



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 #163

May 26, 2009

Software Metrics with Patrick Smacchia

Scott sits down with Patrick Smacchia, lead developer of NDepend, and talks about Software Metrics. What metrics lie beyond Lines of Code?

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



Our Sponsors

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

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

NET 
DEVELOPER'S JOURNAL
<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 #163, recorded live Tuesday, May 26, 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 with Patrick Smacchia, lead developer at NDepend.

Scott Hanselman: Hi, this is Scott Hanselman and this is another episode of Hanselminutes, and I'm making an international call today talking to Patrick Smacchia in France. We're talking about Software Metrics. Patrick is the lead developer at NDepend. Thanks so much, Patrick, for being flexible with me and talking across time zones today.

Patrick Smacchia: You're welcome, sir.

Scott Hanselman: So I want to talk to you about Software Metrics because I'm a big fan of NDepend and a lot of the different metrics that NDepend provides to measure quality and measure coupling and all the different things about software that people don't measure and I knew that you would be the best guy to talk to because I think for a lot of people, lines of code is the metric that they start with and often the metric that they end with. Is line of code a useful metric?

Patrick Smacchia: Actually a line of code represents the chief code metric because we are all working with the size and you can milieu the line of code in a file. So we've been thinking in terms of line of code and maybe a number of five or so, but I just would like to make a point here about the physical, the advantage of logical line of code because this would just count the number of the line of code inside your file which is a lot of guys I've seen are doing that. I'm not sure whichever do you go about metric because it can be dead by the language and be dead by their code what else is coming. There is a lot of things coming inside this so what the tools are doing, like NDepend or Visual Studio, usually they are counting line of code with what we call PDB sequence point. Basically, a sequence point, every developer, .NET developer knows what is a sequence point. It's when you are developing your code, you just hit F9 to put your break point and what is underlined by your F9 is a sequence point, and by counting the sequence point we can say that sequence point is a line of executable line of executable code so we can say that the line of code can be counted by sequence point. I just want you to make this clear, very fast. Like this we have metric that is a language in different, like we know you're doing VB.NET or C# or F# or any other language, you have a metric that can be comparable

across language. Also it is different from the code style and you don't get comment inside their metrical source. So it's a perfect metric, it's a logical metric of line of code.

Scott Hanselman: Okay. So let me see if I understand this though. So I want to make sure that I've understood what you said. You're saying that when you use count the lines of code, when we do this simple line of code counters that we all have done in Computer Science 101 and the teacher says we're going to build a program to count lines, sometimes we'll put in simple checks for avoiding things like counting a curly brace as a line of code, that would be physical lines of code, you're saying that the sequence points, this is based on the PDB file, this is based on the -- is the PDB file required to count these sequence points?

Patrick Smacchia: Yes, exactly. Yeah, yeah. The sequence points are designed inside the PDB and usually they are used by the developer for the debugging experience to link the IL code with the source code, but what most of the .NET tools, whether you are thinking of NDepend, NCover, most of the .NET tools are working actually sequence points in one way or another and usually when you are counting the number of sign-off code, you're using the sequence point from the PDB and Visual Studio also is counting...

Scott Hanselman: Ah.

Patrick Smacchia: Yeah, yeah.

Scott Hanselman: Interesting. So if you can step over it, then you can count it.

Patrick Smacchia: Yeah, exactly, yeah.

Scott Hanselman: Ah, okay. Now in Steve McConnell's book on Software Estimation, he says that line of code really is an efficient way to compare applications.

Patrick Smacchia: Yeah.

Scott Hanselman: But only applications that are developed within the same context -- I mean, can I compare an IronPython application and a C# application with lines of code and get any meaningful result?

Patrick Smacchia: I think that's interesting to use, line of code for comparing. A lot of developers use line of code because they see it as a yardstick to measure productivity and I don't think it's relevant to measure productivity because more of line of code you unload by Cookie Password. You can have more lines of code and it's a really bad thing. So line of code doesn't measure productivity but it's still very useful indeed as Connell said to compare code



because, for example, I just have few numbers here to give, like the .NET framework with the definition of around two million lines of code, like ReSharper has over half a million lines of code, NDepend 80,000, and NHibernate over 35,000, and .NET has 45,000 and the list is very long, and now you see that, with these numbers, if I'm talking like this you can say that, wow, the direct framework is really a big, big, big piece of code and you can compare it to other tool or the software that you know. It's interesting that you can measure your own software and like compare your result in your software, compare with other result that you feel you know about.

