The Significance of Positive Verification in Unit Test Assessment

Kevin Buffardi
California State University, Chico
kbuffardi@csuchico.edu

Pedro Valdivia
California State University, Chico
pvaldivia1@mail.csuchico.edu

Abstract

This study investigates whether computer science students' unit tests can positively verify acceptable implementations. The first phase uses between-subject comparisons to reveal students' tendencies to write tests that yield inaccurate outcomes by either failing acceptable solutions or by passing implementations containing bugs. The second phase uses a novel all-function-pairs technique to compare a student's test performance, independently across multiple functions. The study reveals that students struggle with positive verification and doing so is associated with producing implementations with more bugs. Additionally, students with poor positive verification produce similar number of bugs as those with poor bug identification.

1. Introduction

Although software engineering commonly expends about half of a project's costs on testing [16], learning how to test is neglected in most computer science curricula. However, Association for Computing Machinery's (ACM) most recent recommendation [1] recognizes a need to incorporate testing within two core knowledge areas: Software Development Fundamentals and Software Engineering. The recommendations specifically identify *unit testing* as a topic that may be appropriate to explore as early as introductory programming (CS1) courses. By incorporating more testing in computing education, students may learn to improve their debugging strategies and further their metacognitive problem-solving skills [9].

Accordingly, educators have been exploring how to both effectively teach and evaluate testing in computer science classes. Pedagogical tools [6,13,19] and approaches [7,12] have been developed to help introduce students to testing throughout the curricula.

However, there is not yet a *de facto* standard for evaluating the quality of students' tests. Studies of software testing quality predominantly concentrate on three factors: cost, code coverage, and the ability to find software faults [16]. Automated assessment of students' tests (autograders) typically use coverage, a measurement of how much code has been executed by their tests. Otherwise, some tools and studies evaluate students' tests with a concentration on their ability to identify bugs, as accomplished by failing known faulty implementations [11,13,19]. However, there has neither been thorough discussion nor evaluation of how well students' tests accurately confirm acceptable solutions. To depend on tests as diagnostic tools for software verification, they should effectively differentiate between acceptable and faulty implementations [5]. Most research has focused its attention on the latter.

From an educational perspective, Edwards recognized that "Software testing promotes the hypothesis-forming and experimental validation that are central to [...] reflection in action" [9]. However, experimental validation via software testing requires not only failing faulty code, but also passing proper implementations.

In the studies described in this paper, we investigate the significance of students' ability to positively verify acceptable solutions. In the first phase of the study, we identify trends in students' test outcomes by investigating the relationship between mistakenly failing good implementations and passing those with bugs. In a second phase of the study, we examine each student's testing outcomes as their unit tests were measured against individual functions-under-test.

Both phases of the study explore the role of positively verifying software and its relationship with the complementary goals of discovering bugs and of producing solutions with fewer bugs. Specifically, we address the following research questions: Do novice testers struggle with positive verification? What are the ramifications of deficiencies in positive verification? Within a given student's test suite, how does their test performance compare across different functions? The study reveals a need for more attention to be given to students' abilities to positively verify acceptable solutions in addition to failing implementations with bugs.

HICSS

2. Background

Goldwasser proposed a novel approach to integrating software testing in programming courses by running each student's assignment solution against each other student's tests and evaluating the quality of the tests accordingly. The initial experiment with all-pairs analysis found that students who produced correct solutions for a programming assignment also produced tests that exposed 87% of flawed programs while the tests of those students with incorrect solutions only exposed 63% [14].

Edwards, et al. provided an update to the all-pairs approach to support more flexibility in design as well as integration with an online autograder [12]. In their study of 101 students' solutions, they found that a majority of students passed at least 90% of the test case corpus but only 5 students managed to pass all test cases. In a later study, Edwards and Shams found that while students achieved high coverage scores, most students produced "happy path" tests that the researchers described as "writing basic test cases covering mainstream expected behavior rather than writing tests designed to detect hidden bugs" [10]. However, with the prevalence of coverage tools and their ability to provide quick analysis of tests, many instructors use coverage as a basis for assessing students' unit tests.

Coverage is a metric commonly used in industry that indicates what code has been executed after test cases have been invoked. The most common form (i.e. "line coverage") refers to the percentage of statements or non-comment/non-blank lines of code executed. There are also more comprehensive forms, such as condition/decision coverage, which evaluates whether control operations (decisions) have been evaluated for both true and false outcomes as well as whether each atomic conditional (within boolean expressions) has been evaluated as both true and false. However, as the aforementioned study demonstrated, high coverage does not necessarily signify effective tests.

To the contrary, mutation testing is an approach employed in industry that bears similarity to all-pairs analysis by examining test outcomes on buggy solutions. Instead of depending on a corpus of student solutions, mutation testing tools generate bugs by altering the current implementation. For example, a tool can automatically generate a 'mutant' by changing a condition by replacing an equality comparison to a less-than-or-equal-to operator. Each mutant is intended to change the behavior of the program. Mutation testing involves creating many of these synthetic mutants and assessing the test suite by measuring the percentage of mutants 'killed' by failing them. Aaltonen, et al. recommend combining mutation testing with coverage as complementary assessments of test effectiveness [2]. However, in an experiment comparing techniques for assessing test quality, neither coverage nor mutation testing were as effective as all-pairs testing at predicting how well tests find bugs [11].

