



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 #146

January 12, 2009

Test Driven Development is Design - The Last Word on TDD

Scott Hanselman talks to Scott Bellware about TDD. ScottB says that Test Driven Development is less about Testing and more about Design. Is TDD poorly named? Did Test Smell beget Design Smell beget Code Smell?

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



Our Sponsors

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

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

NET 
<http://dotnet.sys-con.com>





Lawrence Ryan: From 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 #146, recorded live Monday, January 12, 2009. 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, and by .NET Developers Journal, the world's leading .NET developer magazine, online at www.sys-con.com. In this episode, Scott talks test-driven development with Scott Bellware.

Scott Hanselman: Hi, this is Scott Hanselman and this is another episode of Hanselminutes. In this episode, we're sitting down with Scott Bellware, an Agile coach from Austin Texas and he proposed that we talk about Test Driven Development but specifically I want to call this talk Test Driven Development, the last word. Because Scott has really some interesting opinions about how TDD is being perceived and being preached in the community, so I thought that would be a fun discussion. Thanks, Scott, for sitting down with me today.

Scott Bellware: Yeah, thanks for having me on.

Scott Hanselman: So the last word, you know that really implies that the Test Driven Development is being presented in a way that's not entirely accurate and we want to kind of slam the door shut on that and move forward. So what is not presented appropriately with regards to TDD in the community today?

Scott Bellware: I think that it's pretty well represented fairly accurately. The problem, I think, comes in when we lead to the assumption of testing every time that we start talking about Test Driven Development or this conversation is about software testing or software quality assurance. It's almost inevitable now that someone will turn the conversation to Test Driven Development like, you know, well, let's talk about quality and then, well, what do you think about TDD and how does this apply? I think some of -- the little bit of a shocker for me, and maybe it's a shocker for the audience, is that folks who do Test Driven Development assert and have asserted for quite a long time that TDD is more about design than about testing and there's a little bit of benefit to come about talking about TDD when we're talking about testing but only as talking about the side effects, the quality side effects of TDD and it's so, I think, egregious or it has become so egregious over the years that TDD has even been somewhat renamed in recent times too from Test Driven Development to Behavior Driven Development really to draw a line in the sand and say this is not about testing, this is about something else. We've got this unfortunate heritage of TDD that has the word test in the name but it's not and we should really endeavor, I think, more of an effort to preserve the value, the intrinsic

value that TDD offers and TDD has, really give it the old college try to say, hey, every time I'm talking about testing and I feel my mind wanting to stray to TDD, maybe I should program myself and throw an exception or something like that.

Scott Hanselman: I think the name, the simple name Test Driven Development describes the mechanical action of the style of development. Yes, one does tests an aspect of the Test Driven Development but that's the physical aspect of things.

Scott Bellware: It's a bit of a gray area too because even if I say this is about design rather than testing, there is always going to be an aspect of testing in TDD.

Scott Hanselman: Right, but to write tests and have them drive the development isn't necessarily the purpose, the whole reason for existence of this style of development. I haven't felt though that Behavior Driven Development, well, it seems like it feels better to say, for me, it still doesn't quite say it right. You know how they always say an object-oriented programming needs to really name it before you can understand it. I almost feel like we haven't named this style of development in a way, at least for me, that resonates.

Scott Bellware: There's a Behavior Driven Development discussion group on Google Groups. I don't remember the address right now but I'm sure you can Google for it.

Scott Hanselman: Sure, we'll put it on the show notes.

Scott Bellware: That same thing is reflected in Behavior Driven Development, I mean not only the Behavior Driven Development change the name in order to try to resolve this issue about TDD being testing but it also is 10 years later and some of the practices have been changed and it introduces its own set of confusion as any new thing or practice does. So it's questionable whether the name actually does anything except change the name from Test Driven Development and I think that there's a great value in that. I think that it's also quite sad though that no matter how often it was brought up, or how often it was pointed out, or even requested the folks to really consider TDD as design rather than testing that it never really sank in and I find this fascinating. I think that questions and conversations about quality are somehow sometimes even damaged when we let them fall over into conversations about Test Driven Development. If you look at writing tests first as a means to achieve software quality from the perspective of a, you know, a tester's perspective or quality assurance perspective, it's going to seem pretty senseless. TDD will create quality but not the kind of quality that, you know, the word quality applies in a different way. I thought it was fascinating



