# Bug Hide-and-Seek: An Educational Game for Investigating Verification Accuracy in Software Tests

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

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

*Abstract*—**This Innovative Practice Full Paper describes a pedagogical technique for introducing unit testing within software engineering courses. The Bug Hide-and-Seek educational game reinforces testing principles by requiring students to develop some correct solutions as well as some other solutions that intentionally contain bugs. While developing the correct and buggy solutions, students also write corresponding tests that should identify whether each solution contains bugs or exhibits acceptable behavior. Consequently, the first goal of the game is to hide a clever bug that will trick other students' tests into passing the implementation, despite the hidden bug. The second goal is to write thorough tests that can accurately differentiate correct from incorrect software behavior.**

**We introduce the motivation, pedagogy, and preliminary analysis of two variations of the Bug Hide-and-Seek game, while comparing their tradeoffs. The between-subject variation considers each student's test suite in its entirety. The within-subject variation requires more sophisticated analysis, but considers each individual function along with its corresponding tests, which provides more granular insight and specific feedback to students. We conducted a pilot study of both variations over two semesters of students (n=87) playing the Bug Hide-and-Seek game. We found that students' test True Positive Rate and True Negative Rate at verifying implementations are both significant predictors of a lack of bugs in their own solution.**

***Keywords—unit testing, software testing, test accuracy, false positives, false negatives, educational games, computer science education, all-function pairs analysis, true positive rate, true negative rate***

## I. INTRODUCTION

While testing is only one of several components of the Software Development Lifecycle, studies have found that software testing accounts for about half of the cost of development [1]. Since Agile development's boom in popularity, an emphasis on developing and delivering software in short, iterative cycles depends on testing throughout the process to verify the software's expected behavior. Accordingly, testing practices have adapted to enable continuous testing.

Beck [2] introduced Test-Driven Development (TDD), an approach compatible with Agile development. TDD involves writing a test before writing its corresponding implementating and then using that test to guage whether or not the implementation has met the expected behavior. TDD leverages unit testing, which focuses on testing individual functions rather than the system as a whole. By testing individual functions, unit tests help developers verify that each small part of their code behaves as intended while helping to localize the source of bugs when they fail. Whether or not developers adopt the TDD process strictly, unit testing is often leveraged for its concentration on verifying individual functions. Unit testing may also help localize and fix bugs to instill confidence in functions before testing integration between different parts of a system.

Despite the prominence and importance of testing in contemporary software development, there is a substantial inbalance between the attention given to testing in computer science eductation when compared to other tasks such as design and implementation. However, Association for Computing Machinery acknowledged the need to improve software testing in computer science education. The most recent curricula [3] recommends integrating software testing in courses that teach Software Development Fundamentals and Software Engineering. In addition, they specifically identified unit testing as a skill to consider teaching throughout the curriculum, even as early as in introductory classes. Correspondingly, this paper introduces the Bug Hide-and-Seek game for introducing students to the purpose of unit tests by differentiating acceptable and unacceptable implementations.

## II. RELATED WORK

Early adopters of unit testing curricular additions have begun exploring how to teach and assess students. With the availability of automated coverage analysis of tests, code coverage is a convenient choice for assessing the quality of students' tests. For example, instructors require students to write both the implementation and tests for a programming course assignment and a portion of the grade is based on the percentage of the implementation covered. However, using coverage to measure test quality may be misguided.

Coverage in its most common form refers to the percentage of statements in code that have been run by the tests. Identifying lines of code that have been overlooked may help developers identify program conditions that they have may overlooked or unreachable code. However, coverage does not consider the behavior or outcome of the tests. Likewise, coverage cannot help identify a bugs resulting from missing code, such as a conditional operator that is necessary but was not implemented. Consequently, coverage does not assess tests for their ability to discover bugs and verify acceptable implementations.

Perhaps even more problematically, using coverage as a metric for assessment may shift students' attention away from the goal of testing: distinguishing between acceptable and unacceptable implementations. Accordingly, when students were taught to follow TDD and assessed with test coverage, they demonstrated poor adherence to TDD and tended to delay testing until late in development and focused on achieving coverage rather than impartially verifying their implementation [4].

Nevertheless, some educational tools began to emerge in support of learning unit testing skills. Ante Up provides scaffolding by restricting students from beginning their implementation until their corresponding tests have been written and automatically validated [5]. Smith, Tang, Warren, and Rixner developed an automated tool that runs Python programs against a corpus of known buggy solutions to that same problem and measures how well students' unit tests fail each buggy implementation [6]. Unlike depending on coverage, examining tests' capability to fail buggy implementations directly addresses the purpose of tests. However, only checking for tests' performance against buggy solutions does not evaluate the goal of testing in its entirety.

