Check It Off: Exploring the Impact of a Checklist Intervention on the Quality of Student-authored Unit Tests

Gina R. Bai, Kai Presler-Marshall, Thomas W. Price, Kathryn T. Stolee
Department of Computer Science
North Carolina State University, Raleigh, NC, USA
{rbai2,kpresle,twprice,ktstolee}@ncsu.edu

ABSTRACT

Software testing is an essential skill for computer science students. Prior work reports that students desire support in determining what code to test and which scenarios should be tested. In response to this, we present a lightweight testing checklist that contains both tutorial information and testing strategies to guide students in what 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).

By analyzing the combination of student-written tests and survey responses, we found students with the checklist performed as well as or better than the coverage tool group, suggesting a potential positive impact of the checklist (or at minimum, a non-negative impact). This is particularly noteworthy given the control condition of the coverage tool is the state of the practice. These findings suggest that the testing tool support does not need to be sophisticated to be effective.

CCS CONCEPTS

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

KEYWORDS

unit testing, testing education, checklist

ACM Reference Format:
Gina R. Bai, Kai Presler-Marshall, Thomas W. Price, Kathryn T. Stolee. 2022. Check It Off: Exploring the Impact of a Checklist Intervention on the Quality of Student-authored Unit Tests. In *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol 1 (ITiCSE 2022), July 8–13, 2022, Dublin, Ireland*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3502718.3524799

ITiCSE 2022, July 8–13, 2022, Dublin, Ireland

ACM ISBN 978-1-4503-9201-3/22/07…$15.00
https://doi.org/10.1145/3502718.3524799

1 INTRODUCTION

Software testing is widely practiced in industry [25, 41] 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 in recent years [44], including integrating testing in CS1 courses [23, 27, 47], and introducing tools to support students learning testing [10, 15, 47].

However, students often run into trouble when they are learning testing. For example, students make mistakes such as missing boundary testing [5, 9], writing smelly tests [8, 13], testing happy paths only [8, 9], and not testing the program until they are already done with development [40]. Additionally, prior work [8] surveyed 54 students from two universities about their testing perceptions and practices. The students reported they *wanted more tool support for unit test composition.* The two most difficult aspects of unit testing were reported to be "*Determining what to check*" and "*Identifying which code/scenarios to test*".

Checklists have been identified as a useful tool in software engineering for code review [14, 43] or encouraging software security [3]. Prior work [36] 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 [31, 32], we conjectured that a checklist containing testing tips could alleviate some students' testing struggles. By integrating tutorial information into the checklist, knowledge gaps can be filled for students that need it (as not all students require the same level of tool support [30]). We implemented a testing checklist that:

- contains both tutorial information and testing strategies to assist students in determining what to test and how to test,
- addresses common mistakes and smells that have been observed in recent testing education research studies,
- is static and non-interactive, which provides testing support without introducing steep learning curves, and
- is designed to be lightweight enough to transfer across classrooms, instructors, and universities.

The application of checklists to software testing is novel. To assess the potential costs and benefits of this testing checklist for students, we conduct an experimental, controlled, A/B laboratory study with 32 students (both undergraduate and graduate) who were enrolled in software engineering courses. Students answered survey questions and participated in a two-hour lab activity. Stratified by level (undergraduate vs. graduate), students were randomly assigned to the experimental and control groups. The experimental group received the testing checklist and the control group received a

tutorial on a coverage tool, EclEmma (with which all students were already familiar from their coursework). In this way, the control group represents the state-of-the-practice. Students were instructed to test an existing program as thoroughly as possible according to the provided program specifications. With a combination of student-written tests and survey responses, we explore the following two research questions:

RQ1: Do students who see the checklist write better test code than those who do not?

RQ2: How do students engage with the checklist?

Our results show no statistically significant differences in the quality of tests written by students, regardless of group. This suggests that our first attempt at forming a useful lightweight checklist does as well as the state-of-the-practice, and that testing tool support does not need to be sophisticated to be effective. Therefore, a checklist may be an effective supplement to coverage tools.

2 RELATED WORK

We focus on related work in testing education and the adoption of checklists in software engineering research and education.

