Evaluating the Testing Ability of Senior-level Computer Science Students

Jeffrey C. Carver and Nicholas A. Kraft
University of Alabama
{carver , nkraft} @cs.ua.edu

Abstract

Testing is a key skill for computer science students to acquire during their studies. To determine how well students are learning this skill, we conducted an empirical study in two offerings of a senior-level computer science course. The goal of the study was to determine whether students would be able to create a small, complete test suite for a simple program. The students created a test suite first without the aid of a coverage tool and then with the aid of a coverage tool. The results indicate that without a coverage tool, students achieved significantly less than 100% statement, branch or condition coverage. When provided with a code coverage tool, students increased coverage levels. Still, examination of the test suites indicated that they were significantly larger than the minimum required. These results indicate that students cannot conduct adequate testing of even a small program. To provide context for our results, we provide a literature survey summarizing various techniques proposed for teaching testing in the computer science curriculum. We discuss each technique, its strengths, and its weaknesses.

1. Introduction and Background

Testing is vital to the development of quality software and can consume up to 50% of development effort [21]. To be successful practitioners, students must be able to test their software effectively. Based on our observations and on information reported in the literature, most students do not master the ability to test in a typical computer science (CS) curriculum. This lack of understanding of the testing process causes problems for the students, both during their formal education, as they develop software projects for courses, and during their professional careers.

During their educational careers, students often receive lower than expected programming assignment grades. One reason for incorrect expectations is the students' lack of testing knowledge and ability. The students often believe that running a small number of test cases will fully test their software. They do not foresee their programs failing some of the instructor's test cases or the resulting loss of points. Moreover, after their formal education, students are not properly equipped to enter the workplace. In many instances new hires in software companies spend time testing other developers' code before being allowed to contribute their own code. Therefore, adequate testing knowledge and ability are important for students' success. As Shepard et al. note, many students today are graduating with a knowledge gap about software testing [21].

Our goal is to address the following problem: Students, especially those in early CS courses, need help grasping the concepts of testing and help understanding how to fully test their software. To gather evidence for the veracity of the stated problem, we conducted an empirical study evaluating the testing ability of senior-level CS students. In particular, the main goals of this study were to evaluate:

1. the level of test coverage that senior-level CS/MIS/ECE students could obtain with and without the aid of a test coverage tool, and

2. the level of redundancy within a student's test suite (e.g. if a student achieves 100% coverage, but 50% of their test cases are redundant, then that student does not fully understand how to effectively test his software).

The remainder of the paper is organized as follows. Section 2 describes the empirical study. Section 3 reports the study results. In Section 4 we revisit the problem, discussing others' findings, along with how they have addressed the problem. We conclude in Section 5 by discussing the implications of the results and of our review of the literature.

2. The Study

The following subsections discuss the specific details of the study design.

2.1 Goals

One of the most important skills that software developers must acquire is the ability to appropriately test their software. It is our belief that senior-level students are not as capable in this area as desired. This belief is supported by anecdotal information from instructors here at UA as well as by the literature [19, 21]. As a result, the goal of this study was to gather some evidence about the testing ability of senior-level students.

2.2 Hypotheses

The overall study goal led to set of specific hypotheses that drove the design of the study and the definition of the data to be collected. First, if students fully understood how to test, they should be able to generate a test suite that provides full coverage (statement, branch and condition) of a relatively small, simple program. Anecdotally, we believed that students enrolled in this course would not be able to generate such a test suite, especially if they were given no guidance by a tool. Therefore, the first hypothesis is:

H1: Students enrolled in the software engineering course will not be able to fully test a given program (i.e. achieve 100% coverage) without support of a tool.

Second, we argue that if someone who correctly understands testing concepts is made aware that their test suite is incomplete, they should be able to systematically add test cases to increase coverage. That is, each additional test case should result in increased coverage. Conversely, someone who does not correctly understand testing concepts will use an *ad hoc* approach for adding test cases. That is, they will add test cases without a specific expectation of their effect on coverage. They will use the coverage tool to determine the effect. Sometimes the student is lucky and coverage increases. Other times, there is no effect on coverage. We believed that students who had not achieved 100% coverage manually would be able to increase coverage using a tool. We also believed that lack of testing knowledge would necessitate the use of an *ad hoc* rather than a systematic approach to adding test cases, resulting in larger than necessary test suites. Therefore, two additional hypotheses are:

H2: Students enrolled in the software engineering course will be able to obtain greater coverage when provided with a coverage tool.

