Experiences Teaching a Course in Programmer Testing

Andy Tinkham
Florida Institute of Technology
Department of Computer Sciences
andy@tinkham.org

Cem Kaner
Florida Institute of Technology
Department of Computer Sciences
kaner@kaner.com

Abstract

We teach a class on programmer-testing with a primary focus on test-driven development (TDD) as part of the software engineering curriculum at the Florida Institute of Technology. As of this writing, the course has been offered 3 times. Each session contained a mixture of undergraduate and graduate students. This paper discusses the evolution of the course, our lessons learned and plans for enhancing the course in the future.

1. Introduction

Software testing is a key component of the software engineering and computer science curricula (see [1, 2] for examples) and is an area of research and teaching strength at Florida Institute of Technology. Many of our graduates pursue testing careers; it is Florida Tech's intention to provide those students who choose this path with a strong background. Four years ago, as part of the design of our (now-accredited) B.Sc. program in software engineering, the faculty agreed that two testing courses should be required for graduation in software engineering. One would present black box techniques[1] and introduce testing principles to sophomores. The other would build on the testing *and* programming knowledge of seniors. This second course is the focus of this paper.

The three times Florida Tech has offered the course, Kaner was the instructor of record[2]. He co-teaches the course with a doctoral student, as part of his commitment to giving doctoral advisees a closely supervised teaching apprenticeship. In Spring 2003, Kaner and Pat McGee co-taught and jointly designed the course; Tinkham was a student in this class. Tinkham served as teaching assistant in Fall 2003 and took the leadership role in Fall 2004. We expect him to lead the course again in Fall 2005. Kaner assigns final grades to graduate students who take the course and independently regrades their work.

2. Related Work

We have seen no published reports of an undergraduate course focused on unit testing and/or TDD. However, there are several experience reports for teaching TDD in an introductory programming class [3-6], a more advanced class [7-15], and even a high school level computer science class [16]. In general, results were positive in these reports, with one exception: Müller and Hagner [12] found that a group using a variant of TDD took slightly longer to develop an application than a control group, with only small increases in reliability. They did however find that the TDD group had a significant increase in program understanding, as measured by the degree of reuse in the program.

3. Objectives

A black box tester analyzes a product from the outside, evaluating it from the viewpoint of its relationship with users (and their needs) and the hardware and software with which it interacts. The tester designs tests to check (typically, disprove) the appropriateness and adequacy of product behavior [17-20]. Skilled black box testing requires knowledge of (and skill with) a broad variety of techniques and the ability to appraise a situation to determine what processes, tools and techniques are most appropriate under the circumstances [17].

Unfortunately, testers whose only experience is black box can evolve a narrow attitude that doesn't necessarily work effectively with the rest of the development effort [21]. The programmer-testing course is Florida Tech's way of minimizing this risk among its graduates while increasing their sophistication.

[1] *See* http://www.testingeducation.org/BBST/
[2]

We worked through Angelo and Cross's Teaching Goals Inventory [33] to prioritize our goals for this course. After working through this inventory and extensive discussions with other faculty and prospective employers of our students, we developed the following set of core requiements for ths course:

- The course should broaden the perspective of students who will join black box test groups as a career and give them insight into ways they might collaborate with or rely on the programmers.
- The course should help average programmers become more thoughtful, more aware of what they're writing (and why) and more capable of writing code that works.
- The course should introduce students who will become project managers to the idea that many types of bugs are much more cheaply and efficiently detectable with automated programmer tests than black box tests, and that code testability and maintainability are well served by good suites of programmer tests.
- The course should introduce students who will become testing toolsmiths or test architects to the art of designing and creating reliable test tools.
- The course should give students some practice in soft skills, especially teamwork and presentations.
- The course should incent students to create high-quality artifacts that they can show off during job interviews.

