Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance

Stephen H. EDWARDS
Dept. of Computer Science, Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
+1 540 231 5723

edwards@cs.vt.edu

ABSTRACT

There is a need for better ways to teach software testing skills to computer science undergraduates, who are routinely underprepared in this area. This paper proposes the use of test-driven development in the classroom, requiring students to test their own code in programming assignments. In addition, an automated grading approach is used to assess student-written code and student-written tests together. Students receive clear, immediate feedback on the effectiveness and validity of their testing. This approach has been piloted in an undergraduate computer science class. Results indicate that students scored higher on their program assignments while producing code with 45% fewer defects per thousand lines of code.

Keywords: Computer science, software testing, test-first coding, programming assignments, automated grading

1. INTRODUCTION

Many computer science educators have been looking for an effective way to improve the coverage of software testing skills that undergraduates receive [13]. Rather than adding a single course on the subject, some have proposed systematically infusing testing concerns across the curriculum [6, 7, 8, 11, 12]. However, there is no clear consensus on how this goal is best achieved.

One approach is to require students to test their own code in programming assignments, and then assess them on this task as well as on the correctness of their code solution. Two critical issues immediately arise, however:

1. What testing approach should students use? The approach must provide practical benefits that students can see, and yet be simple enough to apply across the curriculum, well before students have received advanced software engineering experience.

2. How will students be assessed on testing tasks? In particular, if students must test their own code, and then be graded on both their code *and* their testing, how can we avoid doubling the grading workload of faculty and teaching assistants while also providing feedback frequently enough and specifically enough for students to improve their performance?

This paper proposes the use of *test-driven development* in the classroom. In conjunction, an automated grading strategy is used to assess student-written code and student-written tests together, providing clear and immediate feedback to students about the effectiveness and validity of their testing.

2. BACKGROUND

The goal is to teach software testing in a way that will encourage students to practice testing skills in many classes and give them concrete feedback on their testing performance, without requiring a new course, any new faculty resources, or a significant number of lecture hours in each course where testing will be practiced [3].

2.1 Why Test-driven Development?

Test-driven development (TDD) is a code development strategy that has been popularized by extreme programming [1, 2]. In TDD, one always writes a test case (or more) before adding new code. In fact, new code is only written in response to existing test cases that fail. By constantly running all existing tests against a unit after each change, and always phrasing operational definitions of desired behavior in terms of new test cases, TDD promotes a style of incremental development where it is always clear what behavior has been correctly implemented and what remains undone.

While TDD is not, strictly speaking, a testing strategy—it is a code development strategy [1]—it is a practical, concrete technique that students can practice on their own assignments. Most importantly, TDD provides visceral benefits that students experience for themselves. It is applicable on small projects with minimal training. It gives the programmer a great degree of confidence in the correctness of their code. It encourages students to always have a running version of what they have completed so far. Finally, it encourages students to test features and code as they are implemented. This preempts the "big bang" integration problems that students often run into when they work feverishly to write all the code for a large assignment, and only then try to run, test, and debug it.

2.2 Prior Approaches to Automated Grading

Without considering testing practices, CS educators have developed many approaches to automatically assessing student program assignments [4, 5, 9]. While such automated grading systems vary, they typically focus on compilation and execution of student programs against some form of instructor-provided test data. Virginia Tech has been using a similar automated grading system for student programs for more than six years and has seen powerful results. Virginia Tech's system, which is similar in principle to most systems that have been described, is called the Curator.

A student can login to the Curator and submit a solution for a programming assignment. When the solution is received, the Curator compiles the student program. It then runs a test data generator provided by the instructor to create input for grading the submission. It also uses a reference implementation provided by the instructor to create the expected output. The Curator then runs the student's submission on the generated input, and grades the results by comparing against the reference implementation's output. The student then receives feedback in the form of a report that summarizes the score, and that includes the input used, the student's output, and the instructor's expected output for reference.

In practice, such automated grading tools have been extremely successful in classroom use. Automated grading is a vital tool in providing quality assessment of student programs as enrollments increase. Further, by automating the process of assessing program behavior, TAs and instructors can spend their grading effort on assessing design, style, and documentation issues. Further, instructors usually allow multiple submissions for a given program. This allows a student to receive immediate feedback on the performance of his or her program, and then have an opportunity to make corrections and resubmit before the due deadline.