There are also eLearning tools specifically for fostering testing skills. Smith, et al. created an automated tool that evaluates students' tests against a large dataset of known buggy solutions and provides feedback based on how well a test fails those bugs [19]. However, like both all-pairs analysis and mutation testing, their tool concentrates on the ability to fail bugs and does not measure rates of positive verification.

On the other hand, Bradshaw recognized that, "a perfect test will only accept implementations that are correct and reject all other incorrect implementations." Accordingly, Bradshaw's Ante Up tool first compares students' tests to the instructor's (correct) solution to make sure they pass, before allowing students to progress onto developing their own solutions. While the tool offers a unique reinforcement of test-first workflows, it does not support other testing methods. Bradshaw has yet to report on the tool's effectiveness and consequently has not published analysis of positive verification [6].

Meanwhile, Bowes, et al. collaborated with industry partners to compile a list of best practices in testing, along with corresponding metrics for evaluating those practices. Their recommendations include the principle of "Happy vs. Sad tests [which are] associated with the goals of testing: to verify the system (also known as happy tests) vs. to break the system (also known as sad tests)" [5]. However, they did not identify any associated metrics to measure these goals.

Consequently, we studied students' abilities to produce happy tests *as well as* sad tests. The previous research indicates that effective sad tests are associated with creating implementations with fewer bugs. To supplement replication of those findings, we investigated potential relationships between happy tests (or a lack thereof) and the prevalence of bugs.

3. Method

We considered the possibility that students may not exhibit the problem of failing acceptable solutions. Consequently, the first research question we addressed was: **(RQ1) do novice testers struggle with positive verification?** Upon observing any struggles, we also set out to investigate **(RQ2) what are the ramifications of deficiencies in positive verification**; and **(RQ3) within a given student's test suite, how**

does their test performance compare across different functions? We studied students' positive and negative testing outcomes with an in-class learning activity.

To emphasize the purpose of differentiating between good and bad implementations, students were randomly assigned to either implement a working solution or to write an implementation that compiles but does not behave correctly. However, all students were instructed to the common task of writing unit tests along with their implementations that should be able to distinguish the good implementations from the faulty ones.

To ensure that students' test cases would be compatible with each other's implementations, they were provided with a common build script and interface for a class—which specifies the signatures of all public functions—along with plain-word descriptions of each function's expected behavior. For our study, the programming assignment was a data model for a Tic-Tac-Toe board with functions to: place pieces onto the board, inspect a location for its piece (or lack thereof), toggle turns between players, and determine the game state (e.g. is game ongoing without a winner, completed as a tie, or which player has won).

The function for placing and inspecting pieces on a board were *not* just one-line accessors/mutators (i.e. getters and setters with no logical branches). Their expected behaviors included validation of the coordinates provided with special return values to indicate out-of-bound coordinates or when preventing a piece from being placed where there is already another.

Although students were assigned to different roles for implementing incorrect and correct solutions, students may not achieve those goals. Therefore, it is necessary to characterize each solution based on its performance against the instructor's reference tests. Using an automated script, the reference tests were run against each student's implementation and recorded the solution as *positive* if it passed all tests or *negative* if it failed any tests. To be consistent with unit testing conventions, failures included tests that timed out (usually indicating an infinite loop) or exited with an unexpected fault or uncaught exception.

In addition, to promote confidence in accuracy of the all-pairs analysis, we reviewed the results for potential mistakes or gaps in their reference tests. It is possible, for example, that a student writes a test case that identifies a bug that the instructor's reference tests did not consider. In that example, the reference tests might pass a negative solution that contains that bug while the student's tests (correctly) fail it.

During this process, special attention should be given to test suites that demonstrate high, but not perfect rates of failing negative solutions and passing positive ones. If the instructor discovers any shortcomings in their test suite, it will be necessary to revise the reference tests accordingly and repeat both phases of reference tests against each solution and all-pairs analysis. This process can be repeated as necessary as a feedback loop to ensure comprehensive and accurate assessment.

In similar instrumentation to all-pairs analysis [12,14], each student's test suite runs against each other student's solution and records whether it passes or fails each implementation. A perfect test suite should pass all positive solutions and fail all negative solutions. Table 1 illustrates how test suite outcomes are assessed based on how well they pass positives (True Positive) and fail negatives (True Negative) and conversely, how they incorrectly fail positives (False Negative) and pass negatives (False Positive).

Both False Positives and False Negatives indicate inaccurate conclusions from the test suite, comparable to Type I and Type II errors in hypothesis testing [18]. However, classification terminology used in this paper should *not* be confused with their application in medical diagnosis testing, where a positive diagnostic test result usually indicates presence of a disease or condition (i.e. confirming bad news). Instead, in the context of software testing, we refer to positive verification (true positive) as confirmation of an acceptable solution.