4. Challenges and Tensions

Our general approach to course development is an evolutionary one:

- We give an easy course the first year. This compensates for the instructional blunders (confusing readings, lectures, examples, etc.) that make the course unintentionally harder. It also compensates for mis-enrollment. The new course has no reputation. Some students won't realize what they've gotten into until it is too late to drop the course. This is a particular problem for non-American students at Florida Tech. Dropping below 9 semester units can trigger Homeland Security interest in a visa student and so some students will doggedly stick with a course when it is too late to replace an inappropriate course with an alternative. Our preference as teachers is to work with the people in the room, helping them grow, rather than persevering with a plan more appropriate to a different group. Our view of the optimal feel of a new course is as a shared experiment in which a willingness to try new things is more important than getting everything right.
- The second iteration is better organized and more demanding than the first, but still transitionary. We should be able to prevent or control many of the problems that came up the first time, but expect there will be new problems and new mistakes will be made. Our standards are still floating, heavily influenced by student performance and feedback.
- By the third iteration, we expect to be able to anticipate and manage the typical-to-this-course student problems and excuses. We reuse and improve a fair bit of material from past years instead of inventing everything. And, at the start of the term, we present an explicit, detailed description of previous patterns of student difficulty and put students on notice of course standards while they still have time to drop and replace this course. We are unwilling to enforce harsh standards on students unless they were advised of them clearly enough and early enough that they could reasonably avoid them or otherwise appropriately change their circumstances or behavior. But given early, meaningful notice, the standards are the standards.

In this particular course, the most striking recurring problem is that some of our students (especially some graduate students) do not expect (and may even refuse to believe all the way through the final exam), that a *testing* course could require them to write clean, working, demonstrably solid code. Even though they have Java coursework in their background, these students are overwhelmed by the programming requirements.

Students who are not motivated programmers can find a multitude of excuses for not getting work done. If we ask them to use a new (to them) programming environment (such as Eclipse), a new tool (such as Ward Cunningham's Framework for Integration Testing (FIT) [3]) or a new language (such as Ruby), we can expect some students to complain that it cannot be installed or run on their platform, that its documentation is unusable, and therefore that it is not their fault that they aren't making progress. This problem will resolve itself over time as the course gains a no-nonsense reputation at Florida Tech, but until then, some of our decisions are defensive, damage controllers that will protect our time by preempting common excuses.

5. Course Implementation

The three instances of the course have shared some common elements:

- The course is a 3-credit one-semester (16 week) course. We offer it to upper-level undergraduates and graduate students who have completed coursework in black box software testing and Java programming.
- Classes are in a teaching lab with at least one computer per student.
- Students were encouraged, but not required, to work on assignments in pairs and co-submit a single piece of

[3] http://fit.c2.com/

work. Getting two college students in the same place at the same time was sometimes challenging (*see also* [9, 14, 15]), but most students resolved these challenges fairly easily.

- Students were required to submit their own tests and exams. They could not collaborate in any way on the midterm. The final exam was an open book take-home exam, and they could consult any source, including other people. Students were required to submit individual exam solutions, showing independent coding, and to acknowledge the people they consulted.
- Students were encouraged, but not required, to make in-class presentations of their work. We awarded bonus points for presentations, and in the second and third iterations, we ran collegial contests (students vote) for some groups of presentation (such as funniest application, tightest implementation, clearest implementation) and gave prizes.
- Several days involved student presentations or discussion / coaching associated with the current project rather than prepared lecture.
- There was a relatively simple in-class mid-term exam intended to test student understanding of basic concepts.
- Apart from polishing the wording, we used the same final exam each year, a version of which is available on Kaner's blog[4]. Students were to write a test harness in Ruby, automating function equivalence testing of OpenOffice Calc against Microsoft Excel. Programming had to be test-driven and students had to submit multiple iterations of their tests and code. We provided students with a Ruby library that Tinkham created to make Calc's COM interface more closely resemble Excel's. In a given test, the harness generated a random number, provided it to a selected worksheet function (or combination of functions) in Excel and to the equivalent in OpenOffice (together forming a function pair) and then compared results within a tolerance which the student specified and checked against as part of the assignment. Students tested each function pair 100 times. Students tested individual functions, pre-built complex (multi-function) formulas, and randomly combined complex formulas in this way. The tool was to provide a summary of each set of 100 results. The exam allowed up to 20 bonus points (of 120 possible points) for a thoughtful suggestion of a method for avoiding or dealing with nested invalid inputs that would block evaluation of a formula (for example, tan(log(cos 90)) is undefined, because cos 90 is 0 which is an undefined input for the log function).

[4] http://blackbox.cs.fit.edu/blog/kaner/archives/000008.html

5.1. Spring 2003

Seven students took the first course. All had been successful in Testing 1 (which many students find difficult) but their programming skills and confidence ranged from first-rate to minimal.