H3: Students enrolled in a software engineering course will not be able to use information from a coverage tool to systematically improve their test suite to obtain a small yet complete set of tests.

2.3 Setting

Based on the study goal and hypotheses, we conducted the study in a senior-level software engineering course at the University of Alabama. This course provides a standard one-semester overview of the software engineering lifecycle (using the Pressman book) including a semester-long team project. The course is required for all computer science majors and an elective for management information systems and computer engineering majors (although it is required for those seeking a CS minor). We conducted the study during two semesters (Spring 2009 and Spring 2010).

We chose this course because it was late enough in the degree program to evaluate how well students learned testing concepts throughout the curriculum. Although testing is not formally covered until this course, students should have been exposed to testing concepts throughout their programming courses and should have been testing their software in these courses. Therefore, we were evaluating the students' knowledge about testing prior to the software engineering course. The computer engineering and management information systems majors enrolled in the course also had completed the introductory programming sequence and should have been exposed to the same level of testing concepts as their computer science counterparts.

2.4 Artifacts

The students tested one of the two short (~200 SLOC) programs developed for this study. The first program was a traditional I/O program (a calendar program taking a date as input and returning the date of the day before, the day after, one week before, or one week ahead). The second program implemented a state-based data structure (a deque). Both programs were written in Java 1.5 using comparable programming style.

2.5 Variables

There was one dependent variable and one independent variable in this study. The Dependent Variable, **test coverage,** was measured with three metrics: *statement coverage*, *branch coverage* and *condition coverage*. The Independent Variable, **use of coverage tool**, had two levels: *without tool* and *with tool*.

2.6 Procedure

The study consisted of two steps: *Step 1 – Manual Creation of Test Suite* and *Step 2 – Use of Code Cover to Improve Test Suite*. The students performed the steps during two successive class meetings. To reduce threats to validity, we randomly assigned half of the class to each program. Figure 1 illustrates the overall study design.

Step 1 –Manually Create Test Suite – During the first class period, each student was given a paper copy of the code for which they were to create a test suite. The students were instructed to create the "most-complete, yet smallest" test suite possible for their given program. The students created these test suites manually (i.e. without executing the code or using any type of coverage tool). At the end of the class period, the students submitted their test cases. The students were given up to 75 minutes (the length of the class meeting) to perform this task. Very few students took the entire time.

Step 2 – Systematically Modify Test Suite – Prior to Step 2, we ran each test suite from Step 1 through Code Cover (http://www.codecover.org), an open source coverage tool, to

obtain statement, branch and condition coverage. Then, during the second class meeting the students were trained on Code Cover. They were given their test cases from Day 1 and the coverage results from the Code Cover tool. They were told to use Code Cover to: 1) obtain 100% test coverage for all three metrics (statement, branch, and condition), and 2) eliminate redundant test cases (i.e. cases that do not increase coverage).

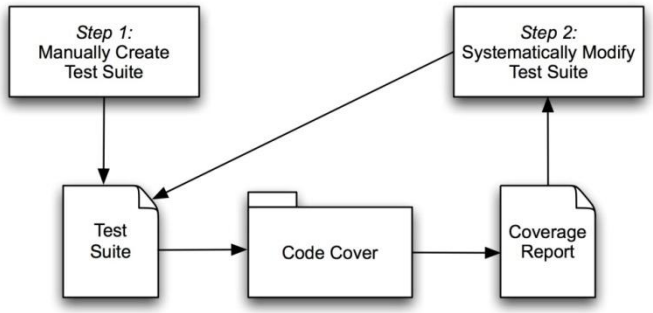


Figure 1 – Study Design

2.7 Data Collected

After Step 1 we collected each student's test suite and computed the three coverage metrics on it. During Step 2, each time a student executed Code Cover, we collected their current test suite and the generated coverage metrics. Thus, for each student we could track their test suite's evolution in terms of test cases added/removed and the percentage of statements, branches and conditions covered over time.

