



Hanselminutes

Hanselminutes is a weekly audio talk show with noted web developer and technologist Scott Hanselman and hosted by Carl Franklin. Scott discusses utilities and tools, gives practical how-to advice, and discusses ASP.NET or Windows issues and workarounds.

## Text transcript of show #103

March 7, 2008

### Quetzal Bradley on Testing after Unit Tests

In this episode Scott talks with Quetzal Bradley, a Microsoft developer on the Connected Systems Architecture Team, about testing after unit tests. Is 100% Code Coverage enough?

(Transcription services provided by [PWOP Productions](#))



*Our Sponsors*

 **telerik**  
*deliver more than expected*  
<http://www.telerik.com>

 **software**  
<http://www.nsoftware.com>

**NET**   
**DEVELOPER'S JOURNAL**  
<http://dotnet.sys-con.com>





**Lawrence Ryan:** From [hanselminutes.com](http://hanselminutes.com), it's Hanselminutes, a weekly discussion with web developer and technologist, Scott Hanselman, hosted by Carl Franklin. This is Lawrence Ryan, announcing show #103, recorded live Wednesday, March 5, 2008. Support for Hanselminutes is provided by Telerik RadControls, the most comprehensive suite of components for Windows Forms and ASP.NET web applications, online at [www.telerik.com](http://www.telerik.com). Support is also provided by .NET Developers Journal, the world's leading .NET developer magazine, online at [www.sys-com.com](http://www.sys-com.com). In this episode, Scott talks with Quetzal Bradley, a Microsoft developer on the Connected Systems Architecture team about testing after unit tests.

**Scott Hanselman:** Hi, this is Scott Hanselman and this is another episode of Hanselminutes and I'm sitting here in Building 42 on the Microsoft campus and I'm sitting down with Quetzal Bradley, a developer on the Connected Systems Architecture Team.

**Quetzal Bradley:** Hello.

**Scott Hanselman:** Thanks for taking the time. I was given your name by Chris Sells who just raved about some of the stuff that you were doing internally and said that you gave a presentation on testing after unit tests and I saw this and immediately said, "Okay, I've got to come and hang out with this guy." So, now that I'm up visiting in Redmond, I'm happy to be able to sit down and chat with you. Now, I've worked on systems that have done testing and unit testing and code coverage, so I thought I pretty much had a handle on that, but you've got some ideas that indicate that maybe I'm not as prepared as I thought I was.

**Quetzal Bradley:** Well, yes. I guess most people are familiar with unit testing and that common metric when you are writing unit test to know that you have sufficient test is code coverage. Oftentimes, the goal will be 80% code coverage that says that your test hit 80% of the lines of code essentially in your product and ideally of course, you would hope to get 100%, but if you have 100% code coverage, does this mean that you're done testing, you can ship your product to your customers and your customers will be delighted? I think the answer typically is no. So, then, the question becomes when are you done testing and how much testing do you need to do and how do you know what test you should do and whether they are sufficient.

**Scott Hanselman:** So, if I had a method that took a Boolean and I have an if statement inside it, if this, else that, I've got two branches and I might run a complexity analysis and do a cyclomatic complexity and it would tell me that there are two ways to get through that method. I then maybe write two tests.

I'd say that I had 100% code coverage and I'm feeling at this point pretty good about my code.

**Quetzal Bradley:** Right, but maybe you shouldn't be. One of the things is that code coverage can't measure the coverage of code that's not there and so, for example, your function may be missing error checking. If you pass an old parameter to there and that is the reference, you're going to get a crash. Your 100% code coverage won't detect that kind of condition. Even if you do have all of your error checking and all the code is in fact present and you still have 100% code coverage, the data itself is what is going to determine the behavior of the function and that isn't measured by code coverage, but code coverage does tell you something really important. Code coverage tells you what code you have not tested. It's a negative metric but we tend to think of it as a positive metric. So, if you have code coverage, it tells you that it is possible for that code to execute correctly at least in your test, but it doesn't mean it's going to work correctly for a customer. On the other hand, if there's a bit of code that has no code coverage, you know that you haven't tested it and so you know that you don't know whether it works or not.