We started with a brief look at differences between black box and programmer-testing approaches—programmer tests are typically simpler, confirmatory in intent (designed to verify expected functioning rather than hunt for bugs), narrower in focus, and more tied to the details of the code than to a usage scenario. These tests are rarely very powerful, but a good collection of them provides a unit-level regression test suite that the programmer can run every time she changes the code. The tests serve as *change detectors*, raising an alert that a tested assumption was invalidated or violated by added code. An extensive unit test library provides powerful support for refactoring [22] and later code maintenance (bug fixes or additions to existing functionality). It also provides examples—unit tests—that show how the units work in detail. This is an important, concrete communication with future maintainers of the code. Test-driven development also provides a structure for working from examples, rather than from an abstraction. Many people operate from a concrete, example-based learning and thinking style [23]; the TDD approach works with that style instead of fighting it.

We expected people to quickly catch on to the test-driven style from Beck's introduction and to be enthusiastic about it. Based especially on coaching from Sam Guckenheimer, Alan Jorgenson and Brian Marick, we had a long list of traditional testing topics we planned to explore in a new way while working in this style.

To our surprise, the course bogged down quickly. Students either didn't understand JUnit, didn't understand the test-driven approach or didn't appreciate its benefits.

In retrospect, part of the problem was that our examples were too simple. We started out with basic sort routines that students already understood. This was intended to keep the programming aspects of the first part of the course simple, but this approach frustrated many students:

- The weakest programmers found even these routines challenging (or, at least, they were unsuccessful in writing their own versions of them), but algorithms and sample code for these routines were readily available from Google. Students were even encouraged to consult these solutions. Given access to the solution, it felt artificial to some students to recreate the solution in baby steps.
- Stronger programmers accepted the tasks but didn't yet see much value in solving a known problem in a seemingly slower way.

We didn't understand that this was a problem at the time, and so we kept the examples simple while introducing new techniques, up to and including FIT.

Brian Marick gave a guest lecture in which he re-introduced the class to TDD and demonsted more complex examples in Ruby. His demonstration also introduced students to testing at the API (application programmer interface).

In subsequent classes, we used Ruby to drive programs, then to create simple tests of programs, leading up to the final exam. The student presentations of their Ruby code and tests looked pretty good.

The final exam went less well. Several students wrote the test harness in a non-test-driven way. Every student appeared to have misunderstood the intent of the task as "test OpenOffice against Excel" instead of "write a test tool in a test-driven way and use a test of OpenOffice against Excel to illustrate your ability to do this type of work." The unit tests should test the test harness, and the harness should test the target program. But rather than using Test::Unit (the Ruby version of JUnit) to test the code they were writing, students used Test::Unit to drive their harness' testing of OpenOffice. Some argued that correct results from application of the test harness to OpenOffice demonstrated that it was working, and so further unit testing was unnecessary. Neither instructor had anticipated this problem

5.2. Fall 2003

Five students enrolled in the course: two undergraduates and three graduate students. Another graduate student audited the course. Programming skills again ranged across the spectrum.

We again introduced test-driven development with Beck [24]. We required the new edition of Paul Jorgensen's classic [25], expecting to cover several traditional issues in glass box testing. And we required students to program in Eclipse, for which there were plenty of good online resources.

This time, we wanted to spend most of the course time on one or two complex examples. We introduced the basic objectives of glass box testing, then introduced test-driven development with a simple example, but moved quickly to an introduction to two testing tools, Jenny[5] (a test tool to support combinatorial testing, written in C) and Multi[6] (a test tool to support testing of code using logical expressions, written in Java). The class split into two groups, one looking at Jenny, one at Multi. Their task was to learn the code, writing unit tests to encapsulate their learning as they went. After they had learned the code, we planned to have them extend the functionality using TDD, probably by improving the tools' user interfaces.

The students who worked on Jenny were good programmers, but they were unable to gain understanding of Jenny's internal structure in the time available. The students working on Multi got stuck on the functionality it provided. We spent what seemed like endless hours of class time on what felt like the same ground, how to generate an adequate set of tests for a logical expression. Neither group made significant progress. Eventually, Kaner cancelled the projects. The midterm exam applied test-driven techniques in Java to a simple program. We worked on FIT where we did some small assignments, and then moved on to Ruby, where we used Ruby to drive programs through the COM interface, and used Test::Unit to support test-first Ruby programming. We had some good student presentations, and proceeded to the final exam.

