Improving Student Performance by Evaluating How Well Students Test Their Own Programs

STEPHEN H. EDWARDS
Virginia Tech

Students need to learn more software testing skills. This paper presents an approach to teaching 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. The strategy is to give students basic exposure to test-driven development, and then provide an automated tool that will assess student submissions on-demand and provide feedback for improvement. This approach has been demonstrated in an undergraduate programming languages course using a prototype tool. The results have been positive, with students expressing appreciation for the practical benefits of test-driven development on programming assignments. Experimental analysis of student programs shows a 28% reduction in defects per thousand lines of code.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms: Measurement

Additional Key Words and Phrases: test-driven development, test-first coding, extreme programming, agile methods, teaching software testing

1. INTRODUCTION

The productivity goals of most software companies can be summed up simply: increase the quality of produced software while decreasing the cost and time of development. At the same time, however, the problem of software defects, or "bugs," continues to grow under this pressure. According to a recent article, "defective code remains the hobgoblin of the software industry, accounting for as much as 45% of computer-system downtime and costing U.S. companies about $100 billion [in the year 2000] in lost productivity and repairs" [Ricadela, 2001]. Software testing is an indispensable tool in moving toward the goal of greater software quality, and often consumes a great deal of development resources in practice [Harrold, 2000; Boehm, 1976; Beizer, 1990].

Unfortunately, testing is perceived as tedious, uncreative, boring work by practitioners, less than 15% of whom ever receive any formal training in the subject [Wilson, 1995]. Despite the importance of the topic, most computer science curricula provide only minimal coverage of the topic. A recent article in *Communications of the ACM* exhorts faculty to teach more software testing; "Students today are not well equipped to apply widely practiced techniques … They are graduating with a serious gap in the knowledge they need to be effective software developers" [Shepard, Lamb, & Kelly, 2001].

Author's address: Department of Computer Science, Virginia Tech, 660 McBryde Hall MS 0106, Blacksburg, VA 24061; email: edwards@cs.vt.edu

Despite the importance of software testing, testing skills are often elusive in undergraduate curricula [Bourque et al., 2001; Bagert et al., 1999]. This may be because test ing pervades all aspects of software development and seems poorly suited as the sole topic for a new course. Even computer science freshmen need exposure to basic testing ideas so they can appropriately "test" small assignments. On the other hand, many testing ideas are not easy to internalize until one has had significant development experience. There does not appear to be one "right" place in the undergraduate experience to place this kind of learning. This conclusion echoes current recommendations for undergraduate software engineering training [Bagert et al., 1999]: software testing is a crosscutting or recurring area that must be visited multiple times with increasing levels of depth and maturity throughout the curriculum.

At the same time, however, most undergraduate computer science curricula are already overstuffed with content, and there is little room to infuse significant coverage of a crosscutting topic in many courses. The goal, then, 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. This alternative approach is made possible by reframing the way student programming assignments are assessed: a student should be given the responsibility of demonstrating the correctness of *his or her own* code. Students are expected and required to submit test cases for this purpose along with their code, and assessing student performance includes a meaningful assessment of how correctly and thoroughly the tests conform to the problem. The key to providing rapid, concrete, and immediate feedback is an automated assessment tool to which students can submit their code. Experimental evidence suggests that this approach may significantly improve the quality of student solutions, in terms of the number of latent defects present in the "final solutions" turned in. As a result, it is worth exploring the practicality of applying this technique across a series of courses or even an entire computer science curriculum.

1.1 RELATED WORK: TEACHING SOFTWARE TESTING

Periodic panel sessions and speakers at SIGCSE have suggested that software engineering concepts in general should be incorporated throughout the undergraduate computer science curriculum [McCauley et al., 1995; McCauley et al., 2000]. While some of the goals are the same as those in this paper—ensuring students acquire sufficient knowledge of best practices for managing development tasks—the focus in such related work is broad, rather than specific to testing. Hilburn and Towhidnejad [2000] have discussed software quality as an important overarching concern worthy of special attention in this regard, although no specific proposals for including testing-based assessment are presented.

Other educators and practitioners have specifically addressed the topic of teaching software testing. Shepard, Lamb, and Kelly [2001] make an appeal for educators to teach more testing, but offer little concrete direction. Goldwasser [2002] describes an interesting classroom technique where students submit test cases along with their code. After the due date, all student test suites are run against all student submissions, and extra credit is given both to the strongest survivor and to the most deviously successful test data. Overall, however, Jones is the most outspoken educator on the subject [Jones, 2000a; Jones, 2001a; Jones, 2001b; Jones, 2001c]. He has suggested systematically incorporating testing practices across the undergraduate curriculum, with the specific aim of producing students who are more realistically prepared for real-world testing tasks upon graduation. Jones has suggested that instructors use testing strategies as an aid in writing more focused and effective program assignment specifications. He has also suggested that a similar approach can be used by instructors in assessing the correctness of student submissions. His SPRAE framework is aimed at teaching students the foundation skills needed for on-the-job testing in industry [Jones, 2000b]. While Jones shares many of the same goals with the work reported here, and his work is the most closely related, the key differences are the use of test-driven development, the focus on evaluating testing practices rather than code writing skills, and the rapid-turnaround feedback emphasized in this paper.

1.2 RELATED WORK: AUTOMATIC GRADING

Without considering testing practices, CS educators have developed many approaches to automatically assessing student program assignments. Many informal grading systems have been developed in-house, although the results are often difficult to port among institutions, and sometimes even among course instructors. Isong [2001] describes an approach typical of such grading systems. Her automated program checker focuses on compiling and executing student programs against instructor-provided test cases, and then assigning a grade based on comparing the actual output of the student program against expected results provided by the instructor. Like many other in-house grading systems, Isong's checker is written as a collection of Unix shell scripts.

Reek [1996] also describes a Unix-based grading system used for introductory courses at the Rochester Institute of Technology. It uses a file-system-based organizational strategy for managing assignments and student uploads. Like Isong's checker, Reek's grading tool focuses on compiling and executing student programs against instructor-provided test data. Instructors can assign categories to different test cases in order to control whether or not students can see the output compared to what was expected, and whether failures halt or abort the grading process.

Luck and Joy [1999] describe the use of BOSS at the University of Warwick. BOSS consists of a collection of Unix-based programs that also use a file-system-based organizational strategy for managing student submissions and automatically testing them using instructor-provided test scenarios. In their description of BOSS, Luck and Joy pay particular attention to security and privacy issues, user interface issues, how rogue programs are sandboxed, and how plagiarism detection can be included.

Jones' Program Submission and Grading Environment (PSGE) is another Unix-based assignment checker [Jones, 2001c]. It supports both semi-automatic and fully automated grading, depending on the instructor's choice and the nature of the assignment. Unlike some other systems, PSGE defers grading until after the due date, so students only receive feedback once, after the assignment is complete. While Jones advocates including software testing in the undergraduate curriculum, with regard to automated grading, he primarily advocates testing strategies to the instructor. By taking a testing-oriented approach, an instructor can produce better assignments and more repeatable, accountable correctness assessments [Jones, 2001c]. PSGE does not handle assessment of or produce feedback on student-written test cases, however.

Mengel and Yerramilli [1999] have proposed the use of static analysis for automatic grading. They have attempted to correlate code metrics to scores assigned in a more traditional fashion by graduate teaching assistants. The hope is that by validating such a correlation, commercial static analysis tools will allow predictive use of metrics for automated grading purposes. Mengel and Yerramilli only consider assessing student programs, however, rather than the effectiveness of student testing.