listening to Carl Franklin's conversation with Dr. Whittaker and it was a pretty good episode of your brother show .NET Rocks! and I hope a lot of people get a chance to listen to Dr. Whittaker's perspective. Some of it I agreed with and some of it I thought was a little bit sci-fi and that's great, at least it was engaging and I appreciate that, but when it got to the Test Driven Development question, it was, to me internally, one of those things that, hey, why is Test Driven Development coming up in the conversation about software quality and software quality tools, and for me it's natural to think that way and I think it's natural to think that way for many folks who do Test Driven Development but it's really kind of hurting Test Driven Development and the value that it brings and I think it also hurt these conversations about quality.

Scott Hanselman: Okay. So let's do this, let's try this thought experiment. Let's back up and listen to the point of view of someone who works and we'll just use my canonical engineer, this is the guy that I call chief architect at the Nebraska Department of Forestry. He's got access machines and he is learning ASP.NET and he maybe has started out as a VB guy but maybe he doesn't really have the deep computer science passion or the deep software engineering passion that a lot of people have. He is just a guy who got a job and he enjoys it but he really wants to be a better person. This is 70% of developers out there, they want to do better but they're saddled by some issues whether they be organizational or technical or whatever. He starts a new project and someone says, "Hey, wait a second. You need to be writing a test first. You need the Test Driven Development." Then they gave him the elevator speech on Test Driven Development, and what I'm hearing you say is that he should immediately say, "Oh okay." Well, this would mean that I'll have lots of tests and I can go to the command line, I can type build up that and test out that and I'll have more confidence that my application works and then he goes on with his life and you're saying that that is completely missing the point. He is missing an opportunity here.

Scott Bellware: Well, I think he is trying to get the opportunity that Test Driven Development brings to the party that he is very possibly missing the point of Test Driven Development, that in the summary that you just presented it didn't really sound like the case of a Test Driven Development was made.

Scott Hanselman: Oh, it doesn't but that's very typical and I've lived through those experiences lots of times where someone has introduced Test Driven Development in a very basic sense.

Scott Bellware: Well, if you introduce it, I mean I would expect anybody to buck if I walk up to him and say, hey, you know, this new unit testing thing, let's do that, it's really good. That person says, yeah, I heard of this, it's a great idea, I wanted to do this. Then I

say, let's start writing our test first. There's no real sort of rationale or law tech that suggest we know that leap from doing unit testing to writing test first, and part of the reason for that is the goal here with Test Driven Development isn't really a unit testing goal although we're borrowing unit testing tools to achieve the goal and does Test Driven Development and frankly there is testing involved, I mean we're using testing, we're borrowing testing, almost like we're sort of demonically possessing unit testing to achieve other means. So I don't think I could sell Test Driven Development from the perspective of the value of unit testing.

Scott Hanselman: Okay. So how do we frame this? So someone has got some foray into testing. He has done some testing and some basic stuff, but it hasn't taken over his life, he hasn't fundamentally changed the way he designs his software.

Scott Bellware: Well, the first thing I would I want to do to frame it is ask whomever I'm working with to just give it a little bit of effort that every time they hear the word Test Driven Development and they automatically react with thinking about testing and quality assurance, that they just suspend this belief every time that moment comes up and just give it a moment of pause and consideration. That goes a long way to understanding what we're after, or to disentangle testing from the goals of TDD or BDD.

Scott Hanselman: Okay.

Scott Bellware: The next thing is to talk about what makes software great and enjoyable to work with and the basic premise is, for me, that you can't really use anything that you don't understand so if software is hard to understand, it's going to be hard to work with.

Scott Hanselman: A-huh.

Scott Bellware: So the easier that we can make software understandable, the better our experience as developers and also the higher our productivity will be. it's interesting that the qualities that make software easier to understand are also the qualities that are highly desirable from a OO Design fundamentals perspective so when you spoke to Bob Martin about the SOLID Principles in a very, very basic abstract and essential level, all of those principles point to designs that are easy for the human mind to get around.

Scott Hanselman: You know, it's funny that you mention that because I run my -- this is going to sound random but I think it make sense, I run very large fonts, I have a really big monitor, I run like 16-point font, people think that's ridiculous. Why do you run a 16-point font? I run a 16-point font, but I really get uncomfortable if I can't see the entire function without scrolling at 16-point font on my monitor which



basically adds up to about 25 lines. If it gets bigger than that, then my small mind just can't conceive what's beyond the scroll bar and I find that it makes what I do better and I'm almost like mechanically force myself to have smaller functions just because I can't possibly suck that much information in another time.