Students did their own work on the final exam (we have electronic copies of all of the exams—there was no hint of similarity between this term's solutions and the previous term's) and average performance was better than in Spring 2003. This is a subjective appraisal, not a report of average grades—we marked these exams a little more harshly than the previous term's. We were more explicit about how we expected this project to be done. We made it clear that we expected test-driven development of the test harness, and that the test harness, as a well-tested standalone program, would test the spreadsheets. Some of the exams did this. Other students still failed to use a test-driven approach, slapping together a largely untested program that could drive OpenOffice Calc and Excel and using Test::Unit to drive the testing of the spreadsheets, rather than the testing the test harness. In a long post-exam interview with the author of the best exam of this type, the student insisted that TDD of a test tool was unnecessary because the results of testing the applications would validate or invalidate the tool.

In retrospect, we like the idea of giving students a test-driven maintenance project. Despite the problems, some of the students learned a fair bit from beating their heads against strangers' code for the first time. We don't expect to use Jenny again, but we would consider using Multi. Next time, however, we'll schedule an open book exam early in the project that requires students to describe Multi's internal structure and some inflexible milestones for adding some groups of unit tests to the code base to encourage students gaining understanding of the program.

Other retrospective decisions: We

- resolved to be explicit to the point of tediousness that this was a course on test-driven development and that if students did not demonstrate competence in test-driven development when given the chance, they would flunk the course,

[5] http://burtleburtle.net/bob/math/jenny.html
[6] http://www.testing.com/tools/multi/

- would adopt one of the recently published books on test-driven development that had other examples and included more discussion of test design,
- would introduce test-driven development with some more complex examples.
- would add a book on Eclipse to the required reading list to deal with students who protested that they couldn't write code because they couldn't understand Eclipse,
- would use a book on glass box testing that was more closely aligned with the test-driven approach.

The Fall 2003 course wasn't what we had hoped it would be, but we felt that we had learned a lot from it, that we had much better insight into strengths and weaknesses of Kaner's teaching and assessment style as applied to this course and into the problems students were having. Based on our experiences in-class, on work submitted, and on other information we gathered, we concluded that some students' issues were more motivational than cognitive and that some specific problems raised by some students during the term were excuses presented to obscure a shortage of effort.

5.3. Fall 2004

Of the 12 students who completed the course in Fall 2004, nine were undergraduates. As in past years, individuals' programming skills ranged from strong to weak.

Most of the Fall 2003 classroom time had been spent on discussion and presentation rather than planned lecture. This time, we forced more structure on the course in several ways. We shifted back to a largely lecture-based style[7], we focused the course more tightly on TDD, agile practices, and testing an application through an API, dropping coverage of some classic unit testing techniques, and we described our (higher) expectations to the students. We did this by describing both the projects they would do and the expectations we had for them. We also referred back to problems students had in previous courses, identifying some types of common mistakes (such as submitting work that had not been demonstrably developed in a test-driven manner) as unacceptable and likely to lead to failure. For the final exam, we distributed a grading chart in class and used it to explain how we would grade the submitted work.

The course texts were Astels' *Test-Driven Development: A Practical Guide* [26], Hunt & Thomas's *Pragmatic Unit Testing in Java with JUnit* [27], and Holzner's *Eclipse* [28]. We also recommended Thomas & Hunt's *Programming Ruby* [23] when the second edition became available partway through the semester. We assigned readings from Astels and Hunt & Thomas.

Astels worked well as the main text book for the semester. This book covers basic topics of TDD for the first half of the book, while the second half is a full example building a program for tracking movie reviews. One project (discussed below) was designed to be similar to this example, and it worked well. We'll use this book again when we teach the course in Fall 2005.

Hunt & Thomas' JUnit covered the basics of JUnit, but Kaner considered the book's approach to test design too shallow and too formulaic. In 2005, we'll use Rainsberger [29] instead.

Holzner [28] served its purpose—students figured out how to use Eclipse without having to come to us for technical or (much) conceptual support. Unfortunately, Holzner predominantly covers Eclipse 2 rather than Eclipse 3. In 2005, we'll use D'Anjou et al. [30], which supports Eclipse 3.

We gave in-class assignments on refactoring, ideas for unit tests, and user stories. These helped students develop their understanding of these basic concepts; we'll use more in-class activities in 2005.