ASSYST is perhaps the most widely scoped automated grading system discussed in the literature [Jackson & Usher, 1997]. It provides the basic compile/test/compare-output features of other systems, but also includes static checks for style and code complexity metrics. Most importantly, ASSYST is the only other grading system that can assess student-provided test cases. It assesses student tests by instrumenting the student program and measuring the statement coverage achieved by the student tests. Interestingly, this is done only to assign deductions (if any) to student-written tests, and the correctness of student code is assessed in the traditional way.

Finally, Virginia Tech uses its own automated grading system for student programs 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 [McQuain, 2003]. 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. Typically, 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. In addition, 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. In this way, the Curator is effective at providing more timely feedback—and more "review and correct" cycles—than an instructor or TA could provide manually. In addition, this practice encourages students to begin assignments earlier. Using such a tool has powerfully affected the programming practices of Virginia Tech CS students.

2. WHY TEST-DRIVEN DEVELOPMENT?

If one wishes for students to practice testing, a testing method must be taught. Although there are many possibilities to choose from, this work is based on the use of test-driven development (TDD), also known as test-first coding. The core idea is to expose students to basic TDD concepts, and then set up programming assignments so that students are always expected to practice TDD on their programming assignments from the beginning.

TDD is a code development strategy that has been popularized by extreme programming [Beck, 2001; Beck, 2003]. 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 [Beck, 2001]—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.

The idea of using TDD in the classroom is not revolutionary. Like other extreme programming practices that suit development in-the-small [Williams & Upchurch, 2001; Nagappan et al., 2003], it is beginning to see use in educational settings. Aleen, Cartwright, and Reise describe its use in a software engineering course, for example [Allen, Cartwright, & Reis, 2003]. What is novel is focusing on assessing student performance at testing, rather than aiming at evaluating program correctness. The real issue is how to overcome the challenges involved in carrying out such assessment activities, which are important considerations.

3. CHALLENGES TO AUTOMATED ASSESSMENT

While automatic grading has proven its classroom utility at many institutions, traditional approaches like the one embodied in the Curator also have 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. This is analogous to a well-known problem in benchmarking [Ar & Cai, 1994]: when the subject self-optimizes for a particular measurement approach, the measure may no longer capture what was intended and construct validity is at risk.
- 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.

In addition, there are three further issues that must be addressed in order for an approach to have a significant effect on student actions in later courses:

- Student practices that are not integrated consistently and continually as **part of the activity of programming** may be seen as separate from programming itself, and therefore not useful to the student's core desire to "learn how to program."
- Students need clear, direct, and useful **feedback** about both how they are performing and how they can improve on testing and programming tasks. More frequent feedback—and more frequent practice—is more effective.
- Students must **value** any practices we require alongside programming activities. A student must see any extra work as helpful in completing working programs, rather than a hindrance imposed at the instructor's desire, if we wish for students to continue using a technique faithfully.

The goal of this paper is to describe experiences with an assessment strategy built around student use of TDD on programming assignments—a strategy that successfully meets all of these challenges.

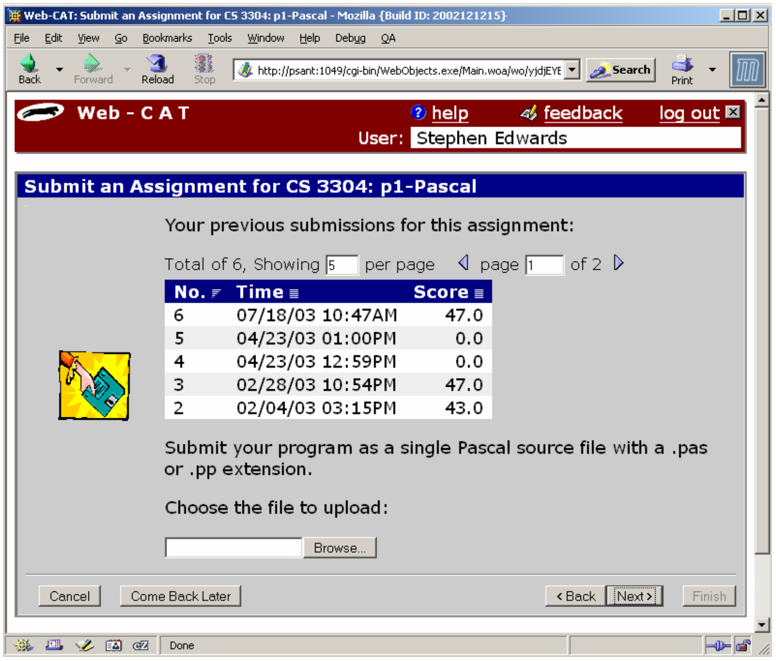
4. ASSESSING TEST-FIRST ASSIGNMENTS

If students are to be assessed on the TDD performance, then assignments must require that test suites be submitted along with code. Ideally, students should be able to "try out" their code-in-progress together with their tests early and often, getting timely feedback at every step. A "test-first assignment" is one that includes such requirements, along with a clearly explained grading rubric so that the student can understand how work will be scored.

In order to provide appropriate assessment of testing performance and appropriate incentive to improve, such a scoring strategy 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. The Web-CAT Grader grades assignments by coming up with three scores:

1. A *test validity score* measures how many of the student's tests are accurate—consistent with the problem assignment.
2. A *test completeness score* measures how thoroughly the student's tests cover the problem. One method is to use the reference implementation as a surrogate for the problem, and measure code coverage over this reference. Other measures are also possible.
3. A *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.

These three measures, taken as percentages, are then multiplied together to come up with a composite score. This formula ensures that no aspect of the approach can be ignored, or the student's score will suffer.


Web-CAT: Submit an Assignment for CS 3304: p1-Pascal - Mozilla {Build ID: 2002121215}
File Edit View Go Bookmarks Tools Window Help Debug QA
Back Forward Reload Stop
http://psant:1049/cgi-bin/WebObjects.exe/Main.woa/wo/yjdjEYE
Search Print
Web - C A T
help feedback log out
Browse...
Cancel Come Back Later < Back Next > Finish
Done

User: Stephen Edwards

Submit an Assignment for CS 3304: p1-Pascal

Your previous submissions for this assignment:

Total of 6, Showing 5 per page page 1 of 2

No.	Time	Score
6	07/18/03 10:47AM	47.0
5	04/23/03 01:00PM	0.0
4	04/23/03 12:59PM	0.0
3	02/28/03 10:54PM	47.0
2	02/04/03 03:15PM	43.0

Submit your program as a single Pascal source file with a .pas or .pp extension.

Choose the file to upload:

Figure 1. The file upload screen for student submissions.

