An Interview with Cem Kaner, Software Testing Authority

Part II: How to Educate and Train Testers

by **Sam Guckenheimer**
Senior Director of Technology for Automated Test
Rational Software

***Cem Kaner**, Ph.D. J.D., is Professor of Computer Sciences at Florida Institute of Technology. He is perhaps the world's most prolific and widely read author, consultant, educator, and attorney in the field of software testing.*

In Part I of this interview, featured last month, I discussed with Cem Kaner, Professor of Computer Sciences at Florida Institute of Technology, his notion of context-driven testing and the course development he has done for Rational over the past year. As perhaps the world's most prolific and widely read author, consultant, educator, and attorney in the field of software testing, Cem concludes this interview with his insights on testing education, its relationship to consulting practice, and his views on "agile" software development.

Guckenheimer: Now that you have been a professor at Florida Tech for two years, what have you learned about educating software testers? Is there anything different now about the way you train software testers in a university setting compared to a commercial setting?

Kaner: As a university professor, I have two luxuries now that I didn't have when teaching in an industrial setting. First, I can actually give my students tests, and they are motivated to take and pass them. I can also

give them homework and evaluations. In an industrial course, you just can't do that. Even if you have a very light test at the end, it's not the same thing as giving someone an assignment that will require a week of intensive work with a colleague. Through giving and grading student assignments, I've learned that some of the concepts I thought were very clear are very confusing to people with little testing experience. For example, looking at a situation and assessing "What are the boundary conditions in this case?" takes a remarkable amount of practice -- at least three to four assignments before most students get really good at it. They need practice, via ungraded or lightly graded assignments, dealing with similar problems time after time. You can talk about it over and over, but the main concept has to spark in the student's head so they go, "Oh, I get it." That typically only happens with practice.

A lot of what we're doing now at Florida Tech is drafting self-paced, self-answering homework questions. For example, I give you a data entry field; you analyze this field and come up with a boundary case, and then I'll give you what our analysis of the same field was. Then we will give you a word problem that asks you to figure out what the field is, or what the variable is, that you're studying, and then we will extend it.

Consider the way we teach boundary analysis. A student enters the highest number possible for a given field, then enters the highest number plus one and tests both of those. What's the reason for these specific values? Historically, we know that the program is a little more likely to fail under these conditions than with a valid number that is big but not the biggest, or with an invalid number that is too big, but not right at the edge. So, as in this example, we teach a theory of error. And what we're doing in boundary testing is identifying a class of test cases: all the valid numbers, all the numbers that are too big. Then we find representatives for these classes: the biggest valid number, the smallest invalid, too-large number. And we say this is a representative of the class that is slightly more likely to show a failure than other members of the class, and since you can't test all members of the class because there's an infinite number of tests you could run -- nobody ever has enough time. Typically, you're restricted to using one or two or some very small number of members of any class you could test. And so you're always looking for better representatives, representatives more likely to produce a problem.

Once students practice with simple boundary analysis and with the question of combining boundaries across several different variables, we start pushing them onto the next notion: What other ways are there of identifying risks? How do you find classes that will expose the risk vs. classes and tasks that will not expose the risk, and how do you come up with representatives that are worth testing?

In my experience, the more practice I can give students with this sort of exercise, which they can do at home, the more likely they are to get the principles behind it. So I have graduate students who are spending a lot of time trying to figure out how to create useful practice exercises. Ultimately, we'll probably come up with a set of materials like you see in Schaum's Outlines, which everybody who has studied either math or physics has probably used. They're just light summaries of technical material with worked examples, then lots of exercises that you practice

until you can finally solve a certain class of problem.

As a consultant, I had thought that people needed more practice with these concepts than they were getting. But there was no way I could experiment with a different style of teaching in a corporate setting, and there was no way that employees with real deadlines would come to a course that included a lot of drills. And it takes a remarkable amount of time to envision the real tasks that require practice and then come up with good exercises to provide that practice.

As a professor, I have the time and a series of involuntary subjects, as it were, to research a better curriculum. I get to try things out that I hope will improve the course, and most of them actually do. I also have students who have gone through the course and are quite enthusiastic about trying to develop practice materials, a squad of intellectuals who will get some academic credit but whom I could never afford to pay if I were a stand-alone consultant.