We also assigned four take-home assignments, covering refactoring, user stories, and using Ruby to drive COM interfaces. These generally consisted of short answer type questions such as "Identify the refactorings that should be applied to the following piece of code:" or involved writing short programs. For the Ruby homework, students had to write two Ruby scripts. One had to launch Microsoft Word, enter a paragraph of text with known spelling errors, then write out the spelling errors and corrections. The other had to control Internet Explorer, cause it to go to a website of a calculator of the student's choice (where calculator was loosely defined as a page which took input values of some sort, processed them, and then returned some results), enter data and then echo the results from the calculator. These were designed to prepare the students for the final exam.

We originally planned for two projects, one focused on creating an interesting program from scratch using TDD, the other focused on a maintenance operation (perhaps another crack at Multi). We designed the first project to be similar enough to Astels' movie review tracking program that students could use his example as a guide, while different enough to make the students apply the concepts themselves. We saw this as scaffolding that was important for weaker programmers. Students were to build an application for tracking a collection of something—the class decided on collectible game cards (Magic the Gathering™ from Wizards of the Coast). The students were largely already familiar with this game and had cards in their personal collections. We provided cards to students who lacked them, along with a student presentation on the basics of the game and hosted a

[7] Materials from the third offering will be available at http://www.testingeducation.org/pt

weekend afternoon of game play to familiarize the rest of the class. The students had about a month to implement 10 user stories in Java, using Eclipse, JUnit, and a Subversion source control server (five of the six pairs used the server).

While we were teaching the course, four hurricanes wreaked havoc on Central Florida, affecting classes (and many other things) at Florida Tech. As the hurricanes and their aftermath progressed, we repeatedly checked with students on their overall workload and adjusted expectations and schedules. Ultimately, we canceled the second project, extended time for the first project and, because the chaos had unfairly disadvantaged some students, offered a make-up project. In the make-up, we took the best example of student code from project 1 (with permission of the students), removed the original students' names, and added three more user stories. The students who did the make-up were now doing maintenance on someone else's code. We'll probably do this again, perhaps using it as the second project.

The mid-term exam had 8 short-answer questions covering the concepts of TDD. The average grade was 82% (5 of 12 students earned A's--above 90%. The lowest grade was a D, 67%). This indicated a reasonable class-wide understanding of the concepts.

The final exam called for the same test tool as in prior iterations, driving comparisons of OpenOffice Calc against Excel. We gave students a grading chart in advance, and gave an extensive lecture on ways students had lost points on this exam in previous classes. We gave students almost 3 weeks to complete the exam and we set aside the last day of classes (a week before the exam was due) for students to bring their exam progress to class and compare notes with other students. Students were not risking anything by collaborating because they knew that we don't grade on a curve—if everyone does well, everyone gets A's. The results: 5 A's, 1 C, 1 D, and 5 F's. The "A" papers showed a good grasp and good application of TDD practices and we are confident that none of the passing papers relied inappropriately on other student work. In contrast, the failing students made the same mistakes as in prior years (despite warnings in classes that most or all of them had attended). They wrote code without writing tests for the code. They didn't give us examples of code iterations, they didn't show refactoring, they didn't answer some sections of the exam at all, and despite explicit requirements stated on the exam and in lecture, they didn't separate the testing of the test harness from testing of the spreadsheets (for which they were supposed to use the harness). The weakness of this work was partially the result of procrastination. We warned students that this task was larger than they might expect and urged them to start early. But from the time course of drafts submitted to the class Subversion server, and nonexistent progress as of the last day of classes, we know that some groups started very late. The last two times we taught the course, we chose to grade final exams more gently. This type of information goes on the grapevine in a small school and may have incorrectly reassured some students that they could ignore our repeated descriptions of how we would grade. Next year, the grapevine will carry a different story.

Despite the high failure rate, performance was better across the board than the first two iterations, all students gained knowledge and skills from the course, and several students gained significantly. The third iteration was a success.

