Web-CAT: An Interactive Environment for Learning Software Testing Skills

Abstract

Defective, or "bug-riddled," software is a serious problem in the software industry, costing U.S. companies as much as $100 billion last year. Software testing is a critical tool for addressing this issue, but unfortunately, it is viewed as boring and uncreative by practitioners and receives little formal treatment in undergraduate programs. The objective of this project is to develop a prototype web-based "automated software testing center" where students can submit their own code for testing services and feedback. This center will answer a wide spectrum of student testing needs by providing a "sliding scale" interface, presenting an appropriate set of services based on a given student's situation, their current assignment, and the instructor's goals. The result will be an array of services with an easy-to-use web interface, allowing students to use the tool in many different classes, allowing instructors to incorporate tool-based testing in their own assignments, and supporting use by students and educators at multiple institutions. Such a tool will serve as an interactive learning environment where students can explore and learn about testing techniques while working on assignments for computer science classes. This environment will have a significant impact on future endeavors to integrate software testing education across the computer science curriculum. It will also provide a direct vehicle for incorporating current research on automated testing as well as practical testing techniques into classroom use.

The prototype web-based testing center will provide four principal services: test driver generation, test data generation, "built-in test" component wrapper generation, and test execution. In addition, the prototype will provide a natural platform for other automated testing services resulting from the principal investigator's research activities. The prototype will focus on testing C++ and Java classes, and will demonstrate the feasibility of providing similar services for other programming languages. To the extent possible, behavioral specifications written in the Java Modeling Language (JML) or in the Abstract State Machine Language (AsmL) will also be supported. This work will build on the success of past work with the Curator, the web-based automated program grading service previously developed with support from Microsoft Research.

1 Introduction

While software companies attempt to increase quality while reducing development time, the problem of software defects, or "bugs," continues to grow. 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 last year in lost productivity and repairs" [29]. 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 [16, 4, 2].

Unfortunately, testing is perceived as tedious, uncreative, boring work by practitioners, less than 15% of whom ever receive any formal training in the subject [35]. Despite the importance of software testing, testing skills are often elusive in undergraduate curricula [35, 3, 30, 33, 5]. 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" [30]. Because testing pervades all aspects of software development, however, there does not appear to be one "right" place in the undergraduate experience to place this kind of learning. Instead, software testing is a crosscutting or recurring area that must be visited multiple times with increasing levels of depth and maturity throughout the curriculum [3].

Virginia Tech's long-term plan for addressing this issue is to develop a series of on-line education modules covering the domain of software testing that will be integrated at separate levels across the undergraduate computer science curriculum. **Web-CAT: the Web-based Center for Automated Testing** is a critical tool in this plan. Web-CAT is an interactive environment for learning about and carrying out software testing tasks. Web-CAT will provide a unique active-learning experience for a wide spectrum of students by providing a "sliding scale" interface, presenting an appropriate set of services based on a given student's situation, their current assignment, and their instructor's learning goals. This project encompasses the development of a Web-CAT prototype that provides four principal services: test driver generation, test data generation, "built-in test" wrapper generation, and test execution. Web-CAT will have a powerful impact on Virginia Tech's ability to develop effective on-line testing materials that students can pursue at their own pace.

2 Goals and Objectives

The proposed work is based on three goals:

1. Develop a **web-based "automated software testing center"** where students can submit their own code for testing services and feedback.
2. **Evaluate** the classroom use of the testing center, using student grades, surveys of students and instructors, and by direct measurement of performance on authentic testing tasks through tracking defect rates (number of bugs introduced) in student program assignments.
3. **Disseminate** results by making Web-CAT available on the Internet, submitting to peer-reviewed educational resource repositories and journals, and presenting at conferences.

By integrating testing across the curriculum and providing student-level tools that make the necessary techniques more accessible while relieving some of the tedium, it will be possible to transform the expectations placed on student assignments. Eventually, testing will become "second nature," a social responsibility that students carry out from habit. At the same time, Web-CAT is a natural venue for integrating the investigator's research in automated testing technology into practical educational use [11, 10].

The remainder of this proposal discusses how the project goals will be achieved. Section 3 describes the on-line testing center. Section 4 presents the evaluation plan. Section 5 outlines the timeline and milestones, while Section 6 discusses the plan for dissemination of project results. Finally, Section 7 presents the budget justification (with full details in Appendix A) and Section 8 describes the qualifications of the principal investigators for pursuing the proposed activities.

3 Project Plan

For educating students, repeated (and successful) experiences testing real (student-level) software are more important than simply presenting testing concepts. Unfortunately, as students start out, they are neither prepared for nor capable of doing a comprehensive job of testing their own assignments. They cannot exhibit proficiency until they have developed more programming maturity and have had more training in testing. Tool support is one way to overcome this initial barrier.