**Scott Hanselman:** Okay, so you just say this again to make sure that this got through -- because I'm drilling this into my own brain at the same time as we're drilling it into the listeners.

**Quetzal Bradley:** Yes.

**Scott Hanselman:** By doing code coverage...

**Quetzal Bradley:** Yes.

**Scott Hanselman:** I determine that it is possible for that code that I've covered to run properly.

**Quetzal Bradley:** It's possible.

**Scott Hanselman:** It's not that it ran properly, it ran properly in this instance.

**Quetzal Bradley:** Right.

**Scott Hanselman:** But there are other parallel universes that very, very likely will not work.

**Quetzal Bradley:** Well, yeah. I think I can give another example, maybe an analogy that would make it easier to understand, which is suppose that you are going to test the roads system in the state, right? So, you know about all the roads, there are little roads, there are driveways, there are freeways, there are highways, and you take your little 3-cylinder hatchback and you drive it down every single road.

**Scott Hanselman:** Take a small car, drive everywhere.



**Quetzal Bradley:** Yes, absolutely, and you may find that you're successful or maybe you run into a power line over the road and you know you need to remove that or maybe there's a bridge that hasn't been built yet. So, you discover those things, you fix them, you've got 100% road coverage and you feel pretty satisfied and then, say, you open up the roads and you let people go and drive on them and maybe the first thing that happens is that a double-decker bus gets onto the freeway and runs into an overpass, crashes into it. I think that it wasn't really sufficient to test every road. You also needed to know what kind of vehicles were going to be traveling on them. Maybe your 3-cylinder vehicle was always going slow and when an 8-cylinder Jaguar turns the corner, it goes flying off the road because it's going 100 miles an hour and the speed limit sign was 120 maybe even. So, if we wanted to test the highway system very effectively and maybe that we do want to send a slow car and a small vehicle down every single road to find the roads that aren't finished, etc., but the way to get the most effective results would be to take a large variety of vehicles with different capabilities and sizes and shapes and speeds and perhaps drivers and send them down every road, but the problem then is it may be that we don't have the time or the resources to send so many vehicles down every single road in the state. You could imagine it might even take years to complete, so we need to prioritize.

**Scott Hanselman:** This is about mapping requirements to testing.

**Quetzal Bradley:** Absolutely. So, not every road is equally important. The freeway is used by much more people than my driveway and so a flaw in my driveway is going to inconvenience me and my immediate neighbors, a flaw on the freeway is going to cripple the transportation system for the entire state. A similar thing comes up in software although we might not think about it. I had this experience with a piece of code that I was writing some test for. It didn't have any test yet, so I had 0% code coverage and I wrote one test case, not even a test but just a test case and then I measured the code coverage and I had 37% or something like that and I was pretty pleased. That's a lot of code covered from one single test, right?

**Scott Hanselman:** Was this a small bit of code?

**Quetzal Bradley:** No, it wasn't that, but it's with the driver actually.

**Scott Hanselman:** Okay. So, you went through a lot -- you bounced around inside that driver for quite a while by that one test case.

**Quetzal Bradley:** Absolutely. I sent an IO request to a driver and it was handled by the entry point and then it was passed to another function, another function, it was, you know, the error checking

was invoked, etc., and it went all the way down to the bottom and it came back up. It actually hit quite a bit of code. I then wrote a second test case. In the second test case, now my code coverage is like 45%, so great. A third test case, my code coverage was like 50%, a fourth test case it was 51%.

**Scott Hanselman:** Kind of asymptotically approaching.

**Quetzal Bradley:** Exactly. And then 51.5%, so getting to 50% was incredibly easy; but getting to 70% is very hard. The part that really surprised me is that I ran my second test case by itself, measured the code coverage and it was about 37% or 38%.

**Scott Hanselman:** Interesting.

**Quetzal Bradley:** In other words, every test case was hitting a whole bunch of the code, exactly the same code every time, and then there was a little bit of sort of the side paths.

**Scott Hanselman:** So, you've got a lot of, you know, imagining this Venn diagram of all the code that you're covering and I'm seeing huge chunks of intersection.

