INSTRUCTOR'S EVALUATION OF THE COURSE CSE 4415/ SWE 5415

CEM KANER, OCTOBER 2009

PREPARED FOR THE ABET REVIEW

We've been teaching this course since Spring 2003. I've been instructor of record six times (three of these times, substantial teaching was done by doctoral students of mine, Pat McGee and Andy Tinkham). A seventh (2007) offering was taught by Pat Bond, who I regard as the most effective teacher (face-to-face instruction) in our department.

This has been a surprisingly difficult course for the students and the instructors. We have redesigned the course at least four times and made significant revisions at least two other times. The 2008 iteration reflected a fundamental shift in direction of the course. I think the specific 2008 iteration worked out less well than I had hoped, but it laid a good foundation for the 2009 redesign.

Intention of the Course

In early 2002, I began planning a course on the testing side of Extreme Programming (XP). My primary goal was to emphasize the new approach to unit testing and XP's heavy reliance on refactoring, building students' programming, testing and test automation skills. I expected other instructors to teach courses like this at other schools and saw this as the field's best shot at laying the foundations for the next generation of software-testing architects.

Common Challenges

Pat McGee joined my lab as a doctoral student after more than 20 years of industrial software development experience. Andy Tinkham joined as another doctoral student, after over 10 years of developing automated testing solutions and teaching these to practitioner audiences. McGee and Tinkham both liked XP and helped me refine the course. We worked through Angelo & Cross's Teaching Goals Inventory (Classroom Assessment Techniques: A Handbook for College Teachers, 2nd Ed. Jossey-Bass, 1993) to prioritize our objectives. We did this three times in the first four iterations of the course.

Each of us led the course at least once. We tried different tactics but faced the same challenges:

Student-Related Challenges

Students find the concept of test-driven development difficult.

- Several of the students flatly refused to believe that a testing course could require significant programming skills. 2009 is the first year that has been free of this problem. However, this is also the first year in which every student is either an undergrad or a

graduate student who studied at Florida Tech as an undergrad. This was the result of a registration-related anomaly. Our student mix will be different next year.

- Several students write programs by cutting and pasting code they find on the Net. Probably as a result, they are remarkably weak at generating their own implementations of basic operations (such as sorting or navigating through a linked list). This is widely discussed in the practitioner community (read Joel Spolsky's blog, *Joel on Software*, or look at the very basic programming quizzes that companies like Microsoft use to screen job applicants.) These students expect to snap together components rather than writing their own code. Test-driven development is mystifying to them because it demands that they build everything from scratch, justifying their implementation's iterations as they go.
- Several students have weak skills in problem decomposition. They don't know how to break a complex task into manageable bits. Some tell me that they are used to having the professor break the program down for them.

Each of us (Bond, McGee, Tinkham and me) saw these problems. We had a much higher failure rate among graduate students than undergrads every year, under every instructor. I interviewed Bond a few times (I was on sabbatical and often not in Melbourne to chat with Bond) about his experiences with the course. In our most detailed discussion of student performance, he indicated that at that time (before the end of the course), it appeared as though only one graduate student would pass. I think I had the same result in 2006 and grad performance in 2008 (with two exceptions) was abysmal as well.

We are not unique in seeing these problems. I host the Workshop on Teaching Software Testing (WTST) every year (this February will be our 9th workshop) and we have had several discussions of the challenges of teaching unit testing skills. In addition, there have been several presentations and posters at SIGCSE and CSEET--I've made a point of comparing notes with several of these speakers. I think these problems are widespread.

As a specific example, Stephen Edwards, at Virginia Tech, also teaches test-driven programming to his students. I think he is one of the most thoughtful testing/programming instructors in the field. He confirmed with me in several discussions that he finds, as we do, that students with significant industry experience are more resistant to adopting this style. He also finds that many graduate students are at a disadvantage in developing these skills. We discussed problem decomposition last year at WTST; several of the faculty said that this was one of the hardest skills for students to learn and one of the skills least explicitly addressed.

We also ran into three other challenges in dealing with student assessments:

- Several students failed to comply with instructions. We required them to develop their code iteratively and submit multiple iterations. They did not. We asked them to deliver code that did specific tasks. They would skip some tasks, sometimes very easy tasks. This was sometimes because a student couldn't do a task but often due to sloppiness (the student simply missed noticing the requirement or forgot to do it) or lack of time or a gamble that we wouldn't notice. (All three of these have come out in post-course interviews, sometimes done in informal discussion with a student a couple of years after the course completed.)

