An Automated System for Interactively Learning Software Testing

Rebecca Smith
Rice University
rjs@rice.edu

Terry Tang
Rice University
terry.tang@rice.edu

Joe Warren
Rice University
jwarren@rice.edu

Scott Rixner
Rice University
rixner@rice.edu

ABSTRACT

Testing is an important, time-consuming, and often difficult part of the software development process. It is therefore critical to introduce testing early in the computer science curriculum, and to provide students with frequent opportunities for practice and feedback. This paper presents an automated system to help introductory students learn how to test software. Students submit test cases to the system, which uses a large corpus of buggy programs to evaluate these test cases. In addition to gauging the quality of the test cases, the system immediately presents students with feedback in the form of buggy programs that nonetheless pass their tests. This enables students to understand why their test cases are deficient and gives them a starting point for improvement. The system has proven effective in an introductory class: students that trained using the system were later able to write better test cases — even without any feedback — than those who were not. Further, students reported additional benefits such as improved ability to read code written by others and to understand multiple approaches to the same problem.

Keywords

Computer science education; Interactive learning; Software testing; Automated assessment

1. INTRODUCTION

Testing is an important part of software development that can take up to half of the total development time [3]. Thus, it is critical to teach students how to test software [4, 5, 6, 19]. Yet, computer science education focuses primarily on software development — including topics such as languages, tools, and design principles — and not on testing. This is particularly true at the introductory level.

Further, many students do not want to test their code, often viewing it as an unpleasant afterthought [23]. While students often derive a great deal of satisfaction from seeing a program that they wrote solve a problem, they do not derive that same sense of satisfaction from exposing flaws in their own programs. However, there is evidence that learning to test helps students to write better code and to do so more quickly [14, 24]. Therefore, it is crucial to design and develop strategies to expose students to testing early in their computer science education [18, 32].

This paper presents an automated and interactive system designed to help students learn how to write better test cases. The system was evaluated through a study that was conducted in an introductory computer science class. In our system, students are given a specification for a program and asked to submit a test suite for that program. Note that they do not need to write the program; rather, they must think about what inputs would stress an implementation of the given specification. After submitting their tests and receiving feedback from the system, students can iteratively improve their tests and resubmit. We believe that several key elements of the system that contribute to its effectiveness.

First, students are not testing their own code. When testing their own code, students are less motivated to find bugs, as bugs expose their own failure to develop a correct program [8, 21]. When there is no personal investment in the code being tested, students are able to view testing as a rewarding activity and are driven to find as many bugs as possible. Second, the submitted tests are evaluated based on the number of buggy programs they detect from a large corpus of buggy programs, not their code coverage. This approach is motivated by the fact that techniques such as mutation analysis and all-pairs testing have been shown to be more effective than code coverage at identifying weak test sets [2, 16], and thus give students a a far more accurate view of the quality of their test cases.

Finally, and perhaps most importantly, the system provides immediate feedback that not only tells students how well their tests perform, but also gives clear examples of buggy implementations that their tests do not detect. It is often difficult for a student to reason about all possible edge cases in a program. Our system provides multiple example implementations with similar bugs, allowing students to understand *why* their test suite is not comprehensive by seeing the type(s) of bugs that their tests do not catch. This feedback not only helps students to develop better test cases, but also gives them practice reading, understanding, and debugging code written by others. Further, it has the added benefit of exposing students to a diverse set of solution strategies to the same problem.

The system has proven to be successful at teaching introductory students how to write better test cases. After using the system, students were able to think about testing in a more comprehensive way and to develop better test cases without continuing to need to see example implementations. With no other training, students

ITiCSE '17, July 03 - 05, 2017, Bologna, Italy

ISBN 978-1-4503-4704-4/17/07…$15.00

DOI: http://dx.doi.org/10.1145/3059009.3059022

that used the system were able to score nearly a standard deviation higher on a testing assessment than students who had not — a statistically significant result ($p < 0.0001$). Furthermore, over 73% of the students reported that the system improved their ability to write comprehensive test cases, to read code written by other people, and to understand multiple approaches to the same problem.

The paper proceeds as follows. Section 2 places our contributions in the context of past work. Section 3 describes our system for teaching students to write better tests. Section 4 describes the study used to evaluate this system, and Section 5 presents the results of this study. Last, Section 6 concludes the paper.