2.1 Testing Education

Software testing enables students to catch bugs early in development [51]. Studies also report that when students write their own tests, they write better code [35, 45]. Wick et al. [51] suggest that unit testing helps group projects by demonstrating the correctness of the code to group members.

To encourage students to write more thorough tests, educators have experimented with diverse approaches to teaching testing, such as requiring students to turn in tests along with their solutions [20, 23, 27], asking students to perform closed-box testing on a program seeded with errors [29], and instructing students to conduct peer testing [22]. To increase interest and motivation in software testing among students, educators also introduced cross-course activities [6, 42], in which students in software testing courses wrote tests for peers who are taking other CS courses.

Closest to this work is prior work that focuses on testing education by providing inquiry-based conceptual information during testing [15]. By contrast, we focus on the testing process. In our checklist, we suggest the use of various testing concepts and provide tutorial information related to the use of the JUnit library. That is, since our course curriculum explicitly exposes students to testing concepts (e.g., boundary value testing), we focus on reminding students to use techniques they have already seen.

2.2 Measurements of Test Quality in Education

Test code quality is usually evaluated by completeness using code coverage metrics (e.g., [4, 38]), effectiveness using mutation coverage (e.g., [25, 50]) and maintainability using test code smells (e.g., [5, 48]). 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 [24].

In education, the quality of student-written test code is usually measured by statement coverage and branch coverage [18, 27]. However, prior studies on industry code have shown that code coverage has no correlation with test suite effectiveness [28], and it can be gamed by writing poor tests that execute code but lack effective assertions [27]. To measure test suite effectiveness, some educators [18, 46] 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. Assessing maintainability has been done through test smells [8, 13], which may cause inaccurate test results as programs evolve and impact the maintainability of unit tests [48]. However, this usually requires manual inspection, and hence is not commonly adopted in testing education at scale.

2.3 Checklists in Software Engineering

Prior work has demonstrated that checklists are an effective tool for mastering various skills or performing various tasks in software engineering, such as code review [14, 43], software inspection [11, 16, 37], and ensuring software security [3]. For example, McMeekin et al. [37] 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. Some students even create their own ad-hoc checklists during testing [40].

Software engineering educators have also adopted checklists to assist teaching and student learning [17, 39, 43]. Rong et al. [43] found a checklist served as a necessary and helpful guideline for inexperienced students to conduct code review. While they encouraged students to construct and maintain their own checklists, they reminded students that the checklist solely did not guarantee a high-quality code review. By contrast, Chong and colleagues [14] investigated 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.

3 THE TESTING CHECKLIST

We designed our checklist to address the challenges that students face during testing, based on prior work [8, 9, 40, 42, 44]. 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 testing 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 1) maps to problems students encounter during software testing (*Addressing issues of...*). 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 [15], 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 [12, 33, 40, 44], 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:

Table 1: The 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 [8]	☐ be executable (i.e., it has an `@Test` annotation and can be run via "Run as JUnit Test")
2. Having no assertions [5] & Testing happy paths only [19]	☐ 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 [5]	☐ evaluate/test only one method
	Each test case *could*:
4. Bad naming tests [5]	☐ be descriptively named and commented
5. Unfamiliarity with test class features [5, 8]	☐ If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using `@Before`)
6. Assertion Roulette [13]	☐ 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 [8]	☐ have at least one test for each requirement
8. Ignoring setup [8]	☐ appropriately use the setup and teardown code (e.g., `@Before`, which runs before each `@Test`)
9. Misinterpretation of failing tests [42]	☐ 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 [5, 8, 9]	☐ 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 [8, 9]	☐ measure code coverage using an appropriate tool, such as EclEmma (installation, tutorial). Inspect uncovered code and write tests as appropriate.

Issue 1: Tests have syntax errors, or lack assertions [5, 8].
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* [5, 8], *No Assertions* [5, 8], or covering the *Happy Path Only* [5, 8, 9].
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 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 [9].
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 [5, 9].
Solution 4: Items #10a-d remind 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 [42]. Prior work [27] 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).

4 STUDY

To assess the value of the checklist, we conducted a controlled experiment with both undergraduate and graduate software engineering students. Study materials are available on GitHub [7].

4.1 Procedure

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.

