Preparing the Next Testers: An Undergraduate Course in Quality Assurance

Peter A. Tucker

ptucker@whitworth.edu

Abstract

For most undergraduate institutions, a degree in computer science has a heavy emphasis on programming skills. Students learn programming fundamentals, data structures, and software development skills. Along the way, they also learn data management, operating system concepts, and computer architecture. Rarely, however, does a degree in computer science include the skills, techniques, and thought processes involved in software quality assurance. Indeed, anecdotal evidence suggests that quality assurance is a career choice considered by some academics as a “lesser” craft. We believe that a course in quality assurance is an important elective in a computer science education; that it will open opportunities for students and make them more attractive to employers looking for QA engineers. Further, we believe a course in quality assurance will help those students who take positions in software development or management better understand their roles in assuring quality in software.

We began offering a course in quality assurance in 2006. Our QA course has gone through four iterations. The most recent iteration focused on the five views of quality outlined by Garvin and then applied to software quality assurance by Kitchenham and Pfleeger. Garvin’s five views are: Manufacturing, Production, User, Value, and Transcendental. By focusing on each of these five views, students are able to take a more structured approach to thinking about quality assurance, and are better able to understand how quality assurance fits in with the difference aspects of software engineering.

This paper outlines our most recent iteration of the QA course. We discuss the different views and how they were presented to students, as well as the various activities students participated in during the semester. We discuss what went well and what kinds of things can be improved. It is our hope that a good discussion can take place between those in academia and industry practitioners to ensure that the exposure students get to QA skills and techniques is appropriate, and does indeed prepare graduates with degrees in computer science for various careers in software engineering.

Biography

Peter A. Tucker has served for the past ten years as a professor of computer science at Whitworth University. His regular course load includes quality assurance, database management, software engineering, and mobile application development. His research interests include quality assurance and data stream management. In addition, Dr. Tucker is a consultant at NextIT in Spokane, WA, where he is exploring QA automation techniques and implementing tests specific to some of their core technology. Dr. Tucker began his career at Microsoft, spending five years as a software design engineer in test, and then three more years as a software design engineer. Dr. Tucker received the Ph.D. degree from Oregon Health & Science University in 2005.

1 Introduction

College students in computer science typically interview for one of three kinds of internship or full-time positions: developer, quality assurance, or program management. Most CS students will take many classes that focus on programming techniques, so they often have some programming skill. They often will have done small-scale software projects as well, so they will have at least a simple understanding of software design and program management. However, students rarely have any experience in software testing, making them ill-prepared for interviews for positions in quality assurance. Those students that do make it through the interview for a QA position are not often prepared for what that job will entail, and have to learn even the most basic QA skills on the job. A course in quality assurance could help toward that end, preparing more students toward a career in quality assurance. Indeed, there exists a recognized need for a course in quality assurance as at least an elective for undergraduate students in computer science:

- Many companies look for testers with strong technical skills in software design and/or development.
- The Association for Computing Machinery (ACM) Computer Science Curricula (2008) mentions that QA was one topic that received attention in discussions with industry.

At Whitworth, we have offered a course in QA four times over the past eight years, designed to be accessible to computer science students in their second year of study. The motivation for the course came from a Whitworth alum who had been in quality assurance at Microsoft for 20 years. She requested that the course be offered to increase the number of employee candidates for QA positions. In their experience, few QA candidates have strong technical skills and strong testing aptitude. Talking with employees at others companies, the need for more strong candidates in QA seems to exist across industry. A course such as this, housed in a computer science department, would expose more students to the technical skills and the quality assurance skills required for a position as a software quality assurance engineer. Partially as a result of offering this class, Whitworth has produced a number of graduates who are or have held positions in quality assurance, including employees at Microsoft.

In the course's most recent iteration, offered in fall 2012, we modelled the course topics using Garvin's (1984) views on quality (and built upon by Kitchenham and Pfleeger [1996]): Manufacturing, Product, User, Value, and Transcendental. Our goal was to expose students to different points of view for approaching software quality. Students were asked to develop a test plan and implement tests, and to write bug reports as they find bugs.

The goal of this paper is to present how Garvin's views on quality played out in our QA course, and to start a dialog on what kinds of things industry would like to see in such a course. We are working toward a textbook on quality assurance, suitable for sophomores in computer science, so more universities can also offer a course in software quality assurance.

