Reconsidering Automated Feedback: A Test-Driven Approach

Kevin Buffardi
California State University, Chico
O'Connell Technology Center 215
Chico, California 95929-0410
+1 530-898-5617
kbuffardi@csuchico.edu

Stephen H. Edwards
Virginia Tech
2202 Kraft Drive
Blacksburg, VA 24060
+1 540-231-5723
edwards@cs.vt.edu

ABSTRACT
Writing meaningful software tests requires students to think critically about a problem and consider a variety of cases that might break the solution code. Consequently, to overcome bugs in their code, it would be beneficial for students to reflect over their work and write robust tests rather than relying on trial-and-error techniques. Automated grading systems provide students with prompt feedback on their programming assignments and may help them identify where their interpretation of requirements do not match the instructor's expectations.

However, when automated grading systems help students identify bugs in their code, the systems may inadvertently discourage students from thinking critically and testing thoroughly and instead encourage dependence on the instructor's tests. In this paper, we explain a framework for identifying whether a student has adequately tested a specific feature of their code that is failing an instructor's tests. Using an implementation of the framework, we analyzed an automated grading system's feedback for programming assignments and found that it often provided hints that may discourage reflective testing.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.5 [**Programming Techniques**]: Object oriented Programming; D.2.5 [**Software Engineering**]: Testing and Debugging.

Keywords
Software development process; Test-Driven Development (TDD); Test-first; Unit testing; Automated testing; Web-CAT; Instructional Technology.

1. INTRODUCTION
Current computer science curricula [1][2] emphasize the importance of students gaining proficiency in software testing. Consequently, programming assignments often require students to write unit tests for their programs. Meanwhile, automated grading systems such as Web-CAT [9] and Marmoset [13] have incorporated features to evaluate the quality of students' tests. In addition, these grading systems provide feedback to students to identify shortcomings in both solution correctness and thoroughness of testing. However, concerns arise about whether automated feedback systems empower students with effective software testing practices. In particular, multiple instructors have independently expressed worries that their students rely upon Web-CAT's automated testing feedback to supplant the need for thorough testing and critical thinking.

Since the point of testing is to verify that software behaves as expected, the practice of writing tests should involve reflecting over a variety of different test cases and confirming that each produces the expected outcome. In performing due diligence in testing, a student should consider not only the mainstream cases, but also which unusual cases may cause bugs, or unexpected outcomes. To the contrary, we have found that students write "happy path" tests—only those mainstream cases unlikely to cause unexpected bugs [11].

After students submit their work to Web-CAT, they receive feedback on the correctness of their solutions—as determined by a set of obscured reference tests provided by the instructor—as well as measurements of how much of their own code they have exercised with their own tests. In addition, failed reference test cases generate hints that instructors use to give students general direction in identifying what is malfunctioning without revealing the specific details of the reference test. Web-CAT's correctness score provides students with confirmation of how well their solution performs according to the project specifications. While the hints do not necessarily indicate exactly which test cases are not working, they also provide students with insight into which features they need to fix in their solution code. Both types of feedback may be well intended to help students make progress on their programming assignments; however, they also relieve at least some of the burden of software testing.

Ideally, students would test their own code thoroughly enough that without any feedback from the instructor, they would have strong confidence that their code behaves correctly by its performance against their tests alone. In non-academic settings, software developers do not usually have the benefit of an instructor's thorough checks to make sure their software is sound and robust. Instead, they would have to rely on their own estimates of the code quality to determine if it is ready to deploy. Nevertheless, it may be unrealistic to expect students to know how to test effectively without assistance.

SIGCSE '15, March 4–7, 2015, Kansas City, MO, USA

ACM 978-1-4503-29966-8/15/03…$15.00
http://dxx.doi.org/10.1145/2676723.2677313

Consequently, programming courses should find a compromise between alleviating the burden of quality assurance from the students and demanding professional-quality testing proficiency immediately. In this paper, we reconsider students' interaction with automated grading systems. We describe a model for scaffolding feedback in Web-CAT to encouraging students to reflect over their code and concentrate on writing effective software tests to reveal bugs instead of relying on external sources.

