Using Peer Review to Teach Software Testing

Joanna Smith
University of Texas at Austin
joanna.smith@utexas.edu

Joe Tessler
University of Texas at Austin
joe.r.tessler@utexas.edu

Elliot Kramer
University of Texas at Austin
ejameskramer@gmail.com

Calvin Lin
University of Texas at Austin
lin@cs.utexas.edu

ABSTRACT

This paper explains how peer review can be used to teach software testing, an important skill that is typically not carefully taught in most programming courses. The goals of such peer review are (1) to frame testing as a fun and competitive activity, (2) to allow students to learn from each other, (3) to demonstrate the importance of testing by uncovering latent bugs in the students' code, and (4) to provide a mechanism for evaluating testing skills. This paper explains how we added peer review to an honors data structure course without significantly reducing its heavy programming load. We evaluate our intervention by summarizing surveys of student attitudes taken throughout the course.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Testing and Debugging

General Terms

Human Factors

Keywords

Peer Review, Education, Software Testing

1. INTRODUCTION

Software testing is a crucial component of the software lifecycle. A 2002 study by the National Institute of Standards and Technology reports that software bugs cost the US economy an estimated $59.5 billion annually and that more than a third of this cost could be eliminated by improved software testing [17]. Bill Gates agrees with the importance of software testing, saying, "[At Microsoft,] we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing [8]."

Unfortunately, there are several reasons why it is difficult to teach software testing. First, most students are not enthusiastic about testing [4]. Second, in most programming courses, testing is not an important aspect of a student's grade, perhaps because testing is a complex process that can be time-consuming to grade. Third, there is a self-fulfilling aspect to the view that testing is unimportant: Students who do not value testing are likely to produce poor test cases, so they do not learn that testing can effectively reveal bugs. Finally, testing is difficult to teach through lectures or through mechanical procedures [12]; instead, "the challenge ... is to develop group activities that can foster insight—a level of abstract understanding that can apply from situation to situation—rather than emphasizing detailed procedural understanding [12]."

The situation is further complicated if we wish to teach white box testing of moderately large and complex programs. As opposed to black box testing, which focuses on a program's externally observable behavior, white box testing adds richness and complexity; for example, it allows testers to create custom test harnesses that stress specific internal interfaces.

Peer review, or peer testing, in which students attempt to break code written by their peers, has the potential to address all of these problems. Because it is competitive, it can be fun and exciting. Because the peer reviews are graded, they can be weighted heavily in the student's grade. Because peer testing often reveals bugs in a student's program, it can illustrate the benefits of good software testing. Finally, because the peer reviews describe the reviewer's testing methodology and because students receive multiple peer reviews, the process exposes students to a variety of testing ideas, allowing students to learn from their peers.

In this paper, we describe a novel approach of incorporating peer testing into a lower-division programming course, where the testing includes white box testing of moderately complex software. We also explain how we address a number of issues that arise from such an activity, including issues of fairness, anonymity, academic integrity, and the increased workload.

More specifically, we add a peer testing component to two of seven programming assignments in a freshman honors data structures course. For these two assignments, students work in pairs, first to produce their solution and then to review four other solutions. The reviews are double-blind, and the goal of each review is not simply to identify bugs, but to provide insights as to the possible causes of the bugs and to explain the reviewer's testing methodology. Finally, each team submits a peer testing report that summarizes what they've learned from the process and that evaluates

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICER'12, September 9–11, 2012, Auckland, New Zealand..
Copyright 2012 ACM 978-1-4503-1604-0/12/09 ...$15.00.

the quality of the four reviews—and their underlying test methodologies—that they received.

We find that we are able to add these peer testing activities without reducing the heavy programming workload. We also find, through surveys taken throughout the course, that students enjoy peer testing, that students find peer testing valuable, and that peer testing increases their perceived ability to test software.