2 Overall Course Structure and Practice

Our QA course is designed to be very hands-on. As new concepts are introduced, students are asked to put those concepts into play. For example, as students learned about boundary test cases, we would present them with a specific function or interface and have them define the boundaries they saw. Over the course of the semester, students were to test a real software product of their own choosing (usually downloaded from a site such as sourceforge.com. Again, as new concepts were considered in class, students were to apply those concepts to the software they were testing.

2.1 Textbooks

The main textbook used in this course is Jorgensen's Software Testing: A Craftsman's Approach (2001). This text takes a practical approach to software quality assurance. It focuses largely on functionality testing, with minimal discussion on other "-ilities" defined in ISO 9126 (reliability, usability, efficiency, maintainability, and portability). As such, it was heavily used early in the semester as we learned fundamental QA techniques, and less as we explored QA from other angles, guided by Garvin's views. In addition, we recommended students use Kaner's Lessons Learned in Software Testing (2002). This text has many short lessons, and we ended nearly every class session with one or two lessons. Students really looked forward to this part of class; the lessons are short and practical, and are often applicable in any career direction in computer science.

2.2 Final Project

Students were also expected to form teams of 2-3 students for a project that would last most of the semester. They were to find and test a real software product. There were only two requirements: first, the software was not considered a release-quality version, and second, that it wasn't software that they had written. Students were encouraged to go to SourceForge.com and find software that was in alpha or beta stage. The motivation for this requirement is that students would be more likely to find bugs in software that was not ready to release. It was also important that the students have access to the developers' source code, so that they could perform code reviews, static code analysis, and code coverage. Details on these code-level tasks are given in Sections 4.1 and 4.2.

In the first iteration of this course, I found the software product that we tested as a whole class. I took this approach for two reasons: First, so that I could ensure that the application was testable, and second, so I could simulate a real world testing organization by assigning different parts of the software to different groups. The downside to this approach, though, was that I picked a software product that most of the class hated. Rather than trying to find software that would be more accepted, I have since allowed students find their own project. When students find something they are interested in testing, they demo the projects to me, so that I can determine testability. This approach has worked better since students are more motivated to work with a product that they actually like.

3 Initial Class Session Details – Fundamental Techniques

To start the semester, we worked on fundamental techniques and practices in quality assurance, so that students would have basic skills. We covered boundary testing, random testing and fuzz testing, equivalence class testing, and decision table testing. All of these topics relied heavily on material from Jorgensen. We then discussed test case creation, version control systems, and issue tracking. I wanted students to understand these basic tools so that we could apply them throughout the semester as we moved into each of Garvin's views.

Homework for these initial sessions asked students to apply these concepts in simple, contrived situations, largely based on exercises from Jorgensen. Students were given simple functions or interfaces, and asked to: come up with boundary tests, apply random value testing and fuzz testing, define equivalence classes, and define decision tables.

3.1 Quality Assurance Techniques

We spent the first three weeks of the semester covering a number of fundamental quality assurance techniques. As a running example, we used the classic `IsTriangle` function (Myers 2004) as a running example, with the added stipulation that the triangle sides had to be in the range [0,200]. The `IsTriangle` function takes three integer parameters representing the lengths of the three sides of a triangle. Based on those side lengths, the function should return the appropriate type of triangle: `Scalene`, `Isosceles`, `Equilateral`, or `NotATriangle`.

We started with boundary testing. That is, use test values that operate at the limits of the input. Following Jorgensen, we built up different levels of boundary tests. Students started with testing at or near the boundaries for a single parameter using legal values (e.g. `IsTriangle(0,10,20)`, `IsTriangle(10,20,199)`, `IsTriangle(10,20,200)`, and so on), then added tests to include illegal values for a single parameter at the boundaries (e.g. `IsTriangle(-1,10,20)`, `IsTriangle(10,20,201)`, and so on). We then added tests where all boundary cases were hit together (e.g. `IsTriangle(0,0,0)`), and finally special case boundary tests. The `IsTriangle` function illustrated this well for students, and they were able to pick up on this concept well. Particularly, `IsTriangle` was a very good illustrative example for the special case boundary tests. We wanted to hit boundaries for each kind of triangle. For example, we could test boundaries for isosceles triangles with `(5,5,8)`, `(5,6,10)`, and `(5,5,10)`.