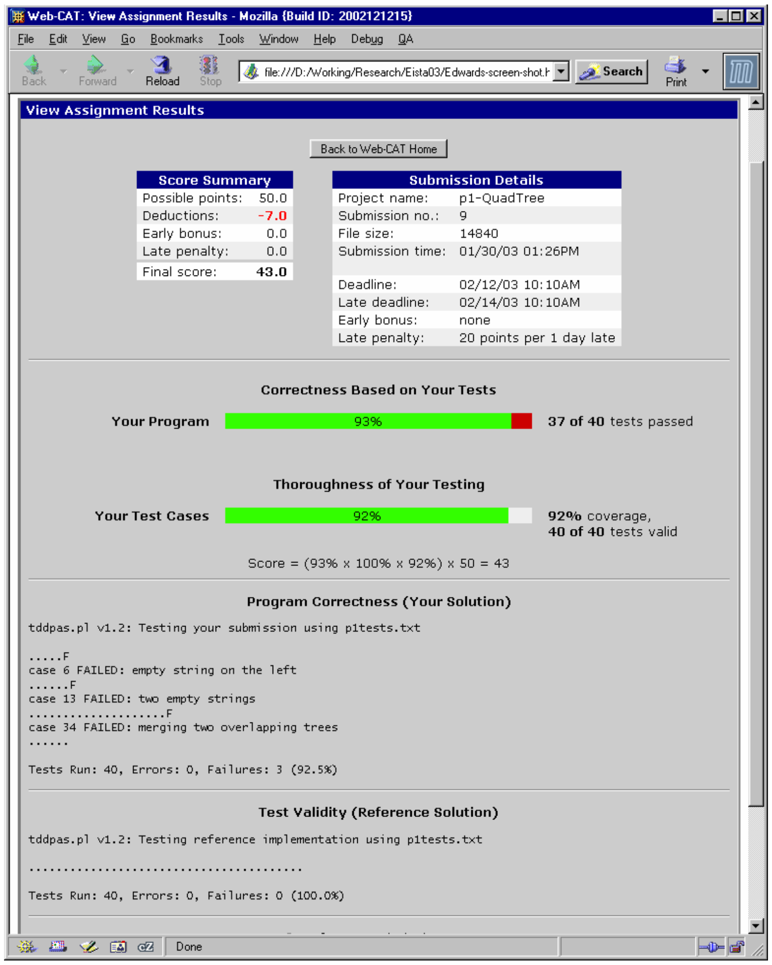
5. WEB-CAT: A NEW AUTOMATED GRADING TOOL

Instead of automating an assessment approach that focuses on the *output* of a student's program, we must focus on what is most valuable: the student's testing performance. To this end, we have developed a prototype tool called the Web-CAT Grader, a service provided by the Web-based Center for Automated Testing. The Web-CAT Grader is designed to:

- Require a student test suite as part of every submission.
- Encourage students to write thorough tests.
- Encourage students to write tests as they code (in the spirit of TDD), rather than postponing testing until after the code is complete.
- Support the rapid cycling of "write a little test, write a little code" that is the hallmark of TDD.
- Provide timely, useful feedback on the quality of the tests in addition to the quality of the solution.
- Employ a grading/reward system that fosters the behavior we want students to have.

Web-CAT is a web application built on top of Apple's WebObjects framework. Its Grader subsystem is designed in a language-neutral way, and presumes little about the actual process required for grading an assignment. It divides the grading process into a series of steps, with the number and nature of steps being fully tailorable by the instructor. The action(s) undertaken in each of the steps can be controlled through scripts or programs uploaded by the instructor. This approach allows the Grader to support everything from dumb file archiving with no automated assessment, to traditional output-based assessment of compiled or interpreted code, to any other approach desired.

The Web-CAT Grader's user interface employs a series of wizard-style pages to walk users through tasks. Figure 1 shows a typical user screen, in this case where a student is uploading a program assignment solution for evaluation. The system supports single-file submissions, as well as bundles of files archived together in common formats, such as zip and jar files. Once a student confirms a submission, it is processed in real time and the student is presented with the results in the web browser.


Web-CAT: View Assignment Results - Mozilla {Build ID: 2002121215}
File Edit View Go Bookmarks Tools Window Help Debug QA
Back Forward Reload Stop
file:///D:/Working/Research/Eista03/Edwards-screen-shot.h
Search
Print
Done

View Assignment Results

Back to Web-CAT Home

Score Summary	
Possible points:	50.0
Deductions:	-7.0
Early bonus:	0.0
Late penalty:	0.0
Final score:	**43.0**

Submission Details	
Project name:	p1-QuadTree
Submission no.:	9
File size:	14840
Submission time:	01/30/03 01:26PM
Deadline:	02/12/03 10:10AM
Late deadline:	02/14/03 10:10AM
Early bonus:	none
Late penalty:	20 points per 1 day late

Correctness Based on Your Tests

Your Program 93% **37 of 40** tests passed

Thoroughness of Your Testing

Your Test Cases 92% **92%** coverage, **40 of 40** tests valid

Score = (93% x 100% x 92%) x 50 = 43

Program Correctness (Your Solution)

```
tddpas.pl v1.2: Testing your submission using p1tests.txt

.....F
case 6 FAILED: empty string on the left
......F
case 13 FAILED: two empty strings
....................F
case 34 FAILED: merging two overlapping trees
......

Tests Run: 40, Errors: 0, Failures: 3 (92.5%)
```

Test Validity (Reference Solution)

```
tddpas.pl v1.2: Testing reference implementation using p1tests.txt

........................................

Tests Run: 40, Errors: 0, Failures: 0 (100.0%)
```

Figure 2. Assignment submission.

Processing the student's submission is the core of the assessment task. A set of instructor-controlled customization scripts have been developed to support TDD-based assignments. The evaluation process requires the instructor to provide a reference implementation that has been instrumented to collect coverage information. At present, branch coverage is being used as the desirable metric, although any other metric could be used instead.

The assessment process begins with compilation of the student's submission, if necessary. Next, the submission is executed against the student-supplied tests, which include both input values and expected output. In effect, the "test runner" capabilities embodied in the TDD support scripts takes the place of the test harness code necessary to set up and run tests, as well as compare results to detect failures. As a result, students do not have to write, provide, or even look at such a test harness infrastructure. For convenience to students, a corresponding downloadable script has been provided that will assume all the test harness responsibilities on the student end. This allows students to run their own tests on their own code without making a submission, if desired. While this does not give any feedback on test completeness or validity, it can be used almost anywhere, including computers that have no internet access.

To produce a grade, the Web-CAT Grader must assign values to the three scores described in Section 4. The code correctness score is computed from the number of student-written tests that are passed. The same set of tests is then run against the reference implementation. The test validity score is computed from the number of student-written tests that are "passed" by the reference implementation; a failed test on the reference implementation means, by definition, that the corresponding test case contains incorrect expected output for the given input. Finally, the test completeness score is computed from the coverage measure obtained from the reference implementation while it was executing the student tests.

In effect, this approach uses the instructor-provided reference implementation as an executable model of the problem. It serves as the oracle determining which tests are valid and which are not. In addition, it serves as the measurement device for determining how completely the full behavior of the problem has been covered. This view of the reference implementation may be problematic in some situations, and removing it as the model of required program behavior is an item for future work.

After this process completes, the student receives a summary screen providing feedback on the submission, as shown in Figure 2. The overall score is captured in two color-coded bars. The first bar indicates program correctness, and shows the percentage of student tests passed by the given program. This percentage also is reflected graphically in the proportion of the bar that is green. The second bar indicates the assessment of the student's test cases. The size of the bar indicates the completeness of testing achieved. The color of the bar indicates test validity: green means all tests were valid, while red means some tests were invalid. The three scores are multiplied together to give the final score for the student. Figure 4 illustrates an assignment worth 50 points.

The feedback received by the student also includes a log of the test cases that were run. Both the testing of the student program and the reference implementation run to assess test validity are shown. Successful tests are represented as a series of dots, and failed tests are explicitly identified so that the student can reproduce the problem and find the source of the error. The output format is based on the textual output produced by JUnit [JUnit, 2003].

6. EVALUATING THE EFFECT ON STUDENT PERFORMANCE

To assess the impact of this approach, the author has applied it in an undergraduate class. The course, CS 3304: "Comparative Languages," is a typical undergraduate programming languages course that serves as a junior-level elective for computer science students at Virginia Tech. Prior to the Spring 2003 semester, the Curator was used to collect and automatically grade student program submissions. In Spring 2003, the Web-CAT prototype was used. Students in the course receive part of their grade from four small programming assignments in three different programming languages exemplifying three separate programming paradigms.