Testing is meant to verify whether or not an implementation meets its expected behaviors. In other words, for test outcomes to be effective, they need to reliably reject (by failing) any implementation that does not exhibit the expected behavior under any circumstance *and* reliably accept (by passing) any implementation that always behaves as expected. Either only rejecting the bad or only accepting the good implementations represents an incomplete evaluation of test effectiveness.

Along this notion, tests can be categorized according to how accurately their outcomes diagnose acceptable and unacceptable implementations. An ideal set of tests should be able to correctly verify any acceptable implementation while also failing any unacceptable implementation. These outcomes represent *true positive* and *true negative* diagnoses, respectively. However, if a test mistakenly fails an implementation that actually has acceptable behavior, the test has produced a *false negative*. Similarly, if a test passes an implementation that contains a bug, the outcome represents a *false positive*. These testing errors can be likened to Type I and Type II errors in hypothesis testing [7]. However, unlike diagnosis medical tests where a true positive represents identifying a malady, our categorization describes true positives as the verification of acceptable implementations. Table I illustrates the categorization of test accuracy in a confusion matrix.

If true positives and true negatives represent outcomes from effective tests, what is the meaning and effect of false positives and false negatives? A false positive reflects a test's inability to identify that there is a bug—in at least one situation, the implementation misbehaves. The consequence of false positives is that developers will have a false sense of confidence in the code's correctness. The ineffectiveness of the test at detecting the bug may result in buggy software that crashes or misbehaves for the customer. Even if the bug is discovered before the software is delivered to the customer, when unit tests are ineffective in their task, it becomes more difficult (and more costly) to localize the cause of the bug when it is not detected until multiple parts of the software are integrated together.

On the other hand, false negative outcomes occur when tests fail an acceptable implementation. Failing tests alert developers that the code needs to be fixed. In the best-case scenario of when a false negative occurs, the developer spends a little time to correct the inaccurate test. However, since tests are meant to express the expectations of the software, a false positive may misguide developers to "fix" an acceptable implementation by changing its behavior to satisfy the (inaccurate) test. The consequence again is increased cost and the risk of delivering software that does not behave as expected. Accordingly, as students learn to test software, they should develop the ability to maximize true positives and true negatives while eliminating false positives and false negatives.

Learning unit testing is a practical software engineering tool but becoming an effective tester requires metacognitive development. An updated Bloom's Taxonomy [8] outlines six levels of cognition: remembering, understanding, applying, analyzing, evaluating, and creating. The first three forms of knowledge are considered Lower Order Thinking Skills (LOTS) and the last three are Higher Order Thinking Skills (HOTS). LOTS demonstrate the fundamentals of a skill while HOTS are essential to metacognition and being able to leverage that skill to problem solve new situations.

Students should adopt each level of cognition to demonstrate mastery of unit testing. For example, the LOTS include: recalling the purpose of a function (remembering); expressing how the function should behave by "constructing a cause-and-effect model" [8] (understanding); apply the procedures to writing a unit test by setting the state/scenario, invoking the function, and asserting its effect to the expected behavior (applying). Meanwhile, HOTS include: testing to distinguish between acceptable and unacceptable behavior (analyzing); judging if tests are sufficient to be confident in their outcomes and checking the tests against different implementations (evaluating); and hypothesizing what errors may exist in possible implementations and planning strategies to test effectively (creating).

The HOTS involved may also demonstrate forms of scientific thinking and "Software testing promotes the hypothesis-forming and experimental validation that are central to […] reflection in action" [9]. Reflection and empirical

TABLE I. TEST OUTCOME VERIFICATION CLASSIFICATION

| | Test Outcome | |
|---|---|---|
| **Implementation** | ***Pass*** | ***Fail*** |
| ***Acceptable*** | True Positive | False Negative |
| ***Unacceptable*** | False Positive | True Negative |

evaluation demonstrate critical thinking and should result in improved problem-solving abilities over weaker strategies such as trial-and-error.

A study of good practices in unit testing [10] revealed that it is essential to write both "happy" and "sad" tests—including cases that verify mainstream, expected situations as well as extraordinary cases that could cause bugs. However, studies of students' tests [1, 9] and eLearning tools for learning to test [6, 11] have concentrated on their ability to create "sad" tests to identify bugs.

Goldwasser pioneered the all-pairs approach by running each student's tests against every other student's code to evaluate how well tests can identify bugs in programming assignments [12]. Goldwasser's 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%.

More recently, an independent replication of the all-pairs approach with more flexible support for unit test frameworks and integration with an online autograder [13] found that a majority of students passed at least 90% of the test case corpus but only five of 101 students managed to pass all test cases. A follow-up study found that students earned high coverage scores but tended to produce only happy tests [14]. This behavior may be expected when students' primary metric of feedback and assessment for their tests is test coverage, which acknowledges code tested but not the tests' ability to distinguish good solutions from those with bugs.