3. Results

The results section is organized around the three hypotheses. Throughout this section, we analyzed data from Study 1 (Spring 2009) and Study 2 (Spring 2010) separately. In all statistical analyses, an alpha value of .05 was used to judge the significance of the results.

3.1 H1 – Achieving 100% Coverage

Hypothesis 1 was *Students enrolled in the software engineering course will not be able to fully test a given program without support of a tool.* To evaluate this hypothesis, we used Code Cover to compute the levels of statement, branch and condition coverage achieved by the test suites submitted at the end of Step 1. Because the students were told to "fully test" the program, a perfect test suite would achieve 100% branch, statement and condition coverage. Due to an instrumentation issue, 100% coverage was not possible for the Calendar program, so the coverage obtained by each student was divided by the maximum possible (making 100% coverage still possible). Table 1 lists the results from Step 1. For each coverage type, the results of the one-sample t-test indicate the students achieved significantly less than 100% coverage. Notice that the standard deviation is much higher for the Branch and Condition coverage metrics than for Statement coverage.

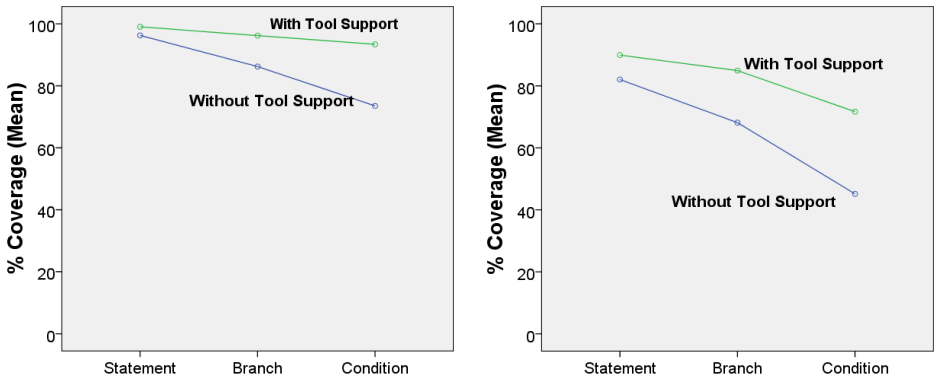
Based on this data, we can conclude that without tool support and without additional training, senior-level students cannot obtain 100% test coverage on a small program.

Table 1 -– Coverage after Step 1

	Study 1 (N = 12)			Study 2 (N = 10)		
	Statement	Branch	Condition	Statement	Branch	Condition
Mean Coverage %	96.26	86.23	73.52	82.04	68.13	45.15
Std. Deviation	3.749	12.835	20.843	11.738	17.207	29.166
t-value	-3.453	-3.717	-4.401	-4.839	-5.857	-5.947
p-value	.005	.003	.001	.001	< .001	< .001

3.2 H2 – Increasing Test Coverage

Hypothesis 2 was *Students enrolled in the software engineering course will be able to obtain greater coverage when provided with a coverage tool*. To analyze this hypothesis, we conducted a 3-way ANOVA with the following Independent Variables: 1) Study #, 2) Coverage Type and 3) Use of Coverage Tool. All three main effects were significant: Study ($F_{120,1}$ = 49.414; p < .001), Coverage Type ($F_{120,2}$ = 24.95; p < .001), and Use of Coverage Tool ($F_{120,1}$ = 32.864; p < .001). Figure 2 shows separate results for each study. Because of the significant effect of the Study variable and because the data in Table 1 indicates that participants in Study 2 were less effective than those in Study 1, we conducted separate 2-way ANOVAs for each study. In each case the effect of the Use of Coverage Tool variable was significant (Study 1 - $F_{65,1}$ = 16.794, p < .001; Study 2 – $F_{55,1}$ = 16.009, p < .001). This result indicates that when provided with a code coverage tool, students were able to significantly increase their branch and condition coverage rate.