2. RELATED WORK

Much of the past work on teaching software testing has focused on broad structural changes to place greater emphasis on testing within computer science curricula [4, 5, 6, 14, 18, 19], touting the importance of integrating testing into the curriculum early and often. Such work is largely orthogonal to the methodological techniques presented in this paper. Several other papers have proposed highly formal approaches to teaching software testing in upper-level courses which require devoting roughly half of the instructional time to teaching software testing [9, 10, 19]. While appropriate in an advanced context, these approaches are too heavyweight for an introductory course where testing is not the primary focus.

Of the past work that has investigated techniques for integrating testing into introductory courses, the most closely-related work focuses on mutation analysis and all-pairs testing. DeMillo *et al.* helped popularize mutation testing [12], in which students simultaneously submit an implementation and set of test cases; the submitted implementation is mutated in various ways, and the mutants are used to determine how effective the test suite is as finding bugs. However, their approach does not assess students consistently, as the mutants presented to each student are highly personalized.

More recently, Aaltonen *et al.* demonstrated the superior ability of mutation analysis, as compared to code coverage, to identify weak test suites [2]. Like DeMillo *et al.*, they used the student's own implementation to generate mutants on the fly. However, they noted a flaw in their approach: students could exploit the grading system by introducing dead code into their solution, which would result in an artificially-inflated score. In contrast to our interactive framework, both Aaltonen *et al.* and DeMillo *et al.* performed mutation analysis after-the-fact to evaluate students' final submissions.

Another popular approach to testing in introductory courses is pairwise testing, which requires that students submit both an implementation and a test suite, and scores the test suite by running it on one or more implementation(s): traditionally, submission(s) of one [21] or more [17, 20, 22, 26] classmates. Such approaches require students to submit a final test suite "blindly", with no opportunity to interact with the implementation(s) on which their tests will be evaluated. Similar work has used alternate implementations including an instructor-provided reference implementation [11], or even a corpus of instructor-provided buggy implementations [29], posing less challenges to interactivity. Still, this past work largely required blind submissions and after-the-fact evaluation.

Recent work has improved pairwise testing by proving the feasibility of applying it synchronously, enabling students to see their scores progress as they develop their test cases [31]. This work required students to develop implementations and test suites simultaneously, and built up a corpus of student implementations to run the tests against as students submitted their work. However, this work faces several challenges. First, as the corpus of student implementations grows, the score associated with a given test suite will decrease, which may frustrate students. Additionally, the quality of the evaluation relies on the diversity of the student submissions, which is unpredictable. Last, the paper does not indicate that students receive any feedback beyond the score to help them complete the exercises, nor does it provide any evaluation of the effectiveness of the tool in improving the students' testing capabilities.

In contrast, our system emits consistent, deterministic scores for a given test suite. Further, it ensures a diverse and thorough pool on which to evaluate students' test suites by using a pre-existing corpus of programs that melds instructor-provided solutions, student solutions from prior iterations of the course, and prefabricated mutants. Additionally, we have experimentally proven the success of our system by performing a thorough evaluation its effectiveness within a controlled experimental environment.

3. INFRASTRUCTURE

Many studies have cited limited time (both preparation time and instruction time) as one of the major barriers to teaching software testing [15, 25]. This motivates the use of an automated system to help students develop testing abilities. Yet, an even larger body of work demonstrates the value of active learning [7, 28, 30]. To balance these concerns, we created a system that evaluates students' test suites immediately and automatically, but fosters interactive learning by creating a constant cycle of formative feedback. This system was implemented for Python programs, but the pedagogical principles underlying it transcend any particular language.

At a high level, students are given a natural language specification for a function and tasked with developing a test suite: a list of test cases, where each test case is a tuple of inputs that adheres to the specification. As students develop tests, they can submit them to the system at any time and receive feedback. This feedback is designed not only to evaluate the quality of the submitted tests, but also to help students to recognize subsets of the input domain that their tests do not cover. They can then iteratively improve their tests based on this feedback until they are satisfied with their results.

