Evaluating the Effectiveness of a Testing Checklist Intervention in CS2: An Quasi-experimental Replication Study

Gina R. Bai
Vanderbilt University
Nashville, TN, USA
rui.bai@vanderbilt.edu

Zuoxuan Jiang
Vanderbilt University
Nashville, TN, USA
allison.z.jiang@vanderbilt.edu

Thomas W. Price
North Carolina State University
Raleigh, NC, USA
twprice@ncsu.edu

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

ABSTRACT

Students often run into trouble when learning and practicing software testing. Recent prior studies demonstrate that a lightweight testing checklist that contains testing strategies and tutorial information could assist students in writing higher-quality tests. Prior studies also suggest that students with lower prior knowledge in unit testing may benefit more from the checklists. However, insights on the potential benefits and costs of the testing checklists in a classroom setting are lacking. To address this, we conducted an operational replication study in a CS2 course with 342 students (171 from Fall 2023 and 171 from Spring 2024) who had no prior experience in unit testing.

In this paper, we report our experience in introducing the testing checklists as optional tool support in a CS2 course. To evaluate the effectiveness of the testing checklists in a classroom setting, we quantitatively analyze a combination of programming assignment submissions and survey responses generated by students. Our results suggest that students who received the testing checklists achieved significantly higher quality in their test code, in terms of code coverage and mutation coverage, compared to those who did not. We also observed that the exposure to the testing checklists in students' early learning process encouraged students to write more unit tests to cover possible testing scenarios.

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, Zuoxuan Jiang, Thomas W. Price, and Kathryn T. Stolee. 2024. Evaluating the Effectiveness of a Testing Checklist Intervention in CS2: An Quasi-experimental Replication Study. In *ACM Conference on International Computing Education Research V.1 (ICER '24 Vol. 1), August 13–15, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3632620.3671102

ICER '24 Vol. 1, August 13–15, 2024, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-0475-8/24/08
https://doi.org/10.1145/3632620.3671102

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 [29, 47], and hence an emerging topic in computer science education. Educators and researchers find that the practices of software testing would enable students to produce cleaner code "faster than ever before" by catching bugs early in development [58]. Studies also report that when students write their own tests, they write better source code [37, 52]. Wick et al. [58] suggest that unit testing is particularly helpful in students' group projects as it could demonstrate the correctness of the code to group members.

However, testing is also a challenging skill for students to learn. Due to the inevitable limited duration and scale of course projects, it is hard for students to realize the importance of testing [12, 46, 48, 51]. Educators report that many students view testing as debugging [12, 48], believe the only necessary skill to perform testing is knowing how to print debugging information [48], and not test their programs until they are already done with development [46]. Students also exhibit low motivation and negative attitudes towards testing due to the increased cognitive load [17, 34, 46, 51], as it requires students to learn new concepts, new syntax, new tools and new libraries. When practicing unit testing, students often make mistakes such as missing boundary testing [7, 12], writing smelly tests [10, 18], and creating tests that mismatched the program specifications [10]. Students also find it challenging to understand the source code, determine when to stop testing, determine what code to test, and which scenarios should be tested [9, 10].

Even if students are taught to avoid these pitfalls and apply the many best practices of software testing, they may still forget to apply this knowledge when writing tests (i.e., they slip [22]). To address this, Bai et al. [9, 11] proposed offering students a *testing checklist* with best practices for writing tests. This study asked students (n = 32) to write a set of test cases with the support from either a testing checklist or a code coverage tool (i.e., EclEmma) and compared their coverage, mutation testing scores, and number of

Table 1: Summary on Study Designs of the Original Study [9] and this Study

	The Original Study [9]	**This Study**
Duration	Two-hour lab session	One-week course assignment
Task	One Java-based unit testing project	One Java-based unit testing project
Survey	Required pre- and post-surveys	Optional post-survey (response rate: 142/171, 83.0%)
# Participants	32 students from NCSU (23 undergrads, 9 grads)	342 undergrad students from Vanderbilt University
Prior Experience	30 yes vs. 2 no, avg of 2.3 years	18 yes vs. 124 no, avg of 0.2 years (Section 3.5.1)
Control Grp	17 students (12 undergrads, 5 grads) Received a tutorial on a standard coverage tool	171 undergrad students in Fall 2023 Received a lecture on unit testing, and demo on coverage tool
Experimental Grp	15 students (11 undergrads, 4 grads) Received a testing checklist	171 students in Spring 2024 Received a lecture on unit testing, and demo on coverage tool Received a testing checklist (Table 3)
Compensation	Extra credit for course	None, it is a required course assessment

identified bugs. They found few differences between experimental (received checklist) and control groups (received tutorial on the code coverage tool), suggesting that a lightweight testing checklist can be as effective as a tool and might be particularly beneficial to students who have lower prior knowledge in unit testing. However, this study had a number of limitations as well: it was a laboratory study and not an authentic classroom study; it lasted only 2 hours, it had a small sample ($n = 32$), and the control condition did not have access to the information contained in the checklist. This suggests the need for further research on the topic.

To better understand the potential benefits and costs of the testing checklists in a classroom setting, in this paper, we present a operational replication [30] of the Bai et al. checklist study [9]. Our replication study makes a variety of changes designed to improve the strength and generalizability of the results of the original (see Table 1 for details), including:

(1) An authentic classroom environment (as opposed to a lab study with students of varying prior experience)
(2) A longer activity (seven days versus one two-hour lab session)
(3) A much larger population (10x larger)
(4) A stronger control condition (all students received a lecture with the learning content of the checklist)

We explore the following two research questions:

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

RQ2: Does having the checklist intervention early in the semester improve students' later testing performance?

Our results confirm the initial findings of Bai et al. [9, 11] that the testing checklists could significantly improve the quality of student-authored tests, and testing tool support does not need to be sophisticated to be effective. Our findings also suggest that the checklists may have had lasting impacts on students' testing behaviors.