The Spring 2001 offering of Comparative Languages included programming assignments in Pascal, Scheme, and Prolog. The same basic task was given in all three languages: given an EBNF grammar for a highly simplified version of English sentences, a student program was to read in candidate strings one per line, and print out the corresponding "diagram" of the sentence's grammatical structure as a parenthesized list. The mini language in the grammar consisted of only 15 words, including nouns, verbs, adverbs, adjectives, and prepositions, that could be combined into simple subject-verb-object constructions. Student programs were also required to produce specific error messages for input sequences that were grammatically incorrect or that contained inappropriate tokens. This example gives students practice in understanding EBNF, implementing simple recognition strategies, and producing basic parse tree representations. The first assignment in Pascal allowed students to grapple with the problem using an imperative programming approach with which they were familiar. By repeating the task in Scheme and Prolog, students could then concentrate more easily on mastering new paradigms.

6.1 Method

In Spring 2001, students submitted their solutions to the Curator. The Curator used an instructor-provided test data generator for grading. The test data generator produced a random input data set of 40 candidate lines for each submission for each student, together with the corresponding correct output. Generated test lines included 22.5% error cases covering the basic error modes covered in the assignment. Each student submission was compiled and run against such a randomly generated input data set. The resulting output was compared line for line against the correct output, and a score was computed by weighting the lines equally. Students were informed that their highest score would be the submission used for grading; in the case of ties, the earliest high score would be used. This strategy was an attempt to encourage students to document and format their code appropriately as it was developed, rather than saving these tasks until the code worked correctly.

Students in 2001 were also limited to a maximum of five submissions to the Curator on an assignment. This policy was typical of instructors using the Curator at our institution, because without it, students would never have to do any testing work on their own and instead could use the grading system as a testing service. In the feedback on each submission, the student received a complete copy of the randomly generated test data used, a copy of the correct expected output, a copy of the output produced by the student's submission, and a score summary. Thus, after one submission, the student could continually re-run the given test inputs on his or her own until satisfactory results were achieved before needing to resubmit to the Curator for further assessment.

The Curator score counted for one half of the student's program grade, with the other half coming from the teaching assistant's assessment of design, style, and documentation. In addition, students received a late penalty for submissions received after the assignment's deadline. The Curator deducted twenty points (out of 100) for late submissions received within 24 hours of the deadline, and forty points for submissions received 24-48 hours late. All student submissions, including the student's own code, the generated test data set, the student program's actual output, the expected output, and the time of the submission were all available for analysis, as was the student's final grade on each assignment.

In Spring 2003, the same set of assignments was used, but students used the new Web-CAT Grader for submissions. Students were allowed an unlimited number of submissions, but were required to submit test cases along with their programs. This change in policy on the number of submissions was founded on two ideas: supporting TDD, and "you get out what you put in." First, TDD encourages students to write tests and code together, and to constantly assess their evolving code base using the tests they have so far. The Curator uses the model of a student submitting a complete, finished program for grading. With the Web-CAT Grader, students were encouraged to submit partial results and to submit often to gauge their progress as their solution was developed, rather than waiting until a complete solution was finished. A small limit on the number of submissions would directly conflict with the goal of supporting TDD. Second, unlike the Curator—which generated new test data for each submission—the Web-CAT Grader did not perform any "testing" on its own. Instead, it simply ran the tests supplied by the student, and compared the results to the expected output described in each of the student's test cases. Thus, unlike the earlier approach, a student could only "get out what she put in" in the sense that the only way to learn more about the correctness of the solution was for the student to write more test cases.

Although "true" unit testing support such as that provided by the various XUnit frameworks would have been ideal, no such support was available for the languages used. As a result, test cases took the form of input/output pairs arranged in a text file for simplicity. Since sample programs for similar tasks from the course text book used a recursive descent approach, for most students the structure of the grammar in the assignment specification mirrored the structure of their solution. This made it easier for students to construct end-to-end input/output test cases that exercised individual "units" within their programs. Students were informed that they would not be allowed any more submissions once they received a perfect score. As in 2001, this encouraged students to document and format their code as it was developed, rather than saving these tasks until the code worked correctly. The Web-CAT score counted for one half of the student's program grade, with the other half coming from the teaching assistant's assessment of design, style, and documentation. Web-CAT applied the same late penalty deductions used in 2001 for submissions received after the assignment deadline. All student submissions, including the student's own code, the student's tests, the assessment information produced, and the time of the submission were all collected for analysis, as was the student's final grade on each assignment.

For the purposes of this paper, only student submissions for the first programming assignment in Pascal were compared. Since all students had been taught the imperative programming paradigm prior to this course as part of Virginia Tech's core curriculum, the main challenges in the first program were using a new language on a new problem. The challenges on later assignments differed, and hinged greatly on how well students were able to adapt to the functional or logic paradigms, making those assignments less generalizable to work in other classes. Examination of the first assignment also eliminates concerns about carryover effects in this case.

Both offerings of Comparative Languages were taught by the author. The same text book, course notes, and instructional style were used. Both classes consisted of approximately 60 students, primarily computer science majors in their junior year. Other than the students enrolled in the course, the central difference between the two semesters were in the submission, grading, and feedback approach used for programming assignments.

6.2 Experimental Results

The first step in assessing the relative performance of students in the two offerings of Comparative Languages is comparing assignment scores. Table 1 summarizes the score comparisons between the two groups. Because different scoring methods were used on the two groups of assignments, there are a number of alternative possibilities for comparing scores.

First, it is easy to compare the final scores recorded in class grade sheets, which include the automated assessment (half the score) and the teaching assistant's assessment (the remaining half), plus any late penalty deductions. These averages for the two course offerings are shown as "recorded grades" in Table 1. All statistical comparisons shown in Table 1 were performed using a two-tailed student's t-test assuming unequal variances between the two groups. Students using test-driven development in 2003 scored approximately half a letter grade higher than their predecessors.

Next, it is possible to separate out the TA assessment component of each student's grade from the "automated" component, ignoring all late penalties. There were no

Table I. Score Comparisons Between Both Groups (Bold Differences Are Significant)

Comparison	Spring 2001 Without TDD	Spring 2003 With TDD	t-score Assuming Unequal Variances	Critical t-value p = 0.05
Recorded grades	**90.2%**	**96.1%**	t(*df* = 62) = **2.67**	2.00
TA assessment	98.1%	98.2%	t(*df* = 65) = 0.06	2.00
Automated assessment	93.9%	94.0%	t(*df* = 71) = 0.07	1.99
Curator assessment	92.9%	96.4%	t(*df* = 71) = 1.36	1.99
Web-CAT assessment	**76.8%**	**94.0%**	t(*df* = 61) = **4.98**	2.00
Test data coverage	**90.0%**	**93.6%**	t(*df* = 69) = **5.14**	1.99
Web-CAT assessment without test coverage	**85.8%**	**100.0%**	t(*df* = 58) = **3.91**	2.00
Time from first submission until assignment due	**2.2 days**	**4.2 days**	t(*df* = 112) = **3.15**	1.98

significant differences between the two groups when comparing these components of the overall score individually. This indicates that relative to the level of scrutiny TAs provide, it is unlikely that there were any differences clearly discernable by reading the source code. This result is consistent with the observations of others that grading program correctness by reading the source code is difficult and error-prone [Luck & Joy 1999]. The averages for "automated assessment," however, indicate only that the evaluation scores produced by the Curator in 2001 were similar to the scores produced by Web-CAT in 2003. Since the two evaluation approaches were measuring different facets of student work, however, these scores are not directly comparable. Further, because there were no significant differences in either TA assessment scores or automated assessment scores, the primary factor in the differences between the recorded grades came from late penalties, which are discussed later in this section.