Scott Bellware: So you're pointing to an essential quality of code that's easy to understand which is it's small.

Scott Hanselman: Right.

Scott Bellware: Or there's less of it to swallow at one time.

Scott Hanselman: Right and then if you start there, it's consequently that it's easier to test and if it's really small, hopefully it has only one purpose to live.

Scott Bellware: Yeah. What was the SOLID Principle that said...?

Scott Hanselman: That a class or method should have only one reason to change.

Scott Bellware: Yeah, the Single Responsibility Principle.

Scott Hanselman: The Single Responsibility Principle.

Scott Bellware: Now all these jargon, you know Test Driven Development is also a part of the jargon. Testability is one of the worse bits of jargon. it's one of the most accurate bits of jargon we have and it's also, I think, one of the most destructive bits of jargon to the cause of software design. When I say testability or when anybody in the progressive, for a lack of a better word, software development community uses a word like testability, it means something inside that community, inside that culture, and it means something that is not understood outside of that culture. Every programmer in the world should reasonably assume that when I'm talking about testability, I'm talking about the ability to test something. So if I look at some piece of code, or if I sit down with Joe, the programmer from Nebraska Department of Forestry, and I look at his code and I say, "Hey, that test is not testable," and he says, "You're out of your mind, I'll show you the test for this. As a matter of fact, I've got high coverage on this code and this code is tested so it must be testable." I've kind of sort of shut myself in the audio's logical foot by using this word testability.

Scott Hanselman: Right.

Scott Bellware: And the assumption that we make in the inner sanctum of the echo chamber is that what we're really saying is it's got easy testability.

Scott Hanselman: Right and I think that that point that you just made, it kind of deserves repeating and paraphrasing and pounding and pounding because I think that it really clarifies things for me. I think it really nails the point. The idea that you did test something doesn't mean that it was easily testable, and I remember when I talked to a guy named Quetzal Bradley who is a tester at Microsoft who made a comment to me, I think it was a year ago when I talked to him, that just because you have 100% code coverage that doesn't tell you that your code works necessarily. It just means you executed every line. It doesn't say anything at all about intent.

Scott Bellware: Yeah.

Scott Hanselman: It's this small, little obvious observations that continually blow my mind when someone says that. You know, I had a guy in my team who was obsessed...

Scott Bellware: That should never be taken to mean by any of your listeners that code coverage is that irrelevant.

Scott Hanselman: Oh no, but I had a guy in my team who was obsessed with gaining 100% and I thought it was great. I said go. Well, if he wants to work that hard and go to 100%, good for him. It made him feel good but it was a completely false sense of security.

Scott Bellware: I think 100% code coverage is a pretty good number to tell you the truth. I think it's a really good number when that number comes out of a team that's using a handful of practices that makes that number mean something. That number on its own only mean something when you're using the kind of designs that suggest that the number is meaningful and this comes back to design again. If I get 100% coverage on code that has what I would call defect detractor design, then you're really not proving that the code doesn't have defects. You're just proving that you've touched every line, but if you had 100% coverage on code that was well-designed, it actually start to mean something more in terms of what you can expect from a defect rate.

Scott Hanselman: Now, who will be doing that?

Scott Bellware: Certain designs are more prone to defects. They attract defects than other kind of designs. The smaller and smaller, you know, and the more cohesive these units of functionality become, in accordance to all of these crazy grandiose terms like testability and the SOLID Principle and so forth, the less amenable they become to attracting defects.

Scott Hanselman: Hi, this is Scott coming at you from another place in time. Are you looking for an Object Relational Mapping Tool for mission critical



projects using LINQ and .NET? I want to share with you Genome, it's specifically designed for developing .NET Enterprise Applications. Genome is a mature LINQ integrated ORM tool. It has been employed in numerous large scale projects in the last six years. Genome was created for the .NET platform as oppose to being a port from Java and it's derived from platform innovations since .NET 1.0. Genome has LINQ since its CTP release in May of 2006. It offers several unique features such as encapsulation and reuse of LINQ Query and Expressions. You can really and fully harness the power of LINQ while benefiting from your database platform's unique features, compose complex LINQ Query, decompose the Query logic in your domain model. LINQ supports all the major database platforms you find in enterprise environments like SQL Server and also Oracle and IBM DB2. You can find out more about how Genome integrates tightly with Visual Studio and what tools Genome offers to reduce irrelevant time at tinyurl.com/trygenome, G-E-N-O-M-E, where you can also download a free and fully functional trial version. I hope you enjoy it.

