Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action

Stephen H. Edwards
Virginia Tech, Dept. of Computer Science
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061 USA
+1 540 231 5723
edwards@cs.vt.edu

ABSTRACT

Introductory computer science students rely on a *trial and error* approach to fixing errors and debugging for too long. Moving to a *reflection in action* strategy can help students become more successful. Traditional programming assignments are usually assessed in a way that ignores the skills needed for reflection in action, but software testing promotes the hypothesis-forming and experimental validation that are central to this mode of learning. By changing the way assignments are assessed—where students are responsible for demonstrating correctness through testing, and then assessed on how well they achieve this goal—it is possible to reinforce desired skills. Automated feedback can also play a valuable role in encouraging students while also showing them where they can improve.

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

Verification

Keywords

Pedagogy, test-driven development, CS1, extreme programming, automated grading.

1. INTRODUCTION

Despite our best efforts as educators, student programmers continue to develop misguided views about their programming activities, particularly during freshman and sophomore courses:

- Once the compiler accepts my code without complaining, I have removed all the errors.
- Once my code produces the output I expect on a test value or two, it will work well all the time.
- My code looks "correct" to me. If it produces the wrong answer, that does not make sense, so there must be something hidden that I do not understand about my code. I will try switching around a few things to see if I can make the problem go away.
- Once my code gives the correct answer for the instructor's sample data, I am finished.

While many computer science students acquire a more balanced view of software development as they learn, other students do not reach such a perspective for many semesters, and some never do so. This situation places both the student and the educator at a significant disadvantage. Anecdotally, many educators report difficulties along these lines [12, 8, 5].

Computer science students will be more successful at learning if they move from this *trial and error* approach to practicing *reflection in action*. "Reflection in action," as originally described by Schön [13], is a characterization of how practitioners complete tasks in the face of uncertainty and novelty. When a technique or part of a solution fails to work, difficulties or confusion cause the practitioner to switch to a reflective mode, examining both the phenomenon at hand and also prior understandings that may have been implicit in his or her behavior. From this reflection, the practitioner then "carries out an experiment which serves to generate both a new understanding of the phenomenon and a change in the situation" [13]. This on-going experimentation is central to finding a viable solution when past experiences do not work in a new context without modification.

Many educators would agree that steering students toward reflection in action is a desirable goal, but typical programming assignments are poor devices for promoting this behavior. Students receive feedback only on the end result they produce and tend to equate a program that "produces the right output" with an "effective solution." The learning process matters little in grade outcomes, and students only receive indirect feedback on what and how they learn via comments on their final solution. Students are often able to succeed at simpler CS1 and CS2 assignments using a trial-and-error approach, which only reinforces a strategy that will handicap their performance in more advanced courses.

This situation can be improved through careful use of *software testing* in programming assignments. From the very first programming activities in CS1, a student should be given the responsibility of demonstrating the correctness of *his or her own* code. Such a student is required to submit test cases for this purpose along with the code. While coding design and style are typically

SIGCSE'04, March 3–7, 2004, Norfolk, Virginia, USA.

assessed using an independent reading of the source code, we must change the way we assess program correctness. Rather than assessing student performance on *whether their programs produce the correct output*, students should be meaningfully assessed on *how well they have demonstrated the correctness of their program* through testing, that is, how correctly and thoroughly their tests conform to the problem.

2. WHY STUDENTS STICK WITH TRIAL AND ERROR

Trial and error is a well-established technique for beginners in any discipline, and it is no surprise that this is where students start out. But why do students persist in this practice long after it becomes a handicap? Buck and Stucki describe one possible reason [4, 5]: most undergraduate curricula focus on developing program *application* and *synthesis* skills (i.e., writing code), primarily acquired through hands-on activities. In addition, students must master basic *comprehension* and *analysis* skills. Without these skills, they are poorly equipped for any strategy beyond trial and error.

Bloom's taxonomy describes six increasing levels of cognitive development that can be used to frame and organize learning objectives, labeled in increasing order of sophistication as: knowledge, comprehension, application, analysis, synthesis, and evaluation. Buck and Stucki provide a concise description of Bloom's taxonomy in a CS education context [4]. Bloom's work suggests that students must master basic comprehension and analysis skills as a prerequisite for effective program writing. Students must develop their abilities in reading and comprehending source code, envisioning how a sequence of statements will behave, and predicting how a change to the code will result in a change in behavior. Yet typical undergraduate curricula focus first and foremost on *writing programs*: application and synthesis skills.

