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 our peer review are (1) to frame testing as a fun and competitive activity, (2) to allow students to learn from each other, and (3) to demonstrate the importance of testing by uncovering latent bugs in the students’ code. This paper describes our experience of introducing peer review to an honors data structures course that has a heavy programming load. We evaluate our intervention by reporting on student attitudes towards software testing throughout the course.
## Contents

1 Introduction .................................................. 3
2 Related work .................................................. 4
3 Background .................................................... 5
4 Solution ........................................................ 6
   4.1 Peer Review .................................................. 6
      4.1.1 The Process ............................................. 6
      4.1.2 The Reviews ............................................. 7
      4.1.3 Choosing Programming Assignments .................... 7
      4.1.4 Challenges .............................................. 7
      4.1.5 Injecting Peer Review .................................. 8
   4.2 Enhancing the Programming Assignments .................... 9
      4.2.1 Assignment 1 - Image Manipulation ................... 9
      4.2.2 Assignment 2 - Random Writing ....................... 9
      4.2.3 Assignment 3 - Critters ................................ 10
      4.2.4 Assignment 4 - Tetris .................................. 10
      4.2.5 Assignment 5 - Boggle .................................. 10
      4.2.6 Assignment 6 - Treaps .................................. 10
      4.2.7 Assignment 7 - Web Crawler and Search Engine ...... 11
5 Evaluation ..................................................... 12
   5.1 Methodology ............................................... 12
   5.2 Evaluation ................................................ 12
      5.2.1 Survey Results ......................................... 12
      5.2.2 Software Test Results .................................. 14
   5.3 Methodological Issues ..................................... 15
   5.4 Omitted Factors ........................................... 16
6 Conclusion .................................................... 17
7 Future Work .................................................. 18
Acknowledgements ............................................. 19
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 [16]. 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, most students have little enthusiasm for testing [4]. We believe that there are several reasons for this lack of enthusiasm. First, students often view testing as a tedious and mundane task. Second, in most programming courses, testing is not an important aspect of a student’s grade: Unless programs are tested thoroughly, grades typically do not depend on a student’s ability to thoroughly test and debug their code. Finally, 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 see how testing can effectively reveal bugs.

Furthermore, teaching testing is complicated by the difficulties in teaching richer forms of testing, such as white-box testing. Whereas black-box testing only requires students to examine the output for a given input into a program, white box testing requires students to test each component of the solution. It is simpler to teach testing on unit testing or in the context of a small program. The richer form of white-box testing introduces such concepts as an automated test harnesses to guarantee that every portion of the program is correct, and not just that the program is producing the expected output.

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 can be graded, they can be weighted heavily in the final grade. Because peer testing can reveal bugs in a student’s program, it can illustrate the benefits of good software testing. Finally, by asking that the peer reviews describe the reviewer’s testing methodology, students can be exposed to a variety of testing ideas and can learn from their peers.

In this paper, we describe how we introduce peer testing into an honors data structure course with a heavy programming component. We modify two of the seven programming assignments to include peer testing. For these two assignments, students work in teams of two, and after submitting their solutions, each team is assigned four other assignments for double-blind review. The goal of each review is to identify as many bugs as possible, to provide insights into the possible causes of the bugs, and to explain the reviewer’s testing methodology. These peer reviews are then evaluated by both the reviewees and the teaching assistant.

Surveys taken throughout the course reveal that students enjoy peer testing, that students find peer testing valuable, and that peer testing increases their perceived testing skills.

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 before summarizing and discussing future work.
2 Related work

Peer review has been used in many teaching situations and has been found to have many benefits: It improves writing skills [13], develops critical thinking skills [5, 15], improves self-efficacy [2], and can provide detailed and careful feedback [15, 14]. In the context of computer science, peer review has been used to critically comment on code quality [4] and to evaluate homework solutions using a web-based system [12], but peer review has not been used to teach software testing.

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 focused on black-box testing methods, where students explored testing through small-scale programming exercises, such as merging two sorted linked lists.

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].

Finally, testing has been the subject of entire courses. Harrison [10] describes a software testing course that includes exercises in which students design, execute, and report on test plans that they apply to a program written for a different project by another student.
3 Background

This paper will explain how we integrated peer testing into the Fall 2011 instance of an honors data structures course offered at University X. The students in this course are generally talented and highly motivated, but the instructor, who had offered the course for years, had been dissatisfied with his students’ ability to test software, despite his efforts to stress the importance of testing, to introduce various testing ideas during lectures, to mandate that students explain how they tested their programs, and to devote time in discussion sections (led by the Teaching Assistant) to share ideas about how the programs were tested.

