How Students Unit Test: Perceptions, Practices, and Pitfalls

Gina R. Bai
North Carolina State University
Raleigh, NC, USA
rbai2@ncsu.edu

Justin Smith
Lafayette College
Easton, PA, USA
smithjus@lafayette.edu

Kathryn T. Stolee
North Carolina State University
Raleigh, NC, USA
ktstolee@ncsu.edu

ABSTRACT
Unit testing is reported as one of the skills that graduating students lack, yet it is an essential skill for professional software developers. 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 black-box test design and one involving white-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.

Our results regarding student perceptions show that students believe code coverage is the most important outcome for test suites. For testing practices, most students were ineffective in finding known defects. This may be due to the task design and/or challenges with understanding the source code. For testing pitfalls, we identified six test smells from student-written test code; the most common were ignoring setups in the test code and testing happy path only. These results suggest the students needed more introduction to these common testing concepts and practices in advance of the study activity. Through this experience, we have identified testing concepts that require emphasis for more effective future studies on testing behavior among students.

CCS CONCEPTS
• Applied computing → Education; • Software and its engineering → Software verification and validation.

KEYWORDS
perceptions; practice; test smell; unit testing; testing education

1 INTRODUCTION
Unit testing is widely practiced in industry [23, 43], and the ACM suggests that software testing should be integrated into Computer Science and Software Engineering curricula [1]. The topic of software testing in education is gaining momentum, with the first Software Testing Education Workshop being held in 2020, which suggests that testing education is an important and timely topic worthy of study.

As with any topic in computer science, there are many approaches for teaching software testing. In an effort to understand students’ misconceptions and challenges—so that we can address those in our educational practices—we conducted a baseline empirical study with 54 undergraduate and graduate students from two universities. In this study, students answered survey questions and participated in a two-hour lab activity. Their testing tasks covered both black-box and white-box testing projects, each with accompanying specifications (adapted from prior work [18]). During the study, students were instructed to test as thoroughly as possible.

Based on our analysis of students’ survey responses and the 794 unit tests written over the course of our study, we reveal students’ perceptions of testing, overall performance on testing projects, and summarize the challenges they encountered. Our findings include:

(1) Students do not have a clear consensus on what makes a unit test good (Section 4.1).
(2) More Java experience and prior education in unit testing correspond with higher code coverage (Section 5.1).
(3) 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 (Section 5.2).
(4) Students will likely to ignore setups and only test the happy path when creating unit tests (Section 5.3).

We summarize our contributions as follows:
• We provide an analysis of students’ unit testing practices.
• We discuss challenges students encounter during unit testing and share insights as to how we can better help students overcome these challenges.

2 RELATED WORK
To our knowledge, this is the first work reporting students’ unit testing practices and perceptions outside the classroom.

2.1 Software Testing in Education
Educators have been exploring different approaches to introduce testing in Computer Science curricula, such as requiring students to turn in tests along with their solutions [16, 22], asking students to...
perform black-box testing in a software seeded with errors [28, 38], and instructing students to conduct peer testing [21, 45].

Closest to our work, Aniche and colleagues [4] survey 84 first-year CS students about challenges learning software testing, and explore the mistakes made in 230 students' labwork. They report eight categories of common mistakes: test coverage, maintainability of test code, understanding testing concepts, boundary testing, state-based testing, assertions, mock objects, and tools. We likewise use a testing survey and labwork for assessing student perceptions.

Prior work explores the quality of student-written test code [4, 9, 13, 14] and students’ perspectives on unit testing [4, 21]. The metrics used in these studies include the number of bugs detected by student-written test cases as well as branch coverage. Kazerouni et al. [30] introduced novel metrics, the balance and sequence of testing effort, to assess the incremental testing practices of the software development projects. Carver and Kraft [9] evaluated the testing ability of senior-level CS students and found that students lack the ability to use test-coverage tools effectively. These studies were conducted in the classroom environment and the activities were graded. In addition, these prior works only asked students to perform white-box testing.

We take a more comprehensive approach to measure individual test and test suite quality via requirement coverage, code coverage, and test code smells. Our study is conducted outside the classroom setting and uses black-box and white-box activities. We study students with a range of experience by recruiting graduate and undergraduate students from two different Universities.

2.2 Developer Surveys about Testing

Beyond surveying students about testing challenges [4], researchers have surveyed professional developers about their testing practices [10, 11, 38, 46] and the level of their related education and training [10, 38]. In this work, we adapt survey questions from prior work [11] 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

2.3 Test Code Quality and Smells


As test smells make test code less maintainable [7, 39, 47], how to identify and get rid of test smells is actively discussed among practitioners and researchers [35, 44, 49]. Various studies have been done on smell prevention [29, 53], smell detection [37, 41, 42, 50], and smell correction [8, 24], though none have focused on student-written test code. In this study, we adopt the definitions and classifications from prior work [20, 35, 49] and report smells in student-written test code. We also qualitatively categorize the mistakes observed in student-written test code.

