Teaching Software Testing: Experiences, Lessons Learned and the Path Forward

Panel Moderator:
W. Eric Wong (University of Texas at Dallas)
Panelists (alphabetical order):
Antonia Bertolino (Italian National Research Council)
Vidroha Debroy (University of Texas at Dallas)
Aditya Mathur (Purdue University)
Jeff Offutt (George Mason University)
Mladen Vouk (North Carolina State University)

Introduction

According to a study commissioned by the National Institute of Standards and Technology in 2002, software bugs cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the nation's gross domestic product (GDP). The same study also found that more than one-third of these costs, or an estimated $22.2 billion, could be eliminated by an improved testing infrastructure. These numbers would be significantly higher if the study were conducted today.

It is notable that software testing continues to be the primary approach used in practice to ensure the development of high quality software. It is also estimated that more than 60% of the development cost is spent on testing for many software systems. However, a large part of the problem is not so much the amount of testing that is performed on software, as it is "who" the software is tested by, and "how" these *testers* go about doing it. Many personnel responsible for software testing are software engineers with a very basic background in testing, mostly restricted to the application of a small set of testing tools. A simple knowledge of how to apply a few testing tools cannot hope to substitute for a strong foundation in software testing principles and methodologies. The fact of the matter is that a significant number of the people responsible for testing the software that we rely on may not be adequately prepared for the task at hand. If we were to trace this *deficiency* in software testing background back to its source, we would end up at the educational institutions responsible for teaching and training people to test software. Thus, if today's software testers are not sufficiently armed with the knowledge required to test software well, then it is most likely because they have not been adequately trained to do so. This is one of the root causes of the current state of software testing, and it is here that we need to begin to remedy the problem.

Each panelist will first give a 5-to-8-minute presentation of a position statement explaining their views on how software testing should be taught in undergraduate and graduate curricula, reporting their classroom experiences in teaching software testing to students in computer science or software engineering, discussing the lessons they have learned, and identifying the improvements which they would like to see. We will then open the floor to the audience to hear their concerns and comments on the current practice of teaching software testing.

Finally, we would like to emphasize a very unique aspect of this panel – having a student as a panelist to voice their opinions from a different perspective. This is critical as learning and teaching are two inseparable activities, and the issue can most effectively be approached from both sides.

978-1-4577-0348-5/11/$26.00 © CSEE&T 2011, Waikiki, Honolulu, HI, USA

Position Statement

W. Eric Wong and Vidroha Debroy
Department of Computer Science
The University of Texas at Dallas
{ewong,vxd024000}@utdallas.edu

All programs, whether large and complex or small and simplistic, must be tested, and they must be tested well. Both research and industry have accepted the importance of software testing, and reports have, and are always being, published about the savings that can be incurred when software is tested properly. Unfortunately, educational institutions are significantly lagging behind with respect to the testing knowledge that is imparted to students, and steps need to be taken to fix this problem immediately.

A large part of the problem is that the subject of software testing rarely appears in the mainstream computer science (CS) undergraduate curriculum, despite its well-established place in literature and its extensive use in industry. Many academic CS programs only briefly cover software testing, limiting the topic to software engineering (SE) courses that may not be mandatory for a CS degree. Furthermore, even if introductory courses cover some aspects of testing, the actual application of testing practices is only explored in-depth at the senior and graduate levels, through a dedicated course. However, offering a single (potentially elective) course is not a complete solution to the question of how to better educate our students in software testing, as testing principles and techniques require repeated practice before they become second nature. Also, teaching students about testing after they have already acquired a foundation in programming is ineffective. Testing should not be added to a student's skill set; it must be built into it from the very beginning.

We therefore propose that students should be exposed to software testing concepts and techniques through courses in the undergraduate curriculum itself, from beginning programming courses right up to the senior-level final project. We believe that students are more likely to incorporate software testing into their programming style if they experience the benefits of effective testing early in their career, and then continue to make use of what they learn in subsequent courses and projects. By extension, students with well-developed testing skills also become better software engineers as the act of testing also forces the integration, application, and enhancement of the other critical software development skills such as analysis, design and implementation.

To take a simple example, adding the requirement (even in introductory courses) that program submissions should be accompanied with test cases used, would serve to not just help students feel more confident about their submitted programs, but it would also allow instructors to gauge the testing abilities of their students. After some practice, students would start to understand the difference between a good test case and a bad one, and instructors could emphasize to them the fact that '*how much*' a program has been tested does not necessarily reflect '*how well*' it has been tested. Students could learn about simple black-box testing techniques early on and begin to test their programs systematically such that creating good test cases becomes habitual for them, even when it is no longer part of an assignment.

We have already begun to adopt this pedagogical model at the University of Texas at Dallas with a project funded by NSF.[1] A careful monitoring and evaluation plan has been implemented to assess the effectiveness in teaching software testing at the undergraduate level using the aforementioned model. Further details can be found at: http://paris.utdallas.edu/CCLI/.

[1] This work is supported in part by the NSF CCLI/TUES Type II awards DUE-1023071.

Position Statement

Aditya Mathur
Department of Computer Science
Purdue University, West Lafayette, IN 47907
apm@purdue.edu

A large number of Computer Science and Computer Engineering students go on to test computer software. However, courses in software testing are only slowly beginning to be integrated into CS curricula. In light of this fact we address below the following question: "*What should be the nature of an undergraduate or a graduate course in software testing*?"