6. Lessons Learned & Plans for Improvement

Over the three iterations of the course, we've learned a few lessons:

- Test-driven development is counterintuitve for many students, especially graduate students who have become established in a different point of view. This makes the material much harder to teach and learn because students have to unlearn or reinterpret prior school and work experience. As with heavyweight processes taught in some software engineering courses, when students are required to apply processes that are more tedious and complex than valuable in the context of the problem they are trying to solve, some will learn contempt for the process. In this course, students need concrete examples that are difficult enough to show the value of the test-driven development.
- Test-driven development is probably not the right approach for all programmers, or all programming students. People differ strongly in cognitive styles, learning styles and thinking styles. [31, 32] Some people will more readily proceed from specific examples to general designs, while others will more naturally develop abstractions that guide their implementations. This doesn't mean that a person who primarily operates with one style *cannot* learn or operate with another, but it does suggest that some students will be predisposed to be turned off by the course, and that to be effective teachers, we have to develop some strategies for motivating them. We see this as our most significant challenge for 2005.
- Not all computer science and software engineering students can program or want to program. This is not unique to Florida Tech. We have seen it discussed by faculty from a reputationally wide range of universities at several academic meetings focused on the teaching of computer science, software engineering, and software testing. These students are not idiots—we think that surviving a computing program to a senior or graduate student level when you can't get high marks from your code must take a lot of compensatory

intelligence and work. But the students face problems when they join a class whose intent is to get them to integrate their existing knowledge of programming with ideas from other fields. Students in this situation need support, such as well-written supplementary readings, in-class activities that facilitate coaching of work in progress, and pairing with students whose skill sets are different from theirs. We think they also need to face a firm expectation that they *will* learn to use the course tools, they *will* do assignments on time and in good order, they *will* demonstrate their *own* programming skill, and that their success in this is primarily their responsibility and not ours.

- We haven't seen this mentioned before so we'll note that JUnit provides an experimental foundation (especially when combined with Eclipse), especially for weak programmers. If students want to understand how a command works or how a few commands work together, these tools facilitate an organized and efficient trial-and-error study. Some of our students seemed to learn well this way.
- Using TDD to develop a new project is different from maintaining or enhancing an existing project. We haven't yet successfully incorporated test-driven maintenance into the course, but we will.
- In the second and third iterations, we had planned to use an assigned project to introduce students to the idea of test-first development or maintenance of a test tool, but the second iteration's assignment failed and the third iteration's was blown away. We are fascinated that this is such a hard concept and wonder whether this is why so many test tools on the market are so full of bugs. In future iterations, whether by project or in-class activity, we will make sure that students work with a layered architecture (independently test a test tool that will then test a product under test) before taking an exam that also requires them to do this. This is an essential lesson for students who will become toolsmiths.
- It's probably time to change the final exam, but we plan to change details while keeping the same approach and leaving the same technical traps for students to fall into or overcome.
- Well-designed in-class activities and homework support learning and give fast feedback to the student and the instructor. They help students develop skills in small steps, and gradually apply them to more complex problems. In his black box testing course[8], Kaner now videotapes lectures in advance, students watch the lectures before coming to class, and all class time is spent on coached activities. It takes enormous work to build such a course. We will evolve this course in that direction, perhaps achieving the full shift over three more iterations.
- JUnit, Eclipse and Subversion all helped students do complex tasks. Next time, we'll add build management with Cruise Control or Ant.
- Ward Cunningham and Rick Mulgridge have a book on FIT that is becoming available this summer. Along with supporting acceptance testing, FIT supports test-driven, glass box integration testing. This is important knowledge for this course. We expect to use this book and also include FitLibrary and FolderRunner from Mugridge[9] and StepFixture from Marick[10] in future iterations.
- We want to work on our students' sophistication as test designers. They come into this course with a testing course, and often test-related work experience, but in the course they apply relatively little of what they know. The assertion that it is possible that one could "test everything that could possibly go wrong" is patently absurd. Instead, we need to frankly face the question, *What test design strategies will help us create the most effective tests, for what purposes, in a reasonable time frame?* The answer is very different for programmer testing than for system testing, but as with system testing [17], we expect many different good answers that depend on the specific development context. We and our students will learn parts of some of the answers to these questions together over the next few years.

7. References

[1] ACM/IEEE Joint Task Force, "Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering," vol. 2005, 2004.

[2] T. Shepard, M. Lamb, and D. Kelly, "More Testing Should be Taught," *Communications of the ACM*, vol. 44, pp. 103-108, 2001.

[3] D. Steinberg, "The effect of unit tests on entry points, coupling and cohesion in an introductory Java programming course," presented at XP Universe 2001, Raleigh, NC, 2001.

[4] M. Wick, D. Stevenson, and P. Wagner, "Using testing and JUnit across the curriculum," presented at 36th SIGCSE technical symposium on Computer science education, St. Louis, MO, 2005.