In greater detail, the system runs the submitted test suite against a corpus of incorrect implementations of the function, and provides the student with two pieces of feedback. First, they receive a score based on the number of incorrect implementations that were "caught" by the submitted test suite, *i.e.* that failed one or more test. Second, they are shown the source code for one or more incorrect programs that were not caught by their tests. Specifically, the system outputs up to three programs that share the same "test signature", *i.e.* that exhibit the same set of bugs. Limiting the feedback to a single test signature scaffolds the students' progress without immediately revealing all uncaught bugs. At the same time, providing multiple examples of implementations that exhibit that single signature exercises students' pattern-matching skills, facilitating the process of identifying weaknesses in their current test suite.

As Figure 1 shows, our system consists of four key phases: creating a corpus of implementations, identifying bugs within these implementations, developing a scaffolded progression through these implementations, and an interface through which students can interact with the scaffolded exercises as they develop their test suites.

3.1 Corpus Constructor

There are many viable means of building a corpus of implementations; we chose to combine two. First, we extracted student solutions S from CodeSkulptor [33], an online interactive development environment (IDE) that automatically stores version history as stu-

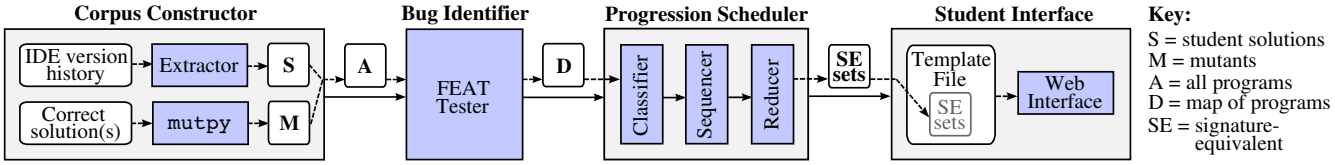


Figure 1: System Infrastructure

Table 1: Corpus Sizes for Training & Test Functions

Function	$\|S\|$	$\|M\|$	$\|A\|$	|SE sets|
`blackjack3`	156	110	266	71
`format`	2295	235	2530	76
`stringtime`	115	263	378	97

dents develop their code. This IDE has been used by many students across multiple MOOCs and physically co-located classes, and thus has amassed a large corpus of student solutions to the programming exercises posed by these courses. By mining implementations directly from the IDE's version history rather than using final submissions, we capture a broader — and thus more interesting for testing — spectrum of buggy and incomplete solutions.

Second, we ran `mutpy` [13], a mutation testing tool, on correct solutions to introduce a set of additional faulty implementations M, giving us an aggregate set of implementations $A = S \cup M$. As a supplement to S, M serves two purposes. First, it introduces new test signatures for students to face as they develop their tests. Second, it increases the number of implementations with the same test signature, allowing the system to provide students with more detailed feedback for debugging. M is also effective in the absence of S, rendering our system viable for small classes, new classes, and other scenarios in which a large corpus of student implementations may not yet exist. Table 1 shows the corpus sizes and number of unique signatures for the training functions (`blackjack3`, `format`) and the test function (`stringtime`) used in the evaluation.

3.2 Bug Identifier

Identifying bugs within A is necessary in order to identify unique test signatures and choose which programs should be displayed as feedback. For this, we used FEAT [34], an existing tool for test case generation and automated programming assessment. FEAT begins by creating an expansive "base test set" through a combination of exhaustive and random generation. For the functions used in our experiment, the input domains were sufficiently constrained to use fully-exhaustive base test sets. FEAT outputs a mapping D associating each $a \in A$ with the set of test cases that it failed on. This set of failed tests is its test signature; implementations that share the same test signature typically have the same or very similar bugs.

3.3 Progression Scheduler

For the third component of our system, we extended FEAT with a new module that classifies and sequences programs based on their signatures. This module first creates a mapping of each signature to the set of programs with that signature ("signature-equivalent" or SE sets). It eliminates the empty test signature, as this represents programs that have no bugs. Next, it sequences the SE sets in order of descending signature size. As students work through the testing exercise, they will be provided with feedback in this order. The intuition for this is that it will be easier for students to create test suites that catch programs that fail on a larger number of cases than on those that fail in only one or two specific scenarios.

