An Interview with Cem Kaner, Software Testing Authority

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.

Last year, Dr. Kaner coauthored, with James Bach and Bret Pettichord, Lessons Learned in Software Testing: A Context-Driven Approach. One of his previous books, Testing Computer Software (coauthored with Jack Falk and Hung Nguyen), is a standard text for training software testers. Many of his articles on software testing are available at www.kaner.com. In addition, as an attorney, Dr. Kaner has been active in developing the law of software quality, and he was elected to the American Law Institute in recognition of his work.

Rational University recently engaged Dr. Kaner to develop content for a new course, Principles of Software Testing for Testers. Dr. Kaner will teach this course on August 17-18, immediately before the Rational User Conference (RUC) in Orlando. (The course will be available from Rational University instructors shortly afterwards.) At RUC, Dr. Kaner and his coauthor James Bach will also deliver a talk on context-driven software testing.

I recently had the pleasure of speaking with Dr. Kaner regarding the new Rational course, his work on context-driven testing, his new book, his curriculum development activities, and some of his foundational ideas in *software testing. I will share the highlights of that conversation with you below.*

Sam Guckenheimer: Let's start with your book *Lessons Learned*, which you published about half a year ago. We liked the book so much we featured part of it on Rational Developer Network. It's generated a lot of interest, a lot of praise, and a little controversy. What drove you with James Bach and Bret Pettichord to do *Lessons Learned*?

Cem Kaner: The three of us were pretty enthusiastic about some aspects of the patterns movement. As I see it, the patterns movement involves a structured writing style for taking well-understood learning and trying to communicate it to other people. Although the structured style of writing in patterns looks very easy -- you have a bunch of subheadings and you fill in stuff for each one -- the method is time-consuming. And the essence of what you have to say gets lost inside all of the other components in which you don't have much special to say. I've done a lot of writing within structured constraints. In the first edition of *Testing Computer Software*, I tried to do something like that in the Appendix for bug descriptions, and discovered that I could make a much better product -- and not hurt the reader at all -- by focusing on the nugget of what I had to say and leaving out the other details. So rather than writing a book called *Patterns of Software Testing*, which came out of our first discussion, we said, Why don't we just write a book called *Lessons Learned*? We would consider a bunch of the things we had learned very well, extract the essence of those, and instead of putting them into a structured form, put them down one at a time and see what developed. *Lessons Learned* was the result.

SG: When you were creating *Lessons Learned*, did you set out to define context-driven software testing based on context and forces in patterns, or did it just emerge?

CK: The idea of context-driven testing had emerged in our group years before. In fact, Brian Marick, James Bach, and I started writing a book in 1997 to define the context-driven school. We opened the software test mailing list (http://groups.yahoo.com/group/software-testing/) specifically as a home list for the context-driven school. We were talking about context-driven testing a lot, but we were talking about it within an inner circle and polishing our ideas before exposing them under that name to the rest of the community. At some point during *Lessons Learned*, we realized that Brian would be happy to have the three of us kind of announce the school without his participation as a co-author. We were hesitant about doing that, since he had done so much work in this area. In any case, by the time we began work on the book, many of the context-driven ideas were quite mature.

SG: How has the reception been to context-driven software testing?

CK: I think a lot of folks have responded that it's what they do anyway, so while they're glad somebody is putting it into words, it's not really a big deal. Other folks have found it liberating, and some have found it intriguing; it's gotten them thinking. And some people are deeply offended by it.

We're not surprised by the negative reactions. Many people in the testing community feel there is "*one* right way" to do certain things. They know the "right" lifecycle, the "right" test documentation method and test techniques. And then we come along, saying, "You know, no technique is good everywhere, and every technique is good somewhere, and the task is to figure out when something will work and when it won't, rather than whether it's good or bad." Some folks think that we're engaging in sloppy thinking and are personally offended by it. Some consultants don't know how to adapt their practices to include it, and simply attack it as something different from what they've been teaching for many years.

SG: I'm curious about the earlier work you've done on paradigms of testing which, of course, explores the notion that people have different ways to do software testing and that all of these different schools claim their method is the right one. Was that a driver behind context-driven testing?