On the other hand, mutation testing is a technique that involves automatically generating a corpus of "mutant" bugs by making small alterations to the existing implementation. Tests are then evaluated by how well they "kill mutants" by failing them. However, a study by Edwards and Shams [15] found that mutation testing was no more effective at predicting tests' ability to find real bugs than all-pairs analysis. Accordingly, our approach to teaching unit testing and evaluating tests' accuracy represents an extension of all-pairs analysis.

## III. Bug Hide-and-Seek

The purpose of the Bug Hide-and-Seek game is to generate a corpus of both correct and incorrect implementations to a programming problem while also engaging students in a game that conveys the goal of testing: to distinguish between the acceptable and unacceptable solutions. In sequential semesters, we implemented two variations of the Bug Hide-and-Seek game, as described in the following sections.

### *A. Between-Subject Design*

During the first semester, we implemented the Bug Hide-and-Seek game, where instructions for the game differed between students. Before beginning the game, students are randomly assigned to one of two roles: *Hider* or *Seeker*. The instructor tells the seekers that their code should be reliable and robust enough to not only pass their own tests, but also to pass all the tests developed by the entire class. Meanwhile, Hiders are instructed to intentionally include behavioral flaws in their code. More precisely, their goal in the game is to write an implementation that behaves correctly in some cases but misbehaves in others. In other words, they want to hide bugs that are difficult to discover.

In the game, each student also submits a suite of unit tests along with their solutions. The primary goal of the game is for students to create tests that accurately differentiate between the correct and incorrect implementations. A secondary goal for the Seekers is to write a solution so robust that they pass as many of the class' tests as they can, while the Hiders try to make a buggy implementation that tricks the most test suites into passing it (without identifying its bug).

All students are assigned to be either a Hider or a Seeker and they are provided instructions to create and submit a suite of unit tests along with their implementation. To ensure that students' test cases are compatible with each other's implementations, they are 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.

To be effective in either role, the students would need to use HOTS to strategize their approach. Creating a robust suite of tests that can distinguish even implementations with the most difficult-to-find bugs from acceptable versions involves analysis of the problem and evaluation of what cases their tests may miss. Meanwhile, creating an effective hidden bug involves considering what special cases other testers may not consider for their tests. Accordingly, success in either role of the game relies on effective reflection and HOTS as well as some mastery of software testing.

### *B. Within-Subject Design*

In the subsequent semester, we designed a variation of the game where every student is assigned one specific function in which to hide a bug, while attempting to correctly implement all other functions in the assignment. Overall, the premise of the game remains the same: students try to create a difficult-to-detect bug while also creating a suite of tests that will be able to differentiate between all acceptable and unacceptable solutions. However, in a within-subject design, students are not separated between Hider and Seeker roles. Instead, every student is randomly assigned to one particular function where they assume the Hider role to hide a bug in an almost-correct implementation. For every other function in the problem assigned, each student is instructed to attempt to write an implementation with no flaws.

In this implementation, every student is charged with the task to hide a bug. However, since students are assigned Hider functions randomly, the bugs should be distributed throughout different functions when considering the corpus of all student submissions. Accordingly, each student is tasked to make the distinction between good and bad implementations for each function rather than judging each program implementation as a whole.

### *C. Programming Challenges*

The Bug Hide-and-Seek game is designed so that it can be applied to any assignment with a problem that involves both programming a solution as well as corresponding tests. For our study, the initial programming assignment was to implement a class that represents a data model for a Tic-Tac-Toe board with applicable functions: *placePiece* for placing the current player's piece onto a specified row and column, while assurring that the coordinates provided are within the board's bounds and does not already contain a game piece; *getPiece* inspects the coordinates provided and determines which game piece is there, if there are no pieces, or if the location is invalid; *toggleTurn* to switch between which player's turn it is; and *gameState* which determines the game state (e.g. player who has won, an ongoing game, or a completed tie game).

The Tic-Tac-Toe game has a finite number of states so theoretically, one could exhaustively consider each possible state. However, the problem also involves multiple functions, several of which contain parameters and instructions for handling cases beyond those states (such as trying to place a piece at invalid coordinates), it would be impractical to try to test every possible parameter permutation for each function for each possible Tic-Tac-Toe board state. Even provided enough time to test so exhaustively, doing so would not be cost-effective in software development of an application that is not safety-critical.

## IV. Evaluation

To assess students' performance in the Bug Hide-and-Seek educational game as well as investigate their strengths and shortcomings in unit testing, we evaluated the verification accuracy of students' tests. Although assigning students to Hider and Seeker roles theoretically results in both incorrect and correct solutions (according to their respective roles), students may not achieve those goals. Therefore, it is necessary to characterize each solution based on its performance against the instructor's reference tests. However, either design to the game requires different analysis.