Figure 2: Sample Function Specification

> The `blackjack3()` function takes as its input three cards, each of which is represented as a single character from the string "23456789TJQKA". For instance, one valid tuple of inputs might be ("A", "K", "5"). In blackjack, number cards (including "T", which represents 10) are worth their value, and face cards ("JQK") are worth 10. Aces can be worth either 1 or 11; the choice between these is made such that the value of the hand is as high as possible without going over 21. (Note that there are some cases in which it is impossible to stay under 21.) Returning to the example of ("A", "K", "5"), the expected output would be 16. Your input file should contain a single Python definition:
>
> - A list `TEST_CASES` containing at most **12 test cases** for the function `blackjack3()`.
> - Each test case in this list should be a list of tuples of length three whose entries are characters in "23456789TJQKA".

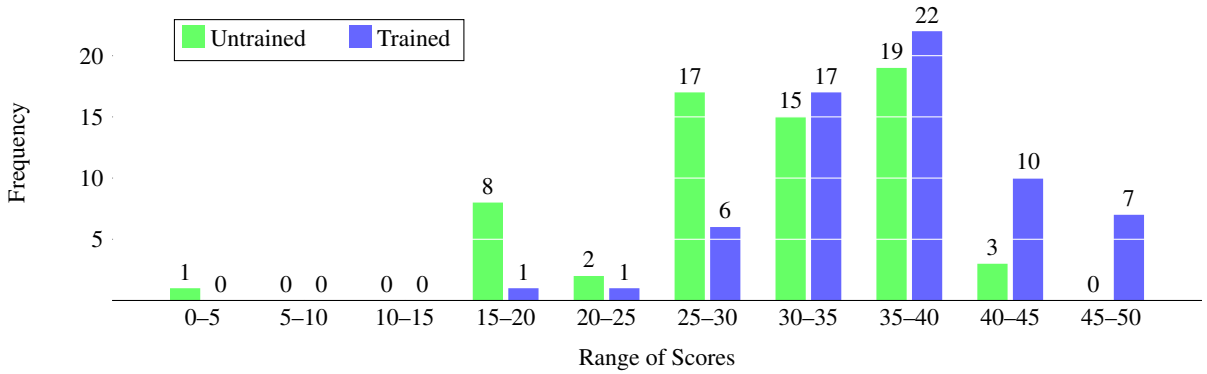
Each signature-equivalent set may be arbitrarily large. However, presenting students with a large number of programs may be overwhelming. So, the module selects a maximum of three programs to represent each signature in the feedback provided to the students. To choose these three programs, it uses the `radon` package [1] to compute the McCabe cyclomatic complexity metric [27] for each program. The module then selects those programs that have the lowest complexity, with the goal of making the feedback maximally approachable for the students. Thus, at the end of this process, the module outputs a list of sets of programs, where each element in the outer list is a signature-equivalent set of size at most three.

3.4 Student Interface

To enable students to interact with these signature-equivalent sets, we used our existing auto-grading system, which requires instructors to upload a file defining how student submissions are processed and scored. We created a template file that imports the submitted test suite, runs a validation function to ensure that it meets the specification, and then runs each of the submitted tests on one representative program from each signature-equivalent set. During this process, it makes note of the first signature that the student's tests fail to catch, as that signature will be used for feedback. It then computes a score based on the proportion of signatures caught.

Last, this file generates feedback: the score and the source code of the program(s) in the first SE set that the submitted tests failed to catch. This scaffolds the exercise such that students receive some direction regarding which cases they're missing, but are not immediately given full "white-box" knowledge. For each function to be tested, the template need only be modified to import the correct SE sets and to provide a specification for the function being tested, a

Figure 3: Assessment: Final Scores



reference solution, and a validation function for the test suite. Optionally, the grading scale may also be customized.

4. METHODOLOGY

We evaluated the framework described in Section 3 within the fall 2016 iteration of an introductory computer science course that teaches students how to solve problems in a computational way and how to implement their solutions in Python. This course has no prerequisites and is taken by a mix of majors and non-majors.

We ran a study consisting of a training exercise, an assessment, and two surveys. Students were randomly assigned to a control or experimental group. Both groups completed the assessment simultaneously; however, the experimental group completed the training exercises beforehand, while the control group did not complete the training until afterwards. Training the control group after-the-fact was not necessary for the study, but preserved fairness in workload and ensured that both groups reaped the benefits of the training. Neither group received any other training in testing prior to the assessment. Due to random assignment, we would expect both groups to perform equally on the assessment, all else being equal. Thus, since the only difference between groups was whether they had completed our training, inter-group differences on the assessment are likely due to that training. Last, both groups completed surveys before the training and after the test, in which they reported their level of confidence in their software testing abilities.