This course has a heavy programming component, with students writing seven programming assignments in Java. The assignments start small and generally become more difficult as the semester progresses. They also start out well-specified and become increasingly open-ended as the semester progresses. Typically, 3 or 4 of the assignments are completed using Pair Programming, in which students are encouraged to work together (with one screen) to work on all aspects of the assignment, as opposed to dividing the assignment into two largely disjoint parts.

The Fall 2011 offering of this course had 51 students, 1 Teaching Assistant, 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?” The results are shown in Table 1.

<table>
<thead>
<tr>
<th>Competency</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not competent</td>
<td>13</td>
</tr>
<tr>
<td>Probably not competent</td>
<td>13</td>
</tr>
<tr>
<td>Probably competent</td>
<td>17</td>
</tr>
<tr>
<td>Competent</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 1: Pre-course Survey: “Are you competent at software testing?”
4 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 the students’ codes.

We implement this experiment with two main components: the introduction of a peer review element to the course, and a few enhancements to the existing CS 314H programming assignments.

4.1 Peer Review

Peer review aims to engage students through a competitive approach to testing and through a collaborative learning environment. While exploring this new learning style, students also have the added benefit of receiving feedback on how to improve their code.

4.1.1 The Process

There are several 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. (2) The process should encourage students to produce good reviews. (3) The process should be fair and minimize the likelihood that a team will receive reviews only from the weaker students in the class. (4) To preserve academic integrity, the process should not allow 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 test using new test harnesses.

A review process can achieve the first requirement by a one-way mapping from a student’s ID to a number. Rather than names or student IDs, these numbers serve as identifiers, ensuring that the students are not aware of who they are interacting with in the process.

The second requirement can be met by making the students accountable, perhaps by allowing the students to grade the reviews they receive on factors such as helpfulness and clarity.

To meet the third requirement, one solution is to rank the students by their performance thus far in the course, and then to randomly assign reviewers from several ranks. This ensures that each student receives fair feedback, as every student will receive some good reviews.

The fourth requirement can be achieved by obfuscating the code before distribution. Once obfuscated, the code is unreadable and easily identifiable if integrated into a student’s solution.

One approach to meeting the final requirement is to have students peer review the assignments they complete. With a defined interface that they work with in developing their solution, the students are able to employ deeper testing strategies as they have access to each method of the solution. Without having to work with the solution as a whole, students are able to explore the deeper layers of software testing.
4.1.2 The Reviews

There are two components to each review. 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 description of the testing methodology 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.

One of the goals of having the students grade each other’s reviews is to make them accountable to their peers and to encourage their natural competitiveness. In highlighting this, we hope the students will be driven to find more bugs than they would otherwise or more interesting bugs than simple errors.

4.1.3 Choosing Programming Assignments

Two main considerations in choosing a programming assignment for peer review are: the timing of the assignment in the semester and whether the assignment is amenable to and suitable for peer review.

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 each method, allowing for deeper testing beyond black-box testing for the entire assignment. The freedom of implementation encourages a variety of development and testing strategies.

The timing of the assignment should not be too early in the course because there needs to be sufficient time to introduce the testing techniques to the students. Throughout the course, the lectures introduce a variety of testing approaches as part of the curriculum, and there should be sufficient time before peer review for the students to learn these techniques.

4.1.4 Challenges

Our proposed peer testing activity introduces several challenges related to the workload.

First, peer testing increases the student workload. They are being asked to do more testing, more writing, and more critical thinking. In particular, in addition to the additional testing itself, 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. The process fails if large numbers of students do not meet their deadlines. In particular, if students do not submit their solutions on time, their peers may not have sufficient time to evaluate their programs. Similarly, if students submit code that does not compile or is badly dysfunctional, then their peers will have nothing to test.

In our particular case, we sought to add peer testing without reducing the programming workload. Thus, for the peer tested assignments, the students were given less time to complete their programming assignment than was typical for this course. One mitigating factor is the pressure that peer review exerts in motivating students to meet their deadline [3].
Third, there are challenges with respect to academic dishonesty. Students need to be able to test each other’s code without making it easy to copy their programs. Our solution is to obfuscate the source code before compiling it to bytecode. Students who decompile the bytecode are left with convoluted code that was clearly not produced by a human.

4.1.5 Injecting Peer Review

We chose two assignments for peer review, which happened to be assignments 5 and 6 in a course with 7 programming assignments. These assignments occur within the last six weeks of the course. For 5, the students write a Boggle game, which is a word game where players identify words from letters randomly arranged in a grid. For 6, they create an interesting binary data structure that uses randomization to provide balance, called a Treap. Both of these assignments are open-ended within a very specific interface, making them perfect choices.

