An Experience Report on Introducing Explicit Strategies into Testing Checklists for Advanced Beginners

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

Sandeep Sthapit
North Carolina State University
Raleigh, NC, USA
ssthapi@ncsu.edu

Sarah Heckman
North Carolina State University
Raleigh, NC, USA
sarah_heckman@ncsu.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

Software testing is a critical skill for computing students, but learning and practicing testing can be challenging, particularly for beginners. A recent study suggests that a lightweight testing checklist that contains testing strategies and tutorial information could assist students in writing quality tests. However, students expressed a desire for more support in knowing *how* to test the code/scenario. Moreover, the potential costs and benefits of the testing checklist are not yet examined in a classroom setting. To that end, we improved the checklist by integrating explicit testing strategies to it (*ETS Checklist*), which provide step-by-step guidance on *how* to transfer semantic information from instructions to the possible testing scenarios. In this paper, we report our experiences in designing explicit strategies in unit testing, as well as adapting the ETS Checklist as optional tool support in a CS1.5 course. With the quantitative and qualitative analysis of the survey responses and lab assignment submissions generated by students, we discuss students' engagement with the ETS Checklists. Our results suggest that students who used the checklist intervention had significantly higher quality in their student-authored test code, in terms of code coverage, compared to those who did not, especially for assignments earlier in the course. We also observed students' unawareness of their need for help in writing high-quality tests.

CCS CONCEPTS

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

KEYWORDS

unit testing, testing education, checklist

[*]This author performed the work while a student at North Carolina State University.

ACM Reference Format:
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 (ITiCSE 2023), July 8–12, 2023, Turku, Finland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587102.3588781

1 INTRODUCTION

Students often run into trouble when they are learning and practicing software testing. To practice testing, students are expected to learn new concepts, new syntax, and new tools and libraries [5, 18, 27], which could be particularly challenging to beginners as they might still struggle with basic programming concepts [11, 22, 32]. Educators also observed that, when practicing testing, students often make mistakes such as missing boundary testing [5, 9], writing smelly tests [8, 12], testing happy paths only [8, 9], and not testing the program until they are already done with development [27].

While educators have sought to support students' testing practices with tools, such as Ante [10], Marmoset [31], and Testing Tutor [15], a recent study pointed out that students desire *more tool support for unit test composition* [8], particularly in determining *what* code to test and *how* to test it. In response to this, we implemented a lightweight testing checklist in our prior work (*Checklist_v1*) [7], which contains testing techniques (e.g., equivalence class partitioning and boundary value analysis) and tutorial information (e.g., test class components and test smells) to support students in writing quality tests. We evaluated this testing checklist in a lab setting, and our preliminary evidence suggested that the testing checklist can help students as effectively as specific coverage tools in terms of writing quality tests. Our results also suggested that students who have lower prior knowledge in Java and unit testing may benefit more from the testing checklist.

In this paper, we introduce the testing checklist as optional tool support to advanced beginners in testing in the context of a CS1.5 course, in which students are expected to unit test their own source code implementation throughout the semester. To alleviate students' challenges in identifying what scenarios need to be tested and generating test cases that match the program specifications [8], we integrate explicit strategies into the testing checklist (*ETS Checklist*, or *checklist* as shorthand), which emphasizes *how* to work on a problem in a specific way. We aim to explicitly guide students to transfer semantic information from the lab assignment instructions to the possible testing scenarios.

To the best of our knowledge, we are the first to adopt explicit strategies in testing education. To gain insights into the potential costs and benefits of the ETS Checklists in a classroom setting, we invite students to reflect on their use and evaluation of the ETS Checklists via weekly surveys during the study.

In this experience report paper, we share our observations from developing explicit testing strategies, and we analyze the effectiveness of the checklist intervention. Moreover, with the survey responses, we discuss students' engagement with the ETS Checklists from the following aspects: **whether** students adopt the checklists as a form of tool support or not? If yes, **how** do they use the checklists? If no, **why** not, and **what** changes should we make to encourage tool adoption? Our results suggest that, introducing the tools, in this case, the ETS Checklists, earlier in students' learning process could significantly improve students' testing performance and may encourage tool adoption. Our contributions include:

- a lightweight testing checklist containing both tutorial information and explicit testing strategies that siginificantly improved the quality of student-authored test code;
- experiences in adopting explicit strategies to testing education;
- lessons learned in introducing an optional testing tool support to students in a classroom setting.

