano, Vim, and Gedit.

The VMs were set up with a script for students to run the test suite on their project

```java
public class US01 {
    @Test
    public void testEmptyFrameIsCreated(){
        final Frame f = new Frame();
        assertNotNull(f);
    }

    @Test
    public void testFrameWithScoreIsCreated(){
        final Frame f = new Frame(1, 2);
        assertNotNull(f);
    }

    @Test ( expected = Exception.class )
    public void testExceptionMoreThan10PinsPerFrame(){
        final Frame f = new Frame(12, 12);
        assertNotNull(f);
    }

    @Test
    public void testFrameIsCreatedWithCorrectName(){
        final Frame f = new Frame();
        assertEquals("Frame", f.getClass().getSimpleName());
    }
}
```

Figure 4.7: Example Test Class for User Story 1 (AR-free Group).

```java
public class US01 {
    @Test
    public void test(){
        Frame f = new Frame();
        assertNotNull(f);

        f = new Frame(1, 2);
        assertNotNull(f);

        try {
            f = new Frame(12, 12);
            fail();
        }
        catch (final Exception e){
            // Exception expected
        }

        f = new Frame();
        assertEquals("Frame", f.getClass().getSimpleName());
    }
}
```

Figure 4.8: Example Test Class for User Story 1 (AR Group).

implementation using Maven's Surefire test runner. After every run of the test suite, the script would display test results to the student and make a timestamped copy of the project, including the current code written and the results of running the test suite. Instructions for using the VM and running the script were included on both the VMs and the GitHub repository, and one of the authors gave a live demonstration in each lab session.

While the use of a script for building the projects and running the tests is a departure from the built-in Eclipse JUnit interface, it allowed us to take a snapshot of the entire project each time the test suite was run in order to study the evolution of the code and the types of changes made. To reduce the barrier from our testing script, we integrated the script with the Eclipse debugger so that students could debug the tests interactively. An example of the test results that students saw is shown in Figure 4.9. The script reported the number of tests run, the number of tests that failed (due to a failing assertion or unexpected exception), the overall status of the build, and detailed results on each test that failed.

4.2.3.4 Participants

We conducted this study with 49 students: 24 graduate students and 25 undergraduate students from North Carolina State University. The graduate students were recruited from a

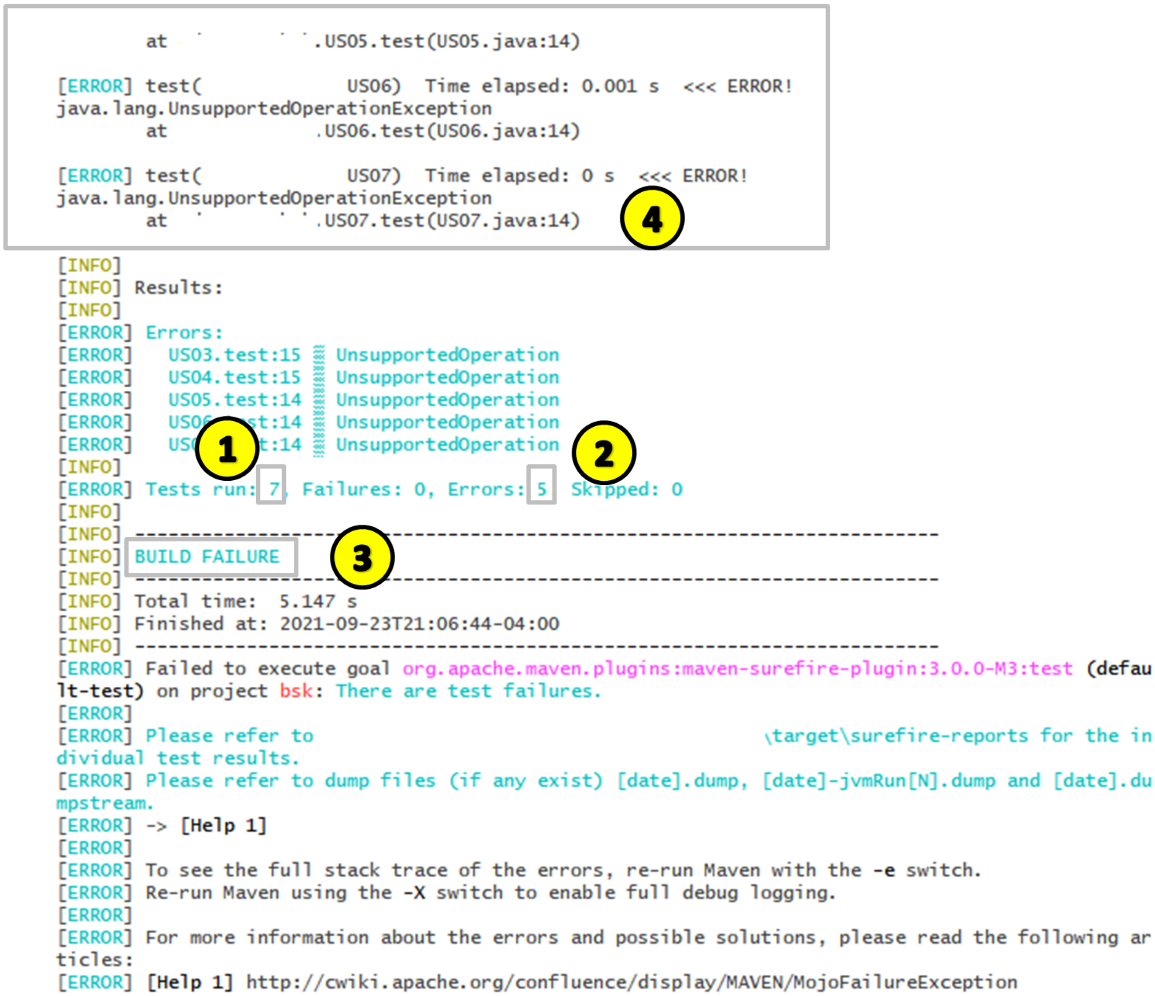

Figure 4.9: Sample output showing the the results of running our build script. The output shows the number of tests run (1) and the number of tests that failed (2). Because at least one test failed, the script marked the build as a failure (3). Detailed results of each failing test are shown, partially cut off, in (4).

graduate-level software testing course. All students in this course were required to complete this study as an in-class workshop to receive credit for the class activity, but were allowed to opt out of having their data analyzed for this study. The undergraduate students were recruited from a CS1-level Java programming course and were eligible to receive extra credit upon completion of the study.

After discarding data from students who did not follow the instructions, we were left with data from the remaining 42 students for analysis. Overall, there were 18 students in the AR group (11 CS1s, 7 grads), and 24 students in the AR-free group (12 CS1s, 12 grads). We did not break down the analysis by gender (AR group: 5 female & 13 male, AR-free group: 5 female & 19 male) due to the small sample size.

4.2.4 Data Analysis

We collected students' timestamped implementations from their assigned VMs upon completion of the study. In total, 42 students generated 537 timestamped copies of the project (AR group: 220 copies, AR-free group: 317 copies) by running test suite. The group condition was concealed during data analysis.

4.2.4.1 Surveys

We converted text Likert scale to numbers, where 1 maps to the lowest score, such as "Considerably below average", and "Highly unlikely", and 7 maps to the highest score, such as "Considerably above average", and "Highly likely". We treat them as interval-scaled data [73].
We report students' **self-assessed abilities in CS** via:

- **csAbilityDiff**:
  The difference of response to *Q1: ability in CS* (Appendix B.2.1) in the post-activity survey from the response to *Q1* in the preliminary survey. For example, if a student considers themselves to be "moderately above average" (score maps to number 6) in the preliminary survey and evaluate themselves to be "considerably above average" (score maps to number 7) in the post-activity survey, the metric *csAbilityDiff* was calculated as $7-6=1$.

As graduate students have already committed to pursuing degrees in CS, we only investigate the impact of *Assertion Roulette* on the persistence intentions of CS1 students in this study. We measure CS1 students' **persistence intentions in CS** with three metrics–*anotherCourseDiff*, *minorDiff*, and *majorDiff*. These three metrics are evaluated separately given a Cronbach's alpha[1] of –0.29:

- **anotherCourseDiff**:
  The difference of response to *Q2: take another course in CS* (Appendix B.2.1) in the post-activity survey from the response to *Q2* in the preliminary survey.

[1] Cronbach's alpha measures the reliability, or internal consistency, of a scale [92].

- **minorDiff**:
  The difference of response to *Q3: minor in CS* (Appendix B.2.1) in the post-activity survey from the response to *Q3* in the preliminary survey.

- **majorDiff**:
  The difference of response to *Q4: major in CS* (Appendix B.2.1) in the post-activity survey from the response to *Q4* in the preliminary survey.

4.2.4.2 Test-Driven Development Project

We collected students' timestamped implementations from their assigned virtual machines upon completion of the study.

We measured students' **programming performance** in terms of accuracy and effectiveness. We adapted the metrics from prior work, and we evaluated the accuracy via pass rate at both assertion and user story level (*passRate_Astn*, *passRate_US*) [8, 89, 184], and the improvements that students made between two consecutive test suite runs (*passRateChange*) [8]. A detailed description of metrics follows:

Accuracy:

- **passRate_Astn (%)**:
  The pass rate at the assertion level. In JUnit, a test case will stop executing when an assertion fails or when an `Exception` is thrown. Consequently, when a test case has multiple assertions, it can be difficult to ascertain precisely which statements would have passed, as the failure of one assertion will skip the execution of all subsequent statements. To make the analysis consistent across the AR and AR-free versions of the project, we ran all student-authored projects against the AR-free version of the test suite, where almost all test cases have just a single assertion. We then marked whether each assertion **P**assed, **F**ailed, or the status was **U**nknown (because a prior assertion failed or an `Exception` was thrown by the student code). The metric *passRate_Astn* was calculated as the percentage of passed assertions out of all 49 assertions.

- **passRateDelta (%)**:
  The differences in assertion-level pass rate between two consecutive test suite runs. This metric was calculated as $passRate\_Astn_{x+1} - passRate\_Astn_x$, where $x \geq 1$. For example, if a student achieved an assertion-level pass rate of 8.2% on the second test suite run, and then achieved an assertion-level pass rate of 14.2% on the third test suite run, the *passRateChange* was logged as +6.0% for the third test suite run.

- **passRate_US (%)**:
  The pass rate at the user story (test class) level. In order to explore the impact of less granular feedback resulting from *Assertion Roulette* in the provided test suite, the pass rate was also calculated at the user story level. This metric was calculated as the percentage of fully passed test classes out of seven total.

The effectiveness of students' programming activities was measured with LOC deleted from (*LOCdeleted*) or added (*LOCadded*) to the project between two consecutive test suite runs [134], the total time spent on the project (*totalDuration*), and the time interval between two consecutive test suite runs (*runInterval*, adapted from prior work [80, 89]). A detailed description of metrics follows:
Effectiveness:

- **totalDuration (minutes)**:
  The total time that students spent on the project. Every student was instructed to follow the live demo of project setup and make an initial test suite run with no implementation. We considered the timestamp of this initial test suite run as the starting point, and the timestamp of the last test suite run as the end point. The metric *totalDuration* represents the difference between these two timestamps.

- **LOCdeleted (count)**:
  The number of LOC deleted from the project between two consecutive test suite runs. We created a Git commit for every timestamped project generated by students, and then extracted the number of LOC deleted from the project between two consecutive test suite runs via `git diff`.

- **LOCadded (count)**:
  The number of LOC added to the project between two consecutive test suite runs. As above, we extracted the number of LOC added to the project between two consecutive test suite runs via `git diff`.

- **runInterval (minutes)**:
  The time interval between two consecutive test suite runs. The metric was calculated as the difference between the timestamps of two consecutive chronologically-ordered projects, using the timestamped copies of the project saved for each run of the test suite.

We adapted metrics from prior work to measure students' **testing behaviors** in terms of testing effort and process. We report students' testing effort with the total number of test suite runs (*totalTestRun*) [8]. We quantify their testing process via how early (*testRun_-First*) [89, 135] and how often (*testRun_Frequency*) [89] they start testing. A detailed description of metrics follows:

Testing effort:

- **totalTestRun (count)**:
  The total number of test suite runs performed during the study. This metric was calculated as the number of timestamped projects minus one (the initial test suite run during the setup process).

Testing process:

- **testRun_First (minutes)**:
  The timestamp of the second test suite run. The metric represents how early students start running the test suite. The timestamp of the second test suite run is adopted as students were instructed to run the test suite once during the setup process to ensure their environment was working properly.

- **testRun_Frequency (one test suite run per X minutes)**:
  The frequency of test suite runs. This metric represents how often students run the test suite, and was calculated as the ratio of *totalDuration* to *totalTestRun*.

4.2.4.3 Statistical Analysis

Treating each of the 14 metrics from Section 4.2.4.1 and Section 4.2.4.2 as a dependent variable, we measured the impact of the following independent variables: $isAR$ for comparing the AR group to the AR-free group, and $isCS1$ for the level in school. We used a two-way ANOVA analysis to explore the impact of each variable independently as well as their interaction ($isAR * isCS1$). We report the p-values in Table 4.7.

4.2.5 Results

We analyzed the timestamped projects and survey responses generated by students in the experimental (AR) group and the control (AR-free) group. We explored the impact of having *Assertion Roulette* in the provided test suite on students' programming performance,

Table 4.7: ANOVA results for comparing the AR group and the AR-free group.

Dependent Variable		Independent Variable		
		$isAR$	$isCS1$	$isAR * isCS1$
Programming Performance	1) passRate_Astn	0.3663	0.0005 ***	0.4542
	2) passRateDelta	0.5852	0.0417 *	0.4515
	3) passRate_US	0.4449	0.0017 **	0.8521
	4) totalDuration	0.7200	0.1500	0.5010
	5) LOCdeleted	0.6450	0.2380	0.6380
	6) LOCadded	0.1390	0.2420	0.2270
	7) runInterval	0.1770	0.0885	0.0678
Testing Behaviors	8) totalTestRun	0.6420	0.3630	0.7000
	9) testRun_First	0.0282 *	0.0058 **	0.0714
	10) testRun_Freq	0.1957	0.0726	0.0662
Self-assessment	11) csAbilityDiff	0.5100	0.6420	0.5350
Persistence Intention	12) anotherCourseDiff	0.0817	0.0657	0.0787
	13) minorDiff	0.9260	0.6030	0.7730
	14) majorDiff	0.8053	0.0196 *	0.0064 **

*$p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

testing behaviors, and their self-assessed abilities in CS. We also discussed the impact of this smell on CS1 students' persistence intentions in CS.

4.2.5.1 RQ2(d): No Statistically Significant Impact on Code Quality Measures

Table 4.8 shows the accuracy metrics of student-authored source code, including the percentage of passed assertions and fully passed user stories (*passRate_Astn*, *passRate_US*), and the progress in terms of the improved assertion-level pass rate between two consecutive test suite runs (*passRateDelta*). Table 4.8 also contains the metrics that reflect effectiveness: the time that students spent on this project (*totalDuration*), the number of LOC changed in two consecutive test suite runs (*LOCdeleted*, *LOCadded*), and how long it took students to edit the code and test it again (*runInterval*). For example, on average, CS1 students in the AR group were able to pass 56.0% of assertions in 76.9 minutes, and during development they ran the project every 3.3 minutes and passed 2.5 more assertions each time (*passRateDelta* of 5.2%) by deleting 1.8 lines of code and adding 5.8 lines of code to the project. Overall, they fully passed the test cases for four user stories out of seven (57.1%). Across the entirety of the project, students added an average of 28.4 lines of code to the project.

We found *Assertion Roulette* had no statistically significant impact on all programming performance measures (Variables 1-7 in Table 4.7). These results indicate students in the

Table 4.8: Measuring students' programming performance in terms of accuracy and effectiveness.

Metrics		Group	Overall		CS1		Grad	
			avg	med	avg	med	avg	med
Accuracy	passRate_Astn (%)	AR	71.1	90.8	56.0	42.9	94.8	91.8
		AR-free	78.7	91.8	65.7	78.6	91.7	91.8
	passRateDelta (%)	AR	+6.0	0.0	+5.2	0.0	+7.0	0.0
		AR-free	+6.1	0.0	+5.2	0.0	+6.9	0.0
	passRate_US (%)	AR	68.6	85.7	57.1	42.9	87.1	85.7
		AR-free	75.7	85.7	61.4	78.6	90	85.7
Effectiveness	totalDuration (min)	AR	71.4	68.0	76.9	77.9	62.7	60.2
		AR-free	69.1	70.3	71.9	79.8	66.3	64.4
	LOCdeleted (count)	AR	1.9	1.0	1.8	1.0	2.0	1.0
		AR-free	1.7	1.0	2.0	1.0	1.5	0.0
	LOCadded (count)	AR	5.5	2.0	5.8	2.0	5.1	2.0
		AR-free	4.2	2.0	4.7	2.0	3.8	1.0
	runInterval (min)	AR	6.4	3.3	7.6	3.6	4.8	2.9
		AR-free	5.7	3.4	6.2	4.2	5.2	3.0

AR-free group achieved no higher pass rates and programmed no more effectively than those in the AR group. However, graduate students performed significantly better than the CS1 students in terms of accuracy (Variables 1-3 in Table 4.7).

4.2.5.2 RQ2(e): Statistically Significant Impact on Testing Behavior Measures

We present the metrics used for analyzing the testing behaviors in Table 4.9, including the number of test suite runs that students made during the study (*totalTestRun*), when the students started testing the code (*testRun_First*), and how often they tested their code (*testRun_Freq*). For example, on average, students in the AR group tested their source code every 8.3 minutes and they tested 12.2 times in total, while the students in the AR-free group ran the test suite every 6.4 minutes and they tested 13.2 times in total. The presence of *Assertion Roulette* had no statistically significant impact on both metrics *totalTestRun* and *testRun_Freq*.

Table 4.9 shows that students in the AR-free group started testing their source code in 9.4 minutes on average, and students in the AR group started testing after 18.7 minutes. The difference was statistically significant ($p = 0.03$ for *testRun_First*, Table 4.7). Along with the observation in Section 4.2.5.1 that students in the AR group achieved slightly lower pass rates, this result supports previous findings [89] that students who test earlier are more likely to have higher-quality implementations. We saw the greatest impact here on CS1

Table 4.9: Measuring students' testing behaviors in terms of testing effort and testing process.

Metrics		Group	Overall		CS1		Grad	
			avg	med	avg	med	avg	med
Testing Effort	totalTestRun (count)	AR	12.2	12.5	11.1	11.0	14.0	14.0
		AR-free	13.2	11.5	12.6	11.0	13.8	13.0
Testing Process	testRun_First (min)	AR	18.7	12.4	26.9	18.9	5.9	5.5
		AR-free	9.4	6.9	12.2	11.6	6.6	4.3
	testRun_Freq (one test run per X min)	AR	8.3	5.3	10.5	7.4	4.7	5.0
		AR-free	6.4	5.5	6.6	6.0	6.2	5.1

students (26.9 min vs. 12.2 min). It is possible that these students, who are less familiar with how to break up a task into smaller pieces, erroneously aimed for passing an entire test case at a time, instead of focusing on one assertion within at a time. Hence, having few large test cases may have discouraged novices from testing early.

4.2.5.3 RQ2(f): No Statistically Significant Impact on Short-term Sociological Measures

Students' self-assessed abilities in CS:

In this study, we found that the presence of *Assertion Roulette* in the provided test suite had no statistically significant impact on changes to students' self-assessed CS abilities given the p-value of 0.51. Additionally, we found the median change for both AR group and AR-free group was zero.

CS1 students' persistence intentions in CS:

We also explored the impact of this test smell on CS1 students' persistence intentions in CS, and the results showed that the median change for both groups across all CS persistence-related questions was zero. It was expected that the study task affected CS1 students differently than graduate students in terms of likelihood of majoring in CS (significant interaction effect of $isAR$ and $isCS1$ on *majorDiff*, $p = 0.006$ in Table 4.7) as graduate students had committed to CS. We concluded that the presence of *Assertion Roulette* in the provided test suite had no impact on persistence intentions in CS for the typical CS1 students in our study, but due to the small sample size, this finding may not generalize.

4.2.6 Takeaways

We found that the impact of *Assertion Roulette* on both CS1 and graduate students in this study was minimal. Along with the findings in recent prior work [29] that the presence of

Assertion Roulette in student-authored test code did not impact test accuracy, we conjecture that *Assertion Roulette* may no longer smell bad to students.

It is noteworthy that having few large test cases discouraged students from testing early. This implies that students, particularly those who are less familiar with how to break up a task into smaller pieces, may treat an entire test case as a milestone, and aim for passing an entire test case instead of focusing on one assertion within at a time. We encourage instructors to discuss how each component in test class functions, as well as how to interpret the assertions based on the program requirements and source code. To encourage novices to test early, we encourage avoiding this smell when providing them with test cases.