This paper is organized as follows. Section 2 places our work in the context of prior work, and Section 3 describes the setting for our experiment. Section 4 describes our peer testing intervention, and we evaluate the results in Section 5.

2. RELATED WORK

Peer review has been used in many teaching situations and has been found to have many benefits: It improves writing skills [14], develops critical thinking skills [5, 16], improves self-efficacy [2], and can provide detailed and careful feedback [16, 15]. In the context of software development, peer review has been used to critically comment on code quality [4] and to evaluate homework solutions using a web-based system [13]. Clark describes how interactive peer testing can be used to effectively review code in a third year capstone course: Peer teams are given 35 minutes to conduct both usability testing and code review, followed by 5 minutes to discuss their findings with one of the code's authors [4]. By contrast, our study explores the use of anonymous and more involved peer review, as students are given three days to write reviews and to describe their test methodology.

Peer review bears some similarities to collaborative learning methods, in the sense that both attempt to engage students to participate actively in the educational process. For example, in JavaFest [11], groups of students compete to design the optimal solution for a provided programming problem. With respect to software testing, most prior work on collaborative learning focuses on test input generation, which applies to black box testing but does not extend to richer forms of white box testing. For example, Carrington [3] describes exercises in which students generate test inputs for various software specifications, and Goldwasser [9] describes an approach in which students competitively submit test cases to break each others' code. However, Goldwasser's approach focuses on black-box testing methods, where students explore testing through small-scale programming exercises, such as merging two sorted linked lists. By contrast, our approach allows students to explore both white-box and black-box testing.

Testing has also been taught in the context of test-driven development, an entire software development methodology in which test cases are written first [6, 7]. Our work instead teaches software testing without asking students to adopt any specific software development methodology.

Finally, testing has been the subject of entire courses. Harrison [10] describes an upper-division software testing course that emphasizes two disparate testing roles—that of the developer and that of the tester. Thus, students serve as developers of one project and testers of a different project. In their role as testers, student perform peer review, but the emphasis of the course is on the importance of distinguishing between these two testing roles.

Response	Count
Not competent	13
Probably not competent	13
Probably competent	17
Competent	8

Table 2: Pre-course Survey: "Are you competent at software testing?"

3. BACKGROUND

The setting for our intervention is the Fall 2011 offering of CS314H, an honors data structure course at The University of Texas at Austin. The students in this course are predominantly freshmen honors students who are talented and highly motivated. Nevertheless, the instructor, who has taught this course for ten years, has been largely dissatisfied with his students' ability to acquire software testing skills, despite his efforts to stress the importance of testing, to introduce various testing ideas during lectures, to mandate that students explain how they have tested their programs, and to devote time in discussion sections (led by the teaching assistaant) to share ideas about how the programs could be tested.

This course has a heavy programming workload. There are seven Java programming assignments. The assignments start small but quickly grow in both scope and complexity. Table 1 summarizes these assignments, showing those for which they should do Pair Programming, and showing the alloted time for each assignment.[1] (The alloted time for Assignment 7 includes the Thanksgiving holiday.)

The Fall 2011 instance of this course had 51 students, 1 TA, and 1 undergraduate grader who worked 20 hours per week. On the first day of class, students were given an anonymous survey which asked various questions about their background. Included in the survey was one open-ended question, "Are you competent at software testing?" We classified their responses into four categories, and the results are shown in Table 2; we conclude that students enter the course largely unsure of whether they are competent software testers.

4. OUR SOLUTION

Our solution has three main goals. First, we want to make testing fun and competitive so that students will put effort into testing. Second, we want students to learn from each other, so that they can see how others approach the same problem, perhaps with a greater degree of creativity than they have. And third, we wish to illustrate the tangible benefits of good software testing by uncovering latent bugs in their code.

4.1 The Process

Before describing our peer review process, we first specify five requirements for our solution.

1. The process should be double-blind, which removes bias and allows reviewers to be honest in their analysis and feedback without the fear of offending a friend.

