

Design for Testability, Agile Testing, and Testing Processes: An Interview with Testing Expert Bret Pettichord

by [Sam Guckenheimer](#)

Senior Director of Technology for Automated Test
Rational Software

[Bret Pettichord](#) is an independent consultant in software testing and test automation as well as a co-author, with Cem Kaner and James Bach, of Lessons Learned in Software Testing. For the past year, he has championed principles of Agile Testing with [Brian Marick](#). His software testing philosophy is context-driven, focusing on good relations with developers and agile methods that get results with a minimum of documentation. I recently asked Bret to share with us his insights about Design for Testability, Agile Testing, and Testing Processes.



Sam Guckenheimer: You just returned from the Pacific Northwest Software Quality Conference (PNSQC), where you gave a talk on your "Design for Testability" paper. What do you see as the main idea behind that concept?

Bret Pettichord: As you know, I've been talking about design for testability for a while, and I finally decided to write down my thoughts in that paper, which started as a catalogue of different things people have done in order to make testability work. I was motivated by the big disconnect I've often found between testers and developers on this issue. Although I've usually been able to bridge the gap in my consulting work and as an employee with different companies, I wasn't really sure I could train other people in how to do that.

I realized that developers and testers had different perspectives. The testers were saying, "I want testability in the code," and the developers

[▶ subscribe](#)[▶ contact us](#)[▶ submit an article](#)[▶ rational.com](#)[▶ issue contents](#)[▶ archives](#)[▶ mission statement](#)[▶ editorial staff](#)

were saying, "Well, that's too hard." I think the developers didn't realize they could meet the testers' requirements easily, and the testers often didn't recognize that some of the simpler things the developers could do would be sufficient. So there was a feeling on both sides that more movement was needed than was really necessary. What I did was scour the literature, look back on my own experience and that of others, and put together a long list, case by case, of what people had done to make products more testable, what had worked for various teams. I found that holding concrete discussions between testing and development teams was the best way to help them come together. The testers said, "This isn't exactly the information we need," and the developers said, "Well, we can add some more information to the logs then." So then the testers started describing the specific features they really needed instead of making nebulous requests like, "You have to make this product testable."

SG: What's your favorite example?

BP: I was working with the testing team for an embedded system with communications technology, and we were trying to figure out how to test to make sure that all the stuff would communicate. What we needed, essentially, was to simulate a request coming from one part of the system after being triggered from another part of the system. I suggested that we talk to the developers, but the test team didn't want to. So I said, "Okay, I'll go talk to them." The half dozen people on the testing team all followed me to the lead developer's cube, and I told her, "This is what we'd like to do." And she replied, "Oh, you know, we were going to do something like that, and we've done 90 percent of the work already. So we'll just finish up that work and put it in." This was a classic case; the testers were so fearful that their request would be considered difficult, but with the right kind of communication the developers said, "This isn't so hard."

SG: What's behind the test team's reluctance to ask for something that is often really easy to do?

BP: A lot of people don't have enough experience to know that it's easy to do. I also think that, frankly, testers have a bad reputation working with development teams. A lot of testing teams believe it's their job to scold development rather than support them, and the developers often expect it. So there is this kind of built-in animosity that gets in the way.

A lot of times you have testers who aren't technical enough to actually describe what they want and to be able to review design documents. I think these capabilities are critical for testers. At least *some* testers need to understand the system design and architecture so they can talk meaningfully with the developers about the testability hooks in the system and where they can access them.

Here's another thing that gets in the way of a lot of testers: They feel like they've been put in a position to judge the software, and therefore they have to seem authoritative, which gets in the way of collaborating with developers on things like testability. They can't come in and say, "Well, we'd like something like this, but we're not exactly sure. What are your thoughts?" That just doesn't reinforce that presumed air of authority.

You get this a lot with people who wear the "quality assurance" label. They feel like the final inspectors, the ones who judge whether the software is good or bad, and the more seriously they take that mission, the more it hurts their ability to be helpful to the team.

SG: You just presented your paper at the conference. How did people take it? What kind of reaction did you get?