Scott Hanselman: I think the thing that surprise me there was that ReSharper has a million lines of code but that's probably a topic for another day.

Patrick Smacchia: I'm not working with ReSharper but I've seen they have lots of generated code because each compiler is related and usually when you develop a compiler you generate tons of code. So I've seen ReSharper have lots of generated code, but it's also interesting to use line of code inside your codebase to compile future implementation like we know that from their feature A we have like 2,000 line of code and feature B we have 4,000 and then maybe you can assess all the feature of feature C, you can access the number of developed code and maybe you can have a good estimation of the time you're going to spend on this feature.

Scott Hanselman: Now line of code though as a metric, it doesn't measure quality, it measures quantity so it's just one metric of I assume several dozen metrics that we might want to use. What is the metric that I can use to really point at some code and it tells me whether this is good code or not?

Patrick Smacchia: First thing of course, it can be use for quality writing like if you have a very big method, usually a method is more than let's say 29 lines of code, it begins to be really help to understand. So for code quality, for maintenance, a line of code can be used at the code level and I would say even at the type level, but what's interesting with a line of code is that it's related to one of the popular metric which is the ratio of code coverage by tests, and whether you are using in the .NET world, NCover, or whether you're using Visual Studio for coverage, both tools are relying on PDB sequence point actually and line of code though sometimes you can add these tools actually for some kind of line of code. So these two metrics are related and in my opinion the vast show of code covered by test is the most important metric you can have because it's not related really to maintenance. Of course you can detect through regulation test but it's also related to correctness which is very important. Correctness is the reader, the signature of every developer and the more your code is covered and you can be sure that the more likely you are there are few verbs in it.

Scott Hanselman: So the ratio of lines of code covered by tests can tell me whether or not that code is going to run as I expected to. It will tell me that it's correct. Does it really measure code quality? I mean, I could write bad code and have great coverage. I guess good or bad is usually a subjective, like beauty is a subjective thing. Is there a more concrete metric I can use to decide whether a code is well written or not?

Patrick Smacchia: Consent to quality, you have some more there on metrics and more or less we all know about it. So there is one very important metric which is named Cyclomatic Complexity. By decree the definition is the number of passwords read can take in a method so maybe designed for method, the concrete method with somebody but more concrete fee. Cyclomatic Complexity is the number after four, is as switch case, etc, statement you have in the body of your method and indeed with this we speak complete with the number of line of code, you can have particularly an idea of method that won't be maintainable in the future anyway. So you have to avoid the order of this metric which is the Nesting Depth which is the deepest encapsulated scope inside the body of the method and so you can define this metric to see really well the big, fat method that is real pain to maintain and to develop and of course you have a lot of guides like the number of parameters, number of variables, number of overloads or method, etc. So all these metrics which the definition is pretty obvious from the developer's point of view, all these metrics can be use with the threshold and is very easy, you just fix your threshold like a threshold applicable in a read world like for example I don't want more than 10 complexity of a method or more than 20 line of code and then you need your CI, your continuous integration process so you can see immediately if you have new guys that not expecting this result.

Scott Hanselman: Okay. So if I have lines of code in my toolbox, I have Cyclomatic Complexity, and I have the percentage of code covered by tests, I can get a good sense of whether or not my code is objectively good code. I can tell whether or not it's going to run as I wish, but when I start looking at the specific functions and I say this function is too complicated and I break it up into pieces, the complexity of the business problem is still there, I've just made it into bite size chunks so I might be able to look at my application and say this is no longer complex from a method perspective but the software itself, the way that the software fits together, the way that it couples with other bits of code may indicate other problems. Don't you think?

Patrick Smacchia: Yeah, I completely agree with you because actually we are talking about two kinds of complexities. The first one that we're talking about is related to your software, it's related to the domain of



your software and other points. You will reach a point where the complexity of the software that you're developing cannot be smaller than what it is now, but by having the right method, very effective, by having all these very bad things, then you are fabricating even more complexity. So the goal here is to reduce the fabricated complexity to finally just keep the amount of complexity, that's what we'll have anyway because if you are doing professional software it is complex, anyway. So there are really two kinds of complexities and there is one you cannot really reduce, but the other one there are tools and there are metrics to try to reduce this one.