### A. Between-Subject Analysis

In the between-subject design, each student submission needs to be characterized as acceptable (positive verification) or unacceptable (negative verification). Using an automated script, the reference tests are run against each student's implementation and records the solution as positive if it passes all tests or negative if it fails any tests, times out (usually indicating an infinite loop) or exits with a fault.

In similar instrumentation to all-pairs analysis [12, 13], each student's test suite ($T_i$) runs against each other student's solution ($F_j$) and records whether it passes or fails each implementation. Fig. 1 illustrates the automation of the between-subject all-pairs approach.

Once the outcome of each pair is recorded, the results of students' test suites are compared to those of the reference tests. The reference tests have already characterized each implementation as positive or negative and a perfect test suite should pass all positive solutions and fail all negative solutions. To assess students' ability to differentiate between positive and negative implementations, we calculated their test suites' True Positive and True Negative rates. The True Positive Rate (TPR) is the percentage of positive solutions passed. The True Negative Rate (TNR) is the percentage of negative implementations failed.

The automation of all-pairs analysis is relatively uncomplicated as long as the class interface for the assignment is held constant. While the amount of time for all-pairs analysis grows exponentially with the number of students, an all-at-once analysis of students' work after their submission are complete is reasonable for traditional class sizes (but perhaps inefficient for giving prompt feedback to students in class enrollments in the hundreds or larger).

However, the greatest limitation of all-pairs analysis is that it only considers implementations entirely acceptable or entirely unacceptable. This perspective runs counter to the intent of unit tests—to verify each function individually. Likewise, to assess tests' ability to verify software that behaves according to expectations, it requires at least some students to produce solutions that the reference tests find no flaws in. However, these cases may be rare. To address both limitations, we evolved the all-pairs approach for function-level analysis in the within-subject implementation of the game.

### B. Within-Subject Analysis

The first task in analyzing the within-subject implementation is to determine the correctness of each function of each student's implementation. Since the instructor reference tests already followed suggested practices for proper unit testing, each of the unit tests already targeted individual functions. Consequently, instead of running 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 unit tests in the subset of those that target a given function, that function implementation is positive.

For example, a subset ($T_{fut}$) of the entire reference suite tests the *getWinner* function (the function-under-test for the current implementation, $FUT_j$); passing each unit test in that

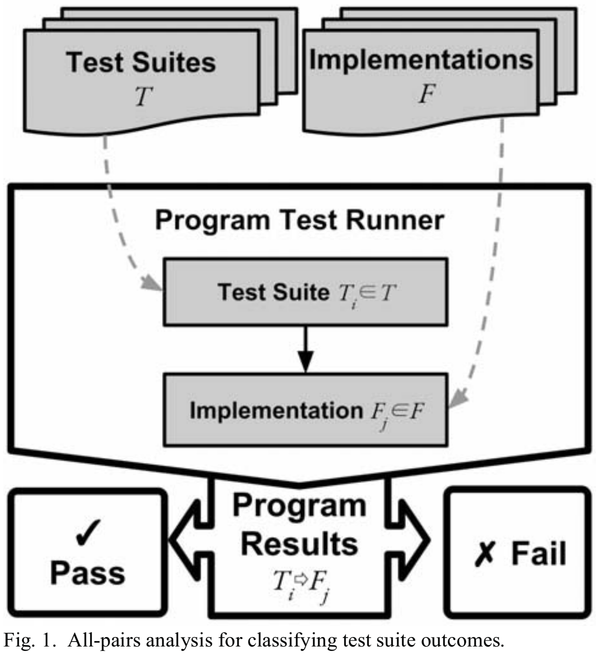

Fig. 1. All-pairs analysis for classifying test suite outcomes.

subset indicates *getWinner* is positive. Otherwise, any failures indicate a negative function implementation. Fig. 2 illustrates the automated process of analyzing tests against each function for each student. Most popular test runners for xUnit frameworks include options for specifying individual tests to run, including *GoogleTest* (C++), *JUnit* (Java), and *unittest* (Python). Therefore, this approach can be instrumented for most popular programming languages.

However, it can be more difficult to identify which function a student's unit test intends to verify if they do not adhere to best practices of testing. There are different plausible solutions to this challenge. One may choose to manually inspect student tests to identify their functions-under-test. Otherwise, using static analysis to identify which function's output is used in the test's assertions may be a sufficient proxy. In an effort to localize bugs in students' programs, previous research proposed an automated approach that may also identify the functional scope of tests by analyzing their coverage [16]. For this study, we manually identified functions-under-test.

After identifying the function-under-test for each unit test, the subset of a student's unit tests that share the same function ($T_{fut}$) can be run against the corpus of all student implementations, one-at-a-time. If an implementation fails any of the unit tests within that subset, the student has produced a true negative (if the implementation is unacceptable) or a false negative (if the implementation is acceptable). However, passing each test in the subset—or in the case of a null subset—the student has produced a true positive (if the implementation is acceptable) or false positive (if the implementation is unacceptable).