Okay. So are there some things, are there some specific things to watch for in these defect attractive design? Because sometimes when we leave the echo chamber or whoever talks about these things, it is very vague, and it is very difficult to take this generic concepts and apply them to something really explicitly and then point to a line or point to something. Is it folly for us to try to look for something really, really concrete when we say, "Well, all right, I understand what you're saying, Scott. Let's get specific."

Scott Bellware: Yeah, here's an example. It's interesting. I think that on your show with Bob Martin, he gave a really interesting example of what I would think of as a defect attractor or a design with a potential for defects which was when he talked about the substitution principle, the Liskov Substitution Principle, and he gave the example of a collection of objects that have a base class of square and somebody implements a rectangle class to be a subclass of square, and then at a higher level there's another programmer writing code that iterates over a collection of squares and calls method on them but the rectangle has different behaviors because it is not a square.

Scott Hanselman: I thought that was a great example. The idea was that one's observations about kind of reality don't necessarily always translate and get easily modeled in software and that was just a perfect example of here's something that you know how it works, you've known since 7th grade, but you can't model it like this.

Scott Bellware: Sure, but I mean the specific example he talked about in the code where he would have a collection of types whose dynamic type is a

base class and then you operate on them and you should be able to operate on any type, but then he said, well, we have this special circumstance when one of those items we're iterating through with the current item is a rectangle and then there's an 'if' statement there and then the 'if' statement has to do with some special processing. That special circumstance that is at that level of the design is likely a place that will cause some sort of defect to happen somewhere else and the reason for that is it's unexpected that it's there and its hidden knowledge, and this goes back again to knowledge and learning and design. So all of these great design qualities and all the Test Driven Development does is drive for the simplest units of understandability or the simplest units of the things that your brain can dissolve into itself and turn into knowledge and learning and you can see this very easily in some of these practices in Test Driven Development. We're really trying to drive at the smallest amount of set-up code so if you have a piece of business logic that has an 'if' statement in it that on one branch of the 'if', let's say, creates a new Database Pro but on the other branch of the 'if' updates a Database Pro, then you might have to set up that test to, well, you need a database, you might need some sample data if you're in the update branch, and then of course on both cases, you'll need to query that row out of the database to see if they work. Well, the presumption there that you need to have actually a live database in play to test the 'if' statement, that the 'if' statement works based on the variables that it was checking. It's the kind of presumption that you get too when you have large designs, large chunky designs when you don't have modules or modularity.

Scott Hanselman: Right and at that point it knows too much, you're starting to bump against Single Responsibility, you're stumbling on design smell.

Scott Bellware: Sure, but even if we don't go into those terminologies, I could look at the test code and see something that is often referred to as test friction which is the amount of set-up code necessary to run this specific scenario, it's more than we find comfortable because I have to create database rows, I have to set up objects in the database, I have to set up the data access layer and all these kinds of stuff. So if you look at what it takes to take any old object in your system, instantiate it into memory and then poke and prod at it and learn how it behaves, the amount of effort it takes to set up that code, that's kind of one of the most important things that we're driving at Test Driven Development. it's how much effort do you have to make before you can learn what an object does.

Scott Hanselman: Wow, that's a really good way to put it, how much effort do you have to make before you can learn what an object does. You know, a couple of years ago, you and I were having kind of a lowercase A argument, discussion about a word that



you use that I think really means a lot right now which is called solubility.

Scott Bellware: Yeah.

Scott Hanselman: Just how easy, how little friction is there between the design and getting it into your brain. Do you have to chew on it for a while or does it just kind of rock immediately and suck into your brain and just smoothly like drinking water?

Scott Bellware: Well, if you look at designs that require lots of set-up to use them...

Scott Hanselman: Exactly.

Scott Bellware: So they H, I said H like a Canadian because I am but I never say H...

Scott Hanselman: Don't tell.

Scott Bellware: The HTTP context object is the sort of classic example of a manager object, the very big chunky multi-role, it's a Swiss army knife that never ends.