We used an experimental, controlled A/B testing study design. 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 (Section 3), or to a control group (the Coverage group), in which students receive the task description along with a tutorial on EclEmma [1], a widely-used Eclipse plug-in, which measures Java code coverage, that students had used in previous course assignments.

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.

4.2 Tasks

4.2.1 Surveys. We adapt the preliminary and the post-activity surveys from prior work [8]. 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

Table 2: Students' self-reported experience with Java, unit testing, and prior experience with unit tests

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

brief demographics information (e.g., prior experience with Java and testing).

4.2.2 Unit Testing Project. We adapted the TDD project - Bowling Score Keeper used in multiple prior testing-related studies (e.g., [8, 21, 52]). 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 a program implemented by others [6, 42], they were *not* expected to fix the bugs or modify the source code.

4.3 Participants

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

Table 2 presents participants' self-estimated experience in years with Java (*avg_yrJava*, numeric) and unit testing (*avg_yrUT*, numeric), as well as if they have prior experience with unit tests (*with_expUT*, binary yes/no response). 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 (all except for two graduate students in the Checklist group) 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.

4.4 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-written test code, we adopted the same metrics as prior work [8], as described in Sections 4.4.1 and 4.4.2. The group condition was concealed during data analysis.

4.4.1 Survey Responses: For rating questions, we convert 5-point Likert scale to numbers and treat them as interval-scaled data [26], where 1 maps to the lowest score, "Not at all helpful", and 5 maps to the highest score, "Extremely helpful".

4.4.2 Unit Testing Project: We measure the test code quality from three aspects:

Table 3: ANOVA results for comparing the use of the testing checklist and the coverage tools

	Independent Variable		
Dependent 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

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

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 rerunning 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 [2] for mutation testing, and we manually identify the number of seeded bugs revealed by student-written 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 [8] and manually identify the smells in student-written test code. No inter-rater reliability was considered in this process as only one author coded the smells.

4.4.3 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.

4.4.4 Statistical Analysis. Treating the six metrics from Section 4.4 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 [34]. We report the p-values in Table 3.

5 RESULTS

We compare the testing performance of students in the experimental (Checklist) group and the control (Coverage) group (Section 5.1) and report how students engage with the checklist (Section 5.2).

Table 4: Measurements of student-written 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

5.1 RQ1: Test Quality

Summary: Students with the checklist and code coverage tool support performed equally well, suggesting that the tool support does not need to be sophisticated to be effective.

Table 4 shows the quality metrics of the student-written 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 [49], and our ANOVA analysis confirms this (Table 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 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-written tests from the Checklist and the Coverage groups.

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

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

5.2 RQ2: Engagement with Checklist

Summary: Most students self-reported that they read the checklist before they wrote any unit tests (13/15). On average, they also found it "very helpful" (3.9 on a 5-point Likert scale).

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 testing 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%, instruction coverage of 91.8%, and branch coverage of 63.6%. The tests written by this student revealed no bugs along with a mutation coverage of 60.0%.

While the checklist seemed to benefit Students One and Two, Student Three may have performed better with more guidance on the testing techniques [15]. This echoes the finding that not all students need the same level of tool support [30].

6 DISCUSSION

Based on our experiment, we find that our first attempt at a 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.

As this was the first investigation into the value of a testing checklist with a small sample size, it warrants further refinement and investigation into how and why it works. Here, we discuss the potential positive impact of adopting the testing checklist on student-written test code, as well as potential threats to validity.

6.1 Positive Impact

While analyzing the student-written test code, we found that the students in the Checklist group achieved higher mutation coverage despite writing fewer assert statements (median number of assert statements were 7.0 vs. 14.0, in Table 4). Though the differences were not statistically significant ($p = 0.21$), we dug into the implementation details and found a potential explanation: even though the achieved requirements coverage was relatively the same for both groups, there were differences in the individual requirements that were being tested. 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 boundary values (checklist item #10b) than simply covering the requirements or lines of code.

Additionally, 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.

6.2 Future Work

For future work, a replication study with a more diverse and larger set of computing students is suggested to further investigate and better generalize the impact of the checklist.

Students in our study were largely unsuccessful at discovering the seeded faults (median 0.0 for both groups). 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.