3.1 An Automatic Software Testing Web Service for Students

To provide practical tool support for students who are learning how to test, a prototype web-based, student-oriented, automated software testing center will be developed. Ideally, one would like to package up the tools and methods resulting from current research into automated testing [10, 11, 7, 13, 15, 17, 22, 24, 25, 26, 27, 32, 34] and make them available as services for undergraduate students to use on basic program parts they have developed themselves.

Because "push button" testing is still in the realm of research, however, a practical testing facility must provide a range of services that vary in degree of automation and in "quality" of testing performed. If a student's component is written to meet a pre-existing behavioral specification, more automation is possible with more effective results. For other components, the student will have to make a choice between providing more information by hand or through a wizard-based interface and accepting less effective results (longer test suites with more invalid test cases, for example). The aim is to address this spectrum of needs using a "sliding scale" interface that will present an appropriate set of services based on the testing situation presented by the student. The result will be an array of services with an easy-to-use web interface so that students can use this tool from many different classes, so that instructors can choose to incorporate tool-based testing into their own assignments, and so that students and educators at multiple institutions can make use of the services provided. The Web-CAT prototype will provide four principal services: test driver generation, test data generation, "built-in test" (BIT) wrapper generation, and test execution. The prototype will focus on testing Java and C++ classes, and will demonstrate the feasibility of providing similar services for other programming languages.

3.1.1 Test Driver Generation

A *test driver*, or *test harness*, is a piece of code created solely for the purpose of testing a software part in isolation, and is typically needed during *unit testing*, often the most cheapest place to find bugs via testing. While test drivers are relatively simple in comparison to the entire application being developed, this "simplicity" has led to significant costs [35]. Because unit and subsystem test drivers are not intended to be included in the final product, they are often one-off programs produced in-house, are of relatively poor quality, provide little debugging support, and end up being thrown away after use. These facts led Rodney Wilson to observe: "Probably one of the biggest problems associated with modern-day software development environments is the major shortfall of available test harness technology" [35].

Web-CAT will provide an array of test driver generation services:

1. Test drivers containing hard-coded test cases:
   - Generation of a generic test driver skeleton for the user to download and customize.
   - Generation of a test driver skeleton for a user-defined class.
2. Test drivers that read and interpret test cases stored externally:
   - Generation of a interpreter-based test driver skeleton for the user to download and customize.
   - Fully automated generation of a test driver for a user-defined class.
   - Fully automated generation of a test driver for a collection of user-defined classes.

3.1.2 Test Data Generation

In addition to creating the scaffolding necessary to test individual software parts, one must also come up with good *test cases* that exercise the software in a way that is likely to find any defects. There are a number of strategies for generating black-box test data from a software component's behavioral description [2]. The generation approach chosen here is taken is adapted from black-box test adequacy criteria described by Zweben et al. [36]. This black box test adequacy work describes how one can construct a flowgraph from a class interface or behavioral specification. The principal investigator has researched automation strategies based on this approach [10]. This strategy is ideal for use in such an educational setting because it is easy to carry out by hand, is easy to automate, can be used with a minimal amount of information about the code being tested, and finds many bugs.

Web-CAT will provide the following test data generation services:

1. Generation of a simple test condition matrix for a user-defined class. A student can then come up with his or her own test case for each slot in the matrix.
2. Automatic generation of random data values for use in test cases.
3. Automatic generation of a set of test cases for a user-defined class, in a format suitable for reading by a testing center-produced test driver.

4. Automatic generation of a set of test cases for a user-defined class, including user-specified constraints on operation sequences and inter-operation dependencies.

3.1.3 BIT Wrapper Generation

The ability to perform testing against a user-provided or automatically generated *test oracle* (another piece of software that can judge the correctness of the behavior of the unit under test) may allow for automatic analysis and reporting of defect information, instead of simply returning the output to the student for a decision. To the extent possible, Web-CAT will provide support for oracles packaged as class wrappers that automatically detect behavioral errors [8, 11, 10]. Such a wrapper can provide many "built-in test" (BIT) capabilities for the component it wraps.

Automatic generation of BIT wrappers requires a rigorous description of the behavior expected from a component. Web-CAT will support behavioral descriptions written in JML [18]; NSF is currently supporting research on automatically detecting behavioral contract violations (NSF grant CCR-0113181), and this choice will allow Web-CAT to leverage that research rather than duplicate it. In addition, we will also explore supporting behavioral descriptions written using Microsoft's AsmL [1], which naturally supports executable specifications and run-time checking of conformance. We will explore supporting the following features within Web-CAT:

1. Generation of BIT wrapper skeletons, where students add their own code to check preconditions, postconditions, and invariants.
2. Fully automatic generation of BIT wrappers from JML behavioral descriptions.
3. Fully automatic generation of BIT wrappers from AsmL descriptions, where an AsmL-based reference implementation runs along side the component under test to spot faults.