[5] V. Jovanovic, T. Murphy, and A. Greca, "Use of extreme programming (XP) in teaching introductory programming," presented at 32nd Annual Frontiers in Education 2002, Boston, MA, 2002.

[6] E. G. Barriocanal, M.-Á. Urbán, I. A. Cuevas, and P. D. Pérez, "An experience in integrating automated unit testing practices in an introductory programming course," *ACM SIGCSE Bulletin*, vol. 34, pp. 125-128, 2002.

[8] http://www.testingeducation.org/BBST/index.html

[9] http://fitlibrary.sourceforge.net/

[10] StepFixture, at www.testing.com/tools.html

[7] S. Edwards, "Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance," presented at International Conference on Education and Information Systems: Technology and Applications EISTA 2003, Orlando, FL, 2003.
[8] R. Kaufmann and D. Janzen, "Implications of test-driven development: a pilot study," presented at 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003), Anaheim, CA, 2003.
[9] G. Melnik and F. Maurer, "Perceptions of Agile Practices: A Student Survey," presented at Agile Universe/XP Universe 2002, Chicago, IL, 2002.
[10] R. Mugridge, "Challenges in Teaching Test Driven Development," presented at XP 2003, Genova, Italy, 2003.
[11] M. Müller and W. Tichy, "Case study: extreme programming in a university environment," presented at Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on, Toronto, Ontario, 2001.
[12] M. M. Müller and O. Hagner, "Experiment about test-first programming," *Software, IEE Proceedings- [see also Software Engineering, IEE Proceedings]*, vol. 149, pp. 131-136, 2002.
[13] T. Reichlmayr, "The agile approach in an undergraduate software engineering course project," presented at Frontiers in Education, 2003. FIE 2003. 33rd Annual, Boulder, CO, 2003.
[14] A. Shukla and L. Williams, "Adapting extreme programming for a core software engineering course," presented at 15th Conference on Software Engineering Education and Training, 2002. (CSEE&T 2002), Covington, KY, 2002.
[15] D. Umphress, T. Hendrix, and J. Cross, "Software process in the classroom: the Capstone project experience," *IEEE Software*, vol. 19, pp. 78-81, 2002.
[16] J. Elkner, "Using Test Driven Development in a Computer Science Classroom: A First Experience," vol. 2005, 2003.
[17] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*: Wiley, 2001.
[18] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd ed: John Wiley & Sons, 1993.
[19] G. J. Myers, *The Art of Software Testing*. New York, NY: Wiley-Interscience, 1979.
[20] J. A. Whittaker, *How to break software: a practical guide to testing*: Pearson Addison Wesley, 2002.
[21] C. Kaner, "The ongoing revolution in software testing," presented at Software Test & Performance Conference, Baltimore, MD, 2004.
[22] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley, 1999.
[23] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby: The Pragmatic Programmer's Guide*, 2nd ed. Raleigh, NC: The Pragmatic Programmers, 2004.
[24] K. Beck, *Test-Driven Development By Example*. Boston, MA: Addison-Wesley, 2003.
[25] P. Jorgensen, *Software Testing: A Craftsman's Approach*, 2 ed. Boca Raton, FL: CRC Press, 2002.
[26] D. Astels, *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall PTR, 2003.
[27] A. Hunt and D. Thomas, *Pragmatic Unit Testing in Java with JUnit*. Raleigh, NC: The Pragmatic Programmers, 2003.
[28] S. Holzner, *Eclipse*, 1st ed. Sebastopol, CA: O'Reilly & Assoc., 2004.
[29] J. B. Rainsberger, *JUnit Recipes: Practical Methods for Programmer Testing*. Greenwich, CT: Manning, 2004.
[30] J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy, *The Java(TM) Developer's Guide to Eclipse*, 2nd ed. Boston, MA: Addison-Wesley Professional, 2004.
[31] R. J. Sternberg, *Thinking Styles*. Cambridge, UK: Cambridge University Press, 1997.
[32] R. J. Sternberg and L.-F. Zhang, "Perspectives on Thinking, Learning, and Cognitive Styles," in *The Educational Psychology Series*, R. J. Sternberg and W. M. Williams, Eds. Mahwah, NJ: Lawrence Erlbaum Associates, 2001, pp. 276.
[33] T,A, Angelo and P. K. Cross, *Classroom Assessment Techniques, A Handbook for College Teachers*, 2nd ed. San Francisco, CA: Jossey-Bass