The training exercise was a mandatory homework assignment in which students used the framework described in Section 3 to develop test cases for two training functions. Students were given five days in which to complete these training exercises. The assessment was a mandatory in-class assignment, but was not counted towards the students' course grades for the sake of fairness. On the assessment, students were given a natural language specification for a function and asked to construct a list of at most 12 test cases for this function. To eliminate any potential advantages due to familiarity with the interface discussed in Section 3, the assessment was completed by hand. Students were given 15 minutes in which to complete the assessment. They were also asked to write down their partial solutions after five and ten minutes to provide greater insight into their process as they progressed through the exercise.

The pre-survey asked the students about their attitudes toward testing and confidence in their testing abilities. The post-survey asked the same questions, so that the delta might show the effectiveness of the training exercises. However, we hypothesized that students might not recognize their lack of testing capabilities until after they completed the training assignment, so the post-survey also included explicit questions about the effectiveness of the training.

Table 2: Assessment: Summary Statistics ($p < 0.0001$)

	Min	Average	Max	Stdev	N
Untrained	0	30.65	43.2	7.81	68
Trained	17.7	36.53	50	6.34	66
Aggregate	0	33.54	50	7.69	134

5. EVALUATION

In fall 2016, 196 students were enrolled in the course in which we evaluated our system. 141 of these students consented to participate in the study. Students were randomly assigned to the control and experimental groups; of the consenting students, 70 were assigned to the control group and 71 to the experimental group. Due to absences, 134 students ultimately took the assessment, of which 68 belonged to the control group and 66 to the experimental group. We ran this study approximately halfway through the semester, at which point students had gained basic familiarity with Python and had likely written some test cases on their own, but had received no formal instruction on software testing within the course.

We used three pieces of data to assess the effectiveness of the training exercises. First, we compared the distributions of assessment scores of the experimental (trained) and control (untrained) groups. Second, we compared the students' self-reported confidence in their abilities before and after completing the training exercises. Last, we examined the students' perceptions of the effectiveness of the training exercises.

Assessments were scored out of 50, where the score was proportional to the number of unique signatures caught; Table 2 presents the summary statistics for each group, and Figure 3 shows the full distribution of scores. While both groups' scores were roughly normally distributed, the trained group outperformed the untrained group by nearly a full standard deviation, with an average score of 36.53 as compared to 30.65 for the untrained group. A one-tailed unpaired t-test yielded a p-value of < 0.0001, which is regarded as statistically significant. Note that students received no feedback during assessment; the fact that the trained students performed better on the test even in the absence of feedback indicates that, while the feedback during training likely helped them to improve, they did not become dependent on it for future success in writing tests.

Table 3: Implicit Feedback

"I am confident in my ability to..."	% Agree (Pre-survey)	% Agree (Post-survey)	Delta
Identify distinct logical categories of input (corresponding to unique code paths) for a particular problem	59.3%	77.6%	+18.3%
Write comprehensive test cases	34.5%	55.2%	+20.7%
Write comprehensive test cases for my own code	51.4%	71.6%	+20.2%
Write comprehensive test cases for code written by other people	28.6%	62.7%	+34.1%
Write comprehensive test cases based on a description of a function, without seeing the implementation (code)	32.9%	36.6%	+3.7%

Table 4: Assessment: Intermediate Results

	Average (5 minutes)	Average (10 minutes)	Average (Final)
Untrained	21.80	26.90	30.65
Trained	26.80	32.55	36.53
Delta	+5.00	+5.66	+5.87

Table 5: Explicit Feedback

"The training exercises improved my ability to..."	% Agree
Write comprehensive test cases	73.1%
Read code written by other people	82.1%
Understand multiple approaches to the same problem	91.0%

Interestingly, examining the scores from the five- and ten-minute checkpoints revealed that the gap between groups increased with time. As Table 4 shows, the trained group averaged a 5.00 point advantage at five minutes; by ten minutes, this had grown to 5.66 points, and by fifteen it had expanded to 5.87 points. One possible explanation is that the untrained students more quickly approached a point of diminishing returns, while the trained students were able to continue reasoning productively about the input domain.