2. BACKGROUND

When students submit programming assignments to Web-CAT, test coverage estimates the quality of their tests. Traditionally, Web-CAT evaluates the percentage of methods, lines of code, and conditions within a student's solution code are executed by her unit tests. This combination of method, line, and condition coverage provides a general perspective of the breadth of code exercised by the student's tests. However, even 100% code coverage does not necessarily indicate that tests have thoroughly checked the program's behavior. For example, consider the following code segment:

```
if ( a && (b || c) )
{
   return true;
}
else
{
   return false;
}
```

Figure 1. Example of IF-ELSE Control with a Compound Condition

The above code can achieve full coverage in two test cases: one where the variables `a` `b` and `c` are true (which returns true) and one where they are all false (which returns false). With these two test cases, we have executed each line in the code and have executed both outcomes of the `if` condition (`true` and `false`). However, with these tests, we have not considered other cases with different combinations of values for the variables. While the compound condition has been both true and false, variations of the atomic decisions are not thoroughly tested. For example, if **`a`** and **`b`** are true but **`c`** is false, does the code's outcome (true) match the expected behavior of the program?

There are alternate approaches for estimating test quality beyond traditional coverage measurements. One study compared coverage to two other techniques—mutation testing and all-pairs testing—to determine how well they predict bugs in students' actual code [10]. Mutation testing involves making multiple versions of a student's source code with minor modifications—each called a "mutant"—and identifying whether the student's tests "kill" mutants by producing different outcomes from the test with its original code. Meanwhile, all-pairs testing involves running a student's tests against all other students' implementations of the same assignment and measures how many bugs it identifies. The study found that all-pairs testing was most effective at determining a test suite's ability to identify bugs while mutation testing was no better than coverage at doing so.

However, all-pairs testing has caveats that limit the practicality of using it in automated grading systems like Web-CAT. First, as the number of students in the class (and the number of tests they write) grows the computational expense of running tests against all other students' code increases exponentially. Perhaps more poignantly, Web-CAT allows students to submit their assignment multiple times throughout development so each student's code and all-pairs score will change over time (until submissions are no longer accepted). Consequently, the first students to submit their code would have less comprehensive all-pairs analysis and the results may change as others submit their work.

Modified Condition / Decision Coverage (MC/DC) is another approach for measuring comprehensiveness of software tests [8]. Unlike conventional coverage where a decision only requires an overall true and false evaluation, MC/DC requires showing that each *atomic* condition affects a "decision's outcome by varying just that condition while holding fixed all other possible conditions" [7]. As one form of MC/DC, *masking MC/DC* takes advantage of setting one operand of an operator so that the other operand cannot affect the value of the operator. Reconsidering the example in **Figure 1**, the operand **`b`** can be masked by holding **`c`** true so that it will evaluate as true regardless of **`b`**'s value; likewise, **`a`** can be masked by holding **`(b || c)`** false so that the expression will evaluate to false regardless of **`a`**'s value. Because masking MC/DC examines conditions with more detail, it subsumes Web-CAT's approach of measuring coverage for each method, line, and control decision.

In addition to challenges of measuring the thoroughness of students' tests, automated grading systems need to encourage students to test their code more thoroughly rather than depend on automated feedback to tell them whether their code works. Instead, it should provoke students to reflect, "What other test cases might possibly break my code?" However, our previous studies found that students' responses to Web-CAT's feedback concentrate predominantly on fixing errors in their code to improve their correctness score [3][4]. While students also aim to earn full credit for testing by obtaining 100% coverage, their approach does not reflect the mentality of trying to expose weaknesses in their code. Rather, their testing behaviors more closely demonstrate a desire to extend the least amount of work to obtain 100% coverage, a relatively superficial objective for testing.