False Negatives mislead software developers to thinking an acceptable solution contains faults. This inaccurately describes the expected behavior from the software. Consequently, False Negatives may add cost to development by dedicating time to trying to discover a non-existent fault or by errantly changing the software's behavior to satisfy the inaccurate test.

On the other hand, False Positives result from tests passing a faulty implementation. Overlooked bugs have negative repercussions since they result in poor quality software. This can be particularly costly if those bugs are not discovered before the software is deployed and faulty software is delivered to the customer. However, False Positives are costly even when the fault is later discovered because localizing and fixing a bug can be more difficult when inaccurate unit tests give a false sense of confidence in a function's acceptability.

Table 1. Classification of Verification

		Implementation Acceptability	
		Positive	Negative
Test	Pass	True Positive	False Positive
Outcome	Fail	False Negative	True Negative

3.1. Phase one: between-subject design

In two Software Engineering classes (one upper-division undergraduate requirement, the other a graduate level requirement in a Masters-only program), students were introduced to the aforementioned learning exercise during a course module on testing. All students had previous object-oriented programming courses but did not have formal instruction to testing in the curriculum. Students were taught how to set up and run GoogleTest (an open source xUnit framework for C++ [15]) on a project and were introduced to the syntax and semantics of unit tests. Students were randomly assigned correct/incorrect implementation roles by the course management system; random assignment was independent for either course so that undergraduate (n=40) and graduate (n=8) classes were each split evenly between the roles. Both the graduate and undergraduate courses had multiple object-oriented programming course prerequisites, but no prior classes explicitly taught unit testing.

Students worked independently and submitted their solutions and test suites at the end of the lecture regardless of whether they considered their work complete. However, when analyzing the results of reference tests ran against the students' solutions, we found that none of the solutions passed *all* of the reference tests. Consequently, we decided to extend the assignment to a second lecture and we instructed all students to attempt a correct solution (including those previously assigned to the incorrect group) while continuing to improve their tests as well, which yielded five positive solutions.

At the end of the course module on testing, all students took a practical quiz with a similar format to the assignment. However, while students were still instructed to write tests that distinguish between positive and negative solutions, all students were asked to try to implement a correct solution for the quiz. The problem posed to the students for the quiz was of similar nature to the Tic-Tac-Toe interface, but used a different game of the instructor's creation. The students worked on the quiz in lecture but were allowed until the end of the day to complete it and submit their work online.

To assess students' ability to positively verify solutions and identify faults, we calculated their test suites' true positive and true negative rates, respectively. The True Positive Rate (TPR) is the percentage of positive solutions passed. The True Negative Rate (TNR) is the percentage of negative implementations failed. From the initial learning exercise, students produced TPR (M=0.61, sd=0.42) and TNR (M=0.72, sd=0.36) with relatively large

variance. Similarly, condition/decision coverage (M=0.57, sd=0.33) achieved varied considerably, within the limited time allowed. We investigated the relationships between TPR and TNR with Spearman's rank correlation found a strong negative correlation (ρ=-0.85, p<.0001) between rates of true positives and true negatives.

A negative correlation between the two measurements of testing accuracy may come as a surprise. However, Figure 1 reveals that the relationship appears to be strongly influenced by test suites at the extremes of either rate. The chart also illustrates an interesting phenomenon that while some students' test suites perform relatively well on both TPR and TNR (data in upper-right corner of chart), there are also several test suites that perform well on one but poorly on the other. There were no students who scored below 75% on both TPR and TNR.

This discovery answers our initial research question, **(RQ1) do novice testers struggle with positive verification?** *Yes, students' tests sometimes exhibit problems with positive verification by producing many false negatives*. Compounding the problem further, those same tests with low TPR also tend to have a high rate of TNR. In other words, some test suites are effective at failing faulty implementations but simultaneously fail good solutions.

In addition, we investigated the relationship between test outcomes and their associated implementations. We calculated a Multiple Linear Regression model for predicting students' implementation correctness (as calculated by the percentage of reference tests passed) based on TPR,
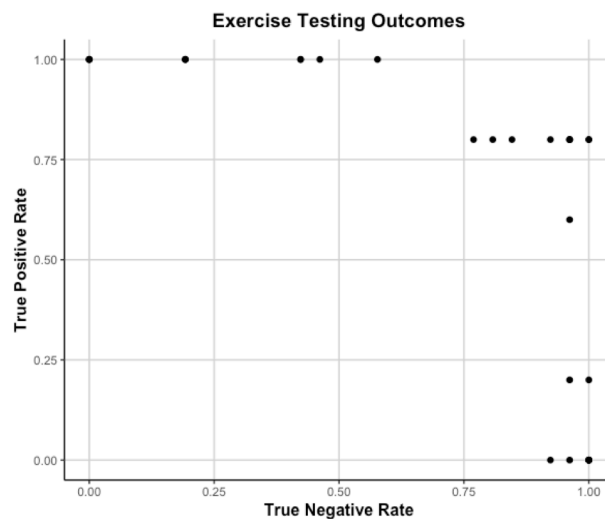
Figure 1: Test suites' rate of passing positive (acceptable) solutions and failing negative (faulty) solutions