Additionally, there are many potential improvements that can be made to the checklist. We plan to use the feedback from students as part of RQ2 to further refine the checklist items so to maximize their value. For example, students found it difficult to navigate the code, program requirements, and checklist, which were all in separate places. A future checklist can integrate the requirements directly into the checklist to reduce context switching. Another potential improvement is to make the checklist interactive by allowing students to check off items and visualize their progress. Integrating coverage tools, such as EclEmma and PITest, into the checklist is also suggested as it can reduce context switching.

6.3 Threats to Validity

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

The lab sessions were conducted synchronously and remotely. To alleviate the concern about potential academic misconduct, and to ensure students followed study procedures, 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. However, since they know they are being observed, this may impact how they test.

Construct: 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: Sampling bias could be introduced in participant recruitment as students were self-selected into this study. The students in the study were relatively advanced students studying software engineering. These results may not generalize to other students or those with less testing experience.

7 CONCLUSION

The application of checklists to software testing is novel and preliminary evidence suggests it is promising for improving the quality of student-written tests. In this study, we implemented and assessed a lightweight testing checklist. Our results showed no statistically significant differences between the Checklist group and Coverage group on any of the metrics studied, suggesting a lightweight checklist can help students as well as specific coverage tools. We observed that most students used the checklist as a guide–they read through the checklist before they wrote any unit tests, and they occasionally consulted the checklist during unit testing. The checklist appeared to help students write better tests, as their tests achieved relatively the same code coverage and higher mutation coverage than those who only used the coverage tool. We also conjecture that students who have lower prior knowledge in Java and unit testing showed benefit more from the checklist.

8 ACKNOWLEDGEMENTS

This work is supported in part by NSF SHF #1749936 and #1714699.

REFERENCES

[1] [n.d.]. EclEmma: Coverage Counters. https://www.eclemma.org/jacoco/trunk/doc/counters.html. Accessed: 2022-01-23.

[2] [n.d.]. PITest: Mutation Operators. http://pitest.org/quickstart/mutators/. Accessed: 2022-01-23.

[3] Mahtab Alam. 2010. Software security requirements checklist. *International Journal of Software Engineering, IJSE* 3, 1 (2010), 53–62.

[4] Tiago L. Alves and Joost Visser. 2009. Static Estimation of Test Coverage. In *International Working Conf. on Source Code Analysis and Manipulation*. 55–64.

[5] Maurício Aniche, Felienne Hermans, and Arie van Deursen. 2019. Pragmatic Software Testing Education. In *ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. 414–420.

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

[7] Gina R. Bai. 2022. *ginaBai/TestingChecklistStudy: TestingChecklistStudyMaterials*. https://doi.org/10.5281/zenodo.6466776

[8] Gina R. Bai, Justin Smith, and Kathryn T. Stolee. 2021. How Students Unit Test: Perceptions, Practices, and Pitfalls. In *ITiCSE 2021: 26th ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 248–254.

[9] Lex Bijlsma, Niels Doorn, Harrie Passier, Harold Pootjes, and Sylvia Stuurman. 2021. How do Students Test Software Units?. In *Int'l Conf. on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 189–198.

[10] Michael K. Bradshaw. 2015. Ante Up: A Framework to Strengthen Student-Based Testing of Assignments. In *Technical Symposium on CS Education*. 488–493.

[11] Bill Brykczynski. 1999. A Survey of Software Inspection Checklists. *SIGSOFT Softw. Eng. Notes* 24, 1 (Jan. 1999), 82.

[12] Ingrid A. Buckley and Winston S. Buckley. 2017. Teaching Software Testing using Data Structures. *International Journal of Advanced Computer Science and Applications* 8, 4 (2017).

[13] Kevin Buffardi and Juan Aguirre-Ayala. 2021. Unit Test Smells and Accuracy of Software Engineering Student Test Suites. In *Conference on Innovation and Technology in Computer Science Education*. 7.

[14] Chun Yong Chong, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2021. 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)*. 20–29.

[15] Lucas Cordova, Jeffrey Carver, Noah Gershmel, and Gursimran Walia. 2021. *A Comparison of Inquiry-Based Conceptual Feedback vs. Traditional Detailed Feedback Mechanisms in Software Testing Education: An Empirical Investigation*. 87–93.

