TEACHING SOFTWARE TESTING FROM TWO VIEWPOINTS*

Neil B. Harrison
Department of Computer Science
Utah Valley University
800 West University Parkway
Orem, Utah 84058
801-863-7312
neil.harrison@uvu.edu

ABSTRACT

Testing software is actually two different activities, depending on whether one is the developer of the software or the tester. As software engineering students may end up in roles as either a developer or a tester, they must learn skills for both. In order to teach both skills, our software testing course has a two-part major project. In the first part, students must develop an application and test it thoroughly, creating a developer test plan and test cases. In the second part, they receive a different program from another student which they must test. They must write system test plan, execute the test cases, and write defect reports.

We have been using this project format in our software testing class for five years. We have found that it has been successful at helping students learn testing from two different perspectives. They learn how to apply different testing techniques, and how to select appropriate testing approaches. A few learn the bitter lesson of the danger of allowing the development schedule to squeeze out testing time.

We have found several keys to success. First, there must be multiple projects so the students can test a project they didn't develop. They must be equivalent in terms of development and testing effort. They should be not overly difficult, but should require a large number of test cases, preferably using more than one testing approach. Finally, students are not evaluated on the number of bugs they find in each others' programs, but on the quality of their testing.

---

1 INTRODUCTION

One of the important lessons that students must learn concerning software is that it isn't sufficient to just write code, but it is necessary to write high quality code. Of course, this means that they must learn about quality assurance and software testing. To this end, many programs, including our Software Engineering program, require a course in software testing and quality assurance. In our course, we teach students testing and quality assurance techniques. Their course of study includes white-box and black-box testing [1], testing non-functional as well as functional testing, test planning [2], defect reporting [3], and numerous other quality related topics.

Yet there is a challenge to teaching software testing that is rooted in the very nature of software testing itself. Testing is a two-headed beast with two very different personalities: one for developers, and a different one for testers. In other words, testing has a different nature if one is a developer than if one is a software tester. There are two particular areas where the differences are especially stark: The first area is that developers and testers can see different things: developers can see the code they write, while testers generally do not see the code. Visibility of the code (i.e., white box testing) has advantages for assessing coverage of the code by the test suite, but also biases which tests are written.

The second area of difference is the attitude toward testing [4]. For developers, the major goal is to finish writing the code, and testing is an activity to show that the code works. On the other hand, testers know that the code is buggy, and their job is to find where the bugs are. Because of this attitude, developers tend to test superficially, often picking "test cases" that are most likely to work. Quite simply, developers are inherently poor testers.

In addition to this, developers and testers have different duties with respect to defects: testers must know how to write defect reports, while developers must learn how to respond to them.

This means that there are quality assurance-related skills for both developers and testers, yet they are different. Students will have different needs from a course on software testing, depending on their professional role. Most students will spend most of their careers as developers, and will benefit most from learning developer-related testing techniques. However, the tester-related testing techniques are important, providing a foundation for quality assurance. Therefore, students should learn both skills; testing as a developer as well as a tester. Regardless of which role they fill in the workplace, both sets of skills will be useful: As developers, knowledge of testing will help them see their code from a different perspective, and help them test it more thoroughly. As testers, knowledge of the developer's testing mindset can help them understand the testing that developers are likely to have done – and not done.

2 BACKGROUND: THE SOFTWARE TESTING COURSE

As part of our software engineering curriculum, we have a course on software testing and quality assurance. The course covers testing methods and testing from the perspectives of both tester and developer. Students have assignments to help them learn

these methods, and they have a major project in which they can practice the testing methods and practices.

The course is upper division. While most of the testing techniques are basic enough that they can be taught in lower division courses, having it an upper division course provides at least two important benefits. First, a course on software engineering is a prerequisite to this course. This allows us to show how testing activities fit into various parts of the software development lifecycle. Among other things, we discuss developer-oriented testing and tester-oriented testing and where in the software lifecycle they are chiefly done. Second, upper division students have sufficient programming experience that they can take on a significant programming project that requires non-trivial testing effort.

The main text for the course is Patton [2], but much material comes from other sources. The main topics in the course are as follows; noted with additional references where used:

1. Tester-Focused Testing
   a. Black box testing techniques such as boundary value testing [1]
   b. Exploratory testing techniques [5, 6]
2. Developer-Focused Testing
   a. White box testing techniques such as structured testing [1, 7]
   b. Code inspections [8]
3. Quality Attribute Testing (both tester and developer) [9]
   a. Usability
   b. Performance
   c. Reliability
   d. Security
4. Test Management Issues
   a. Test planning and test plans [9]
   b. Criteria for stopping testing [1, 4]
   c. Defect reports [3]
   d. Agile testing [10]

The highlight of the course is a major project. This project allows the students to apply many of the concepts of the course, notably both developer-focused testing and tester-focused testing. The rest of this paper describes the project, and how it teaches students testing from both developer and tester perspective. Section 3 describes the project. Section 4 describes the lessons learned, and section 5 gives the conclusions.