2 RELATED WORK

2.1 Educational Interventions in Testing

Software testing is critical but challenging in software development, and it is an essential skill for computing students. The ACM suggests that software testing should be integrated into Computer Science and Software Engineering curricula [3]. Studies have been conducted to experiment and evaluate various approaches to teaching testing. For example, gamifying software testing practices [13, 26], providing automated evaluation of student-authored tests [14, 55], guiding students to conduct peer testing [25, 27], requiring students to turn in tests before [14] or along [25, 28, 33] with their source code.

2.2 Checklists in Education

Bai et al. [9] designed a lightweight checklist containing testing strategies and tutorial information. To evaluate the effectiveness of the testing checklist, they conducted a controlled study in a lab setting with 32 graduate and undergraduate students. The study compared the quality of student-authored test cases supported by the testing checklists against those using the IDE's built-in coverage tool. They suggested that the testing checklist could "assist students in writing high-quality tests without a steep learning curve." In this paper, we evaluate this testing checklist [9] in a classroom setting, specifically a Data Structures (CS2) course. More broadly, checklists have been perceived as a handy and effective tool for assisting teaching and student learning [24, 45, 49], and they are particularly helpful for inexperienced students. Marwan et al. [41] have also reported that novice programming students achieved higher grades on programming assignments when presented with a checklist of subgoals that automatically tracked their progress. Pham and colleagues [46] have found some undergraduate students create their own ad-hoc checklists during testing.

The use of checklists to support students can be motivated from a number of theoretical perspectives. From the perspective of scaffolding theory [57, 59], the checklist may serve as *scaffolding*, allowing learners to "solve a task or achieve a goal that would be beyond [their] unassisted efforts" [59], similar to the support a human tutor might provide. Writing robust tests with high coverage may be beyond what some students can accomplish after a single lesson on the topic (as in our study), and so such scaffolding can play a key role in helping students succeed at this goal. Vygotsky argued that these skills, which are just beyond the students' capability to accomplish unassisted, but within their ability to accomplish *with help*, lie in the student's Zone of Proximal Development (ZPD) [20], and represent good opportunities for learning. Importantly, different students may have different skill levels with respect to a subject

like writing tests, and therefore a task that may require scaffolding for some students, may not require scaffolding for others, and may in fact be too difficult for others even with scaffolding.

In the medical domain, Sibbald et al. [54] ground the use of checklists in cognitive load theory [35, 56], which emphasizes that people have limited working memory to process task-relevant information (e.g., software testing strategies) while working on a task. Sibbald et al. argue that checklists work by allowing users to offload some of this information onto paper, freeing up working memory without losing relevant information. While experts can use learned domain-specific strategies and schema to reduce the *cognitive load* placed on working memory, novices have not yet developed these skills, and therefore benefit more from checklists than experts [53, 54]. In computing education, cognitive load theory has been used to improve instructional design, by reducing the *extraneous cognitive load* imposed by the design of the assignment, freeing up students' cognitive resources for learning (e.g., through worked examples [19, 39, 60]). Unfortunately, because the study we present in this paper took place in an authentic classroom setting, with an out-of-class project (see Section 3.3), we were unable to measure student's cognitive load during their programming task. Therefore, this study cannot directly assess whether cognitive load is a mechanism by which checklists support student outcomes.

Finally, checklists may support students in a similar way to *subgoal labels*, which outline the high-level steps to solving a problem, which may generalize across problems. From a theoretical perspective, these labels help students learn important problem structure and organize information, and they promote self-explanation of the solution [38]. Morrison, Margulieux and colleagues demonstrated that worked examples were particularly effective when combined with such subgoal labels [39, 43], and subgoals labels can also be beneficial for for *code writing* tasks [40, 41]. Similarly, checklists, such as the one presented in this study (Table 3), offer students high-level steps that are transferable across problems.

2.3 Replication Studies in CS Education

Replication studies are an important way to build stronger bodies of evidence for scientific hypotheses, understand the impacts of interventions on diverse populations, and identify results that *fail* to replicate in new contexts. Educators [16, 42] also argue for the importance of replication studies in computing education research (*CER*), encouraging both a cultural shift [4] among the CER community to place greater value on replication studies, as well as specific practices to encourage replication. Despite the calls in the past decade to increase the number of replication studies [16, 31, 42]; Hao et al. [31] systematically investigated replications in CER, and found that between 2009 and 2018 the proportion of CER papers that presented replication studies was about 2.38% (54/2269).

In this paper, we present a quasi-experimental operational replication study of Bai et al. [9] in order to examine the effectiveness of a lightweight testing checklist intervention in testing education. The similarities and differences of these two study are discussed in Section 3.

3 METHOD: OPERATIONAL REPLICATION IN DATA STRUCTURES (CS2) COURSE

Table 1 summarizes the similarities and differences of the context and collected data between the previous study [9] and this study. While there are a number of improvements over the original study (see Section 1), we note two new limitations to our study:

- **Surveys**: While the original study [9] had required pre- and post-surveys, our study had only an optional post-survey. In the original study, student participants had an average of 2.3 years of prior experience in unit testing. Therefore, the researchers collected students for their perceptions of unit testing with the preliminary surveys, and gathered students' feedback on the checklists with the post-surveys. However, in this study, the majority of students had no or very limited prior experience in unit testing, and hence we only deployed the post-survey to gain insights on students' engagement and feedback on the checklists.
- **Quasi-experimental design**: We used two semesters of students and compared data across the two semesters, rather than using a randomized controlled trial. This was necessary to carry out the experiment in an authentic classroom setting, as it would have been neither fair nor practical to have students in the same classroom experience different assignment conditions. We discuss reasons we believe the two semesters are quite similar and comparable in Section 3.4.

While not a limitation, another important difference in this study is that *both* the Control and Checklist Groups were instructed on how to use a built-in code coverage tool, while in the original study, the Checklist Group did *not* receive any instruction on how to use this tool. Since use of the code coverage tool was lower for the Checklist Group in the original study [9], it made conclusions about the usefulness of the checklist *compared* to a coverage tool. By contrast, we directly study the *additional* impact of the checklist on top of a coverage tool.

This study received IRB approval from Vanderbilt University.