2.3 Challenges

Despite its classroom utility, an automatic grading strategy like the one embodied in the Curator also has a number of shortcomings:

- Students **focus on output correctness** first and foremost; all other considerations are a distant second at best (design, commenting, appropriate use of abstraction, testing one's own code, etc.). This is due to the fact that the most immediate feedback students receive is on output correctness, and also that the Curator will assign a score of zero for submissions that do not compile, do not produce output, or do not terminate.

- Students are **not encouraged or rewarded** for performing testing on their own.

- In practice, students **do less testing** on their own.

This last point is disturbing; in fact, many students rarely or never perform serious testing of their own programs when the Curator is used. This is understandable, since the Curator already has a test data generator for the problem and will automatically send the student the results of running tests on his or her program. Indeed, one of the biggest complaints from students has to do with the form of the feedback, which currently requires the student to do some work to figure out the source of the error(s) revealed.

3. WEB-CAT: A TOOL FOR AUTOMATICALLY ASSESSING STUDENT PROGRAMS

In order to consider classroom use of TDD practical, the challenges faced by existing automated grading systems must be addressed. Web-CAT, the Web-based Center for Automated Testing, is a new tool that grades student code and student tests together. Most importantly, the assessment approach embodied in this tool is based on the belief that a student should be given the responsibility of demonstrating the correctness of *his or her own* code.

3.1 Assessing TDD Assignments

In order to provide appropriate assessment of testing performance and appropriate incentive to improve, Web-CAT should do more than just give some sort of “correctness” score for the student’s code. In addition, it should assess the validity and the completeness of the student’s tests. Web-CAT grades assignments by measuring three scores: a test validity score, a test completeness score, and a code correctness score.

First, the *test validity score* measures how many of the student’s tests are accurate—consistent with the problem assignment. This score is measured by running those tests against a reference implementation provided by the instructor to confirm that the student’s expected output is correct for each test case.

Second, the *test completeness score* measures how thoroughly the student’s tests cover the problem. One method to assess this aspect of performance is to use the reference implementation provided by the instructor as a surrogate representation of the problem. By instrumenting this reference implementation to measure the code coverage achieved by the student tests, a score can be measured. In our initial prototype, this strategy was used and branch coverage (basis path coverage) served as the test completeness score. Other measures are also possible.

Third, the *code correctness score* measures how “correct” the student’s code is. To empower students in their own testing capabilities, this score is based solely on how many of the student’s own tests the submitted code can pass. No separate test data is provided by the instructor or teaching assistant. The reasoning behind this decision is that, if the student's test data is both valid (according to the instructor's reference implementation) and complete (also according to the reference), then it must do a good job of exercising the features of the student program.

To combine these three measures into one score, a simple formula is used. All three measures are taken on a 0%–100% scale, and the three components are simply multiplied together. As a result, the score in each dimension becomes a “cap” for the overall score—it is not possible for a student to do poorly in one dimension but do well overall. Also, the effect of the multiplication is that a student cannot accept so-so scores across the board. Instead, near-perfect performance in at least two dimensions should become the expected norm for students.

To support the rapid cycling between writing individual tests and adding small pieces of code, the Web-CAT Grader allows *unlimited* submissions from students up until the assignment deadline. Students can get feedback any time, as often as they wish. However, because their score is based in part on the tests