Many educators try to foster comprehension and analysis abilities through code reading assignments or requiring students to manipulate and reason about non-code artifacts [12]. Buck and Stucki propose an "inside/out" pedagogy for introducing CS1 concepts in a manner inspired by Bloom's levels [4, 5]. While this is a powerful approach in organizing assignments, their focus has been on appropriately situating code writing tasks in a context that constrains and directs students as they learn. Others have added small analytical tasks to regular lab assignments [6].

To advance to reflection in action, however, students need more than just an ability to predict how changes in code will result in changes in behavior. In addition, they need continually reinforced practice in hypothesizing about the behavior of their programs and then experimentally verifying (or invalidating) their hypotheses. Further, students need frequent, useful, and immediate feedback about their performance, both in forming hypotheses and in experimentally testing them.

These activities are at the heart of software testing. To write an effective test, students must do more than just come up with a sequence of code actions—they must also hypothesize what resulting behavior they expect. Yet, in most mainstream CS curricula, students get little feedback on their performance in this area. This idea is complementary to Buck and Stucki's focus on the middle levels of Bloom's taxonomy—without mastering those levels, students cannot effectively test. However, while mastering those levels is necessary for a student to move toward reflection in action, it is not sufficient. Basic software testing provides the experience and setting for natural, recurring hypothesis testing that is important for reflection in action.

At the same time, however, there are **five perceived roadblocks** to adopting software testing practices in assignments:

1. Software testing requires experience at programming, and may be something **introductory students are not ready** for until they have mastered other basic skills.
2. Instructors just do not have the time (in terms of lecture hours) to **teach a new topic** like software testing in an already overcrowded course.
3. The course staff already has its hands full assessing program correctness—it may not be feasible to **assess test cases** too.
4. To learn from this activity, students need **frequent, concrete feedback** on how to improve their performance at many points throughout their development of a solution, rather than just once at the end of an assignment. The resources for rapid, thorough feedback at multiple points during program writing just are not available in most courses.
5. 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.

By combining a suitable testing technique with the right assessment strategy, and supporting them with the right tools, including an automated assessment engine, it is possible to overcome all five of these difficulties.

3. TEST-DRIVEN DEVELOPMENT

To include software testing in student assignments, one must first choose a testing approach in which students will be instructed. Unfortunately, students are likely to view the software testing methods in most student-oriented software engineering texts as something that professional programmers do "out in the real world" but that has little bearing on—and provides little benefit for—the day-to-day tasks required of a student. In this case, the practice of *test-driven development* (TDD) is a better pedagogical match. TDD has been popularized by extreme programming [2]. TDD is a practical, concrete technique that students can practice on their own assignments. In TDD, one always writes one or more test cases before adding new code. The test cases capture what behavior you are attempting to produce. Then, as you write new code, these tests tell you when you have achieved your latest (small) goal.

TDD is attractive for use in an educational setting for many reasons. It is easier for students to understand and relate to than more traditional testing approaches. It promotes incremental development, promotes the concept of always having a "running version" of the program at hand, and promotes early detection of errors introduced by coding changes. It directly combats the "big bang" integration problems that many students see when they begin to write larger programs, where testing is saved until all the code writing is complete. It dramatically increases a student's confidence in the portion of the code they have finished, and allows them to make changes and additions with greater confidence because of continuous regression testing. It increases the stu-

dent's understanding of the assignment requirements, by forcing them to explore the gray areas in order to completely test their own solution. It also provides a lively sense of progress, because the student is always clearly aware of the growing size of their test suite and how much of the required behavior has already been completed. Most importantly, students begin to see these benefits for themselves after using TDD on just a few assignments.

The tool support that is available for TDD is also important. TDD frameworks are readily available, including JUnit [10] for Java, and related XUnit frameworks for other languages. Although these frameworks are aimed at professional developers, similar educational tool support is also becoming available. For example, DrJava [1], which is designed specifically as a pedagogical tool for teaching introductory programming, provides built-in support to help students write JUnit-style test cases for the classes they write. Similarly, BlueJ [11], another introductory Java environment designed specifically for teaching CS1, also provides support for JUnit-style tests. BlueJ allows students to interactively instantiate objects directly in the environment without requiring a separate main program to be written. Messages can be sent to such objects using pop-up menus. BlueJ's JUnit support allows students to "record" simple object creation and interaction sequences as JUnit-style test cases. Such tools make it easy for students to write tests from the beginning, and also mesh nicely with an objects-first pedagogy.