[1] Pair programming is a software development methodology in which two students share a computer to collaborate on all aspects of the assignment.

Assignment	Name	Description	Pairs?	Alloted Time
1	Image Manipulation	Manipulate digital images	No	7 days
2	Random Writer	Use Markov process to generate text that is similar to some corpus of text	No	9 days
3	Critters	Write interpreter for simulate creatures	Yes	14 days
4	Tetris	Implement Tetris game	Yes	14 days
5	Boggle	Implement Boggle game	No	14 days
6	Treaps	Implement Treaps data structure	Yes	9 days
7	Web Crawler	Implement web crawler and search engine	No	21 days

Table 1: The seven programming assignments in previous versions of CS314H.

2. The process should encourage students to take the peer review process seriously, which will increase the chance that students will learn from each other.

3. The process should be fair in its assignment of reviewers to reviewees. Each team should get to review a representative cross section of the class's solutions, as opposed to, for example, only testing the best solutions and finding very few bugs. Similarly, each team should see reviews from a representative cross section of the teams, so that they are more likely to receive some reviews that use good test methedologies.

4. The process should preserve academic integrity by not allowing students to read or copy each other's code.

5. The process should allow rich test methodologies beyond black-box testing, including the ability to do unit testing and to create custom test harnesses.

Our peer review process can now be described in terms of three deadlines that the student teams must meet.

1. At the first deadline, teams submit their solutions to the assignment. The solutions are then anonymized and obfuscated. Four reviewers are then assigned to each team's solution.

2. At the second deadline, teams submit their peer reviews, which include a description of their test methodology and of their findings. These reviews are then anonymized and distributed to the reviewees.

3. At the third deadline, teams submit their Peer Testing Reports, which describes what they learned in this process and also evaluates the peer reviews that they have received. At this point, teams may also submit revised solutions that fix bugs that have been pointed out by peer reviews.

We can now provide details about how our process meets our stated requirements.

To implement a double-blind review process, we use a one-way mapping from a student's ID to a number. These numbers serve as identifiers, ensuring that the students are not aware of who they are interacting with in the process. By giving reviewers obfuscated byte code, reviewers are shielded from any identifying comments or tell-tale stylistic quirks.

To encourage students to think about the quality of the various peer reviews and to encourage students to write good reviews, each reviewee grades each review. The instructional staff also grades each review. In sum, the peer review represents 50% of the grade for Assignments 5 and 6.

To ensure a fair assignment of reviewers, we first place the teams into quartiles based on each team's performance thus far in the course, and we then randomly assign reviewers from each quartile so that each team's solution is reviewed by one team from each quartile. In our case, our 26 teams did not divide 4 evenly, so our process could only guarantee that each team was reviewed by teams from at least three different quartiles.

We preserve academic integrity and anonymity by obfuscating the code before compiling it into bytecode. Only the bytecode is distributed, and if a student does reverse the compilation, the code is extremely difficult to read and easily identifiable as obfuscated code. We employ a Java obfuscator called Smokescreen [1] for this purpose.

We encourage rich testing methodologies by giving students access to byte code with well defined interfaces, which allows them to create test harnesses and to perform unit testing on individual methods of well-defined interfaces. In addition, because the students are creating peer reviews of the same projects that they themselves are asked to complete, the same test harness that they use for peer testing can be directly applied to their own code.

4.2 The Reviews

Each review has two components. The first is an evaluation of the reviewee's code for bugs, and the second is a description of the testing methodology. The first component is further subdivided into two categories, namely, a summary of the most important findings and detailed comments that describe specific test cases and bugs. The second component helps the students learn from good reviewers. It also helps the reviewees understand the bugs that were identified in the peer review, and it helps the reviewee in evaluating the peer reviews.

By having the students grade each other's reviews, we hope to to make them accountable to their peers. We also hope to encourage their natural competitiveness, as teams will want to find more bugs than the other reviewers of the same code.