Nevertheless, students' responses to Web-CAT's feedback is understandable since they are driven to achieve good grades (which 100% coverage allows), and the feedback already provides some insight into how acceptable their code is without having to test it sufficiently themselves. Our previous attempts with adaptive feedback with overt encouragement to test had no long-term impacts on their testing behaviors [3]. However, the adaptive feedback interventions supplemented other feedback that still gave students insight into the quality of their code, perhaps undermining the efforts to encourage testing. Consequently, we designed a model for scaffolding feedback that still provides students with help on improving their code, but with a strategic approach emphasizing testing.

3. METHOD

3.1 A Framework for Targeting

Previous approaches considered overall coverage of the student's entire programming project. Of course, students' projects usually include many methods and often even multiple classes. Consider a situation where a student submits a project that is not behaving

perfectly (according to the instructor's specifications) and has some (but not exhaustive) testing. Web-CAT detects some bugs in the student's code from some of the instructor's tests failing and gathers the hints associated with each failed test. For one feature of the project, the student thoroughly tested her code but the feature was not implemented precisely to the instructor's specifications: the text output in her implementation uses commas to separate values but the instructor expected line breaks instead. The problem with this feature is not one caused by a lack of testing, but rather by a misunderstanding of the requirements. It is unlikely that more testing will expose the problem and without some insight from the hints, it may be frustrating to overcome this relatively superficial bug.

However, in the same submission to Web-CAT, the student may have another feature that is incorrectly implemented but has not discovered the problem because she has not tested that feature as thoroughly. Unlike the previous example, revealing a hint to correct this bug would be inappropriate because the student should first try to reflect over what test cases she neglected in her test suite. Nevertheless, by only considering the coverage of the entire project, the automated grader cannot determine which feature has been thoroughly tested or which hints are appropriate to show.

Consequently, in order to choose appropriate guidance for students, automated feedback must first identify testing quality feature-by-feature. To do so, we developed a custom runner for JaCoCo [12]—an open source coverage tool for Java—that produced a coverage report for each instructor reference test independently. After identifying which reference tests fail against a student's submission, the system then reviews the report for each test and identifies which method(s) within which class(es) of the student's code the test executes. As a result, the system can associate failed tests with particular features within the student's submission. While these features cannot always be localized to a single method, they narrow the scope considerably because unit tests by definition concentrate on small portions of code.

A reference test suite may include many test cases for a single feature and, as a result, create duplicate hints and execute the same methods in the student's code. As Web-CAT traditionally eliminates redundant hints, our system records the number of times a hint occurs for a single submission and considers it one unique hint. After recording the test results, for each unique hint, it determines whether the student has tested the feature associated with that hint enough to warrant revealing the hint. To do so, the system calculates the masking MC/DC of the student's own tests against the method(s) associated with that hint.

If the student has not sufficiently tested the atomic decisions within the methods' control structures, the system will withhold the hint and instead direct the student to the method(s) that need more testing. However, if masking MC/DC is fully satisfied on each of the methods associated with the reference test failure, the system reveals the hint. Accordingly, this framework requires students to reflect over each feature of their work and test it thoroughly before receiving hints. In that manner, students cannot rely upon hints to supplant the burden of testing.

3.2 Evaluation of Existing Technology

The aforementioned framework emphasizes offering hints only on features of the student's code that have been adequately tested. The precision of this feature-by-feature feedback would be demonstrated by a lack of both false positives and false negatives in providing hints. A false positive occurs when a hint is provided to a student when the feature it tests has not been tested well by the student. Meanwhile, a false negative occurs when a student has sufficiently tested a feature but does not receive a hint generated by a reference test that failed the same feature. To compare the framework's approach to hint precision to existing approaches in automated testing, we empirically tested the occurrences of false positives and false negatives using Web-CAT's traditional hint mechanism.