**Quetzal Bradley:** Yes.

**Scott Hanselman:** And then you're getting little bits that are specific to a test case that are letting you get that extra percent or two.

**Quetzal Bradley:** Sure. I think I'd go back to the highway analogy and there's part of your code that's like the freeway and if you're going to go anywhere, you pretty much have to get on the freeway, get close to your destination, then you get off and go on some side roads. If you have a different destination, say, a different test case, you're still going to get on the freeway and drive down that freeway road, but then you'll take a different exit and there will be a little bit that is different.

**Scott Hanselman:** Right, right.

**Quetzal Bradley:** So, that freeway is the easiest code to hit, but it's also the most important code because any flaw there is going to be impactful to your customers. There is some little corner of the code that takes a real special test case to get to it. It may be that very few people would be impacted by flaws there, so even if you do miss it, it's going to have the least cost. So, if you have a set amount of time in order to do your testing, now absolutely you'd like to have breadth coverage which is make sure that all the parts of your code can function, but when you're putting the extra effort in, you would want to have a more diversity of data and more scenarios around that highway, which is the most impactful.



**Scott Hanselman:** So, how do you figure that out? I mean is that about using a profiler and then spending time seeing where we're heading through or basically running a test case with coverage and using coverage not as a number to make you feel better about your testing, but rather as a number to know where in your code you're exploring.

**Quetzal Bradley:** Yeah. You can use the coverage to know in terms of breadth what parts of the code you haven't touched yet, but it doesn't give you any information about depth. Now, about knowing where the highway is, well, the answer is, however your customers are going to use the code, that's the important code and that's the important path. So, this is where it does come back to requirements and so it is a little bit harder because sometimes people use the code in ways or that we don't necessarily anticipate, but to the degree that we can emulate the customers when we're testing, that's where we can go beyond unit testing and create tests that are going to be more effective. So, for example, I've worked on a lot of frameworks and with frameworks, customers are developers. The developer is going to write a program and he's going to use a framework. An excellent, excellent way to test this is to create a sample. A sample is the process of writing a program exactly like the customer is going to do it and you run into issues like usability or awkwardness or discoverability, that's kind of nice, but in addition, when your sample is done, you should execute it as part of your test passes because any flaws that are uncovered as a result of running the sample are very likely to be exactly the flaws that the customers are likely to run in first, in other words, those most serious day 1 kind of bugs.

**Scott Hanselman:** It seems obvious that if you're writing a framework, you need to use it. I've written frameworks before and I've worked at companies who have written frameworks and some of the early versions where basically things went horribly wrong or when we didn't come at it from the outside in, we didn't put ourselves in the shoes of the developer who would be using it, so we didn't write samples and more importantly, our test cases didn't look like samples. They didn't reflect the reality of the user of that framework so I can definitely relate to having code coverage numbers that look right, but in reality I wasn't protected.

**Quetzal Bradley:** Yes. So, again, the unit tests don't look like samples, but they are important for the purpose because they give you that breadth. To get to 80% code coverage with samples would be really difficult because, again, like each additional sample which is costly is going to give you additional point, 1% maybe. So, you have your unit test with the breadth, then you create samples for the depth. The samples aren't going to go on every byway. They're going to be going down that main highway of the code. Now, if you're not creating a framework, there

are still approaches that you can take and the general mentality is around the same code I'm executing over and over again with variations of course, but mostly the same code, but has the diversity of data, that's what customers bring to software that we don't have when we're executing on other developer's machines is they have their own data. It's bigger, it's stranger, it's less consistent. They're creating ways we didn't anticipate. Their environments aren't exactly the same. Our developer machines can be topnotch, maybe they are short on memory, on RAM, on their computer and that can have an impact. So, it's all those differences that make it likely they will run into issues that we don't know about in advance. So, to create a really interesting test, maybe it's a very simple test, but maybe it has a huge variety of data that you can put into it. You can write tests that generate data on the fly or maybe you have a pile of it lying around.

**Scott Hanselman:** So, it seems like when I'm thinking about in terms of generating data, there's generating data from the customer where you can basically make your application phone home, kind of the Windows error recovery Dr. Watson way of doing things. That's free data. If someone's going to hit Send Information, that's good stuff. We can use that.