2 OPEN PROBLEMS AND MOTIVATION

Many students perceive software testing as "uninteresting", "tedious", "unnecessary" and "irrelevant" due to the course-level small-scale projects and exercises [9, 27, 28, 30]. Meanwhile, to perform (automated) testing, students are expected to learn new concepts, new syntax, new tools, and new libraries. This is particularly challenging to beginners as they might still struggle with basic programming concepts [11, 22, 32]. Prior studies [11, 22, 27, 30] also point out that the increased cognitive load placed on learning new tools exaggerate students' negative attitudes towards testing.

Moreover, students have misconceptions about testing and make mistakes in testing. For example, many students view testing as debugging [9, 28], and they often stop testing after designing a test for a presumed bug in the code [9]. Educators also reported that students "rarely saw how test failures can help find bugs in [an] implementation" [20], and hence might modify the failing tests to remove the failure but not the underlying fault [28].

Though the Checklist_v1 [7] helped address some of the problems students encounter during software testing, students expressed their needs for more support in knowing *how* to test. To that end, we adopt explicit strategies for testing education.

The concept of explicit programming strategies was introduced by LaToza et al. [25] as human-executable procedures for accomplishing programming tasks. That is, explicit strategies provide a well-defined series of actions for developers to follow and perform. The researchers found that developers who used explicit strategies were objectively more successful at the design and debugging tasks than those who were free to choose their own strategies. The explicit programming strategies also allowed developers to work in a more organized, systematic, and predictable way, although more constrained at the same time [25].

The goal of the ETS Checklist is to help students transfer semantic information from lab assignment requirements to possible testing scenarios through an unambiguous, step-by-step process. In this work, the explicit strategies focused specifically on requirements coverage. We illustrate the explicit testing strategies using one of the checklists we developed in this study, shown in Section 4.

At the same time, to provide testing support without a steep learning curve, we deliberately keep the ETS Checklists as lightweight as the original version in our prior work [7]. Instead of distributing the checklists via Markdown files on GitHub as prior work, we created a GitHub *Issue* containing the ETS Checklist on each student team's repository, which allows students to check off the items during testing, as suggested in prior work [26].

3 CLASSROOM SETTING

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

Lab Assignments and Team Formation: Students complete a total of 11 labs with three different teams over the semester: they complete Labs 1-4 with *Rotation #1*, Labs 5-8 with *Rotation #2*, and Labs 9-11 with *Rotation #3*. When students rotate to a new team, they are instructed to build off the best existing implementation from their new teammates (i.e., from Labs 4 & 8). Students are randomly assigned into teams by the teaching assistants (TAs). Students collaborate with their teammates and use the same GitHub repository for all labs in one rotation. As the checklist is designed for students who have received some education in software testing but may need some further assistance, we introduce the ETS Checklists to students in the second half of the semester with Labs 6-11.

Educational Support: In Spring 2022, there were nine in-person lab sections, and each lab section has two TAs. All 18 of the TAs received TA training at the beginning of the semester. The TAs start each lab with an overview of the tasks, using a provided set of slides, and recap the concepts and tools that were covered in the lecture course and the methods that students are expected to implement and test. TAs are present throughout the lab to answer students' questions and provide guidance and feedback. Outside of the lab meetings, students could attend the office hours held by all instructors and TAs. They were also encouraged to post questions on Piazza for peer, TA, and instructor support.

This study received IRB approval from NCSU.

4 THE ETS CHECKLIST

We present an example ETS Checklist (as shown in Table 1) in this section to demonstrate the use of explicit testing strategies. While the explicit testing details in this example checklist (item #10) are specific to Lab 6 (Section 4.1) from the laboratory course, the rest of the checklist was used for all labs.

Table 1: One of the ETS Checklists used in this study, which contains the explicit testing strategies designed for Lab 6 (item #10).