Guckenheimer: What kind of background do your students have, and where are they headed?

Today, I deal only with students who can write code, and we teach them how to test their own code or the code of a peer. Everybody who comes into my course is in a software engineering or computer science program and has already taken several programming courses. The first testing course covers traditional black-box testing, and the second course starts them off, first day, working with JUnit.

Many of our students at Florida Tech graduate and become professional testers in software development organizations. So a lot of what we think we're trying to do is to train the next generation of testing architects. Typically, these are people who have a lot of software development insight who either need to build tools themselves or evaluate tools and train their own staff in how to use tools really well, and to write the kind of support materials that make a specific tool useful. There is no test automation tool that solves *all* of an organization's problems, or works perfectly on its own. There is always plenty of work that needs to be done inside a company, either to change the vision of testing or to organize data or code in a way that makes it more compatible with their tool of choice. We're trying to train a generation of folks who can go out and help do that.

Guckenheimer: Are you implying that in the area of software testing, students who lack a certain real-world awareness or experience are at a deficit?

Kaner: I actually do believe that people without practical experience have a lack of perspective in tests. Earlier in my career, when I was a hiring manager, I was very disappointed when I would interview someone who came out of a traditional computer science program, and find that their testing course was fundamentally theoretical. They had no idea how to

apply that theory. We have to work very hard when we teach the testing course to provide a lot of real-life examples. We also go out and get a sample application -- some software that is under development -- and structure the assignments and much of the course around beating this program into the ground. We used Star Office last year, we used Microsoft PowerPoint once, and we used the Texas Interactive Calculator. I'm not sure what application I'm going to use this fall, but it's absolutely essential for these students to get experience with something real, or everything we teach will be academic and not necessarily very useful in the future.

I also teach the brand new metrics course here. I had a class of 15 students, mainly graduate students, and only five of them had substantial, real-life experience in software development. As I talked about when something is used, how it's used, how it can be misused, the risks to the organization of applying this measurement method, and so forth, they would understand what I was saying, because they had lived it. The other ten had incredible trouble understanding what I was getting at. Plus, unless you have the experience to understand which measures are useful when, what risks are associated with a given measure, when a given measure will have some validity, and when you can learn something from the numbers you collect, then you're like a loaded gun in the hands of an organization that really hasn't had any training in how to use it.

The folks who teach software architecture courses experience the same gulf in assimilation of theory between students who have attempted to design a moderately large program under real-world circumstances versus those who have not. So I don't think this phenomenon is unique to testing instruction. I think that, in many fields, returning students who have real-life experience are much more likely to grasp the subtleties than students who are going straight through.

Guckenheimer: I think the National Science Foundation has recently awarded you a grant to provide useable educational materials in software testing more broadly. Is that targeted to working professionals in the field? What can you tell us about that grant?

Kaner: The grant, Improving the Education of Software Testers, focuses on academic instruction for software testing. My application emphasized that there is very little in the way of academic resources -- few courses, no good textbooks, and no practice materials -- in software testing. There is no well-understood method for testing instruction as there is for teaching calculus, for example. So I wanted to put together materials that would help people build testing courses more effectively: practice exercises, and sample course notes and test tools. For example, we're writing a test program for "all pairs," a technique for dealing very efficiently with circumstances involving many variables to test together, and it lets you find a very large percentage of configuration problems with a much smaller series of tests. There's a very fine all pairs test tool on the market, but it's expensive for testing a small number of variables, such as ten, in combination. So one of my students, Nadim Rabbani, in collaboration with another Florida Tech student, Hugh Thompson, is almost finished writing an all pairs test tool that will handle up to ten variables in

combination that have maybe ten values each. These tools will be somewhat useful in industry, where some people have problems on this scale that they can't work out by hand. But where it will be most useful is in a classroom setting, where you can say to the student, "Here's the concept of combination testing, here are some thorny combination problems. Try to work these out by hand first, then use the tool and compare your results." They'll learn what this free software tool can buy them, and if they get into more complex circumstances, they'll understand why they might want to have their company invest in something more expensive.