Web-CAT allows instructors to customize how many hints students receive for any given assignment submission. However, its default setting is to show a maximum of three hints, where hints with multiple occurrences (multiple failed reference tests generating the same hint) are given precedence. If additional hints are generated by failed tests, they are obscured until later submissions resolve the bugs responsible for the revealed hints. Consequently, hints shown may include those generated by reference tests that fail on features that the student has not tested (a false positive). Likewise, if a student has tested a feature well but a reference test's hint is not within the top three hints, this represents a false negative. Since traditional automated graders do not follow our feature-by-feature scaffolding framework, we hypothesized that Web-CAT does not discriminate between earned (true positives) and unearned hints (false positives) any better than it discriminates obscuring unearned hints (true negatives) from earned hints (false negatives).

To test our hypothesis, we analyzed existing student submissions to Web-CAT's traditional automated feedback system. Using the procedures described in the previous section, we performed post-hoc analysis of which features each reference test exercised as well as whether the students earned each hint (both revealed and obscured) according to masking MC/DC. The student submissions were from two programming assignments from the Fall 2011 term of a CS2 course. We previously collected consent from students to analyze the results of this assignment. After excluding submissions from students who chose not to consent and removing errant submissions (such as non-compiling code), we analyzed 3,688 submissions from 86 individual students.

Students averaged submitting to Web-CAT approximately 24 times (M=23.94, sd=19.62) per assignment. For each failed test within a submission, we evaluated the masking MC/DC score for the methods that particular test exercised and recorded whether it earned the hint with complete masking MC/DC on corresponding methods. After analyzing each unique hint generated for a submission, we also recorded how many unique methods in the student's code needed improved MC/DC to earn all hints. On average, each submission required additional testing on 2.70 (sd=2.05) unique methods, or 12.55% (sd=13.16%) of its solution methods; **Figure 2** shows the distribution across all submissions.

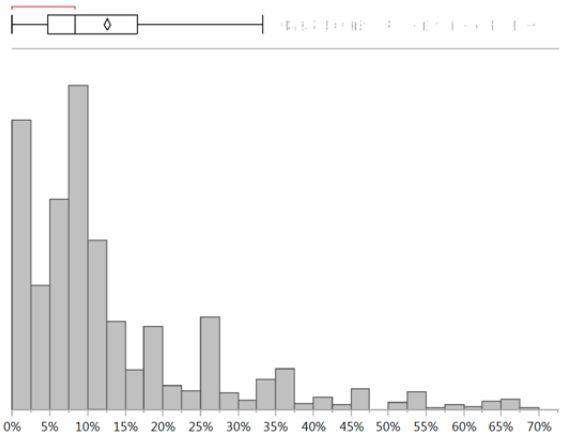

Figure 2. Distribution and Box-plot of Methods Requiring More Testing

Meanwhile, the failed reference tests produced an average of 14.55 (sd=15.38) unique hints per submission. **Figure 3** shows the distribution of earned (full MC/DC for applicable methods) and unearned hints (incomplete MC/DC for applicable methods) that were revealed per submission.

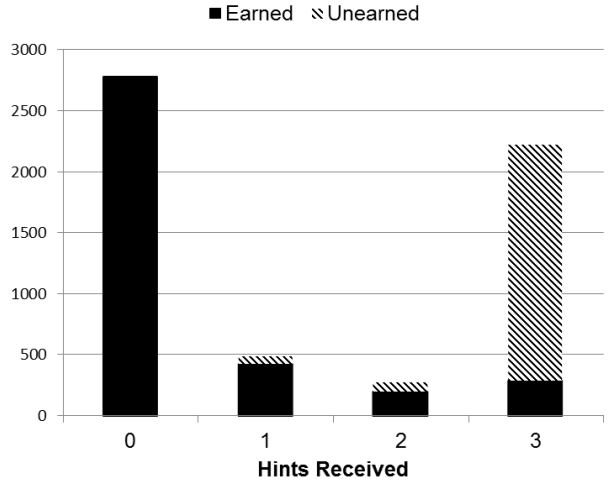

Figure 3. Distribution of Hints Revealed Per Submission, Earned Versus Unearned

As the distribution shows, the vast majority of submission received three hints of which none (0) were earned. This chart reflects only the top three hints that a student could receive on each given submission. The unearned (striped) bars demonstrate instances of false positives: hints that students may depend upon instead of testing the feature robustly. Separately, we also evaluated any remaining hints – those obscured because they were not in the top three – and whether or not they were earned. **Figure 4** shows the distribution of earned and unearned hints among obscured hints.