3.1 Overview of the CS2 Course

We ran this study with students who were taking a 3-credit CS2 lecture course at Vanderbilt University, a research-intensive university in the United States. This course is a Java-based course and is intended to further the understanding of beginning computer science students in the analysis, design, implementation, testing, and debugging of programs, with an emphasis on the utilization of abstract data types and data structures to solve problems. This course is required for all Computer Science majors and minors, and open to non-majors, with 160-180 students per semester.

The primary components of the course consist of eleven weekly programming projects and four exams. All projects are individual assignments, requiring students to implement the source code (except Project #0, details in Section 3.3) and test their own implementation using JetBrains IntelliJ IDEA. The class meets three times a week for 50 minutes, and does not include a lab section.

Table 2: Student Information.

	Freshman	Sophomore	Junior	Senior	Took CS1 (avg course grade)	Bypass CS1 via AP CSA
Fall 2023 (total: 171)	45	104	16	6	101 (90.1%)	63
Spring 2024 (total: 171)	118	36	15	2	143 (89.5%)	28

3.2 Participants

Table 2 shows that the majority of students who enrolled in this CS2 course were first and second year students (Fall 2023: 149/171, 87.1%, Spring 2024: 154/171, 90.1%). At Vanderbilt University, in order to take the CS2 course, student can either pass a CS1 course at the university, or bypass CS1 and directly enroll CS2 if they receive full credit (i.e., 5/5) on Advanced Placement Computer Science A (AP CSA) exam in secondary school. Among 171 students who enrolled in this CS2 course in Fall 2023, 108 of them took CS1 (average course grade: 90.1%) at Vanderbilt University, and 63 of them came in with AP credit. In Spring 2024, 143 of 171 CS2 students took CS1 at the university and achieved an average of 89.5% for course grade.

3.3 Task: Project #0 on Unit Testing

As prior studies [9, 11] suggest that students may benefit more from the checklists in their early learning process, we integrated the checklist into Project #0, which is the first programming project of the semester and is the only project that focuses solely on unit testing throughout the semester.

In *both semesters*, Project #0 was distributed to students after Lecture #4, in which students learned the concept of unit testing, basic testing strategies (e.g., equivalence class partitioning and boundary value analysis), the syntax of JUnit 5 (e.g., test class components and various assertions), how to use the built-in Coverage tool in IntelliJ IDEA, and how to interpret its coverage report. All testing strategies and tutorial information included in the testing checklists are covered and elaborated in Lecture #4. Lecture content was the same in both semesters.

For Project #0, students were expected to perform unit testing on a provided Java class (StringJr.java) that includes 13 methods simulating the behavior of String methods. Students were explicitly instructed to test every single method, and the primary objective is to test the class "as thoroughly as possible". Students were also encouraged to achieve 100% of line coverage and branch coverage. Students had a week to complete the project, and they were required to submit their test code, StringJrTest.java, for grading. The instructor's solution consists of 42 unit tests.

A previous study [11] that explores students adoption of testing checklists in a classroom setting pointed out that when the code, program requirements, and checklist were dispersed across different locations, students exhibited reduced willingness to adopt the checklist due to challenges in navigating between these sources. To address this issue, we have integrated the checklists (Table 3) directly into the starter code as multi-line comments. This approach aims to reduce context switching for students while working on the assignment, suggested in prior work [11]. Students were encouraged to place an "X" in front of the checklist items to mark them as completed.

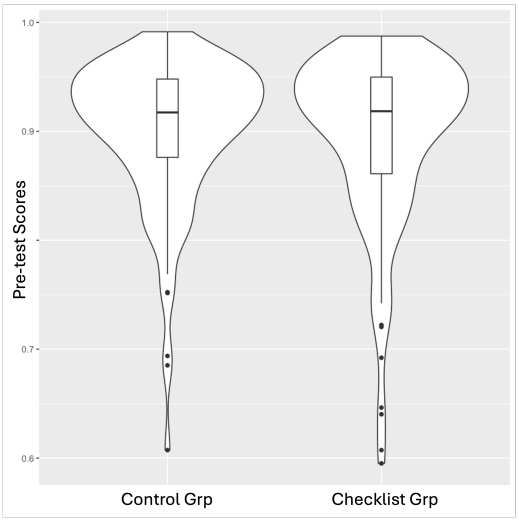


Figure 1: Violin plots showing the distribution of final course grade in CS1 for students in Control Group (101/171 in Fall 2023) vs. Checklist Group (143/171 in Spring 2024). Note that the y-axis cuts off at 0.6 (60%).

To gain a better understanding of students' support needs and engagement with the checklists, we invite students in the Checklist Group (Spring 2024) to reflect on their usage and evaluation of the checklist in an optional post-survey during assignment submission.

3.4 Comparability of Experimental Groups

Because our study is quasi-experimental, we first endeavored to ensure that the Fall 2023 (*Control Group*) and Spring 2024 (experimental, *Checklist Group*) groups were as comparable as possible. Both semesters offered four sections and were taught by the same two faculty (one teaching one section and the other teaching three sections). The course syllabi, learning objectives, assessments, lecture materials, teaching staff support, amount of instructors' and teaching assistants' office hours, and the adoption of Piazza, all remained the same.

We also investigated potential differences in the populations of students in each semester. To do so, we looked at students' final performance in their prior CS1 course (since our experiment took place at the very beginning of the semester). Figure 1 revealed the distribution of students' course grades in CS1 for those that took the course. The distributions are nearly identical ($U = 7732.5$, $p = .986$, $r_{rb} = .001$). Therefore, the primary difference between students in our two conditions is the proportion who took AP CSA. While students who took the CS1 course had experience submitting

Table 3: The testing checklist that integrated into the starter code as multi-line comments. Adapted from the original study [9].