4.2.7 Threats to Validity

Conclusion: The statistically insignificant results might result from the small sample size ($n = 42$) [171].

Construct: Students were required to work on these tasks individually and remotely in a limited time, which could potentially impact their behaviors. Additionally, students were recruited with two different incentives–graduates participated as a required class activity, while the CS1 students self-selected into this study and participated for extra credit. A replication with a more consistent participant recruitment, and a more diverse and larger set of students is needed.

Internal: While students were given the option to complete the TDD activity within the Eclipse IDE, they were required to compile and run the project with a provided script via the command line outside of Eclipse. Compared to the Eclipse JUnit runner, this may impact students' programming performance and particularly testing behavior. However, we discarded data from students who did not follow the instructions, so the impact should be equal on all students' data.

External: The conclusions were drawn based on students' performance and behaviors when using Java and JUnit, which may not generalize to other languages.

4.3 Chapter Summary

Overall, these two studies pointed out that **students desire support in identifying *what code to test* and determining *which scenarios to be tested***, and revealed four barriers that students encountered when learning and practicing unit testing:

1. **Implementation barrier** (Section 4.1). This barrier occurs when students struggle with implementing syntactically correct (i.e., tests that can be compiled and executed as JUnit in this study) and semantically correct (i.e., testing scenarios that match the program requirements) test cases.

2. **Unclear of the goal of testing** (Section 4.1). This barrier leads students to only test the happy paths, and not attempt to uncover the faults.

3. **Lack of exposure to testing tools** (Section 4.1). This barrier occurs when students do not adopt existing testing tools due to unawareness/unfamiliarity of the tools or difficulties in using the tools. For example, students self-reported their challenges in determining when to stop testing (Section 4.1), but we observed no one adopting the coverage tools (e.g., EclEmma) that could help.

4. **Lack of hands-on testing practices** (Sections 4.1 and 4.2). This barrier impacts students in various ways, such as 1) having difficulties in understanding the source code (Section 4.1), 2) being able to better design a test than implement the test (Section 4.1), and 3) erroneously aiming for passing an entire test case instead of focusing on one assertion within at a time (Section 4.2).

CHAPTER

5

IMPROVING TESTING EDUCATION WITH CHECKLISTS

In response to the barriers that students encountered when developing (Chapter 3) and testing (Chapter 4) software, we designed a checklist intervention that supports students learning testing. This chapter presents a useful lightweight checklist that does as well as the state-of-the-practice (Section 5.1), and an evolved version of the checklist that contains explicit testing strategies that assists students in determining how to test code (Section 5.2). We summarize the insights on tool support in testing education in Section 5.3. Additional details and study materials for these experiments can be found in Appendix C.

These studies address the bolded parts of my thesis statement:

> **Motivated by the identification of barriers** students have during software engineering activities, **in particular software testing**, we show that a **lightweight checklist intervention improves student testing practices**.

5.1 Study 1: Testing Checklists

This study received IRB approval from NCSU (Protocol Number #18025), and was published in ITiCSE '22 [12].

Gina R. Bai, Kai Presler-Marshall, Thomas Price, and Kathryn T. Stolee. "Check It Off: Exploring the Impact of a Checklist Intervention on the Quality of Student-written Unit Tests". In: *27th ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '22. 2022

5.1.1 Study Rationale

Educators have sought to establish and enhance students' testing skills in recent years [153], including integrating testing in CS1 courses [66, 74, 164], and introducing tools to support students learning testing [24, 38, 164].

However, students often run into trouble when they are learning testing. For example, students make mistakes such as missing boundary testing [4, 20], writing smelly tests [13, 29], testing happy paths only [13, 20], and not testing the program until they are already done with development [135]. Chapter 3 also pointed out that students *wanted more tool support for unit test composition*, and the two most difficult aspects of unit testing were reported to be, "*Determining what to check*" and "*Identifying which code/scenarios to test*".

In response to this, we present a lightweight testing checklist that contains both tutorial information, which addresses students' needs on code examples (Chapter 3), and testing strategies to guide students in what to test and how to test. To assess the impact of the testing checklist, we conducted an experimental, controlled A/B study with 32 undergraduate and graduate students. The study task was writing a test suite for an existing program. Students were given either the testing checklist (the experimental group) or a tutorial on a standard coverage tool with which they were already familiar (the control group). The application of checklists to software testing is novel, and this study addresses the general research question RQ3, which is broken into two sub-questions:

RQ3: How to improve students' testing practices with checklists?

> **RQ3(a):** Do students who see the checklist write better test code than those who do not?
>
> **RQ3(b):** How do students engage with the checklist?

Table 5.1: The testing checklist used in our study. Each checklist item is inspired based on issues observed in prior work, and those mappings are in the *Addressing issues of...* column.

Addressing issues of...	**Test Case Checklist**
	Each test case *should*:
1. Having syntax errors [13]	☐ be executable (i.e., it has an `@Test` annotation and can be run via "Run as JUnit Test")
2. Having no assertions [4] Testing happy paths only [50]	☐ have at least one assert statement or assert an exception is thrown. Example assert statements include: `assertTrue`, `assertFalse`, and `assertEquals` (click for tutorials). For asserting an exception is thrown, there are different approaches: `try...; fail(); catch(Exception e) assertThat...;`, `@Test(expected = exception.class)` in JUnit 4, or `assertThrows` in JUnit 5 (click for tutorials).
3. Having eager test [4]	☐ evaluate/test only one method
	Each test case *could*:
4. Bad naming tests [4]	☐ be descriptively named and commented
5. Unfamiliarity with test class features [4, 13]	☐ If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using `@Before`)
6. Assertion Roulette [29]	☐ If there are too many assert statements in a single test case (e.g., more than 5), you might split it up so each test evaluates one behavior.
	Test Suite Checklist
	The test suite *should*:
7. Low requirements coverage [13]	☐ have at least one test for each requirement
8. Ignoring setup [13]	☐ appropriately use the setup and teardown code (e.g., `@Before`, which runs before each `@Test`)
9. Misinterpretation of failing tests [139]	☐ contain a fault-revealing test for each bug in the code (i.e., a test that fails)
10a-d. Incomplete test sets & Testing happy paths only [4, 13, 20]	☐ For each requirement, contain test cases for: ☐ Valid inputs ☐ Boundary cases ☐ Invalid inputs ☐ Expected exceptions
	To improve the test suite, you *could*:
11. Low code coverage [13, 20]	☐ measure code coverage using an appropriate tool, such as EclEmma (installation, tutorial). Inspect uncovered code and write tests as appropriate.

5.1.2 The Testing Checklist

We designed our checklist to address the challenges that students face during testing, based on prior work [13, 20, 135, 139, 153]. The baseline assumption is that students have received some education in software testing but need reminders on how to handle things such as the syntax for expecting exceptions. This design philosophy makes sense in our context, where the checklist is used for upper-level undergraduate or graduate students.

Each item in the checklist (shown in Table 5.1) maps to problems students encounter during software testing (the *Addressing issues of...* column). It contains tutorial information and testing tips in two sections, one for individual test cases (*Test Case Checklist*) and one for the entire test suite (*Test Suite Checklist*). Within each section, there are two lists, one of things the test case/suite *should* do, representing the most essential elements, and one that each test case/suite *could* do, representing best practices. These are intended to represent hard and soft requirements, respectively, for quality tests.

Unlike prior work [38], the checklist does not educate students on how to use testing techniques such as boundary value testing, but rather serves as a reminder to consider various testing approaches, such as invalid values, during the testing process. As the cognitive load associated with learning a new tool contributes to students' negative attitude towards testing [28, 88, 135, 153], the checklist is deliberately lightweight and provides testing support without a steep learning curve. We sought to address the following issues that students have had during software testing:

Issue 1: Tests have syntax errors, or that lack assertions [4, 13].
Solution 1: To ensure a test has fault-finding capabilities, we remind students to ensure the test executes (item #1) and includes at least one assert statement (item #2).

Issue 2: Tests have smells, such as *Bad Naming* [4, 13], *No Assertions* [4, 13], or covering the *Happy Path Only* [4, 13, 20].
Solution 2: Students are encouraged to write concise tests or split up larger ones to help avoid smells such as Eager Tests (item #3) and Assertion Roulette (item #6). We prompt the students to use appropriate test class features to reduce redundant code with items #5 and #8. We also ask students to name the tests descriptively (item #4) to improve readability and maintainability.

Issue 3: Stopping testing preemptively, for example, after designing a test for a presumed bug in the code [20].
Solution 3: The Test Suite Checklist encourages students to think beyond an individual test, about testing as a collection of tests (e.g., a test for every requirement, item #7). It also reminds students that each bug should have a fault-revealing test (item #9).

Issue 4: Test suites insufficiently cover boundary values and other aspects of program requirements [4, 20].

Solution 4: Items #10a-d remind the students to consider testing requirements with a variety of different techniques, including equivalence class partitioning (e.g., valid and invalid inputs) and boundary value analysis (i.e., boundary cases). Additionally, item #11 encourages students to use code coverage to identify untested code.

Issue 5: Misinterpreting failing tests, or modifying the test to remove the appearance of a failure [139]. Prior work [74] also suggests that students may not realise a test failure implies a bug, and that students *"rarely saw how test failures can help find bugs in [an] implementation"*.

Solution 5: In the Test Suite Checklist, we point out that a test that reveals the existence of bugs should fail (item #9).

5.1.3 Methodology

To assess the value of the checklist, we conducted an experimental, controlled A/B testing study with both undergraduate and graduate software engineering students. Students were given one preliminary survey and one post-activity survey via Qualtrics, and one Java-based unit testing project to perform in the Eclipse IDE.

After stratifying the participants based on level (undergraduate vs. graduate), we randomly assigned students to an experimental group (the Checklist group), in which students receive the task description along with a unit testing checklist, or to a control group (the Coverage group), in which students receive the task description along with a tutorial on EclEmma [48], a widely-used Eclipse plug-in, which measures Java code coverage, that students had used in previous course assignments.

As shown in Figure 5.1, this study was conducted in a two-hour lab, held synchronously online via Zoom. To start, students received a 15-minute introduction on the procedure of the study and an overview of the unit testing project, including the requirements and a walk-through of example tests. Students were shown a GitHub repository containing links to the surveys and the unit testing project and then assigned to Zoom breakout rooms to work individually. Students were allowed to consult online resources.

5.1.3.1 Participants

We conducted this study with 32 students: 23 undergraduate students and nine graduate students. All students were enrolled in a software engineering course at North Carolina State University. Students were eligible to receive extra credit upon completion of the study.

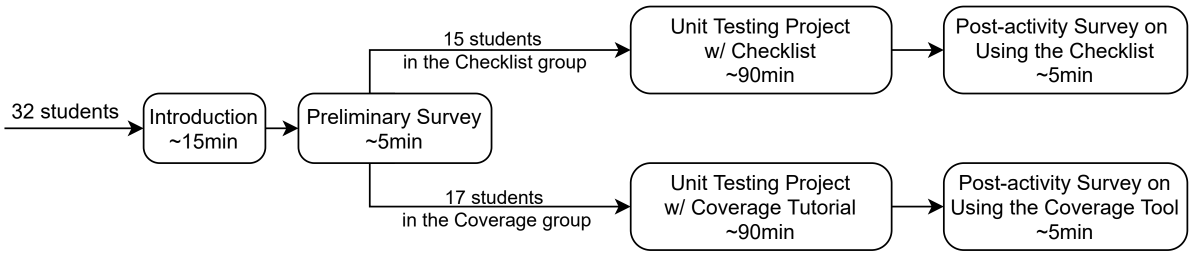

Figure 5.1: Study Procedure.

Table 5.2: Students' self-reported experience with Java & unit testing in years (numeric response), and prior experience with unit tests (binary yes/no response).

Group		**avg_yrJava**	**avg_yrUT**	**with_expUT**
Checklist	Undergrad (11)	3.5	3.0	11/11
	Graduate (4)	0.1	0.4	2/4
	Overall (15)	2.6	2.3	13/15
Coverage	Undergrad (12)	3.8	2.7	12/12
	Graduate (5)	1.4	1.4	5/5
	Overall (17)	3.1	2.3	17/17
Overall		2.8	2.3	30/32

Table 5.2 presents participants' self-estimated experience in years with Java (*avg_yrJava*) and unit testing (*avg_yrUT*), as well as if they have prior experience with unit tests (*with_expUT*). Overall, students have an average of 2.8 years of programming experience in Java, and 2.3 years of experience in unit testing. The majority of students have experience with unit tests (30/32). On average, students in the Checklist group have slightly less experience in Java and very similar experience with unit tests than students in the Coverage group. The undergraduate students are more experienced than the graduate students.

5.1.3.2 Screen Capture

Students were instructed to screen record their testing process, including browser activities, but technical difficulties prevented the capture of complete videos for many students. We were able to retain 19 videos, eight from the Checklist group and eleven from the Coverage group.

5.1.3.3 Surveys

We adapted the preliminary and the post-activity surveys from prior work [13]. The preliminary survey asks about students' prior unit testing experience. The post-activity survey asks students for their experiences with the coverage tool (e.g., if and how they used it, and how helpful they found it) or checklist (e.g., if and how they used it, which checklist items they found most helpful, and what, if anything, they would change), and collects brief demographics information (e.g., prior experience with Java and testing).

5.1.3.4 Unit Testing Project

We adapted the TDD project - Bowling Score Keeper used in in multiple prior testing-related studies (e.g., [13, 60, 183]). Students were expected to create JUnit tests to verify the behavior of an implemented program that calculates the score of a single bowling game given 1) the program requirements, 2) the source code with three malfunctioning methods (three classes, total lines of code = 86), and 3) two sample tests. These three faults are intentionally seeded into the implementation and could be revealed by testing unhappy paths through the code. While students were instructed to test programs implemented by others [5, 139], they were *not* expected to fix the bugs or modify the source code.

5.1.4 Data Analysis

In total, 32 students completed 64 surveys and wrote 297 test cases (Checklist group: 135 test cases, Coverage group: 162 test cases). To analyze the surveys responses and the quality of student-authored test code, we adopted the same metrics as prior work [13], as described in Sections 5.1.4.1 and 5.1.4.2. The group condition was concealed during data analysis.

5.1.4.1 Surveys

For rating questions, we converted 5-point Likert scale to numbers and treat them as interval-scaled data [73], where 1 maps to the lowest score, “Not at all helpful”, and 5 maps to the highest score, “Extremely helpful”.

5.1.4.2 Unit Testing Project

We measured the test code quality from three aspects:

Completeness:

Measured using **requirements coverage**, **instruction coverage** and **branch coverage**. We manually measure the requirements coverage with the program specifications. The provided example tests introduce a *baseline requirements coverage of 7.7%*. We adopt EclEmma to measure the instruction coverage (*baseline = 20.3%*) and branch coverage (*baseline = 13.6%*).

Effectiveness:

Mutation testing measures the effectiveness of a test suite by injecting a single fault at a time (known as a mutant) and re-running the test suite to see if it is detected (i.e., if a test fails). The **mutation coverage** (*baseline = 11.5%*) is the percentage of mutants caught, or killed, by the test suite. We use the mutation coverage and the **number of identified bugs** as the two metrics to represent test suite effectiveness. We use PITest [137] for mutation testing, and we manually identify the number of seeded bugs revealed by student-authored tests. As the students wrote test code and did not modify the system under test, the set of mutants produced by PITest was the same for all students.

Maintainability:

Measured using **test smells**. We adopt the definitions of test smells from prior work [13] and manually identify the smells in student-authored test code. No inter-rater reliability was considered in this process as only one author coded the smells.

5.1.4.3 Statistical Analysis

Treating the six metrics from Section 5.1.4.2 as dependent variables, we measured the impact of the following independent variables: $isChecklist$ for comparing the coverage group to the checklist group, and $isGrad$ for the level in school. We used a two-way ANOVA analysis to explore the impact of each variable independently as well as their interaction ($isGrad*isChecklist$). While this is less resilient to the non-normal tendencies in some of our data than a Kruskal-Wallis ANOVA, it is necessary to understand both factors and their interaction. We report the p-values in Table 5.3.

5.1.5 Results

We compare the testing performance of students in the experimental (Checklist) group and the control (Coverage) group, and report how students engage with the checklist.

Table 5.3: ANOVA results for comparing the use of the testing checklist and the coverage tools.

Dependent Variable	Independent Variable		
	$isGrad$	$isChecklist$	$isGrad * isChecklist$
1) Requirements Cov	0.0349*	0.6919	0.4528
2) Instruction Cov	0.0149*	0.7346	0.3886
3) Branch Cov	0.0036**	0.7974	0.6120
4) Mutation Cov	0.6990	0.1400	0.5380
5) #Identified Bugs	0.0598	0.8591	0.7200
6) #Smelly Tests	0.2880	0.8170	0.5810

$^*p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

5.1.5.1 RQ3(a): Tool Support Does Not Need to be Sophisticated to be Effective

Table 5.4 shows the quality metrics of the student-authored test code, including the requirements coverage (*Requirement (%)*), instruction coverage (*Instruction (%)*), branch coverage (*Branch (%)*), mutation coverage (*Mutation (%)*), the number of identified bugs (out of three total) (*Identified Bugs*), and the number of smelly tests (*Smelly Tests*). For example, the average student in the Checklist group wrote 13.4 assertions (*numAssertions*) and achieved a requirements coverage of 74.4%.

We found that students overall in the Checklist group did no worse than those in the Coverage group in terms of completeness (i.e., requirements coverage and code coverage). Though 60% of students (9/15) in the Checklist group also used EclEmma during unit testing, using a coverage tool is not a mandatory item in the checklist. It is noteworthy that students in the Coverage group were encouraged to achieve 80% instruction coverage on every non-test class[1] while no such threshold was mentioned for students in the Checklist group. Given the small number of participants, we did not expect to find significant differences in test quality between the Checklist and the Coverage groups [171], and our ANOVA analysis confirms this (Table 5.3).

The biggest difference between the Checklist group and the Coverage group can be seen in the mutation coverage (median 65.0% vs. 51.0%, in Table 5.4). While the difference is not statistically significant ($p = 0.14$), the effect size was medium ($Cohen's\ d = 0.55$), indicating that the Checklist group did moderately better. However, using the testing checklist had no statistically significant impact on the number of identified bugs ($p = 0.86$).

Moreover, we observed no statistically significant difference ($p = 0.82$) in the number of smelly tests among the student-authored tests from the Checklist and the Coverage groups.

[1]Consistent with the requirements in our undergraduate software engineering courses.

Table 5.4: Measurements of student-authored tests quality.

Metrics	Group	Overall		Undergrad		Grad	
		avg	med	avg	med	avg	med
numAssertions	Checklist	13.4	7.0	16.0	16.0	6.3	6.5
	Coverage	18.8	14.0	21.6	19.0	12.0	13.0
Requirement (%)	Checklist	74.4	76.9	77.6	84.6	65.4	61.5
	Coverage	70.6	76.9	78.2	80.8	52.3	53.8
Instruction (%)	Checklist	81.4	87.0	83.9	89.7	74.5	75.4
	Coverage	79.2	86.2	85.0	88.0	65.4	58.3
Branch (%)	Checklist	66.1	66.0	70.5	70.5	54.0	54.5
	Coverage	64.0	65.9	70.8	69.4	47.8	45.7
Mutation (%)	Checklist	59.6	65.0	59.0	62.0	61.3	65.5
	Coverage	50.5	51.0	52.3	52.0	46.2	40.0
Identified Bugs	Checklist	0.7	0.0	0.8	0.0	0.3	0.0
	Coverage	0.6	0.0	0.8	0.5	0.0	0.0
Smelly Tests	Checklist	0.6	0.0	0.5	0.0	0.8	1.0
	Coverage	0.7	0.0	0.5	0.0	1.2	1.0

Finally, none of the interaction results were statistically significant, indicating that the checklist did not impact graduate and undergraduate students differently.

5.1.5.2 RQ3(b): Most Students Used the Checklist as a Guide

Most students in the Checklist group reported that they read the checklist before they wrote any unit tests (13/15), and approximately half consulted the checklist during unit testing (7/15). They found it to be very helpful, rating it an average of 3.9 on a 5-point Likert scale. All students who used the checklist agreed that checklist item #2, *"Each test case should have at least one assert statement or asserts an exception is thrown"* was the most helpful item. They also considered the following checklist items to be helpful: item #1 (11/15), #10a (11/15), item #10b (10/15) and item #4 (10/15).

To add nuance to our understanding of checklist engagement, we also watched the eight screen capture videos of the Checklist group. To illustrate mechanisms by which the checklist might have been helpful, we present two students who demonstrated successful use of the checklist and one student who struggled with it.