TNR, and condition/decision coverage as potential predictors. After setting a more rigorous standard by adjusting the critical value for considering multiple factors (α=0.0167), we found that condition/decision coverage (B=0.01, Std. Error=0.07) was not significant (p=0.87) while both TPR (B=0.17, Std. Error=0.06, p<.01) and TNR (B=0.24, Std. Error=0.07, p<.01) had significant coefficients. We recalculated the model by excluding the non-significant condition/decision coverage factor and found a significant regression equation (F(2,27)=10.92, p<.001, R^2=0.45) where students' predicted implementation correctness is equal to 0.06 + 0.25*TNR + 0.18*TPR. Students' solution correctness increased 25 percentage points for failing all negative implementations and 18 points for passing all positive implementations; TNR (p<.001) and TPR (p<.001) were both significant predictors.

Next, we examined whether the students' test suites for the quiz also demonstrated similar traits when analyzed only with naturally occurring bugs. We repeated the automated analysis of characterizing implementations as positive or negative and then calculated each test suite's TPR (M=0.62, sd=0.39) and TNR (M=0.82, sd=0.17) after running all-pairs analysis against each implementation. Intentionally created bugs from the previous assignment posed a potential threat to validity so we also investigated the phenomena with the quiz, where all bugs were naturally occurring. When plotting the outcomes from the quiz, Figure 2 illustrates a similar relationship between TPR and TNR as found during the learning exercise.

Consequently, we combined test suite assessments from the exercise and quiz together and performed k-means clustering (using the R statistical package) to identify three primary clusters: *False Negatives* that failed most negative and positive solutions; *False Positives* that passed most positive and negative solutions; and *True discriminators* that had few incidences of either type of error. Each cluster's TPR and TNR scores are summarized in Table 2 and plotted in Figure 2.

To test both hypotheses and compare how each cluster of students performed on their respective implementations, we performed three Wilcox-Mann-Whitney tests for pairwise comparison between each cluster's solution correctness. To account for increased likelihood of significance when making multiple comparisons, we used the Bonferroni method for a more conservative critical value (α=0.0167). We found that True Discriminators (M=0.94, sd=0.13) had significantly better solution correctness (p<.0001) than False Negatives (M=0.73, sd=0.31) as well as significantly better correctness (p<.0001) than False Positives (M=0.77, sd=0.27). However, there was no

significant difference (p=0.82) in correctness between False Negatives (M=0.73, sd=0.31) and False Positives (M=0.77, sd=0.27).

Both *False Negatives* and *False Positives* are associated with worse solutions than *True Discriminators*, as might be expected. Perhaps the simplest explanation could be that students who make mistakes in their implementations are also likely to make mistakes in their tests. Consequently, we investigated whether testing errors could be attributed to individual differences between students.

Findings from our between-subject comparisons provide initial insight into our research question **(RQ2) what are the ramifications of deficiencies in positive verification?** *Test suites with high rates of any kind of outcome inaccuracies are associated with more bugs than those with few errors*. However, the results from this phase also suggest that *either type of testing error is associated with corresponding implementations with*

Table 2. Descriptive statistics for true positive and true negative rates, grouped by three clusters

	True Positive Rate		*True Negative Rate*	
	Mean	std dev	Mean	std dev
False Negative	0.03	0.06	0.96	0.07
False Positive	0.99	0.02	0.24	0.19
True Discriminator	0.82	0.09	0.84	0.14
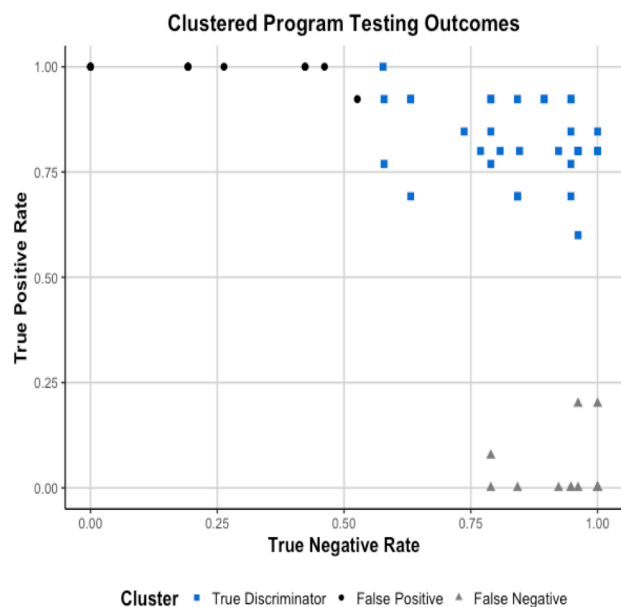
Figure 2. False positive, True discriminator, and False negative clusters

comparable number of bugs. Consequently, deficiencies in positive verification were neither more nor less harmful than deficiencies in fault identification.