Test Case Checklist
Each test case *should*:
- ☐ be executable (i.e., it has an `@Test` annotation and can be run via "Run as JUnit Test")
- ☐ have at least one assert statement or assert an exception is thrown.
- ☐ evaluate/test only one method

Each test case *could*:
- ☐ be descriptively named and commented
- ☐ If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using `@BeforeEach`)
- ☐ 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*:
- ☐ have at least one test for each requirement
- ☐ appropriately use the setup and teardown code (e.g., `@BeforeEach`, which runs before each `@Test`)
- ☐ contain a fault-revealing test for each bug in the code (i.e., a test that fails)
- ☐ For each requirement, contain test cases for:
  - ☐ Valid inputs ☐ Boundary cases ☐ Invalid inputs ☐ Expected exceptions

To improve the test suite, you *could*:
- ☐ measure code coverage using an appropriate tool, such as "Run with Coverage" in IntelliJ. Inspect uncovered code and write tests as appropriate.

their source to an online auto-grading platform, and interpreting the auto-generated test results (which indicated pass/fail outcomes with highlighted differences between actual output and expected output), the CS1 course does not explicitly teach students testing strategies or expect them to perform unit testing. Students with a score of 5/5 on the AP CSA exam should be, if anything, more advanced and more motivated than students who did not come in with this credit. Therefore, we argue that any differences in performance on Project #0 that favor the experimental group are likely due to the intervention, rather than differences between the populations. Further, our results in Section 4.2 also provide meaningful evidence that these two populations were comparable, and effects we see are due to the checklist intervention, rather than inherent differences between semesters. However, we discuss the potential limitations to this quasi-experimental comparison at the beginning of Section 3.

3.5 Data Analysis

3.5.1 Survey Responses. In total, 142 of 171 students completed the optional post-survey via Qualtrics (Figure 2). Among these 142 students, 124 of them self-reported to have no prior experience (0 years) in unit testing, 13 students indicated they had one year of experience in unit testing, and five claimed two years of experience. On average, students had 0.16 years of prior experience in unit testing.

For Q6, we converted 5-point Likert scale to numbers and treat them as interval-scaled data [32], where 1 maps to the lowest score (i.e., "Not at all helpful"), and 5 maps to the highest score (i.e., "Extremely helpful").

3.5.2 Student-authored Test Code. We measured the test code quality with seven standard metrics:

> **1) Mutation coverage** [2, 6, 9–11, 21]
> The percentage of killed mutants with the total number of mutants. We measured this effectiveness metric via PITest with its default mutation operator.

(1) **What challenge(s) did you encounter when creating or editing the test cases?**
_____

(2) **Please name one thing you would have liked to receive help with during the study.**
_____

(3) **Did you use the checklist during the study?**
○ Yes (jump to Q4) ○ No (jump to Q7)

(4) **(Used Checklist) 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

(5) **(Used Checklist) Which items in the checklist were most useful? (Check ALL that apply)**
... Same as the testing checklist in Table 3 ...

(6) **(Used Checklist) How would you rate this checklist in terms of helpfulness?** (followed by Q8)
○ Extremely helpful ○ Very helpful
○ Moderately helpful ○ Slightly helpful
○ Not at all helpful

(7) **(Did not use Checklist) Please briefly explain why you chose not to use the checklist.** (followed by Q8)
_____

(8) **(For all) How many years of experience with unit testing do you have?**
_____

(9) **(For all) How do you describe your experience in unit testing?**
○ No experience ○ Novice ○ Advanced Beginner
○ Competent ○ Proficient ○ Expert

Figure 2: Questions in the optional post-survey. The survey was distributed via Qualtrics with a branching design.

Table 4: Seven measurements of student-authored tests quality (sorted from the most to least significant for Benjamini-Hochberg procedure) for comparing the quality of student-authored tests in Control Group (Fall 2023) vs. in Checklist Group (Spring 2024). For each measurement, the table presents its average (standard deviation), median, Mann–Whitney U test p-values (unadjusted), and Rank-Biserial correlation coefficients.

Metrics	Group	Average (SD)	Median	p-value	r_{rb}
numTests	Checklist	50.4 (33.6)	44.0	**7.56e-09** ***	0.361
	Control	33.9 (28.4)	25.0		
lineCov	Checklist	95.1% (11.0%)	98.1%	**1.985e-05** ***	0.260
	Control	92.0% (12.6%)	96.3%		
mutationCov	Checklist	90.1% (13.2%)	94.0%	**3.375e-05** ***	0.258
	Control	85.9% (14.9%)	90.0%		
numAssertions	Checklist	77.6 (64.1)	63.0	**0.0008** ***	0.211
	Control	62.4 (47.6)	47.0		
branchCov	Checklist	92.3% (13.3%)	97.5%	**0.0030** **	0.183
	Control	88.9% (15.2%)	92.5%		
numAssertThrows	Checklist	14.0 (18.3)	9.0	**0.0097** **	0.161
	Control	11.4 (15.6)	7.0		
methodCov	Checklist	98.0% (13.3%)	100.0%	0.1207	0.060
	Control	96.9% (15.2%)	100.0%		

*$p < 0.05$, ** $p < 0.01$, *** $p < 0.001$ (unadjusted); **bold** indicates significant after Benjamini-Hochberg procedure

2) Line coverage & 3) Branch coverage & 4) Method coverage [1, 5, 9–11, 23, 36, 44]
We measured these two completeness metrics via IntelliJ IDEA Code Coverage Runner.
5) The number of tests & 6) The number of assertions & 7) The number of unhappy path tests that contains `assertThrows` [8, 11]
We adopt this measurement of completeness from prior work [11], and only consider the unhappy path tests that assert an exception is thrown.