3 THE TWO-HEADED TESTING PROJECT

In order to help students learn both perspectives of testing, the project requires them to test as both a developer and a tester, and then compare and contrast the two different testing perspectives. It works as follows: The students are required to develop a program and test it as a developer. They then receive a different program from another student which they test from the system tester's perspective. In this way they learn through experience how these two testing perspectives are similar, and how they differ. The mechanics of the project are as follows:

Each student receives a program specification which they must develop and test. Because students will later test each others' programs, it is desirable to have different programs. The minimum number of different programs is needed is two; three if there is an odd number of students in the class. We use either 4 or 5 different programs in a class. Using several programs gives us opportunities to discuss several different testing approaches later. Each of these programs has the following characteristics:

- It is sizeable, but not particularly difficult – no tricky algorithms are needed, for example.
- It has many things to test, requiring a large number of test cases.
- It includes a number of error conditions (e.g., invalid input) that must be tested.
- It has clear testing approaches, such as boundary value or state-based testing. (Students are graded in part by how appropriate their testing approaches are.)
- All projects are roughly the same size in terms of complexity, features, and required.
- The project can be done with a command line interface. However, some students choose to implement a GUI for their project.

For example, one program we often use is one to simulate moves in a chess game. The program shows a chess board with pieces, accepts move requests, and moves the appropriate piece. The user enters moves for both white and black alternatively (the computer makes no moves itself.) The program must make the moves correctly, and must inform the user of illegal moves. It must also detect piece captures. (To keep it straightforward, students are to ignore check, checkmate, castling, and en passant pawn captures. We also defer to simplicity by changing the rule for the end of the game – the game ends when a king is captured, rather than in checkmate.) It can be programmed with a GUI or with the board drawn with text characters. This program is not trivial, but not overly tricky. It has many test cases, and a few obvious testing approaches, such as boundary value testing.

A second example program processes payroll records. It generates monthly earnings statements for a set of employees, handling withholding for taxes, insurance, and savings plans. It requires involved numerical boundary condition testing. For example, there are different levels of tax withholding rates at different steps of income, therefore taxes withheld must be tested around each of these income steps.

A third example program simulates a monitor of vital signs (heart rate, blood pressure, and blood oxygen percentage) of a patient in a hospital. It requires boundary testing as well as state testing. For example, the system is in an alarm state if any of the vital signs is out of a "safe" range, and the severity of the alarm depends on how far out of range (e.g. low blood pressure can be a concern, or can be life-threatening), and which vital sign (e.g., heart rate is more critical than blood oxygen level).

3.1 The First Phase: Development and Developer Testing

The first phase of the project stresses testing from the developer's viewpoint. This requires that the students write and deliver a working system. They must also write a test plan and associated test cases. Their testing revolves around white box testing and unit testing. However, they are also able to perform requirements-based black box testing. (Later, when their program is tested by someone else, they learn that their knowledge of

the code biases their black-box testing, creating "blind spots." This is an important learning of the project.)

The students turn in their working project, their test plan, the test cases, and the results of executing their test cases. They are graded on the appropriateness of their test plan, the completeness of their test plan and test cases, and the correctness of their program. In order to ensure consistency, the correctness grade is based on the instructor's assessment, and not the results of the student testing (phase 2.)The first phase ends about three weeks before the end of the semester.

The students work individually on the projects; they do not work as groups. The reasons are that individual work requires them to demonstrate their own testing skills, and that they are not distracted by teamwork issues.

3.2 The Second Phase: Testing a Different Project

The second phase of the project consists of testing another student's project. Each student must deliver the working software to another student. In addition, the student must deliver operating instructions as well as any clarifications to the specification (the specs aren't perfect, and students need a few lines of clarification for each program.)

Students then write test plans and test cases, and execute the test cases against the software. As they find bugs (they always do), they write defect reports and deliver them to the original programmer. The programmer is not required to fix the bugs unless the bug prevents the tester from further testing. In practice, this is very rare; testers are almost always able to complete all or nearly all of their planned testing without any fixes from the developer.

Note that all aspects of the delivered package are fair game: students may write defect reports against functionality, usability, performance, documentation, etc. (All these areas are covered in the class.) Students often find defects in areas other than functionality; typically, if they find few functional defects, they find non-functional defects.

As testers, the students are graded on the quality of their test plans, including both completeness of testing and appropriateness of test approaches. Because effective communication of defects is critical to successful testing, they are also graded on the quality of their defect reports. They are not graded on how many bugs they find, since it depends on the quality of the software they receive. (Finding all the bugs is covered in a previous assignment.)

3.3 The Third Phase: Reflection

The final part of the assignment is a short paper describing the testing approaches used for both testing efforts, along with justification for them. The students are asked to compare and contrast the two testing approaches they used. The purpose of this phase is to encourage the students to critically analyze how they approached testing in both phases, and to think about how they could have done better. They finish the project by giving class presentations about the projects, both as developer-testers and system testers.

This gives the students a chance to learn about testing approaches used by others, and to exchange "war stories" about what went well and what didn't go well.

The following table gives a rough timeline of the project.