4.3 The Choice of Programming Assignments

Assuming that peer testing is not incorporated into every programming assignment, there are two factors to consider in deciding where it should be incorporated: (1) the timing of the assignment within the semester and (2) the suitability of the assignment for peer review.

The issue of timing is not clear cut. On the one hand, if

peer testing is introduced early in the course, it may provide lasting benefits for the remainder of the course. On the other hand, if too many students have poor testing skills, then peer review may not be fruitful, so it may be useful to delay peer review until students have learned some basic concepts, terminology, and techniques as applied to some early, simpler programming assignments.

The issue of suitability is more clear cut. An ideal candidate for peer review is a programming assignment with a well-defined interface with plenty of freedom in the implementation. The well-defined interface ensures that students will be able to access important methods, allowing for deeper testing beyond black-box testing of the overall program. The freedom of implementation encourages a variety of development and testing strategies.

4.4 Challenges

Our peer testing activities introduces two challenges with respect to workload.

First, peer testing increases the student workload. They are being asked to do more testing, more writing, and more critical thinking. In particular, the students need to write the actual peer reviews, along with two additional reports for each peer-tested assignment.

Second, peer testing as we have defined it introduces strict deadlines, where each tight deadline builds upon the previous one. The process fails if large numbers of students do not meet their deadlines. For example, if a team does not submit their solution on time, their peers may not have sufficient time to conduct their reviews. Worse, if a team submits code that does not compile or is badly dysfunctional, then their peers cannot meaningfully test it. Similar issues arise from missing the second deadline.

4.5 Implementation Details

Choice of Assignments.

We add peer testing to Assignments 5 and 6, which occur late in the course. We do not introduce peer review earlier because we first use lectures, discussion sections, and the early assignments to teach the students various testing concepts, such as black-box testing, white-box testing, unit testing, and the notion of a test harness, as they apply these concepts to simpler programs.

In addition, both Assignment 5 and 6 are ideal for peer testing because they give students great freedom in implementing well-defined interfaces and they provide testing challenges. In particular, the solutions to Assignment 5 typically make heavy use of recursion, which is a confusing topic for many students. Assignment 6 requires students to use randomization to produce balanced trees, and the random behavior introduces testing challenges.

We choose to not add peer testing to Assignment 7 because it is designed to evaluate all of the skills that each student has acquired over the course of the semester. Thus, this assignment is completed individually. In addition, this assignment is the most challenging and open-ended of the assignments, which makes it a poor candidate for peer testing, because the behavior and interfaces are not well defined.

Pair Programming.

We allow students to work in pairs on these assignments both because it reduces the increased workload and because it is often helpful to be able to discuss ideas with a partner, particularly when developing relatively new skills.

Changes to the Schedule.

To preserve the number of programming assignments in our course, we add peer testing to Assignment 5 while retaining the 14 days that are allocated for this assignment. (See Figure 1.) Thus, the additional five days of peer testing shrinks the amount of time that students have to complete the assignment. To mitigate the effects of this shrinkage, we allow students to work in pairs, whereas students in previous classes worked individually (see Table 1).

Because Assignment 6 is one that has always been done in pairs, we simply add 5 days for peer testing to the end of the original schedule. As a result, the deadlines for Assignment 6 are identical to those for Assignment 5, ie, there are 9 days until the first deadline, then 3 days until the second deadline, and then 2 days for the final deadline. To preserve the overall course schedule, we remove a five day gap that formerly existed between in the schedule between Assignments 4 and 5.

Submission Logistics.

The entire peer testing process is made easy by the existence of Google Docs—specifically Google Forms. With basic scripting, we easily map the students to a number and distribute the bytecode via email. All reviews are then submitted online via forms generated by another script that automatically fills in the proper reviewer and reviewee ID number. The student grades for a review are submitted in a different form with the same scripting algorithm. By using Google Forms, the instructor can easily evaluate all reviews and grades in a single spreadsheet.