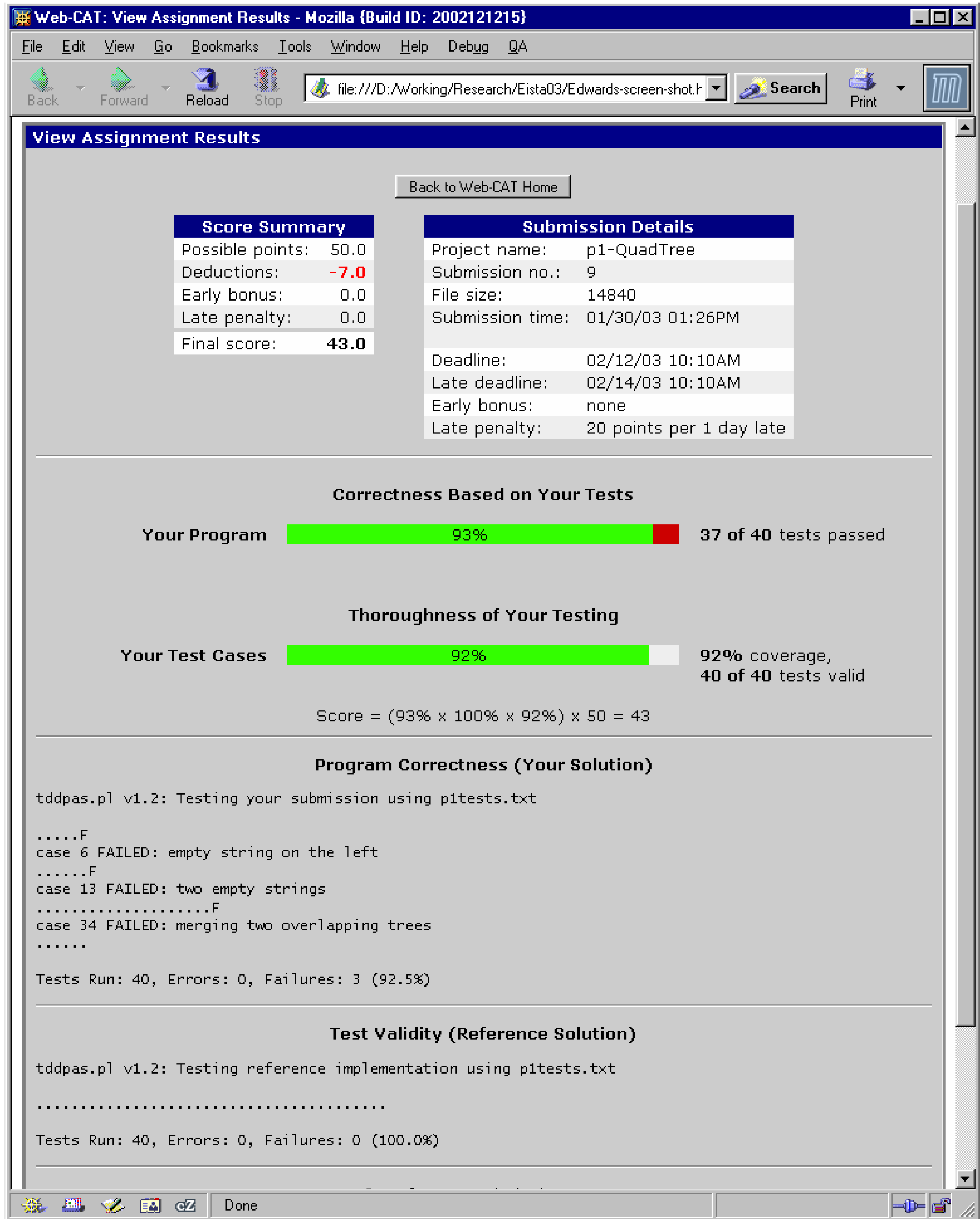

Figure 1: Web-CAT feedback on a program assignment submission.

they have written, and their program performance is only assessed by the tests they have written, to find out more about errors in their own programs, it will be necessary for the student to write the test cases.

3.2 Providing Feedback to Students

The Web-CAT Grader uses a web interface for student submissions and for reporting. The feedback provided to students was inspired by JUnit's GUI TestRunner: "when the bar is green the code is clean" [10]. Figure 1 shows a sample screen shot of the results viewed by students after a submission. The three component scores and the final cumulative score are graphically summarized in two bars. The first bar assesses the program, showing the percentage of test cases passed. Since this bar is static, rather than the dynamic progress bar in the JUnit TestRunner, part of the bar is shown in green and part in red, based on the proportion of student-provided tests that have been passed. The second bar assesses the student's test suite. The size of the bar reflects the degree of coverage achieved by the test suite. The color of the bar indicates the validity of the test suite: green means all tests were valid, and red means at least one test's expected output disagreed with the reference implementation.

Below the bar graph summary, the details of the test runs are printed in a format similar to the one produced by JUnit's text output TestRunner. First, the student program's test execution information is presented. Each failed test case is specifically identified with a descriptive message, so the student can determine where to go next to find the problem. Second, the reference implementation's test execution information is presented in the same format. Any "failed" test cases in this section indicate student-written test cases that have incorrect expected output.

4. PRELIMINARY EVALUATION OF WEB-CAT

To evaluate the practicality of this approach, it was tried out in an upper-division undergraduate computer science course. The course used for this evaluation was CS 3304: "Comparative Languages," a typical undergraduate programming languages course. Students in the course normally write four program assignments throughout the semester, each requiring two to three weeks to complete. The basic evaluation strategy was to provide basic instruction in TDD and employ Web-CAT for grading all programming assignments, and then compare student performance with that achieved in a past offering of the course when TDD was not used.