Item	Test Case Checklist
	Each test case *should*:
#1	☐ be executable (i.e., it has an `@Test` annotation and can be run via "Run as JUnit Test")
#2	☐ have at least one assert statement or assert an exception is thrown.
#3	☐ evaluate/test only one method
	Each test case *could*:
#4	☐ be descriptively named and commented
#5	☐ If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using `@Before`)
#6	☐ If there are too many assert statements in a single test case (e.g., more than 5), you might split it up so each test evaluates one behavior.
	Test Suite Checklist
	The test suite *should*:
#7	☐ have at least one test for each requirement
#8	☐ appropriately use the setup and teardown code (e.g., `@Before`, which runs before each `@Test`)
#9	☐ contain a fault-revealing test for each bug in the code (i.e., a test that fails)
#10	☐ test an `FSM` (*Explicit Testing Strategies for Lab 6*)
	☐ Test every state:
	☐ Initial state ☐ States in between ☐ Final/End state
	☐ For each state, consider the following scenarios:
	☐ Transition on Letter
	☐ Pick an input with a valid number of transitions
	☐ Min ☐ Between min and max ☐ Max
	☐ Pick an input with an invalid number of transitions
	☐ Less than min ☐ Greater than max
	☐ Transition on Digit
	☐ Pick an input with a valid number of transitions
	☐ Min ☐ Between min and max ☐ Max
	☐ Pick an input with an invalid number of transitions
	☐ Less than min ☐ Greater than max
	☐ Transition on Other
	☐ Invalid, causing an exception
	To improve the test suite, you *could*:
#11	☐ measure code coverage using an appropriate tool, such as EclEmma. Inspect uncovered code and write tests as appropriate.

4.1 Requirements in Lab 6

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

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

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

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

4.2 The ETS Checklist Designed for Lab 6

Unlike the original checklist [7], which reminds students to consider the testing techniques (i.e., equivalence class partitioning and boundary value analysis) at a high level, this ETS Checklist explicitly guides students to divide the possible inputs into partitions and select representatives from each range (item #10 in Table 1).

To design the explicit strategies, one of the authors followed a five-step process, and we validated the checklist with a second author. We illustrate each step using the example of Lab 6:

- **Step 1: Determine the methods that need to be tested.**
  For Lab 6, students are expected to test an FSM that validates a course name string, specifically `isValid()`, which accepts a `String` parameter and returns a `boolean`.
- **Step 2: Apply domain knowledge when necessary.**
  We first remind students that it is important to "*Test every state*" in an FSM, though it is not explicitly stated in the lab assignment instructions.
- **Step 3: Apply testing technique *equivalence class partitioning.***
  For each state, we divide the input based on its type: "*Transition on **Letter/Digit/Other***", which addresses the *Req1 & Req3*. We further divide the partitions into sub-partitions based on the validity of the input: "***valid/invalid** number of transitions*", which addresses the *Reqs1-3*.
- **Step 4: Apply testing technique *boundary value analysis* on each partition.**
  Since values at or close to boundaries of a range often cause problems [3], we remind students to create tests that include representatives of **boundary values** in each range (e.g., "*Min/Between min and max/Max*", addressing *Req1*).
- **Step 5: Instruct students to create a test for each partition.**
  We explicitly ask students to "*Pick an input...*" in each partition and create a test case.

To validate the explicit testing strategies designed for Labs 6-11, we conducted a small pilot study with a first-year graduate student in CS who had no prior experience with this CS1.5 course or other laboratory courses offered at NCSU. This graduate student was instructed to implement the methods required in the lab, and write tests strictly following the ETS Checklist. We measured their test code in terms of instruction coverage and branch coverage. As needed, we modified the wording of the explicit testing strategies to avoid ambiguity and adjusted the sub-items to ensure it was possible to achieve 100% instruction coverage and branch coverage by following the ETS Checklist suggestions.

5 CLASSROOM CHECKLIST INTEGRATION

To gain insights into how students adopt the ETS Checklists and when the support from the ETS Checklists is most appreciated, we divided the in-person lab sections into two groups (balanced in terms of meeting times):

Table 2: Group Assignment and Student Information

	Labs 6 - 8			Labs 9 - 11		
	#Students	#Teams	w/ETS	#Students	#Teams	w/ETS
Grp1	97	33	Yes	94	33	No
Grp2	95	32	No	92	32	Yes

Table 3: Survey Response Rates

	Labs 6 - 8			Labs 9 -11		
	Short_1	Short_2	Comp	Short_1	Short_2	Comp
Grp1	7 (7.2%)	72 (74.2%)	68 (70.1%)	-	-	-
Grp2	-	-	-	72 (78.3%)	71 (77.2%)	45 (48.9%)