Student One used the testing checklists as a guide and consulted the provided tutorial information:

Student One tried to assert an exception is thrown, but was encountering syntax errors. They opened the checklist and followed the link in the second checklist item to the relevant tutorial. After 4 minutes, they successfully constructed a test case that asserts an exception is thrown. This student voted item#2 as the most helpful, and found the checklist "extremely helpful".

The test code written by Student One achieved requirements coverage of 100.0%, instruction coverage of 92.2%, branch coverage of 86.4%, and a mutation coverage of 68.0%. However, Student One did not reveal any bugs.

The second student adopted the checklists as an inspiration of what and how to test:

Student Two placed the checklist side by side with Eclipse. They tried to write a test that asserts an exception is thrown, but was encountering syntax errors. Instead of using the tutorial link, this student copied keywords from the checklist and pasted them into a search engine, and then consulted StackOverflow where they found sample tests. The student successfully implemented the desired test within three minutes of using the checklist. This student agreed that item#2 is one of the most helpful and found the checklist "very helpful" overall.

This student successfully revealed two of three seeded faults in the source code, along with a mutation coverage of 82.0%; they covered 84.6% of the requirements, achieved instruction coverage of 94.0%, and branch coverage of 86.4%. All these scores are above the average and median values for the group.

However, we also observed cases where students still struggled to test their code effectively, even with the checklist's guidance:

Student Three self-reflected in the post-activity survey that, "*I had a hard time understand[ing] how to test some methods, therefore, I could not spare time to explicitly follow the checklist. However, I was able to follow the checklist somewhat vaguely. The checklist itself was fine, I would not recommend any changes.*"

Student Three had a more average performance: they achieved requirements coverage of 76.9% (higher than average), instruction coverage of 91.8% (higher than average), and branch coverage of 63.6% (lower than average). The tests written by this student revealed no bugs (lower than average) along with a mutation coverage of 60.0% (similar to average).

While the checklist seemed to benefit Students One and Two, Student Three may have performed better with more guidance on the testing techniques [38]. This echoes the finding that not all students need the same level of tool support [85].

5.1.6 Takeaways

Based on our experiment, the testing checklist does as well as, and possibly better than, a coverage tool alone. This is important because students may be hesitant to adopt new tools or struggle to use them effectively. Checklists are flexible and adoptable with a minimal learning barrier.

5.1.6.1 Potential Benefit to Novices

We observed that the adoption of lightweight tool support, like a checklist, might be more helpful for students with lower prior knowledge in Java and unit testing. The four graduate students in the Checklist group had minimal experience in Java (avg: 0.1 yrs) and unit testing (avg: 0.4 yrs), but they produced higher mutation coverage than more-experienced graduate students in the Coverage group (median 65.5% vs. 40.0%). Due to the small sample size, this finding may not generalize.

5.1.6.2 Potential Impact on Mutation Coverage

While analyzing the student-authored test code, we found that the students in the Checklist group achieved higher mutation coverage despite writing fewer assert statements (Table 5.4). Though the differences were not statistically significant ($p = 0.21$), we dug into the implementation details and found students in the Checklist group covered more of the complicated requirements, such as calculating bonus points for strikes and spares, with logic that is mathematical and prone to mutants. This may also suggest that using a checklist allows students to focus more on the boundary values (checklist item #10b) than simply covering the requirements or lines of code.

5.1.7 Threats to Validity

Conclusion: The two-way ANOVA analysis we used assumes the data are normally distributed; however, not all of the data was normally distributed. Hence, this may impact some conclusions drawn.

The lab sessions were conducted synchronously and remotely. To alleviate the concern about potential academic misconduct, we required students to record their screen activities while working on the unit testing project, and submit the videos along with their test code upon completion of the study.

Construct: Sampling bias could be introduced in participant recruitment as students were self-selected into this study. Students worked individually and remotely, which could potentially impact their behavior. Metrics, such as time, may not be consistent across students due to a lack of control in the study environment. This influence was reduced by asking students to record their screen activities.

Internal: We only observed students' interactions with an unfamiliar codebase. Students may perform differently on programs implemented by themselves or their peers due to the familiarity.

Response bias may be introduced in the student self-reported surveys, and hence impact the validity of surveys. Students' interpretation of survey questions and options may vary and potentially be imprecise, which could bias the results as well.

External: The students in the study were relatively advanced students studying software engineering, and hence these results may not generalize to other students or those with less testing experience.

5.2 Study 2: Explicit Testing Strategies

This study received IRB approval from NCSU (Protocol Number #24796).

5.2.1 Study Rationale

Section 5.1 demonstrated that the lightweight testing checklist we designed effectively supported students in writing quality tests. However, the checklist does not explicitly address students' challenges in determining *what* code/scenarios to test and identifying *how* to test (Chapter 3). To that end, we adopt and apply the concept of explicit strategies [93, 102] to testing, which provide a series of actions for students to follow to guide them through the testing process. We then integrate the explicit testing strategies into the checklist.

To better understand the contexts in which a testing checklist would be most helpful, we conducted an experimental, controlled A/B study with 192 undergraduate students in a classroom setting. Students in the experimental group received the explicit testing strategy checklists (ETS checklists) as additional resources when completing their programming lab assignments, while the students in the control group received no additional support.

To best of our knowledge, the application of explicit strategies in testing is novel, and this study addresses the general research question RQ3, which is broken into one additional sub-question:

RQ3: How to improve students' testing practices with checklists?

> **RQ3(c):** How do students engage with the explicit testing strategy checklists in a classroom setting?

5.2.2 Classroom Setting

We ran this study in Spring 2022 with students who were taking a 3-credit CS1.5 course (lecture course) and its associated 1-credit laboratory course at NCSU. This CS1.5 course is a Java-based course taken by all CS majors and minors and is open to non-majors. In this course, students learn fundamentals in object-oriented design and development, basic software engineering concepts, and linear data structures. In the lab, students practice the concepts and tools covered in lecture. Students attend weekly lab meetings (110 minute each) and work in teams (2-4 students) to implement and test new functionality into a semester-long project.

5.2.2.1 Lab Assignments and Team Formation

In the lab, students complete a total of 11 labs with three different teams over the semester: they complete Labs 1-4 with *Team #1*, Labs 5-8 with *Team #2*, and Labs 9-11 with *Team #3*. When students rotate to a new team, they are instructed to build off their labs (e.g., Labs 5 & 9) of the best existing implementation from their new teammates.

Students are randomly assigned into teams by the teaching assistants (TAs). When working with a team, students are expected to collaborate with their teammates and use the same GitHub repository.

As the checklist intervention is designed for students who have received some education in software testing but may need some further assistance, we conduct this study in the second half of the semester with Labs 6-11.

5.2.2.2 Educational Support

In Spring 2022, there were nine in-person lab sections, and each lab section has two TAs (18 TAs in total). All of the TAs received a TA training at the beginning of the semester.

The TAs of the lab sections led the lab meetings with overview of the lab assignment of the week, including a recap on the concepts/tools that covered in the lecture course, and the methods that students were expected to implement and test. All overview slides for

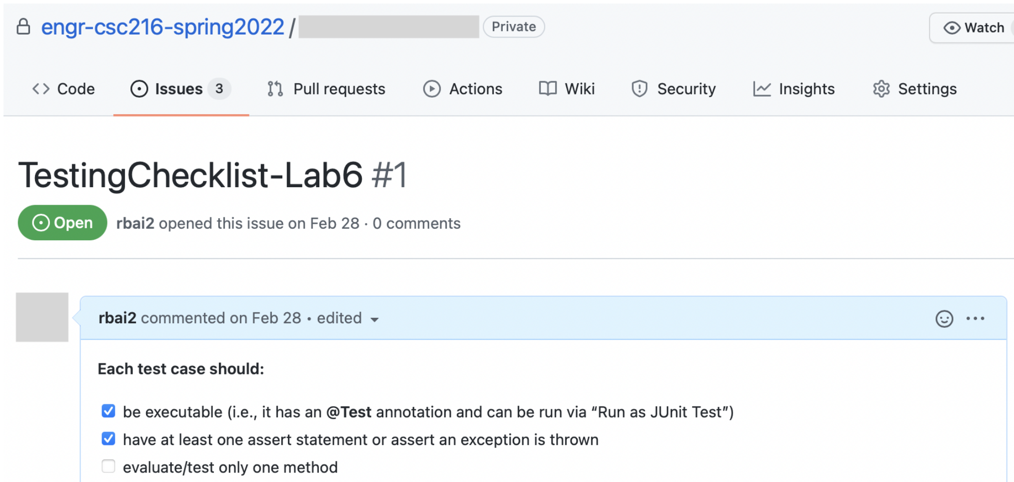

Figure 5.2: Excerpt from the testing checklist designed for Lab 6 as an GitHub *Issue*.

Labs 6-11 were provided to the TAs. TAs are present for the entire duration of lab to answer students' questions and provide guidance and feedback. Students could attend the office hours held by all teaching staffs, including instructors and TAs, outside of the lab meetings. Students were also encouraged to post questions on Piazza, which could be answered by all teaching staffs and their peers.

5.2.3 Explicit Testing Strategy Checklist

We present an example ETS checklist (as shown in Table 5.5) in this section to demonstrate the use of explicit testing strategies. While the explicit testing details in this example checklist (item #10) are specific to Lab 6 from the laboratory course, the rest of the checklist was used for all labs. This ETS checklist was provided to students in the experimental group as an additional resource for this lab assignment: on each student repository, we created a GitHub *Issue* containing the checklist (as shown in Figure 5.2), which allows students to check off the items during testing, as suggested in prior work [110].

5.2.3.1 Requirements in Lab 6

In Lab 6, students were expected to implement and test a finite state machine (FSM) that validates a course name string. The requirements stated that:

Req1 : *A valid course name begins with 1-4 letters, followed by exactly 3 digits, followed by an optional 1 letter suffix.*

Req2 : *If a course name does not meet the description, the course name is invalid.*

Req3 : *Spaces are no longer allowed between the prefix and number.*

5.2.3.2 Explicit Testing Strategy Checklist Designed for Lab 6

Unlike the original checklist (Table 5.1), which reminds students to consider the testing techniques (i.e., equivalence class partitioning and boundary value analysis) at a high level, this ETS checklist explicitly guides students to transfer semantic information from the lab assignment instructions to the possible testing scenarios (as shown in item #10, Table 5.5). Additionally, we remove the linked tutorials in items #2 and #11 from the original checklist (Table 5.1) as all students in this study were already familiar with testing exceptions and using the coverage tool EclEmma.

To design the explicit strategies, we follow five steps, and we illustrate each step using the example of Lab 6:

- **Step 1: Determine the methods that need to be tested.**
  For Lab 6, students are expected to test an FSM that validates a course name string, specifically `isValid()`, which accepts a `String` parameter and returns a `boolean`.

- **Step 2: Apply domain knowledge when necessary.**
  We first remind students that it is important to "*Test every state*" in an FSM, though it is not explicitly stated in the lab assignment instructions.

- **Step 3: Apply testing technique *equivalence class partitioning.***
  For each state, we divide the input based on its type: "*Transition on **Letter/Digit/Other***", which addresses the *Req1* & *Req3*. We further divide the partitions into sub-partitions based on the validity of the input: "***valid/invalid** number of transitions*", which addresses the *Reqs1-3*.

- **Step 4: Apply testing technique *boundary value analysis* on each partition.**
  Since values at or close to boundaries of a range often cause problems [3], we remind students to create tests that include representatives of **boundary values** in each range (e.g., "*Min/Between min and max/Max*", addressing *Req1*).

- **Step 5: Instruct students to create a test for each partition.**
  We explicitly ask students to "*Pick an input...*" in each partition and create a test case.

Table 5.5: One of the testing checklists used in this study. This checklist contains the explicit testing strategies designed for Lab 6 (item #10).

Item	Test Case Checklist
	Each test case *should*:
#1	☐ be executable (i.e., it has an `@Test` annotation and can be run via "Run as JUnit Test")
#2	☐ have at least one assert statement or assert an exception is thrown.
#3	☐ evaluate/test only one method
	Each test case *could*:
#4	☐ be descriptively named and commented
#5	☐ If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using `@Before`)
#6	☐ If there are too many assert statements in a single test case (e.g., more than 5), you might split it up so each test evaluates one behavior.
	Test Suite Checklist
	The test suite *should*:
#7	☐ have at least one test for each requirement
#8	☐ appropriately use the setup and teardown code (e.g., `@Before`, which runs before each `@Test`)
#9	☐ contain a fault-revealing test for each bug in the code (i.e., a test that fails)

#10	☐ test an `FSM` (*Explicit Testing Strategies for Lab 6*)
	  ☐ Test every state:
	    ☐ Initial state
	    ☐ States in between
	    ☐ Final/End state
	  ☐ For each state, consider the following scenarios:
	    ☐ Transition on Letter
	      ☐ Pick an input with a valid number of transitions
	        ☐ Min
	        ☐ Between min and max
	        ☐ Max
	      ☐ Pick an input with an invalid number of transitions
	        ☐ Less than min
	        ☐ Greater than max
	    ☐ Transition on Digit
	      ☐ Pick an input with a valid number of transitions
	        ☐ Min
	        ☐ Between min and max
	        ☐ Max
	      ☐ Pick an input with an invalid number of transitions
	        ☐ Less than min
	        ☐ Greater than max
	    ☐ Transition on Other
	      ☐ Invalid, causing an exception

	To improve the test suite, you *could*:
#11	☐ measure code coverage using an appropriate tool, such as EclEmma. Inspect uncovered code and write tests as appropriate.

To validate the explicit testing strategies designed for Lab 6, we conducted a small pilot study with a first-year graduate student in CS who had no prior experience with this CS1.5 course or other laboratory course offered at NCSU. This graduate student was instructed to implement the methods required in Lab 6, and write tests strictly following the ETS checklist. We measured their test code in terms of instruction coverage and branch coverage. As a result, we modified the wording of the explicit testing strategies to avoid ambiguity, and adjusted the sub-items to ensure a full (i.e., 100%) instruction coverage and a full branch coverage.

For each of the Labs 6-11, we designed a new ETS checklist and validated it with a pilot study as above. The ETS checklists are available in Appendix C.2.3.

5.2.4 Methodology

To better understand the contexts in which a testing checklist would be most helpful, we conducted an experimental, controlled A/B testing study with 192 students who were enrolled in the in-person lab sections by the time of the study.

5.2.4.1 Experimental Group vs. Control Group

With a goal of balancing the lab sections in terms of meeting times (i.e., morning vs. afternoon sections, and days of the week) in each group, we divided the lab sections into two groups:

- **Group 1 - five sections in total**
  - 1 morning section, 4 afternoon sections
  - 2 Monday sections, 2 Tuesday sections, 1 Wednesday section
- **Group 2 - four sections in total**
  - 1 morning section, 3 afternoon sections
  - 2 Monday sections, 1 Tuesday section, 1 Wednesday section

For Labs 6-8 (in which students worked in *Team #2*), Group 1 served as the experimental group (*isExp* in Table 5.6) and received the testing checklists, while Group 2 was the control group and received no additional resources. For Labs 9-11 (in which students worked in *Team #3*), we swapped the experimental group and the control group. That is, Group 2 was

Table 5.6: Group Assignment and Student Information.

	Labs 6 - 8			Labs 9 - 11		
	#Students	#Teams	isExp	#Students	#Teams	isExp
Group 1	97	33	Yes	94	33	No
Group 2	95	32	No	92	32	Yes

Table 5.7: Survey Response Rates.

	Labs 6 - 8			Labs 9 - 11		
	Short_1st	Short_2nd	Comp	Short_1st	Short_2nd	Comp
Group 1	7 (7.2%)	72 (74.2%)	68 (70.1%)	–		
Group 2	–			72 (78.3%)	71 (77.2%)	45 (48.9%)

now our experimental group and received testing checklists with explicit strategies for each lab, while Group 1 received no additional resources (but still had access to testing checklists for Labs 6-8). Students in all groups completed their lab assignments and committed to their team repositories for all labs.

5.2.4.2 Weekly Surveys

In this study, students in the experimental group were invited to complete three weekly surveys in total: two Short Surveys in weeks 1 & 2 (*Short_1st* and *Short_2nd* in Table 5.7, respectively) and one Comprehensive Survey in week 3 (*Comp*). The Comprehensive Survey (Appendix C.2.2) asked students how they used the ETS checklist, how useful they found it, as well as the checklist items that they found the most helpful. This survey consists of Likert-scale rating, selection, and open-ended questions. The Short Survey (Appendix C.2.1) only focused on students' use and evaluation of the ETS checklist. This survey consists of Likert-scale rating and open-ended questions. We did not survey students in the control group as they received no additional resources.

The completion of the surveys and all survey questions were optional, and students were not compensated for completing them. Table 5.7 shows the response rates.

The first Short Survey (*Group 1 - Short_1st*) was distributed via Qualtrics; all subsequent surveys were paper-based, a change we made to improve the survey response rates.

5.2.4.3 Students' Lab Submissions

In each lab, students implement new functionality and new tests for the semester-long project (Section 5.2.2). That is, each lab builds off of the previous lab, which was implemented by students themselves or their peers due to team rotation. Consequently, the labs are not fully independent of each other, and thus we choose Labs 8 and 11 as representative of these rotations to study students' testing practices.

For each lab assignment, the last GitHub commit before the deadline is considered as the "final submissions", and used for grading. Hence, we adopted these final submissions for Lab 8 and Lab 11 from both groups for data analysis.

5.2.5 Data Analysis

In total, we collected 335 surveys responses (Group 1: 147 surveys, Group 2: 188 surveys) and 130 lab submissions (65 submissions from each group) for Labs 8 & 11, as they were the final team products of rotations *Team #2* and *Team #3*, respectively.

5.2.5.1 Weekly Surveys

For rating questions, we converted 5-point Likert scale to numbers and treat them as interval-scaled data [73], where 1 maps to the lowest score (e.g., "Not at all helpful", "Extremely unlikely ", and "Never"), and 5 maps to the highest score (e.g., "Extremely helpful", "Extremely likely", and "Always ").

5.2.5.2 Students' Lab Submissions

We measured the test code quality with four metrics:

1) Mutation coverage
We measured this effectiveness metric via PITest.

2) Instruction coverage & 3) Branch coverage
We measured these two completeness metrics via EclEmma.

4) The number of unhappy path tests
Our prior study in Section 4.1 revealed that students tend to only test the happy paths (i.e., the test smell *Happy Path Only*), and we address this issue via the checklist intervention. However, the large scale of this study prevented us from manually

identifying test smells (as we did in Section 5.1.4.2), and to our knowledge, there is no existing approach to automatically detect the unhappy path tests. Hence, we opted to only consider the tests that assert an exception is thrown. We used three different patterns to check for tests that expected exceptions: 1) `assertThrows`, 2) `@Test(expected = exception.class`, or 3) `Try-Catch` blocks. We counted and adopted these as **the number of unhappy path tests**. This is a measure of completeness.

5.2.6 Results

With the combination of survey responses and tests generated by students, we explore how students engage with the ETS checklists.

5.2.6.1 Students' Hesitation in Adopting Checklists

> Observation 1: While most students considered the checklists moderately/slightly helpful, about half of them self-reported that they did not use the ETS checklists when working on the lab assignments.

Students in the experimental group self-reported their use (Table 5.8) and evaluations (Table 5.9) of the ETS checklists in weekly surveys (two Short Surveys (*Short_1st*, *Short_-2nd*) and one Comprehensive Survey (*Comp*)). While most of students in Group 1 rarely consulted the checklists, they found them "moderately helpful" (medians of 3.0 in Table 5.9). Having gained more knowledge and experience in testing by the time students in Group 2 saw the ETS checklists, most of them claimed that they never consulted the checklists, and considered the checklist "slightly helpful" (medians of 2.0 in Table 5.9). Though students did not frequently use the checklists, they were more likely to use the checklist while performing unit testing (*Unit Testing* in Table 5.8) or discussing the testing tasks with their teammates (*Team Discussion*) than in system testing.

From the survey responses, we observed declines in both students' adoption and evaluation of the checklists. There are several possible interpretations of this result. One is that the ETS checklists were scaffolding that could be removed as students gaining more knowledge and experience in testing. A related interpretation is that all Labs 7-11 focused on implementing and testing linear structures such as array-based list, linked list, stack, and queue (Appendix C.2), in which students practiced and internalized the skills to test a linear structure and hence needed less support from the checklists. Another interpretation

Table 5.8: Answers to Survey Question "How often did you consult the checklist during..." (5-point Likert Scale, where 1 maps to "Never", 2 maps to "Rarely", and 3 maps to "Sometimes").