CK: The paradigm notion was, for me, a very important driver. That early research was the first time that I had worked with anyone else to crystallize some of the notions of context. The history of the paradigms is kind of fun. When I was first breaking into consulting in the 1980s, I was working full time but would do consulting at night and on weekends for anybody desperate enough to hire me. You can imagine what sort of test manager or development manager would be willing to give up a late Saturday night to talk with a testing consultant. Those people were in deep trouble.

I would, in those days, tell them about domain testing, using boundary conditions, and about what I now call scenario testing, based on real-life examples of how people use the product or how we would like to imagine different users working with the product. And they would follow those principles, things would get much better, and they would think I was a genius -- and, of course, I thought I was pretty knowledgeable back then, too. Then I became a full-time consultant and started selling my services to people who weren't so desperate that they were willing to meet with me at midnight. They expected me there at normal business hours, they weren't in terrible trouble, and I would see them using methods that were "wrong." I don't know how else to put it. Fortunately, before telling them that everything they were doing was crazy, I had enough sense to ask for access to their customer support database, and I would take a look at what bugs they had in their bug tracking system and what complaints they were getting from customers. This revealed what bugs they had found and missed, and I realized that they had found things with their techniques that would have been very hard for me to find. There were a few things I had the ability to find that they had missed, but often, we simply had different visions of what good testing was, and these visions were yielding different, though quite effective, styles of testing.

Now, while these folks needed consulting -- they were certainly not as effective as they wanted to be -- it was nevertheless remarkable how much progress they could make following relatively few of the design principles that I thought were basic. So I would take that company's ideas and put them in my toolkit and go on to the next company, only to find out they were doing something *else* that was different.

I had identified nine basic styles by the time I met James Bach at an American Society for Quality meeting in Dallas. We had e-mailed for years, and our first face-to-face meeting was extremely productive. We spent six hours in the Dallas airport talking about test design. He pulled out a list of nine basic testing styles, and lo and behold: They were the same as mine. The names were slightly different, but they were the same. For each style, we could name a company that relied almost exclusively on that style; if you talked to them about some other style, they'd say, "that's not testing," or "that's not interesting," or "that reveals bugs that nobody cares about."

As we further discussed these nine styles of testing, we agreed that the phenomenon looked very much like what Thomas Kuhn described in *The Structure of Scientific Revolutions* as pre-scientific paradigms. A paradigm really defines your scientific worldview. You have a set of data, you have a theory associated with those data, and you have measurement techniques or experimental techniques that are considered useful for finding new research results. Plus, you have a bunch of unanswerable questions that are out of scope relative to what you're currently working on and what you expect to continue working on within your field. All fields of science, over time, undergo revolutions in the ways problems are identified and resolved. Problems that used to be considered out of scope eventually offer an entirely new way of looking at the field. Practitioners start viewing those previously uninteresting and out-of-scope issues as central to the field of study. Bach and I both had had the experience of persuading people to adopt one or two new testing styles in their company and watching a transformation in their attitude about certain kinds of problems and test methods. So we started working on ways to communicate our style list to others.

Now, as soon as you come up with the notion that there are styles that overlap but are far from completely overlapping, you end up asking the question: If I were aware of all or most of these styles, how would I know when to use one or the other? What's the cost-benefit associated with one versus another? And you get into absolute contextual reasoning at that point. You have to ask questions like: What are the skill sets of the people who are doing this? What are the quality standards of people who have influence over development?

Quality standards are a funny thing, by the way. I was at Electronic Arts when we built Chuck Yeager's Flight Simulator. When I talk about context sometimes, I contrast EA's simulator with the kind that the Air Force would use. I point out that good testing for the Boeing flight simulator would be very different from good testing for the Chuck Yeager simulator. The response I often get back is, of course, that it would have been fine for the game flight simulator to be of lower quality, so we can use less rigorous approaches. But they miss the point. It's not that the entertainment Flight Simulator is of a lower quality -- it's of a *different* quality.