We do not introduce peer review earlier because we first teach the students several testing principles, such as black-box testing, white-box testing, unit testing, and how to construct a test harness.

When choosing assignments to incorporate peer review with, we believe the assignment should be pair-programming. Many assignments in this course do allow students to work with a partner, and as peer review increases the burden of work for the students, it helps them to have a partner to discuss ideas with. Because of this, we modify assignment 5, which had always been individual, to now be pair-programming for peer review.

Ultimately, we do not continue with peer review for the final assignment, which is a web crawler, because it represents the culmination of the students’ ability to put together everything they have learned. This is the least-specific assignment, so they now work alone to show their skills. Without pair-programming, the assignment is not an ideal choice for peer review in our course.

To implement peer review, we create seven steps to the process, which can be viewed in Table 2. The “Day” of each step refers to the number of days that have passed since the project was assigned. For example, in the past, students had 14 days from receiving the handout to submit a solution. In our use of peer review, students now had 9 days before submission, and peer review spanned the remaining 5 days.

Our review process is double-blind via a one-way mapping from a student’s ID to a number. All code is referenced by this number, and all reviews by the number of the reviewee with the number of the reviewer.

We encourage the students to write good reviews by adding a reviewee’s grade to the instructor’s grade for each review, making the reviewers doubly accountable.

To ensure fairness in the quality of reviews for each student, we assign reviewers to reviewees by first sorting the class into four quartiles of students. These quartiles are determined by the students’ performance thus far in the course. The goal is to randomly assign reviewers from each quartile so that each student receives a fair range of feedback. However, this approach is affected by whether the number of teams divides evenly into 4. For our purposes, we aim for representation of a minimum of three of the quartiles.

We preserve academic integrity by obfuscating the code before compiling it to bytecode. Only the
<table>
<thead>
<tr>
<th>Step</th>
<th>Day</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>Students submit their solutions</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>The solutions are anonymized and obfuscated</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>4 reviewers are assigned to each solution</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>Students have 3 days to complete the reviews</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>The reviews are anonymized and distributed</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>Students grade the reviews they receive</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>Students have 2 days to improve their code and resubmit</td>
</tr>
</tbody>
</table>

 bytecode is distributed, and if a student does reverse the compilation, the code is unreadable and easily identifiable if integrated into their solution. We employ a Java obfuscator called Smokescreen [1] for this purpose.

We encourage rich testing methodologies by having students peer review the assignment they complete. Having already designed a solution within the provided interface, the students are poised to use their existing test harnesses when reviewing their classmates’ code, however extensive that harness may be.

This 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. Then all reviews are 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 done by a different form with the same scripting algorithm. By using Google Forms, the instructor is able to evaluate all reviews and grades with ease in a single spreadsheet.

### 4.2 Enhancing the Programming Assignments

To highlight the benefits of good software testing, and to stress the importance of software testing, we elevate the significance of testing in the course. This section highlights the differences in the programming assignments between 2011 and previous years.

#### 4.2.1 Assignment 1 - Image Manipulation

Assignment 1 is a basic image manipulation program, where the students implement alteration effects. In previous years, the sole reference to testing came as a question to describe the testing approach in the final project report. This year, we create a testing component in the assignment, where we provide a set of basic images for use with black box testing.

#### 4.2.2 Assignment 2 - Random Writing

Assignment 2 is a random writing program that generates text by computing and storing the probabilities of characters following a given character in a training text. The students learn that with more given characters, the randomly chosen character forms a closer approximation to the training text. Once again, we add a testing component that introduces white box testing, which is facilitated by creating
new methods that make the interface more accessible than the existing methods, by ‘breaking open’ the method into smaller components.

4.2.3 Assignment 3 - Critters

Assignment 3 introduces pair programming, as the students write a simple interpreter that translates a set of ‘critter’ instructions into actions in a simulated CritterWorld, and where the students create a test harness for their interpreter. Previous versions of this assignment strongly encourage the use of a test harness, but for 2011, it became mandatory, comprising one-third of the final assignment grade. In this test harness, students have the chance to employ unit testing to confirm the correctness of each instruction interpretation.

4.2.4 Assignment 4 - Tetris

For assignment 4, the students again work in pairs to implement the game of Tetris, in a method that emphasizes the idea of decomposing a large problem into smaller problems that can be independently tested. The assignment is broken into two parts. The first creates the game pieces. And the second requires building the board, implementing the game functionality, creating a ‘Brain’ that plays automatically, and an ‘Adversary’ that chooses the least optimal next piece for the ‘Brain.’ This assignment also includes testing milestones, which require the students to construct thorough test harnesses. This assignment remains unchanged from previous years.