4 RESULTS

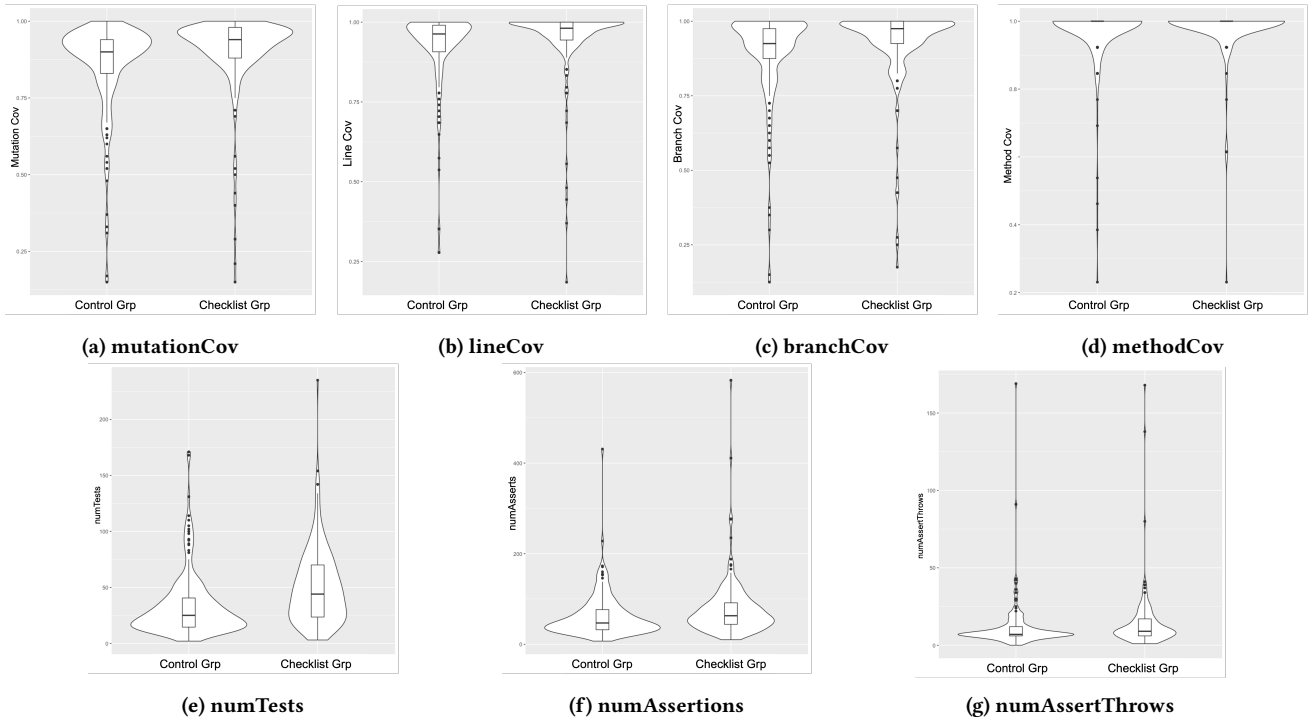
4.1 RQ1: Do students who receive the checklist write better test code than those who do not?

Table 4 and Figure 3 show the seven quality metrics of the student-authored test code (defined in Section 3.5.2), including effectiveness metrics (i.e., *mutationCov*) and completeness metrics (e.g., *lineCov*, *branchCov*, and *numAssertThrows*). For example, the table shows that students in the Checklist Group wrote on average 50.4 test cases (*numTests*) and 77.6 assertions (*numAssertions*), compared to the Control Group's average of 33.9 tests and 62.4 assertions.

Treating these seven metrics from Section 3.5.2 as dependent variables, we measured the impact of the independent variable, *isChecklistGroup* (i.e. Control/Fall vs. Checklist/Spring groups). Since the data were not normally distributed, we adopted non-parametric Mann–Whitney U tests, and we report the p-values, and the Rank-Biserial correlation coefficients (as a measure of effect size) in Table 4. The Rank-Biserial Correlation ranges from -1 to 1, with 1 indicating the strongest positive relationship between the intervention and the outcome variable. We also applied the Benjamini-Hochberg procedure to control the false discovery rate (FDR) [15] at 0.05, and we highlight the p-values that remain significant in bold.

The analysis of test code quality suggests that the testing checklists had a statistically significant impact on the effectiveness and completeness of student-authored tests. The differences between Control and Checklist Groups were significant for each measures besides *methodCov*. As students were explicitly instructed to test every single method, it is expected that the adoption of checklist would not have a large impact on method coverage of student-authored tests. Figure 3b shows the same trends visually, comparing the distribution of students' metrics in the Control (left) and Checklist (right) Groups. Our findings differ somewhat from those of the original study [9], which did not find much difference in students' coverage between the Checklist and Control Groups, and which did not find significant differences in test effectiveness (i.e., *mutationCov*) between the groups (possibly due insufficient sample size in the original study to detect this effect). One possible reason that our results may contrast with the original study [9] is students' engagement with the checklist itself. Most students in the Checklist Group self-reported that they walked through the checklist before submission (91/142, 69.47%), read the checklist before they wrote any unit tests (83/142, 63.36%), and approximately half consulted the checklist during unit testing (59/142, 45.04%). This is in contrast to prior work [11], where only 50.7% of students adopted the testing checklist as a form of tool support. Overall, students found the checklist to be moderately helpful, rating it an average of 3.4 on a 5-point Likert scale. We discuss this contrast and potential reasons for it further in Section 5.

Our results also show that the checklist may have impacted students' approach to testing as well, in ways that affect the *style* of their tests more than their test quality. For example, we see that students in the Checklist Group wrote more tests, and more assertions, but actually had fewer assertions *per* test (77.6/50.4 = 1.54 for the Checklist Group vs 62.4/33.9 = 1.84 for the Control



(a) mutationCov **(b) lineCov** **(c) branchCov** **(d) methodCov**

(e) numTests **(f) numAssertions** **(g) numAssertThrows**

Figure 3: Violin plots showing the distribution of the measurements of student-authored tests quality. Left: Control Group, Right: Checklist Group.

Group). This may reflect the Checklist's instruction to avoid having "too many `assert` statements in a single test."