Group	Survey	System Testing		Unit Testing		Team Discussion		Overall	
		avg	med	avg	med	avg	med	avg	med
Group 1	Short_1st	1.7	2.0	2.4	3.0	2.3	3.0	2.1	2.0
	Short_2nd	1.6	1.0	2.0	1.0	1.9	2.0	1.8	1.0
	Comp	1.8	1.5	2.0	2.0	2.0	2.0	1.9	2.0
Group 2	Short_1st	1.4	1.0	1.8	1.0	1.6	1.0	1.6	1.0
	Short_2nd	1.4	1.0	1.7	1.0	1.6	1.0	1.5	1.0
	Comp	1.4	1.0	1.7	1.0	1.5	1.0	1.5	1.0

Table 5.9: Answers to Survey Question "How would you rate this checklist in terms of helpfulness during..." (5-point Likert Scale, where 1 maps to "Not at all helpful", 2 maps to "Slightly helpful", and 3 maps to "Moderately helpful").

Group	Survey	System Testing		Unit Testing		Team Discussion		Overall	
		avg	med	avg	med	avg	med	avg	med
Group 1	Short_1st	2.5	2.5	3.2	3.0	2.8	3.0	2.8	3.0
	Short_2nd	2.5	3.0	2.9	3.0	2.7	3.0	2.7	3.0
	Comp	2.5	3.0	2.8	3.0	2.6	3.0	2.7	3.0
Group 2	Short_1st	2.3	2.0	2.6	3.0	2.2	2.0	2.4	2.0
	Short_2nd	2.1	2.0	2.4	2.0	2.1	2.0	2.2	2.0
	Comp	2.0	2.0	2.4	2.0	2.2	2.0	2.2	2.0

is that as the semester approached to the end, students had established their own testing workflow, as well as had less time to adopt additional resources even though they found them helpful.

5.2.6.2 Students' Unawareness of the Needs for Help

> Observation 2: Students self-claimed that they did not need extra support in testing or were not motivated to use the ETS checklists; however, student-authored tests indicated that the support from the ETS checklists would be helpful.

In 49.3% (165/335) of the weekly surveys students claimed to not use the ETS checklists at all (Group 1: 54/147 surveys, 36.7%; Group 2: 111/188 surveys, 59.0%). Among these 165 survey responses, in 83 of them students provided open-ended comments on the ETS checklists, including why they did not use them. We summarize the two major reasons students provided:

1. Students **did not need the support** from the checklists (20 responses, 24.1%), which echoes the finding that not all students need the same level of tool support [85].

    - "*Did not use, I already have my own checklist in my head.*"
    - "*I just didn't really use it. I've never felt I needed the extra help.*"
    - "*No, I'm a strong independent programmer who does not need a checklist.*"

2. Students were **not motivated to use** the checklist (29 responses, 34.9%), since

    - they forgot to use the checklist (18 responses, 21.7%).
        - "*Full disclosure - I forgot there was a checklist.*"
        - "*Forgot about the checklist so I didn't use it, but it seems helpful.*"
        - "*I honestly forgot we had the checklist when we did the lab. [...] During group discussion, it was very helpful. Next time I would definitely use it.*"
    - they did not want to adopt new tools or new testing practices (7 responses, 8.4%).
        - "*Introduce them at the beginning of the semester - at this point we all have a certain way of delegating and checking off tasks.*"
        - "*I have done fine without the checklist and see no need to change my habits this closer to the end of the semester.*"
        - "*By this time in the semester, I'm not motivated enough to add a new extra step to my testing process.*"
    - the adoption of the checklist would not directly improve their grades (e.g., extra credit) in lab assignments (4 responses, 4.8%).
        - "*I did not think it was necessary to get a good grade, so I did not use it much since it just took more time.*"
        - "*The checklist seems useful but many students share a similar sentiment about it: if you give a student extra work on top of an already long assignment and it isn't for a grade, they won't do it.*"
        - "*I have been pretty busy during weekdays lately and didn't want to devote a lot of time to testing, which I am already getting good grades on, so I didn't pay much attention to the checklist.*"

Table 5.10: Measurements of the quality of student-authored tests: mutation coverage (%), instruction coverage (%), branch coverage (%), and the number of unhappy path tests (%, #StdDev (normalized to number of standard deviations from average, as this metric continually increases as the project continues to build up over the semester)).

Lab	**Group**	**Instruction**		**Branch**		**Mutation**		**#UnhappyPath**	
		avg	med	avg	med	avg	med	avg	med
Lab 8	Group 1_isExp	85.0	85.3	77.6	76.9	47.9	47.7	15.8, -0.13	15.7, -0.27
	Group 2	84.2	84.7	76.5	77.2	47.3	48.0	16.0, 0.13	16.0, 0.03
Lab 11	Group 1	82.3	82.1	74.4	75.0	46.3	46.5	16.0, -0.16	16.0, -0.16
	Group 2_isExp	81.7	82.5	74.1	74.4	46.1	46.1	16.0, 0.16	16.5, -0.05

Some students in Group 2 also believed that the checklists with explicit testing strategies could benefit the beginners. They reported that, "*Checklist are not helpful this late into the lab process*", "*In my opinion, at this point in the semester, we already know most of the things on the checklist. I think this checklist would be extremely helpful near the beginning of the semester, when we started writing unit tests*" and, "*I think the tests are helpful for beginners, but since we have been making tests this whole semester, we didn't really use the checklist*".

However, the quality of student-authored tests suggested the opposite, that students may actually need the support from the ETS checklists. Table 5.10 shows the quality metrics of the student-authored test code, including the instruction coverage (*Instruction* (%)), branch coverage (*Branch* (%)), mutation coverage (*Mutation* (%)), and the number of unhappy path tests (*#UnhappyPath* (%, #StdDev)). For example, the average team in Group 1 achieved a branch coverage of 77.6% with the checklist in Lab 8, and achieved a branch coverage of 74.4% without the checklist in Lab 11. The *#UnhappyPath* accumulated over the labs as students adding new implementation and tests to the semester-long project, and hence we normalized the values for comparison. For example, 15.8% of the assertions wrote by an average team in Group 1 in Lab 8 addressed the unhappy paths of the program, and the number was 0.13 standard deviations below the average among both Groups. Though students in both Groups achieved over 80% instruction coverage (given the threshold of 80% statement coverage on every non-UI and non-test class for each lab assignment submission), we observed that most students did not perform well in both Labs 8 and 11 in terms of mutation coverage (i.e., effectiveness) and the number of unhappy tests. This indicated that students are likely testing to achieve a coverage level, and need support in writing tests that cover the input values at boundaries as well as the unhappy paths of the programs, which could be addressed by the explicit testing strategies (item #10) in the checklists.

Table 5.11: The overall rank of the most helpful checklist items (rated by students in the Comprehensive Survey).

Item	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11
Group 1 (Response rate: 70.1%)	4	2	11	5	1	3	5	8	5	9	9
Group 2 (Response rate: 48.9%)	8	2	2	8	1	6	4	6	10	10	4

The explicit testing strategies were ranked low in terms of helpfulness by both groups (Table 5.11). This could be caused by the classroom setting in this study. For example, students worked in teams, and hence they could consult their teammates for scenarios that should be tested. They reported that, "*I just found myself rarely consulting the checklist and instead consulted with my teammates directly*". Students could also seek help from their TAs when they did not know why a test failed or how to setup the test case to cover the code. Ko et al. [93] observed the similar behaviors that "asking for help is a more efficient strategy" compared to independently solve a problem. Additionally, the well-written instructions and detailed requirements for each method in the lab assignments might have reduced students' needs for support. Students reflected in the weekly surveys that "*There [are] already many things to track*," and, "*It feels redundant and adds to the swath of things we already need to consult or track*". Another potential explanation is that the explicit strategies were too explicit and became long to read, which discouraged the students from using it, as they self-reported that, "*It's a useful tool that we probably should have used more. I found it very thorough, although the time investment needed to complete it was more than we were able to afford*".

This observation echoes the findings in prior studies that students may avoid help due to 1) unawareness of the need for help [61], 2) concerns about the help-giver's (in this case a checklist) competence [115, 140], and 3) a desire for independence [115].

5.2.7 Takeaways

In this study, we surveyed students on their use and evaluation of the checklists with explicit testing strategies. This information shed light on why students use or do not use the checklists, and how to improve the checklists to better address their needs.

5.2.7.1 Low Motivation to Adopt New Tools

In Section 5.1, we conjectured that the checklist could potentially address students' hesitation to adopt new tools or struggle to use them effectively given the minimal learning barrier. However, in this study, we found that students tended to only adopt tools that are required for grading purposes or could directly reveal the flaws in their code and hence improve their grades.

Overlooking the importance of testing could also lead to the low motivation of tool adoption. Students reported that, "*Testing is pretty time consuming. Maybe it would help if more test classes were provided and we could devote more time to high-quality tests for the remaining classes*". This echoes the challenge in testing education discussed in Section 2.1.1.1.

5.2.7.2 Presentation of The Checklist

Based on students' comments, we suggest several potential improvements on the presentation of the checklist and the explicit testing strategies. For example, instead of presenting the explicit testing strategies designed for all methods in the checklist, we could offer the explicit testing strategies as an optional support for students (e.g., hidden in a question mark button), and provide them next to its associated methods in the lab assignment instructions. This could reduce the amount of reading and hence avoid overwhelming students, as well as avoid overtaxing those who do not need extra support in testing these methods.

5.2.8 Threats to Validity

Conclusion: The two-way ANOVA analysis we used assumes the data are normally distributed; however, not all of the data was normally distributed. Hence, this may impact some conclusions drawn.

The lab assignment instructions explicitly required students to achieve 80% statement coverage on every non-UI and non-test class. This threshold of code coverage was applied to all submissions from both groups, and might have led to the null results in the impact of the checklists on the completeness metrics.

Construct: Sampling bias could be introduced in survey collection as students were self-selected into this study activity.

Internal: We only observed students' interactions with codebase that they were familiar with (either implemented by students themselves or their peers). Students may perform differently on an unfamiliar codebase.

Because the survey on use of the testing checklist was optional, it subject to response bias, and consequently conclusions drawn from it may not generalize. However, we received high response rates (>70%) from both groups, so conclusions are drawn from a majority of the students.

External: The students in the study were advanced beginners who had been exposed to unit testing for more than one semester, and hence these results may not generalize to other students or those with less testing experience.

5.3 Chapter Summary

> *"It is difficult to interest student programmers in thoroughly testing their own programs since every fault found represents a psychological blow to their programming ego."* (Carrington [31])

Overall, these two studies pointed out that **the tool support does not need to be sophisticated to be effective** (Section 5.1), and students who have **lower prior knowledge in Java and unit testing may benefit more** from the lightweight checklist intervention (Section 5.1). These two studies also pinpointed an open problem in testing education that need to be addressed: What features should a testing tool have to be effective to students?

Providing timely support in testing and feedback on test quality is important, but challenging, especially at scale. To that end, researchers and educators have developed various tools to assist students learning and practicing testing. But how do we motivate students to adopt these testing tools? Our study demonstrates that students may be hesitant to adopt tool support due to unawareness of the need for help, as well as the concerns about the help-giver's (in this case a checklist) competence (Section 5.2).

CHAPTER

6

CONCLUSIONS

This dissertation presents the studies we have conducted to support the thesis statement:

> Motivated by the identification of barriers students have during software engineering activities, in particular software testing, we show that a lightweight checklist intervention improves student testing practices.

We first investigated students' activities in searching and using online resources (Chapter 3). The studies revealed that students frequently searched for code examples; however, formulating queries with correct terminologies could be a barrier for students, particularly when they were searching for code in an unfamiliar programming language. Students also encountered the barrier from language migration when reusing the online resources.

Second, we explored how students perceive and perform testing, specifically unit testing (Chapter 4). The results demonstrated that students struggled with understanding source code, implementing tests, and determining when to stop testing. Meanwhile, students self-reported that they desired more support in determining *what* and *how* to test. We also revealed students' insufficient understanding of testing, such as having no consensus on what make a test good.

These findings motivated the need for tool support that contains both tutorial information and testing strategies. To that end, we proposed a checklist intervention to provide testing support without a steep learning curve. We evaluated the effectiveness of the testing checklists both within and outside the classroom (Chapter 5). The results suggested that this checklist intervention is at least as effective as a coverage tool for writing quality tests, and indicated that testing tool support does not need to be sophisticated to be effective. We also found that students who have lower prior knowledge in Java and unit testing may benefit more from the checklist intervention (Section 5.1). However, in a classroom setting, we observed some student hesitation in adopting the checklists (Section 5.2).

6.1 Future Work

In this section, we discuss future work motivated by the studies we have conducted. We welcome and encourage replication studies to improve the generalizability of our findings (Section 6.1.1). In Section 6.1.2, we suggest study methodologies that could be adopted and might bring insights on students' software engineering activities. Section 6.1.3 focuses on potential tool support in searching. The possible improvements on the checklist intervention are discussed in Section 6.1.4.

6.1.1 Replication Studies

Replications of the studies in Chapters 3-5 with a more diverse and larger set of participants, as well as a more diverse set of study tasks are encouraged.

For example, in Section 3.2, we noticed that students composed more verbose queries, and required longer time to scan through the search result than professional developers did [150]. However, this could be caused by the differences of the context. In this case, students were working with APIs that they were not familiar with, while the professionals were conducting their daily work. Replication studies with programming languages or APIs that students are familiar with can help us better understand students' searching activities.

Section 4.2 demonstrates that the impact of the presence of *Assertion Roulette* is minimal. However, the negative results might result from the small sample size. Replication studies with a more diverse and larger set of students can better generalize how *Assertion Roulette* impacts testing education.

In Section 5.1, we observed that students who have lower prior knowledge in Java and unit testing showed benefit more from the testing checklist. However, due to the small sample size, this finding may not generalize. Additionally, students in this study were largely unsuccessful at discovering the seeded faults. This may be due to the complexity of the task requirements. Future studies should include simpler tasks and more complex ones to better understand the contexts in which a testing checklist would be most useful.

6.1.2 Adoption of Think-aloud or Eye-tracking

Prior empirical studies with code search do not indicate what factors lead to successful searches, in large part because obtaining evidence of search success is tricky. Relying on result clicks [57] is incomplete as often the answer appears in the preview accompanying a search result. Relying on evidence of query reformulation can reliably find failed searches [76, 84], but not successful ones. In Section 3.2, we addressed this shortcoming using *in situ* surveys that explicitly learn about search success and query reformulation.

However, this approach did not answer questions about students' decision making process. For example, what information motivate them to click into a link? Is it the ordering of search results? Or any keywords in the previews? How do the search results influence their next search activity? Hence, to better understand students' decision making during searching, we suggest future studies to require students to think-aloud or track their eye movements.

In Section 4.1, we found that while students do not admit to having trouble with understanding specifications, they frequently created test cases that mismatched the program specifications. Think-aloud studies could also help us better understand how students interpret the specifications and find the associating implementation, how they pick the scenarios to test the code, and how they convert the scenarios to test cases.

6.1.3 Tool Support in Searching

Section 3.2 reveals that when searching for examples in an unfamiliar language, students frequently borrowed terms from languages with which they are familiar. Similar borrowing behavior was also reported in prior work [6, 158], and developers claim that previous knowledge supports retrieval of new information; our results concur.

For future work, techniques that can determine the relative APIs/libraries based on the context in the queries, and provide query reformulation suggestions according to the search goals are suggested. Tools like "API translators" that map APIs between programming languages and present sample code are also suggested. Another direction of future work is to explore how to detect and correct the term mismatch in queries during the language migration.

In the studies focusing on students' search activities (Sections 3.1 and 3.2), we observed that students sometimes adopted the auto-complete queries generated by the search engines. However, these adoptions and their impacts on search success were not studied. Hence, we encourage future work to explore the impact of auto-complete query on the search success, which help inform better searching practices and potential improvements on search engines.

6.1.4 Extending the Checklist Intervention

Students found it difficult to navigate the code, program requirements, and checklist, which were all in separate places (Sections 5.1 and 5.2). Future studies in classroom settings could integrate the testing checklist directly into the program requirements to improve the skimmability and reduce context switching when working on the assignment.

Section 5.2 reveals that students did not engage with the checklists as a way of seeking help when performing testing in a classroom setting. Therefore, exploration of students' help-seeking behaviors through platforms like Piazza would allow us to better understand students' needs in a particular class and its associating programming labs, and hence improve the checklist intervention in terms of content (e.g., what to cover in the checklists) and context (e.g., when to introduce the checklists).

For future work, a testing checklist that supports automated real-time feedback is also suggested. For example, a checklist that supports students with 1) a progress report, 2) coverage reports, such as code coverage from EclEmma and mutation coverage from PITest, and 3) hints on how to address any shortcoming of the tests, such as test smells, uncovered code and requirements. These features can reduce the amount of tools that students need to learn to evaluate the quality of their tests.

Another potential direction of future work is to integrate the testing checklist into intelligent tutoring systems that facilitate individualized learning.

6.2 Final Remarks

Learning is a journey. Though I do not believe there exists a "best" way to teach and learn testing, I hope this dissertation directs us to a path that is less bumpy.

BIBLIOGRAPHY

[1] Mahtab Alam. “Software security requirements checklist”. In: *International Journal of Software Engineering, IJSE* 3.1 (2010), pp. 53–62.

[2] Tiago L. Alves and Joost Visser. “Static Estimation of Test Coverage”. In: *International Working Conference on Source Code Analysis and Manipulation*. 2009, pp. 55–64.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2nd ed. USA: Cambridge University Press, 2017.

[4] Maurício Aniche, Felienne Hermans, and Arie van Deursen. “Pragmatic Software Testing Education”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE ’19. Minneapolis, MN, USA: ACM, 2019, 414–420.

[5] Andrea Arcuri. “Teaching Software Testing in an Algorithms and Data Structures Course”. In: *Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2020, pp. 419–424.

[6] William R. Aue, Amy Criss, and Matthew D. Novak. “Evaluating mechanisms of proactive facilitation in cued recall”. English (US). In: *Journal of Memory and Language* 94 (June 2017), pp. 103–118.

[7] Prashant Baheti, Laurie Ann Williams, Edward Gehringer, and David Stotts. “Exploring Pair Programming in Distributed Object-Oriented Team Projects”. In: *Companion of the 17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’02. 2002.

[8] Gina R. Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T. Stolee. “Exploring Tools and Strategies Used During Regular Expression Composition Tasks”. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019, pp. 197–208.

[9] Gina R. Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T. Stolee. “Exploring Tools and Strategies Used During Regular Expression Composition Tasks”. In: *Proceedings of the 27th International Conference on Program Comprehension*. ICPC ’19. 2019, pp. 197–208.

[10] Gina R. Bai, Joshua Kayani, and Kathryn T. Stolee. "How Graduate Computing Students Search When Using an Unfamiliar Programming Language". In: *Proceedings of the 28th International Conference on Program Comprehension*. ICPC '20. Seoul, Republic of Korea, 2020, 160–171.

[11] Gina R. Bai, Kai Presler-Marshall, Susan R. Fisk, and Kathryn T. Stolee. "Does Assertion Roulette Smell Bad? An Experiment from the Perspective of Testing Education". In: *2022 IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC '22. 2022.

[12] Gina R. Bai, Kai Presler-Marshall, Thomas Price, and Kathryn T. Stolee. "Check It Off: Exploring the Impact of a Checklist Intervention on the Quality of Student-written Unit Tests". In: *27th ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '22. 2022.

[13] Gina R. Bai, Justin Smith, and Kathryn T. Stolee. "How Students Unit Test: Perceptions, Practices, and Pitfalls". In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE '21. Virtual Event, Germany: ACM, 2021, 248–254.

[14] Albert Bandura. *Social foundations of thought and action: A social cognitive theory*. Prentice-Hall, Inc, 1986.

[15] Chetan Bansal, Thomas Zimmermann, Ahmed Hassan Awadallah, and Nachiappan Nagappan. *The Usage of Web Search for Software Engineering*. 2019. arXiv: `1912.09519`.

[16] Veronika Bauer, Jonas Eckhardt, Benedikt Hauptmann, and Manuel Klimek. "An Exploratory Study on Reuse at Google". In: *International Workshop on Software Engineering Research and Industrial Practices*. SER&IPs 2014. Hyderabad, India, 2014, pp. 14–23.

[17] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. "An empirical analysis of the distribution of unit test smells and their impact on software maintenance". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 2012, pp. 56–65.

[18] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea Lucia, and Dave Binkley. "Are Test Smells Really Harmful? An Empirical Study". In: *Empirical Software Engineering* 20.4 (Aug. 2015), 1052–1094.

[19] Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. “Regviz: Visual debugging of regular expressions”. In: *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM. 2014, pp. 504–507.

[20] Lex Bijlsma, Niels Doorn, Harrie Passier, Harold Pootjes, and Sylvia Stuurman. “How do Students Test Software Units?” In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 2021, pp. 189–198.