3.1.4 Test Execution

Finally, Web-CAT will also provide the capability of automatically executing tests on a user-provided software component through the following services:

1. Automatic execution of user-provided test data on a user-provided class, with output returned to the student for inspection.
2. Automatic execution of automatically generated test data on a user-provided class, with output returned to the student for inspection.
3. Automatic testing of components with BIT wrappers, including a report of detected defects.

3.2 Leveraging Existing Work

Developing the on-line automated software testing center is a major undertaking. To achieve this goal within the limited resources of the project, significant reuse and retargeting of previous work will be required. Virginia Tech already has a history in web-based, automated systems for collecting, compiling, and executing student-written software in **the Curator System** [http://ei.cs.vt.edu/~eags/Curator.html, 23]. The Curator has evolved from 30 years of experience with automated program grading systems, and allows a course instructor to:

- Collect assignments (of any type, including programming assignments) over the Internet, and archive those assignments for later grading.
- Automatically grade collected programming assignments against an instructor-provided solution (typically using custom-generated input created fresh for each submission).

The Curator, which is available to other universities, is used in a number of lower-division courses at Virginia Tech, processing thousands of student submissions each year. The benefits gained from building on existing resources are critical to successful completion of this project. By basing the testing center prototype on the Curator's infrastructure, the project will be able to reuse a proven solution to a host of technical issues, including:

- Student authentication.
- Secure uploading, archiving, and tracking of student-submitted work.
- Protected compilation and execution of student-provided (and bug-filled) code.
- Detecting and reporting syntax as well as run-time errors in student code.
- Detecting and correcting student program crashes and infinite loops.
- Preventing misbehaving programs from damaging the server or other submissions.
- Automatic testing of programs against instructor-provided solutions.

4 Evaluation Plan

Web-CAT will be evaluated to assess its effectiveness. The plan for evaluation encompasses traditional techniques, such as feedback surveys, as well as a novel approach to measuring task-based performance on student-level software testing tasks. The evaluation plan has five parts:

1. **Student feedback opportunities**: Web-CAT pages will provide an on-line student feedback mechanism for students to comment on the various features of the prototype. This is an important part of the formative evaluation, particularly during initial use in courses. Providing a visible mechanism for student feedback empowers students to express their opinions, provides them with a better sense of control over the experience, and provides an immediate way for them to take action to address problems they uncover.
2. **Student surveys**: At the end of each course where Web-CAT is used, students will be given an on-line opinion survey eliciting their overall experience with the prototype, views on its effectiveness, and suggestions for improvement.
3. **Instructor surveys**: At the end of each course, the instructor(s) involved will also be given an on-line opinion survey to assess their overall experiences with using Web-CAT in class, views on its effectiveness, and suggestions for improvement.
4. **Student grades**: At the end of each course where the prototype is used, student grades on programming assignments throughout the semester will be analyzed relative to historical data for prior offerings of the course to look for any statistically significant difference in overall programming performance.
5. **Measurement and tracking of defect rates**: Courses using Web-CAT will also use the Curator to collect programming assignments, allowing the instructor to archive student programs for later analysis. Further, past use of the Curator at Virginia Tech has provided an excellent historical body of student programs (with student identification removed for anonymity) for the targeted courses, providing a unique basis for comparison. By developing comprehensive test suites and oracles for a representative group of class assignments, then running all student submissions against the corresponding test suite, it is possible to measure the number of program defects in each. By tracking and comparing defect rates, student performance on an authentic testing task—testing their own programs before submitting them for a grade—can be measured.

REFERENCES

[1] M. Barnett and W. Schulte The ABCs of specification: AsmL, behavior, and components. *Informatica*, to appear, 2002.

[2] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, NY, 1990.

[3] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* Wiley, New York, NY, 1995.

[4] B. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, December 1976.

[5] P. Bourque, R. Dupuis, A. Abran, J.W. Moore, L. Tripp, K. Shyne, B. Pflug, M. Maya, G. Tremblay. Guide to the Software Engineering Body of Knowledge—A Straw Man Version, Montréal, Université du Québec à Montréal, 1998. See http:/www.swebok.org.

[6] S.I. Brown. *The Art of Problem Posing*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1990.