- Some students gambled that we wouldn't actually read their code and wrote the code in a way that didn't actually accomplish the task but did appear to pass its unit tests.
- Several students submitted weak work because they started working only a few hours (or for the takehome final, only a few days) before it was due. As many other instructors have reported at SIGCSE and other meetings, students who start a programming assignment relatively late are likely to do poorly. We have replicated that result year after year.

Instructional Materials-Related Challenges

We started teaching the course using Kent Beck's book on Test-Driven Development. It presented the basic use of jUnit well but failed to give students guidance on test design. The examples were too simple. The book ran out of content too soon relative to the level of skill and understanding our students needed. Web-based resources were oriented toward experienced practitioners and left our weaker students confused. Our better students could fend for themselves, but when half the class are trying to do the work but repeatedly fail to produce anything satisfactory, the course is inadequate.

We concluded that the students needed a better textbook, one that would take them pretty far along the learning curve before we demanded that they start doing significant tasks all on their own.

We adopted David Astels' 2003 book, Test-Driven Development: A Practical Guide with good initial results.

- Astels wrote his book around the test-driven development of a database application. We created a parallel assignment, creating a database of Magic cards instead of musical recordings. We broke development into stories (iteration specifications) similar to Astels, and encouraged students to use Astels' implementation as training wheels. Surprisingly many students ignored Astels and submitted code that was only sometimes adequate. The programming assignments in this course got much harder after the Astels project, and these students typically failed. However, the students who took our advice and worked through Astels in detail as a parallel task to writing their code built a good foundation for later work.
- Unfortunately, jUnit evolved rapidly as did the other tools relied on by Astels. Each year, students got more confused by the mismatches between Astels' text and the tools they were using. Ultimately, Pat Bond reported that in his 2007 class, students were spending more time trying to work around the errors (obsolete parts) of Astels than they spent writing their own code.

There haven't been any good introductions to the use of jUnit since Astels. I have spoken to exasperated faculty at other universities who are considering writing their own books, but so far, nothing has come of this.

One of the challenges is that even though "Agile" development has become popular, it has also become watered down. Recent surveys in Dr. Dobbs suggest that in a readership that boasts an over-80% adoption rate of "Agile Development", only about 13% use test-driven development. Most follow a process they call "Scrum" (I have heard enough mutually exclusive descriptions of Scrum, from enough people who are credentialed as Certified Scrum Masters on the basis of

taking a several-day industrial course, that I have no idea what someone means when they tell me they do "Scrum." But generally, when people using it at work describe it to me (as distinct from consultants at conferences), it seems that it doesn't involve much unit testing.)

This very low rate of adoption and weak sell-throughs of previous books on unit testing has led to limited demand for more books in the area. (As a successful author with a well-connected agent, I have a sense of how urgently publishers would like books on different test-related topics.) When there was a lot of TDD evangelism (2002-2005), publishers were very interested, but now, no one is aggressively soliciting manuscripts to introduce novice programmers to TDD.

There have been some good new books on TDD, but all of them are targeted to experienced programmers who are often already playing with TDD. These assume more programming sophistication than my students have.

The problem is not just with the books; it is also with the websites that support TDD. Early presentations of xUnit were often quite detailed, with a tutorial flavor that was inviting to relatively inexperienced programmers. xUnit has evolved in complexity but the online documentation for jUnit 4 is much more tersely written (and there are fewer online articles introducing it).

I tried Langr's *Agile Java* in 2008 after trying it in introductory programming courses. This is an allegedly-introductory book on Java that introduces TDD from the start. The book has several weaknesses, not least that it teaches the mechanics of test-driven development (how to use several tools) but provides little guidance either in test design or in refactoring. The most serious problem, though, is that the book follows a strict order for 230 pages and assumes knowledge of those 230 pages throughout the rest of the book. The exercises are cumulative (you can't skip to every second one) but demotivatingly tedious.