BP: I'm happy that it received a lot of interest. The best reaction I got was from someone I talked to afterward; he started brainstorming with a co-worker about adding testability into their product before the talk was even over. And that's what I was hoping to do, to provide a catalyst to make these kinds of things happen.

SG: Let's drill into the content a little bit. In the paper and in the book you co-wrote with Cem Kaner and James Bach, *Lessons Learned in Software Testing*, you talk about design for testability as a matter of visibility and control. What do you mean by those things?

BP: I think there are two difficult things about software. One is that software is practically invisible. When it operates, we see very little sign of what is actually happening, how it got there, what went on. In classic black box testing, all we see are the inputs and the outputs, and from that data we have to somehow derive correctness. Essentially, testability isn't compatible with a black-box way of thinking. It asks, how do we get into the system and see what's happening inside? And, given the nature of software, the only way that you can see the inside is by having some kind of facility that lets you do that.

The second part is control. With the standard inputs you have to the system, it's difficult to exercise all the different code portions, especially if you are looking at things like exception paths and opportunities for test automation. Testers need different ways to operate the system, to be able to send it inputs and put it into different states to test how it handles all of the software's possible states.

SG: Does that apply to GUI testing, or is it better applied at the API level?

BP: Several years ago, I found that in order to make GUI test automation work, you need some testability in the GUI layer itself. And as I developed techniques for GUI testing, I began to realize that if we're going to modify the system to improve testability, it might be more efficacious to do that at some level other than the GUI. So the question has become, when do we modify the GUI to make it more testable? When do we find other ways to test the product besides using the GUI?

I think that one of the biggest problems with test automation is that people automate through GUI interfaces, which are fundamentally unstable. They have a design flexibility that is somewhat necessary to the success of the software. So you have this classic maintenance problem of how to update the test and how to keep things going. I think the right solution is to do more API-based testing with system interfaces at an API level that allows you to get into the core product. This has to happen before we're really going to make big progress on the test automation

front.

SG: Do you find that most testers you speak with understand the value of API over GUI testing?

BP: It varies quite a bit. A large section of the testing community is focused on the GUI testing approach. Many people have invested a lot in learning the tools and techniques for making that work, so they're reluctant to adopt a totally different view. In other cases, especially if the testers have tried API testing, they see the advantages.

SG: Last month we talked to Brian Marick about test-first design. He remarked that if you practice test-first design, then the need to design for testability is obvious, and there's no contention -- it just happens. Do you agree with that? Do you see test-first design as necessary for achieving design-for-testability?

BP: I don't think it's absolutely necessary, although I do agree with Brian that it helps a lot when you have people doing test-first design. Many development organizations tell me that because they're using test-first design, they often design products differently. They divide up the functionality in different ways to make each piece more similar and more testable in isolation. So I think that it does a lot to help testability on the small scale.

But Brian and I have different perspectives from the standpoint of scale. Brian looks at testing from the developer's perspective, which considers smaller scale testing. I work a lot more with systems and larger scale testing problems, so I think it's important to consider not only design for testability, but also an architecture for testability. So my questions are: How do we conduct end-to-end testing on the system? What kind of testability hooks do we have in place, and what are we missing?

Of course, we have less experience doing test-first development from a system perspective, so whatever we say about that is going to be a little bit more speculative.

SG: You described how the concept of testability can open that up for better coverage of error paths and so forth. Have you tracked this with code coverage as well?

BP: I don't focus a lot on measuring the code coverage. Most of the teams I work with really aren't at a point where they can appreciate these distinctions, because they are just trying to get the basic things tested, get their routines automated. But yes, there are times when error handling is not just a unit level issue. Certain types of errors are really systems errors, and they are often handled in a complicated way between different components. For instance, consider how a system handles running out of system memory. If you actually try testing these things, and you find your exception handlers themselves start allocating memory, then you can't test because you're out of memory. So you can run into pretty serious problems. In these cases, you could test the individual components, and they would look as if they're okay -- until you realized

that your assumptions don't quite hold when you chain all the components together.

SG: I'm interested in the key role you played at another conference, ChiliPLoP 2001, which was held last February in Arizona, where you were one of the organizers of the workshop on patterns for GUI test automation. Can you fill us in on what happened there?