Scott Hanselman: I've used NDepend over the years and we used it at Corillean CheckFree, the banking company that I worked at, and I always found that it took me a while to get my brain around the way that it displays the information. It can be a little overwhelming when you try to present such a large amount of information over a very, very large...

Patrick Smacchia: Project?

Scott Hanselman: Library. I always show people the .NET framework, you know, thousands of types and it almost feels like I don't have enough pixels on the screen to present this. What is the name of the view that you use to show this; it's kind of like a grid and then there are colors and numbers within the grid and you can twirl in and out, what is that called that you're using?

Patrick Smacchia: So by decree, what we've done in the tool NDepend is that we have a seatwork panel and we try to provide several view on your code whether it's about dependency or whether it's about metrics or evolution, but when it comes to metrics the view you are talking about is named TeamUP and basically a TeamUP is just a view sort of rectangle, you have view to find other rectangles, so you use this rectangle as a read, as interface and it defines a sure path and DID is that the design interface, you just pick it up, release code it demands and its value. So for example you are looking at the number of the lines of code of type inside your library or inside the .NET framework for example, we can say that like there are two million lines of code. We can say that those sure path of the front-end is actually two million lines of code, and then what we want to represent each class inside the .NET framework, we just use that proportional to the number of line of code of the consume class. So by decree you'll get rectangles and lots and lots of sub rectangles inside this rectangle and the big served rectangle will actually be the big type.

Scott Hanselman: Hi, it's Scott here from another place and time. I hope you're enjoying the show so far. I apologize for interrupting it but I wanted to let you know that assembling a podcast like this every week isn't free. Certainly the bandwidth bill crushes

us every month so I want to let you know that this show is sponsored by Telerik. They make the show possible and they make some pretty cool products as well. For example, if you're trying to build a Web 2.0 AJAX application trying to use the Web 1.0 components, it's kind of difficult. You got to get to the next gen stuff to kind of build the next gen websites, and that's exactly what the folks at Telerik have got in their new upcoming product which is codenamed RadControls Prometheus. It's a big pack of web controls built entirely on top of the Microsoft ASP.NET AJAX stuff that you already understand. It's going to give you a lot of performance interactivity in your next project. They mirror the ASP.NET AJAX API so the development is really straightforward. Client scripture is shared, loading time is pretty fast, it sets a couple of properties, it even bind the web services for a really efficient operation. The new RadEditor for ASP.NET AJAX loads up to four times faster than before, and the RadGrid will do thousands of records in milliseconds. But of course it's better to try these things for yourself so you can visit www.telerik.com/aspnetajax and download a trial. Thanks a lot for listening and we'll get right back to the show.

Okay, so the size of the rectangle is something I can choose. I can say that this rectangle is going to be large because of the lines of code or because of some other metric.

Patrick Smacchia: Exactly.

Scott Hanselman: So the size of the rectangle is another access in my graph that I'm trying to make multidimensional data appear in two dimensions.

Patrick Smacchia: Exactly. That's exactly that and what's interesting also with the tree-map view, it's that the metric, excuse me, the class that are related like for example class C is inside of the same namespace will be placed close to the -- it will be located near inside the tree map so you can see that this namespace is big, this assembly is big, or this one is very small. So it not only gives you a size for method type but also for namespace and not somebody. So you can see the whole structure of the codebase in terms of the metric you are choosing and here is the number of the line of code.

Scott Hanselman: And then because of the grouping, things are grouped by namespace and by assembly. I could look at something, kind of I could back away from my large monitor and I could tell whether one assembly is a problem based on the complexity that appears in one assembly versus another. I could probably use this also to measure the complexity of open source applications that I bring in...

Patrick Smacchia: Oh yeah.



Scott Hanselman: Libraries that may feel good but when I look at the numbers, they might be trouble in the future.