1. *Rigor and formality*: Any course in software testing ought to be rigorous and formal. There is a belief among many in academia that courses in the general area of Software Engineering, including software testing, are "fluffy." This belief perhaps stems from the nature of books in the area of software testing. It is the responsibility of software testing educators to wipe out this belief.

2. *Algorithms and tools*: A course in software testing must introduce students to algorithms and tools for test generation, test selection, and test assessment, among other concepts and techniques. Students must be exposed to the strengths and limitations of the techniques and tools.

3. *Foundations versus Practicality*: Have we ever seen a course such as "Practical Compiler Design" or "Practical Networking" or "Practical Operating Systems"? We must divorce ourselves from the notion of "Practical Software Testing." Instead, we ought to have courses that educate students in the fundamentals of software testing, expose them to tools, and offer them opportunities to work on real-life projects to give them a flavor of what they are likely to find when they join the workforce.

4. *Coverage*: Software testing is a broad field and hence there is more material available than what can be covered in a one-semester course at any level. It is best if an undergraduate course focuses on the basics of software testing in the areas of terminology, test generation, test assessment, and test automation. An advanced course could then cover topics such as testing of concurrent programs, formal specifications based testing, and test generation from timed automata.

5. *Textbook*: The choice of a textbook is critical, especially in a course in software testing. While there exist several well-written texts in software testing, they rarely present the material in a rigorous and formal manner. For example, nearly every book in software testing discusses regression testing though almost none present the beautiful algorithms that are available and implemented in commercial tools for test selection for regression testing. We ought to write books on software testing that present the material in rigorous manner similar to how the material is presented in, for example, "The Art of Computer Programming by Donald Knuth, or "Compilers" by Aho, Sethi and Ullman?

Position Statement

Jeff Offutt
George Mason University

Finding failures and fixing faults during unit and integration testing is orders of magnitude cheaper than fixing them during system testing or after deployment. Major software companies now emphasize developer testing (unit) to improve their bottom lines and as a commercial advantage. For example, Google's software developers are required to solidly test each class, and held responsible for unit-level software faults that are found during system testing or deployment. Unfortunately, most undergraduate CS students learn very little about unit testing. My position is that it is past time that universities supported this trend! Most CS undergraduate programs teach a few lectures on testing in a senior software engineering overview class, with almost no emphasis on unit testing. **This is too late** to build lifelong habits of testing and quality software.

To build high quality software products, testing must be well integrated with development in a synergistic way. Developers must work with testers in respectful ways, rather than against testers. A first step toward respecting testers is to respect what they do: test. This can start at the low level, where unit testing is inherently integrated with programming. I suggest that **educators should integrate unit testing into introductory programming courses**. Specifically, I advocate the following steps in a first or second level programming class:

1. Give an assignment with one moderately complicated class.
2. After the students complete the assignment, the instructor provides high quality unit tests for the class. The tests should be automated in a tool such as JUnit.
3. Have the students run those tests on their original class and submit the results.
    - Then have them fix any bugs they find.
4. Teach the students how to design good unit tests and how to automate them. Reasonable test criteria are branch coverage (execute every branch in the class) or combinatorial coverage (all truth values of all predicates).
5. In the next programming assignment, require the students to work in pairs. Each person implements the program, then designs and automates tests for their partner's program.
    - An alternative would be to have the students use test-driven design.
6. Subsequent assignments should require automated tests to be submitted as a matter of course, just like comments and basic design documents.

This educational innovation has three goals. First we teach students **why** to unit test their classes, then we teach them **how** to unit test. Finally, we build a **habit** to unit test their software always. This habit will help them perform better in subsequent programming courses and make them better employees.

Position Statement

Mladen A. Vouk[2]
Computer Science Department, Box 8206
North Carolina State University, Raleigh, NC 27695

Testing is the process of executing or evaluating a system, or a system component, by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results or behaviors. It is an essential part of any software engineering process, and as such it needs to be taught to computer science students.

The technical part of testing is relatively straightforward and is covered in many ways in all software engineering books and many other places. A more subtle component of testing, and of other software engineering "writings" or "genres," is its relationship to the domain-specific communication capabilities of the students – they need to be fluent in the "language" of software engineering. To be successful in their careers, computer science graduates need, in addition to in-depth technical knowledge, the ability to communicate and collaborate with a variety of audiences – from very naïve and unfamiliar with computer science and software engineering, to very sophisticated.. To achieve this goal, it is important that students have instruction and practice in computational thinking and in communication skills throughout their curriculum.

How and when this training takes place, what is learned, and how it is executed is probably as important as the technical approach(es) used to effect testing. As part of an NSF project called "Incorporating Communication Outcomes into the Computer Science Curriculum" we have been creating a transformative approach to fully integrate communication instruction and activities throughout the curriculum in ways that enhance rather than replace their learning of technical content. As part of this panel, we[3] plan to discuss the research activities undertaken so far, and how the "testing genre" and its writing, reading, speaking and teaming components, including classical documents such as testing plans and interaction activities such as code inspections, are being incorporated into curricula at NC State University, Miami University, and several other schools.

[2] This work is supported in part by the NSF CPATH II awards CCF-0939081 and 0939122. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).
[3] The author is grateful to his CPATH co-PIs – Drs. Anderson, Burge, Carter and Gannod for their insights and advice.