³Note that the prior work surveying students on their testing challenges [4] was done concurrently with our own, so our survey questions are adapted from those used to study industry developers.

3 STUDY

Our goal is to explore how students perceive unit testing, identify the challenges students encountered when practicing unit testing, and recognize the smells that appear in student-written test code. We explore the following research questions:

Perceptions:

RQ1: What standards do students perceive make unit tests good?
We adapt survey questions from prior work [11], to explore students’ perceptions of test quality.

RQ2: What aspects of unit testing do students perceive to be challenging?
We analyze survey responses about the challenges of creating, editing, and maintaining unit tests.

Practices & Pitfalls:

RQ3: How well do students perform unit testing?
We quantitatively analyze student-written unit tests, measuring requirements coverage, code coverage, and mutation

RQ4: What challenges do students encounter when creating or editing tests?
We investigate students’ responses to the open-ended survey question, “What challenge(s) did you encounter when creating/editing the test cases?” in post surveys.

RQ5: Does student-written test code smell good?
We adopt the definitions and classifications of test smells from prior studies [20, 35, 49] and qualitatively classify test smells in student-written tests.

3.1 Procedure

Students were given two surveys (one preliminary testing perceptions survey and one post-activity survey) via Qualtrics and two testing projects to perform in the Eclipse IDE. 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 (Figure 1).

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 [6] containing links to both surveys, a black-box testing project, and a white-box testing project. Students were instructed to complete the tasks in sequence, as shown in Figure 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.
3.2 Surveys
We adapted the survey from Daka, et al. [11] for the Test Perceptions Survey (preliminary survey) [6]. 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.

At the end of the study, students completed a Post-activity Survey [6] 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.

3.3 Testing Projects
There were two testing projects. The black-box testing task focuses on designing a test suite based on specifications. The white-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.” Study materials are available at [6].

3.3.1 Test Design Project - Mars Rover API. We adopt this task from a prior study [18], as it is also used in other testing-related studies (e.g. [19, 48]). 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). 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.

3.3.2 Implementation Project - Bowling Score Keeper. We adopt this task from a prior study [18]; it is also used in multiple other testing-related studies (e.g. [19, 48]). 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.

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

3.5 Analysis
3.5.1 Survey Responses. For rating questions, we convert 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 treat them as interval-scaled data [25], and report the average numeric value of each option (e.g., Table 1). We use 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 (e.g., Table 2).

3.5.2 Metrics for Test Design Project. Since students are not expected to implement test code in the Test Design Project, we evaluate students’ performance by manually measuring the requirement coverage with the project specifications. An example test is provided that covers the requirement that, “the rover moves one position on the grid towards the direction it is facing given command f”, introducing a baseline requirements coverage of 10%.

3.5.3 Metrics for Implementation Project. For the Implementation Project, we first manually measure the requirement coverage with the project specifications. An example test is provided that covers the requirement that, “the score of a frame is the sum of two throws and it ranges from 0 to 10”, introducing a baseline requirements coverage of 7.7%.

Next, we analyze the quality of student-written test code. We adopt EclEmma\(^\text{*}\) to measure the completeness via instruction coverage (baseline = 20.3%) and branch coverage (baseline = 13.6%). We measure 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\(^\text{4}\).

We identified potential smells in student-written unit tests via test smell detector tsDect [41] and then manually validate the smells using their definitions [20, 35, 44, 49] as well as classifications from prior work [20]. No inter-rater reliability was considered in this process as only one author coded the smells, and the process required little personal interpretation [34].

3.6 Study 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. Five students, including one undergraduate student and four graduate students, have working experience in industry. 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

\(^{\text{3}}\)https://www.eclemma.org/jacoco/trunk/doc/counters.html
\(^{\text{4}}\)http://pitest.org/
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.

In Section 4, 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-written test code generated by all 54 students in Section 5.

4 RESULTS - PERCEPTIONS

We report the general agreement on standards of good unit tests from the 37 students (17 undergraduates, 20 graduates) who have experience with creating/editing/maintaining unit tests (Section 4.1), and the aspects of unit testing they found challenging (Section 4.2).