BP: I had been thinking about patterns as ways of describing solutions or common techniques, and I had put together a set of patterns for test automation. At the conference, Brian Marick, Bob Hamner, Nancy Landau, and I spent a couple of days going over some of the notes I'd been making. Brian's perspective was grounded in the work of Richard Gabriel, who views patterns as more than simply a grab bag of solutions but rather as a set of interlocking solutions with a common set of values. Looking at my work, Brian suggested that we explore the underlying values and principles at stake in the patterns I had defined. We basically were able to divide them into two categories. Some were based on what I call the "silo principle" -- testing works in a separate space from development, and there are a number of solutions for making test automation work. Other patterns were derived from a collaborative relationship where there is no silo, where you have a lot of crossing over between development and testing. Basically, we agreed that the silo patterns revealed limitations, whereas the collaborative patterns were where we wanted to focus. We needed to pay less attention to automating the software as it's given to us, and instead to take advantage of whatever interfaces we have and build up whatever apparatus is necessary to make that kind of testing effective.

In the traditional silo approach to GUI test automation, you try to figure out how to test through the GUI. Typically, you build up several abstraction layers to insulate your tests from expected changes in the user interface. But that takes extra engineering time, and as the software changes, you have to maintain your separate automation system in parallel with the software test. So we pushed our discussion in this other, collaborative direction -- not only collaborating with the developers, but also more specifically working with the Agile development community on ways to adapt testing. Essentially, our Agile testing project got started at that workshop.

SG: Now you've worked with Brian to articulate principles for Agile testing.

BP: Right. For a while the two of us were working with a few people, trying to define an intersection between our school of software testing and the Agile development community. It was largely speculative, and it's an ongoing project. Right now, we've boiled the principles down to four key ideas. One is conversational test creation, which I think is a powerful idea, because defining the test includes lots of people. I've done test-first stuff where the testers went off by themselves in a separate environment and defined tons of tests. They waited for the software to come through the gate and then start executing the tests. What was missing was conversation -- sharing tests with other people and discussing tests early in the project. Conversational test creation gives the test visibility, and there are some interesting implications. For instance, it means you can't

have all of your tests coded up in a scripted fashion, because you want them to be written in a form that lots of people can examine and use as a basis for conversation.

Another idea is what I call coaching tests, and what Brian calls rapid feedback guidance tests. These tests can direct the project and set goals. This is a form of test-first development, but it also illustrates that tests can be a form of requirements instead of a form of finding mistakes. That is a big shift in thinking: How do we encapsulate requirements within tests, instead of looking at testing as a bug finding activity?

SG: One of the phrases starting to appear more frequently -- I think Martin Fowler coined it, and Brian used it in last month's *Edge* interview -- is *specification-by-example*.

BP: Yes, that's a great term. It means doing requirements by example on the system level. For many, many years, we've had difficulty doing good requirements analysis and finding a way of specifying requirements clearly. I think it's actually harder to specify requirements than it is to design a system. The problem is, we're actually working at a higher level of generality -- the design we use is just one possible solution that meets the problems essentially defined by the requirement space. So doing requirements well requires this very complicated definition that could have multiple implementations and multiple solutions.

We're asking, what are the things that the system has to do? There has been a lot of talk about requirements having to be testable. We're trying to go one step further by saying that requirements have to *be* the tests. I think this is really a powerful notion, because requirements had been the hardest part of the Agile testing agenda for me to embrace, and I now think it's one of the strongest parts.

Here's what we're saying: Let's just use an example of something the system has to do. Let's not focus on defining the whole space, but rather on points in that requirements space. And that's what these coaching tests are -- they focus on one activity the test has to support.

SG: RUP expresses basically the same idea with the term *use-case realization*. It doesn't quite have the same ring as *specification by example*.

BP: Yes. I think "coaching tests" is more mnemonic, but it's the same idea. Whatever process you use to get there, you need to make sure you get specific. As I mentioned before, testers and developers have difficulty communicating. And it's worse trying to communicate at an abstract level, since different groups have different definitions for abstract terms. When we get specific -- for example, so and so must buy this product for this much money and it's going to be debited to their credit card -- then we know how the system is going to support that activity, what the results are supposed to be, and how the whole thing is going to come together.