Task	Deliverables	Time
Develop & Test Project	- Working code - Source code - Test Plan & cases - Test Results - Any spec clarifications	5 Weeks (Other small assignments interspersed)
Test Other's Project	- Test Plan & cases - Test Results - Defect Reports	3 Weeks (No other work)
Final Report	- Test methods used and why - Comparison of the two parts	1 week or less (Reports given last or next to last day of class)

Table 1: Project Deliverables and Timeline

4 RESULTS

We have been using this project for five years. It has had a high degree of success among students. We have found that the students learn the intended skills, and the workload is appropriate. Importantly, the students appear to enjoy the project a lot.

4.1 Considerations and Lessons Learned

One might give all the students the same project to test. There are two reasons we use different projects: 1. Part of quality assurance is delivering software to someone else, and responding to questions and defect reports (they are not required to fix the defects found unless the defects are so severe that they block further testing.) The second reason is we already do that – it's a warm up for the big project.

There are some potential problems with having several different projects for the students. They have different testing approaches and different test cases, which may give students slightly different experiences. The upside to this is that the students hear reports about all the projects, and thus get exposure to testing of many different types of projects, not just the one they themselves test. The downside is that we have to be careful that the projects require roughly the same amount of work and have about the same number of test cases. To help ensure this, we collect statistics about how much time the students spend on each project. Of course, different students can spend very different amounts of time of the same project, so we have to look at aggregate efforts. Note also that the projects have been tuned over time to be about the same amount of work to develop as well as to test; i.e., the volume of tests is about equal.

In general, we use half as many different projects as there are students in the class; i.e., if there are 12 students, we have six different projects. This means that during the presentations at the end, each project is covered twice. This gives great breadth of presentations, but makes it likely that each project is covered adequately. (Among other things, in the event that a student performs poorly, there is a backup report on that project.) This ratio has worked well.

The use of GUIs tends to reduce the opportunity for input errors. For example, input of a month can be implemented in a GUI with a drop-down list, preventing any erroneous input. While this is useful information, and most programmers these days work in GUIs, it can skew the work required, and can introduce some inequality in the amount of testing required. In order to keep things equal, we may forbid the use of GUIs in this project.

One of the causes of poor performance in the project is poor planning – leaving the project until the last minute. This is a multi-week project: procrastinators tend to skimp on testing. This is a painful, but powerful lesson; however, it does reduce the amount of testing practice such students get. In order to help students, we will invite students to turn in their test plans for feedback early in each phase of the project – 2 weeks into the development phase, and 1 week into the testing phase. Of course, we are realists, and recognize that some students will leave everything to the last minute no matter what we do.

We have learned that some students are simply better programmers than others. Some programs are extremely clean, while others are fraught with bugs. This means that when testing others' programs, some students will find more bugs than others. Therefore, students are graded not on how many bugs they find in the other program, but how through and complete their testing was. And every student is able to practice writing defect reports – no program has yet been completely defect free!

5 SUMMARY

We have found that the double-headed testing project is a successful way for students to learn to apply testing techniques used by both developers and testers. It demonstrates the different techniques needed for developer and tester testing. It also gives experience in the major quality assurance activities, including test planning, writing test plans, testing, and writing defect reports. It helps the students appreciate the differences between developer-focused testing and tester-focused testing, and helps them understand the need for both types of testing.. In addition, it gives the students a chance to hear about different types of projects and how others have approached testing them. The chief challenges for the project are ensuring that the students have equivalent experience, both in the difficulty of developing and in the challenges of testing them. We have found that these challenges can be overcome with careful attention to the projects to ensure that they require about the same amount of development and testing effort.

REFERENCES

[1] Copeland, Lee. *A Practitioner's guide to Software Test Design*. s.l. : Artech House Publishers, 2004. 978-1580537919.

[2] Patton, Ron. *Software Testing*. s.l. : Sams Publishing, 2005. 978-0672327988.

[3] Bugzilla http://www.bugzilla.org, accessed 06/10/2010.

[4] Kaner, Cem et al. *Lessons Learned in Software Testing*. s.l. : Wiley, 2001. 978-0471081128.

[5] Whittaker, James A. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*, Addison-Wesley Professional, 2009, 978-0321636416.

[6] Itkonen, J. and K. Rautiainen, "Exploratory Testing: A Multiple Case Study", in *Proceedings of ISESE*, 2005, pp. 84-93.

[7] McCabe, T. "A Complexity Measure" *IEEE Transactions on Software Engineering*: 1976, pp. 308-320.

[8] Fagan, M. E. "Design and Code Inspections to Reduce Errors in Program Development" *IBM Systems Journal* 15(3), pp. 182-211.

[9] Desikan, S. and Ramesh, G. *Software Testing: Principles and Practices*, Pearson Education, 2006.

[10] Hendrickson, E. "Agile Testing, Nine Principles and Six Concrete Practices for Testing on Agile Teams" http://testobsessed.com/wordpress/wp-content/uploads/2008/08/AgileTestingOverview.pdf 2008, accessed 07/09/2010.