5. EVALUATION

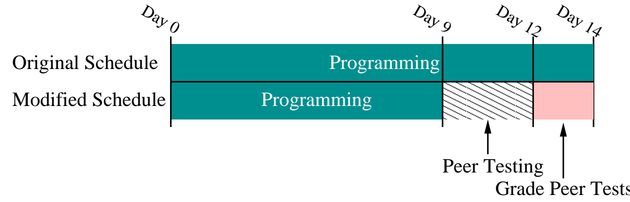
5.1 Methodology

There are several possible methods of evaluating our peer review intervention. One method is to split the class in half, so that one group acts as a control group against the peer testing group. We discard this approach as being potentially unfair, since one group could gain an advantage over the other that could impact their final grade. Another option is to split the class in half and alternate their use of peer testing, but this approach runs the risk of a spillover effect, where those who do peer review earlier may perform better on the next assignment even without peer review. Another method is to use two classes with similar populations where one is the control, but for logistical reasons, two such courses were difficult to find.

We choose instead to introduce peer review to one entire class at the same time and to focus on observing changes in student attitudes over the course of the semester. To track the students' attitudes and self-perceptions, we conduct surveys at the end of each assignment. The questions remain constant throughout the semester, and are brief, simple statements with which students can Strongly Disagree, Disagree, Agree, or Strongly Agree. Each survey also includes an open text box that allows students to provide arbitrary comments.

5.2 Survey Results

Figure 1: Modifying the schedule for Assignment 5 to include peer testing.



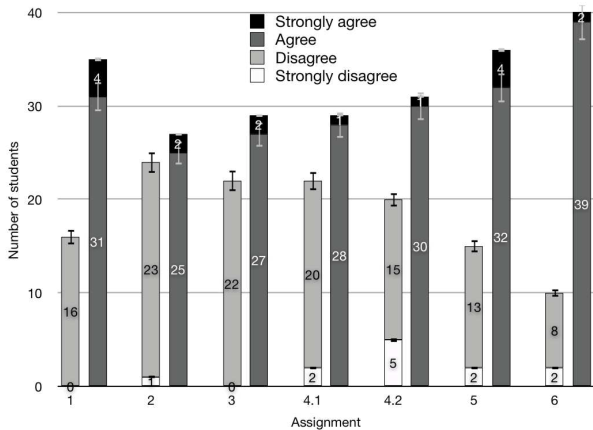
We observe that early in the semester, students believe that they are good testers (See Figure 2). These results differ from the pre-course survey, where students express skepticism in their testing abilities, and we conjecture that because the surveys are taken immediately after students submit their programs, these confident responses refer to the students' perceived ability to test the just-submitted assignments.

As the semester progresses, we see that student confidence drops. Here, we believe that as the assignments become more difficult and as students receive low testing scores on previous assignments, students begin to understand the limits of their abilities.

Beginning with Assignment 4, testing confidence begins to rise, and confidence rises further after Assignments 5 and 6, which incorporate peer testing. These changes in attitude suggest that peer testing has been successful.

Over the course of the semester, we see similar trends for the question "I like testing software" (see Figure 3).

Figure 2: I'm good at testing software.

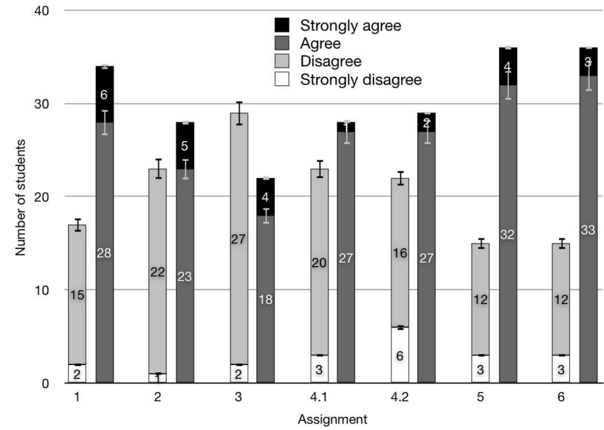


Error bars are represented by the vertical lines overlapping each bar.