(a) Study1 *(b) Study 2*

Figure 2 – Increase in Test Coverage

3.3 H3 – Systematically Increasing Test Coverage

Hypothesis 3 was *Students will not be able to use information from a coverage tool to systematically improve their test suite to obtain a small, yet complete, set of tests*. Code Cover did not specifically provide information about whether test cases were redundant. Thus, we had to analyze this hypothesis indirectly. We used the information gathered each time a student executed Code Cover to track the size of the test suite over time. We could then compare the size of the test suite to the obtained coverage percentage. Further, we computed the minimum number of tests required to completely test each program and compared the sizes of the student test suites to this minimum.

First, relative to the size of the test suite, the data indicates that the students did not understand how to create a small test suite. The minimum number of tests required to fully test the programs was 19 for the Calendar and 24 for the Deque. We computed a one-sample t-test to see if the size of the students' test suite was significantly larger than these minimum values. We conducted the analysis separately for each program both before and after using the Code Cover tool. The results (Table 2) show that in most cases the students created significantly more tests than necessary. The only exception is for the Calendar program in Study 2. However, combining these results with Figure 2 indicates that, given their low coverage scores, the students were missing test cases, helping to explain the small test suite size in this case. The large test suites result from the inclusion of redundant test cases suggesting that the students did not fully understand how to test.

Table 2 – Size of Test Suite

	Calendar (Min=19 test cases)				Deque (Min=24 test cases)			
	Study 1 (n=21)		Study 2 (n=20)		Study 1 (n=14)		Study 2 (n=10)	
	Before	After	Before	After	Before	After	Before	After
Mean # tests	43.57	56	21	19.29	86.6	44.36	56.33	51.30
Std. Dev	37.804	45.965	18.058	5.711	50.01	26.105	23.817	7.134
t-value	2.979	3.711	.508	.229	4.849	2.918	4.073	12.101
p-value	.007	.001	.617	.821	< .001	.012	.004	< .001

In general, it appears that students did not use a systematic approach to add test cases to their test suites. When we plotted the increase in coverage against the number of attempts none of the coverage metrics showed a positive trend. This observation suggest that students often added test cases without much thought as to how the test cases would increase coverage. A full analysis of this data is beyond the scope of this paper due to space limitations. In a follow on work we plan to analyze the test suites in detail to identify any patterns that the students may have been following during this process.

These results indicate that while students may be able to achieve a high level of coverage (H2), they reach this level simply by adding test cases to their test set rather than relying on a deep understanding of exactly which test cases would increase coverage.

4. Discussion and Related Work

Based on the findings of our study, it is clear that senior-level students have inadequate testing ability. To better understand this situation occurred and what others are doing to address it, we performed a literature survey. We group the results into two categories: Educational Approaches (Section 4.1) and Tools for Teaching Testing (Section 4.2).

4.1 Educational Approaches

In this section we describe and critique four approaches to testing education.

Approach 1: Students learn to test by submitting test cases along with source code

Noonan and Prosi observed that an important side effect of requiring test case submissions was to increase the quality of students' source code (e.g., lower coupling between classes because higher coupling makes testing more difficult) [19]. In another approach, Goldwasser based a student's grade both on how many faults their test cases identified in the other students' code and on how many of the other students' test cases

their own code passed. Therefore, students are encouraged to write both high quality code and high quality test cases [11]. Many educators using Test-Driven Development (Approach 4) also require the submission of test cases. The main drawback to Approach 1 is that instructors are required to do extra grading of test cases. Thus, providing students with the benefits of submitting test cases without imposing on instructors the overhead required to manually grade these test cases could increase the adoption of this approach.