4.1 Method

In the Spring 2003 semester, 59 students completed the Comparative Languages course, using Web-CAT to submit all programming assignments. The Spring 2001 offering of the course was chosen for comparison, so the original programming assignments from that semester were given again in Spring 2003. In Spring 2001, students did not use TDD and instead used the older automated grading system, Virginia Tech's Curator, when submitting assignments. Fortunately, an electronic archive of all submissions made during that semester was available for detailed analysis. Web-CAT also maintained a detailed archive of new submissions for comparison. A total of 59 students completed the course in Spring 2001, for a total of 118 subjects split between the two treatments.

Unfortunately, while TDD practices are strongly supported in many object-oriented languages, students in the Comparative Languages course write programs using a number of other programming paradigms, including procedural programming, functional programming, and logic programming, in languages like Pascal, Scheme, and Prolog. Since no generally available TDD tools exist for these languages, a simple and easy-to-use infrastructure for writing and executing TDD-oriented test cases was developed. Students were given approximately 30 minutes of classroom instruction on TDD, how to write test cases, how to execute tests, and how to interpret results. Further, when example programs were developed "live" in class sessions later in the semester, the same TDD infrastructure was used by the instructor to model expected behavior properly.

4.2 Results

After assignments were turned in, the final submission of each student in both semesters was analyzed. This analysis was restricted to the first programming assignment (in Pascal) due to manpower limitations.

Table 1: Comparing program submissions between groups; all differences are significant at $\alpha = 0.05$.

	With TDD (2003)	Without (2001)
Web-CAT Score	94.0%	76.8%
Code Coverage	93.6%	90.0%
Defects/KSLOC	38	70

Table 1 summarizes the results obtained when comparing the program submissions between the two groups of students. Because Web-CAT and the older Curator system use different grading approaches, the Spring 2001 submissions were also submitted through Web-CAT for scoring. In Spring 2001, however, students did not write test cases. Rather than using a fixed set of instructor-provided test data, the 2001 programs were graded using a test data generator provided by the instructor. This generator produced a random set of 40 test cases for each new submission, providing broad coverage of the entire problem. To re-score each 2001 submission using Web-CAT, the generator-produced test cases originally produced for grading that submission in 2001 were simultaneously submitted as if they were produced by the student. As shown in Table 1, students in 2003 scored significantly higher than students in 2001.

This should be no surprise, since students using Web-CAT received specific feedback on the quality and thoroughness of their testing effort, information that was unavailable to the other group of students. By seeing a direct measure of the coverage produced by their test suite, and having unlimited opportunities to add test cases and try to improve, students using Web-CAT were able to increase their coverage scores. As shown in Table 1, the branch coverage scores achieved by student-written test suites in 2003 were significantly higher than the code coverage scores achieved by the random test data generator used for automated grading in 2001. Since students in 2003 did not have access to the reference implementation, they could only increase their coverage scores by creatively "guessing" what kinds of behavior or features they should test, either by looking at the assignment specification or by looking at their own solution more closely. In effect, this simple feedback mechanism

implicitly encouraged students to practice and develop their skills at developing good black box test cases.

Finally, the student programs were analyzed to uncover the bugs they contained. One of the most common ways to measure bugs is to assess defect density, that is, the average number of defects (or bugs) contained in every 1000 non-commented source lines of code (KSLOC). On large projects, defect density data can often be collected by analyzing bug tracking databases. For student programs, however, measuring defects can be more difficult.

To provide a uniform treatment in this experiment, a comprehensive test suite was developed for analysis purposes. A suite that provided 100% condition/decision coverage on the instructor's reference implementation was the starting point. Then all test suites submitted by 2003 students and all randomly generated suites used to grade 2001 submissions were inspected, and all non-duplicating test cases from this collection were added to the comprehensive suite. For this experiment, two test cases are "duplicating" if each program in each of the student groups produces the same result (pass or fail) on both test cases. Non-duplicating test cases are thus "independent" for at least one program under consideration, but may provide redundant coverage for others. Once the comprehensive test suite was constructed, every program under consideration was run against it.

While the resulting numbers capture the relative number of defects in programs, they do not represent defect density. To get defect density information, a selection of 18 programs were selected, 9 from each group. These programs had all comments and blank lines stripped from them. They were then debugged by hand, making the minimal changes necessary to achieve a 100% pass rate on the comprehensive test suite. The total number of lines added, changed, or removed, normalized by the program length, was then used as the defects per KSLOC measure for that program. A linear regression was performed to look for a relationship between the defects/KSLOC numbers and the raw number of test cases failed from the comprehensive test suite in this sample population. This produced a correlation significant at the 0.05 level, which was then used to estimate the defects/KSLOC for the remaining programs in the two student groups.