In 2009, we are using a different objects-first Java book, with our own assignments that are based on the book but tailored to the class. This seems to be working better, but we really need a book that integrates an introduction (or re-introduction) of an object-oriented programming language with xUnit for that language with advice on techniques for implementation-level (e.g. unit-, protocol-, and below-the-GUI integration-) testing and advice on designing production classes in ways that make them testable but still secure. This course will stay unstable until we have that book. I don't have time to write it. I expect that over the years, this course will drift away from teaching competent application of xUnit (or more generally TDD) and toward more emphasis on design of implementation-level tests.

Based on discussions with some members of the Agile community, I don't expect the current thought leaders to write many (perhaps not any) introductory articles or books on xUnit over the next year or two. This year's Workshop on Teaching Software Testing will take teaching implementation-level testing as its theme. I hope to put together a group of university instructors who will develop some textual and video tutorial-level materials and peer review them before we publish them on our websites and revise our courses to rely on them.

Task-Related Challenges

Until 2008, we included a maintenance programming assignment in each class. Students would work in teams on a small open-source test tool, adding a lot of implementation-level tests and a

little bit of enhancement. Our best year for this was probably 2006. The task was too complex for any of the teams, but we allowed the teams to merge into a classwide collaboration that made a lot of progress. Students who worked hard in the team gained a lot of insight. The tool improved. And several students skated past the work. In general, the diversity of skill and attitudes in the classes (exacerbated by the mismatch between grads and undergrads) and the difficulty of finding an open source tool that is challenging enough but not overwhelming, made this unsustainable. I was especially concerned about students who were slow (inexperienced and not terribly confident) programmers, who put in such long hours on this course that it seriously interfered with their ability to succeed in other courses. We were crossing the reasonable limit on student workload, and I decided to drop this feature.

If this course was only for undergraduates, and especially if this course was only for undergraduates who enrolled voluntarily (rather than having it as a required course), I would bring back the maintenance project. I think the 2009 students could probably handle it. But as long as we have significant grad student enrolment, the burden on the strong students to carry the weaker students becomes unmanageable and the burden on the weaker students to do adequate work is impossible.

In several iterations of the course, we included student presentations, in which students showed off some code to the rest of the class or laid out a topical area (e.g. refactoring, source control, etc.). In some years, the study also earned bonus points by assuming responsibility for coaching the rest of the students in this topic over the rest of the term. This was good training for the students in communication skills (which were not part of the core objectives of the course) but they slowed our coverage of key topics and the class got awfully confused when they had to apply something that had been poorly presented. We included some presentations every year, through 2008, but are no longer including presentations in 2009.

Designing the 2008 Iteration

In 2004 to 2008, I opened the course with a description of the preceding year's final exam and the ways that students had blown the exam. I had three objectives:

(a) chase away the students who are not going to pass the course. This might sound harsh, but given our large foreign student population and the inflexibility of Homeland Security toward foreign students, I want to reduce the frequency of desperate students begging me to raise their grade to a pass or desperately attempting to cheat their way to a pass.

(b) alert students to common mistakes in the course, hopefully to encourage this cohort to not make them

(c) initiate the effort to persuade students to begin their final (takehome) exam early enough, so that they will have a chance of passing it.

In addition, in 2008, I gave students an opening survey, to assess their programming skills. As expected, the course had a bimodal distribution of student skills, whether I measured this in terms of self-evaluation or by their answers to the closing two questions (one on simple testing of an unsigned integer--but in a way that gives insight into whether they have a clue of how numbers are stored and the other on floating point precision.)

Given that distribution, which persisted through the first few weeks of the course, I targeted the level of the course to the middle of the distribution, cut out the maintenance assignment, decided to stick closely to Langr for the 1st half of the term, and focused lectures either on the skills students needed to do better testing than Langr's examples, or on the skills / knowledge students needed in order to get through Langr.

I decided to focus on programming within a single language (we added test-driven web programming using Ruby and then applied this knowledge--of testing through the COM interface--back to a final exam in which students wrote a test tool in Ruby that did high-volume function-equivalence testing of Excel versus Open Office Calc)

By the third week, several of the weakest students had dropped the course voluntarily. Shortly after that, three others were confronted with evidence of plagiarism of their assignments and they withdrew from the course. We still had a broad distribution of programming skill, but many of the students most likely to fail had left. Based on this, I decided to be a little more ambitious in my design of the final exam.