4.2 RQ2: Does having the checklist intervention early in the semester improve students' later testing performance?

To answer this question, we further analyzed the quality of students' test code from Projects #1 - #3, in which the checklists were *not* provided to students in *both* Fall 2023 and Spring 2024 semesters. This allows us to measure any longer-term *learning* impact of the checklist intervention. We treat this as an open question, as prior work does not suggest whether to expect an effect of the checklist on learning. One could imagine that as students adopted best practices of testing in Project #0, they might learn these practices; however, one could also imagine that the checklist serves only as a temporary memory aid, and has little effect afterwards. Projects #1 - #3 asked students to implement and unit test: 1) a Java class that represents a user-defined object, 2) a Java class that represents a collection of the user-defined objects. The underlying implementations of the collection were 1) fix-sized arrays in Project #1, 2) dynamic arrays in Project #2, and 3) linked lists in Project #3. While the class names varied in these two semesters, the functionality of these two classes remains the same. In both semesters, students were required to submit both their source code, which accounted for 90% of project grades, and their test code, which accounted for 10% of project grades. Additionally, students were explicitly instructed to achieve a minimum of 80% of line coverage in their source code to receive the full credit on the test code.

Table 5 represents the quality of student-authored tests in Projects #1 - #3 in Fall 2023 (Control) and Spring 2024 (Prior Checklist Exposure). We report the same statistics, including unadjusted p-values from comparative Mann-Whitney U tests, and bold those p-values that remain significant after the Benjamini-Hochberg proecdure to control the false discovery rate at 0.05 [15].

Overall, we see that the *Control* Group performs marginally better on most measures of test coverage (*lineCov*, *branchCov*, *methodCov*) and effectiveness (*mutationCov*) across all three projects, with one significant improvement on *branchCov* for Project #1. This suggests that the checklist intervention on Project #0 did *not* have a positive impact on students' testing performance later in the semester. It also serves as compelling evidence that the effects we *did* see on Project #0 (RQ1) were likely due to the impact of the checklist, rather than inherent differences between semesters and our quasi-experimental design, because we only see these differences when the checklist was present, and not later in the semester.

However, we do see some evidence that the Checklist had a continuing impact on students' testing *behavior*, if not their performance. We observed that the students who received the checklist in Project #0 (i.e., in Checklist Group) continued to write significantly more tests (*numTests* $p < 0.001$) on all three projects, and continued to include more assertions (*numAsserts* $p < 0.01$, *numAssertThrows* $p < 0.001$) on Projects #1 and #2, compared to those in Control Group. This indicates that students in Checklist Group continued

Table 5: The averages and Mann–Whitney U test p-values (unadjusted) of seven measurements for comparing the quality of student-authored tests in Projects #1 - #3 in Control Group (Fall 2023) vs. in Checklist Group (Spring 2024).

Metrics	Group	Project #1		Project #2		Project #3	
		Average	p-value	Average	p-value	Average	p-value
mutationCov	Checklist	82.0%	0.2704	82.8%	0.1903	85.2%	0.0991
	Control	84.8%		84.5%		87.7%	
lineCov	Checklist	92.1%	0.5164	93.4%	0.8781	92.8%	0.0801
	Control	94.2%		94.1%		93.8%	
branchCov	Checklist	86.8%	**0.0064** **	88.7%	0.0499 *	86.8%	0.4091
	Control	90.4%		90.5%		88.9%	
methodCov	Checklist	95.4%	0.0595	96.6%	0.1409	94.9%	0.0453
	Control	97.2%		97.4%		95.6%	
numTests	Checklist	57.7	**2.724e-08** ***	88.4	**1.394e-15** ***	78.6	**4.497e-15** ***
	Control	36.9		46.1		41.5	
numAsserts	Checklist	94.5	0.6290	162.8	**0.0034** **	146.3	**0.0015** **
	Control	97.5		136.5		123.5	
numAssertThrows	Checklist	6.5	0.1422	9.0	**7.753e-05** ***	6.8	**5.914e-05** ***
	Control	5.7		6.4		4.9	

*$p < 0.05$, **$p < 0.01$, ***$p < 0.001$ (unadjusted); **bold** indicates significant after Benjamini-Hochberg procedure

to produce more granular test results (<2 assertions per test), compared to those generated by the Control Group (>2.5 assertions per test). While we cannot definitely attribute this difference to the checklist intervention, with the quasi-experimental nature of our study, it seems likely, given that the magnitude of the differences on Projects #1 - #3 (e.g. almost 2x as many tests written on average) parallels the magnitude of differences on Project #0.

5 DISCUSSION

5.1 RQ1: Impact of the Checklist on Performance

We found strong evidence that the checklist did improve students coverage and test quality. The original study [9] presented evidence that the checklist was similarly useful to a tutorial of a code coverage tool (EclEmma), and presented some suggestive evidence (not significant) that the checklist improved students' mutation coverage (which EclEmma did not help with). Our results offer much stronger support for the checklist, and show that this benefit is *complementary* to a code coverage tool, as students in both groups had access to and instruction on using a code coverage tool. These results agree somewhat more with Bai et al.'s later study of a different, *explicit strategy* checklist intervention for testing [11], which offered suggestive evidence that, early in the semester, such a checklist may have improved test quality. Similarly, we found that there is an effect early in the semester, but our results are also much more conclusive. For educators, our study provides clear evidence supporting the use of checklists when first introducing students to testing. The checklist is a lightweight intervention, only requiring the instructor to share the resource or embed it in an assignment, making adoption straightforward.

These results are somewhat consistent with an understanding of the checklist as a form of scaffolding that helps students to accomplish a task they otherwise may not have been able to do unassisted [57]. We note that most students in the control group seemed capable of achieving relatively high line and mutation coverage *without assistance*, with median 96.3% and 90.0%, respectively (see Table 4). This suggests that writing a set of relatively effective tests was already within students capabilities. However, with the checklist, we see that students achieve many more perfect or near-perfect coverage scores (see Figure 3c). One interpretation of this result is that the checklist served as useful scaffolding specifcally for some of the harder concepts of testing (e.g. testing edge cases), which students otherwise would have struggled with. Future work could investigate what specific testing concepts the checklist helped most with.