We also observe that the students both like and believe that they learned from peer review, as is shown in Figures 4 and 5. These results show that the students believe that peer review is an effective use of their time and that it engages them in the process of learning to test software.

One common comment is: "It was fun to try to break other people's code." Another common comment is something akin to "they definitely helped us catch some good bugs that we missed." On the negative side, students often complain that peer review would be easier if each reviewee's assumptions were clearly stated, and if output was standardized.

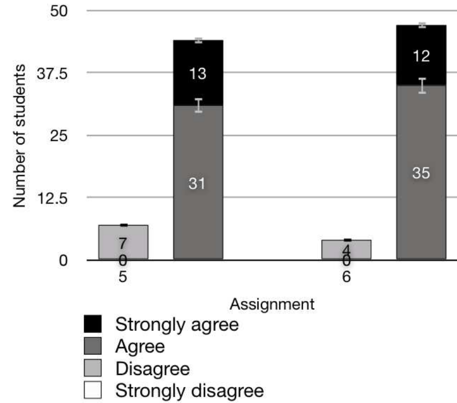
Figure 3: I like testing software.



Error bars are represented by the vertical lines overlapping each bar.

Finally, several students indicate that they wanted to do peer review for all of their remaining assignments.

Figure 4: I liked peer review.



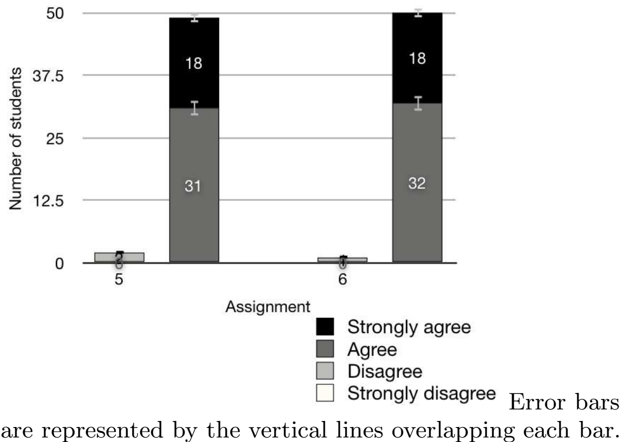
Error bars are represented by the vertical lines overlapping each bar.

5.3 Other Results

With a decade of experience, we believe that the heavy workload in this course causes some students to miss deadlines, particularly late in the semester.[2] Hence, the need to adhere to strict deadlines for our peer review activities was an early concern. However, in this case study, only two teams submitted late solutions for Assignment 5. In one

[2]Unfortunately, we do not have the detailed historical records needed to definitively support this claim.

Figure 5: I learned something through peer review.



Error bars are represented by the vertical lines overlapping each bar.

case, the delay was minimal, a matter of hours, and they were able to fully participate in the remaining peer review. In the other case, the team was a day late and received no peer review. Remarkably, for Assignment 6, all deadlines were met. For Assignment 7, which did not include peer review, one student submitted an assignment one day late, while another submitted an assignment two days late. We conjecture that the power of peer pressure helps students meet deadlines, a point that has been observed elsewhere [4], but our main observation here is that we were able add both extra work and additional deadlines without creating additional missed deadlines.