Since the two course offerings used different scoring formulas, all assignments were re-scored using both approaches to provide a better basis for comparison. First, all submissions from 2003 were run through the original Curator grading setup used in 2001. Student programs from this offering were thus tested against the random test data generator used in the prior offering. The "curator assessment" comparison in Table 1 shows the average scores received using the 2001 grading scheme across all programs from both groups. While the average for 2003 students is slightly higher than that for 2001 students,
the difference is not statistically significant. One possible interpretation for this situation is that, if any difference exists between the code produced by the two groups, the assessment approach used in 2001 was not sensitive enough to detect it.

Second, all submissions from 2001 were run through the Web-CAT Grader. Web-CAT requires each submission to be accompanied by test cases, however, and students in 2001 did not write their own test cases. Instead, when each student program was originally graded in 2001, a randomly generated test data set was produced by the Curator. The generator was designed to provide complete problem coverage, and essentially used the grammar describing valid input as a probabilistic generator. Thus, each student's automatically generated test set was fed along with the corresponding submission into the Web-CAT Grader. Table 1 shows the result for the "Web-CAT assessment," indicating that students in 2003 outperformed students in 2001 under the 2003 grading scheme.

Since students in 2003 were given explicit feedback about how thoroughly they were testing all aspects of the problem specification, they had an opportunity to maximize the completeness of their tests to the best of their ability. This difference in feedback on the completeness of testing raises the question of possible differences between the student-generated test cases in 2003 and the test cases produced by the instructor-provided generator in 2001. In particular, how well do students measure up against a tool designed by a professional? The "test data coverage" comparison in Table 1 shows how the randomly generated test sets from 2001 compare to the student-written tests from 2003, in terms of measured condition/decision coverage on the instructor's reference solution. Students outperformed the generator at producing test cases and achieved higher test coverage scores.

Since test coverage scores were used in Web-CAT's grading formula, it is possible that the lower coverage of the randomly generated test sets might artificially lower the scores of 2001 submissions on the Web-CAT Grader. To explore this issue, Web-CAT scores for all submissions were recalculated without including test coverage, so that only test pass rates and test validity were used to compute each score. However, the "Web-CAT assessment without test coverage" comparison still reveals a striking difference between the two groups. Such a result was expected, since students in 2003 received explicit feedback on which of their tests were passed and which were invalid. Thus, any student putting in effort easily could maximize both of these aspects of their score, leaving test coverage as the primary factor in determining their program grade. This behavior was clearly evident in the data, where all but two students from 2003 received perfect scores when test coverage was removed from consideration.

While analyzing score data explores some aspects of the experiment, other facets are also of interest. In particular, did the use of test-driven development encourage students to start programming or making submissions earlier? Figure 3 shows the distribution of times for the first submission of each student, in terms of hours ahead of the due date and time. Students in both course offerings faced a stiff score penalty for submitting an assignment after the deadline. Because students in 2001 were limited to only five submissions, many waited until they had their program "complete" before submitting. This appeared to be a result of expectations rather than policy, since the Curator did not prevent submissions of partially operating code. In fact, a creative student could submit a simple "hello world" program before starting a solution and then receive a complete set of randomly generated test data and the corresponding expected output in reply. The student then could use this data repeatedly during development of a solution, providing as many cycles of self-assessment as desired until the student judged another Curator submission was prudent. While this approach is possible, few if any students took this path.

In 2003 on the other hand, students were encouraged to submit code early and often, even when it was only partially implemented. While some students always start early, Figure 3 shows that in 2001 more students waited until the last two days to submit assignments. Table 1 shows the average "time from first submission until assignment due" for the two groups. In 2001, a student's first submission was 2.2 days (53 hours) before the deadline on average. In contrast, students in 2003 had an average initial submission time of 4.2 days (101 hours) before the deadline.

More information can be gleaned by comparing the number of programs that were submitted late, after the deadline had passed. Students were assessed a 20% deduction per day late for such programs, which were accepted up to two days late. Table 2 shows the number of late submissions received in each group. Note that there were **no late submissions in 2003**, compared with **15% turned in late in 2001**. This is the source of

Table II. Comparison of On-Time and Late Submissions Between the Groups

Status	Spring 2001 Without TDD	Spring 2003 With TDD
On time submissions	50	57
Late one day	5	0
Late two days	4	0

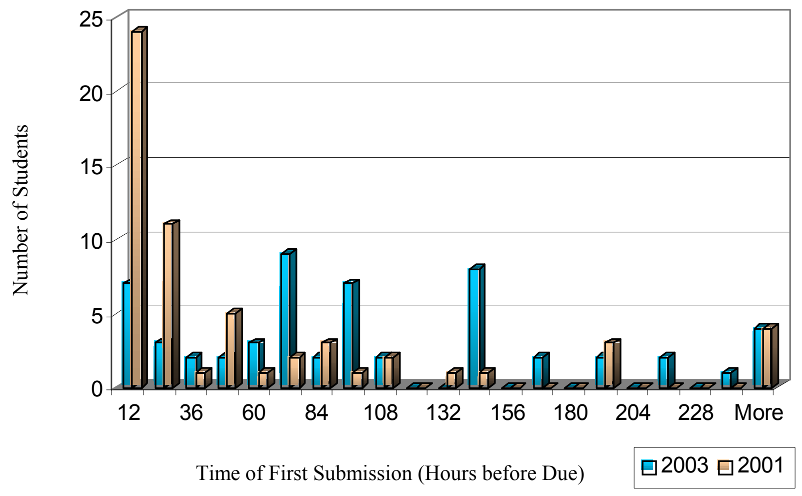


Figure 3. Distribution of initial submission times for the two groups.

the statistically significant difference in recorded grades shown in Table I. The chi-squared value for the categorical data shown in Table II is 9.46 (*df* = 2), well above the critical value of 5.99 for p = 0.05. As a result, the proportion of students submitting assignments late between the groups is statistically significant. In other words, rather than simply starting earlier, students in 2003 were more likely to *finish* their programs earlier.

While the use or encouragement of TDD may be a factor in the reduction in late submissions, the change in policy to allow unlimited submissions may be more critical. However, it is important to note that allowing unlimited submissions was only considered pedagogically appropriate because of the change in the assessment strategy. Using the Web-CAT Grader, students could no longer sit back and let the grading system do all the testing work. To learn more about how completely or correctly a solution matches the problem, the student must write more tests: "you only get out what you put in."

6.3 WHAT ABOUT CODE QUALITY?

While the experimental analysis of program scores and submission times is encouraging, it is also important to examine the code produced by students for objective differences in quality. The goal of teaching software testing to undergraduates is to enable them to test software better, and thus hopefully produce code with fewer bugs. The normal industry measure for software "bugginess" is defect density: the number of defects (bugs) per thousand non-commented source lines of code (often abbreviated KNCSLOC, or just KSLOC). An exploration of bug densities in final programs submitted by students in this experiment was undertaken.

Unfortunately, quantifying bugs is hard. In industry, defect tracking systems are routinely used to record identified defects, assign responsibility for addressing them, and track them to resolution. Data from such systems can be invaluable in computing measures such as defect density. In a student programming assignment, however, no such information is available. The naïve approach of attempting to debug every program under consideration and count up the number of bugs found is intractable. Such an approach is extremely manpower intensive. Further, when a student fails to implement a significant chunk of behavior in their solution (perhaps because they ran out of time), how can one quantify the "number of defects" represented by such an omission?