- **Group 1 (*Grp1*) - five sections in total**
  - 1 Morning section, 4 Afternoon sections
  - 2 Monday sections, 2 Tuesday sections, 1 Wednesday section
- **Group 2 (*Grp2*) - four sections in total**
  - 1 Morning section, 3 Afternoon sections
  - 2 Monday sections, 1 Tuesday section, 1 Wednesday section

We distributed the ETS Checklists to these two groups in different phases of the semester. For Labs 6-8 (in which students worked in *Rotation #2*), Group 1 received the ETS Checklists (*w/ETS*=Yes in Table 2), while Group 2 did not. For Labs 9-11 (in which students worked in *Rotation #3*), Group2 received the ETS Checklists while Group1 did not. Students in all groups completed their lab assignments and committed to their team repositories for all labs. Study materials are available on GitHub [6].

In this study, students who received the ETS Checklists (*w/ETS group*) were invited to complete three weekly surveys in total: two Short Surveys in weeks 1 & 2 (*Short_1* and *Short_2* in Table 3, respectively) and one Comprehensive Survey in week 3 (*Comp*). The Comprehensive Survey [6] asked students how they used the ETS Checklists as well as how useful they found it. This survey consists of Likert-scale rating, selection, and open-ended questions. The Short Survey [6] only focused on students' use and evaluation of the ETS Checklists. This survey consists of Likert-scale rating and open-ended questions. We did not survey students in the control group as they received no additional resources.

Completion of the surveys and all survey questions were optional. Students were not compensated for completing them. Table 3 shows the response rates. In total, we collected 335 survey responses (Grp1: 147 surveys, Grp2: 188 surveys). The first Short Survey (*Grp1 - Short_1*) was distributed via Qualtrics. To improve the response rates, all subsequent surveys were paper-based.

To analyze the survey responses, we converted the 5-point Likert scale in the rating questions to numbers and treat them as interval-scaled data [19], where 1 maps to the lowest score (e.g., "Not at all helpful" and "Never"), and 5 maps to the highest score (e.g., "Extremely helpful" and "Always"). We qualitatively analyzed the open-ended questions.

6 RESULTS

We discuss how students engage with the ETS Checklists with a series of questions.

Table 4: Answers to Survey Question "How often did you consult the checklist during..." (5-point Likert Scale, where 1="Never", 3="Sometimes", and 5="Always")

Grp.	Survey	System		Unit		Disc		Overall	
		avg	med	avg	med	avg	med	avg	med
Grp1 (L6-8)	Short_1	1.7	2.0	2.4	3.0	2.3	3.0	2.1	2.0
	Short_2	1.6	1.0	2.0	1.0	1.9	2.0	1.8	1.0
	Comp	1.8	1.5	2.0	2.0	2.0	2.0	1.9	2.0
Grp2 (L9-11)	Short_1	1.4	1.0	1.8	1.0	1.6	1.0	1.6	1.0
	Short_2	1.4	1.0	1.7	1.0	1.6	1.0	1.5	1.0
	Comp	1.4	1.0	1.7	1.0	1.5	1.0	1.5	1.0

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

Grp.	Survey	System		Unit		Disc		Overall	
		avg	med	avg	med	avg	med	avg	med
Grp1 (L6-8)	Short_1	2.5	2.5	3.2	3.0	2.8	3.0	2.8	3.0
	Short_2	2.7	3.0	3.2	3.0	3.0	3.0	3.0	3.0
	Comp	2.6	3.0	3.0	3.0	2.7	3.0	2.8	3.0
Grp2 (L9-11)	Short_1	2.5	3.0	3.2	3.0	2.4	2.0	2.7	3.0
	Short_2	2.4	2.0	3.0	3.0	2.6	3.0	2.7	3.0
	Comp	2.4	2.0	3.0	3.0	2.6	2.0	2.7	2.0

Q: Do students appreciate the ETS Checklists?

About half of the students who had the access to the checklists self-reported that they did not use the checklists when working on the lab assignments. However, students who chose to consult the ETS Checklists found them moderately helpful.