Further, students reported increased confidence on a variety of testing-related skills. For each statement in Table 3, students rated their confidence using a four-point Likert scale (Strongly Disagree, Disagree, Agree, Strongly Agree) at the beginning and end of the study. In every case, the percentage of students who responded with Agree or Strongly Agree increased. Most notably, the percentage of students who felt confident in their ability to write comprehensive test cases increased from 34.5% to 55.2%, those who felt confident in their ability to test their own code increased from 51.4% to 71.6%, and those who felt confident in their ability to write comprehensive test cases for code written by others more than doubled, from 28.6% to 62.7%. Students also indicated notably higher confidence in their ability to reason about the input domain.

Surprisingly, a fairly small percentage of students felt improved confidence in their ability to write test cases without actually seeing the code they were testing. However, the results of the assessment, which asked students to do exactly that — develop a black-box test suite —, indicate that this lack of confidence was unfounded.

When asked to provide explicit feedback on the training exercises using the same four-point Likert scale, the students strongly corroborated their effectiveness. Over 73% of students agreed that the exercises improved their ability to write comprehensive test cases. Moreover, students reported auxiliary benefits of the training exercises. Over 82% of students felt increased confidence in their ability to read code written by others, and over 91% of students felt better equipped to understand multiple approaches to the same problem — both hugely valuable skills in software development.

6. CONCLUSION

This paper has presented an interactive system for helping students learn to develop better test suites. The system was evaluated within an introductory class, and has been shown to improve students' testing abilities by a statistically significant degree. Given the benefits the students derived from the system, we plan to continue to use it in our introductory class in the future to teach testing.

Three key features contribute to the system's effectiveness. First, students are not testing their own code. This allows them to focus on developing test cases, rather than worrying about bugs in their own code. Second, the submitted test suites are evaluated based on how many buggy programs they detect from a large corpus of programs. This gives a more objective measure of test quality than many other metrics such as code coverage. Finally, students are given instantaneous feedback on their submitted test cases in the form of programs that contain bugs which are undetected by their tests. This enables students to reason about the quality of their test cases and to understand why they are not sufficient.

Students who trained with our system were able to construct better test cases than those who had not: on an assessment that required students to develop black-box tests for a natural language specification, the average performance of the trained group was nearly a full standard deviation higher than that of the untrained group. This indicates that our system teaches students to think about testing in a more comprehensive way without continuing to need to see example implementations. Further, the students recognized the improvements in their abilities, reporting increased confidence in their ability to write comprehensive test cases.

The nature of the feedback system also provides additional benefits. Examining buggy programs not only helps students to develop better test cases, but also gives them practice reading, understanding, and debugging code written by others. Similarly, being exposed to these buggy programs helps students to reason about a variety of different solution strategies to the same problem. These are both extremely valuable skills for software developers to have.

Testing is a critical part of the software development process. Despite this, most computer science curricula devote little time to teaching students how to test. Therefore, it is essential to develop new systems and techniques for integrating testing into the curriculum. This paper has introduced one such system and evaluated it within an introductory course, demonstrating how the combination of automation and interactive feedback can quickly and effectively yield striking improvements in student testing capabilities.

7. ACKNOWLEDGEMENTS

This material is based upon work supported by NSF Award CCF-1320860: "Computer-Aided Grading, Feedback, and Assignment Creation in Massive Online Programming Courses" as well as an NSF Graduate Research Fellowship under Grant No. 1450681.

8. REFERENCES

[1] radon. https://pypi.python.org/pypi/radon.

[2] K. Aaltonen, P. Ihantola, and O. Seppala. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM SIGPLAN Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2010.

[3] D. S. Alberts. The economics of software quality assurance. In *Proceedings of the 1976 National Computer Conference and Exposition*, 1976.

[4] E. F. Barbosa, J. C. Maldonado, R. LeBlanc, and M. Guzdial. Introducing testing practices into objects and design course. In *Proceedings of the 16th IEEE Conference on Software Engineering Education and Training*, 2003.

[5] E. F. Barbosa, M. A. G. Silva, C. K. D. Corte, and J. C. Maldonado. Integrated teaching of programming foundations and software testing. In *Proceedings of the 38th ASEE/IEEE Frontiers in Education Conference*, 2008.