To address this concern, a two phase approach was used. First, a comprehensive master test suite was developed. The goals for this test suite were to produce a collection of tests that covered all aspects required by the assignment, and to include no redundant tests—that is, test cases that duplicated the same features. This suite was produced by hand-generating a core set of 54 test cases that provided 100% condition/decision coverage of the instructor's reference implementation. To this, all of the unique tests used in grading all 118 student submissions were added: 2,891 Curator-generated test cases from 2001 submissions and 47,127 student-written test cases from 2003 submissions. The result was a massive test suite with 50,072 test cases. While this set seemed to cover the entire problem well, it clearly included many redundant test cases.

To eliminate redundant test cases, the entire suite was run against every program in the experiment. Two tests were considered redundant if no program in the experiment produced different pass/fail results for the two cases. By tabulating case-by-case results for each program, duplicates were eliminated. This reduced the master suite to 1064 test cases. For any pair of test cases in the reduced set, there was at least one "witness" among the programs that passed one but failed the other. Note that this does not guarantee that the suite is "orthogonal," that is, without any significant overlap between test cases. For a given program, several of the test cases might vary in a way that is only detectable at exactly the same point in that program. Those test cases would then be considered overlapping for that program. Here, a weaker notion of orthogonality, relative to the entire collection of programs as a whole, was used. Ironically, none of the original cases hand-written by the course staff were included, since all were redundant with student-written test cases.

All programs were then run against the reduced master test suite. By removing as much redundancy as possible from the master suite, there is greater confidence that each failed test case corresponds to some actual defect or discernable difference in the program. For each program from both groups, the total number of test cases failed (out of 1064) was calculated. Table 3 summarizes the results for the two groups, showing the mean number of test case failures per program under each of the two conditions. On average, programs developed in Spring 2003 failed about 125 fewer test cases each when compared with those from 2001, a reduction of 32%. Figure 4 provides a graphical summary of the comparison, showing the relationship between percentile ranks of individual programs and the corresponding number of test case failures for that program. For example, at the 50th percentile, half the programs submitted in 2001 failed 388 or fewer test cases, while half the programs submitted in 2003 failed 271 or fewer test cases. At every point along the spectrum, students in 2003 produced less-buggy work. Notably, while the best-performing submission from 2001 still failed 213 test cases, two submissions from 2003 passed **every** test case, and a full 20% of the 2003 submissions failed two or fewer test cases out of 1064. Failing two or fewer test cases in this scenario translates into a defect rate of approximately 4 defects per KSLOC, which is comparable to most commercial-quality software written in the United States.

Table III. Defect Comparisons Between Groups (Bold Differences Are Significant)

Comparison	**Spring 2001 Without TDD**	**Spring 2003 With TDD**	**t-score Assuming Unequal Variances**	**Critical t-value p = 0.05**
Average test case failures from master suite (out of 1064)	**390 (36.7%)**	**265 (24.9%)**	t(*df* = 84) = **3.48**	1.99
Estimated Defects/KSLOC	**54.0**	**38.9**		

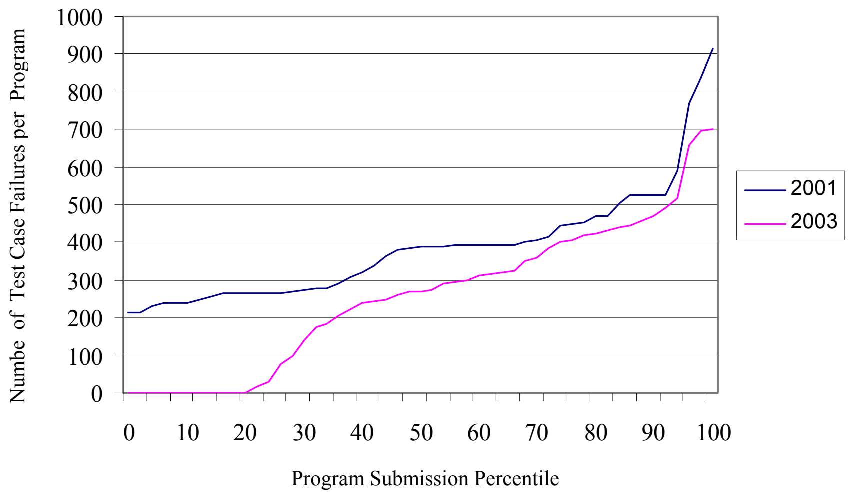


Figure 4. Distribution of test case failures by program percentiles.

Using such a master test suite makes it possible to directly quantify the proportion of behavior that is correctly implemented by a student solution. This reduces the problem of estimating bugs/KSLOC to determining the relationship between test case failures in the master suite and latent bugs hidden in student programs. To identify this relationship, a group of nine programs were randomly selected from each course offering. All 18 programs were then debugged by hand until they passed all test cases in the master suite. The programs were then stripped of comments and blank lines. Differences between the debugged version and the original were measured by counting the number of source code lines modified, new source lines added, and old source lines deleted. This line-oriented count was then used as the "defect count" for the corresponding program.

Next, a linear regression analysis was performed between the number of test cases in the master suite failed and the defect count obtained through by-hand debugging for the 18 programs in the random sample. A statistically significant linear relationship was found with a correlation coefficient of 0.755, and with *F*(1, 17) = 21.2, compared to the

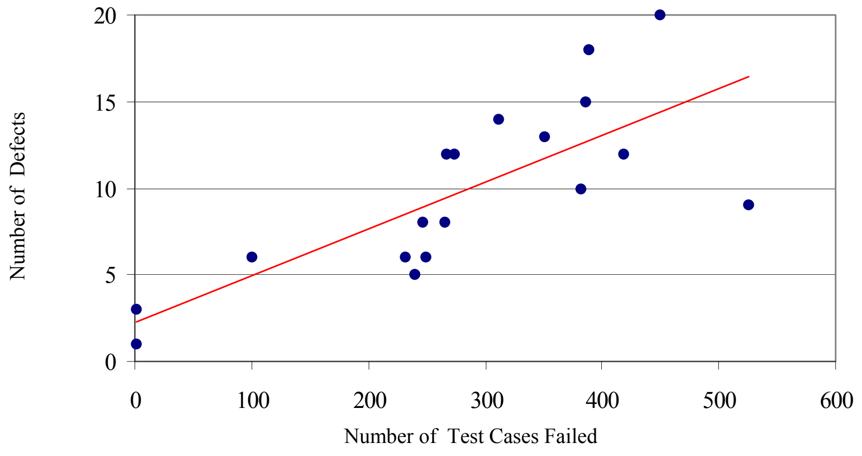


Figure 5. Relationship between test case failures and number of latent defects.

critical value of 8.40 at the p = 0.01 level. Figure 5 shows the corresponding scatter plot and regression line. Alternative models were explored, including multiple regression including program size and treatment codition, as well as polynomial regression, but a simple linear regression produced the best fit.

The resulting linear equation was used to estimate the defect densities—that is, number of defects normalized by program size—in the remainder of the population. Table III shows the average defect densities for the two groups generated by this process. It is worth noting that industrial data, especially from larger defense sector companies, suggest typical defect densities of four to seven defects per thousand lines of code are common in a production environment. While the average numbers achieved by students are poor in comparison, remember that these are small assignments produced by individuals without any commercial experience and without any independent testing or review. More importantly, the data indicate that students in 2003 produced code with **approximately 28% fewer defects** per thousand lines.