We then moved into random value testing, by using randomly-generated values as testing values. Students understood quickly how to apply random value testing. We included discussion on the issue of test reproducibility when using random value generation as well. This discussion helped motivate the need for seeding a randomizing function, so that the same sequence of "random" numbers could be generated multiple times. Again, the `IsTriangle` function served us well here. Students saw the problem of blindly come up with three random values for a triangle, and instead were able to see how to generate combinations of random values that defined specific kinds of triangles. Finally, students were shown fuzz testing (Miller 1990) as an application of random value testing. A fuzzer takes a string input, and modifies it slightly, and randomly. For example, given the string "hello world", a fuzzer function might return any of the following strings: "hell▬ wo∟ld", "helo world", "hello wûrld", or "hhello?wArld". Students were shown a simple fuzzer function, and were expected to use it in their own testing.

Our next topic was equivalence classes, where groups of tests are classified as redundant, and we motivated this topic by discussing the number of tests required for `IsTriangle`. For example, the test cases `(5,5,5)`, `(10,10,10)`, and `(100,100,100)` are of the same class (equilateral triangles), and so executing all three would likely be redundant. Students were able to see that many of the test cases belonged to the same class. Students also could see the applicability of boundary testing random value testing here: to test boundaries within an equivalence class, and be able to come up with random values within a class, as useful testing strategies. Again using the case of equilateral triangles, `(0,0,0)` and `(200,200,200)` are boundary tests in the class, and then we could pick some random value `x` in `[1,199]` and test `IsTriangle(x,x,x)`.

3.2 Quality Assurance Tools and Practices

With some QA techniques in mind, students learned about test case development, using whichever programming language they were comfortable in. Students were given some general tips, including: test cases should one test one thing, and be self-contained; test cases should clean up after themselves; and QA engineers should look for opportunities to share code through libraries.

Students were exposed to some existing testing tools and frameworks. Most in-class software development at Whitworth uses Microsoft's tools, so students were shown the Unit Testing Framework in Visual Studio. Students learned about the `Assert` class, and how to check results against expected values through `Assert.Fail()`, `Assert.AreEqual()`, and `Assert.IsTrue()`. Students also saw how to develop specific test cases for library functions (we used vector and List classes as examples), and how to execute them in Visual Studio. From there, we worked on the UI automation tools also in Visual Studio. Students were able to record scripts through the tools and generate the code used for those scripts, then again execute them.

Finally, we discussed bug reports and issue databases. We used Bugzilla as an illustration. We discussed a bug's lifecycle (from bugzilla: http://www.bugzilla.org/docs/2.18/html/lifecycle.html), and the different parts of a bug report. Students were given various tips for writing bug reports, including discussion on the differences between priority and severity, the importance of being specific in reproduction steps, and how critical it is to make the title short and descriptive.

As students worked on testing their software product, they were expected to use these tools, or similar tools as appropriate. Test scripts were to be checked into a version control system so I could keep up with their work, and bugs were to be posted to some issue tracking system.

4 Class Sessions Applying Garvin's Views

With the fundamental techniques and tools behind us, we moved into Garvin's views on quality and how they applied to software testing. Thinking through all five views gives students more angles to think about as they consider the quality of the software they are testing. Students were expected to apply what they were learning directly to the software project they were testing. Homework, therefore, consisted of some contrived scenarios for testing, but also some more open-ended work as they considered how best to test their product.

4.1 Manufacturing View

According to Garvin, the manufacturing view focuses on the supply side of software development, particularly with an eye on engineering and manufacturing practices. Software should be developed using accepted practices and foundational techniques. In considering manufacturing practices, we focused on ISO/IEC 25010:2011, and the Capability Maturity Model. We also discussed other QA tools, including code coverage tools, source profiling tools, and test automation strategies. One difficulty we encountered in this portion of the class was in finding good, free tools for each project. Since we left the assignment open, and allowed students to find software to test that interested them, they all needed different tools. For example, each project was in a different language (C#, C++, and Python), so each team needed to find their own code coverage tool. This search was easier for some languages (e.g. Python) than others (e.g. C++). However, just the act of searching was a good lesson. Next time we offer the course, we will do a better job of providing students links to various tools that are available.