Some might consider it more difficult to write sad tests that effectively identify bugs (and produce a high TNR) than it is to write happy tests that positively verify solutions (with high TPR), deficiencies in either appears to be similarly harmful in terms of fault prevalence.

When students attempted the initial learning exercise, it resulted in no implementations that passed 100% of the instructor's tests. Even after more time was provided, relatively few students achieved their goal of passing all reference tests. This low incidence rate of "perfect" solutions uncovers an impediment to evaluating test accuracy using all-pairs analysis: it depends on some students producing full program implementations without any known faults. Low rates of students who can produce implementations with no bugs was similarly observed in Edwards et al.'s all-pairs analysis [12].

It is a hindrance to depend on students producing implementations without any bugs when they do so at a low rate. In addition, assessing unit tests by their ability to find a fault in an entire program ignores the primary objective of unit tests: to test individual functions rather than a program or module as a whole [5]. Given a cohort of students working on the same programming assignment, it is more likely to yield positive implementations of individual functions within a class than it is to yield an entire class comprised of entirely positive implementations.

Furthermore, we considered the possibility that the association between implementation and test quality might be explained by individual differences in student aptitude: strong students may do well at both while weaker students may perform poorly at both. Consequently, we developed all-function-pairs analysis as an approach to assessing whether unit tests accurately pass or fail different implementations of individual functions. With test outcomes measured at the scope of individual functions rather than entire programs, all-function-pairs analysis allowed us to investigate within-subject performance across testing multiple functions.

3.2. All-function-pairs analysis

The first task in all-function-pairs analysis is to determine the acceptability of each function within each student's program. Since the instructor reference tests already followed suggested practices for proper unit testing, each of the reference tests already targeted individual functions. We identified subsets (T_{ful}) of the reference test suite by their respective function-under-test (*FUT*). Consequently, instead of running our automated all-pairs analysis on the entire reference test suite at once, we only ran one reference test at a time and recorded its pass or failure. If a student's implementation passes each of the reference tests in the subset for a given function, that function implementation is positive.

For example, the entire reference test suite targeted the *TicTacToeBoard* class, but a subset of the suite tests the *getWinner* function. Passing each unit test in that subset indicates *getWinner* is positive. Otherwise, any failures indicate a negative function implementation for *getWinner*. Test runners for xUnit frameworks usually include options for specifying individual tests to run, including popular packages such as GoogleTest (C++), JUnit (Java), and unittest (Python). Therefore, this approach can be instrumented for popular CS1 programming languages that support xUnit testing.

Next, all-function-pairs analysis needs to similarly identify the function-under-test for each unit test in students' test suites. It can be more difficult to identify which function a student's unit test intends to verify if the student did not adhere to best practices of unit testing. There are different plausible solutions to this challenge. An instructor may choose to manually inspect student tests to identify their functions-under-test. Alternatively, they may require students to follow test naming conventions or annotations to self-identify the function-under-test for each unit test they write. Otherwise, using static analysis to identify which function's return value (or its output or other side-effect) is used in the test's assertions may serve as a sufficient proxy. In an effort to localize bugs in students' programs, Buffardi & Edwards [8] proposed an automated approach that may also identify the functional scope of tests by analyzing their coverage. For this study, we manually identified functions-under-test.

After identifying the function-under-test for every unit test, the subset can be run against the corpus of all student implementations, one at a time. If an implementation fails on any of the unit tests within that subset, the student has produced a true negative (if the function implementation is negative) or a false negative (if the function implementation is positive). If the implementation passes each test in the subset—or in the case of a null subset—the student has produced a true positive (if the function implementation is positive) or false positive (if the function implementation is negative). Similarly to the feedback loop we recommended for all-pairs analysis, it is advisable to examine the test results to revise the
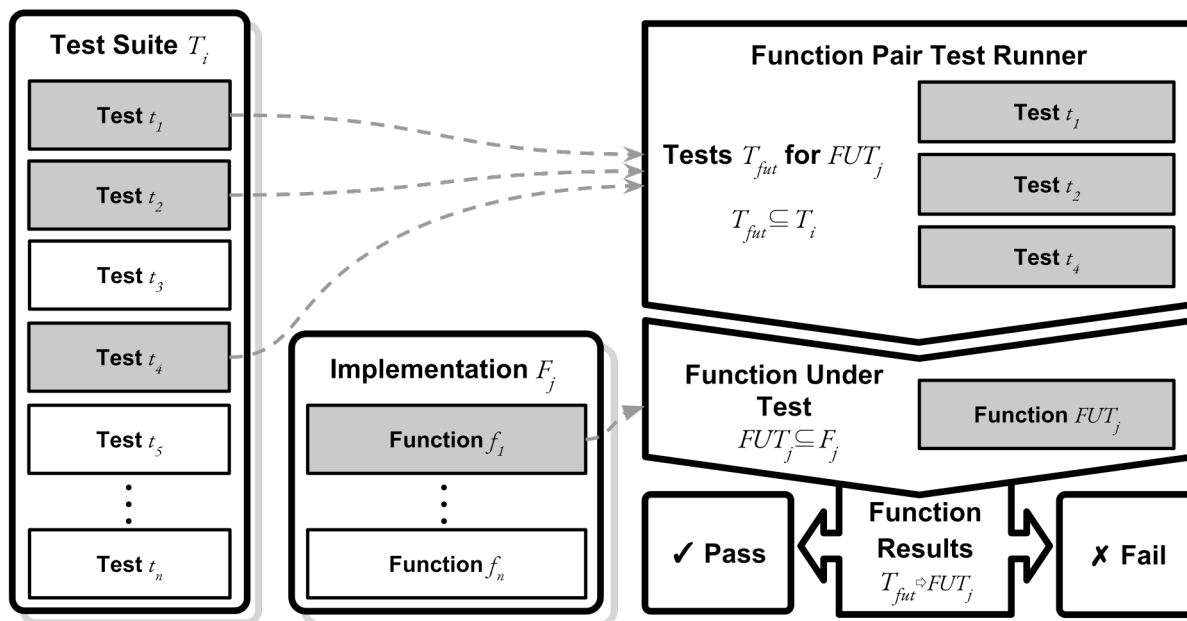
Figure 3. All-function-pairs analysis