SG: So that describes conversational test creation and coaching tests. What are the other ideas?

BP: Providing test interfaces and exploratory learning. Coaching tests are written first, as commands in a scripting language or data in a spreadsheet. They need to be executed, so some facility for executing has to be provided. Typically this is done with test interfaces and fixtures. It's a consequence of making testability a requirement.

Reading the XP literature made us think that there was something missing, that unit testing and acceptance testing wasn't enough. So we defined the idea of exploratory learning. This is basically exploratory testing, but the focus is as much on finding out new ideas for how you want to evolve the system as it is on uncovering hidden defects. And you can't really automate it. At XP Universe, we learned that most XP groups were actually doing this, but they called it "playing around." It was something that usually happened at the end of an iteration, when the developers naturally wanted to show off their baby, and people from outside the development team would play with the software and see what it could do. All we are doing is giving this activity a little more focus. We think there are ways to structure this activity to ensure that you get the most from it.

SG: How do you find the reception to these ideas among testers and developers? Or requirements analysts for that matter?

BP: It depends on what communities you're talking to. I recently gave a talk based on these ideas at the SoftPro bookstore near Boston, which is run by an old colleague of mine. At the end of my talk, he shook his head and said to his brother, "We've been doing this stuff for twenty years." So, to a certain degree, none of this is new. What is groundbreaking, though, is that people are talking about it and writing about it and treating it as good methodology. Before, although people were doing the same things, they were so busy creating products that they weren't writing or talking publicly about them.

I have been quite surprised at the welcome reception we've had from developers regarding many of these ideas, especially those in the Agile and Extreme Programming communities. We've been coming in not only with these ideas, but also with some sharp criticisms of what XP theory misses regarding testing. We expected resistance, but instead, this community has embraced what we're saying and encouraged us to talk more about these things.

SG: Why did you expect resistance on these points from XP developers?

BP: Some of the elements of XP methodology contradict our views, and we had met people who regarded most aspects of XP as dogma --for example, the principle that you have to have 100 percent test automation. Martin Fowler had said that the use of manual testing on a team indicated a problem. But we've found great value in manual testing and exploratory testing; you just need to understand how to use it and when to use it effectively. And after we talked to Martin about how manual testing fits into the whole project, he completely rethought that position.

SG: I recall a workshop at the Software Test Automation Conference in

which some people with testing backgrounds seemed quite threatened by the idea of Agile Testing. I'm wondering whether that has to do with differences among industries, whether they were used to a highly regulated environment, or whether that comes from an innate resistance to change.

BP: Well, regarding agility, some testers look at what's going on and say, this is cowboy programming; we've seen this all before. It's just going to get us into trouble, and there is no reason we should be corroborating it. In fact, because of these objections, I've recently put together a list of four schools of software testing.

First school is the "comprehensive" school, which promotes a set of techniques for thoroughly testing software. I think most techniques for coverage measurement have come from this school, and I think of Boris Beizer as one of the leaders. Academia embraces this school, because mathematical techniques can be employed in comprehensive testing. And the telecommunications industry often needs it because they have some of the highest software reliability requirements.

People in this community look at Agile Development and test-first development and say, "You call that testing? That's not testing." It doesn't match their quantification requirements, and it doesn't use any of the techniques developed in schools for comprehensive testing coverage.

The second school I call the "factory" school. These are the people who believe testing is an item on a checklist that needs to get done. They view their teams as groups with limited skill and ability, working hard to check off all the tasks on the list. The factory school is very management focused and centered on logistics: "Have we checked off every requirement? Have we done the minimum necessary in order to say that we've performed a complete testing job?"

The ways in which we encourage collaboration, and the extent to which the boundaries of trust between testers and developers are open to negotiation is very worrisome to people of this community, because they want, more than anything else, to have a manageable activity.

SG: Would you say the BCS (British Computing Society) material is an example of that?

BP: Yes. I think that the BCS or the ISEB (Information Systems Examinations Board) curriculum fits into that type of approach. It is focused on contract software development and IT software development, and it's often appropriate for certain types of developers who simply don't want to re-think the testing agenda.