Focusing on ISO/IEC 25010:2011 gave us the motivation for developing test plans, as part of its General Requirements. Thus, we were able to get students thinking about test plans for their software early in the process. Initially we kept the test plan document fairly simple. We built it up throughout the semester as we learned new concepts. See Figure 1 for the initial test plan contents.

- Title
  - Software, including version and release numbers
  - Document Revision History
  - Table of Content
- Purpose of the document
  - Intended audience
  - Software overview
- Naming Conventions
- Test Outline
  - Testing approaches used by feature, functionality, process, …, as applicable
  - Boundary value analysis, equivalence classes, decision tables
- Test automation
- QA Tools to be used

Figure 1 – Initial Test Plan Contents

Students then were introduced to practices and tools for code and test case reviews. To think about code reviews, we used concepts from Spinellis' Code Reading, and considered specific code snippets to review. We considered specific "red flags" QA might look for in code reviews, including:

- Using conditionals to determine boundary cases
- Reviewing loops to try to cause forever loops, and also to look for test cases that make sure the loop executes zero times, one time, and many times
- Reviewing pointer and resource usages
- Reviewing storage, particularly temporary storage
- Sterilizing user input

With these ideas in mind, students were presented with simple code snippets, and asked to look for instances of those red flags.

Finally, we also considered static analysis tools, and how they can be used in QA. We considered the results from tools including Visual Studio's Code Analyze, Lint, and Resharper. As we worked through these tools, students were asked to consider how the results could be used to help in quality assurance. To motivate this discussion, we considered the results reported by Ruberto (2012), and how the quality of that mature product improved through the use of existing static analysis tools.

4.2 Product View

In the product view, quality is measurable, and determined by specific attributes of the product. Are there features of this product that make it unique from other, similar products? Is the software more efficient than existing products, in terms of speed or memory? Are there security features that make it a higher quality product? The product view focuses on measurable attributes, and so we can talk here about common software metrics including bug counts and bug rates, code coverage results, and mean time to failure.

We started this section with the practice of exploratory testing. Certainly, this discussion could have fallen with the user view just fine. In fact, we will probably move it to the user view next time. We put it here as a way to discover unique features of the software. Students were given tips and direction, then asked to perform exploratory testing on a particular dialog in GIMP, an open source image processing tool (http://www.gimp.org). At first, students were lost. They didn't know where to start. However, once they started down a path, they did a good job of keeping curious, of writing things down, and of learning new features of the product.

To discuss code coverage, we first took the approach of describing code as a directed graph, based on introductory discussion in Ammann and Offutt (2008). Students were given a small section of source code and asked to define a graph for that code. They were then asked to cover the graph with test cases. This visual approach really helped students understand what we mean by some of the different levels of code coverage. We then got into code coverage tools, and students were shown some examples. Finally, we considered the discussion from Roseberry (2013) regarding the myth that high percentage of code coverage means high quality. Students quickly picked up on why high coverage doesn't necessarily mean high quality software, but it was a good discussion that helped students think more carefully about what code coverage results really mean.

Finally, we used this area to introduce security testing, starting with the concept of taintedness. We took ideas from Whittaker on testing for security, and tried out many of his attacks. Obviously, students found this section fun. Security testing could easily be an entire semester of material; what we did here was simply intended to be an introduction.

Student homework during this section of the course consisted of applying the material to testing their own project. Students were to find tools for performing code coverage on their software, looking for places where they'd need to add more tests. In Section 5, we will see that this part of the assignment caused quite a lot of frustration. Student groups were to spend 30 minutes in exploratory testing, documenting all they did and the issues they ran into. They also were to try out Whittaker's security testing, and again reporting any issues. Finally, they were to update their test plan to include more discussion on how exploratory testing and security testing would be employed on their project.

4.3 User View

In the user view, quality "lies in the eye of the beholder." That is, if the product is of high quality, the user will tell you so. (Or, more likely, the user will tell you when the product is low quality.) For our class, it allowed us to focus on the "-ilities" outside of functionality: reliability, usability, maintainability, and portability, and ways to address testing in those areas. Students were introduced to stress testing, targeting heavy network activity, huge data files, high levels of concurrent activity, and activities that are long running. We also considered performance testing as both a way of testing reliability (especially with some standard benchmarks) and usability. Here students were asked to look at the PerfMon tool in Windows, to be able to watch CPU and memory activity during testing. For usability and portability, we discussed the notion of "eating your own dogfood" as well as usability labs. Additionally, and related to usability, we discussed user experience data and usability tools from Page et al. (2008).