Table 1 summarizes the results of this analysis, which show that students who used TDD and Web-CAT submitted programs containing approximately **45% fewer defects** per 1000 lines of code. While the defects/KSLOC rates shown here are far above industrial values, with values often cited around 4 or 5 defects/KSLOC, this is to be expected for student-quality code developed with no process control and no independent testing.

While the results summarized in Table 1 indicate that students do produce higher quality code using this approach, it is also important to consider how students react to TDD and Web-CAT. The 2003 students completed an anonymous survey designed to elicit their perceptions of both the process and the prototype tool. All students in the Spring 2003 semester had used an automated grading/submission system before (the Curator).

Students expressed a strong preference for Web-CAT over their past experiences. Specifically, they found that Web-CAT was more helpful at detecting errors in their programs than the Curator (89.8% agree or strongly agree). In addition, they believed it provided excellent support for TDD (83.7% agree or strongly agree).

Students also expressed a strong preference for the benefits provided by TDD. Using TDD increases the confidence that students have in the correctness of their code (65.3% agree or strongly agree). Using TDD also increases the confidence that students have when making changes to their code (67.3% agree or strongly agree). Finally, most students **would like to use** Web-CAT and TDD for program assignments in future classes, **even if it were not required** for that course (73.5% agree or strongly agree).

5. CONCLUSIONS AND FUTURE WORK

Preliminary experience with TDD in the classroom and with automated assessment is very positive, indicating a significant potential for increasing the quality of student code. We plan to apply this technique in our introductory programming sequence, where students will program in Java and use JUnit [10]. We also plan to extend the empirical comparison of student programs to this larger group of students. Further, we plan to adapt the assessment approach used to measure test completeness so that the tool can provide specific, directed feedback to students about how and where they can improve the completeness of their testing efforts.

ACKNOWLEDGMENTS

I gratefully acknowledge the contributions to this work provided by Anuj Shah, Amit Kulkarni, and Gaurav Bhandari,, who implemented many of the features of the automated grading and feedback system described here.

REFERENCES

[1] Beck, K. Aim, fire (test-first coding). *IEEE Software*, 18(5): 87-89, Sept./Oct. 2001.

[2] Beck, K. *Test-Driven Development: By Example*. Addison-Wesley, Boston, MA. 2003.

[3] Goldwasser, M.H. A gimmick to integrate software testing throughout the curriculum. . In *Proc. 33rd SIGCSE Technical Symp. Computer Science Education*, ACM, 2002, pp. 271-275.

[4] Isong, J. Developing an automated program checker. *J. Computing in Small Colleges*, 16(3): 218-224.

[5] Jackson, D., and Usher, M. Grading student programs using ASSYST. In *Proc. 28th SIGCSE Technical Symp. Computer Science Education*, ACM, 1997, pp. 335-339.

[6] Jones, E.L. Software testing in the computer science curriculum—a holistic approach. In *Proc. Australasian Computing Education Conf.*, ACM, 2000, pp. 153-157.

[7] Jones, E.L. Integrating testing into the curriculum—arsenic in small doses. In *Proc. 32nd SIGCSE Technical Symp. Computer Science Education*, ACM, 2001, pp. 337-341.

[8] Jones, E.L. An experiential approach to incorporating software testing into the computer science curriculum. In *Proc. 2001 Frontiers in Education Conf. (FiE 2001)*, 2001, pp. F3D7-F3D11.

[9] Jones, E.L. Grading student programs—a software testing approach. *J. Computing in Small Colleges*, 16(2): 185-192.

[10] JUnit Home Page. Web page last accessed Mar. 21, 2003: <http://www.junit.org/>

[11] McCauley, R., Archer, C., Dale, N., Mili, R., Robergé, J., and Taylor, H. The effective integration of the software engineering principles throughout the undergraduate computer science curriculum. In *Proc. 26th SIGCSE Technical Symp. Computer Science Education*, ACM, 1995, pp. 364-365.

[12] McCauley, R., Dale, N., Hilburn, T., Mengel, S., and Murrill, B.W. The assimilation of software engineering into the undergraduate computer science curriculum. In *Proc. 31st SIGCSE Technical Symp. Computer Science Education*, ACM, 2000, pp. 423-424.

[13] Shepard, T., Lamb, M., and Kelly, D. More testing should be taught. *Communications of the ACM*, 44(6): 103–108, June 2001