[21] Rodrick Borg and Martin Kropp. “Automated Acceptance Test Refactoring”. In: *Workshop on Refactoring Tools*. WRT ’11. 2011, pp. 15–21.

[22] Matt Bower and Annabelle McIver. “Continual and Explicit Comparison to Promote Proactive Facilitation during Second Computer Language Learning”. In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’11. New York, NY, USA, 2011, 218–222.

[23] Jon Bowyer and Janet Hughes. “Assessing Undergraduate Experience of Continuous Integration and Test-driven Development”. In: *28th International Conference on Software Engineering*. ICSE ’06. 2006, pp. 691–694.

[24] Michael K. Bradshaw. “Ante Up: A Framework to Strengthen Student-Based Testing of Assignments”. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE ’15. Kansas City, Missouri, USA: ACM, 2015, pp. 488–493.

[25] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. “Example-Centric Programming: Integrating Web Search into the Development Environment”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’10. New York, NY, USA, 2010, 513–522.

[26] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. “Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’09. New York, NY, USA, 2009, 1589–1598.

[27] Bill Brykczynski. “A Survey of Software Inspection Checklists”. In: *SIGSOFT Software Engineering Notes* 24.1 (Jan. 1999), p. 82.

[28] Ingrid A. Buckley and Winston S. Buckley. “Teaching Software Testing using Data Structures”. In: *International Journal of Advanced Computer Science and Applications* 8.4 (2017).

[29] Kevin Buffardi and Juan Aguirre-Ayala. "Unit Test Smells and Accuracy of Software Engineering Student Test Suites". In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE '21. Virtual Event, Germany: ACM, 2021, 234–240.

[30] Kevin Buffardi, Pedro Valdivia, and Destiny Rogers. "Measuring Unit Test Accuracy". In: *Technical Symposium on Computer Science Education*. SIGCSE '19. 2019, 578–584.

[31] David Carrington. "Teaching Software Testing". In: *Proceedings of the 2nd Australasian Conference on Computer Science Education*. ACSE '97. The Univ. of Melbourne, Australia: ACM, 1997, 59–64.

[32] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. "How the Experience of Development Teams Relates to Assertion Density of Test Classes". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 223–234.

[33] F. T. Chan, T. H. Tse, W. H. Tang, and T. Y. Chen. "Software testing education and training in Hong Kong". In: *Conference on Quality Software (QSIC'05)*. 2005, pp. 313–316.

[34] Carl Chapman and Kathryn T. Stolee. "Exploring Regular Expression Usage and Context in Python". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. New York, NY, USA, 2016, 282–293.

[35] Chunyang Chen, Zhenchang Xing, and Yang Liu. "What's Spain's Paris? Mining Analogical Libraries from Q&A Discussions". In: *Empirical Software Engineering* 24.3 (June 2019), pp. 1155–1194.

[36] Chun Yong Chong, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. "Assessing the Students' Understanding and their Mistakes in Code Review Checklists: An Experience Report of 1,791 Code Review Checklist Questions from 394 Students". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 2021, pp. 20–29.

[37] *Computer Science Curricula Recommendations: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. `https://www.acm.org/education/curricula-recommendations`. ACM, 2013.

[38] Lucas Cordova, Jeffrey Carver, Noah Gershmel, and Gursimran Walia. "A Comparison of Inquiry-Based Conceptual Feedback vs. Traditional Detailed Feedback Mechanisms in Software Testing Education: An Empirical Investigation". In: *52nd ACM Technical Symposium on Computer Science Education*. 2021, 87–93.

[39] Shelley J. Correll. "Constraints into Preferences: Gender, Status, and Emerging Career Aspirations". In: *American Sociological Review* 69.1 (2004), pp. 93–113. eprint: `https://doi.org/10.1177/000312240406900106`.

[40] Shelley J. Correll. "Gender and the Career Choice Process: The Role of Biased Self-Assessments". In: *American Journal of Sociology* 106.6 (2001), pp. 1691–1730.

[41] Tonya Custis and Khalid Al-Kofahi. "A New Approach for Evaluating Query Expansion: Query-document Term Mismatch". In: *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '07. Amsterdam, The Netherlands, 2007, pp. 575–582.

[42] Ermira Daka and Gordon Fraser. "A Survey on Unit Testing Practices and Problems". In: *2014 IEEE International Symposium on Software Reliability Engineering (ISSRE '14)*. 2014, pp. 201–211.

[43] Ermira Daka, José Miguel Rojas, and Gordon Fraser. "Generating Unit Tests with Descriptive Names or: Would You Name Your Children Thing1 and Thing2?" In: *International Symposium on Software Testing and Analysis*. ISSTA 2017. 2017, pp. 57–67.

[44] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. "Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-Use and Portability of Regular Expressions". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, 443–454.

[45] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. "Assessing Diffusion and Perception of Test Smells in Scala Projects". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 457–467.

[46] *Debuggex: Online visual regex tester. JavaScript, Python, and PCRE*. `https://www.debuggex.com`.

[47] Arie van Deursen, Leon M.F. Moonen, A. Bergh, and Gerard Kok. *Refactoring Test Code*. Tech. rep. 2001.

[48] *EclEmma: Coverage Counters*. `https://www.eclemma.org/jacoco/trunk/doc/counters.html`. Accessed: 2021-08-10.

[49] Stephen H. Edwards and Zalia Shams. “Comparing Test Quality Measures for Assessing Student-Written Tests”. In: *36th International Conference on Software Engineering*. ICSE Companion 2014. 2014, 354–363.

[50] Stephen H. Edwards and Zalia Shams. “Do Student Programmers All Tend to Write the Same Software Tests?” In: *Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’14. 2014, 171–176.

[51] Stephen H. Edwards, Zalia Shams, Michael Cogswell, and Robert C. Senkbeil. “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*. SIGCSE ’12. Raleigh, North Carolina, USA: ACM, 2012, 221–226.

[52] S. Elbaum, S. Person, J. Dokulil, and M. Jorde. “Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable”. In: *29th International Conference on Software Engineering (ICSE’07)*. 2007, pp. 688–697.

[53] Sebastian Elbaum, Gregg Rothermel, and John Penix. “Techniques for Improving Regression Testing in Continuous Integration Development Environments”. In: *International Symposium on Foundations of Software Engineering*. FSE 2014. 2014, pp. 235–245.

[54] H. Erdogmus, M. Morisio, and M. Torchiano. “On the effectiveness of the test-first approach to programming”. In: *IEEE Transactions on Software Engineering* 31.3 (2005), pp. 226–237.

[55] Marc Fisher and Gregg Rothermel. “The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms”. In: *SIGSOFT Software Engineering Notes* 30.4 (2005), 1–5.

[56] Susan R. Fisk, Tiah Wingate, Lina Battestilli, and Kathryn T. Stolee. “Increasing Women’s Persistence in Computer Science by Decreasing Gendered Self-Assessments of Computing Ability”. In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE ’21. Virtual Event, Germany: ACM, 2021, 464–470.

[57] Steve Fox, Kuldeep Karnawat, Mark Mydland, Susan Dumais, and Thomas White. “Evaluating Implicit Measures to Improve Web Search”. In: *ACM:TOIS*. Vol. 23. 2. 2005, pp. 147–168.

[58] Davide Fucci, Simone Romano, Maria Teresa Baldassarre, Danilo Caivano, Giuseppe Scanniello, Burak Turhan, and Natalia Juristo. “A Longitudinal Cohort Study on the Retainment of Test-driven Development”. In: *International Symposium on Empirical Software Engineering and Measurement*. ESEM ’18. ACM, 2018, 18:1–18:10.

[59] Davide Fucci, Giuseppe Scanniello, Simone Romano, Martin Shepperd, Boyce Sigweni, Fernando Uyaguari, Burak Turhan, Natalia Juristo, and Markku Oivo. “An External Replication on the Effects of Test-driven Development Using a Multi-site Blind Analysis Approach”. In: *International Symposium on Empirical Software Engineering and Measurement*. ESEM ’16. 2016, 3:1–3:10.

[60] Davide Fucci and Burak Turhan. “A Replicated Experiment on the Effectiveness of Test-First Development”. In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM ’13)*. 2013, pp. 103–112.

[61] Sharon Nelson-Le Gall. “Help-seeking: An understudied problem-solving skill in children”. In: 1.3 (1981), pp. 224–246.

[62] Vahid Garousi and Barıs Kücük. “Smells in software test code: A survey of knowledge in industry and academia”. In: *Journal of Systems and Software* 138 (2018), pp. 52 –81.

[63] Vahid Garousi, Austen Rainer, Per Lauvås, and Andrea Arcuri. “Software-testing education: A systematic literature mapping”. In: *Journal of Systems and Software* 165 (2020), p. 110570.

[64] Alessio Gaspar, Sarah Langevin, Naomi Boyer, and Ralph Tindell. “A Preliminary Review of Undergraduate Programming Students’ Perspectives on Writing Tests, Working with Others, & Using Peer Testing”. In: *ACM SIGITE Conference on Information Technology Education*. 2013, pp. 109–114.

[65] David P. Gilliam, Thomas L. Wolfe, Josef S. Sherif, and Matt Bishop. “Software security checklist for the software life cycle”. In: *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003*. 2003, pp. 243–248.

[66] Michael H. Goldwasser. “A Gimmick to Integrate Software Testing Throughout the Curriculum”. In: *Technical Symposium on Computer Science Education*. SIGCSE ’02. 2002, pp. 271–275.

[67] O.C.Z. Gotel and C.W. Finkelstein. “An analysis of the requirements traceability problem”. In: *Proceedings of IEEE International Conference on Requirements Engineering*. 1994, pp. 94–101.

[68] Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C. Gall. “Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 336–347.

[69] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C. Gall. “Scented since the beginning: On the diffuseness of test smells in automatically generated test code”. In: *Journal of Systems and Software* 156 (2019), pp. 312–327.

[70] Giovanni Grano, Fabio Palomba, and Harald C. Gall. “Lightweight Assessment of Test-Case Effectiveness using Source-Code-Quality Indicators”. In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1.

[71] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. “Automated Detection of Test Fixture Strategies and Smells”. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 322–331.

[72] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. “Automatic query reformulations for text retrieval in software engineering”. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 842–851.

[73] Spencer E. Harpe. “How to analyze Likert and other rating scale data”. In: *Currents in Pharmacy Teaching and Learning* 7.6 (2015), pp. 836–850.

[74] Sarah Heckman, Jessica Young Schmidt, and Jason King. “Integrating Testing Throughout the CS Curriculum”. In: *Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2020, pp. 441–444.

[75] Johannes Holvitie, Teemu Rajala, Riku Haavisto, Erkki Kaila, Mikko-Jussi Laakso, and Tapio Salakoski. “Breaking the Programming Language Barrier: Using Program Visualizations to Transfer Programming Knowledge in One Programming Language to Another”. In: *2012 IEEE 12th International Conference on Advanced Learning Technologies*. 2012, pp. 116–120.

[76] Jeff Huang and Efthimis N. Efthimiadis. “Analyzing and Evaluating Query Reformulation Strategies in Web Search Logs”. In: *Proceedings of the ACM Conference on Information and Knowledge Management*. CIKM ’09. Hong Kong, China, 2009, pp. 77–86.

[77] Michael Hucka and Matthew J. Graham. “Software search is not a science, even among scientists: A survey of how scientists and engineers find software”. In: *Journal of Systems and Software* 141 (2018), pp. 171 –191.

[78] Laura Inozemtseva and Reid Holmes. “Coverage is Not Strongly Correlated with Test Suite Effectiveness”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, 435–445.

[79] Ursula Jackson, Bill Z. Manaris, and Renée A. McCauley. “Strategies for Effective Integration of Software Engineering Concepts and Techniques into the Undergraduate Computer Science Curriculum”. In: *Technical Symposium on Computer Science Education*. SIGCSE ’97. 1997, pp. 360–364.

[80] Matthew C Jadud. “A First Look at Novice Compilation Behaviour Using BlueJ”. In: *Computer Science Education* 15.1 (2005), pp. 25–40.

[81] Bernard J. Jansen, Amanda Spink, Chris Blakely, and Sherry Koshman. “Defining a Session on Web Search Engines: Research Articles”. In: *Journal of the American Society for Information Science and Technology* 58.6 (Apr. 2007), pp. 862–871.

[82] David Janzen and Hossein Saiedian. “Test-driven Learning in Early Programming Courses”. In: *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’08. Portland, OR, USA: ACM, 2008, pp. 532–536.

[83] Willy Jimenez, Amel Mammar, and Ana Cavalli. *Software Vulnerabilities, Prevention and Detection Methods: A Review*. 2009.

[84] Thorsten Joachims, Laura Granka, Bing Pan, Helene Hembrooke, Filip Radlinski, and Geri Gay. “Evaluating the Accuracy of Implicit Feedback from Clicks and Query Reformulations in Web Search”. In: *ACM Transactions on Information Systems* 25.2 (Apr. 2007).

[85] Brittany Johnson, Rahul Pandita, Emerson Murphy-Hill, and Sarah Heckman. “Bespoke Tools: Adapted to the Concepts Developers Know”. In: *Foundations of Software Engineering*. ESEC/FSE 2015. 2015, 878–881. ISBN: 9781450336758.

[86] Brittany Johnson, Rahul Pandita, Justin Smith, Denae Ford, Sarah Elder, Emerson Murphy-Hill, Sarah Heckman, and Caitlin Sadowski. “A Cross-Tool Communication Study on Program Analysis Tool Notifications”. In: *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, 73–84.

[87] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. “Why don’t software developers use static analysis tools to find bugs?” In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681.

[88] Edward L. Jones. “Integrating Testing into the Curriculum — Arsenic in Small Doses”. In: *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’01. New York, NY, USA: ACM, 2001, 337–341.

[89] Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. “Quantifying Incremental Development Practices and Their Relationship to Procrastination”. In: *ACM Conference on International Computing Education Research*. ICER ’17. 2017, 191–199.

[90] Dong Jae Kim. “An Empirical Study on the Evolution of Test Smell”. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2020, pp. 149–151.

[91] Dong Jae Kim, Tse-Hsun Chen, and Jinqiu Yang. “The secret life of test smells - an empirical study on test smell evolution and maintenance”. In: *Empirical Software Engineering* 26 (2021), p. 100.

[92] Rex B. Kline. *Principles and Practice of Structural Equation Modeling*. 4th. Guilford Publications, 2015.

[93] Amy J. Ko, Thomas D. LaToza, Stephen Hull, Ellen A. Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. “Teaching explicit programming strategies to adolescents”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 2019, pp. 469–475.

[94] Amy J. Ko and Brad A. Myers. “Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’04. Vienna, Austria: ACM, 2004, pp. 151–158.

[95] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. “Six Learning Barriers in End-User Programming Systems”. In: *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*. VLHCC ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 199–206.

[96] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks”. In: *IEEE Transactions on Software Engineering* 32.12 (Dec. 2006), pp. 971–987.

[97] Negar Koochakzadeh and Vahid Garousi. “TeCReVis: A Tool for Test Coverage and Test Redundancy Visualization”. In: *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*. TAIC PART’10. Windsor, UK: Springer-Verlag, 2010, 129–136.

[98] Ken Koster. “A State Coverage Tool for JUnit”. In: *Companion of the 30th International Conference on Software Engineering*. ICSE Companion ’08. 2008, pp. 965–966.

[99] Attila Kovács and Kristóf Szabados. “Test software quality issues and connections to international standards”. In: *Acta Universitatis Sapientiae, Informatica* 5 (May 2014), pp. 77–102.

[100] Filippo Lanubile and Teresa Mallardo. “Inspecting Automated Test Code: A Preliminary Study”. In: *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and Extreme Programming*. XP’07. Como, Italy: Springer-Verlag, 2007, 115–122.

[101] Jill H. Larkin and Herbert A. Simon. “Why a Diagram is (Sometimes) Worth Ten Thousand Words”. In: *Cognitive Science* 11.1 (1987), pp. 65–100.

[102] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Amy J Ko. “Explicit programming strategies”. In: *Empirical Software Engineering* 25.4 (2020), pp. 2416–2449.

[103] Otávio Augusto Lazzarini Lemos, Fábio Fagundes Silveira, Fabiano Cutigi Ferrari, and Alessandro Garcia. “The impact of Software Testing education on code reliability: An empirical assessment”. In: *Journal of Systems and Software* 137 (2018), pp. 497–511.

[104] Otavio Lemos, Fábio Silveira, Fabiano Ferrari, and Alessandro Garcia. “The Impact of Software Testing Education on Code Reliability: An Empirical Assessment”. In: *Journal of Systems and Software* (Mar. 2017).

[105] Colleen M. Lewis, Ken Yasuhara, and Ruth E. Anderson. "Deciding to Major in Computer Science: A Grounded Theory of Students' Self-Assessment of Ability". In: *Proceedings of the Seventh International Workshop on Computing Education Research*. ICER '11. Providence, Rhode Island, USA: ACM, 2011, 3–10.

[106] Dor D. Ma'ayan. "The Quality of Junit Tests: An Empirical Study Report". In: *International Workshop on Software Qualities and Their Dependencies*. SQUADE '18. ACM, 2018, pp. 33–36.

[107] Will Marrero and Amber Settle. "Testing First: Emphasizing Testing in Early Programming Courses". In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '05. Caparica, Portugal: ACM, 2005, pp. 4–8.

[108] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W. Price, and Tiffany Barnes. "Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science". In: *Proceedings of the 2020 ACM Conference on International Computing Education Research*. ICER '20. Virtual Event, New Zealand: ACM, 2020, 194–203.

[109] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W. Price, and Tiffany Barnes. "Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science". In: *2020 ACM Conference on International Computing Education Research*. ICER '20. 2020, 194–203.

[110] Samiha Marwan, Yang Shi, Ian Menezes, Min Chi, Tiffany Barnes, Thomas W Price, and North Carolina. "Just a Few Expert Constraints Can Help : Humanizing Data-Driven Subgoal Detection for Novice Programming". In: *International Conference on Educational Data Mining*. 2021, pp. 1–13.

[111] Philip Mayer and Alexander Bauer. "An Empirical Analysis of the Utilization of Multiple Programming Languages in Open Source Projects". In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. EASE '15. Nanjing, China: ACM, 2015, 4:1–4:10.

[112] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. "Reliability and Inter-Rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice". In: *Proceedings of the ACM on Human-Computer Interaction* 3.CSCW (2019).

[113] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi. "Building More Usable APIs". In: *IEEE Software* 15.3 (1998), pp. 78–86.

[114] David A. McMeekin, Brian R. von Konsky, Elizabeth Chang, and David J.A. Cooper. “Checklist Based Reading’s Influence on a Developer’s Understanding”. In: *19th Australian Conference on Software Engineering (aswec 2008)*. 2008, pp. 489–496.

[115] Hans Van der Meij. “Constraints on Question Asking in Classrooms.” In: *Journal of Educational Psychology 80, 3 (1988), 401–405.* (1988).

[116] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley Signature Series (Fowler). Pearson Education, 2007.

[117] Iain Milne and Glenn Rowe. “Difficulties in Learning and Teaching Programming—Views of Students and Tutors”. In: *Education and Information Technologies* 7 (2002), pp. 55–66.

[118] M. D. Mohamed Suffian, S. Ibrahim, and M. R. Abdullah. “A proposal of postgraduate programme for software testing specialization”. In: *2014 8th. Malaysian Software Engineering Conference (MySEC)*. 2014, pp. 342–347.

[119] H.James Nelson, Gretchen Irwin, and David E. Monarchi. “Journeys up the mountain: Different paths to learning object-oriented programming”. In: *Accounting, Management and Information Technologies* 7.2 (1997), pp. 53 –85.

[120] Helmut Neukirchen, Benjamin Zeiss, and Jens Grabowski. “An approach to quality engineering of TTCN-3 test specifications”. In: *International Journal on Software Tools for Technology Transfer* 10.4 (2008), p. 309.

[121] Sebastian P. Ng, Tafline Murnane, Karl Reed, Doug Grant, and Tsong Y. Chen. “A preliminary survey on software testing practices in Australia”. In: *2004 Australian Software Engineering Conference. Proceedings.* 2004, pp. 116–125.

[122] *Online regex tester, debugger with highlighting for PHP, PCRE, Python, Golang and JavaScript.* `https://regex101.com`.

[123] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. “On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study”. In: *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*. 2016, pp. 5–14.

[124] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. “Automatic Test Case Generation: What if Test Code Quality Matters?” In: *International Symposium on Software Testing and Analysis.* ISSTA 2016. 2016, pp. 130–141.

[125] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. “Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 523–533.

