Paper ID #21636

Teaching Software Testing with Automated Feedback

James Perretta
Dr. Andrew DeOrio, University of Michigan

Andrew DeOrio is a lecturer at the University of Michigan and a consultant for web, machine learning and hardware projects. His research interests are in ensuring the correctness of computer systems, including medical devices, internet of things (IOT) devices, and digital hardware. In addition to teaching software and hardware courses, he teaches Creative Process and works with students on technology-driven creative projects.

Teaching Software Testing with Automated Feedback

James Perretta and Andrew DeOrio
jameslp@umich.edu, awdeorio@umich.edu
Department of Electrical Engineering and Computer Science
University of Michigan

1 Abstract

Computer science and software engineering courses commonly use automated grading systems to evaluate student programming assignments. These systems provide various types of feedback, such as whether student code passes instructor test cases. The literature contains little data on the association between feedback policies and student learning. This work analyzes the association between different types of feedback and student learning, specifically on the topic of software testing.

Our study examines a second-semester computer programming course with a total of 1,556 students over two semesters. The course contained five programming projects where students wrote code according to a specification as well as test cases for their code. Students submitted their code and test cases to an automated grading system. These test cases were evaluated by running them against intentionally buggy instructor solutions. The first semester comprised the control group, while the second semester comprised the experiment group. The two groups received different kinds of feedback on their test cases. The control group was shown whether their tests were free of false positives. In addition to the same feedback as the control group, the experiment group was shown how many intentionally buggy instructor solutions their tests exposed.

Our results measured the quality of student test cases for the control and experiment groups. After students in the experiment group completed two projects with additional feedback on their test cases, they completed a final project without the additional feedback. Despite not receiving additional feedback, their test cases were of higher quality, exposing on average 5% more buggy solutions than students from the control group. We found this difference to be statistically significant after controlling for GPA and whether students worked alone or with a partner.

2 Introduction

Testing is an integral part of software development that helps ensure that software behaves according to its requirements. We will discuss testing practices used in software engineering and how they relate to teaching software testing. We then introduce the theoretical and conceptual frameworks that form the foundation for our study.

2.1 Software Testing

Testing is a critical part of software development. By some estimates, 41% of information technology budgets in North America are spent on quality assurance and testing.[1] Software testing helps ensure the correctness of the software being developed, and there are several test suite quality metrics used in industry to ensure that a test suite properly verifies the behavior of the software it tests.

One widely-used test suite quality metric is code coverage. The goal of a code coverage metric is to determine the percentage of a program's source code executed by a test suite. Coverage tools may be configured to provide statement coverage or branch coverage. Statement coverage measures the percentage of executed statements, while branch coverage measures the percentage of branches taken, such as those in conditionals and loops. Test suites with high coverage measurements expose important errors that would go undetected otherwise.[2] Furthermore, code coverage tools are relatively inexpensive to use, and open source code coverage tools exist for commonly used programming languages such as C++, Java, and Python.[3,4,5]

Another way to measure test suite quality is mutation testing. A shortcoming of code coverage is that it does not consider the pre- and post-conditions of functions verified by a test suite. For example, consider a function that divides two numbers and should throw an exception when the denominator is zero. If the function fails to check for division by zero, a test suite without a test case that passes in zero as the denominator could still achieve 100% coverage, even though the test suite is incomplete. Mutation testing attempts to address this shortcoming. The goal of mutation testing is to evaluate a test suite's ability to expose errors in a program.[6] The process of mutation testing begins with making small modifications to a program's syntax tree, usually with an automated tool. Each modified version of the syntax tree is saved as a "mutant." Each mutant is then evaluated by the original program's test suite. If any of the tests fail, the mutant is considered "killed."[6] If two test suites for the same program are run against a set of mutants, the test suite that kills more mutants is a higher-quality test suite.

2.2 Teaching Software Testing

Despite the focus on testing in industry, researchers have observed that relatively little time in CS courses is devoted to teaching software testing.[7,8,9] Educators have a variety of approaches to teaching software testing available to them, including automated grading systems to provide feedback that students can use to improve the quality of their test cases.