Note that what is going on here is ***not*** that I am varying a course "standard" to suit the students. The course has no settled standard to vary. I am trying to calibrate the course to a level that (a) addresses my learning objectives for the course (b) at a level that is challenging but not impossible and not unfairly time consuming to a hardworking C student. We found a level we were happy with in 2005 but the obsolescence of Astels and the lack of a suitable replacement text changed the student experience and the difficulty of the course.

Running the 2008 Iteration

I collected short surveys from students on most days to get feedback on my teaching on an ongoing basis. This is tactical-level feedback. I've included a copy of the form that I use, but I consider the responses to be private data, much like the tracking data that Watts Humphrey encourages programmers to keep--and keep private--in his *Personal Software Process*.

The feedback (including several private discussions with individual students) and student performance reinforced my opinion that we needed an objects-first introductory text to our programming language (Java, C#, Python or Ruby were all plausible candidates) because many of the students were encountering Java in a way that they never had before, and they were essentially relearning it. This time around, students were learning how to code:

- In a way that let them always know the state of their code
- In a way that required them to get each piece of the solution working before moving on to the next
- In a way that relied heavily on tools (an IDE, a code coverage monitor, a style checker)
- In a way that required them to use a version control system, document changes from iteration to iteration, and make intentional, predictable choices about when to treat an iteration as complete (checking it into the VCS).

For most of the students, this was their first experience with what I would call "software engineering" as distinct from "programming."

Unfortunately, It became clear as the course progressed that Langr wasn't the ideal book. For 2009, I would either create a fundamentally different stream of exercises to use with Langr or choose a different book and create exercises/assignments suited to that one.

The main mistake that I made in 2008 involved the exam.

The National Institute of Science and Technology published a set of requirements for VoteTest, a test tool that assesses the adequacy of electronic voting equipment. I presented to the class the idea of implementing parts of VoteTest (test-first, under source control) as the take-home final exam and they liked the idea. I circulated drafts of the proposed exam, we reviewed them in class, hammered out ambiguities, added a couple of supporting lectures, and then started the exam three weeks before it was due.

Up to this point, if I had it to do over, I would do everything in essentially the same way.

Unfortunately, I forgot to dramatically overemphasize the need to start on this exam early. The students thought they understood the exam and I mistakenly thought that they understood its difficulty. In fact, they were overconfident. All of them started late, and all of them got into last-minute crises. No one finished the exam. I gave a 1-day extension but could not extend beyond that because we have an appropriate policy that within-term work (including takehome exams) cannot extend into exam week.

Design for 2009

I worked with three of the 2008 students on the 2009 redesign. Two of them suggested ideas for homework and assignments. The third helped me search through books on Java, Python and C#, looking for an appropriate replacement text. I hired him as the TA for the course and we are co-teaching the 2009 iteration.

This year's course is more traditionally structured, with a higher emphasis on lectures and less in-class time for labs. There is still a lot of homework, though less than last year. The midterm and the final are both programming take-home exams. The midterm, like last year, is a relatively simple programming task (though more complex than last year's). Student performance on this will serve as a critical diagnostic and I will reappraise progress in the course on the basis of it.

I intend to reuse the 2008 final exam, with modest revisions. However, I am revising the course structure to better prepare students for this exam:

- Students were remarkably weak at file I/O, despite our coverage of this topic in class (lecture and homework). The notion of driving tests from input-data files and of comparing test results against expected-results files was challenging for all of the students. We had discussed it in class but their mental model of tests was either manual input or data hardcoded into a jUnit test. The voting system requires configuration files (e.g. specifications of the candidates and their parties, specification of the layout of the ballot) as well as data files (simulated votes, simulated output). This complexity was very difficult for students to manage intellectually, even though we had discussed this aspect of the exam in class when reviewing the draft questions and draft scope of the exam.

In 2009, we did homework on test-driven development of file I/O before the midterm and students are working with input-data files, output-data files and test oracle files as part of the midterm exam.

- Students were remarkably weak on the use of a random number generator to simulate random errors. (Imagine approaching a ballot-scanner type of voting machine with a pile of test ballots. We know what's on the ballots. We can simulate the reading of the ballots by the machine by generating a test result file in a way that usually copies every input correctly to the result file but sometimes randomly changes a vote. In our testing, we can bias the random-corruption method to make more or fewer errors and to make the errors more or less systematically. Systematic corruption manifests as several errors that favor one candidate or party over another. Now, can we assess our result files by comparing them to the known good input sets and statistically test for systematic versus random mistakes. In the practitioner community, there are quite a few variations on this theme.