6. CONCLUSIONS

Software testing is a subject that can be difficult to teach, perhaps because it relies heavily on experiential learning; at the same time, because it is an activity that most students do not enjoy, students tend to expend minimal effort on testing. In this paper, we have described our experience in incorporating peer testing into a course with a heavy programming component and a tight schedule. We were able to do so without removing or significantly simplifying any of the programming assignments. Our results show that despite the extra work, the vast majority of students enjoyed peer testing and found it worthwhile, and many students expressed an interest in doing additional peer testing. The larger point, of course, is that students are often willing to do more work if the extra effort comes in the form of enjoyable activities that show tangible benefits.

Peer testing as we have described it imposes additional burdens on the student. In our experience with honors students, the students welcomed these additional burdens, but it would be interesting to see if peer testing would achieve similar results with a more general student population.

With our encouraging results, we would like to see further study in the use of peer testing in introductory programming classes, ideally studies that use a properly selected control group. Another avenue for future work is to explore the conjecture that with improved ability to test software, students produce software with fewer defects.

7. ACKNOWLEDGMENTS

We thank George Veletsianos and Bill Press for their valuable comments on this work. This work funded in part by NSF grant CNS-1138506.

8. REFERENCES

[1] Smokescreen. *http://www.leesw.com/smokescreen/*.

[2] K. Anewalt. Using peer review as a vehicle for communication skill development an active learning. *J. Comput. Small Coll.*, 21(2):148–155, 2005.

[3] D. Carrington. Teaching software testing. In *ACSE '97, the 2nd Australasian Conference on Computer Science Education*, pages 59–64. ACE, 1997.

[4] N. Clark. Peer testing in software engineering projects. In *ACE '04, the 6th Australasian Conf. on Computing Education*, volume 30, pages 41–48. ACE, 2004.

[5] R. Davies and T. Berrow. An evaluation of the use of computer peer review for developing higher-level skills. *Computers in Education*, 30(1):111, 1998.

[6] C. Desai, D. Janzen, and K. Savage. A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin*, 40(2):97–101, June 2008.

[7] S. H. Edwards. Teaching software testing: Automatic grading meets test-first coding. In *OOPSLA '03, the 18th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 318–319. SPLASH, 2003.

[8] J. Foley and C. Murphy. Q&A: Bill Gates On Trustworthy Computing. *Information Week*, May 2002.

[9] M. H. Goldwasser. A gimmick to integrate software testing through curriculum. In *SIGCSE '02, the 33rd SIGCSE Technical Symposium on Computer Science Education*, pages 271–275. CSE, 2002.

[10] N. B. Harrison. Teaching software testing from two viewpoints. *Journal of Computing Sciences in Colleges*, 26(2):55–62, December 2010.

[11] M. Hauswirth, D. Zaparanuks, A. Malekpour, and M. Keikha. The javafest: A collaborative learning technique for java programming courses. In *PPPJ '08, the 6th Int'l. Symposium on Principles and Practice of Programming in Java*, pages 3–12. PPPJ, 2008.

[12] C. Kaner and S. Padmanabhan. Practice and transfer of learning in the teaching of software testing. In *CSEET '07, the 20th Conf. on Software Engineering Education & Training*, pages 157–166, 2007.

[13] E. Z.-F. Liu, S. S. J. Lin, C.-H. Chiu, and S.-M. Yuan. Web-based peer review: The learner as both adapter and reviewer. *IEEE Transactions on Education*, 44(3):246–251, August 2001.

[14] J. Liu, D. T. Pysarchik, and W. W. Taylor. Peer review in the classroom. *Bioscience*, 52(9):824–829, September 2002.

[15] E. Silva and D. Moriera. Webcom: a tool to use peer review to improve student interaction. *J. Educ. Resour. Comput.*, 3(1):1–14, 2003.

[16] W. J. Wolfe. Online student peer reviews. In *Proceedings of the 5th Conference on Information Technology Education*. ACM Press, 2004.

[17] W. E. Wong. Teaching software testing: Experiences, lessons learned and the path forward. In *24th Conf. on Software Engineering Education and Training (CSEE&T), 2011*, pages 530–534. IEEE, 2011.