Several approaches to teaching software testing have been applied with some success, such as test-driven development[10,11] or Jones's SPRAE principles.[8] Test-driven development emphasizes a short, frequently-repeated cycle of writing and running test cases while making small additions and changes to the code being developed. SPRAE focuses on a set of essential principles to testing and quality assurance of software: specification, where explicit behaviors must be specified for testing to be possible; premeditation, where testing requires a systematic design process; repeatability, where the results of the testing process must be independently reproducible; accountability, where others must be able to review the testing process; and economy, where testing should be resource-efficient.[8]

In order to accommodate high enrollment in CS courses, instructors may use automated grading systems to evaluate student code and provide feedback to help students fix their mistakes.[12,13] One common approach is for automated grading systems to evaluate student code by running it against an instructor-written test suite and verifying that student code produces the correct output.[13] In recent years, there has been some research into using PL techniques, such as symbolic execution and program repair, to provide automated feedback to students about specific programming errors they made.[14]

Similarly, there are several approaches for automatically evaluating student test cases. These approaches overlap with the test suite quality metrics discussed earlier. For example, Edwards's approach to automatically evaluating student tests centers around code coverage metrics.[9] Since high coverage does not guarantee high test case quality,[10] some instructors turn to a form of mutation testing: student tests are run against a series of intentionally buggy instructor solutions and evaluated based on how many of those buggy solutions are exposed by the tests.[13] Rather than automatically generating these buggy solutions, instructors may choose to write them by hand. This method requires more effort from the instructor, but gives greater control over the bugs student tests are evaluated against.

When providing an automated feedback and grading mechanism to students, instructors must decide what feedback students should receive from the automated grading system. Should test case feedback be immediate, or should it be withheld until after the project deadline? There is some evidence to suggest that students are more likely to write test cases early in the development cycle when they are given a proper incentive, such as automated feedback.[10]

2.3 Theoretical Framework

The theories of active learning and constructivism are central to teaching software testing with automated feedback. Bonwell and Eison define active learning strategies as "instructional activities involving students in doing things and thinking about what they are doing."[15] Although active learning literature generally focuses on student engagement in the classroom,[16] students interacting with an automated feedback tool, implementing a piece of software, receiving feedback on their work, and making changes to it shares many of the same goals as active learning techniques. Similarly, the theory of constructivism emphasizes learning through practice rather than passively receiving information. Automated feedback systems can give students immediate feedback on their work, which helps students discover obstacles and learn how to overcome them while practicing writing software. Our study is concerned with how to improve the active learning processes that are common in teaching software engineering.

Our conceptual framework centers around common practices and philosophies of providing automated feedback in programming assignments. As enrollment numbers increase, CS courses have increased their use of automated grading of programming assignments. Several common ways of evaluating student test cases have emerged as a result of this shift to automated grading. Instructors check for false positives, use code coverage metrics, and use mutation testing to evaluate student tests for true positives. These practices are well-established due to their use in industry and CS courses. The challenge for instructors is to decide what kind of feedback to show students after evaluating their tests.

The amount of time dedicated to teaching software testing in CS courses is small compared to the amount of time devoted to testing in real-world software engineering. While automated feedback provides a possible way to bridge this gap, we lack evidence-based best practices. The goal of our study is to collect data on the effects of two concrete types of automated feedback on student learning of software testing.

Even with more classroom time devoted to teaching software testing, testing is a skill that requires a lot of practice in order to become proficient. By improving our understanding of types of automated feedback, we can help CS courses better utilize automated feedback in order to improve students' software testing skills over the course of their CS curriculum.

2.4 Contributions

In this paper, we evaluate the effectiveness of different types of automated grading feedback on student test case quality. In particular, we examine two research questions:

1. Does automated feedback improve students' ability to write high-quality test cases?
2. What type of automated feedback best encourages student learning of software testing?

3 Methods

Our goal in this study is to measure the effect of automated feedback on student test case quality and student learning of software testing. The participants in this study were enrolled in a second-semester computer programming course over two semesters. We conducted a controlled experiment in which the control and experiment groups received different automated feedback on their software test cases. Our independent variables are the type of feedback students received and whether students worked alone or in a partnership. We also controlled for GPA, normalized to 1. GPA for a partnership was computed as the mean of the two students' GPAs. Our dependent variables are the quality of student test cases measured as a percentage of instructor buggy solutions exposed by the student test cases. We also measure student code correctness as the percentage of instructor-written test cases passed.