[126] Helen J Parkin, Stuart Hepplestone, Graham Holden, Brian Irwin, and Louise Thorpe. “A role for technology in enhancing students’ engagement with feedback”. In: *Assessment & Evaluation in Higher Education* 37.8 (2012), pp. 963–973.

[127] Chris Parnin and Christoph Treude. “Measuring API Documentation on the Web”. In: *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*. Web2SE ’11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 25–30.

[128] Joan Peckham, Peter Stephenson, Jean-Yves Hervé, Ron Hutt, and Miguel Encarnação. “Increasing Student Retention in Computer Science through Research Programs for Undergraduates”. In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’07. Covington, Kentucky, USA: ACM, 2007, 124–128.

[129] Fabiano Pecorelli. “Test-Related Factors and Post-Release Defects: An Empirical Study”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: ACM, 2019, 1235–1237.

[130] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. “On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study”. In: *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. CASCON ’19. Toronto, Ontario, Canada: IBM Corp., 2019, 193–202.

[131] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. “tsDetect: An Open Source Test Smells Detection Tool”. In: *Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. 2020.

[132] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. “TsDetect: An Open Source Test Smells Detection Tool”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: ACM, 2020, 1650–1654.

[133] Kai Petersen and Jefferson Seide Molléri. "Preliminary Evaluation of a Survey Checklist in the Context of Evidence-based Software Engineering Education". In: *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2021, Online Streaming, April 26-27, 2021*. Ed. by Raian Ali, Hermann Kaindl, and Leszek A. Maciaszek. SCITEPRESS, 2021, pp. 437–444.

[134] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. "An Empirical Study of Iterative Improvement in Programming Assignments". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: ACM, 2015, 410–415.

[135] Raphael Pham, Stephan Kiesling, Olga Liskin, Leif Singer, and Kurt Schneider. "Enablers, Inhibitors, and Perceptions of Testing in Novice Software Teams". In: *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. 2014, 30–40.

[136] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. "Understanding Myths and Realities of Test-suite Evolution". In: *Foundations of Software Engineering*. FSE '12. ACM, 2012, 33:1–33:11.

[137] *PITest: Mutation Operators*. `http://pitest.org/quickstart/mutators/`. Accessed: 2022-01-23.

[138] Joe Gibbs Politz, Shriram Krishnamurthi, and Kathi Fisler. "In-flow peer-review of tests in test-first programming". In: *tenth annual conference on International computing education research*. 2014, pp. 11–18.

[139] Upsorn Praphamontripong, Mark Floryan, and Ryan Ritzo. "A Preliminary Report on Hands-On and Cross-Course Activities in a College Software Testing Course". In: *Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2020, pp. 445–451.

[140] Thomas Price, Zhongxiu Peddycord-Liu, Veronica Catete, and Tiffany Barnes. "Factors Influencing Students' Help-Seeking Behavior while Programming with Human and Computer Tutors". In: Mar. 2017.

[141] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. "SCOTCH: Test-to-code traceability using slicing and conceptual coupling". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 63–72.

[142] Md Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayan, Federico Andres Lois, Sebastian Fernandez Quezada, Christopher Parnin, Kathryn T. Stolee, and Baishakhi Ray. “Evaluating how developers use general-purpose web-search for code retrieval”. In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. 2018, pp. 465–475.

[143] Mohammad M. Rahman, Chanchal K. Roy, and David Lo. “Automatic query reformulation for code search using crowdsourced knowledge”. In: *Empirical Software Engineering* (2019).

[144] Mohammad M. Rahman, Chanchal K. Roy, and David Lo. “RACK: Automatic API Recommendation Using Crowdsourced Knowledge”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 349–359.

[145] *Regular expression visualizer using railroad diagrams*. `https://regexper.com`.

[146] Stefan Reichhart. *Assessing Test Quality: TestLint*. Verlag nicht ermittelbar, 2007.

[147] Martin P. Robillard. “What Makes APIs Hard to Learn? Answers from Developers”. In: *IEEE Software* 26.6 (2009), pp. 27–34.

[148] Guoping Rong, Jingyi Li, Mingjuan Xie, and Tao Zheng. “The Effect of Checklist in Code Review for Inexperienced Students: An Empirical Study”. In: *2012 IEEE 25th Conference on Software Engineering Education and Training*. 2012, pp. 120–124.

[149] Per Runeson. “A survey of unit testing practices”. In: *IEEE Software* 23.4 (2006), pp. 22–29.

[150] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. “How Developers Search for Code: A Case Study”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy, 2015, pp. 191–201.

[151] Railana Santana, Luana Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor Costa, and Ivan Machado. “RAIDE: A Tool for Assertion Roulette and Duplicate Assert Identification and Refactoring”. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. SBES ’20. Natal, Brazil: ACM, 2020, 374–379.

[152] Simone C. dos Santos, Maria da Conceição Moraes Batista, Ana Paula C. Cavalcanti, Jones O. Albuquerque, and Silvio R.L. Meira. “Applying PBL in Software Engineering Education”. In: *2009 22nd Conference on Software Engineering Education and Training*. 2009, pp. 182–189.

[153] Lilian Passos Scatalon, Jeffrey C. Carver, Rogério Eduardo Garcia, and Ellen Francine Barbosa. “Software Testing in Introductory Programming Courses: A Systematic Mapping Study”. In: *ACM Technical Symposium on Computer Science Education*. SIGCSE ’19. 2019, pp. 421–427.

[154] Lilian Passos Scatalon, Jorge Marques Prates, Draylson Micael de Souza, Ellen Francine Barbosa, and Rogério Eduardo Garcia. “Towards the Role of Test Design in Programming Assignments”. In: *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*. 2017, pp. 170–179.

[155] Jean Scholtz and Susan Wiedenbeck. “Learning second and subsequent programming languages: A problem of transfer”. In: *International Journal of Human-Computer Interaction* 2.1 (1990), pp. 51–72.

[156] Zalia Shams and Stephen H. Edwards. “Toward Practical Mutation Analysis for Evaluating the Quality of Student-Written Software Tests”. In: *ACM Conference on International Computing Education Research*. ICER ’13. 2013, 53–58.

[157] Duane F. Shell, Leen-Kiat Soh, Abraham E. Flanigan, and Markeya S. Peteranetz. “Students’ Initial Course Motivation and Their Achievement and Retention in College CS1 Courses”. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (2016), 639–644.

[158] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. “Here We Go Again: Why is It Difficult for Developers to Learn Another Programming Language?” In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE ’20. Seoul, South Korea: ACM, 2020, 691–701.

[159] Susan Elliott Sim, Charles L. A. Clarke, and Richard C. Holt. “Archetypal source code searches: a survey of software developers and maintainers”. In: *Program Comprehension, 1998. IWPC ’98. Proceedings., 6th International Workshop on*. 1998, pp. 180–187.

[160] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. “How Well Do Search Engines Support Code Retrieval on the Web?” In: *ACM Transactions on Software Engineering and Methodology* 21.1 (Dec. 2011), 4:1–4:25.

[161] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. “An Examination of Software Engineering Work Practices”. In: *Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON ’97. Toronto, Ontario, Canada, 1997.

[162] Elvys Soares, Márcio Ribeiro, Guilherme Amaral, Rohit Gheyi, Leo Fernandes, Alessandro Garcia, Baldoino Fonseca, and André Santos. “Refactoring Test Smells: A Perspective from Open-Source Developers”. In: *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*. SAST 20. Natal, Brazil: ACM, 2020, 50–59.

[163] Bruno Pedraça de Souza, Rebeca Campos Motta, and Guilherme Horta Travassos. “The First Version of SCENARIotCHECK: A Checklist for IoT Based Scenarios”. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. SBES 2019. Salvador, Brazil: ACM, 2019, 219–223.

[164] Jaime Spacco and William Pugh. “Helping Students Appreciate Test-driven Development (TDD)”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 907–913.

[165] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. “On the Relation of Test Smells to Software Code Quality”. In: *International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 1–12.

[166] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. “On the Relation of Test Smells to Software Code Quality”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 1–12.

[167] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. “Investigating Severity Thresholds for Test Smells”. In: *Mining Software Repositories*. MSR ’20. 2020, 311–321.

[168] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. “Investigating Severity Thresholds for Test Smells”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR ’20. Seoul, Republic of Korea: ACM, 2020, 311–321.

[169] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. “Solving the Search for Source Code”. In: *ACM Transactions on Software Engineering and Methodology* 23.3 (June 2014), 26:1–26:45.

[170] Jeffrey Stylos and Brad A. Myers. “Mica: A Web-Search Tool for Finding API Components and Examples”. In: *Visual Languages and Human-Centric Computing (VL/HCC ’06)*. 2006, pp. 195–202.

[171] Matthew S. Thiese, Brenden Ronna, and Ulrike Ott. “P value interpretations and considerations”. In: *Journal of Thoracic Disease* 8.9 (2016).

[172] Joseph Timoney, Stephen Brown, and Deshi Ye. “Experiences in Software Testing Education: Some Observations from an International Cooperation”. In: *2008 The 9th International Conference for Young Computer Scientists*. 2008, pp. 2686–2691.

[173] Federico Tomassetti and Marco Torchiano. “An Empirical Assessment of Polyglotism in GitHub”. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’14. London, England, United Kingdom, 2014, 17:1–17:4.

[174] Ayse Tosun, Oscar Dieste, Davide Fucci, Sira Vegas, Burak Turhan, Hakan Erdogmus, Adrian Santos, Markku Oivo, Kimmo Toro, Janne Jarvinen, and Natalia Juristo. “An Industry Experiment on the Effects of Test-Driven Development on External Quality and Productivity”. In: *Empirical Software Engineering* 22.6 (2017), 2763–2805.

[175] Meng-Jung Tsai, Jyh-Chong Liang, Huei-Tse Hou, and Chin-Chung Tsai. “University students’ online information searching strategies in different search contexts”. In: *Australasian Journal of Educational Technology* 28.5 (2012).

[176] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. “An Empirical Investigation into the Nature of Test Smells”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, 4–15.

[177] Medha Umarji, Susan Elliott Sim, and Crista Lopes. “Archetypal Internet-Scale Source Code Searching”. In: *Open Source Development, Communities and Quality*. Ed. by Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, and Giancarlo Succi. Boston, MA: Springer US, 2008, pp. 257–263.

[178] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. “On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test”. In: *IEEE Transactions on Software Engineering* 33.12 (Dec. 2007), pp. 800–817.

[179] Jeffrey Voas. “How assertions can increase test effectiveness”. In: *IEEE Software* 14.2 (1997), pp. 118–119.

[180] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. "Predicting performance in an introductory programming course by logging and analyzing student programming behaviour." In: *Proceedings of the 2013 IEEE 13th International Conference on Advanced Learning Technologies (ICALT 2013)*. Outstanding Paper Award. Piscataway, NJ: IEEE Computer Society, 2013, pp. 319–323.

[181] Christopher Watson, Frederick W.B. Li, and Jamie L. Godwin. "No Tests Required: Comparing Traditional and Dynamic Predictors of Programming Success". In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. Atlanta, Georgia, USA: ACM, 2014, 469–474.

[182] Michael Wick, Daniel Stevenson, and Paul Wagner. "Using Testing and JUnit across the Curriculum". In: *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '05. St. Louis, Missouri, USA: ACM, 2005, 236–240.

[183] Laurie Williams, E. Michael Maximilien, and Mladen Vouk. "Test-driven development as a defect-reduction practice". In: *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* 2003, pp. 34–45.

[184] Laurie Williams and Richard L. Upchurch. "In Support of Student Pair-Programming". In: *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '01. Charlotte, North Carolina, USA: ACM, 2001, 327–331.

[185] John Wrenn and Shriram Krishnamurthi. "Executable Examples for Programming Problem Comprehension". In: *ACM Conference on International Computing Education Research*. ICER '19. 2019, 131–139.

[186] Heather M. Wright and N. Burçin Tamer. "Can Sending First and Second Year Computing Students to Technical Conferences Help Retention?" In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Minneapolis, MN, USA: ACM, 2019, 56–62.

[187] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. "What Do Developers Search for on the Web?" In: *Empirical Softw. Engg.* 22.6 (Dec. 2017), pp. 3149–3185.

[188] Tao Xie, Darko Marinov, and David Notkin. *Improving Generation of Object-Oriented Test Suites by Avoiding Redundant Tests*. Tech. rep. 2004.

[189] Jun Xu, Wei Wu, Hang Li, and Gu Xu. "A Kernel Approach to Addressing Term Mismatch". In: *Proceedings of the International Conference Companion on World Wide Web*. WWW '11. Hyderabad, India, 2011, pp. 153–154.

[190] Xiangmin Zhang, H.G.B. Anghelescu, and Xiao-Jun Yuan. “Domain knowledge, search behaviour, and search effectiveness of engineering and science students: An exploratory study”. In: *Information Research* 10 (Jan. 2005).

[191] Le Zhao and Jamie Callan. “Automatic Term Mismatch Diagnosis for Selective Query Expansion”. In: *Proceedings of the International Conference on Research and Development in Information Retrieval*. SIGIR ’12. Portland, Oregon, USA, 2012, pp. 515–524.

APPENDICES

APPENDIX

A

STUDY MATERIALS FOR CHAPTER 3

A.1 Study 1: "Exploring Tools and Strategies Used During Regular Expression Composition Tasks"

Artifacts are available at

`https://github.com/softwarekitty/regexCompositionStudy`

A.1.1 Regular Expression Composition Tasks

Figures A.1 and A.2 contain the example of *Validation* type task, which used the `Pattern.matches()` function to validate the entire content of a string against a regular expression.

```java
/** A line of text will contain at most one newline and only then at the end of
 * the string (this input will not have multiple lines). This function should
 * take one line of text and verify that the entire string is composed of one
 * valid email. Extra characters like whitespace before or after, or anything
 * that would invalidate the email are not allowed (except newline at the end).
 */
public class ValidEmail {
   public boolean isValidEmail(String line) {

   // TODO compose a regex to complete the challenge
      String regex = "";
      return Pattern.matches(regex, line);
   }
}
```

Figure A.1: Code and Description of Task ValidEmail.

```java
public class ValidEmailTest {

   private static ValidEmail validEmail = null;

   @BeforeClass
   public static void setup() {
      validEmail = new ValidEmail();
   }

   @Test
   public void testIsValidEmail_1() {
      //a typical email
      String anyLine = "###/+-?^_`{|}~$$$***@weird.do";
      boolean correctAnswer = true;
      boolean compositionAnswer = validEmail.isValidEmail(anyLine);
      assertTrue(compositionAnswer == correctAnswer);
   }

   // ... More tests, eight test cases in total
}
```

Figure A.2: One Sample Associated JUnit Test of Task ValidEmail.

Figures A.3 and A.4 contain the example of *Extraction* type task, which used `Matcher.find()` to examine the strings against a given pattern and extract pertinent information from a string subject to a regular expression.

```java
/** Returns true if any alphanumeric word in the text contains no vowels. So
 * the strange sentence: "I have ctmpts training to go to!" should return true.
 * Input does not need to be a sentence, and words are separated by whitespace
 * as usual, ignoring punctuation Words are composed of 2 or more lowercase or
 * uppercase letters.
 */
public class NoVowelsWord {
   public boolean hasNoVowelsWord(String content) {

      // TODO compose a regex to complete the challenge
      String regex = "";
      Pattern pattern = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
      Matcher matcher = pattern.matcher(content);
      return matcher.find();
   }
}
```

Figure A.3: Code and Description of Task NoVowelsWord.

```java
public class NoVowelsWordTest {

   private static NoVowelsWord noVowelsWord = null;

   @BeforeClass
   public static void setup() {
      noVowelsWord = new NoVowelsWord();
   }

   @Test
   public void testHasNoVowelsWord_1() {
      //the word 'CTMPTS' has no vowels
      String anyLine = "April and Ron are taking CTMPTS training";
      boolean correctAnswer = true;
      boolean compositionAnswer = noVowelsWord.hasNoVowelsWord(anyLine);
      assertTrue(compositionAnswer==correctAnswer);
   }

   // ... More tests, five test cases in total
}
```

Figure A.4: One Sample Associated JUnit Test of Task NoVowelsWord.

A.1.2 Survey Questions

1. **What year are you in school (closest match)?**
   (a) Freshmen
   (b) Sophomore
   (c) Junior
   (d) Senior
   (e) Graduate Student
2. **How would you rate your Java programming experience?**
   (a) Novice
   (b) Intermediate
   (c) expert
3. **What is your experience with regular expressions?**
   (a) I have never heard of them
   (b) I have heard of them, but have no experience
   (c) I have used them once or twice
   (d) I use them regularly
   (e) I am an expert
4. **What is your experience with JUnit tests?**
   (a) I have never heard of them
   (b) I have heard of them, but have no experience
   (c) I have used them once or twice
   (d) I use them regularly
   (e) I am an expert
5. **How frequently do you use web searches while programming?**
   (a) I have never heard of them
   (b) I have heard of them, but have no experience

(c) I have used them once or twice

(d) I use them regularly

(e) I am an expert

6. **How many years have you been programming?**

   ____

7. **How many years have you been programming in Java?**

   ____

A.2 Study 2: "How Graduate Computing Students Search When Using an Unfamiliar Programming Language"

Artifacts are available at:

`https://github.com/ginaBai/CodeSearch_VBA`

A.2.1 VBA Programming Tasks

Students were instructed to complete the following tasks in order:

Task 1 provides participants a list of numbers in a column, and asks participants to record a macro and create a button that returns the average value of the numbers in this column when the button is clicked. This task can be completed with or without VBA code.

Task 2 requires participants to create a button that creates a message box with "Hello World!" on it once clicked.

Task 3 provides participants a list of scores in a column called *Grade*, ranging from 43 to 100, and ask participants to create a button that populates a column called *P/F* with P or F for Pass or Fail, respectively, depending on *Grade*: P is for a score that is greater than or equal to 60 in column *Grade*, and F otherwise.

Task 4 provides participants an unsorted inventory list of unique labels in a column called *Item* with their prices listed in a column called *Price*. Their goal is, given a separate target list of labels, to create a button that populates a column *Price* for the items in the target list based on the inventory list. The Excel built-in function `VLOOKUP` is not allowed in this task.

Task 5 provides participants a list of phone numbers in the format of *(XXX)XXX-XXXX*, and asks participants to create a button that populates a column called *Area Code* with unique area codes and their associated counts in a column called *Count*.

A.2.2 Survey Questions

There are four types of surveys in this study: 1) *Preliminary Survey* (Appendix A.2.2.1), 2) *Context Survey* (Appendix A.2.2.2), 3) *Query Reformulation Survey* (Appendix A.2.2.3), and 4) *Search Tab Closed Survey* (Appendix A.2.2.4).

A.2.2.1 Preliminary Survey

1. **How would you rate your overall programming experience?**

   (a) Novice

   (b) Intermediate

   (c) Expert

2. **What languages do you work with most frequently?**

   ☐ Java ☐ C ☐ C++ ☐ Python ☐ C# ☐ PHP ☐ JavaScript
   ☐ Visual Basic .Net ☐ Perl ☐ Assembly ☐ Ruby ☐ Delphi
   ☐ Object Pascal ☐ Swift ☐ Objective C ☐ MatLab ☐ R
   ☐ PL/SQL ☐ COBOL ☐ Groovy ☐ Other: ________________

3. **In what contexts or for what purpose(s) do you search for codes via websites during software development activities?**

   ____
   ____
   ____
   ____

4. **When solving a programming/development/technical problem, how often do you: (rank the following from most often to least often with number 1, 2, and 3)**

   (a) **Ask a colleague/friend**
   i. 1 ii. 2 iii. 3

   (b) **Web or online search**
   i. 1 ii. 2 iii. 3

   (c) **IDE search/exploration**
   i. 1 ii. 2 iii. 3

   (d) **Other:** ______________________

5. **What's your occupation?**

   (a) Undergraduate Student

   (b) Graduate Student

(c) Student with working experience in the industry

(d) Other: ____________

6. **How would you identify your gender?**

   (a) Female

   (b) Male

   (c) Non-binary

   (d) Prefer not to disclose

A.2.2.2 Context Survey

1. **What languages is your search related to?**
   (separate each language with a newline [hitting the enter key])

   ____

2. **What are you doing?**

   (a) Exploring (Looking into a certain programming language, etc.)

   (b) Debugging code (Why does ABC return null?, How to use XYZ debugger?, etc.)

   (c) Designing/Developing a feature

   (d) Maintaining old code

   (e) Other: ____________

3. **How familiar are you with the code you're looking for?**

   (a) Very Familiar (e.g., I've written something similar in the past)

   (b) Somewhat Familiar (e.g., I've seen something like it before)

   (c) Not familiar (e.g., this is entirely new to me)

4. **What else have you tried to answer this question?**

   ☐ Searched local codebase

   ☐ Consulting a friend/colleague in person

   ☐ Consulting a friend/colleague via email, IM, etc.