4.2.5 Assignment 5 - Boggle

Assignment 5 involves the implementation of the game of Boggle, which is a word game where players identify words from a random assortment of letters on a grid. The implementation requires the design of a number of recursive algorithms, as well as the consideration of various implementation strategies for the dictionary.

In the past, this assignment has been individual. However, this year, it is a pair programming assignment, due to the introduction of peer review for this project. In addition, we include a new method in the interface, designed to inspire the students to remove the random elements from the program before testing.

4.2.6 Assignment 6 - Treaps

For assignment 6, the students work in pairs once more to implement a unique data structure called a treap. The key challenge is to carefully and thoroughly test the data structure for correctness. Once again, we introduce peer review to aid the students in their testing.

This year, as opposed to previous years, we again provide the students with a new method in the interface to remind them to remove the random elements from their programs for testing.
4.2.7 Assignment 7 - Web Crawler and Search Engine

This final assignment asks the students to individually create a web-crawler and a search engine. There are three core components to this assignment: to crawl a portion of the web, to build an index that allows for quick access of this portion of the web, and to respond to various queries or web searches.

We make no changes this assignment compared to previous years, save for the addition of “Words of Wisdom” to remind the students to manage their time carefully at the end of the peer review process. We include this advice to help students realize the increased complexity of this final assignment, since they are now working alone for the first time since assignment 2.
5 Evaluation

5.1 Methodology

When determining how to evaluate our peer review intervention, we consider a few different options. One method is to split the class in half, so that one group will act as a control group against the peer testing group. We deemed this approach to be unfair, as the peer testing students could gain an advantage over the others that could impact their grade. Or, it’s possible that the non-peer testing group could have an advantage by having more time to work on their programs. Another option would be to split the class in half and trade off when they do peer review, so that some students review the first assignment and the others review the following assignment. However, this runs the risk of a spillover effect, where those who review earlier stand to do better on the next assignment.

We decided that we would introduce peer review to the entire class at the same time and 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 that the students can Strongly Disagree, Disagree, Agree, or Strongly Agree with. At the end of each survey, the students are given an open text box and asked to provide any additional comments they have.

In addition, we try to get some indication as to whether our intervention is hurting or helping the students. To facilitate this, we try to compare the Fall 2011 students against previous classes. We believe this a fair comparison due to the course having the same professor every year, largely the same material, and the same acceptance criteria for the students from year-to-year, although there are variations from class to class.

However, it is not a perfect comparison due to the number of factors that changed for this course: peer review, the time allotted for assignments 5 and 6, the wording of the assignments, and the adjustment of assignment 5 to pair programming from individual programming.

But we want to examine the gross results, so we compare the number of tests successfully passed in an automated test suite ran on all code submitted for assignment 6 and assignment 3 since 2008. We want to evaluate if there is marked progress from 3, before peer review, to 6, with peer review, in 2011 that exceeds 2008-2010.

5.2 Evaluation

5.2.1 Survey Results

For several questions we observe that early in the semester students believe that they were good testers (See Figure 1). These results differ from the pre-course survey, where students expressed skepticism in their testing abilities, and we conjecture that because the surveys are taken immediately after students submit their programs, that these confident responses refer to their perceived ability to test the just-submitted assignments.

As the semester progress, 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 are
beginning 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 2).

In addition, we observe that the students both like and believe they learned from peer review, as is shown in Figures 3 and 4. These results show that the students believe that peer review was an effective use of their time, and are engaged in the process while learning software testing.

A common comment from assignment 5 was: “It was fun to try to break other people’s code.” Another that showed up in both assignment 5 and 6 was something similar to “they definitely helped us catch some good bugs we missed.” Both surveys had some negative feedback. Students often complained that it would have been easier to do peer review if each reviewee’s assumptions were clearly stated, and
if output was standardized. We created a venue for students to offer their assumptions to help their reviewers, and we are considering the testing benefits of standardizing all program output in the future. Several students indicated that they wanted to do peer review for all of their remaining assignments.

5.2.2 Software Test Results

After recording the number of tests successfully passed, by each student in 2008-2011, in the test harnesses for assignments 3 and 6, we run a one-way ANOVA calculation on the scores to determine if any year held statistical significance.