Students in the *w/ETS group* self-reported their use (Table 4) and evaluations (Table 5) of the ETS Checklists in two short, weekly surveys (*Short_1*, *Short_2*) and one Comprehensive Survey (*Comp*). While most of the students in Grp1 rarely consulted the checklists, the ones who consulted the checklists considered them "moderately helpful". Having gained more knowledge and experience in testing by the time students in Grp2 saw the ETS Checklists, less of them, compared to Grp1 students, chose to consult the checklists, but they still found the checklists "moderately helpful".

There are several possible interpretations of the reduction in checklist adoption. One is that the checklists were scaffolding that could be removed as students gain more knowledge and experience in testing, particularly since Labs 7-11 all focused on implementing and testing linear data structures such as array-based lists, linked lists, stacks, and queues [6]. Consequently, as students practiced and internalized the skills to test a linear structure, they needed less support from the checklists. Another interpretation is that as the semester approached the end, students had established their own testing workflow, as well as had less time or energy to adopt additional resources even though they found them helpful.

Some students in Grp2 also believed that the ETS Checklists should be introduced earlier and could benefit the beginners. As one student reported in the Comprehensive Survey, "*In my opinion, at this point in the semester, we already know most of the things on the checklist. I think this checklist would be extremely helpful near the beginning of the semester, when we started writing unit tests*".

Q: Why do students opt to not using the ETS Checklists?

Students were not motivated to use the ETS Checklists, or claimed that they did not need extra support in testing.

As 49.3% (165/335) of the weekly surveys students claimed to not use the ETS Checklists at all (Grp1: 54/147 surveys, 36.7%; Grp2: 111/188 surveys, 59.0%), we further explored why students did not use them. Among these 165 survey responses, 83 of them provided open-ended comments on the checklists, including why they did not use them. We summarize the two major reasons students provided:

(1) Students were **not motivated to use** the checklist (29 responses, 34.9%), since
  - they forgot to use the checklist (18 responses, 21.7%).
    E.g., "*Forgot about the checklist so I didn't use it, but it seems helpful*" and, "*I honestly forgot we had the checklist when we did the lab. [...] During group discussion, it was very helpful. Next time I would definitely use it*".
  - they did not want to adopt new tools or new testing practices (7 responses, 8.4%).
    E.g., "*Introduce them at the beginning of the semester - at this point we all have a certain way of delegating and checking off tasks*" and, "*I have done fine without the checklist and see no need to change my habits this closer to the end of the semester*".
  - the adoption of the checklist would not directly improve their grades in lab assignments (4 responses, 4.8%).
    E.g., "*The checklist seems useful but many students share a similar sentiment about it: if you give a student extra work on top of an already long assignment and it isn't for a grade, they won't do it*" and, "*I did not think it was necessary to get a good grade, so I did not use it much since it just took more time*".

(2) Students **did not perceive a need for help** from the checklists (23 responses, 27.7%), since
  - not all students need the same level of tool support, which echoes the finding in prior study [21] (20 responses, 24.1%).
    E.g., "*Did not use, I already have my own checklist in my head*", "*I just didn't really use it. I've never felt I needed the extra help*" and, "*No, I'm a strong independent programmer who does not need a checklist*".
  - students had other sources of help (e.g., teammates and TAs), which echoes the finding that "asking for help is a more efficient strategy" compared to independently solving a problem [23] (3 responses, 3.6%)).
    E.g., "*I just found myself rarely consulting the checklist and instead consulted with my teammates directly*".

Another potential explanation is that the explicit strategies were too explicit and became long to read, which discouraged the students from using them. Students reflected in the weekly surveys that "*It's a useful tool that we probably should have used more. I found it very thorough, although the time investment needed to complete it was more than we were able to afford*" and, "*It feels redundant and adds to the swath of things we already need to consult or track*".

Q: Do students need support from the ETS Checklists?

The quality of student-authored tests suggests the need for more support, and students who chose to use the ETS checklist wrote higher quality tests.

Table 6: Measurements of the quality of student-authored tests: instruction coverage (%), branch coverage (%), mutation coverage (%), and the number of unhappy path tests (%)

Lab	Group	Instruction.	Branch	Mutation	#Unhappy
Lab8	Grp1 (w/ETS)	85.0	77.6	47.9	15.8
	Grp2	84.2	76.5	47.3	16.0
Lab11	Grp1	82.3	74.4	46.3	16.0
	Grp2 (w/ETS)	81.7	74.1	46.1	16.0

Table 7: ANOVA results for comparing the project teams that consulted the ETS Checklists and the teams that did not.