5.2 RQ2: Impact of the Checklist on Learning

We saw no evidence that the impact of the checklist on students' ability to write high-coverage and high-mutation-coverage tests lasted beyond the original assignment. We therefore cannot conclude that the checklist lead to learning, or improved performance *without* the checklist. This may have been because of the short exposure to the checklist – a single project – and future work should investigate whether a longer exposure may create a more enduring effect. With additional exposure to the checklist, it is possible that the steps of the checklist may be able to serve as generalizable subgoals, similar to subgoal-labeled worked examples or coding problems [39, 40]. However, we did not observe evidence of this effect in this shorter study. For educators, our current evidence lends support to the view of checklists as a tool to support memory and recall [49, 50], rather than as as instructional material that enhances learning. However, we note that there is value in scaffolding students to do well on their first assignment, e.g. in improving their course grades, and potentially improving their confidence for future assignments. Instructors could consider continuing to provide the checklist in later assignments, as this requires minimal instructor effort; however, our results do no speak directly to how effective this would be.

We do see some evidence that the checklist impacted the number of tests and asserts that students wrote in later projects. One possible explanation is that the differences between the two populations in later project may simply be due to incoming differences not controlled for by our quasi-experimental design, though we see no evidence of this in our pre-measures (Section 3.4). Another possible explanation is that the checklist used in the first assignment had a lasting impact on how students construct tests. The checklist specifically encouraged students to create multiple tests for each requirement, corresponding to valid inputs, boundary cases, invalid inputs and expected exceptions. It may be that students who started off creating these finer-grained tests in the first assignment continued to do so in later projects.

5.3 Threats to Validity

The grading rubrics differ among projects. Project #0 expects students to achieve 100% coverage on *both* line coverage and branch coverage for full credit, as this Project focuses solely on unit testing. Projects #1 - #3 consist of both implementation and testing tasks, and therefore only require students to achieve 80% line coverage on every source code class (i.e., the implementation) for full credit on the test code (10% of project grades). These thresholds were applied to all submissions in both semesters, and might have led to overestimation of students' performance on completeness metrics.

The adoption of the checklists and the completion of the post-surveys on the use of checklists were optional, are subject to selection bias, and consequently conclusions drawn may not generalize. Additionally, response bias may be introduced in the student self-reported surveys, and hence impact the validity of survey responses.

6 CONCLUSION & FUTURE WORK

To assist students in writing high-quality test code without introducing a steep learning curves, a previous study [9] designed a lightweight testing checklist that contains both testing strategies and tutorial information and confirmed its effectiveness in a lab setting. To explore the potential benefits of the testing checklists in a classroom setting, we conducted a operational replication with two different semesters of students and compared students' testing performance on the same course assessment across the two semesters. Our results echoed the findings in the previous work [9, 11] that the testing checklist could assist students write significantly better tests, in terms of effectiveness and completeness, compared to those who with no tool support from the testing checklist.

Our replication experience provided additional evidence that the testing checklist intervention may be effective in various contexts. It also demonstrated that the testing checklists are lightweight enough to be adapted to other classrooms, instructors, and universities. In this study, we did not observe a lasting impact on students' ability to write high-coverage and high-mutation-coverage tests, as the testing checklist was only provided for the first assignment. Future work could scaffold students with the checklists throughout the first half of the semester and explore their testing practices and performance towards the end of the semester.

The large-scale classroom setting of this study posed a challenge for us in measuring students' actual engagement with the checklist. Some students, as instructed, checked off an item by placing an 'X' in front of it, some students checked off items by deleting them, and some students self-reported that they used the checklist but deleted the entire checklist before submission. It is unclear to us whether and how students interact with the checklist during unit testing. It is also unclear to us whether the students in the Checklist Group consulted the checklist when completing Projects #1 - #3, and hence the potential impact of testing checklists on students' learning, rather than solely focusing on testing performance when received tool support, remains understudied. Future work in classrooms could investigate students' actual engagement with the checklist as tool support, encourage students to write their own testing checklists, and explore the potential impact of the checklist on students' perceptions towards testing.

ACKNOWLEDGMENTS

This work is supported by NSF IUSE #2141923.

REFERENCES

[1] [n. d.]. EclEmma: Coverage Counters. https://www.eclemma.org/jacoco/trunk/doc/counters.html. Accessed: 2024-06-06.
[2] [n. d.]. PITest: Mutation Operators. http://pitest.org/quickstart/mutators/. Accessed: 2024-06-06.
[3] ACM, 2013. Computer Science Curricula Recommendations: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. https://www.acm.org/education/curricula-recommendations.
[4] Alireza Ahadi, Arto Hellas, Petri Ihantola, Ari Korhonen, and Andrew Petersen. 2016. Replication in computing education research: researcher attitudes and experiences. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling 2016)*. ACM.
[5] Tiago L. Alves and Joost Visser. 2009. Static Estimation of Test Coverage. In *International Working Conference on Source Code Analysis and Manipulation*. 55–64.
[6] Michael Andersson. 2017. An Experimental Evaluation of PIT's Mutation Operators. , 27 pages.
[7] Maurício Aniche, Felienne Hermans, and Arie van Deursen. 2019. Pragmatic Software Testing Education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. ACM, New York, NY, USA, 414–420.
[8] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering* 40, 11 (Nov 2014), 1100–1125.
[9] Gina R. Bai, Kai Presler-Marshall, Thomas Price, and Kathryn T. Stolee. 2022. 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)*.
[10] Gina R. Bai, Justin Smith, and Kathryn T. Stolee. 2021. How Students Unit Test: Perceptions, Practices, and Pitfalls. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. ACM, New York, NY, USA, 248–254.
[11] Gina R. Bai, Sandeep Sthapit, Sarah Heckman, Thomas W. Price, and Kathryn T. Stolee. 2023. An Experience Report on Introducing Explicit Strategies into Testing Checklists for Advanced Beginners. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (, Turku, Finland,) *(ITiCSE 2023)*. ACM, New York, NY, USA, 194–200.
[12] Lex Bijlsma, Niels Doorn, Harrie Passier, Harold Pootjes, and Sylvia Stuurman. 2021. How do Students Test Software Units?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 189–198.
[13] Raquel Blanco, Manuel Trinidad, María José Suárez-Cabal, Alejandro Calderón, Mercedes Ruiz, and Javier Tuya. 2023. Can gamification help in software testing education? Findings from an empirical study. *Journal of Systems and Software* 200 (2023), 111647.
[14] Michael K. Bradshaw. 2015. Ante Up: A Framework to Strengthen Student-Based Testing of Assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. ACM, New York, NY, USA, 488–493.
[15] David I. Broadhurst and Douglas B. Kell. 2007. Statistical strategies for avoiding false discoveries in metabolomics and related experiments. *Metabolomics* 2 (2007), 171–196. https://api.semanticscholar.org/CorpusID:7638542
[16] Neil C. C. Brown, Eva Marinus, and Aleata Hubbard Cheuoua. 2022. Launching Registered Report Replications in Computer Science Education Research. In

Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1 (ICER 2022). ACM.