We distinguish this approach from Goldwasser's and Edwards' previous experiments by referring to it as *all-function-pairs* analysis. All-function-pairs should gain higher probability of yielding positive implementations since they only need to behave correctly for an individual function rather than for the entire program. Consequently, the approach allows for more granular analysis of both implementations and test suites. Likewise, it makes a variation of the Bug Hide-and-Seek game possible, where each student may participate as both a Hider and Seeker.

### C. *Analyzing Test Accuracy*

As introduced in the sections on between-subject and within-subject analysis, both approaches enable assessment of students' tests by their ability to verify acceptable implementations and reject buggy implementations. These qualities are indicated by their true positive (TPR) and true negative rates (TNR). After an implementation is categorized as positive (acceptable) or negative (unacceptable), students' tests are measured by how well they can distinguish the corpus of positive from the corpus of negative implementations. Accordingly, tests are then classified as accurately passing acceptable (true positive) and failing unacceptable (true negative) or inaccurately passing unacceptable (false positive) and failing acceptable (false negative) implementations. Fig. 3 illustrates verification assessment, where the *test runner* can either use *all-pairs analysis* to evaluate program-level outcomes or *all-function pairs analysis* to evaluate function-level outcomes.

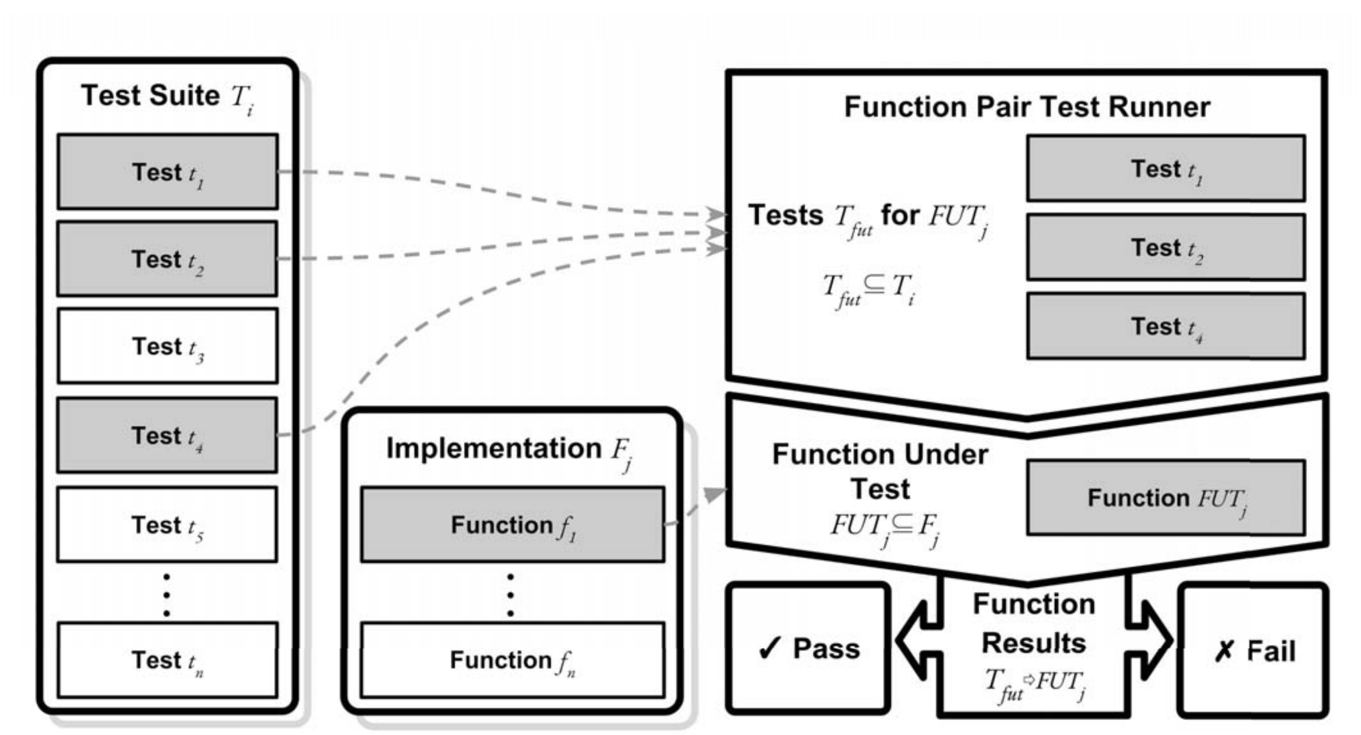

Fig. 2. All-function pairs analysis of test outcomes against each function from each student.