	Independent Variable		
Dependent Variable	*usedETS*	*isGrp1*	*usedETS ∗ isGrp1*
1) Instruction Coverage	0.0200 *	0.3310	0.0033 **
2) Branch Coverage	0.0130 *	0.4935	0.0025 **
3) Mutation Coverage	0.0961	0.7243	0.2851
4) #Unhappy Path Tests	0.0664	0.4499	2.17e-06 ***

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

A primary reason students gave for not using the checklist is that it was *not needed* to create high-quality tests, especially later in the semester (Grp2). However, the analysis on the quality of student-authored tests suggests that the checklist could have helped them write better tests, but they were unaware that they needed help.

We measured the quality of student-authored tests on Labs 8 and 11 (the final team products of *Rotations #2* and *#3*, respectively) with four metrics, and present the averages in Table 6:

- **1) Instruction coverage & 2) Branch coverage**
  We measured these two completeness metrics via EclEmma [1].
- **3) Mutation coverage**
  We measured this effectiveness metric via PITest with its default mutation operators [2, 4, 14].
- **4) The number of unhappy path tests**
  Prior studies [7–9] revealed that students tend to only test the happy paths, and we address this issue via the checklist intervention. However, to our knowledge, there is no existing approach to automatically detect the unhappy path tests. Hence, we opted to only consider the tests that assert an exception is thrown: 1) `assertThrows`, 2) `@Test(expected = Exception.class`, or 3) `Try-Catch` blocks. We counted and adopted these as **the number of unhappy path tests**. This is a coarse measure of completeness.

Treating these four metrics as dependent variables, we measured the impact of the following independent variables: *usedETS* for comparing the teams that self-reported consulting the ETS Checklists (39/130) to the teams that did not (91/130), and *isGrp1* for the time of receiving the checklist intervention (Grp1 vs. Grp2). Note that *usedETS* is only true for students who both had access to the ETS Checklists (randomly assigned) and chose to use it (self-selected), so it combined the *treatment effect* of the ETS Checklists and the *selection effect* of students willing to use it. We used a two-way ANOVA analysis to explore the impact of each variable independently as well as their interaction (*usedETS ∗ isGrp1*). 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 [24]. We report the p-values in Table 7.

The analysis of test code quality suggests that using the ETS Checklists had a statistically significant impact on the completeness of student-authored tests ($p = 0.0200$ for instruction coverage and $p = 0.0130$ for branch coverage). The adoption of the

ETS Checklists also had statistically significant interaction effects ($usedETS * isGrp1$) on instruction coverage, branch coverage, and the number of unhappy path tests. This indicates that students benefited more from the ETS Checklists in their early learning process, which was conjectured in prior work [7].

For both Grp1 and Grp2 and for all labs, the average mutation coverage is less than 50%, meaning that less than half of the mutated code could be detected by the student-authored tests. We found that using ETS Checklists had a large effect size on the mutation coverage with Cohen's d of = 1.92, though it is statistically non-significant ($p = 0.0961$). This suggests that the checklist could have helped students achieve higher mutation coverage, but they were unaware that they needed help.

These findings echo the observations in prior studies that students may avoid help due to 1) unawareness of the need for help [17], 2) concerns about the help-giver's (in this case, a checklist) competence [16, 29], and 3) a desire for independence [16].

7 DISCUSSION

In this study, we surveyed students on their use and evaluation of the ETS Checklists, and we analyzed the effectiveness of the checklists with student-authored tests. This information shed light on potential practices of introducing the ETS Checklists as a form of testing tool support to a classroom.

7.1 Low Motivation to Adopt New Tools

In a prior study [7], we conjectured that the checklist could potentially address students' hesitation to adopt new tools or struggle to use them effectively given the minimal learning barrier. However, in this study, we found that students tended to not adopt tool support that is not integrated into their workflow – despite the fact that our results suggest they both needed help and may have benefited from the checklist. Students overlooking the importance of testing could also lead to the low motivation of tool adoption.