As before, student homework was to discuss the different -ilities, and how they were to be used in testing their own project. Would they incorporate some kind of stress testing? If so, what would those tests look like? Would they use monitoring tools such as PerfMon to track performance, or develop other performance testing tools? And finally, would they set up dogfood sessions, or “bug bashes” to test the product’s usability. This discussion was also added to their test plans.

4.4 Value View

The value view allows for cost to be considered in the discussion of quality. Users may be willing to sacrifice a level of quality if it means paying less for the software, as long as the software continues to meet the needs of the user. In class, we discussed managing the tension between quality, time, scope, and resources. In terms of the tension of quality and time, we discussed bug triages, and the decision of which bugs to ship with. Regarding quality and scope, we focused on decisions related to which features to keep or drop in order to ship on time. For quality and resources, we talked about hiring good people, the costs of configuration labs, and interactions with clients. All put the value view into a very practical light for students. We also got into some discussion from Jones and Bonsignor (2011), and the cost of releasing low-quality software. Students were very surprised at the costs of low-quality software after the product is release.

At this point in the semester, students were finishing up their final work on testing their software. There wasn’t any specific homework assignment here, other than to continue testing and prepare for their final presentation.

4.5 Transcendental View

Finally, according to the transcendental view the quality of software is obvious to those who use it. Quality is not measured. From Pirsig, “But even though quality cannot be defined, *you know what Quality is*.” Certainly, this is the hardest view to teach, as it requires experience as much as anything else. We worked through discussion here from Emery (2012) on “testing quality in”. We focused on the need for quality to be a culture within the development organization, including management, designers, developers, and testers. Everyone on the project needs to expect quality in their work. Further, we discussed practices that can help within each group in the organization, including hiring the right people, employing good practices (e.g. test-driven development), and respecting the work of others.

5 Course Evaluation

Over the years, the quality assurance course has been well-received. The course is not required for our majors, but we always get a good number of students in the class. Students realize that having some exposure to quality assurance may help them in searching for jobs, whether in a QA role or as a developer or designer. At the very least, students get job interviews in part because they had a course in quality assurance on their resume. I motivate the subject to all computer science students by telling them that QA skills can be applied in every aspect of software development.

In many ways, the course has been a success. Our first goal was to try to encourage more students in computer science to consider quality assurance as part of a career path, and that has worked. Students who have taken this course have moved on to take internships and full-time jobs in QA, at companies including Microsoft, Adobe, NetApp, and NextIT. Some of those students have remained in QA roles, while others have moved into other roles in software development. Even those students who took development or design positions right after college have reported that taking the quality assurance course helped them succeed in their careers.

More generally, students report that taking the QA course has opened their eyes to the different careers in software development. When they enter the major, most students believe that development is the only career path. This course has shown them that there are other interesting directions they can take.

Students offer two main critiques of the class. First, that finding tools to help them test their project is not easy. Students pick their own software project to test, then they are asked to find tools for static analysis, code coverage, memory profiling, and so on. These tools are easily available for some projects (e.g. we were able to get code coverage in Python easily), but other projects are more difficult (e.g. we weren't able to get good code coverage results for some of our C++ projects). Students would like more help in finding these tools.

- Introduction
  - Title
  - Software, including version and release numbers
  - Document Revision History
  - Table of Content
  - Objective of Testing Effort
- Software product overview
  - Relevant related document list, such as requirements, design documents, other test plans, etc.
  - Relevant standards or legal requirements
- Overall software project organization and personnel/contact-info/responsibilities
  - Test organization and personnel/contact-info/responsibilities
  - Assumptions and dependencies
  - Testing priorities and focus
  - Scope and limitations of testing
- Test outline - a decomposition of the test approach by test type, feature, functionality, process, system, module, etc. as applicable
  - Outline of data input equivalence classes, boundary value analysis, decision tables
  - Test environment - hardware, operating systems, other required software, data configurations, interfaces to other systems
  - Code reviews and test reviews, including a schedule of what was done, and when
  - Code coverage – discuss tools used, and code coverage goals
  - Exploratory testing, including a schedule of what was done, and when
  - Security testing techniques and tools
- Non-functional Testing
  - Reliability
  - Usability
  - Maintainability
  - Portability
- Test automation and tools - justification and overview
  - Test tools to be used, including versions, patches, etc.
  - Test script/test code maintenance processes and version control
  - Problem tracking and resolution - tools and processes

Figure 2 - Final Test Plan Contents

The second critique is in the text books we've used. Most texts for QA seem to be written for students in at least their fourth year of undergraduate education, and assume students have a good foundation of software engineering. I want this course to be open to students who have completed their second semester of computer science, so that they can use these skills in their projects starting in their second year of their education.