[16] Bruno Pedraça de Souza, Rebeca Campos Motta, and Guilherme Horta Travassos. 2019. The First Version of SCENARIotCHECK: A Checklist for IoT Based Scenarios. In *Brazilian Symposium on Software Engineering* (Salvador, Brazil) *(SBES 2019)*. Association for Computing Machinery, 219–223.

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

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

[19] Stephen H. Edwards and Zalia Shams. 2014. Do Student Programmers All Tend to Write the Same Software Tests?. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE '14)*. 171–176.

[20] Stephen H. Edwards, Zalia Shams, Michael Cogswell, and Robert C. Senkbeil. 2012. Running Students' Software Tests Against Each Others' Code: New Life for an Old "Gimmick". In *Technical Symposium on CS Education*. 221–226.

[21] Davide Fucci and Burak Turhan. 2013. A Replicated Experiment on the Effectiveness of Test-First Development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 103–112.

[22] Alessio Gaspar, Sarah Langevin, Naomi Boyer, and Ralph Tindell. 2013. 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*. 109–114.

[23] Michael H. Goldwasser. 2002. A Gimmick to Integrate Software Testing Throughout the Curriculum. In *Technical Symposium on CS Education*. 271–275.

[24] Orlena CZ Gotel and CW Finkelstein. 1994. An analysis of the requirements traceability problem. In *Int'l Conf. on Requirements Engineering*. 94–101.

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

[26] Spencer E. Harpe. 2015. How to analyze Likert and other rating scale data. *Currents in Pharmacy Teaching and Learning* 7, 6 (2015), 836–850.

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

[28] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *36th Intl. Conf. on Software Eng. (ICSE 2014)*. 435–445.

[29] Ursula Jackson, Bill Z. Manaris, and Renée A. McCauley. 1997. 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)*. 360–364.

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

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

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

[33] Edward L. Jones. 2001. Integrating Testing into the Curriculum — Arsenic in Small Doses. *SIGCSE Bull.* 33, 1 (Feb. 2001), 337–341.

[34] William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47, 260 (1952), 583–621.

[35] Otávio Augusto Lazzarini Lemos, Fábio Fagundes Silveira, Fabiano Cutigi Ferrari, and Alessandro Garcia. 2018. The impact of Software Testing education on code reliability: An empirical assessment. *J. of Sys. and Software* 137 (2018), 497–511.

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

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

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

[39] Kai Petersen and Jefferson Seide Molléri. 2021. Preliminary Evaluation of a Survey Checklist in the Context of Evidence-based Software Engineering Education. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, Raian Ali, Hermann Kaindl, and Leszek A. Maciaszek (Eds.). 437–444.

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

[41] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding Myths and Realities of Test-suite Evolution. In *Foundations of Software Engineering (FSE '12)*. Article 33, 11 pages.

[42] Upsorn Praphamontripong, Mark Floryan, and Ryan Ritzo. 2020. 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)*. 445–451.

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

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

[45] Lilian P. Scatalon, Jorge M. Prates, Draylson M. de Souza, Ellen F. Barbosa, and Rogério E. Garcia. 2017. Towards the Role of Test Design in Programming Assignments. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*. 170–179.

[46] Zalia Shams and Stephen H. Edwards. 2013. Toward Practical Mutation Analysis for Evaluating the Quality of Student-Written Software Tests. In *ACM Conference on International Computing Education Research (ICER '13)*. 53–58.

[47] Jaime Spacco and William Pugh. 2006. Helping Students Appreciate Test-driven Development (TDD). In *Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. 907–913.

[48] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. 2020. Investigating Severity Thresholds for Test Smells. In *Mining Software Repositories (MSR '20)*. 311–321.

[49] Matthew S. Thiese, Brenden Ronna, and Ulrike Ott. 2016. P value interpretations and considerations. *Journal of Thoracic Disease* 8, 9 (2016).

[50] Jeffrey Voas. 1997. How assertions can increase test effectiveness. *IEEE Software* 14, 2 (Mar 1997), 118–119.

[51] Wick, Michael and Stevenson, Daniel and Wagner, Paul. 2005. Using Testing and JUnit Across the Curriculum. In *36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)*. 236–240.

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