So the third school I call the "QA police." These are the process-focused people, who look at testing as a cudgel to beat developers into following some proper process.

When I put together my ideas on the four schools of software testing, I tried to find a way to give each of them their share of credit, but I had a hard time finding much credit to give this school of thinking. The QA police

work in organizations that believe it's important for testers and QA people to scold the programmers for not having done things right. For them, agile development is just making it up as you go along -- a form of general madness -- and they have no tolerance whatsoever for the ideas we've been discussing.

The last school of software testing is the "context-driven" school of testing, which, as you know, I belong to, along with Brian Marick, Cem Kaner, James Bach, and many others. We're the people trying to figure out how to pull this all together and how we can get testers to work effectively with Agile Development.¹

SG: Several times during this interview you've referred to a distrust or other forms of boundaries between testers and developers. What is your view of the project manager's role in assessing and understanding the place of testing, and in bridging some of those cultural gaps, especially from the context-driven point of view?

BP: I think testing brings essential information to the project manager. If he or she is not getting useful information from the testing team about the project -- how things are progressing, what's working and what's not -- then the testing team is failing. The other schools of testing focus on the testing group's quality assurance role, but this concern really properly belong to the project manager. That's the person who has to make trade-offs between various activities and considerations for the project: Do you want to spend more time fixing bugs or more time doing design? Where do you put project resources in order to get a successful result? The testing team should play a central role in helping the project manager make those calls.

Now, if testing team members refuse to cooperate with developers, or say, "We won't do any testing until we get a complete specification," they simply should be fired. That's an unacceptable attitude for a tester. One reason we wrote *Lessons Learned in Software Testing* was to get that message to project managers -- to let them know that there were some reasonable expectations they could have of testing teams -- expectations that, unfortunately, a lot of testing teams hadn't been meeting.

Although the book presents itself as a work for software testers and tells the reader how to do a better job of software testing, we did specifically intend that it be used by project managers as well. There are a lot of lessons there that we want project managers to overhear us telling the tester, so if the tester doesn't get the message, then the project manager will.

SG: If you roll the calendar forward ten years, what will be different about software testing practice?

BP: I think a large part of the community will not have changed at all; I expect to see a lot of inertia. But I do think the trend toward outsourcing is going to affect the way testing is conducted. A lot of development projects are being moved overseas, and companies are having trouble outsourcing the testing. I was just talking to someone recently about a

project that sent the development work to India; when the issue of testing arose, there was uncertainty about what needed to be done and how to get it done. It's hard to talk realistically about improving collaboration between testers and developers if the developers are half a world away and the testers are still here.

So much of the communication required isn't simply between people who call themselves developers and people who call themselves testers; we need communication between people who are in a position to understand the customer's perspective and the people who are building the system. If the systems are going to be used here but are being built in other areas of the world -- whether that's Russia, China, or India -- I think we're going to have a bigger gap in customer understanding.

SG: Do you see that context-driven testing will still be as controversial, or do you think acceptance for it will grow as time goes on?

BP: As acceptance grows for the principles of test-first design, I think really good developers will be thinking more about testing ten years from now than they are right now. They're going to be building testability into the product; they're going to be thinking about how to automate tests; they're going to understand that key testing problems are in fact key design problems, and that there's not a clear barrier between the two things.

One of the reasons developers are getting more interested in testing is that they can no longer pretend to understand how an entire system works, and they can't model the whole thing in their own heads. As they use more and more third-party components, as they use more tools to build the components that they're creating themselves, they need to understand the software that they're using -- and that means testing it.

Sometimes when I talk to testers, I mention that we do not need to know the correct result before we run a test. And yet, I've found that testers can be very interested in test results. Developers, too, are often very curious to know the actual performance characteristics and what limits there are. When this information is presented factually and not under the guise of blame, developers have a lot of interest in it. And that is really what software testing is: a process of experimenting with software and finding out what it does. I think that it is going to become more and more interesting to developers; as systems get larger and more complex, they need ways to figure out what is it they're actually dealing with.

Notes

¹ For more on context-driven testing, see the [interview with Cem Kaner](#) in the July issue of *The Rational Edge*.

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

Copyright [Rational Software 2002](#) | [Privacy/Legal Information](#)