3.1 CS2 Course

Our study examines a second-semester computer programming course at a large research institution with a total of 1,556 students over two semesters. The course contained five programming projects where students wrote code and test cases according to a specification. A typical programming project contained the following components: implementing one or more abstract data types (ADTs) according to specification, writing test cases for those ADTs, and writing a command-line program using those ADTs. Instructor project solutions had an average length of 380 lines of code (excluding test cases). Students submitted their code for these components to an automated grading system. Students received automated feedback on their code and test cases on up to three submissions per day during a project. Partnerships collectively received feedback on their tests the same number of times as those working alone.

Students in the course attended three hours of lecture per week and a two hour lab session each week. Lab sessions consisted of a short exercise worksheet and a programming activity designed to supplement material from lecture. Lecture and lab sections in the course were synchronized to ensure that students learned the same material regardless of which section they attended. Students were evaluated using two exams: a midterm and final exam. The grading rubric used for the course was as follows:

Lab exercises	5%
Programming projects	40%
Midterm exam	25%
Final exam	29%
Participation in course surveys, etc.	1%

Student ADT implementations were graded using an instructor-written test suite. Students received feedback on a few publicly-available instructor tests. Most instructor tests were hidden until after the project deadline.

Student test cases were evaluated using intentionally buggy instructor solutions. First, each student-provided test case was compiled and run against a correct instructor solution. Test cases that incorrectly reported an error on the correct instructor solution (a false positive) were marked as invalid and discarded. Then, each valid (free of false positives) test case was run against a series of intentionally buggy instructor solutions. If at least one valid student test case reported an error for a buggy instructor solution, that buggy solution was marked as exposed. Students were awarded points based on the number of buggy instructor solutions their tests exposed.

The buggy instructor solutions used in our study were designed to mimic common logic errors students might make in their programming projects. In Figure 1, we give an example of an instructor buggy solution. The philosophy of this approach is based on two hypotheses proposed by mutation testing. First, the Competent Programmer Hypothesis states that the mistakes programmers tend to make are relatively small and therefore not significantly different from the correct version.[6] Second, the Coupling Effect states that if a test suite is able to expose bugs that are caused by small mistakes, that test suite should also expose bugs caused by large mistakes.[6]

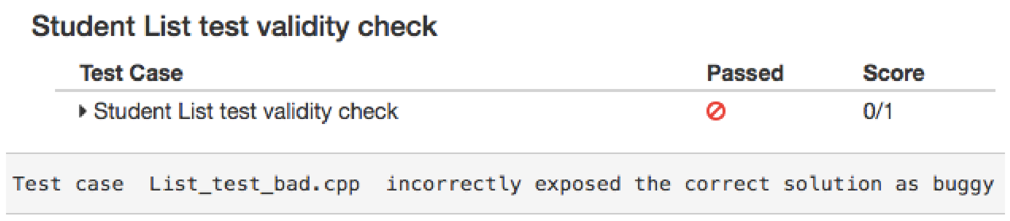
```
// CORRECT implementation.
template <typename T>
void List<T>::push_back(
      const T &datum) {
   Node *np = new Node;

   if (empty()) {
      np->prev = 0;
      first = np;
   } else {
      np->prev = last;
      last->next = np;
   }
   np->next = 0;
   np->datum = datum;
   last = np;
   ++num_nodes;
}
```

```
// BUGGY implementation: Fails
// to set some pointers
// correctly if the list is
// empty.
template <typename T>
void List<T>::push_back(
      const T &datum) {
   Node *np = new Node;

   np->prev = last;
   last->next = np;

   np->next = 0;
   np->datum = datum;
   last = np;
   ++num_nodes;
}
```

Figure 1: An instructor buggy solution containing an error in a linked list function. The code on the left is a correct implementation and the code on the right is a buggy implementation.

3.2 Control and Experiment Groups

The first semester comprised the control group, while the second semester comprised the experiment group. Both groups received the same lecture material and lab curriculum on software testing. The two groups received different kinds of feedback on their test cases. Students in the control group received the same type of feedback on all projects. This feedback, shown in Figure 2, included only whether student test cases passed when run against a correct instructor solution (free of false positives).