6.4 Student Perceptions

In addition to collecting student programs, information on student perceptions and opinions were also gathered. Before any programming assignments were given in the 2003 offering, students were asked about their current testing practices on a written homework assignment. Students were asked to describe how they currently test their programs; some sample responses give an indication of typical student testing practices:

- "I run them through some simple tests to ensure that it is operating as expected. But for the most part I have always relied on supplied test data."
- "I don't think about test cases until I am confident my program is 100% working. Of course, it almost never is …"

- "I usually write the whole thing up and then start doing rapid-fire tests of everything I can think of."

Students were also asked their feelings on using test-driven development once it had been described in class:

- "I am very excited about using TDD."
- "I agree that TDD can be beneficial and I'm glad we are being required to experiment with it in this course."
- "If it increases the effectiveness of my programming and decreases the time I spend debugging, then I am all for it."
- "[Previously,] I had to quit my detailed testing and stick to making the program appear to work with the sample data given every time a deadline drew near. With [TDD], the tests are such an integral part of the project that no time-conserving measure will save me."

Toward the end of the course, students also completed a survey of their opinions and attitudes about the approach. Table 4 summarizes the responses received from students. A five point Likert scale was used to elicit opinions. Students agreed that the Web-CAT Grader was easier to use than the Curator and that it provided excellent support for TDD. More importantly, however, students clearly perceived the benefits of TDD on assignments, agreeing that it increased confidence when making changes to programs and increased confidence in correctness of the result. Students also felt that TDD made them more systematic in devising tests and made them more thoroughly test their solutions. Most importantly, a majority of students expressed a preference for using the approach and tool in future classes, *even if it were not required*.

Table IV. Student Survey Responses

Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	Average
1. Web-CAT is easier to use than the Curator system	0	5	17	21	5	3.5
2. Results produced by Web-CAT are more helpful in detecting errors in my program than those produced by the Curator	0	1	4	25	19	4.3
3. Web-CAT provides better help features than the Curator	0	0	22	16	9	3.6
4. Using TDD increases my confidence in the correctness of my programs	1	5	11	14	18	3.9
5. Using TDD helps me complete my programming assignments earlier	6	11	20	10	2	2.8
6. Using TDD increases my confidence when making changes to my programs	0	4	12	24	9	3.8

Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	Average
7. TDD increases the amount of time I need to complete programming assignments	1	18	14	11	5	3.0
8. Using TDD makes me test my own solution more thoroughly	0	5	7	28	9	3.8
9. Using TDD makes me take a more systematic approach to devising tests	0	5	8	30	6	3.8
10. With TDD, I spend more time writing code.	0	18	28	3	0	2.7
11. With TDD, I spend more time writing tests	0	2	9	27	11	4.0
12. With TDD, I spend more time debugging code	1	17	21	8	1	2.8
13. In the future I am more likely to use TDD even if it is not required by the assignment	2	8	19	16	4	3.2
14. As a result of using TDD in this class, I am now able to write better test cases	1	5	10	24	9	3.7
15. Using TDD adversely affected my grade	6	20	14	7	2	2.6
16. Without using TDD, I would have scored higher on this assignment	8	21	10	6	4	2.5
17. The tddpas.pl script distributed for student use was hard to use	14	20	8	4	2	2.1
18. I preferred using Web-CAT instead of the provided tddpas.pl script I could run myself	10	16	7	14	2	2.6
19. Even if it were not required, I would like to use Web-CAT to test my programs for class before turning them in	0	3	10	28	8	3.8
20. The Web-CAT environment provides excellent support for programming using TDD	0	1	7	29	12	4.1

Students were also given open-ended questions on the survey regarding what they found most and least useful about the approach and the tool. Thirty-three percent of students requested more direct feedback about how they could improve their testing scores (i.e., test coverage). Sixteen percent requested improving the stability of the prototype. Twelve percent requested faster submission screens with fewer mouse clicks required for navigation. Ten percent praised the amount of assurance they received from the process about the correctness of their work *while developing*. Finally, eight percent said that feedback on which specific test cases failed was the most useful aspect of the system.

7. FUTURE WORK

The success of this preliminary experience with the Web-CAT Grader has energized interest in using it across other courses in our curriculum. We intend to introduce the technique in Fall 2003 in Virginia Tech's CS1 course, and then spread it to others. With this expansion in mind, several possible extensions and enhancements to the approach are underway.

First, Virginia Tech's CS1 course will be taught in Java starting in Fall 2003. As a result, Web-CAT will be modified to support JUnit-style test cases [JUnit, 2003]. In addition, rather than measuring test thoroughness by instrumenting a reference implementation, student code will be directly instrumented to collect coverage data. In Java, a tool such as Clover can be used [Cortex, Inc., 2003]. Further, we plan to explore alternatives to using a reference implementation as a proxy representation of the problem to be solved. Instead, it should be possible to use an instructor-provided reference test suite. Students need not be exposed to the test suite at all. Instead, failure to pass one or more tests in the reference suite indicates that the student has not fully tested his or her own code, or that the student's solution fails to completely implement all required behavior. The proportion of reference test cases failed can be used to estimate the completeness of the student implementation.

Another problem that must be addressed has to do with validating student tests. In particular, how will students find out if their test cases are incorrect? In the Web-CAT prototype, the instructor's reference implementation served this purpose, and it was possible to validate all student tests because all tests necessarily were executable on the reference implementation. When students begin to use JUnit or other approaches to test the internal structure of their own solutions, many tests will be structurally specific, and may not apply to a reference implementation. On the other hand, it is desirable to assess the validity of test cases where ever possible so that students can spot their own errors earlier. A solution to this issue is needed.

Further, as student comments indicate, it is important to be able to give students concrete feedback on how to improve their testing. Fortunately, some test coverage tools provide detailed feedback reports that indicate which parts of the code under consideration were executed. Clover, for example, can generate a color-coded HTML rendering of the source code that highlights the source lines or control structure decision points that have not been executed. This form of feedback will be added to the Web-CAT Grader so that students can receive concrete feedback about which portions of their code require further testing [Edwards, 2003].

It is also possible to add static analysis tools to the assessment approach. Checkstyle, for example, is an open source Java tool that provides a large suite of style-oriented checks on source code [Checkstyle, 2003]. PMD is a similar program that focuses on potential coding errors [PMD, 2003]. We plan to experiment with integrating this form of feedback into the assessment performed by Web-CAT. This will allow automatic grading of simple stylistic issues, presence and proper use of JavaDoc comments, and identification of potential coding errors. Students will be able to receive this feedback quickly, as often as desired, and make modifications in response.

8. CONCLUSIONS

The goal of the work described here 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. The strategy is to give students basic exposure to test-driven development, and then provide an automated tool that will assess student submissions on-demand and provide the feedback necessary. This approach has been demonstrated in an undergraduate programming languages course using a prototype tool. The results have been extremely positive, with students expressing clear appreciation for the practical benefits of TDD on programming assignments, and with their code showing a 28% reduction in defects per thousand lines of code.

This strategy succeeds by addressing all of the challenges in Section 4. Unlike existing automated grading systems, this approach does not focus on output correctness. Instead, it places the responsibility to demonstrate correctness on the students, and empowers them by using their tests to determine the assessed correctness of their program solution. It directly rewards students for desired testing behavior by explicitly including a measure of test validity and of test completeness in the scoring model. The result is that students do more testing, and appreciate its value more.

At the same time, it is clear that this approach presents testing as a valuable, integrated activity in the student programming process, not yet another bureaucratic burden added to student duties. Students also receive clear, immediate feedback on the success and quality of their testing efforts. With future enhancements, they can also receive concrete suggestions for where to improve their testing, as well as detection of common coding mistakes. In addition to this feedback, it is clear that students can see real value to using the approach. This value comes in the form of increased confidence in solution correctness, increased confidence when making changes or modifications to code, and the assurance of always having a “running version” ready to go as the solution is being developed incrementally. These benefits, together with scoring incentives, encourage and reinforce the behavioral changes that are desired.