In this figure, unearned (solid) bars represent true negatives, while earned (dotted) bars greater than zero (0) represent false negatives—hints that were earned but not revealed. False negatives may unnecessarily withhold insight into student's misunderstandings of the assignment specifications.

We used the Shapiro-Wilk test to check for normality of these distributions. The test rejected the null hypothesis (that the data are

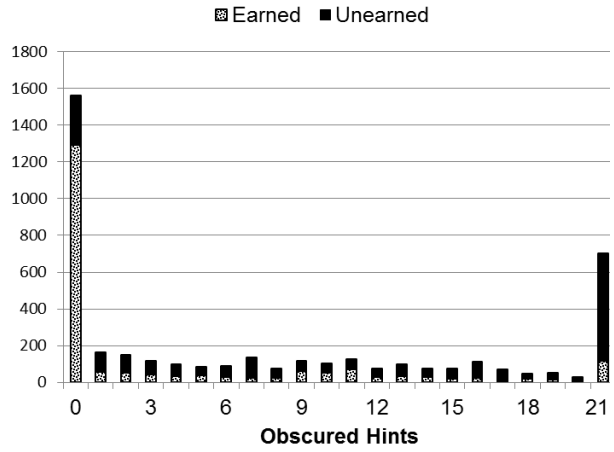

Figure 4. Distribution of Obscured Hints Per Submission, Earned Versus Unearned

normally distributed) for both the unique methods ($p<.01$) and unique hints ($p<.01$). Consequently, to test our hypotheses by comparing the incidents of false positives and true positives for revealed hints to false negatives to true negatives of obscured hints, we used the Friedman's test for within-subject comparisons of non-parametric distributions.

4. RESULTS

To test our hypothesis, we performed a Friedman test comparing the rate of earned-to-unearned hints between those revealed and obscured, when averaged across multiple submissions within-subject. The test found that there was no significant difference ($F(2,1)=0.01$, $p=0.92$) between the rate at which revealed hints were earned ($M=0.20$, $sd=0.36$) and obscured hints were earned ($M=0.20$, $sd=0.31$). This test rejects the null hypothesis that revealed hints have a higher rate of earned-to-unearned hints than obscured hints.

We also investigated the false positive (revealed but unearned hints) and false negative (obscured but earned hints) rates for each assignment independently. **Table 1** shows the false positive and false negatives of the earlier (minesweeper game) and later (linked and array based queue implementations) projects.

Table 1. False Positive and False Negative Rates, by Project

Project	Submissions	False Positives (M, sd)	False Negatives (M, sd)
Minesweeper	1604	0.72, 0.41	0.19, 0.33
Queues	2084	0.84, 0.32	0.21, 0.30

Note that both projects experienced high false positive rates, showing that regardless of the assignment, most hints revealed were not earned. Moreover, both projects maintained false negative rates, where on average, about one-out-of-five hints that were obscured had actually been earned.

5. DISCUSSION

The post-hoc analysis of hints that Web-CAT natively reveal and obscure supported our hypothesis that the automated feedback often results in both false positives and false negatives. False

positives inadvertently relieve some of the burden of testing thoroughly by providing insight into failing tests for features that have not been thoroughly tested. In fact, false positives (revealing unearned hints) were roughly four times more common than true positives (earned hints revealed). Meanwhile, earned hints were no more likely to be revealed than they were to be obscured. With such high false positive and false negative rates, it should not be surprising that the current hint mechanism does not promote students to practice reflective testing before trial-and-error in response to failed hints.