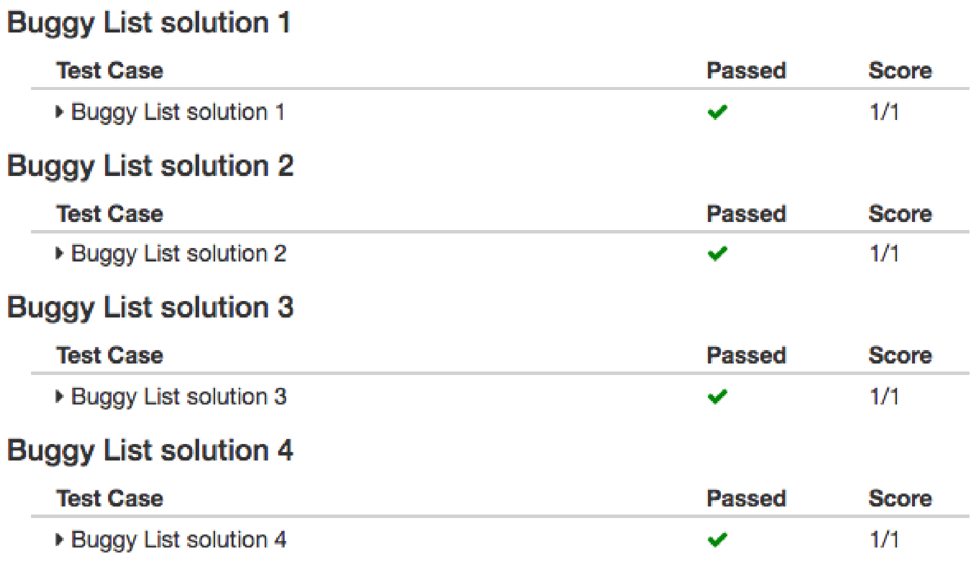
Student List test validity check

Test Case	Passed	Score
▸ Student List test validity check	⊘	0/1

```
Test case  List_test_bad.cpp  incorrectly exposed the correct solution as buggy
```

Figure 2: Automated test case feedback for the control group. This feedback indicates which student test cases produced false positives, if any.

Students in the experiment group received additional feedback on their test cases on Projects 3 and 4. In addition to all of the same feedback that the control group received, the experiment group was shown the number of buggy instructor solutions their test cases exposed (true positives). This feedback is shown in Figure 3. On Project 5, the experiment group received the same feedback as the control group (whether their tests were free of false positives).


Buggy List solution 1

Test Case	Passed	Score
▸ Buggy List solution 1	✓	1/1

Buggy List solution 2

Test Case	Passed	Score
▸ Buggy List solution 2	✓	1/1

Buggy List solution 3

Test Case	Passed	Score
▸ Buggy List solution 3	✓	1/1

Buggy List solution 4

Test Case	Passed	Score
▸ Buggy List solution 4	✓	1/1

Figure 3: Automated test case feedback for the experiment group, which also received the same feedback as the control group, shown in Figure 2.

3.3 Data Collection and Variables

We measured student learning in two ways: the quality of student solution code, and the quality of student test case code. We collected the percentage of buggy solutions that student test cases exposed as a measure of the quality of student test cases. We also collected student scores on the instructor-written ADT tests as a measure of solution code quality. In our statistical analysis, we analyze the distributions of these datasets using analysis of variance.

We consider the following independent variables in our analysis:

- Test case feedback type (control and experiment groups).
- Partnership status (whether students worked alone or in a partnership).
- Student GPA, normalized to 1. GPA for a partnership was computed as the mean of both students' GPAs.

We measure these dependent variables:

- Student test case quality, measured as the percentage of instructor buggy solutions that student tests exposed.
- Student solution quality, measured as the score that student solution code received on the instructor test suite.

4 Results

We examine the quality of student test cases on three projects, both in the experiment and control groups. We first analyze the association between test case feedback and test case quality. Then, we analyze the relationship between partnership status and test case quality.

Our independent variables are the type of feedback students received (control and experiment groups) and whether students worked alone or with a partner. We control for GPA in our analysis. Our dependent variables are test case quality on three projects, measured as the percentage of instructor-written buggy solutions that were exposed by student test cases.

4.1 Test Case Feedback and Test Case Quality

In our first experiment, we examine the relationship between the type of feedback students received and test case quality. The control and experiment groups received different feedback on Projects 3 and 4. On Project 5, both groups received the same feedback. Our independent variable is test case feedback. We control for GPA and partnership status.