4.1 RQ1: What standards do students perceive make unit tests good?

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Overall Rating</th>
<th>UGrad Rating</th>
<th>Grad Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>How easily faults can be localised... or debugged if the test fails</td>
<td>3.0</td>
<td>3.0</td>
<td>3.1</td>
</tr>
<tr>
<td>Code coverage</td>
<td>3.0</td>
<td>2.9</td>
<td>3.0</td>
</tr>
<tr>
<td>Robustness against code changes</td>
<td>3.0</td>
<td>2.6</td>
<td>3.3</td>
</tr>
<tr>
<td>Sensitivity against code changes</td>
<td>2.9</td>
<td>2.7</td>
<td>3.1</td>
</tr>
<tr>
<td>How realistic the test scenario is</td>
<td>2.9</td>
<td>2.8</td>
<td>3.0</td>
</tr>
<tr>
<td>How easily the test can be updated... when the underlying code changes</td>
<td>2.9</td>
<td>2.6</td>
<td>3.1</td>
</tr>
<tr>
<td>Execution speed</td>
<td>2.4</td>
<td>1.8</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Table 1: “How important are the following aspects for you when you write new unit tests?” (5-point Likert Scale (0 - “Not at All Important”, 2 - “Moderately Important”, 4 - “Extremely Important”), Preliminary Survey, 37 respondents)

4.2 RQ2: What aspects of unit testing do students perceive to be challenging?

Summary: Students find it challenging to comprehend source code and handle flaky tests when fixing failing tests.

We investigated two aspects of difficulty: 1) challenges that arise when writing new unit tests, and 2) challenges that arise when fixing failing tests.

Table 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.

As for students’ perceptions toward failing tests, they responded that it was complicated to fix a failing test when the code under test is difficult to understand, or the test fails non-deterministically.

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 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 reported that writing (19/37) and maintaining tests is difficult (21/37).

Our takeaway for teaching is to present students with scenarios in which each of these aspects is challenging in its own right. As an example, for testing private classes, isolating the unit under test can be particularly challenging, yet students find it to be the least challenging aspect of the process. For hard-to-reach code, finding input values can be particular challenging as well. Introducing students to these situations can help show how the testing process can be challenging in a variety of ways.
Table 3: “Please indicate your level of agreement with the following statements” (7-point Likert Scale (0 - “Strongly Disagree”, 3 - “Neither Agree Nor Disagree”, 6 - “Strongly Agree”), Preliminary Survey, 37 respondents)

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

5 RESULTS - PRACTICES & PITFALLS

In this section, we assess students’ performance in testing with data from all 54 students (Section 5.1). We discuss the challenges that all 54 students encountered during the testing process (Section 5.2) and the smells in student-written test code (Section 5.3).

5.1 RQ3: How well do students perform unit testing?

Summary: Experience and prior education in testing correspond with better testing practices. Most students use print statements.

In this survey, when asked, “How do you test your own code”, 96% (52/54) of students claimed to use print statements (including 100% of the undergraduate students), 80% (43/54) of students claimed to use unit tests, 39% (21/54) claimed to use testing tools when testing their own code. The popularity of print-statement testing echoes prior research showing it is the most popular testing strategies for students [2, 36].

In the Implementation Project, students did not perform particularly well when it comes to discovery of the seeded faults. Students were informed that there was at least one bug in the source code and the smells in student-written test code (Section 5.3).

Table 4: Coverage achieved by students in the Design Project and the Implementation Project

<table>
<thead>
<tr>
<th>Coverage</th>
<th>Overall (%)</th>
<th>UGrad (%)</th>
<th>Grad (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imp Requirement Branch Mutation</td>
<td>44.9 55.1 62.8</td>
<td>61.6 66.7 69.6</td>
<td>81.6 81.1 81.7</td>
</tr>
<tr>
<td>Des Requirement Instruction Branch Mutation</td>
<td>49.9 61.1 49.9</td>
<td>61.1 66.7 41.9</td>
<td>61.6 60.3 29.5</td>
</tr>
<tr>
<td>Grad (%)</td>
<td>61.1 66.7</td>
<td>33.9 29.5</td>
<td></td>
</tr>
</tbody>
</table>

5.2 RQ4: What challenges do students encounter when creating/editing the test cases?

Summary: Students reported that the most challenging aspects of testing were “understanding source code” (17/54) and “determining when to stop testing” (14/54). Although only seven students reported having difficulty understanding program specifications, over 85% of students generated at least one test case that mismatched the program specifications.