[6] E. G. Barriocanal, M.-A. S. Urban, I. A. Cuevas, and P. D. Perez. An experience in integrating automated unit testing practices in an introductory progamming course. *ACM SIGCSE Bulletin*, 34(4), 2002.

[7] C. C. Bonwell and J. A. Eison. *Active Learning: Creating Excitement in the Classroom*. ASHE-ERIC Higher Education Report No. 1., Washington, D.C.: The George Washington University, School of Education and Human Development, 1991.

[8] D. Carrington. Teaching software testing. In *Proceedings of the 2nd ACM SIGCSE Australasian Conference on Computer Science Education*, 1997.

[9] D. Carrington. Teaching software design and testing. In *Proceedings of the 28th ASEE/IEEE Frontiers in Education Conference*, 1998.

[10] T. Y. Chen and P.-L. Poon. Experience with teaching black-box testing in a computer science/software engineering curriculum. *IEEE Transactions on Education*, 47(1), 2004.

[11] D. M. de Souza, S. Isotani, and E. F. Barbosa. Teaching novice programmers using progtest. *International Journal of Knowledge and Learning*, 10(1), 2015.

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4), 1978.

[13] A. Derezinska and K. Halas. Experimental evaluation of mutation testing approaches to python programs. In *Proceedings of the 7th IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2014.

[14] S. H. Edwards. Improving student performance by evaluating how well students test their own programs. volume 3, 2003.

[15] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, 2004.

[16] S. H. Edwards and Z. Shams. Comparing test quality measures for assessing student-written tests. In *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014.

[17] S. H. Edwards, Z. Shams, M. Cogswell, and R. C. Senkbeil. Running students' software tests against each others' code: New life for an old "gimmick". In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, 2012.

[18] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Prorgams: An Introduction to Programming and Computing*. MIT Press, 2010.

[19] S. Frezza. Integrating testing and design methods for undergraduates: Teaching software testing in the context of software design. In *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference*, 2002.

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

[21] N. B. Harrison. Teaching software testing from two viewpoints. *Journal of Computing Sciences in Colleges*, 26(2), 2010.

[22] M. Hauswirth, D. Zaparanuks, A. Malekpour, and M. Keikha. The javafest: A collaborative learning technique for java programming courses. In *Proceedings of the 2008 International Conference on Principles and Practices of Programming on the Java Platform*, 2008.

[23] T. B. Hilburn and M. Townhidnejad. Software quality: A curriculum postscript? *ACM SIGCSE Bulletin*, 32(1), 2000.

[24] J. J. Li and P. Morreale. Enhancing cs1 curriculum with testing concepts: A case study. *Journal of Computing Sciences in Colleges*, 31(3), 2016.

[25] H. Liu, F. Kuo, and T. Chen. Teaching an end-user testing methodology. In *Proceedings of the 23rd International IEEE Conference on Software Engineering Education and Training*, 2010.

[26] W. Marrero and A. Settle. Testing first: Emphasizing testing in early programming courses. In *Proceedings of the 10th Annual ACM Conference on Innovation and Technology in Computer Science Education*, 2005.

[27] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 4, 1976.

[28] C. Meyers and T. B. Jones. *Promoting Active Learning: Strategies for the College Classroom*. Jossey-Bass Inc., Publishers, 350 Sansome Street, San Francisco, CA 94104, 1993.

[29] J. G. Politz, S. Krishnamurthi, and K. Fisler. In-flow peer review of tests in test-first programming. In *Proceedings of the 10th International Computing Education Research Conference*, 2014.

[30] M. Prince. Does active learning work? a review of the research. *Journal of Engineering Education*, 93(3), 2004.

[31] Z. Shams and S. H. Edwards. Toward practical mutation analysis for evaluating the quality of student-written software tests. In *Proceedings of the 9th International Computing Education Reseach Conference*, 2013.

[32] T. Shepard, M. Lamb, and D. Kelly. More testing should be taught. *Communications of the ACM*, 44(6), 2001.

[33] T. Tang, S. Rixner, and J. Warren. An environment for learning interactive programming. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 2014.

[34] T. Tang, R. Smith, S. Rixner, and J. Warren. Data-driven test case generation for automated programming assessment. In *Proceedings of the 21st Annual ACM Conference on Innovation and Technology in Computer Science Eduation*, 2016.