Scott Hanselman: Its always there, it's in the background, you can pluck it out, it's simplistic parameters, it's just ever present.

Scott Bellware: But how can you ever instantiate one in memory. So let's say this because this is a great example. If I had a code behind a page for ASP.NET web forms application and I wanted to test that, my ideal scenario would be to instantiate the code behind class in memory and call a method on it and then I could see the text that was rendered from the code behind, you know, from the method. That's really not possible and it's really difficult to understand what a code behind page does and to learn what it does and there are articles and books written on what we think of as the, what is the pipeline people call it traditionally the HTTP...

Scott Hanselman: Right, yeah, the HTTP pipeline and all the eventing and stuff because things are getting called kind of via magic.

Scott Bellware: It is and it's that black magic so when we go back to the scenario of the business logic that calls the database where the 'if' statement on the first chase in the first branch might do an update and the second branch might do an insert or what have you, if the business logic has a dependency on this data, database module, then that dependency has to be made very clear, very obvious, very transparent. Let's say in that business logic module the constructor instantiates a database access class, a DOW object, if it does that instantiation in the object, then you can't really tell from the outside of that object that that's what's going on.

Scott Hanselman: Right.

Scott Bellware: So when I start, say I instantiate that business object class and I start poking and prodding at it to see how it works, I'll probably get a no reference exception or some crazy thing that happens once the code hits the DOW.

Scott Hanselman: Yup.

Scott Bellware: That's what you would think of with more esoteric terminology as an opaque dependency. Those things make it harder and harder and harder to understand what that business logic does. Does that make sense?

Scott Hanselman: It does, it makes total sense. So I'm used to things like Kent Beck's term Code Smell, you know we don't like that. It's a great way to just say, hah, this isn't right and I think that there are definitely tests smells and you're describing kind of design patterns that are really designed anti-patterns that then spill over into testing so you've got all sorts of test anti-patterns or test smells like you said.

Scott Bellware: Sure.

Scott Hanselman: A lot of context, a lot of implicit context that's coming in, it's not being passed indirectly from a parameter. You got lots of set-up code, anything that's in any concept like a session where you're plucking things out of some holding area outside of yourself...

Scott Bellware: Outside of the concern of the test.

Scott Hanselman: I was just thinking, outside of the concern...

Scott Bellware: So if we're talking about this business logic module that has an 'if' statement in it, if the data access layer, if I want to control which data access object goes into the test so that I can verify how it was used, I could make the data access object a singleton object so I would call, instead of calling new on the data access object, I would call get instance, but I have a hook in that class that says here is the instance that I want you to give. So I'd set up the data access object with an instance of the data access object so that when it returns, when some code in this constructor of the business logic module - and this is really I don't think coming across in a conversation as well as it would in a whiteboard...

Scott Hanselman: Yeah, I agree.

Scott Bellware: But if you have a registry of data access object that you use and you have to set up that registry, then you're not actually setting up the business logic object and that is a code smell. If



you're setting up factories and if you're setting up containers and if you're setting up these registries and sessions date and all of this stuff, yeah, you're starting to bring in concepts into this set-up of a test and let's not suggest the set-up of an object that bring in more concepts, more issues, more classes. HTTP context is a classic example of this. If you want to use an HTTP context, then you have to set up a lot of stuff before you could even test the thing that you're actually concern with which is the operation in a code behind page or a code behind class.

Scott Hanselman: Right.

Scott Bellware: So this insularity concerns that we have to bring into our test, this is test friction.

Scott Hanselman: Exactly.

Scott Bellware: My ideal scenario is I want to run the Click method on a page and I just want that to work. I want the design behind all of this to be fairly simple so that I can just make that happen.

Scott Hanselman: Right.

Scott Bellware: So there's a framework in the Ruby world called Merb and when you test for all intents and purposes a web handling method, you just call it, you instantiate the controller and call it and that method returns checks to the text as to web page and you can inspect the web page. There's nothing more simple than this, there's no true deep necessity, I'm over simplifying, but there's no real deep necessity to set up all of these insularity stuff.

Scott Hanselman: Right, right.