  In 2009, rather than spending more time on topics specific to xUnit and very popular in the more advanced books (such as the use of mock objects or the most effective ways to organize test methods into classes and packages), we're spending much more time on random numbers and simulators because I think these are core test tools.

If the 2009 course is enough to prepare students well for the 2009 final, I'll adopt the same strategy next year:

- Before the course starts or early in the course, identify an application or task that would work well for the final exam and rough-draft the exam
- Present students with the rough draft
- Restructure some of the lectures and assignments to practice up skills the students will need on the exam

Cem Kaner

From: Cem Kaner [kaner@kaner.com]
Sent: Tuesday, August 19, 2008 12:51 PM
To: 'wds@cs.fit.edu'
Cc: 'Karen Brown'
Subject: course file, CSE 4415 / SWE 5415
Attachments: Syllabus5415-2008.pdf; ProgrammerTestingPretest.html

Please file in the binder for CSE 4415.

I spoke at length with Pat Bond last year, while he was struggling through CSE 4415/SWE 5415. Pat was the third instructor (Pat, Tinkham, me) who had a very high graduate student failure rate. We compared notes on the underlying problems and I think the following represent a consensus view across several years of teaching. (I also met with Andy Tinkham while in Minneapolis at MERLOT two weeks ago).

1. There are no good books for teaching test-first programming to relatively inexperienced programmers. Of the books that have attempted to reach this market, all but one are unusably outdated. Pat attempted to use one of the outdated books and his students reported spending more time working around the book's errors (mismatches with current tools) than on designing / writing their code.
2. There are a few books for working professionals who have significant programming experience. These might be useful with several of our undergraduates, who are often strong programmers. Unfortunately, for most of our students, including our typical undergrads (who are stronger programmers than the typical grad students), these books assume too much practical, real-world intuition and provide too few examples and too little scaffolding. I ordered the most recently published of this group from Manning Press, to serve as the course text for this year. However, it provides very weak scaffolding. The first 6 chapters use a trivial example (hello world) as a background to many interesting discussions of programming practice. I enjoyed these chapters. But for a student just learning this material, the main work would be in assignments and examples developed outside of the book, by the instructor. This is always possible to do, but it takes a lot of time. There are very few learning objects in the repositories to support this.
3. Many of our students, especially graduate students, are too inexperienced to start at this level. Unfortunately, by the time many realize they are in deep trouble, it is too late for them to substitute a course for this one, and they cannot drop without going underload, a big problem for foreign students because homeland security looks askew as underload students.
4. Students have several other types of resistance to this material, which I laid out in the syllabus. I spoke with Steve Edwards while I was at NSF last week (Edwards is teaching test-first programming of GUI objects, with some cool tools and good results). His students, and the students of some other instructors who have used Steve's materials (and were dropping by his poster presentation area to visit), seem to do the same things, and have the same problems as ours.

I am now convinced that the problems with CSE 4415/SWE 5415 have not been in the teaching but are in the mismatch between the course and the students.

I gave students a half-hour knowledge inventory last night and then discussed the course design with them until the end of class. I think we have a consensus on a new approach that will work them very hard but less ambitiously in terms of the final projects. I think a couple of students are disappointed and have offered to coach them on a more ambitious variation of the course, in which they do some of the stuff we're dropping from last year as bonus work.

I don't think the overall course objectives will change. I do think that we will revise the prioritization of objectives. I also think that the weaker programmers among the grad students, if they work hard, will come out much stronger in programming and in testing, which is a better result than coming out flunking.

We are doing a bottom-to-top redesign of this course. I'll know more about its projected long-term structure in a month.