In addition, two of my master's students, Giridhar Vijayaraghavan and Ajay Jha, are studying how programs fail. *Quality Week* will soon publish Giri's taxonomy of shopping cart software problems, which classifies a broad range of risks. If you just went to Amazon.com and imagined how to test a shopping cart, you'd come up with a few examples of what might go wrong. But with Giri's taxonomy you can start thinking by analogy about how particular programs might fail and come up with hundreds of test cases that will uncover real problems.

Though the focus of the funded work is academic, testing is an applied area; it would be foolish to think about how to teach it without considering how testing is conducted in the world. Any of the materials that we make available to faculty we're also making available to corporate teachers and trainers through a site we will soon be opening called "TestingEducation.org" Anyone will be able to download materials, like my course notes, for free. People who teach, whether in a commercial or university context, will be able to get a special password and access things like examination materials, exercises, and teaching tips that students won't have access to, but eventually we'll have practice exercises for students. The public pays for my National Science Foundation Grant, so they're entitled to this Web site.

Guckenheimer: That's great news for the testers out there. Of course, a lot of *Rational Edge* readers are not testers and test managers. How does your work touch other players in the development life cycle: requirements analysts, developers, and others.

Kaner: Everyone who goes through the software engineering program at Florida Tech is required to take two full courses on testing -- whether they want to become architects, requirements analysts, programmers, or testers. That's because we think testing is a core competency for anyone doing development. A programmer who tests his own code -- and most people do -- is going to learn better testing strategies in this course. Another takeaway from a testing course is wisdom on how to manage a project that involves many testers. And the Rational course I helped develop offers a lot of wisdom regarding where testers fit in the lifecycle and how they will interact with the rest of the company.

Our Web site will focus more on practiceable and trainable skills, which means the site is going to be very boring for somebody who doesn't want to learn how to do the technical parts of testing really well.

Guckenheimer: One final thread. We've just been talking about the connections among different participants in the development lifecycle. Through the course at Florida Tech and your own research, you've had some exposure to the Rational Unified Process.® I'm interested in your perspectives on RUP® and other process movements, such as the Agile community, and how they address testing.

Kaner: I don't want to speak to Agile Development in general, but I will speak to Extreme Programming (XP) and say that, like RUP, it has a very strong vision of lifecycle. It also has a very strong vision of some types of testing. But most of the most skilled testing that my colleagues and I know how to do doesn't fit in the XP approach. In place of strong, test-first programming (which is a wonderful practice), XP substitutes customer stories and either testing by a customer or testing by a customer's advocate, against what really look like scenarios based on use cases. This approach can expose a whole lot of problems, but it will also miss a whole lot of problems, and the framework for having an open, intelligent discussion about what the other methods of testing are and how they might fit into this scheme just isn't there. XP has a fairly narrowly patterned "right way" to go about doing things -- it's pretty good for many contexts, and not so good for others.

The Rational Unified Process is much more flexible. It's more tailorable to many circumstances; you can imagine using its iterative lifecycle approach on very small projects like computer games. And it can scale up to large telephony systems. The testing styles would have to be very different for those larger and smaller systems, and that poses a challenge to the RUP authors in terms of describing different styles and when they're needed. For example, a boundary-condition style tester will interact with folks and produce one kind of deliverables through a particular set of questions, whereas a scenario tester who bases most of his work on use cases and models developed for the system is going to come in with a whole different series of questions. And different styles of testing might be called for on a large project at different points in the lifecycle.

I was motivated to work on Rational's *Principles of Software Testing for Testers* course because I would like to see Rational extend the practical guidance available for testers in RUP. Two of my graduate students are also writing RUP extensions to provide guidance on some of the testing techniques covered in the course.

In particular, I'd like to see RUP go deeper on this problem of how testers in an iterative development lifecycle will do different kinds of testing at different times, and how they can adapt to a project team that is following a lifecycle that has a traditional basis, but is really its own variation. Over time, RUP needs to extend the library of templates and checklists, and cover skills that we drill in the course, such as bug advocacy, i.e., the effective communication of change requests so that other teams members will act on them appropriately.

Guckenheimer: We're really glad to have worked with you on the course and we're looking forward to incorporating those extensions. Thanks very much.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!