Scott Bellware: It's all the insularity things that we have to set up the objects before we can get them in memory and a good state so that we can start to learn from them. That's what the noise is, that's where the test friction is and any really good programmer should be able to say, yeah, but I'm a good programmer, I'm not really daunted by that, I'm beyond that, my skills are such that I can handle the pain, I can take the pain, but the reality is we tend to stop considering the other people on the team who are coming into our code likely from a cold start. I've never seen this code before or I have just been away from this code for a couple of days or a couple of hours or you know, whatever that time span is that reflects how quickly my memory fades.

Scott Hanselman: Yup and then you get yourself into a situation where you're working on a design and I was to call this a write only design. You can slap the keyboard all you want but no one is going to come in and ever read this and figure it out. It is write only.

Scott Bellware: Well, the problem with seeing what sort of judging whether or not the thing that you

did, just did, is good, is that you're judging it from having, you know, your brain has already warmed up to it. You don't see how difficult it is to come out from a cold start because you yourself are not in a cold start position.

Scott Hanselman: Yes.

Scott Bellware: This is the problem of creating usability or user experience. When we create a user interface -- most of the people who create user interface, they end up being user interfaces that are thought of as poor user experiences, aren't really sitting down to say, hey, this is a poor user experience. They really try to look at it and say, hey, this works, I think this is getting it, I'm going to go with this. The reason for that is they're already warm.

Scott Hanselman: Yeah, that makes total sense. They're warm, they have already positive attitudes towards the thing, they had this pride, there's ego involved.

Scott Bellware: So knowing that that happens, and it happens to every developer, I mean, the best developers are some of the worse user experienced people because they have such a powerful facility of memory that they keep this mental map of the system in their mind and they say, well, I know where all this is, I don't really need to create smaller objects just to help create new monic and the reality of the scenario is that's the minority of the developers, that's the hero developer, the developer with a photographic memory. The reality of the rest of the world is most developers benefit from dealing with objects and systems of objects that are much smaller and when they get smaller and when we apply some of these -- well, let's not even talk about applying design principles. If they're smaller and you can set them up in a few lines of code, so write down some example code and set them up and poke at them and see what they do, well, you've got a software that's learnable and like I said coincidentally that software probably expresses the quality of testability that people talk about in a developer testing community and it also will, more than likely, express many of the SOLID principles and if you just make the effort of writing down some sample codes saying here's how it's simple, I want setting up this object to be, I don't want it to be anymore complicated than this. If you make that effort, then the resulting implementation of that example that you just wrote of the level of simplicity that you want, the implementation will more than likely necessarily reflect all that principles and I can say categorically that most of the people that I know that I learned or that I came up with in OO and TDD didn't really start from the perspective of, hey, here is this great acronym from Bob Martin and here are these definitions of object oriented design principles that we found on Wikipedia. We kind of back into those things, really back into them like kind of backing up, kind of like one of those cliché Scooby-Doo scenes



where Fred is backing up and Scooby is backing up and they bumped in each other and they get surprised. We backed into all of these design principles and said, wow, look at this code that we're creating, it actually reflect these things that other people said are really good and desirable, and from that perspective of looking at the code we created, we really begin to get a good understanding of what these design principle were and then we started leveraging them explicitly and purposely. You don't really have to go at it from SOLID, it's great to know what they mean. Like if you just write these little examples of simplicity, you'll go a long way and that is what Test Driven Development is.

Scott Hanselman: Is the idea that you would write down an example, say this is how I want it to behave, this is how I want it to look, see how easy it ought to be. Is that Behavior Driven Development? I'm saying that it should behave like this, it should be this simple, it should be this basic.

Scott Bellware: I think Behavior Driven Development, and I can't speak for Dan North in the variable, you know, to hit up Dan North...

Scott Hanselman: Or mark him down, yeah.

Scott Bellware: About why Behavior Driven Development is called Behavior Driven Development, but there is a notion in specification, and by specification I mean test which is sort of specification as a terminology in TDD for test. We're really interested in developer testing, in unit testing, in testing behaviors because the value of the system is the things that it does.

Scott Hanselman: Exactly.

Scott Bellware: So we are doing our analysis from the perspective of behaviors, we're doing our design from a perspective of behaviors, kind of like this entity pattern or entity objects, we tend to call them behavioral objects, they're really there to represent clusters of behavior. Bob Martin talked about this in his conversation with you about Single Responsibility Principle. He is talking about creating smaller objects that have more smaller sets of behavior that they represent.

Scott Hanselman: He wants them to do less really.