We propose following approaches to encourage adoption of the ETS Checklists:

7.1.1 Integrate the ETS Checklists into the program requirements in the assignments. Instead of presenting the explicit testing strategies designed for all methods in the checklists, we could offer them as optional support for students (e.g., hidden in a question mark button), and provide them next to its associated methods in the lab assignment instructions. This could reduce the amount of reading and hence avoid overwhelming students, as well as avoid overtaxing those who do not need extra support in testing these methods.

7.1.2 Have students develop their own ETS Checklists. Ko et al. [23] introduced the explicit strategies in debugging and code reusing to an introductory programming course. They found that though provided the more systematic strategies, students preferred to engage in rapid cycles of editing and testing, without deeply understanding their code. They also pointed out that the strategies might be too sophisticated to learn while also learning basic programming concepts. Meanwhile, Chong and colleagues [13] reported that having students create their own code review checklist stimulated students in developing their analytical skills for code reviews. Hence, we encourage exploration of potential benefits of having students develop their own ETS Checklists. This activity may help students understand the value of testing, and it could also lead to insights into students' decision-making process when practicing testing and hence inform better teaching practices.

7.2 Threats to Validity

Conclusion: The lab assignment instructions explicitly required students to achieve 80% statement coverage on every non-UI and non-test class. This threshold of code coverage was applied to all submissions from both groups, and might have led to overestimation of students' performance on completeness metrics.

The "#Unhappy Path Tests" metric focused only on tests in which an exception is thrown while there are other ways in which unhappy paths can be exercised. This incomplete approximation of unhappy paths may impact the conclusions drawn from the metric.

Internal: The adoption of the ETS Checklists and the completion of weekly surveys on use of the checklists were optional, it subject to selection bias, and consequently conclusions drawn from it may not generalize. However, the code coverage thresholds were applied to all student submissions, and we received high response rates (>70%) from both groups, so conclusions are drawn from a majority of the students. We only observed students' interactions with a codebase that they were familiar with (either implemented by students themselves or their peers). Students may perform differently on an unfamiliar codebase.

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

8 CONCLUSION & FUTURE WORK

In response to students' needs for support in identifying *how* to test their code [8], in this study, we designed explicit strategies in unit testing, which provide step-by-step guidance on how to test their own source code implementation with scenarios that match to the lab assignment requirements. We integrated the explicit testing strategies to a lightweight checklist [7] that has been found to be helpful in assisting student in writing quality tests, and we introduced the ETS Checklist as optional tool support to advanced beginners in a CS1.5 course. Our results suggest that consulting the ETS Checklists could significantly improve the quality of student-authored tests; however, most of the students were unaware of their needs for help. This study also reveals that students did not engage with the checklists as a way of seeking help when performing testing in a classroom setting. To better understand and hence address students' needs in a particular class and its associated programming labs, future work could investigate students' help-seeking behaviors through platforms like Piazza. Future studies in classroom settings could integrate the ETS Checklists directly into the program requirements to improve the skimmability and reduce context switching when working on the assignment. We also encourage studies on exploring the potential benefits of having students write their own explicit strategies.

9 ACKNOWLEDGEMENTS

This work is supported in part by NSF IUSE #2141923 and NSF SHF #1749936.

REFERENCES

[1] [n.d.]. EclEmma: Coverage Counters. https://www.eclemma.org/jacoco/trunk/doc/counters.html. Accessed: 2022-08-19.

[2] [n.d.]. PITest: Mutation Operators. http://pitest.org/quickstart/mutators/. Accessed: 2022-08-19.

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

[4] Michael Andersson. 2017. An Experimental Evaluation of PIT's Mutation Operators. , 27 pages.

[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] Gina R. Bai. 2023. *ginaBai/ExplicitTestingStrategiesStudy:StudyMaterials*. https://doi.org/10.5281/zenodo.7702953

[7] 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)*.

[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 *2021 IEEE/ACM 43rd International Conference 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] 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).

[12] 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.

[13] 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.

[14] 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.

[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. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 87–93.

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

[17] Sharon Nelson-Le Gall. 1981. Help-seeking: An understudied problem-solving skill in children. 1, 3 (1981), 224–246.

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

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Amy J Ko. 2020. Explicit programming strategies. *Empirical Software Engineering* 25, 4 (2020), 2416–2449.

[26] 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*.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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