Teaching Software Testing: Automatic Grading Meets Test-first Coding

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

A new approach to teaching software testing is proposed: students use test-driven development on programming assignments, and an automated grading tool assesses their testing performance and provides feedback. The basics of the approach, screenshots of the sytem, and a discussion of industrial tool use for grading Java programs are discussed.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*.

General Terms

Languages, Verification..

Keywords

Test-driven development, laboratory-based teaching, CS1, extreme programming, Java.

1. INTRODUCTION

Virginia Tech has been seeking to improve the coverage of software testing skills in our undergraduate program. Rather than introducing a new course, we are attempting to apply an active-learning approach to introducing testing concepts across the entire CS curriculum [6]. Testing techniques for object-oriented software are of particular interest, since our introductory sequence teaches objects-first using Java. The goal is to teach 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 additional lecture hours.

The resulting strategy is founded on two ideas: have students use test-driven development on their programming assignments from the beginning, and then use an automated grading tool to meaningfully assess their testing performance while also providing rapid, concrete feedback on how to improve. This strategy has been piloted to positive student reactions; an analysis of student programs revealed that students produced **45% fewer bugs** per thousand lines of code using this approach [4].

2. TEST-DRIVEN DEVELOPMENT

Unfortunately, in most undergraduate programs, students get little practical training in how to test their own code and often have poor skills (and even poorer expectations) in this area. In order to produce a cultural shift in the in the way our students acquire and apply testing skills, a new approach is needed. The core idea underlying this approach is that students should always practice test-first coding, also known as test-driven development (TDD), on their programming assignments from the beginning, across all of their core courses.

TDD has been popularized by extreme programming. In TDD [1], 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. TDD is attractive for educational use. 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 (if incomplete) 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. Most importantly, students begin to see these benefits for themselves after using TDD on just a few assignments.

3. AUTOMATED GRADING

The key to implementing TDD across the board is a powerful strategy for assessing student performance. The assessment approach should:

- 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.

Unfortunately, instructors and teaching assistants are already overburdened with work while teaching computer science courses and have little time to devote to additional assessment activities. As a result, an automated tool for grading student programs is desirable. Many educators have used automated systems to assess and provide rapid feedback on large volumes of student programming assignments [5, 8]. Such systems typically focus on compilation and execution of student programs against some form of instructor-provided test data. This approach ignores any testing the student has performed and fails to provide the both the assessment and the feedback necessary to properly facilitate TDD.

As a result, we have designed and implemented a general-purpose automated grading tool and incorporated it into Web-CAT, the Web-based Center for Automated Testing. Instead of automating an assessment approach that focuses on the *output* of a student's program, instead we must focus on what is most valuable: the student's testing performance. To provide a meaningful assessment of how correctly and thoroughly the tests conform to the problem, the Web-CAT Grader examines three facets of the student's submission. First, Web-CAT assesses the *validity* of the student's tests in terms of how correctly they reflect the problem. This can be done by running student tests against a (correct) reference implementation, and providing feedback on which tests are incorrect. Second, Web-CAT assesses the *completeness* of the student's tests. This can be done by measuring the code coverage achieved by the student's tests on their code, as well as by using a reference test suite intended to capture the full space of the problem. Feedback on which portions of the code were not properly covered is returned to the student. Third, the style and quality of the student's code is assessed using static analysis tools that point out specific problems.

Web-CAT is a web-based application implemented using Apple's WebObjects framework. It is designed to be language independent, but this poster focuses on grading object-oriented programs written in Java. For Java programs, students write JUnit-compatible test cases and submit them along with the other classes in their assignment. Web-CAT uses Clover [3] to instrument code for coverage analysis, and uses Checkstyle [2] and PMD [7] to perform static analysis of coding and commenting style and to spot potential coding issues. The reports produced by these tools are merged into one seamless source code markup viewable on the web by the student.

To support the rapid cycling between writing individual tests and adding small pieces of code, the Web-CAT Curator will allow unlimited submissions from students up until the assignment deadline. Students can get feedback any time, as often as they wish. However, their score is based in part on the tests they have written, and their program performance is only assessed by the tests they have written. As a result, to find out more about errors in their own programs, it will be necessary for the student to write the test cases. The feedback report will graphically highlight the portions of the student code that are not tested so that the student can see how to improve. Other coding or stylistic issues will also be graphically highlighted.

4. EXPERIENCE AND CONCLUSION

This technique has been piloted in a junior-level undergraduate class of 59 students using an earlier version of the Web-CAT Grader. Students preferred this approach over that used in prior classes, and tested their programs more thoroughly [4]. As a result, using TDD in class holds great promise for improving testing skills. Providing a system for rapid assessment of student work, including both the code and the tests they write, and ensuring concrete, useful, and timely feedback, is critical. In addition to assessing student performance, students can get real benefits from using the approach, and these benefits are important for students to internalize and use the approach being advocated.

5. ACKNOWLEDGMENTS

This work is supported in part by the Virginia Tech Institute for Distance and Distributed Learning and by the National Science Foundation under grant DUE-0127225. Any opinions, conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the NSF. The author acknowledges the contributions of the students who have implemented parts of the system: Anuj Shah, Amit Kulkarni, and Gaurav Bhandari.

6. REFERENCES

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

[2] Checkstyle home page. http://checkstyle.sourceforge.net/

[3] Clover: a code coverage tool for Java. http://www.thecortex.net/clover/

[4] Edwards, S.H. Using test-driven development in the classroom: providing students with automatic, concrete feedback on performance. In *Proc. Int'l Conf. Education and Information Systems: Technologies and Applications*, 2003, pp. 421-426.

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

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

[7] PMD home page. http://pmd.sourceforge.net/

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