[17] 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).

[18] Kevin Buffardi and Juan Aguirre-Ayala. 2021. *Unit Test Smells and Accuracy of Software Engineering Student Test Suites*. ACM, New York, NY, USA, 234–240.

[19] Michael E Caspersen and Jens Bennedsen. 2007. Instructional design of a programming course: a learning theoretic approach. In *Proceedings of the third international workshop on Computing education research*. 111–122.

[20] Seth Chaiklin. 2003. The Zone of Proximal Development in Vygotsky's Analysis of Learning and Instruction. *Vygotsky's Educational Theory in Cultural Context* (09 2003).

[21] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452.

[22] Albert T Corbett and John R Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction* 4 (1994), 253–278.

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

[24] 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.

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

[26] Gordon Fraser, Alessio Gambi, Marvin Kreis, and José Miguel Rojas. 2019. Gamifying a Software Testing Course with Code Defenders. In *Proc. of the ACM Technical Symposium on Computer Science Education (SIGCSE) (SIGCSE'19)*. ACM.

[27] 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.

[28] Michael H. Goldwasser. 2002. A Gimmick to Integrate Software Testing Throughout the Curriculum. In *Technical Symposium on Computer Science Education (SIGCSE '02)*. 271–275.

[29] 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.

[30] Omar S. Gómez, Natalia Juristo, and Sira Vegas. 2014. Understanding replication of experiments in software engineering: A classification. *Information and Software Technology* 56, 8 (2014), 1033–1048.

[31] Qiang Hao, David H. Smith IV, Naitra Iriumi, Michail Tsikerdekis, and Amy J. Ko. 2019. A Systematic Investigation of Replications in Computing Education Research. *ACM Transactions on Computing Education* 19, 4 (Aug. 2019), 1–18.

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

[33] 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.

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

[35] Liesbeth Kester, Fred Paas, and Jeroen J. G. Van Merrienboer. 2010. Instructional Control of Cognitive Load in the Design of Complex Learning Environments. *Cognitive Load Theory* (04 2010).

[36] Ken Koster. 2008. A State Coverage Tool for JUnit. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. 965–966.

[37] 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. *Journal of Systems and Software* 137 (2018), 497–511.

[38] Lauren E Margulieux and Richard Catrambone. 2016. Improving problem solving with subgoal labels in expository text and worked examples. *Learning and Instruction* 42 (2016), 58–71.

[39] Lauren E Margulieux, Briana B Morrison, and Adrienne Decker. 2020. Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples. *International Journal of STEM Education* 7 (2020), 1–16.

[40] Lauren E Margulieux, Briana B Morrison, Baker Franke, and Harivololona Ramilison. 2020. Effect of Implementing Subgoals in Code. org's Intro to Programming Unit in Computer Science Principles. *ACM Transactions on Computing Education (TOCE)* 20, 4 (2020), 1–24.

[41] Samiha Marwan, Bita Akram, Tiffany Barnes, and Thomas W Price. 2022. Adaptive immediate feedback for block-based programming: Design and evaluation. *IEEE Transactions on Learning Technologies* 15, 3 (2022), 406–420.

[42] Monica M. McGill. 2019. Discovering Empirically-Based Best Practices in Computing Education Through Replication, Reproducibility, and Meta-Analysis Studies. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research (Koli Calling '19)*. ACM.

[43] Briana B Morrison, Lauren E Margulieux, Barbara Ericson, and Mark Guzdial. 2016. Subgoals help students solve Parsons problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 42–47.

[44] 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 (ISSTA 2016)*. 130–141.

[45] Kai Petersen and Jefferson Seide Molléri. 2021. 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*, Raian Ali, Hermann Kaindl, and Leszek A. Maciaszek (Eds.). SCITEPRESS, 437–444.

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

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

[48] 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.

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

[50] Guoping Rong, Jingyi Li, Mingjuan Xie, and Tao Zheng. 2012. The effect of checklist in code review for inexperienced students: An empirical study. In *2012 IEEE 25th Conference on Software Engineering Education and Training*. IEEE, 120–124.

[51] 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 *ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. 421–427.

[52] Lilian Passos Scatalon, Jorge Marques Prates, Draylson Micael de Souza, Ellen Francine Barbosa, and Rogério Eduardo 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.

[53] Matthew Sibbald, Anique BH De Bruin, and Jeroen JG van Merrienboer. 2014. Finding and fixing mistakes: do checklists work for clinicians with different levels of experience? *Advances in Health Sciences Education* 19 (2014), 43–51.

[54] Matthew Sibbald, Anique B H de Bruin, and Jeroen J G van Merrienboer. 2013. Checklists improve experts' diagnostic decisions. *Medical Education* 47, 3 (2013), 301–308.

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

[56] John Sweller. 2011. CHAPTER TWO - Cognitive Load Theory. Psychology of Learning and Motivation, Vol. 55. Academic Press, 37–76.

[57] Janneke van de Pol, Monique Volman, and Jos Beishuizen. 2010. Scaffolding in Teacher–Student Interaction: A Decade of Research. *Educational Psychology Review* 22 (09 2010), 271–296.

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

[59] David Wood, Jerome S Bruner, and Gail Ross. 1976. The role of tutoring in problem solving. *Journal of child psychology and psychiatry* 17, 2 (1976), 89–100.

[60] Rui Zhi, Thomas W Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. 2019. Exploring the impact of worked examples in a novice programming environment. In *Proceedings of the 50th acm technical symposium on computer science education*. 98–104.