**Quetzal Bradley:** Absolutely, you should use that and certainly if you have such data, you should use it to find out what is the kind of problems that are occurring and find out why your testing isn't discovering that and that will inform about what kind of testing you need that you might not have. One way to get data for putting in your test cases is depending on your relationship with your customers is maybe they can give it to you. If they have a share full of input, files have accumulated, take that and use that. That's going to be the best kind of data you could have because it really is from customers.

**Scott Hanselman:** Now, you would give an example to me or you'll walk in the hallway or you're in a situation where you had had some faults that had occurred and some software that you'd been involved in that was indicating there was a problem basically, something went wrong and something phoned home...

**Quetzal Bradley:** Right.

**Scott Hanselman:** So you went to the code to find out what was going on.

**Quetzal Bradley:** Right. Well, this was an investigation of, just like I said, an investigation of what kind of flaws were being discovered by customers impacting them that we didn't discover in testing. This particular component was not well tested internally. It had 50% code coverage which doesn't mean it was 50% tested; it means it was 50% untested. So, we would expect that the problems



would be lurking in that untested code. I mean isn't that what you would expect?

**Scott Hanselman:** That's what you said earlier, that really has changed my perspective is that code coverage is an inverse number. It's something that I want to say. I know what's -- well, actually this is funny. There was a movie with Hugh Grant where they had a baby and it was unexpected baby and he said, "Well, I don't know what happened. The protection I was using was 99% effective." The answer was, "Well, that makes it 1% completely ineffective."

**Quetzal Bradley:** Right, exactly. Yes. Personally, I expected to find the flaws in the uncovered code because we haven't tested it at all internally, at least not formally with the formal unit test that got measured by code coverage. So, in actual fact of the three parts of the code that were being hit by customers with crashes causing a serious problem, all three of them were in the covered code in the covered blocks and so now I think I have an understanding about why that would be, that 50% that was covered by test was the highway. It was the highway through the code that everybody hits. Naturally, that's where the flaws were found that were hit by customers. It may very well be that there were flaws in the uncovered code that we didn't cover on testing and as it turns out, the same code that the customers weren't running and so they weren't discovering those flaws.

**Scott Hanselman:** What kind of metrics can I apply? It sounds like you're shaking my confidence in code coverage, that it's a number that I could use for confidence, I mean, now it's just code coverage has become just another data point that I can use. I feel like I need another dimension.

**Quetzal Bradley:** Yes. Well, in a perfect world, you would use state coverage and state coverage is all of the states that my software can be in. Unfortunately, state coverage is typically infinite, but please don't despair because...

**Scott Hanselman:** I'm sorry, it's too late. I've already begun to despair.

**Quetzal Bradley:** Sorry about that. Not all the states are equally important, so again the states that the customers are going to get their software into are the most important ones. I think that there are efforts maybe in research and in other places where people are trying to tackle this problem; but practically speaking, in everyday that they like it here, we have to do something more practical and what that really comes down to is getting feedback and iteration, so once you have your high unit test coverage, code coverage has heights, you know that there's not much code that you're never looking at it at all. That's a good starting point. Then you want to do samples or

customer oriented testing, stress testing, especially with the functional testing a diversity of data that is likely to cover the things that you haven't even thought about whether they are data generators or data from the customers or approaches like that. Once you've done that, you should think to yourself, well, at least I've covered the basics now. There's some hope that this is pretty good quality. That's where you need feedback and the feedback is get people to try it, it's looking at what is your bug find rate from your internal deployment or from manual testing. One of the things that can be very effective is directed at hard testing where you take people and you say, "Here is an area to look at. Just try it out as if you were a customer and see what kinds of things you run into." A lot of times, that will uncover things that your test automation just never even thinks about.

**Scott Hanselman:** We had a podcast a couple of episodes ago where we spoke with Peli de Halleux from Microsoft Research and his pet project which uses some kind of mathematical model link to look at a method and say, here is the domain in which I want to explore. So it takes code coverage almost like chess, you put your finger on the chess piece and there are so many moves that you could make. Humans can only make so many moves, but a computer can go further and further and further down the decision tree; and hopefully we can see some of that soon from Microsoft Research, but then that's more of a brute force way of trying to stumble on these kinds of issues where understanding more of the behavior driven aspect of things, a behavior driven design will be a better way to find these highways.