reference test suite if any missing test cases are revealed by the corpus of student tests.

We distinguish this approach from Goldwasser's and Edwards' previous experiments by referring to it as all-function-pairs analysis since it evaluates individual functions rather than whole programs. All-function-pairs analysis benefit from higher probability of yielding positive implementations since it only requires individual functions—rather than the entire system—to behave acceptably. Consequently, the approach allows for more granular analysis of both students' implementations and their tests. Figure 3 illustrates the process involved in all-function-pairs analysis.

3.3. Phase two: within-subject design

In the first part of the study, we implemented a between-subject design to evaluate test quality on a learning exercise as well as a quiz. Evaluating true positive and false positive rates of tests using traditional all-pairs analysis depends on some students producing complete solution without any bugs. However, the all-function-pairs analysis approach described in the previous section only depends on some students producing implementations of individual functions without bugs.

With insight into test outcomes for individual functions, we concentrated on the research question: **(RQ3) within a given student's test suite, how does their test performance compare across different functions?** In particular, we needed to test the hypothesis that the association between test and solution quality could just be attributed to general differences in aptitude between individual students. To do so, within-subject comparisons were necessary to identify whether students tended to produce the same kind of testing error (or lack thereof) across multiple functions.

For the subsequent semester, we designed a variation of the learning activity where each student is assigned *one specific function* in which to purposely hide a bug, while attempting to *correctly implement all other functions in the assignment*. Instead of just differentiating between completely flawless and buggy programs, the students were challenged to distinguish between acceptable and unacceptable implementations of each function. We assigned each student (n=39) randomly to one (of four) functions for the aforementioned *Tic-Tac-Toe* data model. As a result, for each function, one-quarter of the students deliberately hid bugs while the remaining three-quarters attempted a correct solution. Additionally, function implementations are each evaluated independently so all-function-pairs analysis yields a *k*-fold increase of test outcomes for each student over simple all-pairs analysis (where *k* is the number of functions, in this case *k*=4).

First, we investigated the relationships between tests that produce false positives and false negatives with their corresponding implementations to validate

our analysis from the between-subject study. As we gain more granular insight into implementations and tests, we expected to find similar phenomena at the function-level as we did at the program-level.

After performing all-function-pairs analysis, we found the True Positive Rate (TPR, M=0.75, sd=0.39) and True Negative Rate (TNR, M=0.56, sd=0.39) of students' tests at testing individual functions. We plotted the testing outcomes for each student on each function and found a similar pattern to the first part of the study: many students had both TPR and TNR above chance (*True Discriminator*), but some students had high TPR with low TNR (*False Negative*) while others had high TNR with low TPR (*False Positive*). This result replicates the phenomenon observed in phase one and is illustrated in Figure 4 with k-means cluster analysis that identifies False Negatives, False Positives, and True Discriminator clusters.

We performed three Wilcox-Mann-Whitney tests for pairwise comparison between each cluster and the corresponding function correctness (calculated by the percentage of reference tests passed from the function-under-test subset, excluding the function in which each student purposely hid a bug) with adjusted critical value (α=0.0167). We found that True Discriminators (M=0.87, sd=0.18) had significantly better correctness (p<.0167) than False Negatives (M=0.7, sd=0.27) and approaching significantly better correctness (p=0.029) than False Positives (M=0.72, sd=0.3). There was no significant difference (p=0.65) in correctness between False Negatives and False Positives.
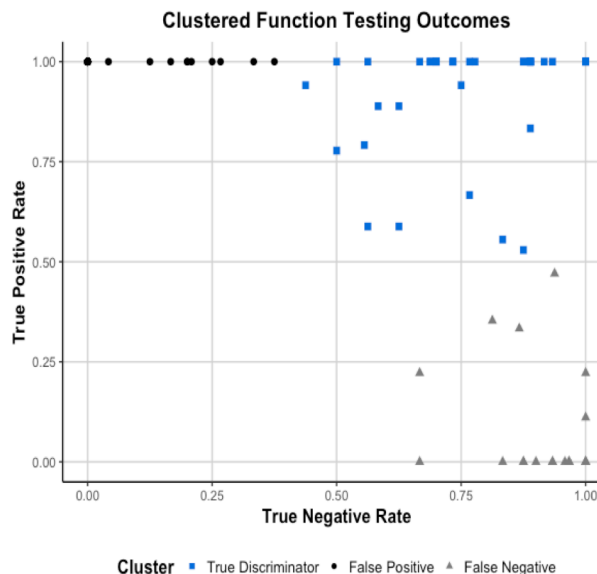
Figure 4. False positive, true discriminator, and false negative clusters for testing functions