It is also important to note that this approach imposes additional responsibilities for the course staff who are writing programming assignments. Assignments must be clearly defined, especially with regard to the details of input and output [Luck & Joy, 1999]. As a practical matter, it is also necessary to construct a reference solution that has been thoroughly tested. Although this may involve more preparation time than some instructors currently invest, one can argue that this work is necessary to fully smoke test a new programming assignment even when automated grading is not employed. Compared to other automated grading approaches, however, preparing assignments for TDD-based grading imposes only a small amount of additional effort, if any. At Virginia Tech, setting up Web-CAT assignments takes approximately the same amount of preparatory work as setting up a more traditional Curator assignment where TDD is not used.

While these results are preliminary, they indicate significant potential value in this strategy. As a result, Virginia Tech plans to expand its use of TDD in the classroom, including an exploration of its use beginning as early as the first programming course. A long-term vision of TDD in the classroom across the computer science curriculum has already been articulated, combining it with other educational techniques, such as lab-based teaching, pair programming, and active learning approaches that support reflection in action [Edwards 2003]. As the Web-CAT Grader matures, it will also be made available to faculty at other institutions. Further studies of effectiveness in mainstream computer science courses will help refine the approach and build toward a different understanding of how to assess student programming activities.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant DUE-0127225, and by a research fellowship from Virginia Tech's Institute for Distance and Distributed Education. Any opinions, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or IDDL. I also wish to acknowledge Anuj Shah, Amit Kulkarni, and Gaurav Bhandari,, who implemented many of the features of the automated grading and feedback system described here.

REFERENCES

ALLEN, E., CARTWRIGHT, R., AND REIS, C. 2003. Production programming in the classroom. In *Proc. 34th SIGCSE Technical Symp. Computer Science Education*, ACM Press, 89-93.

AR, S., AND CAI, J.-Y. 1994. Reliable benchmarks using numerical instability. In *Proc. 5th Annual ACM-SIAM Symp. Discrete Algorithms*, Society for Industrial and Applied Mathematics, 34-43.

BAGERT, D., HILBURN, T., HISLOP, G., LUTZ, M., MCCRACKEN, M., AND MENGEL, S. 1999. Guidelines for Software Engineering Education Version 1.0, Technical Report CMU/SEI-99-TR-032, Software Engineering Institute, Pittsburg, PA.

BECK, K. 2001. Aim, fire (test-first coding). *IEEE Software*, 18, 5, 87-89.

BECK, K. 2003. *Test-Driven Development: By Example*. Addison-Wesley, Boston, MA.

BEIZER, B. 1990. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY.

BOEHM, B. 1976. Software engineering. *IEEE Transactions on Computers*, C-25, 12, 1226–1241.

BOURQUE, P., DUPUIS, R., ABRAN, A., AND MOORE, J.W., eds. 2001. Guide to the Software Engineering Body of Knowledge—Stone Man Trial Version 1.00, IEEE Computer Society, Washington, available at: <http:/www.swebok.org>.

BUCK, D., AND STUCKI, D.J. 2000. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. In *Proc. 31st SIGCSE Technical Symp. Computer Science Education*, ACM press, 75-79.

CHECKSTYLE. 2003. Checkstyle home page. Web page last accessed Mar. 21, 2003.
<http://checkstyle.sourceforge.net/>.

CORTEX, INC. 2003. Clover: a code coverage tool for Java. Web page accessed Mar. 21, 2003.
<http://www.thecortex.net/clover/>.

EDWARDS, S.H. 2003. Rethinking computer science education from a test-first perspective. In *Addendum to the 2003 Proc. Conf. Object-oriented Programming, Systems, Languages, and Applications* (Educator's Symposium), to appear.

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

HARROLD, M.J. 2000. Testing: A road map. In *The Future of Software Engineering*, A. Finkelstein, ed., ACM Press, New York, NY, 61–72.

HILBURN, T.B., AND TOWHIDNEJAD, M. 2000. Software quality: A curriculum postscript? In *Proc. 31st SIGCSE Technical Symp. Computer Science Education*, ACM Press, 167-171.

ISONG, J. 2001. Developing an automated program checker. *J. Computing in Small Colleges*, 16, 3, 218-224.

JACKSON, D., AND USHER, M. 1997. Grading student programs using ASSYST. In *Proc. 28th SIGCSE Technical Symp. Computer Science Education*, ACM Press, 335-339.

JONES, E.L. 2000a. Software testing in the computer science curriculum—a holistic approach. In *Proc. Australasian Computing Education Conf.*, ACM Press, 153-157.

JONES, E.L 2000b. SPRAE: A framework for teaching software testing in the undergraduate curriculum. In *Proc. ADMI 2000*, Hampton, VA, 1-4 June 2000.

JONES, E.L. 2001a. Integrating testing into the curriculum—arsenic in small doses. In Proc. 32nd SIGCSE Technical Symp. Computer Science Education, ACM Press, 337-341.

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

JONES, E.L. 2001c. Grading student programs—a software testing approach. *J. Computing in Small Colleges*, 16, 2, 185-192.

JUNIT. 2003. JUnit home page. Web page last accessed Mar. 21, 2003. <http://www.junit.org/>.

LUCK, M. AND JOY, M. 1999. A secure on-line submission system. *Software—Practice and Experience*, 29, 8, 721-740.

NAGAPPAN, N., WILLIAMS, L., FERZLI, M., WIEBE, E., YANG, K., MILLER, C., AND BALIK, S. 2003. Improving the CS1 experience with pair programming. In *Proc. 34th SIGCSE Technical Symp. Computer Science Education*, ACM Press, 359-362.

MCCAULEY, R., ARCHER, C., DALE, N., MILI, R., ROBERGÉ, J., AND TAYLOR, H. 1995. The effective integration of the software engineering principles throughout the undergraduate computer science curriculum. In *Proc. 26th SIGCSE Technical Symp. Computer Science Education*, ACM Press, 364-365.

MCCAULEY, R., DALE, N., HILBURN, T., MENGEL, S., AND MURRILL, B.W. 2000. The assimilation of software engineering into the undergraduate computer science curriculum. In *Proc. 31st SIGCSE Technical Symp. Computer Science Education*, ACM Press, 423-424.

McQuain, W. 2003. Curator: An electronic submission management environment. Web page last accessed July 24, 2003. <http://www.cs.vt.edu/curator/>.

MENGEL, S.A., YERRAMILLI, V. 1999. A case study of the static analysis of the quality of novice student programs. In *Proc. 30th SIGCSE Technical Symp. Computer Science Education*, ACM, 78-82.

PMD. 2003. PMD home page. Web page last accessed Mar. 21, 2003. <http://pmd.sourceforge.net/>.

REEK, K.A. 1996. A software infrastructure to support introductory computer science courses. In *Proc. 27th SIGCSE Technical Symp. Computer Science Education*, ACM Press, 125-129.

RICADELA, A. 2001. The state of software: Quality. *InformationWeek*, 838, 43, May 21, 2001.

SHEPARD, T., LAMB, M., AND KELLY, D. 2001. More testing should be taught. *Communications of the ACM*, 44, 6, 103–108.

WILLIAMS, L., UPCHURCH, R.L. 2001. In support of student pair-programming. In *Proc. 32nd SIGCSE Technical Symp. Computer Science Education*, ACM Press, 327-331.

WILSON, R.C. 1995. *UNIX Test Tools and Benchmarks*. Prentice Hall, Upper Saddle River, NJ.

Received April 2004; accepted August 2004.