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering, Florida Institute of Technology
www.kaner.com
www.testingeducation.org
http://www.satisfice.com/kaner/

CSE 4415 / SWE 5415 Opening Survey

NAME:

Are you currently enrolled in this course? Yes / No

1. How strong are your skills in Java programming?

- ○ Expert
- ○ Very competent
- ○ Adequate
- ○ Still learning
- ○ Weak

2. How recent is your programming experience?

- ○ I've been doing significant programming very recently
- ○ I did significant programming a while ago but I remember most of it and will pick it back up very quickly
- ○ I did some programming a while ago and will need some time to rebuild my skill
- ○ I studied programming in school but have not done significant production programming and will need time to pick up what I used to know
- ○ I studied programming in school but was never a really strong programmer

3. Please comment on your answers (how strong / recent is your programming experience)

4. Describe your use of assertions in your programming.

5. Have you done unit testing? What tools did you use? What was your biggest challenge? Your biggest success?

6. What is test-driven programming? Describe any experience you have doing test-driven programming.

7. What is refactoring? Describe an example of refactoring that you have done of your own code.

8. Have you done maintenance of someone else's code? If so, please describe an example of what you did and what the major challenges were.

9. What is domain testing? What is boundary analysis? Describe tests you have designed using domain testing and/or boundary analysis.

10. Imagine testing an Integer Square Root function. The function reads a 32-bit word that is stored in memory, interprets the contents as an unsigned integer and then computes the square root of the integer, returning the result as a floating point number.

- What values can you input to this function?
- Can you imagine any invalid inputs to this function, inputs that should cause the function to return an error message?
- If you were to test ALL of the inputs to this function, how many tests would there be?

11. If a program computed the square root of 4 and reported 1.9999999999999999, would that be a passing result or a failure? What about 2.00000000000000001? How close would the answer have to be to 2.0 for the result to be a pass? Why?

Experiences Teaching a Course in a Programmer Testing

Andy Tinkham
Florida Institute of Technology
Department of Computer Sciences
150 West University Blvd
Melbourne, FL 32901
andy@tinkham.org

Cem Kaner, J.D., Ph. D.
Florida Institute of Technology
Department of Computer Sciences
150 West University Blvd
Melbourne, FL 32901
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. 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/

Specific course objectives for the programmer-testing course vary from year to year, but they fall within some broader requirements:

- 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

Kaner's general approach to course development is evolutionary:

- He gives 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. Kaner's preference as a teacher is to work with the people in the room, helping them grow, rather than persevering with a plan more appropriate to a different group. His 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. He can prevent or control many of the problems that came up last time, but there will be new problems and he will make new mistakes. Kaner's standards are still floating, heavily influenced by student performance and feedback.
- By the third iteration, he can anticipate and manage the typical-to-this-course student problems and excuses. He can reuse and improve a fair bit of material from past years instead of inventing everything. And, at the start of the term, he can 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. Perhaps because of his background as an attorney (and a prosecutor), Kaner is 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. If one of Kaner's courses will ever be too demanding, it will probably be in his third or fourth teaching of it.

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) or a new tool (such as Ward Cunningham's Framework for Integration Testing (FIT) [2]) 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

Here are some elements common among the three instances of the course:

- 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 work. Getting two college students in the same place at

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

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[3]. 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).

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 programming 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. Kaner even encouraged students 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 slow 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 here-introduced the class to TDD and demonsted more complex examples in Ruby. His demonstration also introduced

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

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 programming 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 programming with a simple example, but moved quickly to an introduction to two testing tools, Jenny[4] (a test tool to support combinatorial testing, written in C) and Multi[5] (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 programming when given the chance, they would flunk the course,
- would adopt one of the recently published books on test-driven programming that had other examples and included more discussion of test design,
- would introduce test-driven programming with some more complex examples.
- would add a book on Eclipse to the required reading list to deal with students who protested that they

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

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[6], 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 majority of students were already familiar with this game and had cards. We provided cards to students who lacked them, along with a student presentation on the basics of the game and hosted a weekend afternoon of game play. 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

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

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. Kaner 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 Subversion, 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 programming is contra-paradigmatic 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 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 programming.
- Test-driven programming 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, especially JUnit with Eclipse, provide an experimenter's foundation, 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 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, 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 iterations.
- JUnit, Eclipse and Subversion all helped students do complex tasks. Next time, we'll add build management with Cruise Control or Ant.
- We understand that a book on Ward Cunningham's FIT is coming soon. Along with supporting acceptance testing, FIT supports test-driven, glass box integration testing. This is important knowledge for this course. We expect to also include FitLibrary and FolderRunner from Mugridge[7] and StepFixture from Marick.[8]
- 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] http://fitlibrary.sourceforge.net/

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

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.

[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.