Approach 2: Students learn to test by testing code written by someone else

Chmiel and Loui had students perform two sets of debugging exercises on provided code: 1) code reading to find a known number of faults; 2) use of debugging tools to find bugs [4]. In a Software Design and Testing course, Carrington had students work in teams to conduct three testing assignments on someone else's code. In the first assignment, the students created black box tests. In the second and third assignments, the students created design and testing frameworks [3]. In a more advanced course (i.e., after data structures), Frezza had students test pre-written code [9]. While these methods do encourage students to test code more thoroughly, they do not provide feedback about why a test suite is incomplete. However, in each of these methods, students could succeed through trial-and-error or brute-force. If students were provided with feedback about why their test suites are incomplete, they could learn to improve these test suites systematically.

Approach 3: Students learn to test through integration of testing across multiple courses

Jones developed the SPRAE (Specification, Premeditation, Repeatability, Accountability, and Economy) framework to support testing in early courses. He included testing experiences in a core course and in testing elective and lab courses. He also proposed specific methods for integrating testing across the curriculum: 1) grade other student's programs, 2) write test cases first, 3) identify seeded bugs, 4) generate test cases from a formal specification, and 5) implement a coverage analyzer [16]. Wick et al. teach students how to use JUnit to improve their testing ability. During the first course, students execute pre-defined test cases. In the second course (algorithms and data structures), the students design their own JUnit test cases. Finally, in a senior-level software engineering course and in a capstone course, the students use design patterns to create their own tests [23]. These approaches do encourage the systematic inclusion of testing across the curriculum, but they do not provide facilities to tailor the type of feedback provided to the student. Beginning students likely need more detailed feedback than more senior students.

Approach 4: Students learn to test through use of Test-Driven Development

Much of the work on introducing testing early in the curriculum has focused on using test-driven development (TDD) to encourage students to test their programs [14]. In TDD a developer writes a test case that will fail because no code exists yet. They then write code to make the test case pass and refactor this code. [1]. A panel discussion at SIGCSE 2008 advocated the early introduction of testing into curriculum, especially via TDD [12]. Christensen argues that testing should be part of the "core knowledge" of computer science and use of TDD throughout the curriculum is the way to accomplish this task [5]. Janzen and Saiedian also advocate the introduction of testing early in the curriculum via TDD [15]. TDD often uses JUnit. But, Proulx argued that JUnit is too difficult for introductory students and developed a new tool to support TDD in early courses [20]. Janzen and Saiedian also advocate 'test-driven learning', which teaches programming by introducing new concepts via automated unit tests (often written before the code). This approach can be included in courses at all levels of the curriculum [13].

The use of TDD in early computer science courses has not seen universal success. Desai et al. showed that the use of TDD in an introductory programming course resulted in students passing a significantly higher number of test cases. Furthermore, they evaluated the effect of grading the test cases by comparing students whose test cases were graded to those students whose test cases were not graded. As the programming projects increased in difficulty, the code coverage decreased both for the graded test group and for the ungraded test group, although the graded test group did obtain more coverage (65% vs. ~30% on the most difficult project) [7]. In an introductory programming course, Marrero and Settle used TDD by requiring students to submit tests before submitting code. They compared the use of TDD to a more traditional approach. Contrary to earlier results, in the Java I course, TDD students did worse than non-TDD students. Although, TDD students did have more success in the Java II course [17].

4.2 Tools for Teaching Testing

We also identified four educational methods with tool support. This section describes each of these methods and tools, again noting drawbacks and possible improvements.

Murphy and Yildirim's approach is based on the belief that students do not always know which test cases to write and will learn by seeing test cases written by others. Students develop the code and test cases for a Java function prototype. After testing their code against their own test cases, students gain access to test cases written by others [18]. We believe that added benefit could result from providing students with feedback on their test suites that is based on a comparison to an instructor-provided test suite.

Collofello and Vehathiri developed a testing simulator that executes student test cases against built-in buggy programs (students cannot submit their own programs). After running the student's tests, the tool reports the following metrics: 1) *test completeness* – a measure of input coverage, 2) *flow coverage* – a measure of statement/path coverage, 3) *correctness* – a measure of agreement between student test outputs and expected test outputs, and 4) a *fault detection metric* – a measure of the effectiveness of test cases in finding faults. After the testing exercise, the tool gives the students the 'answers' (i.e., the correct set of tests) [6]. This tool helps students learn by simply showing them the missing test cases. However, it does not help the students understand why those test cases are needed. Providing students with feedback about the testing concepts (e.g., boundary conditions) that were not fully exercised would help those students to better understand how to develop their own correct and complete set of test cases.