The insignificant difference between False Positives and False Negatives replicates phase one's finding that both types of errors are associated with implementations containing comparable numbers of bugs. Phase two's finding confirms this association with implementations of individual functions.

Nevertheless, we considered the possibility that a relationship between test accuracy and implementation correctness might result from differences between students. For example, highly skilled students may produce fewer bugs in their implementations and write more accurate tests than students who are less diligent or less skilled. Accordingly, weak students potentially could be to blame for all of the False Positives and False Negatives while only strong students could have produced all the True Discriminator function tests.

We examined each student's three (excluding their purposely buggy) functions and identified the cluster to which their corresponding tests belonged. We then examined how often a student's test belonged to the same cluster for multiple functions. Overall, 63% of students produced True Discriminator test outcomes for at least one function; 56% of students produced False Positives for at least one function; and 47% of students produced False Negatives for at least one function.

By examining how many different clusters were represented among each student's three functions, we found that 63% of students had more than one cluster type. Since most students produced a mix of different test outcomes, we expected that the association between test accuracy and function implementation *was not* a simple result of divergent strong and weak students.

For due diligence, we accounted for differences between students that could affect the overall relationship between test accuracy and implementation correctness. Therefore, we performed a repeated measures analysis of variance (rANOVA) to compare the effect of a student's test clusters on the correctness on their (three) corresponding function implementations.

We found a significant effect of test clusters on function correctness (F(2,95)=4.34, p<.05) in within-subject comparisons. The rANOVAs significant effect suggests that unit test accuracy is still associated with fewer bugs in corresponding functions when compared within subjects.

Finally, we calculated a Multiple Linear Regression model for predicting students' function correctness based on TPR and TNR as potential predictors (with adjusted critical value α=0.025). We found that both TPR (B=0.28, Std. Error=0.08, p<.001) and TNR (B=0.26, Std. Error=0.08, p<.001) for a function had significant coefficients and found a significant

regression equation $(F(2,93)=8.064$, $p<.0001$, $R^2=0.13$). Students' predicted implementation correctness for a given function was equal to $0.43 + 0.28*TPR + 0.26*TNR$ for the TPR and TNR of the tests for that same function.

The within-subjects comparison addresses the question **(RQ3) within a given student's test suite, how does their test performance compare across different functions?** Most students tested well (with high rates of true negatives and true positives) for at least one function but also struggled with testing errors (either false positives or false negatives) on other functions. Individuals did not consistently produce the same type of testing deficiencies (or lack thereof) across all functions. However, struggling with either type of error when testing a function corresponds to a comparable number of bugs in the function's implementation. This finding even applies to an individual student's testing and implementation of difference functions, regardless of their overall aptitude.

4. Conclusions

In our two-phase study, we explored whether novice student testers struggled to positively verify software by writing tests that accurately confirm when implementations are acceptable. We found that students commonly struggled with either positive verification or fault identification, but never both simultaneously on the same function code to a substantial degree. Problems with positive verification are demonstrated when tests produce many false negatives: failing implementations from a corpus of acceptable solutions. Meanwhile, problems with fault identification are characterized by producing many false positives: passing implementations that contain faults.

Previous studies concentrated on evaluating student tests by their performance at fault identification. Our results confirmed previous findings that tests' ability to identify faults is associated with writing an implementation with fewer bugs. However, our study also revealed that the rate at which tests positively verify a corpus of known good solutions is also associated with a corresponding implementation with fewer bugs.

Congruently, we found the rate of positive verification (true positive rate, TPR) and the rate of fault identification (true negative rate, TNR) are both significant predictors of implementation correctness. Tests with high rates of both TPR and TNR correspond with implementations with fewer bugs than those with a low rate of just one or the other. More notably, our study suggests that writing tests that produce high rates of false negatives or false positives are associated with implementations with a comparable number of bugs.

These relationships between errors in positive verification and in fault identification were found both when evaluating the quality of an entire programming assignment or an individual function. Likewise, a within-subject comparison revealed that most students wrote tests that performed differently—with a mix of test outcomes that discriminated well, produced false positives, or produced false negatives—across different functions. Accordingly, our results suggest that the association between positive verification and implementations with fewer bugs cannot just be attributed to differences between individual students.