**Quetzal Bradley:** Yes, that's very interesting and there are two big kinds of approaches that you can take there. Essentially, these are forms of model-based testing, and so I believe with pecks, from what I understand, it looks at your code and at first uses patterns from the code in order to run test cases through it. That is very interesting. It does suffer from the flaw, again, that if the code itself is missing functionality or is wrong, its inferences will be based on what has been written rather than what was intended. So that could still be a very valuable way of giving effective code coverage for your breadth testing, I believe, and other potential as well, especially as we can add more metadata about our intentions that a tool like that could use. Another approach is where you have your code but you create a model completely separately which is based on my way of saying, this is what I intended." Now obviously, that model has to be simpler than the code itself, otherwise, you've just written the program twice, once in the modeling language and once in the actual implementation. I have seen this happen where people have gone to the lengths of trying to create a model which completely represented their entire functionality of the software, and that tends to not be



very successful; but where I have seen tremendous success with the model-based testing is creating a model that's very simple because it's only focusing on one aspect of the functionality that is expected, and so for example, in a framework, maybe the object timeline of your framework objects. If you have complicated rules around when they can be destroyed, when they can be cleaned up, and the relationship between parent and child objects; you create a model just around that and get a lot of value without trying to create something that is so complicated you can't even understand the results of what it is creating. The thing that models give you which are really great is they give you a test oracle. A test oracle is the code that knows when a test runs, whether it's right or wrong -- the result that you've got. That is actually the hardest part of creating any kind of test. Writing code to execute into your software, pretty straightforward; but then, well, it did the test pass or fail. In the simplest, naïve case, you say, "Well, I didn't throw in any exceptions. It must have been correct."

**Scott Hanselman:** What is my assertion?

**Quetzal Bradley:** Yeah, exactly.

**Scott Hanselman:** One of the things that a boss of mine really draw into my head at my last job, was he said if the system gets -- and we're speaking about .NET code here -- if the system gets into a state that it shouldn't be, this is not the time to be throwing in an exception. He says `debug.assert`. He says stop, panic, freak out because `debug.assert` is just like the most underused thing he could think of in the framework because there are exceptional cases that time when data coming in should throw an exception and you should handle that occasionally, but if there is a state that a system can get in that it has absolutely no business getting into like this should never happen, you know it, I know it, the guy who is working with you on the program knows it, but if you don't say it to the code, `debug.assert` that X is null. If X is never supposed to be null, then do that; and he says nothing will make someone fix a bug like a big `debug.assert` in the middle of the running of an application.

**Quetzal Bradley:** Well, that's a fact, absolutely.

**Scott Hanselman:** It seems to me, I'm kind of put this in a context of various smaller ISV or maybe some of our listeners may have small kind of two pizza teams...

**Quetzal Bradley:** Sure.

**Scott Hanselman:** Smaller teams that would -- I like to say two pizza teams, that's what Amazon says, the size of a project should always be the number of people that you can feed comfortably with two pizzas...

**Quetzal Bradley:** Sounds reasonable.

**Scott Hanselman:** Anything bigger, that's unruly. That the best way for a smaller organization to get this data is to instrument their code to "phone home" like a tool, like smart assembly that Microsoft has Dr. Watson. Can we plug, can I make my code instrumented with Watson?

**Quetzal Bradley:** Can you make your code instrumented with Watson? I'm not 100% sure on that myself. I do know that it is possible for Microsoft to get that data on your products.

**Scott Hanselman:** So if you're blowing up majorly and you had sent information that is being held out there, I have to dig out those impressive applications, and smart assembly is the product that come to mind which is an aspect oriented thing that weaves in to your assembly after the fact and when an interception occurs, would phone home and speak to what was feature end-point.

**Quetzal Bradley:** I think that's an excellent idea because despite our best efforts and maybe that where we thought the highway was and where it actually is, aren't the same, and phoning home is the way that we're going to be able to discover that.