Spacco et al. developed and evaluated Marmoset to provide students with guidance when testing their class projects. For each project, there are four sets of test cases: 1) *student tests* – written by the students, 2) *public tests* – written by the instructor and given to the students, 3) *release tests* – instructor's confidential test cases, results of which are selectively made available to students, and 4) *secret tests* – instructor's additional confidential tests (never made available to the students). Once a student's program passes all of the student tests and public tests, the student can run the release tests. Marmoset tells the student how many release tests failed and gives them the name of two of the failed tests (regardless of how many failed). Students can run the release tests up to three times every 24 hours [22]. This tool appears to provide some support to help students more fully test their code. But, like the other tools, Marmoset does not tell the students *why* their test suite is incomplete, just the names of two failed tests. Thus, students still focus on passing 100% of the set of available test cases rather than understanding how to fully test their code. Without this understanding, students can pass all of the release tests

but still fail some of the secret tests. More importantly, students will not learn how to improve their testing approach for future projects. Again, providing feedback at the conceptual level could help to improve student understanding.

Edwards developed the WebCAT system that automatically grades student code and test cases based on three factors: 1) *code completeness* – % of a student's test cases that his code passes; 2) *test completeness* – % of the student's code that his test cases cover; and 3) *test validity* – % of the problem (instructor's solution) that the student's test cases cover. The three 0% - 100% scores are multiplied to get the final grade, so students must score well on each factor for a good grade. WebCAT can provide students with detailed feedback about which tests failed. It can also annotate the student's code with suggestions on how to improve it or to indicate why points were deducted. The main focus of WebCAT is to be an automated grading tool that can provide the students with some level of feedback during the development process [8]. Most of the feedback provided by WebCAT helps students improve their grade on the current assignment by showing them which test cases failed and by identifying problematic portions of the code. Providing additional information about why a test suite is incomplete would help students to learn fundamental testing concepts and thus to become better testers.

4.3 Discussion

One of the main drawbacks with these existing tools is that they tend to provide the students with the 'answers' (usually after the students meet some criteria). For example, if the instructors have developed their own complete set of test cases, often the students are able to find out which of those test cases are missing from their own test suite. In addition, code coverage tools provide the students with information about exactly which portions of their code have not been fully tested. While this information is quite useful for improving the solution to the current problem, it does little to help students learn how to be better testers. Beginning students are not able to use the information about a missing test case or an untested portion of the code to abstract the fundamental testing concept that caused the problem. In fact, research has shown that simply reading and re-reading information, e.g. missed test cases, is not as helpful to the learning process as actively recalling information that has been learned, e.g. understanding why test cases are missing and developing those missing test cases on their own [2, 10].

Therefore, there is a need for a method which helps students understand why their test suites are inadequate rather than just providing information about the missing tests. As in other types of education, understanding why something is incorrect is more valuable for long-term learning than simply understanding what is incorrect. If students can understand why their test suite is incomplete, they can recall this information in future testing experiences to improve their overall effectiveness.

5. Conclusions and Future Work

Based on our empirical study and survey of the literature, we can draw the following conclusions. First, senior-level computer science students are not adequately prepared in the arena of testing. This lack of preparation is problematic, because for many students, their first job will include at least some type of testing activity. Second, based on the literature survey, educators have proposed many different techniques for teaching testing. While these techniques all have strengths, we have identified shortcomings that must be addressed by new tools for teaching testing, especially early in the computer science

curriculum. In our immediate future work, we plan to conduct a more extensive evaluation of the evolution of the students' test suites.

6. Acknowledgements

We thank Dr. Randy Smith for allowing us to conduct the study during his course. We also thank the students for participating in the study.

7. References

[1] Beck, K., *Extreme Programming Explained: Embrace Change.* Reading, MA: Addison-Wesley, 2000.