Scott Bellware: I'm sorry, I'm sorry, yeah, I got distracted. Yes, they have less groups of behaviors.

Scott Hanselman: Right.

Scott Bellware: They're not just sort of objects named person, manager, where we hang every concept of person off of this object.

Scott Hanselman: Right.

Scott Bellware: But that kind of idea of a person, manager class is really a procedural programmer pattern. It's the person function library. So BDD helps us to focus our minds on this essential aspect of software which is the logic and so did the Test Driven Development as well. It's just really hard to talk about what Test Driven Development is because people keep getting really snag and hang-up on this word test.

Scott Hanselman: Yeah, I think it's a little amorphous but I feel that you've definitely enlightened me and I feel more like I'm kind of circling the drain here. I think like I'm heading in the right direction. I think that I can use this information to get a better understanding about test smells, about when a test doesn't feel right and a test smell inevitably points to a design smell...

Scott Bellware: Absolutely.

Scott Hanselman: And then underneath that, I may or may not find code smell but if there isn't joy in writing my test, if I'm doing lots of set-up, if my tests are full of reaching and pain and I don't enjoy writing them, then there's something wrong with my design and that, regardless of what we call this thing called TDD or BDD, that has become the essence of Test Driven Development, to unearth those things and to discover that not that there's a problem with my test but there's a problem with my code. My design is not sufficient if it can't be use effortlessly.

Scott Bellware: Right. Above and beyond that, this notion of effortless set-up and effortless exercise, the side effects of these, like I said, are the SOLID principles and the SOLID principles just aren't about testability. We used the word testability or easy testability to describe it but really the ability to do the things with software that we want to leverage it in more meaningful ways, that comes out of these principles. The ability to decouple software and distribute them, distribute objects in service oriented architectures, the ability to use plug-ins to do provider pattern kind of stuff, the ability to scale and parallelized also comes out of these in object oriented systems, also comes out of these principles. Once you get at this sort of quality of design, all of the other things that we need from design, that we require from design become much more readily available as a side-effect so it's we're going to focus on writing sample code that shows how easy it is to set up and learn from an object and the benefit from that isn't just testing, it's high quality design and even if you don't know how to do high quality design and everybody who has done Test Driven Development for a long time will say the same thing, even if you don't know what high quality design is, if you start with simplicity and expectations and demands for simplicity, you're going to end up at high quality design or at least a



higher quality design and the more you do it the more this will just become automatic and it's amazing what doors to knowledge about design this practice opens up. Like I said, no one have to start memorizing the esoteric terms and their meaning, and frankly those terms and those meanings are going to make a whole lot more sense when you've written some code that demonstrate it.

Scott Hanselman: So what are some resources or some books or preferred ways that people can dig into this, some action they can take after listening to this podcast? Is there a particular bible that you think people should read?

Scott Bellware: I think there's a mantra that we can start with.

Scott Hanselman: Okay.

Scott Bellware: If we just get people to say Test Driven Development is design...

Scott Hanselman: Test Driven Development is design.

Scott Bellware: That's a good place to start, after that there are so many resources about Test Driven at the moment but most of them, I think, are written for people who are already into Test Driven Development unfortunately. The lure and the love and the almost addiction to words and terminology of Test Driven Development and object oriented design immediately create an echo chamber. I think seeing fancy words produce serotonin in the programmer's brain and I'm definitely guilty of this but the problem is that it makes harder for people who don't know these terms. So most of the writing, most of the really good writing is written for an audience that is already somewhat initiated. The book that made it really turn on for me was James Newkirk's book in MSPress. He might be screaming at his iPod right now but his book Test Driven Development with Microsoft .NET or in Microsoft .NET is a pretty good set of exercises to try to get use to .NET, and I think Jim said recently that somebody needs to rewrite that book. It's a fairly, you know, I guess from a technology perspective, time-line perspective, it's a fairly aged book but it's the only book I've seen for .NET programmers to start fresh. Aside from that, there are lots of podcast, there are lots of blogs on those techies, on CodeBetter and sites like that tries to explain Test Driven Development but it's very, very difficult to explain something that has to be experienced firsthand.

Scott Hanselman: All right. Well, thank you very much, Scott Bellware, for sitting down and talking Test Driven Development and you will remember now that Test Driven Development is design. This has been another episode of Hanselminutes and we'll see you again next week.