Patrick Smacchia: This is another application of this kind of tool for measuring quality is good to measure your own quality but for example when you are buying all your library or when you order someone to do library for you, it's interesting to trust a minimal metrics to abide like that. You can be not really sure that the quality will be here but at least you can access a few more quality. I think this is a new kind of way to see the relation between the different actor inside the software development because now you can say I want this minimum of quality like you then knew though and retrieve when you are ordering the building or villa or house or anything, you want the minimum quality and it's visible with eye, and with this kind of tool now you can do it also on software.

Scott Hanselman: Right. Yeah, the ability to be able to stop a build because of something being wrong has always been important to me when I'm doing a CI, when I'm doing continuous integration. Being able to say that the build has failed because of some syntax error is really step zero. Being able to stop the build and declare that the build is bad because of a test failure is important but the idea that you could stop a test based on a design flaw I always thought was a cool idea that I could say this is just too complex for any human to understand and I want to stop the build and perhaps I could put in an exception.

Patrick Smacchia: Yeah but I have a caveat about that here, it's that now more and more and it's for covering more and more we have a lot of generated code and actually just very recently on your show with Kathleen Dollard about code generation and one question was about the quality of generated code and what we can see is that the real fun, the quality of code generated is not there because this code actually won't have the situation to be understood by the humans. It's not really a good thing but when tool analyzing code, at first it doesn't make the difference between generated and human handcrafted code...

Scott Hanselman: Sure.

Patrick Smacchia: So you don't want to follow metrics too much about that. For example, in NDepend we are providing complexity like you can use some regular expression or you can require the type outside of this assembly won't be inspected or things like that, but still you have this generated code, maybe you don't really want to hear about inside your report so this is a good thing to stop the build, to fight the build because of quality, but you want to make sure first that you have found this potential problem.

Scott Hanselman: Right. It sounds like a hell lot of exceptions definitely. Now lines of code are still -- this is still pretty basic. We are kind of moving from

the basic metrics to the more complex. I think that everyone has used lines of code at some point, people as they start getting into test driven development and respecting their test as they start caring about code coverage. The things that I think are the most interesting metrics are the ones that I'm afraid you use a lot of math and they use a lot of charts and graphs to explain. The one that I found the most useful when I was working in the banking industry was this notion afferent and efferent coupling. Afferent, A-F-F, and efferent, E-F-F and I think they're really horribly named because I can't quite keep them straight, and the idea is that who is using me and what am I using. Is that a fair way to put it?

Patrick Smacchia: Exactly, yeah.

Scott Hanselman: How do I keep track of which one is afferent coupling and which one is efferent?

Patrick Smacchia: Yeah, so these two metrics actually they are related to something very popular for every architect which is everybody now they know what age and it is a good thing, everybody said I want low coupling and I want high cohesion inside my code. What's interesting is that we have metric for that. So metric concerning coupling, as you said they are inconvenient outgoing or as they run and if they are uncoupling and this metric can be very interesting because, for example, let's say if they're uncoupling, what does it mean is you have the high of they're uncoupling let's say for class. The highest if they're uncoupling means that you have a class that is using lots of the class.

Scott Hanselman: So efferent coupling means I have a lot of outgoing references. I'm using a lot of stuff.

Patrick Smacchia: Exactly. Imagine you have a class that is using a lot of stuff, then you can say almost sure that this class is breaking the single responsibility principle. So imagine a class that is using some WPF things and you're choosing those and maybe some switching things, etc, you sure are observing your class like this kind of big, fat class that is using a lot of things, then you can see that the single responsibility principle that just stayed actually, that your class should have just one reason to change or they just stated that a class have just one concern, we can see that it's always broken in naturally stuff conveys I see there is a lot of such classes. These classes they can dispose it with line of code or the complexity but if you run coupling this is another good way to start. This is a real key concern class and concerning the afferent coupling, I mean the number of guy that use me, imagine that you have a class that use very popular that is use inside your codebase, what does it mean, it means that if this class is changing, it probably broke a lot of what I think high afferent coupling. On a class, I think it's best thing too and here there is a twitch, the twitch is to transform



your class inside of other -- you transform your implementation to an abstraction. So you have a high afferent coupling and in a codebase you cannot avoid it, there will be some type that will be very popular because codebase made it like that but I've seen it's a good thing that this type will be implementation. Even when you shall do one interface for one class, it's a good thing because at least you can agree on a contract, you can define your interface as the contract and at least you can agree with all these users that they should use their interface this way. Maybe the implementation evolved, it won't be seen by user.