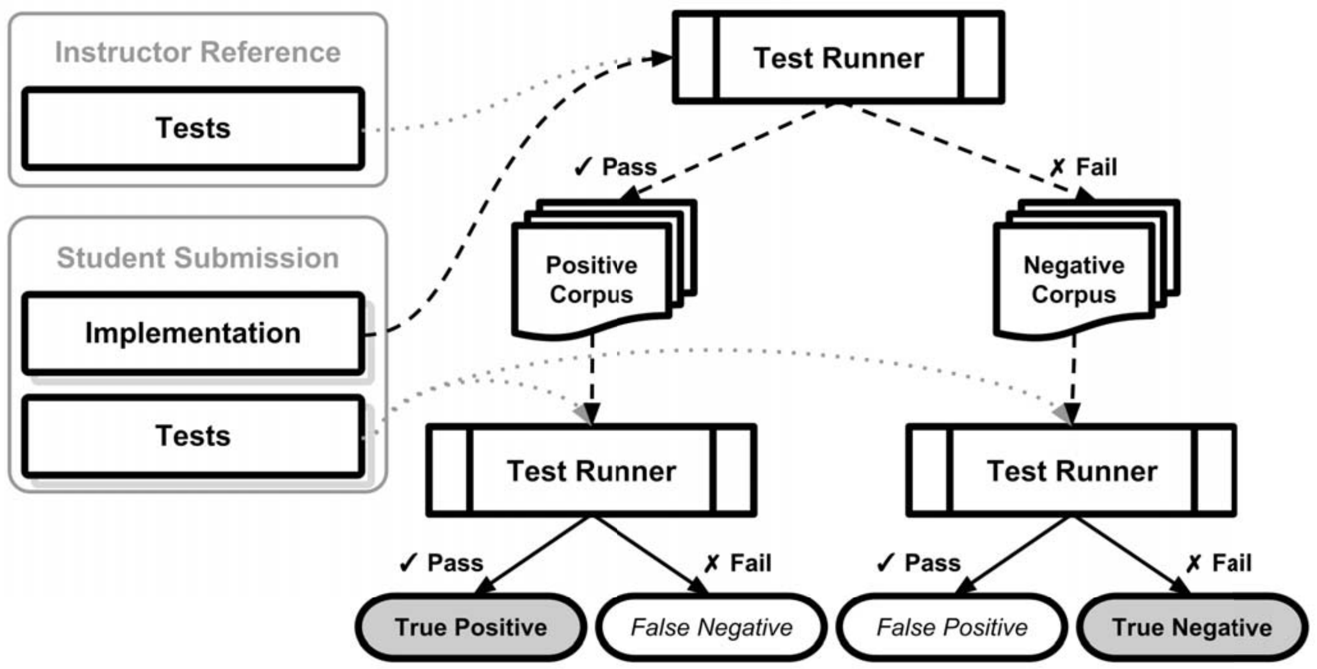

Fig 3. Verification accuracy classification for either all-pairs or all-function pairs analysis.

## V. Observations

First and foremost, we evaluated the practicality of implementing the two variations of the Bug Hide-and-Seek game along with corresponding test assessment. In the first semester, students in two Software Engineering classes were introduced to the game during a course module on testing. The classes were upper-division undergraduate and Masters-level graduate classes. Students had previous experience with multiple programming courses but limited or no exposure to unit testing. Before the game, students were taught how to set up and run the unit testing framework on a project and introduced to the syntax and semantics of unit tests. Students were randomly assigned game roles by the course management system; random assignment was independent for each course so that undergraduate (n=40) and graduate (n=8) classes were each split evenly between the Hider and Seeker roles.

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. Without any positive solutions, we would have not been able to assess tests for their ability to produce true positives. Consequently, we decided to extend the assignment to a second lecture and instruct all students to assume the Seeker role to attempt a correct solution (while continuing to improve their tests as well), which yielded five positive solutions.

To assess each student's ability to differentiate between positive and negative implementations, we calculated their test suites' True Positive and True Negative rates. 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 game assignment, students averaged relatively low on TPR (M=0.61, sd=0.42) and TNR (M=0.72, sd=0.36) but both rates also indicate large variance. Similarly, they achieved relatively low condition/decision coverage (M=0.57, sd=0.33) within the limited time allowed. Condition/decision coverage is a more rigorous metric than simple statement coverage since the former measures whether each condition within each boolean decision has been evaluated as both true and as false.