**Scott Hanselman:** Ultimately, what happens in reality, no matter how lofty your ideas were, no matter how much planning and putting into the model, you may think to use it this way, but the fact is, they do use it that way.

**Quetzal Bradley:** Right, exactly. I guess, one other thing I would want to say is that on these days, there a lot of focus and effort on automated testing and automation, and the thing that I really noticed about that, is that it is really a trade-off and it's not always the right thing to automate all of your tests because what happens if you focus on 100% test automation, is you don't just write the tests that are too hard to automate and then these very important things don't get covered. This is a reality of what happens. So an example is of things that are extremely hard to automate are what does your test measure on this running. If you're product has a user interface and the user interacts with it, maybe you use UI automation and press some buttons. Of course, that's a high cost to maintain kind of test to write it and also to keep it working as a UI changes, but let's say you've done that, what the test doesn't know is it doesn't see that the color is strange or that the screen flickers or maybe you have timing to make sure that when you click the button, something reasonable happens within 30 seconds but maybe the fact that it is taking one second, to the user it makes the application seems like sluggish. So that is a flaw in those kinds of automation that makes their value less, not zero value, but just less, and the cost high



because of the continual changing you have to do especially when the UI's is under frequent change. Suppose you have a manual test -- the problem is it's boring to do manual testing from a lot of people's point, but there are people actually who do enjoy that and so it would be very interesting to find some of them and see if they would like to help you out, but in any case, if you're going to do this yourself or your two pizza team and you're not going to pick someone else up, you don't have to do it a lot. If everybody sat down for thirty minutes at the end of the week and went through it that would be quite satisfactory to catch these kinds of issues, most likely.

**Scott Hanselman:** For applications that have a user interface, one of the neat ideas that I saw when office 2007 was being worked on was this little smiley face that would sit on the dialog box, it was a happy face and a sad face.

**Quetzal Bradley:** Okay.

**Scott Hanselman:** If you saw something that you like, you click the happy face, and it would take a screen shot, highlight it, annotate it and then send it directly to the team. We actually wrote a little thing up and a number of people made libraries for Win Forms that you could just add in and while you're in your testing phase someone can go at any point in any dialog and say, I like this or I don't like this, and it will do a screen shot and then you can get all these fantastic data going directly in your plugged database with pictures of real peoples' screens, doing whatever it was that they're not supposed to do. You get both negative and positive feedback and because it is presented in such a pleasant way to simply a button in the title bar with a happy face or a sad face, everyone understood it and everyone is excited to give that feedback. That's another way to collect data that I'm trying to realize is more and more important.

**Quetzal Bradley:** Absolutely. I think that's a really interesting idea. Now, suppose you've clicked that funny face a few times and so with some others and so in development, the part now is changed, that interface changes. If you're using a low cost script with manual testing to keep that area covered or when a human comes into that and sees the difference, they're not going to fall over and say the test has failed, go spend half an hour and reinvestigate to try and find out why this test case failed and then fix it. It will just go right pass, that's where the efficiency comes from. Now, of course, once you've shift and you're in maintenance mode, it may make a lot of sense to do UI automation in order to catch aggressions especially when you're not looking at it frequently and when you're occasional bug fix comes in. So the point there is that automation is not a panacea, you should try and make that trade-off. It is the time that I invest in creating this automation, don't forget to count the maintenance cost, is going to give me a return versus

doing the manual testing at this point in my project versus another project, so I'm very happy about all the automation that people are doing but sometimes I worry that people will get carried away too far.

**Scott Hanselman:** Yeah. When I think, and you're bringing up a very important point about regression testing, one just can't just remind oneself too much about how important it is when a bug comes in to write a test that causes that bug to happen and then watch it fail, I understand why this bug happened and then to fix it, that is, red-green unit testing is important but it is so much more important within the context of regression because there have been a million times where I've fixed something, I thought it was great, but just didn't backed it up. Again this is the notion of making an assertion in your head versus backing it up with an assertion in the code, it's so important.

**Quetzal Bradley:** Absolutely.

**Scott Hanselman:** Well, I really appreciate you taking the time to sit down with me today. This has been really interesting. This has been another episode of Hanselminutes. We'll see you again next week.