For Projects 3 and 4, the control group received feedback on whether their tests were free of false positives. The experiment group received additional feedback on the percentage of buggy solutions their test cases exposed (true positives). We see several statistically significant associations with test case quality: type of feedback students received, GPA, and partnership status. We discuss this in more detail in section 5.1. In Figure 4 and Table 1, we see that the mean percentage of bugs exposed on Project 3 by the control group was 62%, whereas the mean for the experiment group was 74% of buggy solutions, a difference of 12%. We also see that the mean percentage of buggy solutions exposed in Project 4 by the control group was 70%, while the mean for the experiment group was 83%, a difference of 13%.

For Project 5, the control and experiment groups received the same feedback. Both groups received feedback on whether their tests were free of false positives. We see several statistically significant associations with test case quality: the type of feedback students received, GPA, and partnership status. We discuss this in more detail in section 5.1. In Figure 4 and Table 1, we see that the mean percentage of bugs exposed by the control group was 64%, whereas the the mean percentage of bugs exposed by the experiment group was 69%, a difference of 5%.

Test Quality Summary for Test Case Feedback

	Project 3		Project 4		Project 5	
	Control	Exp	Control	Exp	Control	Exp
N	433	631	428	625	406	593
Mean	62%	74%	70%	83%	64%	69%
Stdev	29%	29%	22%	20%	21%	23%

Table 1: Summary of student test case quality for control and experiment groups. Compared to the control group, the experiment group mean test quality was 12% higher on Project 3, 13% higher on Project 4, and 5% higher on Project 5.

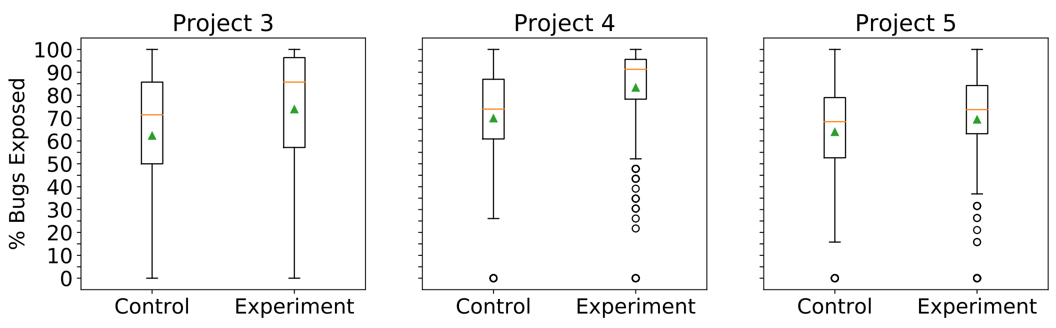


Figure 4: Student test case quality for control and experiment groups. Test quality is shown on the Y-axis. The X-axis indicates the control or experiment group. The triangle indicates the mean.

	Project 3				Project 4				Project 5			
	df	Sum Sq.	F	PR(>F)	df	Sum Sq.	F	PR(>F)	df	Sum Sq.	F	PR(>F)
Group † ∗ ‡	1	2.2	40.95	2.34e-10	1	3.43	114.92	1.64e-25	1	0.46	12.04	5.44e-04
Partner † ∗ ‡	1	3.03	56.32	1.31e-13	1	1.59	53.38	5.45e-13	1	1.24	32.29	1.75e-08
Group x Partner ∗	1	0.01	0.11	7.39e-01	1	0.27	8.97	2.81e-03	1	0.14	3.6	5.82e-02
GPA † ∗ ‡	1	25.91	481.46	3.19e-88	1	11.76	394.25	1.08e-74	1	9.66	251.18	1.36e-50
GPA x Group	1	0.02	0.34	5.60e-01	1	0.0	0.12	7.26e-01	1	0.04	1.02	3.14e-01
GPA x Partner ∗	1	0.0	0.0	9.63e-01	1	0.15	4.9	2.71e-02	1	0.0	0.02	8.88e-01
GPA x Group x Partner	1	0.0	0.07	7.87e-01	1	0.07	2.4	1.21e-01	1	0.06	1.56	2.11e-01
Residual	1056.0	56.83			1045.0	31.17			991.0	38.12		

Table 2: ANOVA for student test case quality. The independent variables were control or experiment group, partnership status, and GPA. The dependent variables were test case quality scores on three projects. Statistically significant associations are indicated for project 3†, project 4∗, and project 5‡. Test case quality and GPA are normalized to 1.