Consequently, our study concludes that positive verification should be considered as a factor when assessing the quality of students' unit tests. Although there is not yet a consensus on an effective metric for evaluating students' tests, we are confident that such a metric should consider positive verification in addition to (the more widely adopted) fault identification. Furthermore, our findings suggest that both factors are better predictors of implementation correctness than code coverage, despite its popularity.

While our study revealed an association between positive verification and implementation quality, it did not explore cost of testing. Our results suggest that poor positive verification or poor fault identification may have similar impacts on the number of bugs in the implementation. However, it is possible that testing errors that result in false negatives may be quicker to recognize and correct than when false positives fail to identify existing bugs. Therefore, further study on the cost of either type of error is warranted.

The conclusions of this study should be considered within the confines of its design. When assessing students' tests on the initial assignment, the corpus of unacceptable implementations included artificial bugs. While similar phenomena were observed between those tests and others that only included naturally occurring bugs, the artificial bugs may be a threat to validity. Similarly, the study only investigated testing behaviors of novice testers on two relatively small programs and different outcomes may be expected with more experienced testers and/or with larger projects.

Finally, to evaluate test results for individual functions, this paper introduces the *all-function-pairs* approach that evaluates students' tests against other students' implementations of individual functions, rather than only considering programs as a whole. The technique may also help improve assessment and automated feedback for students as they learn to use unit tests to verify their own function implementations.

12. References

[1] ACM. "Computer Science 2013: Curriculum Guidelines for Undergraduate Programs in Computer Science." Retrieved September 2018, from http://www.acm.org/education/curricula-recommendations.

[2] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. 2010. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM, New York, NY, USA, 153-160. DOI: https://doi.org/10.1145/1869542.1869567

[3] Kent Beck. 1999. "Embracing change with extreme programming." Computer 32(10): 70-77.

[4] Kent Beck. 2003. "Test-driven development by Example," Addison Wesley.

[5] David Bowes, Tracy Hall, Jean Petrić, Thomas Shippey, and Burak Turhan. 2017. How good are my tests?. In Proceedings of the 8th Workshop on Emerging Trends in Software Metrics. IEEE Press, Piscataway, NJ, USA, 9-14. DOI: https://doi.org/10.1109/WETSoM.2017..2

[6] 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. ACM, New York, NY, USA, 488-493. DOI: http://dx.doi.org/10.1145/2676723.2677247

[7] Kevin Buffardi and Stephen H. Edwards. 2014. Responses to adaptive feedback for software testing. In Proceedings of the 2014 conference on Innovation & technology in computer science education. ACM, New York, NY, USA, 165-170. DOI: http://dx.doi.org/10.1145/2591708.2591756

[8] Kevin Buffardi and Stephen H. Edwards. 2015. Reconsidering Automated Feedback: A Test-Driven Approach. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education. ACM, New York, NY, USA, 416-420. DOI: http://dx.doi.org/10.1145/2676723.2677313

[9] Stephen H. Edwards. 2004. Using software testing to move students from trial-and-error to reflection-in-action. SIGCSE Bull. 36, 1. March 2004, 26-30. DOI: http://dx.doi.org/10.1145/1028174.971312

[10] Stephen H. Edwards and Zalia Shams. 2014. Do student programmers all tend to write the same software tests? In Proceedings of the 2014 conference on Innovation & technology in computer science education. ACM, New York, NY, USA, 171-176. DOI: http://dx.doi.org/10.1145/2591708.2591757

[11] Stephen H. Edwards and Zalia Shams. 2014. Comparing test quality measures for assessing student-written tests. In Companion Proceedings of the 36th International Conference on Software Engineering. ACM, New York, NY, USA, 354-363. DOI: http://dx.doi.org/10.1145/2591062.2591164

[12] 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. ACM, New York, NY, USA, 221-226. DOI: http://dx.doi.org/10.1145/2157136.2157202

[13] Sebastian Elbaum, Suzette Person, Jon Dokulil, and Matt Jorde. 2007. Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable. In Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, 688-697. DOI: http://dx.doi.org/10.1109/ICSE.2007.23

[14] M. H. Goldwasser, 2002. A gimmick to integrate software testing throughout the curriculum. In Proceedings of the 33rd ACM Technical Symposium on Computer Science Education

[15] Google. "GoogleTest" https://github.com/google/ Retrieved September, 2018.

[16] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective Regression Test Case Selection: A Systematic Literature Review. ACM Comput. Surv. 50, 2, Article 29. May 2017, 32 pages. DOI: https://doi.org/10.1145/3057269

[17] Gerard G. Meszaros. 2010. XUnit test patterns and smells: improving the ROI of test code. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM, New York, NY, USA, 299-300. DOI: https://doi.org/10.1145/1869542.1869622

[18] Jerzy Neyman and Egon S. Pearson. 1967. "On the Use and Interpretation of Certain Test Criteria for Purposes of Statistical Inference, Part I". Joint Statistical Papers. Cambridge University Press.

[19] Rebecca Smith, Terry Tang, Joe Warren, and Scott Rixner. 2017. An Automated System for Interactively Learning Software Testing. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). ACM, New York, NY, USA, 98-103. DOI: https://doi.org/10.1145/3059009.3059022