Scott Hanselman: Ah, okay.

Patrick Smacchia: So that's two usage of this metric that I find interesting, and finally, another usage of afferent coupling is interesting because if you have zero, it means that your class is not sure at all and this is not always true because of course you can use it with reflection, you can use it in putting more tiers in more things like that but definitely 24:56 you can see if a class is not used and in NDepend we are using three tools looking for that decree that's called. That's another usage of, interesting usage of afferent coupling.

Scott Hanselman: Okay. So afferent coupling, the incoming usage, you're saying that having a lot of people rely on me puts a lot of pressure on me as a class.

Patrick Smacchia: Exactly.

Scott Hanselman: It also tells me I'm going to be a problem if I change so I like that. Afferent coupling is a good signal that you should be thinking about interfaces if you haven't already been.

Patrick Smacchia: Yeah, exactly.

Scott Hanselman: Very cool.

Patrick Smacchia: There is another little order of this metric which is the ranking metric inspired from the Google ranking page algorithm. In NDepend we implemented this metric on both type and method, or maybe better said on the graph of type and the graph of method and now we can see with this metric which one is very important inside your code. So just as an example, if we apply this metric on the .NET framework, immediately the top 10 types will eject integers, strings, Boolean, etc, all these primitive types that are extremely important and actually used everywhere. But now if we translate this metric on another codebase, it's very interesting because if you don't know a codebase you should just begin help for codebase, you just write, then you can see what's really important, what really matters inside the codebase and then you can begin questioning this thing and see how they are related to components,

etc, and this is also a good metric to use, to discover a codebase.

Scott Hanselman: This is one of the things that I think is exciting about Software Metric. It seems obvious to you, I think, because you work in this stuff everyday, but the idea that you could come upon a metric which is really just an equation and you could say, oh, this would be fun, let's see how we can apply this to code so the idea that you took Google page rank as a popularity and applied it to the code is a way I wouldn't have thought, I think I wouldn't have thought of.

Patrick Smacchia: And this is very interesting. I'm not sure I'm the first one that did that. That is pretty interesting research.

Scott Hanselman: Well, certainly no one is the first one to do anything anymore.

Patrick Smacchia: I know that they'd leave you alone like now, which is now I'm on Microsoft Research on the text, I know this metric app thing before me, it is metric inside .NET framework...

Scott Hanselman: Ah yeah. One of the metrics that stood out that I thought was fun was this metric called CRAP, the CRAP metric which stands for Change Risk Analyzer and Predictor. It's a Java metric that basically tells you if your code is crap.

Patrick Smacchia: Yeah. The problem with -- here we have some other metrics like actually the metric compilation. So now so far we just talk about Project Metric that can be really initiate image at least from the codebase and the idea here is to put some equation and to compose this metric. So for example, the CRAP is the Cyclomatic Complexity of square, multiply of I one and minus the coverage, etc, etc, so you have a big formula and at the end an experience to you that as you're given by this formula, in this experience it would be very bad and this is when it will be acceptable etc, so you can use metric, you can compose metric. There is another example of metric composition in Visual Studio, you have to maintain ability index which is a mix of all the metric and Cyclomatic Complexity, line of code, etc, and the idea is that here you get the number without dimension. When you say line of code, you should mention this one line of code. Why? Or when you say, yes, I run coupling, what you mention is the number of user, but here you get the metric without dimension that is supposed to give you reasons about what's really bad and in my opinion I don't really believe in this metric because I've never seen extra composed metric because it's very easy to fake this metric with false positive and counter example, then be sure that it doesn't work all the time. I've seen that the definition of Cyclomatic as to state if you understand both for everybody and the minimal requirement is that you can understand the dimension of the metrics.



Scott Hanselman: The thing that I like about this metric and why I think the CRAP metric is fun, and not just fun but useful, is that in this instance where we're composing a metric about complexity, we're composing a metric using Cyclomatic Complexity, and composing it with coverage, we're basically saying that this is complex but I'm protected by this tests.