4. AUTOMATED GRADING

Providing appropriate feedback and assessment of student performance is critical. Many educators have used automated systems to assess and provide rapid feedback on large volumes of student programming assignments, but past approaches focus on the traditional view of program assessment—does the student submission "produce the correct output." Such a system has been in use at Virginia Tech for many years with success. Unfortunately, such tools often do little to address the issues raised here. Instead, 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 students are given a clear message (say, from a zero score) when submissions do not compile, do not produce output, or do not terminate. In addition, students are not encouraged or rewarded for performing testing on their own. In practice, students do less testing on their own, often relying solely on instructor-provided sample data and the automated grading system.

In order to make 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 prototype tool developed at Virginia Tech for this purpose. The Web-CAT Grader grades student code and student tests together, requiring both to be present on every submission [9]. It places the burden of demonstrating correctness on the student, and then uses an assessment formula that focuses on testing performance. The Web-CAT Grader assigns scores using three measures: a score of code correctness, a score of test completeness with respect to the code, and a score of test completeness and validity with respect to the problem.

First, 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 from the instructor or teaching assistant is used in this score.

Second, the *test completeness score with respect to the code* measures how thoroughly the student's tests cover the student's code. For Java code, the Web-CAT Grader uses Clover [7] to instrument the student code. Coverage data is collected as student tests are run. The instructor has the option of using method coverage, statement coverage, branch coverage, or some mathematical combination to derive a measure of how thoroughly the student's code has been exercised by the student's tests.

Third, the *test completeness and validity score with respect to the problem* measures how thoroughly the student's tests cover the behavior required in the assignment. Mechanically, this is similar to a more traditional program assessment—an instructor-provided reference test suite that captures all essential behaviors is run against the student program. However, if the student program passes all the student tests, and the student tests provide reasonable coverage of the student code, then the only reason any of the reference tests can fail is because either (a) the corresponding behavior is not implemented, and thus not tested for by the student, or (b) one or more of the student-provided tests are inconsistent with the behavior required in the assignment.

All three of these measures are taken on a 0%–100% scale, and then multiplied together to produce a single composite score. As a result, the score in each dimension becomes a "cap"—it is not possible for a student to do poorly in one dimension but do well overall. Also, a student cannot accept so-so scores across the board. Instead, near-perfect performance in at least two dimensions becomes the expected norm.

To support the rapid cycling between writing individual tests and adding small pieces of code that is characteristic of TDD, 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, their program correctness is only assessed by the tests they have written, so to find out more about errors in their own programs, a student must write the corresponding test cases. Currently, the Web-CAT Grader also applies Checkstyle and PMD, two industrial-quality static analysis tools, to assess how well the student has conformed to expected coding conventions, and all such feedback is produced in one seamless source code markup report viewable by the student on the web.

5. EXPERIENCES IN A JUNIOR COURSE

This approach has been piloted using an early version of Web-CAT in CS 3304: "Comparative Languages," a typical junior-level programming languages course at Virginia Tech. Students in the course normally write four program assignments, each requiring two to three weeks to complete. Basic instruction in TDD was provided to students, consisting of about one lecture hour of course time and several reading assignments outside of class.

In spring 2003, 59 students in the course used Web-CAT to submit all programming assignments. These students were given the same assignments used during the Spring 2001 offering of the course, where a conventional output-correctness-based automated grading system was used without TDD (students were still in-

Table 1: 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
Curator assessment	93.9%	96.4%	t(*df* = 71) = 1.36	1.99
Web-CAT assessment	**76.8%**	**94.0%**	t(*df* = 61) = **4.98**	2.00
Time from first submission until assignment due	2.2 days	4.2 days	t(*df* = 112) = **3.15**	1.98
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	**70.0**	**38.3**		

structed to test their own code before submission and given educational materials on basic testing practices). 59 students completed the course during spring 2001. Program submissions from both semesters were then available for detailed analysis. 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 due to manpower limitations.

Table 1 summarizes the results obtained when comparing the program submissions between the two groups. Because Web-CAT and the earlier grading system called the Curator 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 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 submitted as if they were produced by the student.

In Table 1, "Recorded grades" represents the average final assignment score recorded in the instructor's grade book. Half of each score came from the automated assessment and half from an independent review of the student's source code by a graduate teaching assistant. "TA assessment" reflects the average amount of credit received for the TA portion of the student's grade. "Curator assessment" reflects the average amount of credit given by the traditional automated grading approach, while the "Web-CAT assessment" is the amount of credit given by the new automated assessment prototype tool.