Table 5: “What challenge(s) did students encounter when creating/editing the test cases?” (Post Survey, 54 respondents)

<table>
<thead>
<tr>
<th>Students encountered these challenges...</th>
<th>nPar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code comprehension</td>
<td>17 (31.5%)</td>
</tr>
<tr>
<td>Determine “Testing Enough”</td>
<td>14 (25.9%)</td>
</tr>
<tr>
<td>Specification comprehension</td>
<td>7 (13.0%)</td>
</tr>
<tr>
<td>Mastery of Java</td>
<td>6 (11.1%)</td>
</tr>
<tr>
<td>Mastery of JUnit</td>
<td>5 (9.3%)</td>
</tr>
<tr>
<td>Unable to create test case for a scenario</td>
<td>4 (7.4%)</td>
</tr>
<tr>
<td>Creation of test cases</td>
<td>3 (5.6%)</td>
</tr>
<tr>
<td>Familiarity of Eclipse IDE</td>
<td>2 (3.7%)</td>
</tr>
</tbody>
</table>

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). Although only seven students reported having difficulty understanding program specifications, over 85% of students generated at least one test case that mismatched the program specifications. Additionally, the implementation barrier stands out as important; teaching students how to design tests to achieve various objectives (e.g., coverage) is important to build confidence in testing techniques. However, it should not replace the task of actually writing the test code. The quality of the tests should be adjudged against the original specification of the program, possibly by running the tests against a reference implementation [16].

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 students (14.8%) created tests cases with incorrect syntax and were unable to fix the errors. Among 54 students, 46 (85.2%) generated at least one test case whose testing scenario did not match the program specifications. Similar student behaviors that misunderstanding the problem leads to flawed implementations were also observed in prior work [52].
As a takeaway for teaching, we plan to 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 [27]. 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.

5.3 RQ5: Does student-written test code smell good?

Summary: We observed six smells among student-written test code. The most frequently occurring smells were Refused Bequest (68.5%) and Happy Path Only (44.4%).

We ran the student-written tests against the test smell detector tsDetect[41]. We identified three code-related smells: Test Redundancy [31, 53] (13 students, 32 tests), Bad Naming [12, 44] (20 students, 48 tests), and Lack Comments [33] (10 students, 43 tests); one test semantic/logic related smells: Happy Path Only [20] (24 students); and two smells in test steps: Refused Bequest [7, 44, 49] (37 students, 96 tests), No Assertions [44] (5 students, 6 tests).

We found that students were most likely to introduce smells related to test steps: 37/54 (68.5%) students ignored the setups provided in the test class in at least one test case (Refused Bequest test code smell). Furthermore, over 40% of students tested the happy path only. This matches the observation in prior work [15] that students tend to write basic test cases to cover expected behaviors rather than detect hidden bugs.

As a takeaway for teaching, students might not fully understand how each component in test class functions, such as the usage of the Before annotation; more instruction is needed in general and especially before a study such as this. As for the happy-path testing, this represents a common misconception about the testing process, where the purpose is to verify expected behavior in addition to uncover faults.

6 DISCUSSION

We observed that students found it challenging to understand the source code and program specifications. This could be potentially introduced by the study design, as both of the testing projects we used were originally designed for a study involving professionals. In this section, we briefly compare the students in our study to professionals according to survey responses and discuss threats to validity.

6.1 Students vs. Professionals

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 [11]—this aspect ranked last among graduate students. Moreover, while professionals consider isolating the unit under test somewhat challenging [11], 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.

6.2 Threats to Validity

Students were self-selected into this study. All graduate students were from same university, same with the undergraduate students, which made the sample relatively homogeneous. A replication with a more diverse and larger set of students is needed.

We only observed students’ interactions with two unfamiliar codebases. Students may perform differently on their own or peers’ programs with which they are more familiar.

The testing projects were relatively small, so test smells observed may not be representative of smells in a larger codebase. Given the time constraints of the study and the goal of testing as thoroughly as possible, 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.

7 CONCLUSION

Our goal was to better understand how students perceive and perform unit testing. We adopted study designs from prior work with industrial participants and found our population of students often struggled to understand the source code and specifications. Some of this may have been due to the study design (e.g., short time limits) or due to the artifacts themselves. Nevertheless, our results show that students did not have a clear consensus on what makes a unit test good, but they believed that the ease of bug localization and code coverage are important. We found that the major challenges that students had during testing was to understand the source code and program specifications. These challenges frequently led to mismatches between testing intention and program specifications, which also suggests a potential barrier where students had a hard time to implement unit tests that correctly and precisely reflect the testing intentions. We also identified six test smells from student-written test code; Refused Bequest and testing Happy Path Only are the most common smells. Overall, students who had more experience and prior education on unit testing consistently outperformed those who were less experienced in both designing and implementing unit tests.

These results pinpoint the testing concepts and the real-world testing scenarios that should to be presented and discussed in classes. These concepts may better prepare the students for confidence and success in testing practices, and for participation in studies that can dive a little deeper into the specific misconceptions that arise during software testing activities.

ACKNOWLEDGMENTS

This work is supported in part by NSF-SHF-#1749936.

REFERENCES