We found that while some students' test suites perform relatively well on both TPR and TNR, there are also several students who perform well on one but poorly on the other. There were no students who scored below 75% on both TPR and TNR. This phenomenon reinforces the notion that students' tests should not only be assessed for their ability to find bugs (with a high TNR) but also by their ability to verify good behavior (with a high TPR). However, we also investigated how these different testing outcomes might impact the quality of their 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, TNR, and condition/decision coverage as potential predictors. After 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 coverage factor and found a significant regression equation (F(2,27)=10.92, p<.001, $R^2$=0.45) where students' predicted implementation correctness was equal to 0.06 + 0.25*TNR + 0.18*TPR. Students' correctness increased 25 percentage points for failing all negative solutions and 18 points for passing all positive solutions; TNR (p<.001) and TPR (p<.001) were both independently significant predictors.

These findings complement a previous study's conclusion that all-pairs analysis is better than coverage at predicting tests ability to find bugs [15] by indicating that all-pairs assessment is also better at predicting an association with developing an implementation with fewer bugs. Nevertheless, our experience with the between-subject design also exhibited the shortcomings of its dependence on all-pairs analysis. Collating a positive corpus of entire program implementations depends on some students solving the programming challenge with no bugs. In our first semester, no students achieved as much on their first attempt of the game so we had to extend the assignment and also instruct Hiders to abandon their roles and try to correct all their bugs so that we could improve the likelihood of yielding positive results.

Even after adjusting the design to improve our odds, implementations without any bugs were rare. Consequently, a rationale for adopting the within-subjects design of the game is that—as long as there are more than two functions required in the challenge—there will be more students attempting an implementation *of any given function* than in the between-subjects design. Moreover, the within-subjects design only requires a function by itself to behave acceptably to be a positive function. Consequently, the odds of producing multiple positive solutions for each function should improve.

As expected, performing *all-function-pairs* analysis on the within-subject design in second semester of the study found that even with a marginally smaller class enrollment (n=39, compared to 48 the previous semester), the students successfully yielded multiple positive and multiple negative implementations for each of the four Tic-Tac-Toe functions. Consequently, we were able to complete the all-function-pairs analysis and test assessment without changing the rules of the game nor extending the assignment.

In analyzing students' tests at the function-level, we found that their tests had a moderately low True Positive Rate (TPR, M=0.75, sd=0.39) and True Negative Rate (TNR, M=0.56, sd=0.39) at testing individual functions. We examined 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, but some students had high TPR with low TNR while others had high TNR paired with low TPR.

When examining which students produced each of these different function testing outcomes, we found that 62.5% of students wrote tests for at least one function that resulted in a combination of TPR and TNR above chance. Meanwhile, 56.25% of students produced a combination of high TPR and low TNR (which indicates many false positives) on at least function. Just under half of the students (46.88%) erroneously tested at least one function with a combination of high TNR and low TPR (which indicates many false negatives). No students produced tests for any functions that had below chance for both TPR and TNR.

The observed phenomena appear to suggest that the variation in verification accuracy does not just reflect the differences between individual students. In other words, there is no evidence that the performance of tests with good accuracy (high TPR and TNR) is just reflecting work of strong students and the tests with poor accuracy (with either false positives or false negatives) only come from weaker students. Instead, there appears to be variation for most students in their test accuracy on different functions.

We used k-means clustering to identify students with similar outcomes and categorized three clusters: *True Discriminators* were those with high TNR and TPR; *False Positives* were those with high TPR and low TNR; and *False Negatives* were those with high TNR and low TPR. We manually reviewed the test suites in the separate clusters to search for trends. We found that while the false negative tests tended to have fewer test functions, their test functions also tended to be longer and more complex. Test functions for unit tests should be brief in most circumstances to concentrate on individual test cases [10].

To the contrary, we found that tests that produce high rates of false negatives often wrote test functions that were not focused on concise test cases. Instead, they often had multiple assertions and even had code that included loops and conditional controls. Consequently, just a small mistake in complex test functions would inadvertently fail solutions that should have passed. On the other hand, test suites that yielded many false positives tended to have appropriately-sized test cases; however, they did not specify enough tests to thoroughly verify function implementations. Those errors may be able to be reconciled by teaching students to improve on their "sad" tests.

The association found between TNR and TPR of tests and their respective implementations does not necessarily indicate a causal relationship. However, we found no evidence that would suggest that the association just describes a correlation based on differences between individual students. Our findings warrant further investigation to explain possible relationships between verification accuracy (at both the function and program-level) and implementation correctness in a controlled research experiment.

Anecdotally, we observed students demonstrating high-engagement in the active learning game. They were presumably motivated to create a clever hidden bug that other students could not detect. Both designs of the Bug Hide-and-Seek game emphasize the goals of testing to distinguish between good and bad implementations. Rather than aiming for high coverage scores (that are not necessarily associated with fewer bugs), students aspire to write discriminating tests. Accordingly, in future work, we plan to compare the Bug Hide-and-Seek approach to learning how to test using traditional tools, techniques, and assessments.