Patrick Smacchia: I think you can have a more linear way to compose metrics. So for example, the Depth approach we are taking in NDepend is that you can just define some rule and you cannot, for example, for complex method, those are not covered because you can define some kind of what we call sequel query so by decree we are querying the code as you would query a .NET base and maybe you can really ask or select a method where the complexity is higher than maybe 10 and the best attached coverage is lower maybe than 10 for some.

Scott Hanselman: And you get effectively the same thing is what you're saying.

Patrick Smacchia: Exactly but it's more linear and at least it remains understandable and if you have a false positive it will be very easy to understand why here you are with a problem but usually by composing linearly the metric you can understand the reason result which is really the most...

Scott Hanselman: Yeah, that's a good point. The CRAP metric is interesting because it squares the complexity number and then it cubes the code coverage number so it's putting more weight on one aspect of the metric than another. As with all of these things, they have a reason for it but I think your point is well-taken. You can't really easily look at the value and say here's what I need to do to pass this metric. Another compose metric that you use is this VS maintainability index. Is this the sort that the FXCop team uses? It's an index that's more -- it's not percentage, it's not 1 to 100, is it? I don't quite understand the number.

Patrick Smacchia: That's exactly the point I was making. You don't get the number because there is no dimension in that number because it's a mix between several metrics like complexity, line of code, etc, and so you cannot put a dimension and instead Microsoft just tell you that you'll have a number between 0 and 100 and maybe if it's up to 20 it will be good but you just force to listen to them because you cannot understand it, the dimension of metric, and you cannot do choice by your own. This is the proper time with composing not linearly metrics, I think.

Scott Hanselman: Yeah, I agree. It's very confusing. Now I've been playing with .NET 4.0 and I know that you have too and you've been looking at, I think you call them evolutionary metrics, or metrics around how things change so being able to compare

multiple builds and you had a blog post a couple of days ago where you were looking at .NET 3.5 and .NET 4.0 and seeing what was changed and what was added to the .NET framework.

Patrick Smacchia: Yeah, I've seen that the evolution and the changes inside the codebase is extremely important and whether you're using little framework like the .NET framework and want to see what was evolving or whether you are just seeing the evolution on your own code, I think it's very important because think about it, when you release there some code, when the code is on production, usually you don't have verbs or you have few verbs when the code is in production or at least this is the code you want to achieve. So between the release, $N - 1$, and the next release number N , all the issue runs between the two snapshot of codebase, can contain potential verbs that at least haven't in production yet. So I think that focusing on two, the Delta between two releases is extremely important and in my opinion there are few development teams that are using this powerful feature which can be used actually. In NDepend, you can couple this feature with, for example, quality metrics. So for example, you can write some rule and just ask for, tell me about methods that have been agreed or maybe refactored the way the code was changed and I want to know about this method and I want to make sure that all these methods, maybe 90% coverage by tests or even 100% or maybe you can require that this method maybe like a complexity or lower than 10. What is really cool here is that everybody is dealing with legacy because now you are giving very huge contest.

Scott Hanselman: Right.

Patrick Smacchia: And you can say from now all the refactored method, all the agile method, I want them to expect the minimum quality. From my perspective, this is the best feature of the project to allow you to focus to the changes and to do some rules to make the quality better on the changes because now we know that the code gets refactored and we know that in one year, in two years, in three years, legacy codebase will be more or less transformed to tally. So if you focus on delta after delta, on the quality and we are talking about coverage or Cyclomatic Complexity, etc, you should focus on quality just on the delta, you don't need to say that from now we must stop everything for three months and refactor everything to about quality metrics. Here, the interesting thing is that just the delta we are purchasing now since the last release, we won't focus on the quality on this one and if you apply this very simple principle, maybe in one year, in two years, and each year you will get a good codebase.

Scott Hanselman: All right, I agree. I think that overtime we'll be able to set a bar and make sure that you don't regress, that you don't go back in time, to a



time that was -- the quality of your code was lesser, I think is very important. So the product that you selling is called NDepend and people can see that in ndepend.com and you blog about Software Metrics at codebetter.com and people can find your blog there and I will put links to all of these in the show notes. I really appreciate you taking the time to come and chat with me on the show today, Patrick.

Patrick Smacchia: Okay, you're welcome. I appreciate you too.

Scott Hanselman: All right. This has been another episode of Hanselminutes and I'll see you again next week.