In a flight simulator game, it doesn't matter if the cockpit is shown perfectly accurately. What matters is that somebody who has never flown an airplane can have fun dealing with a very complex virtual instrument. And if they can experience some of the thrill of flying without having to go through pilot training, then you have a game that might be not only commercially successful, but also entertaining in the best sense of the word. Boeing doesn't have to worry about making their simulator fun. Instead, they have to make their simulator absolutely realistic and structure it so that it will operate properly under all sorts of circumstances that test pilots are going to face. A kid crashing the game flight simulator has a very different emotional experience from a pilot crashing a training flight simulator. We don't have to worry as much about game players crashing; in fact, for some folks that's fun. We *do* have to worry about the screen being absolutely predictable and grouping the game controls in ways that novices will find appropriate. So the quality standards we used to create the game simulator at Electronic Arts were not necessarily higher or lower than the quality standards at Boeing, but the quality criteria -- playability, entertainment value, educational value -- are very different from the criteria for an Air Force flight simulator -- which are based on getting someone ready to fly a plane accurately and skillfully.

So the test techniques that you're going to use for the two flight simulators will really be very different, and at the end you will have two incredibly different products. The testers of one product might not be in any way competent to test the other product, but both products might still be absolutely successful and well respected. That's one of the best illustrations of context-driven testing that I can think of.

SG: You talk about this kind of spread in the Principles of Software Testing for Testers course as well. If an organization is like those you mentioned earlier - married to one kind of testing style but really should be using others too -- what does it take to get them to see the benefit?

CK: Generally, they have to notice that they are missing problems or spending too much finding the problems that they do find. Often it takes a crisis, like costly recalls that are visible to senior management. Sometimes the frustration comes from slow response time. If your product is really taking off and getting used under an increasingly wide range of circumstances out in the field, customers will encounter problems under the new usage conditions. If your testing style requires a long time to develop and document tests, you won't be able to keep up with all the problems and their fixes. Some test groups notice that their programmers and tech support staff catch the problems before they become disasters. But when you see problems caught by programmers or customer support that should have been caught by testers, you know it's only a matter of time before some problems will be missed by two or three levels of folks, and you're at great risk of serious failures or recalls. That's when people start thinking, Hmmm, maybe we need to do our testing a little differently than we've been doing it.

SG: The course that you helped Rational with, of course, covers all of the approaches. Would you say that the most important take-away from the course is the ability to appreciate new approaches to testing and take home ideas for new ones you can try?

CK: Actually, I think the Rational course offers several valuable things. Certainly laying out several different test techniques should be of value to anyone who takes the course. We also spend a lot of time on communication, on problem reporting.

I think that problem reporting is among the most fundamental skills that testers can have, yet it's among the least well-practiced. Rational publishes an excellent problem tracking tool, but if someone doesn't know what to write, then a well-structured database only helps them put stuff in that no one will read. Imagine that, when you discover a bug, someone other than you has to make an informed business decision about whether to fix it or even do follow-up research on it. Your goal, as a tester, is to give that person the best information they can have to make that decision; in some cases, that means pushing them pretty hard with data to get them to understand how serious the problem is, so they'll make a decision to go for higher quality even on a tough schedule or at a high cost. Under these circumstances, you need techniques to make the severity of the problem more clear, to make the circumstances under which the problem appears simpler to explain and imagine, and to make the presentation itself easier to read. In the Rational course, we drill these techniques, and I think that's a very important thing for testers.

I did a study at one company across six of their products, looking partly at the question of why certain bugs had escaped into the field and caused recalls. As I wandered through their bug tracking results, what struck me most was that they had many testers who were not writing really good problem reports. This came as a surprise to the company. They had such confidence that their engineers would fix problems if they understood them, that many testers felt that all they ad to do was to get a problem to the point where it was reproducible and write a description that was accurate. But too often, the descriptions were rambling, over-detailed, and not necessarily focused on the problem's effect on a customer or another stakeholder. By looking at the readability and focus of the report, I was able to predict whether a reported problem was likely to be fixed or not. I think many companies overlook very serious problems just because their tracking reports are weak. Programmers reading those reports would probably tend to turn their attention to fixing much less serious problems simply because their descriptions were easier to understand. So I think effective reporting is a fundamental skill for testers -- to be able to take what they learn through testing and communicate it very well in writing.

Coming next month: Part II of this series, with a focus on education and training for software testers.