Meanwhile, our introduction of all-function pairs analysis builds upon a technique that been shown to assess student tests more effectively than either test coverage or mutation testing. All-function pairs analysis improves upon the ability to assess the accuracy of tests for each function by itself, which holds true to the purpose of unit testing. While most of the all-pairs analysis was automated, we manually identified the subsets for each test suite according to their respective function-under-test. In future work, we hope to fully automate this task so that all-function pairs analysis may be available for automated feedback.

## VI. Conclusions

The Bug Hide-and-Seek educational game introduced in this paper provides a unique approach to exposing students to unit testing. Since the game compels students to consider that their unit tests need to differentiate between all acceptable and unacceptable implementations, it should keep them focused on doing so as the primary goal of testing. While finding code coverage is a popular mechanism for examining tests—both in industry and academia—it does not assess whether or not tests are effective at verifying good implementations and at finding bugs. Our study confirmed previous findings [15] that coverage is an ineffective assessment of tests since it is not a significant predictor of a lack of faults in an implementation.

However, that is not to say that coverage is useless. In fact, during our study, we taught students how to use a coverage tool to help them identify test cases they may have missed. The key difference is that in this usage, a coverage tool is used to troubleshoot a problem, akin to using a debugger. Meanwhile, coverage is *not* portrayed as the goal of testing.

Nevertheless, this study did not directly compare the Bug Hide-and-Seek game to other learning techniques. While this paper outlines some benefits of the game's approach, we do not have evidence to suggest it results in more effective learning. In future work, we plan to conduct an experiment that compares the Bug Hide-and-Seek game to more traditional approaches to introducing unit testing. In particular, comparisons should be drawn between Bug Hide-and-Seek and methods that do not involve students purposely creating their own buggy solutions.

Meanwhile, this paper also introduces all-function-pairs analysis, which evaluates tests on their passage or failure of individual functions. By leveraging all-function-pairs analysis, assessment is more granular by identifying specific functions that students have tested either accurately or inaccurately. By doing so, all-function-pairs analysis also enables the within-subject variation of the Bug Hide-and-Seek game. This paper also suggests how all-function-pairs analysis may be fully automated in future work, which would empower its use in autograders and other learning tools.

Finally, the study identified two measurements of tests: True Positive Rate (TPR) and True Negative Rate (TNR). TPR indicates how often tests pass acceptable solutions while TNR indicates the rate of failing faulty implementations. High rates of both TPR and TNR indicate an effective test suite. Meanwhile, we found a low rate of either TPR or TNR is associated with also having more bugs in function implementations. Future work should explore TPR and TNR as measurements for grading students' tests on programming assignments.

## References

[1] 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. DOI: https://doi.org/10.1145/3057269

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

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

[4] 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 (ITiCSE '14). ACM, New York, NY, USA, 165-170. DOI: http://dx.doi.org/10.1145/2591708.2591756

[5] 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 (SIGCSE '15). ACM, New York, NY, USA, 488-493. DOI: http://dx.doi.org/10.1145/2676723.2677247

[6] Rebecca Smith, Terry Tang, Joe Warren, and Scott Rixner. 2017. An Automated System for Interactively Learning Software Testing. ITiCSE '17. ACM, New York, NY, USA, 98-103. DOI: https://doi.org/10.1145/3059009.3059022

[7] 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.

[8] Anderson, L. W., Krathwohl, D.R, Airasian, P.W., Cruikshank, K.A., Mayer, R.E., Pintrich, P.R., Raths, J. & Wittrock, W.C. (2001). A taxonomy for learning and teaching and assessing: A revision of Bloom's taxonomy of educational objectives, Addison Wesley Longman.

[9] Stephen H. Edwards. 2004. Using software testing to move students from trial-and-error to reflection-in-action. SIGCSE Bull. 36, 1, 26-30.

[10] 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 (WETSoM '17). IEEE Press, Piscataway, NJ, USA, 9-14. DOI: https://doi.org/10.1109/WETSoM.2017..2

[11] 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 (ICSE '07). IEEE Computer Society, Washington, DC, USA, 688-697. DOI=http://dx.doi.org/10.1109/ICSE.2007.23

[12] M. H. Goldwasser, 2002. A gimmick to integrate software testing throughout the curriculum. SIGCSE '02.

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

[14] Stephen H. Edwards and Zalia Shams. 2014. *Do student programmers all tend to write the same software tests?*. ITiCSE '14. ACM, New York, NY, USA, 171-176. DOI: http://dx.doi.org/10.1145/2591708.2591757

[15] Stephen H. Edwards and Zalia Shams. 2014. Comparing test quality measures for assessing student-written tests. ICSE '14. ACM, New York, NY, USA, 354-363.

[16] Kevin Buffardi and Stephen H. Edwards. 2015. Reconsidering Automated Feedback: A Test-Driven Approach. SIGCSE '15. ACM, New York, NY, USA, 416-420. DOI: http://dx.doi.org/10.1145/2676723.2677313