   ☐ Consulted a book/textbook

5. **What question are you trying to answer?**

____

____

____

6. **What type of code are you working on?**

   (a) Backend (database, cloud storage, etc.)

   (b) Front-end GUI

   (c) Middleware

   (d) Other: ____________

A.2.2.3 Query Reformulation Survey

1. **When you searched, [Previous Query], did you solve the problem?**

   (a) Yes (jump to Q2-yes)

   (b) No (jump to Q2-no)

2. **(yes) What solved it?**

   ☐ Found a needed API (how to modify CSS with jQuery)

   ☐ Found a needed implementation (bubble sort implementation in Java)

   ☐ Asked a colleague in person

   ☐ Asked a colleague online/email

   ☐ Other method: ____________

   (no) Is your search, [Current Query], related to the problem?

   (a) Yes (jump to Q3)

   (b) No

3. **What do you need to solve the earlier query?**

____

____

____

A.2.2.4 Search Tab Closed Survey

1. **When you searched, [Previous Query], did you solve the problem?**

   (a) Yes (jump to Q2-yes)

   (b) No (jump to Q2-no)

2. **(yes) What solved it?**

   ☐ Found a needed API (how to modify CSS with jQuery)

   ☐ Found a needed implementation (bubble sort implementation in Java)

   ☐ Asked a colleague in person

   ☐ Asked a colleague online/email

   ☐ Other method: ____________

   (no) Is your search, [Current Query], related to the problem?

   (a) Need more related information

   (b) Need to reformulate the query

   (c) Other: ____________

APPENDIX

B

STUDY MATERIALS FOR CHAPTER 4

B.1 Study 1: “How Students Unit Test: Perceptions, Practices, and Pitfalls”

Artifacts are available at

`https://github.com/ginaBai/TestingPerformanceStudy`

In this study, students were instructed to complete a Test Perceptions Survey (preliminary survey, Appendix B.1.1) and a Post-activity Survey (Appendix B.1.2).

B.1.1 Test Perceptions Survey (Preliminary Survey)

1. **Please enter your email address**

   ____

2. **How do you test your own code? (Check ALL that apply)**
   If you test differently depending on the language or programming environment, please explain.

   ☐ Print statements

   ☐ Unit tests

   ☐ Testing tools

   ☐ Other: ____________

3. **Do you have experience with creating/editing/maintaining unit tests?**

   (a) Yes

   (b) No

4. **Please rank the following motivations for writing unit tests based on your experience (Drag up or down to reorder).**

   (a) To ensure that a unit functions as expected

   (b) To accept a unit from other sources

   (c) To specify a unit (test first)

   (d) To improve the program quality in general

   (e) To meet program requirements

   (f) Other: ____________

5. **How do you spend your program development time (in percentage)?**

   (a) Writing new code: ____________

   (b) Writing new tests: ____________

   (c) Debugging and fixing: ____________

   (d) Refactoring: ____________

   (e) Other: ____________

6. **How important are the following aspects for you when you write new unit tests?**

	Not at all important	Slightly important	Moderately important	Very important	Extremely important
Code coverage	◯	◯	◯	◯	◯
Execution speed	◯	◯	◯	◯	◯
Robustness against code changes	◯	◯	◯	◯	◯
How realistic the test scenario is	◯	◯	◯	◯	◯
How easily faults can be localised/debugged if the test fails	◯	◯	◯	◯	◯
How easily the test can be updated when the underlying code changes	◯	◯	◯	◯	◯
Sensitivity against code changes	◯	◯	◯	◯	◯

7. **Please select which techniques you apply when writing new tests (Check ALL that apply)**

☐ Automated test generation

☐ Code coverage analysis

☐ Mutation analysis

☐ Test-driven development

☐ Systematic testing approaches

☐ Mocking/stubbing

☐ Other: ____________

☐ Never

8. **What do you use automated unit test generation for? (Check ALL that apply)**

☐ Exercising specifications

☐ Exercising assertions in the code

☐ Exercising parameterised unit tests

☐ Finding crashes and undeclared exceptions

☐ Regression testing

☐ To complement manually written tests

☐ Other: ____________

9. **Please rank the following aspects of writing a new unit test according to their difficulty (Drag up or down to reorder)**

   (a) Determining what to check

   (b) Finding and creating relevant input values

   (c) Identifying which code/scenarios to test

   (d) Finding a sequence of calls to bring the unit under test into the target state

   (e) Isolating the unit under test

10. **What makes it difficult to fix a failing test? Please rank by importance (Drag up or down to reorder)**

    (a) The test reflects outdated behavior

    (b) The test is difficult to understand

    (c) The code under test is difficult to understand

    (d) The test reflects unrealistic behavior

    (e) The test is flaky (it fails non-deterministically)

11. **Please indicate your level of agreement with the following statements**

	Strongly disagree	Disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Agree	Strongly agree
Writing unit tests is difficult	○	○	○	○	○	○	○
I enjoy writing unit tests	○	○	○	○	○	○	○
I would like to have more tool support when writing unit tests	○	○	○	○	○	○	○
I would like to have more unit tests	○	○	○	○	○	○	○
Maintaining unit tests is difficult	○	○	○	○	○	○	○
I usually have sufficiently many unit tests	○	○	○	○	○	○	○

B.1.2 Post-activity Survey

1. **Please enter your email address**

   ____

2. **What challenge(s) did you encounter when creating/editing the test cases?**

   ____

   ____

   ____

   ____

3. **Please name one thing you would have like to receive help with during the study**

   ____

   ____

   ____

   ____

4. **What's your occupation?**

   (a) Undergraduate student

   (b) Graduate student

   (c) Student with working experience in the industry

   (d) University faculty

   (e) Professional developer/tester

   (f) Other: ____________

5. **How many years of programming experience do you have?**

   ____

6. **How do you describe your experience in programming?**

   (a) No experience

   (b) Novice

   (c) Advanced beginner

(d) Competent

(e) Proficient

(f) Expert

7. **How many years of programming experience with Java do you have?**

____

8. **How do you describe your experience in Java?**

   (a) No experience

   (b) Novice

   (c) Advanced beginner

   (d) Competent

   (e) Proficient

   (f) Expert

9. **How many years of experience with unit testing do you have?**

____

10. **How do you describe your experience in unit testing?**

   (a) No experience

   (b) Novice

   (c) Advanced beginner

   (d) Competent

   (e) Proficient

   (f) Expert

11. **Have you ever taken or are you currently taking a Computer Science course that involves Unit Testing?**

   (a) Yes

   (b) No

12. **Have you ever taken or are you currently taking a Computer Science course on Software Testing?**
    (a) Yes
    (b) No
13. **How would you identify your gender?**
    (a) Woman
    (b) Man
    (c) Non-binary
    (d) Prefer not to disclose
    (e) Prefer to self-describe: ____________

B.2 Study 2: "Is Assertion Roulette still a test smell? An experiment from the perspective of testing education"

Artifacts are available at

`https://github.com/ginaBai/AssertionRouletteStudy`

In this study, students were instructed to complete the preliminary survey at least one day prior to the lab session, and they were guided to fill out an identical post-activity survey at the end of the lab session.

B.2.1 Preliminary and Post-activity Surveys

1. **Please describe your ability in Computer Science:**

    (a) Considerably below average

    (b) Moderately below average

    (c) Slightly below average

    (d) Average

    (e) Slightly above average

    (f) Moderately above average

    (g) Considerably above average

2. **How likely would you be to take another course in Computer Science?**

    (a) Highly unlikely

    (b) Moderately unlikely

    (c) Slightly unlikely

    (d) Neither likely nor unlikely

    (e) Slightly likely

    (f) Moderately likely

    (g) Highly likely

3. **How likely would you be to minor in Computer Science?**

   (a) Highly unlikely
   (b) Moderately unlikely
   (c) Slightly unlikely
   (d) Neither likely nor unlikely
   (e) Slightly likely
   (f) Moderately likely
   (g) Highly likely

4. **How likely would you be to major in Computer Science?**

   (a) Highly unlikely
   (b) Moderately unlikely
   (c) Slightly unlikely
   (d) Neither likely nor unlikely
   (e) Slightly likely
   (f) Moderately likely
   (g) Highly likely

APPENDIX

C

STUDY MATERIALS FOR CHAPTER 5

C.1 Study 1: "Check It Off: Exploring the Impact of a Checklist Intervention on the Quality of Student-authored Unit Tests"

Artifacts are available at

`https://github.com/ginaBai/TestingChecklistStudy`

After stratifying the participants based on level (graduate or undergraduate), we randomly assigned students to an experimental group (the Checklist group) or to a control group (the Coverage group). Students in the Checklist group received a Preliminary Survey (Appendix C.1.1) and a Post-activity Survey (Appendix C.1.2), while the students in the Coverage group received the same Preliminary Survey but a different version of Post-activity Survey (Appendix C.1.3).

C.1.1 Preliminary Survey

1. **Please enter your email address**

   ____

2. **How do you test your own code? (Check ALL that apply)**
   If you test differently depending on the language or programming environment, please explain.

   ☐ Print statements

   ☐ Unit tests

   ☐ Testing tools

   ☐ Other: ____________

3. **Do you have experience with creating/editing/maintaining unit tests?**

   (a) Yes

   (b) No

4. **Please rank the following motivations for writing unit tests based on your experience (Drag up or down to reorder).**

   (a) To ensure that a unit functions as expected

   (b) To accept a unit from other sources

   (c) To specify a unit (test first)

   (d) To improve the program quality in general

   (e) To meet program requirements

   (f) Other: ____________

5. **How do you spend your program development time (in percentage)?**

   (a) Writing new code: ____________

   (b) Writing new tests: ____________

   (c) Debugging and fixing: ____________

   (d) Refactoring: ____________

   (e) Other: ____________

6. **How important are the following aspects for you when you write new unit tests?**

	Not at all important	Slightly important	Moderately important	Very important	Extremely important
Code coverage	◯	◯	◯	◯	◯
Execution speed	◯	◯	◯	◯	◯
Robustness against code changes	◯	◯	◯	◯	◯
How realistic the test scenario is	◯	◯	◯	◯	◯
How easily faults can be localised/debugged if the test fails	◯	◯	◯	◯	◯
How easily the test can be updated when the underlying code changes	◯	◯	◯	◯	◯
Sensitivity against code changes	◯	◯	◯	◯	◯

7. **Please select which techniques you apply when writing new tests (Check ALL that apply)**

☐ Automated test generation

☐ Code coverage analysis

☐ Mutation analysis

☐ Test-driven development

☐ Systematic testing approaches

☐ Mocking/stubbing

☐ Other: ____________

☐ Never

8. **What do you use automated unit test generation for? (Check ALL that apply)**

☐ Exercising specifications

☐ Exercising assertions in the code

☐ Exercising parameterised unit tests

☐ Finding crashes and undeclared exceptions

☐ Regression testing

☐ To complement manually written tests

☐ Other: ___________

9. **Please rank the following aspects of writing a new unit test according to their difficulty (Drag up or down to reorder)**

    (a) Determining what to check

    (b) Finding and creating relevant input values

    (c) Identifying which code/scenarios to test

    (d) Finding a sequence of calls to bring the unit under test into the target state

    (e) Isolating the unit under test

10. **What makes it difficult to fix a failing test? Please rank by importance (Drag up or down to reorder)**

    (a) The test reflects outdated behavior

    (b) The test is difficult to understand

    (c) The code under test is difficult to understand

    (d) The test reflects unrealistic behavior

    (e) The test is flaky (it fails non-deterministically)

11. **Please indicate your level of agreement with the following statements**

	Strongly disagree	Disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Agree	Strongly agree
Writing unit tests is difficult	○	○	○	○	○	○	○
I enjoy writing unit tests	○	○	○	○	○	○	○
I would like to have more tool support when writing unit tests	○	○	○	○	○	○	○
I would like to have more unit tests	○	○	○	○	○	○	○
Maintaining unit tests is difficult	○	○	○	○	○	○	○
I usually have sufficiently many unit tests	○	○	○	○	○	○	○

C.1.2 Post-activity Survey - Checklist/Experimental Group

1. **Please enter your email address**

   ____

2. **What challenge(s) did you encounter when creating/editing the test cases?**

   ____
   ____
   ____
   ____

3. **Please name one thing you would have like to receive help with during the study**

   ____
   ____
   ____
   ____

4. **Did you use the checklist during the study?**

   (a) Yes

   (b) No (jump to Q10)

5. **How did you use the checklist? (Check ALL that apply)**

   ☐ I read the checklist before I wrote any unit tests

   ☐ I frequently consulted the checklist during unit testing

   ☐ I consulted the checklist during unit testing, but not regularly

   ☐ I used the checklist as a guide on what to test next

   ☐ I used the checklist as a guide on when to stop unit testing

   ☐ I walked through the checklist before I submitted the test code

6. **How many unit tests (approximately, in percentage) in your test code satisfy the checklist items?**

   ____

7. **Which items in the checklist were most useful? (Check ALL that apply)**

   ☐ The test case is executable (i.e., it has an `@Test` annotation and can be run via "Run as JUnit Test")

   ☐ The test case has at least one assert statement or asserts an exception is thrown

   ☐ The test case evaluates/tests only one method

   ☐ The test case is descriptively named and commented

   ☐ If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using `@Before`)

   ☐ If there are too many assert statements in a single test case (e.g., more than 5), you might split it up so each test evaluates one behavior

   ☐ The test suite has at least one test for each requirement

   ☐ The test suite appropriately uses the setup and teardown code (e.g., `@Before`, which runs before each `@Test`)

   ☐ The test suite contains a fault-revealing test for each bug in the code (i.e., a test that fails)

   ☐ For each requirement, the test suite contains unit tests for: Valid inputs

   ☐ For each requirement, the test suite contains unit tests for: Boundary cases

   ☐ For each requirement, the test suite contains unit tests for: Invalid inputs

   ☐ For each requirement, the test suite contains unit tests for: Expected exceptions

   ☐ To improve the test suite, you could measure code coverage using an appropriate tool, such as EclEmma (installation, tutorial). Inspect uncovered code and write tests as appropriate.

8. **How would you rate this checklist in terms of helpfulness?**

   (a) Extremely helpful

   (b) Very helpful

   (c) Moderately helpful

   (d) Slightly helpful

   (e) Not at all helpful

9. **Was anything about the checklist unclear or confusing?** (skip Q10)

____
____
____
____

10. **Please briefly explain why you chose not to use the checklist**

____
____
____
____

11. **If you could change just one thing about this checklist, what would it be?**

____
____
____
____

12. **What's your occupation?**

    (a) Undergraduate student

    (b) Graduate student

    (c) Student with working experience in the industry

    (d) University faculty

    (e) Professional developer/tester

    (f) Other: ____________

13. **How many years of programming experience do you have?**

____

14. **How do you describe your experience in programming?**

    (a) No experience

    (b) Novice

(c) Advanced beginner

(d) Competent

(e) Proficient

(f) Expert

15. **How many years of programming experience with Java do you have?**

    ____

16. **How do you describe your experience in Java?**

    (a) No experience

    (b) Novice

    (c) Advanced beginner

    (d) Competent

    (e) Proficient

    (f) Expert

17. **How many years of experience with unit testing do you have?**

    ____

18. **How do you describe your experience in unit testing?**

    (a) No experience

    (b) Novice

    (c) Advanced beginner

    (d) Competent

    (e) Proficient

    (f) Expert

19. **Have you ever taken or are you currently taking a Computer Science course that involves Unit Testing?**

    (a) Yes

    (b) No

20. **Have you ever taken or are you currently taking a Computer Science course on Software Testing?**
    (a) Yes
    (b) No
21. **How would you identify your gender?**
    (a) Woman
    (b) Man
    (c) Non-binary
    (d) Prefer not to disclose
    (e) Prefer to self-describe: ____________

C.1.3 Post-activity Survey - Coverage/Control Group

1. **Please enter your email address**

   ____

2. **What challenge(s) did you encounter when creating/editing the test cases?**

   ____
   ____
   ____
   ____

3. **Please name one thing you would have like to receive help with during the study**

   ____
   ____
   ____
   ____

4. **Did you use the checklist during the study?**

    (a) Yes, I used EclEmma

    (b) Yes, I used (please type in the name of the coverage tool): ____________

    (c) No (jump to Q7)

5. **How did you use the coverage tool? (Check ALL that apply)**

    ☐ I ran the coverage tool before I wrote any unit tests

    ☐ I frequently ran the coverage tool during unit testing

    ☐ I ran the coverage tool during unit testing, but not regularly

    ☐ I used the coverage tool as a guide on what to test next

    ☐ I used the coverage tool as a guide on when to stop unit testing

    ☐ I measured my unit tests with the coverage tool before I submitted the test code

6. **How would you rate this coverage tool in terms of helpfulness?** (jump to Q9)

    (a) Extremely helpful

    (b) Very helpful

    (c) Moderately helpful

    (d) Slightly helpful

    (e) Not at all helpful

7. **Please briefly explain why you chose not to use the coverage tool**

    ____

    ____

    ____

    ____

8. **Would a checklist like this help you unit testing during the study? (attached: screenshot of the checklist)**

    (a) Yes

    (b) Maybe

    (c) No

9. **What's your occupation?**
    (a) Undergraduate student
    (b) Graduate student
    (c) Student with working experience in the industry
    (d) University faculty
    (e) Professional developer/tester
    (f) Other: ____________
10. **How many years of programming experience do you have?**

    ____
11. **How do you describe your experience in programming?**
    (a) No experience
    (b) Novice
    (c) Advanced beginner
    (d) Competent
    (e) Proficient
    (f) Expert
12. **How many years of programming experience with Java do you have?**

    ____
13. **How do you describe your experience in Java?**
    (a) No experience
    (b) Novice
    (c) Advanced beginner
    (d) Competent
    (e) Proficient
    (f) Expert

14. **How many years of experience with unit testing do you have?**

    ____

15. **How do you describe your experience in unit testing?**

    (a) No experience
    (b) Novice
    (c) Advanced beginner
    (d) Competent
    (e) Proficient
    (f) Expert

16. **Have you ever taken or are you currently taking a Computer Science course that involves Unit Testing?**

    (a) Yes
    (b) No

17. **Have you ever taken or are you currently taking a Computer Science course on Software Testing?**

    (a) Yes
    (b) No

18. **How would you identify your gender?**

    (a) Woman
    (b) Man
    (c) Non-binary
    (d) Prefer not to disclose
    (e) Prefer to self-describe: ____________

C.2 Study 2: "Introducing Explicit Testing Strategies to Advanced Beginners"

Artifacts are available at

`https://github.com/ginaBai/ExplicitTestingStrategiesStudy`

The Explicit Testing Strategy checklists can be found as `Issues`.

C.2.1 Short Survey

1. **What's your Unity ID (e.g., rbai2)?**

   ____

2. **Lab Section Lab Repo (e.g., CSC217-LL-200-1)**

   ____

3. **How often did you consult the checklist...**

	Never	Rarely	Sometimes	Often	Always
during system testing	○	○	○	○	○
during unit testing	○	○	○	○	○
when discussing test design with your teammates	○	○	○	○	○

4. **How would you rate this checklist in terms of helpfulness...**

	Not at all helpful	Slightly helpful	Moderately helpful	Very helpful	Extremely helpful
during system testing	○	○	○	○	○
during unit testing	○	○	○	○	○
when discussing test design with your teammates	○	○	○	○	○

5. **Do you have any suggestions on how to make the checklist more useful? Or do you have any other comments about the checklist?**

____

____

____

____

C.2.2 Comprehensive Survey

1. **What's your Unity ID (e.g., rbai2)?**

____

2. **Lab Section Lab Repo (e.g., CSC217-LL-200-1)**

____

3. **How often did you consult the checklist...**

	Never	Rarely	Sometimes	Often	Always
during system testing	◯	◯	◯	◯	◯
during unit testing	◯	◯	◯	◯	◯
when discussing test design with your teammates	◯	◯	◯	◯	◯

4. **How would you rate this checklist in terms of helpfulness...**

	Not at all helpful	Slightly helpful	Moderately helpful	Very helpful	Extremely helpful
during system testing	◯	◯	◯	◯	◯
during unit testing	◯	◯	◯	◯	◯
when discussing test design with your teammates	◯	◯	◯	◯	◯

5. **Do you have any suggestions on how to make the checklist more useful? Or do you have any other comments about the checklist?**

   ____

   ____

   ____

6. **Which items in the checklist were most useful? (Check ALL that apply)**

   ☐ The test case is executable (i.e., it has an `@Test` annotation and can be run via "Run as JUnit Test")

   ☐ The test case has at least one assert statement or asserts an exception is thrown

   ☐ The test case evaluates/tests only one method

   ☐ The test case is descriptively named and commented

   ☐ If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using `@Before`)

   ☐ If there are too many assert statements in a single test case (e.g., more than 5), you might split it up so each test evaluates one behavior

   ☐ The test suite has at least one test for each requirement

   ☐ The test suite appropriately uses the setup and teardown code (e.g., `@Before`, which runs before each `@Test`)

   ☐ The test suite contains a fault-revealing test for each bug in the code (i.e., a test that fails)

   ☐ Explicit testing strategies for the lab assignments

   ☐ To improve the test suite, you could measure code coverage using an appropriate tool, such as EclEmma (installation, tutorial). Inspect uncovered code and write tests as appropriate.