4.2 Partnership Status and Test Case Quality

We now examine the relationship between partnership status and test case quality, measured as percentage of bugs exposed. Students worked either alone or with an optional, student-selected partner. Our independent variable is partnership status. We control for GPA and feedback type.

We see a statistically significant association with test case quality from partnership status, the type of feedback students received, and GPA. We discuss this in more detail in section 5.2. In Figure 5 and Table 3, we see that the mean percentage of bugs exposed in Project 3 by the students who worked alone was 63%, whereas the mean for students who worked with a partner was 77% of buggy solutions, a difference of 14%. We also see that the mean percentage of bugs exposed in Project 4 by the students who worked alone was 74%, whereas the mean for students who worked with a partner was 83% of buggy solutions, a difference of 9%. In Project 5, the mean percentage of bugs exposed by the students who worked alone was 63%, whereas the mean for students who worked with a partner was 71% of buggy solutions, a difference of 8%.

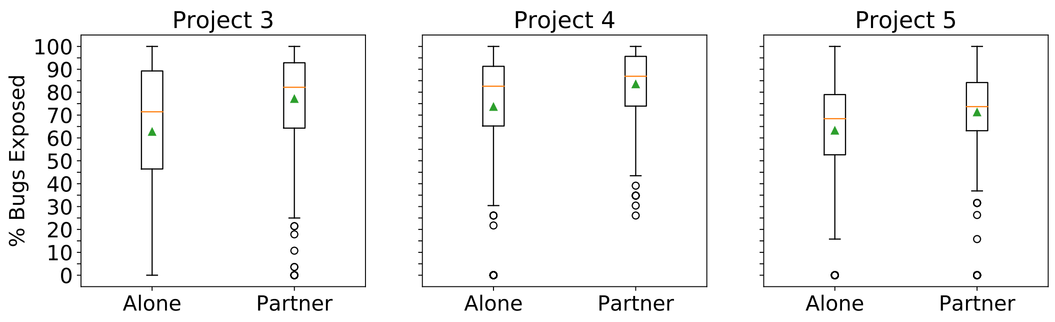


Figure 5: Student test case quality with respect to partnership status. The X-axis indicates whether students worked alone or with a partner and test case quality is shown on the Y-axis. The triangle indicates the mean.

Test Quality Summary for Alone vs. Partner

	Project 3		Project 4		Project 5	
	Alone	Partner	Alone	Partner	Alone	Partner
N	588	476	601	452	510	489
Mean	63%	77%	74%	83%	63%	71%
Stdev	33%	22%	25%	15%	24%	19%

Table 3: Summary of test quality statistics for partnership status. We see that the mean test quality for students who worked with a partner was 14% higher on Project 3 than the mean test quality for students who worked alone, 9% higher on Project 4, and 8% higher on Project 5.

5 Discussion

We observed a statistically significant association between increased automated test case feedback and student test case quality. Notably, when the augmented feedback was taken away, students still produced higher quality test cases. We also temper our discussion by noting that the effective size was small compared to the contribution of GPA. We also examined student code quality as measured by student scores on instructor ADT tests, but we did not find a statistically significant association.

5.1 Test Case Feedback

In Projects 3, 4, and 5, we see that the experiment group consistently achieved higher mean test case quality than the control group. After controlling for partnership status, we see that the association between increased test case feedback and test case quality was comparable to the association between partnership status and test case quality. We also note that the magnitudes of these associations were much smaller than that of GPA.

In Projects 3 and 4, where the control and experiment groups received different feedback on their test cases, we observed that the experiment group achieved higher test case quality than the control group. The experiment group was given additional feedback on their test cases, and these students had time to act on the additional feedback and improve their tests. In Table 2, we see that the difference in test case quality between the two groups is statistically significant in Projects 3 and 4.

For Project 3, we see that the mean test case quality for the experiment group was 12% higher than that of the control group. This difference translates to about 3 additional instructor buggy solutions exposed by students in the experiment group. For Project 4, we see that the mean test case quality for the experiment group was 13% higher than that of the control group. This difference translates to about 3 additional instructor buggy solutions exposed by students in the experiment group.