[7] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME'93: Industrial-Strength Formal Methods*, J.C.P. Woodcock and P.G. Larsen, eds., Springer-Verlag, 1993, pp. 268–284.

[8] S. Edwards, G. Shakir, M. Sitaraman, B.W. Weide, and J. Hollingsworth. A framework for detecting interface violations in component-based software. In *Proceedings of the 5th International Conference on Software Reuse*, IEEE CS Press, Los Alamitos, CA , 1998, pp. 46–55.

[9] S.H. Edwards. Can quality graduate software engineering courses *really* be delivered asynchronously online? In *Proceedings of the 22nd International Conference on Software Engineering*, ACM Press, New York, NY, 2000, pp. 676–679.

[10] S.H. Edwards. Black-box testing using flowgraphs: An experimental assessment of effectiveness and automation potential. *Journal of Software Testing, Verification and Reliability*, 10(4): 249–262, December 2000.

[11] S.H. Edwards. A framework for practical, automated black-box testing of component-based software. *Journal of Software Testing, Verification and Reliability*, 11(2): 97–111, June 2001.

[12] S.H. Edwards. Toward providing quality graduate software engineering courses on-line. *Journal of Interactive Multimedia in Education*, July 2001, to appear.

[13] P. Frankl, and R. Doong. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[14] M.A. Freeman and J.M. Capper. Educational innovation: Hype, heresies and hopes. *ALN Magazine*, 3(2), December 1999.

[15] M.J. Harrold and G. Rothermel. Performing dataflow testing on classes. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, December 1994, pp. 154–163.

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

[17] D. Hoffman and P. Strooper. The testgraph methodology: automated testing of collection classes. *Journal Object-Oriented Programming*, 8(7):35–41, November/December 1995.

[18] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. Chapter 12 in Haim Kilov, Bernhard Rumpe, and Ian Simmonds (eds.), *Behavioral Specifications of Businesses and Systems*, Kluwer, 1999, pp. 175–188.

[19] D.S. Levin and M.G. Ben-Jacob. Using collaboration in support of distance learning. *ED 427 716*, 1998.

[20] T.J. Long, B.W. Weide, P. Bucci, D.S. Gibson, J. Holilngsworth, M. Sitaraman, S. Edwards. Providing intellectual focus to CS1/CS2. *SIGCSE Bulletin*, 30(1): 252–6, March 1998.

[21] T. Long, B. Weide, P. Bucci, and M. Sitaraman. Client view first: An exodus from implementation-biased teaching. *SIGCSE Bulletin*, 31(1): 136–40, March 1999.

[22] J. McDonald and P. Strooper. Translating Object-Z specifications to passive test oracles. In *Proceedings of the 2nd International Conference on Formal Engineering Methods*, IEEE Computer Society Press, Los Alamitos, CA, 1998, pp.165–74.

[23] W. McQuain. Curator: An electronic submission management environment. http://ei.cs.vt.edu/~eags/Curator.html.

[24] L. Murray, J. McDonald, and P. Strooper. Specification-based class testing with ClassBench. In *Proceedings of the 1998 Asia Pacific Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, 1998, pp.164–73.

[25] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proceedings of UML''99*, Springer-Verlag, Berlin, 1999, pp.416–29.

[26] T. O'Malley, D. Richardson, and L. Dillon. Efficient specification-based test oracles. In *Proceedings of the 2nd California Software Symposium*, April 1996.

[27] R. Pargas, M.J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verifications, and Reliability*, 9(4)263–282, September 1999.

[28] L.C. Ragan. Good teaching is good teaching: An emerging set of guiding principles for the design and development of distance education. *Cause/Effect*, 22(1), 1999.

[29] A. Ricadela. The state of software: Quality. InformationWeek, (838): 43, May 21, 2001.

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

[31] J.E. Stice. *Developing Critical Thinking and Problem-Solving Abilities*. Jossey-Bass Inc., San Francisco, CA, 1987.

[32] B. Tsai, S. Stobart, and N. Parrington. A method for automatic class testing (MACT) object-oriented programs using a state-based testing method. In *EuroSTAR'97*, 1997.

[33] A.B. Tucker, ed. Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force. ACM Press, New Your, NY, 1990. Available on-line at http://www.acm.org/education/curr91/. The draft 2001 revision is available on-line at http://www.acm.org/education/cc2001/.

[34] E.J. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[35] R.C. Wilson. *UNIX Test Tools and Benchmarks*. Prentice Hall, Upper Saddle River, NJ, 1995.

[36] S. Zweben, W. Heym, and J. Kimmich. Systematic testing of data abstractions based on software specifications. *Journal of Software Testing, Verification and Reliability*, 1(4):39–55, 1992.