7. **How likely would you be to...**

	Extremely likely	Somewhat likely	Neither likely nor unlikely	Somewhat unlikely	Extremely unlikely
continue to use the checklist in future lab assignments	○	○	○	○	○
recommend this checklist to your peers/friends in CS	○	○	○	○	○

8. **Was anything about the checklist unclear or confusing?**

    ____

9. **If you could change just one thing about this checklist, what would it be?**

10. **Would you be available to have a short interview (15-20 min, via zoom) with us to share your experience using the checklist and improve its design?**

    (a) Yes

    (b) No

11. **What's your gender?**

    (a) Woman

    (b) Man

    (c) Two spirit

    (d) Non-binary

    (e) Prefer not to disclose

    (f) Prefer to self-describe: ____________

C.2.3 Explicit Testing Strategies Designed for Labs 7-11

This section includes the explicit testing strategies that we designed for Labs 7-11.

C.2.3.1 Explicit Testing Strategies Designed for Lab 7

- ☐ Test the `Schedule` class (*Explicit Testing Strategies for Lab 7*)
  - ☐ When testing the constructor `Schedule.Schedule()`, consider the following scenarios:
    - ☐ Length of the newly constructed Schedule
    - ☐ Title of the newly constructed Schedule
  - ☐ When testing `Schedule.addCourseToSchedule(Course)`, consider the following scenarios:
    - ☐ Valid addition
      - ☐ Add at least two courses for better practice
    - ☐ Invalid addition, causing an exception
      - ☐ Null
      - ☐ Duplicates
  - ☐ When testing `Schedule.removeCourseFromSchedule(Course)`, consider the following scenarios:
    - ☐ Valid removal
      - ☐ Remove all courses
      - ☐ Remove a non-existing course
    - ☐ Invalid removal, causing an exception
      - ☐ Remove a course from an empty Schedule
  - ☐ When testing `Schedule.resetSchedule()`, consider the following scenarios:
    - ☐ Length of the Schedule after being reset
    - ☐ Functionality after being reset, such as add and remove
  - ☐ When testing `Schedule.getScheduledCourses()`, consider the following scenarios:
    - ☐ Length of the Schedule
    - ☐ Value at each index of the Schedule
  - ☐ When testing `Schedule.setTitle(String)`, consider the following scenarios:
    - ☐ Valid title
    - ☐ Invalid title (e.g., null), causing an exception
- ☐ Test the `ArrayList` class (*Explicit Testing Strategies for Lab 7*)
  - ☐ When testing the constructor `ArrayList.ArrayList()`, consider the following scenarios:
    - ☐ Size of the newly constructed ArrayList
      - ☐ Not null
      - ☐ Empty

- ☐ When testing `ArrayList.add(int, E)`, consider the following scenarios:
  - ☐ Valid addition
    - ☐ Add an element to the front of the ArrayList
    - ☐ Add an element to the middle of the ArrayList
    - ☐ Add an element to the back of the ArrayList
    - ☐ Add elements that exceed the initial capacity of the ArrayList
  - ☐ Invalid addition, causing an exception
    - ☐ Null
    - ☐ Duplicates
    - ☐ Add an element to an index that is out of bounds
      - ☐ Less than min
      - ☐ Greater than max
- ☐ When testing `ArrayList.remove(int)`, consider the following scenarios:
  - ☐ Valid removal
    - ☐ Remove an element from the front of the ArrayList
    - ☐ Remove an element from the middle of the ArrayList
    - ☐ Remove an element from the back of the ArrayList
    - ☐ Remove all elements from the ArrayList
  - ☐ Invalid removal, causing an exception
    - ☐ Remove an element from an empty ArrayList
    - ☐ Remove an element from an index that is out of bounds
      - ☐ Less than min
      - ☐ Greater than max
- ☐ When testing `ArrayList.set(int, E)`, consider the following scenarios:
  - ☐ Valid set
    - ☐ Set an element at the front of the ArrayList
    - ☐ Set an element at the middle of the ArrayList
    - ☐ Set an element at the back of the ArrayList
  - ☐ Invalid set, causing an exception
    - ☐ Set an element in an empty ArrayList
    - ☐ Set an element to null
    - ☐ Set a duplicate element
    - ☐ Set an element at an index that is out of bounds
      - ☐ Less than min
      - ☐ Greater than max

- ☐ When testing `ArrayList.get(int)`, consider the following scenarios:
  - ☐ Valid get
    - ☐ Size of the ArrayList
    - ☐ Element at each index of the ArrayList
  - ☐ Invalid get
    - ☐ Get an element from an index that is out of bounds
      - ☐ Less than min
      - ☐ Greater than max

C.2.3.2 Explicit Testing Strategies Designed for Lab 8

- ☐ Test the `CourseRoll` class (*Explicit Testing Strategies for Lab 8*)
  - ☐ When testing the constructor `CourseRoll.CourseRoll()`, consider the following scenarios:
    - ☐ Capacity of the newly constructed CourseRoll
    - ☐ The number of open sets of the newly constructed CourseRoll
  - ☐ When testing `CourseRoll.setEnrolllmentCap()`, consider the following scenarios:
    - ☐ Without student enrollment
      - ☐ Valid set
        - ☐ Min enrollment
        - ☐ Between min and max enrollment
        - ☐ Max enrollment
      - ☐ Invalid set, causing an exception
        - ☐ Less than min enrollment
        - ☐ Greater than max enrollment
    - ☐ With student enrollment
      - ☐ Valid set
        - ☐ Greater than or equal to the number of currently enrolled students
      - ☐ Invalid set, causing an exception
        - ☐ Less than the number of currently enrolled students
  - ☐ When testing `CourseRoll.enroll(Student s)`, consider the following scenarios:
    - ☐ Valid enrollment
      - ☐ Enroll at least two for better practice
    - ☐ Invalid enrollment, causing an exception
      - ☐ Null

        - ☐ Duplicates
        - ☐ Enrollment that exceeds the capacity
  - ☐ When testing `CourseRoll.drop(Student s)`, consider the following scenarios:
    - ☐ Valid drop
      - ☐ Drop a non-existing student
      - ☐ Drop at least two for better practice
    - ☐ Invalid drop, causing an exception
      - ☐ Null
  - ☐ When testing `CourseRoll.canEnroll(Student s)`, consider the following scenarios:
    - ☐ Capacity
    - ☐ Duplicate student
- ☐ Test the `LinkedAbstractList` class (*Explicit Testing Strategies for Lab 8*)
  - ☐ When testing the constructor `LinkedAbstractList.LinkedAbstractList()`, consider the following scenarios:
    - ☐ Size of the newly constructed LinkedAbstractList
      - ☐ Not null
      - ☐ Empty
  - ☐ When testing `LinkedAbstractList.add(int, E)`, consider the following scenarios:
    - ☐ Valid addition
      - ☐ Add an element to the front of the LinkedAbstractList
      - ☐ Add an element to the middle of the LinkedAbstractList
      - ☐ Add an element to the back of the LinkedAbstractList
    - ☐ Invalid addition, causing an exception
      - ☐ Null
      - ☐ Duplicates
      - ☐ Add elements that exceed the initial capacity of the LinkedAbstractList
      - ☐ Add an element to an index that is out of bounds
        - ☐ Less than min
        - ☐ Greater than max
  - ☐ When testing `LinkedAbstractList.remove(int)`, consider the following scenarios:
    - ☐ Valid removal
      - ☐ Remove an element from the front of the LinkedAbstractList
      - ☐ Remove an element from the middle of the LinkedAbstractList
      - ☐ Remove an element from the back of the LinkedAbstractList
      - ☐ Remove all elements from the LinkedAbstractList

            - ☐ Invalid removal, causing an exception
                - ☐ Remove an element from an empty LinkedAbstractList
                - ☐ Remove an element from an index that is out of bounds
                    - ☐ Less than min
                    - ☐ Greater than max
        - ☐ When testing `LinkedAbstractList.set(int, E)`, consider the following scenarios:
            - ☐ Valid set
                - ☐ Set an element at the front of the LinkedAbstractList
                - ☐ Set an element at the middle of the LinkedAbstractList
                - ☐ Set an element at the back of the LinkedAbstractList
            - ☐ Invalid set, causing an exception
                - ☐ Set an element in an empty LinkedAbstractList
                - ☐ Set an element to null
                - ☐ Set a null element
                - ☐ Set a duplicate element
                - ☐ Set an element at an index that is out of bounds
                    - ☐ Less than min
                    - ☐ Greater than max
        - ☐ When testing `LinkedAbstractList.get(int)`, consider the following scenarios:
            - ☐ Valid get
                - ☐ Size of the LinkedAbstractList
                - ☐ Element at each index of the LinkedAbstractList
            - ☐ Invalid get
                - ☐ Get an element from an index that is out of bounds
                - ☐ Less than min
                - ☐ Greater than max
    - ☐ Test the updated `Schedule` class (*Explicit Testing Strategies for Lab 8*)
        - ☐ When testing `Schedule.getScheduleCredits()`, consider the following scenarios:
            - ☐ Add a course to the Schedule
            - ☐ Remove a course from the Schedule
        - ☐ When testing `Schedule.canAdd()`, consider the following scenarios:
            - ☐ Add a course to the Schedule
    - ☐ Test the updated `Student` class (*Explicit Testing Strategies for Lab 8*)
        - ☐ When testing `Student.canAdd()`, consider the following scenarios:
            - ☐ Add a course to the Schedule

C.2.3.3 Explicit Testing Strategies Designed for Lab 9

- ☐ Test the `LinkedAbstractList` class (*Explicit Testing Strategies for Lab 9*)
  - ☐ When testing `setCapacity(int)`, consider the following scenarios:
    - ☐ Valid set
      - ☐ Greater than size of LinkedAbstractList
    - ☐ Invalid set, causing an exception
      - ☐ Negative value
      - ☐ Less than the current size of LinkedAbstractList
- ☐ Test the `ArrayQueue` and the `LinkedQueue` classes (*Explicit Testing Strategies for Lab 9*)
  - ☐ When testing the constructors `ArrayQueue` and `LinkedQueue`, consider the following scenarios:
    - ☐ Capacity
  - ☐ When testing `enqueue(E)`, consider the following scenarios:
    - ☐ Valid addition
      - ☐ Add an element to the back of the Queue
      - ☐ For better practice
        - ☐ Add at least two elements
        - ☐ Check the size of the Queue after addition
        - ☐ Interleave additions and removals
    - ☐ Invalid addition, causing an exception
      - ☐ Add elements that exceeds the initial capacity
  - ☐ When testing `dequeue()`, consider the following scenarios:
    - ☐ Valid removal
      - ☐ Remove an element at the front of the Queue
      - ☐ For better practice
        - ☐ Remove all elements from the Queue
        - ☐ Check the size of the Queue after removal
        - ☐ Interleave additions and removals
    - ☐ Invalid removal, causing an exception
      - ☐ Remove from an empty Queue
  - ☐ When testing `isEmpty()`, consider the following scenarios:
    - ☐ The Queue is empty
    - ☐ The Queue is not empty

- ☐ When testing `size()`, consider the following scenarios:
  - ☐ The Queue is empty
  - ☐ The Queue is not empty
- ☐ When testing `setCapacity(int)`, consider the following scenarios:
  - ☐ Valid capacity
    - ☐ Greater than or equal to the current size of the Queue
  - ☐ Invalid capacity, causing an exception
    - ☐ Negative value
    - ☐ Less than the current size of the Queue

☐ Test the `ArrayStack` and the `LinkedStack` classes (*Explicit Testing Strategies for Lab 9*)

- ☐ When testing the constructors `ArrayStack` and `LinkedStack`, consider the following scenarios:
  - ☐ Capacity
- ☐ When testing `push(E)`, consider the following scenarios:
  - ☐ Valid addition
    - ☐ Add an element to the top of the Stack
    - ☐ For better practice
      - ☐ Add at least two elements
      - ☐ Check the size of the Stack after addition
      - ☐ Interleave additions and removals
  - ☐ Invalid addition, causing an exception
    - ☐ Add elements that exceeds the initial capacity
- ☐ When testing `pop()`, consider the following scenarios:
  - ☐ Valid removal
    - ☐ Remove an element at the top of the Stack
    - ☐ For better practice
      - ☐ Remove all elements from the Stack
      - ☐ Check the size of the Stack after removal
      - ☐ Interleave additions and removals
  - ☐ Invalid removal, causing an exception
    - ☐ Remove from an empty Stack
- ☐ When testing `isEmpty()`, consider the following scenarios:
  - ☐ The Stack is empty
  - ☐ The Stack is not empty

- ☐ When testing `size()`, consider the following scenarios:
  - ☐ The Stack is empty
  - ☐ The Stack is not empty
- ☐ When testing `setCapacity(int)`, consider the following scenarios:
  - ☐ Valid capacity
    - ☐ Greater than or equal to the current size of the Stack
  - ☐ Invalid capacity, causing an exception
    - ☐ Negative value
    - ☐ Less than the current size of the Stack

C.2.3.4 Explicit Testing Strategies Designed for Lab 10

- ☐ Test the `Faculty` class (*Explicit Testing Strategies for Lab 10*)
  - ☐ When testing the constructor `Faculty.Faculty()`, consider the following scenarios:
    - ☐ Valid number of parameters
      - ☐ For each parameter, consider the following scenarios:
        - ☐ Valid
        - ☐ Invalid, causing an exception
          - ☐ For first name, last name, id
            - ☐ Null
            - ☐ Empty string
          - ☐ For email address
            - ☐ Null
            - ☐ Empty string
            - ☐ Does not contain an '@' character
            - ☐ Does not contain a '.' character
            - ☐ The index of the last '.' character in the email string is smaller than the index of the first '@' character
          - ☐ For password
            - ☐ Null
            - ☐ Empty string
          - ☐ For max course
            - ☐ Less than 1 (min)
            - ☐ Greater than 3 (max)
    - ☐ Invalid number of parameters, causing an exception

- ☐ When testing `Faculty.setMaxCourses(int)`, consider the following scenarios:
  - ☐ Valid value
    - ☐ Min
    - ☐ Between min and max
    - ☐ Max
  - ☐ Invalid value, causing an exception
    - ☐ Less than min
    - ☐ Greater than max
- ☐ To test `Faculty.hashCode()`:
  - ☐ Create several Faculty, and compare their hash code with
    - ☐ Faculty with the same information
    - ☐ Faculty with different information
- ☐ To test `Faculty.equals(Object)`:
  - ☐ Create several Faculty, and compare them with
    - ☐ Faculty with the same information
    - ☐ Faculty with different information
- ☐ To test `Faculty.toString()`:
  - ☐ Check each value

☐ Test the `FacultyDirectory` class (*Explicit Testing Strategies for Lab 10*)

- ☐ When testing the constructor `FacultyDirectory.FacultyDirectory()`, consider the following scenarios:
  - ☐ Empty
- ☐ To test `FacultyDirectory.newFacultyDirectory()`:
  - ☐ Check if an non-empty FacultyDirectory becomes empty after the method call
- ☐ When testing `FacultyDirectory.loadFacultyFromFile(String)`, consider the following scenarios:
  - ☐ Valid load
    - ☐ File contains valid Faculty information
    - ☐ File contains invalid Faculty information
  - ☐ Invalid load, causing an exception
    - ☐ File does not exist
- ☐ When testing `FacultyDirectory.addFaculty(String, String, String, String, String, String, int)`:
  - ☐ For each parameter, consider the following scenarios:
    - ☐ Valid

- ☐ Invalid, causing an exception
  - ☐ For first name, last name, id
    - ☐ Null
    - ☐ Empty string
  - ☐ For email address
    - ☐ Null
    - ☐ Empty string
    - ☐ Does not contain an '@' character
    - ☐ Does not contain a '.' character
    - ☐ The index of the last '.' character in the email string is smaller than the index of the first '@' character
  - ☐ For password
    - ☐ Null
    - ☐ Empty string
    - ☐ Two entries do not match
  - ☐ For max course
    - ☐ Less than 1 (min)
    - ☐ Greater than 3 (max)

☐ When testing `FacultyDirectory.removeFaculty(String)`, consider the following scenarios:

- ☐ Valid removal
  - ☐ Remove one Faculty, check the remaining elements in the list
- ☐ Invalid removal
  - ☐ Remove from empty FacultyDirectory
  - ☐ Remove of non existing Faculty

☐ When testing `FacultyDirectory.getFacultyById(String)`, consider the following scenarios:

- ☐ Faculty that exists in the directory
- ☐ Faculty that does not exists in the directory

☐ To test `FacultyDirectory.saveFacultyDirectory(String)`:

- ☐ Create an non-empty FacultyDirectory linked list, save the FacultyDirectory to a file (actual output), and then compare the actual output against the expected output
  - ☐ Hint: throw exception if unable to write to the file
  - ☐ Hint: create a helper method to compare two files

- ☐ Test the `FacultyRecordIO` class (*Explicit Testing Strategies for Lab 10*)
  - ☐ When testing `FacultyRecordIO.readFacultyRecords(String)`, consider the following scenarios:
    - ☐ Valid read
      - ☐ Check the Faculty record at each row
    - ☐ Invalid read, causing an exception
      - ☐ Invalid file path
      - ☐ Invalid Faculty record
  - ☐ To test `FacultyRecordIO.writeFacultyRecords(String, LinkedList<Faculty>)`:
    - ☐ Create an non-empty Faculty linked list, write the records to a file (actual output), and then compare the actual output against the expected output
      - ☐ Hint: throw exception if unable to write to the file
      - ☐ Hint: create a helper method to compare two files
- ☐ Test the `RegistrationManager` class (*Explicit Testing Strategies for Lab 10*)
  - ☐ To test `RegistrationManager.getFacultyDirectory()`:
    - ☐ Check if the directory starts empty
    - ☐ Load Faculty information from file and then check the length and content of the list
    - ☐ Add/remove a Faculty and then check the length and content of the list
  - ☐ When testing `RegistrationManager.login()`, consider the following scenarios:
    - ☐ Log in with correct ID and password, and then log out and check the current users
    - ☐ Invalid login
      - ☐ Non existing id, causing an exception
      - ☐ incorrect password

C.2.3.5 Explicit Testing Strategies Designed for Lab 11

- ☐ Add additional tests for the `LinkedListRecursive` class (*Explicit Testing Strategies for Lab 11)*
  - ☐ When testing `LinkedListRecursive.add(E)`, consider the following scenarios:
    - ☐ Valid addition
      - ☐ Add an element to the empty list
      - ☐ For better practice
        - ☐ Add at least two elements
        - ☐ Check the size of the list after addition

- ☐ Invalid addition, causing an exception
    - ☐ Duplicates
- ☐ When testing `LinkedListRecursive.remove(E)`, consider the following scenarios:
  - ☐ Valid removal
    - ☐ Remove an element from the front of the list
    - ☐ Remove an element from the middle of the list
    - ☐ Remove an element from the back of the list
    - ☐ Remove all elements from the list
    - ☐ Check the size and the elements in the remaining list for better practice
  - ☐ Invalid removal
    - ☐ Remove an element from an empty list
    - ☐ Remove a null element

- ☐ Test the `RegistrationManager` class (*Explicit Testing Strategies for Lab 11)*
  - ☐ To test `RegistrationManager.addFacultyToCourse(Course, Faculty)`:
    - ☐ When the current user is not null and is the registrar:
      - ☐ Get a valid course from the catalog and a valid faculty from the directory by ID, and then
        - ☐ Check if the method call returns true
        - ☐ Check if the faculty has this course in the schedule
        - ☐ Check if the course has the assigned faculty as the instructor
  - ☐ To test `RegistrationManager.removeFacultyFromCourse(Course, Faculty)` and `RegistrationManager.resetFacultySchedule(Faculty)`:
    - ☐ When the current user is not null and is the registrar:
      - ☐ Get a valid course from the catalog and a valid faculty from the directory by ID, add this faculty to this course, and then
        - ☐ Check if the faculty has this course in the schedule
        - ☐ Check if the course has the assigned faculty as the instructor
        - ☐ Call the method, and then
          - ☐ Check if the faculty has zero course in the schedule
          - ☐ Check if the course has null instructor
          - ☐ Add the faculty to multiple courses for better practice