Unfortunately, this is not the case. Our results are inconclusive. The ANOVA P-values can be seen in Table ???. Because we are comparing 4 groups of data, the P-value would need to be less than 0.01 to indicate a significant difference. Because no year stands out, we cannot conclude anything about this comparison. For reference, the pairwise comparison P-values for assignments 3 and 6 are listed in Table 4a and Table 4b, respectively.
Ran over the number of software tests passed by each student’s code in 2008, 2009, 2010, and 2011.

Table 3: One-way ANOVA P-values

<table>
<thead>
<tr>
<th>Assignment</th>
<th>P-value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.029</td>
<td>Not significant</td>
</tr>
<tr>
<td>6</td>
<td>0.074</td>
<td>Not significant</td>
</tr>
</tbody>
</table>

Table 4: Pairwise Comparisons

P-Values from a One-Way ANOVA ran over the number of software tests passed by each student’s code 2008-2011.

What is interesting to note, though, is that the Fall 2011 students had 5 fewer days than 2008-2010 students did to write the code that was tested in that comparison, as most 2011 students chose not to resubmit an improved version of assignment 6 after peer review. Due to this lack of data, we ran the comparison with the original code submissions. In spite of adding much more material in the same amount of time, it does not appear that we have hurt the students—it has not led to worse programs. There is weak support that the students are getting better, but we do not know the exact cause for this, nor can we quantify this improvement at this time.

5.3 Methodological Issues

The main issue with this experiment is the lack of a control group leaves us with no true way to compare this year to previous years, due to the absence of surveys in the past. This was a conscious choice on our behalf, as we made a lot of decisions, some of which were constrained by making a good experience for the students, which we favored over doing the right thing scientifically.

Another issue arises from the time limitations of the deadlines necessary for peer review. This hinders the students, as they no longer have the option to turn in assignment 5 or 6 late, which is a noticeably common thing to do for these assignments. Even with the late penalty, many students make a significant improvement in that extra day that earns them points they would not have received by turning their code in on time. We underestimated the handicap we gave these students with these additional deadlines.

If we were to begin again, scientifically, the first thing we should do is to adjust all of the assignments to fit well with automated testing. This would allow for simple and comparative tests of every assignment across the years. Then we would begin the experiment, starting with a control group that filled out iterative surveys, but did not participate in peer review. The following year, we would introduce peer review in the same course, with no other modifications.

Another approach is to find a peer review process from another field that is transferable to software testing and implement that process on this course, comparing the changes noted to those recorded in that separate domain. This approach would provide insight into trends that are standard in peer review,
or unique to a given field.

5.4 Omitted Factors

For comparison to the CS314H students, we intended to conduct iterative surveys on another course, and an exit survey on the Fall 2010 CS 314H students. However, the class we attempted to reach out to was uninterested in the surveys, and we were unable to gather sufficient data for analysis. This would be an interesting method to try again in the future, though.

For a better cross-year comparison, we would have liked to have been able to compare the results of every assignment to the previous years of the course. However, due to time constraints and the difficulty of automated testing for certain assignments, we were unable to do so. This would be an interesting follow-up experiment over the summer, though, to see if any significant trends appear.

We also considered performing our own version of peer review on a sample of code from the previous years, to see how the reviews compared. The idea was to take some of the test harnesses submitted by the Fall 2011 students, and run them against the code from earlier years. Unfortunately, we did not have the time to build the framework required to be able to automate such a process. And we chose not to make this experiment a priority because we did not feel that the results would be as significant as running our own test suite against everyone’s code. But we still believe that it would most likely prove interesting.
6 Conclusion

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—without removing any existing 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 much more 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.
7 Future Work

The most pressing next step would be to repeat this experiment, with a control group from a prior year, so that cross-year comparisons could be fairly made.

Another thought is to expand peer review into another course, such as Software Engineering. Trying it with students who are older and have more experience with testing would provide interesting insight into how strong an effect peer review can have, or if it is best done earlier in the course of a computer science education.

A final interesting approach would be to track the progress of these CS 314H students throughout the honors program. Do they succeed in courses that are known to be challenging? Do more students make A’s in later classes? Do more finish the program? It would be interesting to begin tracking all students in the program, to see if an early testing education truly does set the students up for success in the future.
Acknowledgements

I would like to acknowledge my advisor, Dr. Calvin Lin, who introduced this project to me and allowed me to experiment on his class, in addition to providing invaluable insight from an instructor’s perspective. Also, this research would not have been successful in so short a time without the help of Elliot Kramer and Joe Tessler, who were instrumental in the redesign of an entire course in only a few months. In particular, I would like to thank them for taking the time to run the automated test suites that we designed, as I know that waiting for these tests is a frustrating experience. Their help is greatly valued.

In addition, I thank George Veletsianos and Bill Press for their valuable feedback on this work.
References