In Project 5, the control and experiment groups received the same feedback. This feedback indicated only whether student tests were free of false positives. A difference in test case quality between the control and experiment groups for Project 5 therefore supports the hypothesis that prior test case feedback influenced student learning. We see that the mean test case quality for the experiment group was 5% higher than the mean test case quality for the control group. This difference translates to about 1 additional instructor buggy solution exposed by students in the experiment group.

5.2 Partnership Status

We observed that students working with a partner produced higher quality test cases than students working alone. We found this to be an intuitive result, as two people coming up with test cases instead of one seems more likely to produce a more robust test suite. In Projects 3, 4, and 5, we see that students who worked with a partner consistently achieved higher test case quality than students who worked alone. In Table 2, we see these results to be statistically significant when controlling for GPA. We also see that the magnitude of the association of partnership status and test case quality was small compared to GPA.

For Project 3, the mean test case quality for students who worked with a partner was 14% higher than for students who worked alone. This translates to exposing about 4 more instructor buggy solutions. For Project 4, the mean test case quality for students who worked with a partner was 9% higher than for students who worked alone. This translates to exposing about 2 more instructor buggy solutions. For Project 5, the mean test case quality for students who worked with a partner was 8% higher than for students who worked alone. This translates to exposing about 1-2 more instructor buggy solutions.

5.3 Limitations

Our study design was experimental, but several factors were beyond our control. The three projects included in our experimental design may have varied in their difficulty for students as well as the difficulty of exposing instructor buggy solutions.

Our control and experiment groups came from two different semesters of the course. This design decision was in the interest of fairness to students, rather than dividing students from one semester into two groups. We note that both semesters were very consistent in their organization and material.

Another factor beyond our control was student partnerships. Students chose whether to work with a partner or alone. Furthermore, students chose their own partners.

6 Conclusions

We have examined the relationship between automated feedback and student learning of software testing. We performed an experimental study where two groups of students received different types of automated feedback on their test cases. We found that students who received additional feedback on the number of instructor buggy solutions their tests exposed wrote higher-quality test cases, even after this augmented feedback was taken away.

We also examined the relationship between student partnerships and test case quality, finding that students who worked with a partner consistently wrote higher-quality test cases. We also note that the magnitude of these associations was small compared to that of the association between GPA and test case quality.

Future work could present students with software testing challenges that can be used to more consistently evaluate student test case quality. An additional study could evaluate the effects of students using code coverage metrics on the quality of their test cases.

Computer science educators can use the results of this study to help inform their decisions on how to evaluate student test cases and what sort of test case quality feedback to provide to students. While prior work and conventional wisdom indicate that students will act on the feedback they are given, this study suggests that providing them with specific test quality metrics will improve their ability to test software.

References

[1] Dan Hannigan and Michelle Walker. World quality report 2015-16. Technical report, 2015.

[2] A. Dupuy and N. Leveson. An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software. *19th Digital Avionics Systems Conference*, 2000.

[3] Coverage.py. https://coverage.readthedocs.io/en/coverage-4.5.1/.

[4] Cobertura: A code coverage utility for java. http://cobertura.github.io/cobertura/.

[5] gcov - a test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[6] Yu Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37:649–678, 2010.

[7] David Carrington. Teaching software testing. *Proceedings of the Australasian conference on computer science education*, pages 59–64, 1997.

[8] Edward L. Jones and Christy L. Chatmon. A perspective on teaching software testing. *Journal of Computing Sciences in Colleges*, 16:92–100, 2001.

[9] Stephen H. Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing*, 3, 2003.

[10] Jaime Spacco and William Pugh. Helping students appreciate test-driven development (TDD). *Proceedings of OOPSLA*, pages 907–913, 2006.

[11] Chetan Desai, David Janzen, and Kyle Savage. A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin*, 40:97–101, 2008.

[12] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Computers and Education*, 41(2):121 – 131, 2003.

[13] Kirsti M Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15:83–102, 2007.

[14] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *Proceedings of PLDI*, pages 15–26, 2013.

[15] James A. Bonwell, Charles C.; Eison. Active learning: Creating excitement in the classroom. *ASHE-ERIC Higher Education Reports*, 1991.

[16] Michael Prince. Does active learning work? a review of the research. *Journal of engineering education*, 93: 223–231, 2004.