One might consider configuring Web-CAT to remove the limitation to a maximum of three hints per submission. Doing so would eliminate all false negatives. However, the false positive rate would continue to overwhelm the number of earned hints and continue to undermine attempts to encourage students to rely on their own testing instead of using Web-CAT as a crutch. While the interventions described in our previous studies [4][5][6] avoid static restrictions on how many tests Web-CAT reveals, their designs do not account for precision of hints served on a feature-by-feature basis. Consequently, there is no reason to believe that they would have any better success at preventing false positives and false negatives than the default Web-CAT settings did.

To the contrary, our proposed framework offers a unique approach to scaffolding feedback to guide students to revisit their extent of their testing without depending on Web-CAT to determine behavioral acceptability for them. For example, without any test cases, our framework could first indicate only that features in the student's assignment have not been tested—no indication of the program's correctness or results from reference tests. Then, as the student adds test cases and resubmits, the feedback scaffold would give more direction by identifying features that are not working properly and need more thorough testing. Finally, when the student has thoroughly tested a feature but has not resolved its disparity from the project specifications, the framework would provide detailed hints about why the feature fails reference tests.

This framework's approach to scaffolding feedback is modeled after Vygotsky's Zone of Proximal Development [14]. Before a student has written tests, her immediate need in effective software development and problem solving is to reflect over her solution and consider its possible flaws. As she continues to reflect and test thoroughly, she will gradually approach a mastery of the problem where she can use hints to reconcile differences between her implementation and the expected solution.

In addition, it is worth acknowledging that there is no *de facto* standard for measuring test effectiveness. Therefore, while our use of masking MC/DC in our application of the framework should provide more accurate estimations of test thoroughness than conventional coverage, the framework is not explicitly tied to masking MC/DC. As we continue to study methods of evaluating student testing on programming assignments, other measurements could replace masking MC/DC without upsetting the framework's infrastructure. In conclusion, we hope to evaluate how interaction with our framework influences student testing behaviors while continuing to investigate options for assessing test quality.

6. REFERENCES

[1] ABET (2013). "Criteria for Accrediting Computing Programs, 2013-2014." Retrieved November, 2013, from http://www.abet.org/DisplayTemplates/DocsHandbook.aspx?id=3148.

[2] ACM (2014). "Computer Science 2014: Curriculum Guidelines for Undergraduate Programs in Computer Science." Retrieved Nov. 2014, from http://www.acm.org/education/curricula-recommendations.

[3] Buffardi, K. and S.H. Edwards (2014). "A Formative Study of Influences on Student Testing Behaviors." Proc. of Computer Science Education Symposium.

[4] Buffardi, K. and S.H. Edwards (2014). "Responses to Adaptive Feedback for Software Testing." Proceedings of the 19th ACM annual conference on Innovation and technology in computer science education. Uppsala, Sweden

[5] Buffardi, K. and S. H. Edwards (2012). Exploring influences on student adherence to test-driven development. Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education. Haifa, Israel, ACM: 105-110.

[6] Buffardi, K. and S. H. Edwards (2012). "Impacts of Teaching test-driven development to Novice Programmers." International Journal of Information and Computer Science 1(6): 9.

[7] Chilenski, J. J. (2001). "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion."

[8] Chilenski, J. J. and S. P. Miller (1994). Applicability of modified condition/decision coverage to software testing. Software Engineering Journal 193-200.

[9] Edwards, S. H. "Web-CAT.", from https://web-cat.cs.vt.edu.

[10] Edwards, S. H. and Z. Shams (2014). Comparing test quality measures for assessing student-written tests. Companion Proceedings of the 36th International Conference on Software Engineering. Hyderabad, India, ACM.

[11] Edwards, S. H. and Z. Shams, (2014). Do Student Programmers All Tend to Write the Same Software Tests? Proceedings of the 19th ACM annual conference on Innovation and technology in computer science education. Uppsala, Sweden.

[12] "JaCoCo Java Code Coverage Library." 2014, from http://www.eclemma.org/jacoco/

[13] Spacco, J. "Marmoset." 2013, from http://marmoset.cs.umd.edu/.

[14] Vygotsky, L. S. (1978). Mind in Society: Development of Higher Psychological Processes, Harvard University Press.