While the "Curator assessment" 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. The "Web-CAT assessment" differences are significant, however. This result is understandable, since students in 2003 were given explicit feedback about how thoroughly they were testing all aspects of the problem specification, and thus had an opportunity to maximize the completeness of their tests to the best of their ability.

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 grad-

ing/submission system before (the Curator). Students expressed a strong preference for Web-CAT over their past experiences. 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).

6. EXPERIENCES IN CS1

As a result of experiences with this approach at the junior level, it is now being integrated into Virginia Tech's core curriculum. The fall 2003 semester began with incoming freshmen in CS1 writing basic tests of their own code in the very first laboratory session during the first week of classes. CS1 is taught in Java using BlueJ. Students are taught using an aggressive objects-first pedagogy, and begin with a variation of Bergin's Karel J. Robot simulator [3] for initial assignments. Bergin's implementation allows students to write pure Java programs using a provided Karel class library, and also provides support for JUnit-style testing. With minimal introduction to testing concepts, students readily use BlueJ to interactively instantiate objects, and then interactively "record" sequences of actions—and assertions about expected outcomes—as test cases. Finally, the Web-CAT Grader supports BlueJ's assignment submission abilities, so a student can send an assignment to the grading system just using a menu entry in their IDE, with the results popping up in their web browser.

To date, the experience has been quite positive. Allowing unlimited submissions, with a web-viewable, color-highlighted feedback report available in less than a minute, encourages frequent use by students. Further, students readily grasp the up-front emphasis that the assessment strategy gives to testing, and their natural pursuit of higher scores reinforces the desired skills. The simplicity of the tools does make this accessible, even at the CS1 level, and with minimal class time devoted to teaching testing concepts. The natural benefits that students see, together with the assessment approach, drives their use of the technique.

7. CONCLUSION

Despite the best efforts of computer science educators, CS students often do not acquire the desired analytical thinking skills that they need to be successful until later than we would like, if at all. It is possible to infuse continual practice and development of comprehension, analysis, and hypothesis-testing skills across the programming assignments in a typical CS curriculum using TDD activities. Using automated grading and feedback generation to provide for frequent, quick-turnaround assessments of student performance helps to encourage and reinforce desired behaviors. Furthermore, students see real benefits from using this approach, an important factor for its continued use across multiple courses.

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 assess the outcomes of apply this technique in our introductory programming sequence to better characterize its impact.

8. 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.

9. REFERENCES

[1] Allen, E., Cartwright, R., and Stoler, B. DrJava: a lightweight pedagogic environment for Java. In *Proc. 33rd SIGCSE Technical Symp. Computer Science Education*, ACM, 2002, pp. 137-141.

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

[3] Bergin, J., Stehlik, M., Roberts, J., Pattis, R. Karel J. Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java. Unpublished manuscript available at: <http://csis.pace.edu/~bergin/KarelJava2ed/>

[4] Buck, D., and Stucki, D.J. 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, 2000, pp. 75-79.

[5] Buck, D., and Stucki, D.J. JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. In *Proc. 32nd SIGCSE Technical Symp. Computer Science Education*, ACM, 2001, pp. 16-20.

[6] Comer, J., and Roggio, R. Teaching a Java-based CS1 course in an academically-diverse environment. In *Proc. 33rd SIGCSE Technical Symp. Computer Science Education*, ACM, 2002, pp. 142-146.

[7] Cortex, Inc. Clover: a code coverage tool for Java. Web page accessed Mar. 21, 2003: <http://www.thecortex.net/clover/>

[8] Decker, R. and Hirshfield, S. The top 10 reasons why object-oriented programming can't be taught in CS 1. In *Proc. 25th Annual SIGCSE Symp. Computer Science Education*, ACM, 1994, pp. 51-55.

[9] Edwards, S.H. Rethinking computer science education from a test-first perspective. In *Addendum to the 2003 Proc. Conf. Object-oriented Programming, Systems, Languages, and Applications*, ACM, to appear.

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

[11] Kölling, M. BlueJ—The Interactive Java Environment. Web page, last accessed Mar. 21, 2003: <http://www.bluej.org/>

[12] Krause, K.L. Computer science in the Air Force Academy core curriculum. In *Proc.13th SIGCSE Technical Symp. Computer Science Education*, ACM, 1982, pp. 144-146.

[13] Schön, D. *The Reflecting Practitioner: How Professionals Think in Action*. London: Temple Smith, 1983.