[2] Callender, A. A. and McDaniel, M. A., "The limited benefits of rereading educational texts," *Contemp. Educ. Psychol.*, 34(1): 30. 2009.

[3] Carrington, D. "Teaching software design and testing." In *Proceedings of the ASEE/IEEE Frontiers in Education.* 1998. pp. 547-550.

[4] Chmiel, R. and Loui, M. "Debugging: From novice to expert." In *Proceedings of the 35th ACM Technical Symposium on Computer Science Education.* 2004. pp. 17-21.

[5] Christensen, H. B. "Systematic testing should not be a topic in the computer science curriculum!" In *Proceedings of the 2003 Annual Joint Conference Integrating Technology into Computer Science Education.* June 30 - July 2, 2003. pp. 7-10.

[6] Collofello, J. and Vehathiri, K. "An environment for training computer science students on software testing." In *Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference.* 2005.

[7] Desai, C., Janzen, D. S. and Clements, J. "Implications of integrating test-driven development into CS1/CS2 curricula." In *Proceedings of the 39th Technical Symposium on Computer Science Education.* March 3-7, 2009. pp. 148-152.

[8] Edwards, S. H. "Using software testing to move students from trial-and-error to reflection-in-action." In *Proceedings of the 35th ACM Technical Symposium on Computer Science Education.* 2004. pp. 26-30.

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

[10] Glenn, D., "Close the Book. Recall. Write It Down." *Chronicles of Higher Education,* vol. 55, pp. 1, May 1. 2009.

[11] Goldwasser, M. H. "A gimmick to integrate software testing throughout the curriculum." In *Proceedings of the ACM Technical Symposium on Computer Science Education.* 2002. pp. 271-275.

[12] Hanks, B., Wellington, C., Reichlmayr, T. and Coupal, C., "Integrating agility in the cs curriculum: practices through values," *SIGCSE Bull,* 40(1): 19-20. 2008.

[13] Janzen, D. S. and Saiedian, H. "Test-driven learning in early programming courses." In *Proceedings of the 38th Technical Symposium on Computer Science Education.* 2008. pp. 532-536.

[14] Janzen, D. S. and Saiedian, H. "Test-driven learning: Intrinsic integration of testing into the CS/SE curriculum." In *Proceedings of the 37th Technical Symposium on Computer Science Education.* 2006. pp. 258.

[15] Janzen, D. S. and Saiedian, H., "Test-Driven Development: Concepts, Taxonomy, and Future Direction," *Computer*, 38(9): 43-50. September. 2005.

[16] Jones, E. L. "Integrating testing into the curriculum -- arsenic in small doses." In *Proceedings of the ACM Technical Symposium on Computer Science Education.* 2001. pp. 337-341.

[17] Marrero, W. and Settle, A. "Testing first: Emphasizing testing in early programming courses." In *Proceedings of the 2005 Annual Joint Conference Integrating Technology into Computer Science Education.* 2005. pp. 4-8.

[18] Murphy, M. C. and Yildirim, B. "Work in progress - testing right from the start." In *Proceedings of the ASEE/IEEE Frontiers in Education.* 2007.

[19] Noonan, R. E. and Prosi, R. H. "Unit testing frameworks." In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education.* 2002. pp. 232-236.

[20] Proulx, V. "Test-driven design for introductory OO programming." In *Proceedings of the 40th Technical Symposium on Computer Science Education.* 2009. pp. 138-142.

[21] Shepard, T., Lamb, M. and Kelly, D., "More Testing Should be Taught," *Commun. ACM*, 44(6): 103-108. 2001.

[22] Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J. and Padua-Perez, N. "Experiences with marmoset: Designing and using and advanced submission and testing system for programming courses." In *Proceedings of the 2006 Annual Joint Conference Integrating Technology into Computer Science Education.* 2006. pp. 13-17.

[23] Wick, M., Stevenson, D. and Wagner, P. "Using testing and JUnit across the curriculum." In *Proceedings of the 2005 ACM Technical Symposium on Computer Science Education.* 2005. pp. 236-240.