6 Conclusion

As mentioned earlier, this course has become a regular offering at our school, and has been quite successful. Students appreciate learning the different skills and processes involved in quality assurance, and are glad to see new career paths for their skills.

One important goal we have heard from professionals in quality assurance is that students learn about writing test plans. As mentioned, students created an initial test plan (Figure 1), and then many of the homework assignments had students update their plan as we discussed new material. The final test plan was more complete, though certainly there could have been more added. The final test plan contents are shown in Figure 2.

In the next iteration, we will keep the focus on Garvin. It allows students to think about the whole task of software development. As mentioned before, we will move the exploratory testing piece to the user view discussion, and look for better tools that students can use. As we continue with this work, material will be compiled into a textbook.

Also, as mentioned earlier, we need to do a better job of helping students find tools they can use. For example, free code coverage tools are easier for students to find for projects written in Python than for projects in C++. More work needs to be done before the semester begins to find these tools so that students can use them.

It would also be worthwhile to devote more time to the “-ilities” of quality assurance. This section seemed a bit more ad hoc, and the requirement of adding appropriate testing for the “-ilities” was too vague. For example, we might devote a class session as a bug bash session, awarding prizes to students who find the most bugs. Another addition might be to spend more time with some of the efficiency measuring tools, including perfmon, as well as giving students time in class to use them.

7 Acknowledgements

The author would like to thank Microsoft Corporation, and particularly Cherie Ekholm, for their support for our QA course, as well as Whitworth University for its support as the course has developed. Finally, the author would like to thank the reviewers for their helpful comments on early drafts of the paper.

References

ACM/IEEE-CS Joint Interim Review Task Force. 2008. Computer Science Curriculum 2008: An Interim Revision of CS 2001, Report from the Interim Review Task Force. http://www.acm.org/education/curricula/ComputerScience2008.pdf

Ammann, Paul and Jeff Offutt, 2008. Introduction to Software Testing, Cambridge University Press.

Emery, Dale, 2012. Testing Quality In. *Pacific NW Software Quality Conference*, Portland, OR.

Garvin, David A., 1984. What Does “Product Quality” Really Mean? *Sloan Management Review*, (Fall):25–35.

ISO/IEC 25010:2011. Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models.

ISO/IEC 9126-1:2001. Software Engineering – Product Quality – Part 1: Quality Model

Jones, Capers and Oliver Bonsignor, 2011. The Economics of Software Quality. Addison-Wesley.

Jorgensen, Paul, 2001. Software Testing: A Craftman's Approach, CRC Press, Inc., Boca Raton, FL.

Kaner, C, James Bach, and Bret Pettichord, 2002. Lessons Learned in Software Testing, Wiley.

Kitchenham, Barbara and Shari Lawrence Pfleeger, 1996. Software Quality: The Elusive Target. *IEEE Software*, 13(1):12–21.

Miller, Barton P., Lars Fredriksen, and Bryan So, 1990. "An Empirical Study of the Reliability of UNIX Utilities", Communications of the ACM 33, 12.

Myers, Glenford J., Tom Badgett, Todd M. Thomas, and Corey Sandler, 2004. The Art of Software Testing, Wiley.

Page, Alan, Ken Johnston, and BJ Rollison, 2008. How We Test Software At Microsoft. Microsoft Press.

Pirsig, Robert M., 1974. Zen and the Art of Motorcycle Maintenance. Bantam.

Roseberry, Wayne, 2012. Code Coverage Isn’t Quality. It Isn’t Even Coverage. *Pacific NW Software Quality Conference*, Portland, OR.

Ruberto, John, 2012. Introducing Static Analysis in a Mature Codebase. *Pacific NW Software Quality Conference*, Portland, OR.

Spinellis, Diomidis, 2003. Code Reading: The Open Source Perspective. Addison-Wesley.

Whittaker, James A., 2003. How to Break Software Security. Addison-Wesley.