



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 #145

January 5, 2009

SOLID Principles with Uncle Bob - Robert C. Martin

Scott sits down with Robert C. Martin as Uncle Bob helps Scott understand the SOLID Principles of Object Oriented Design.

(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 #145, recorded live Monday, January 5, 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 about the SOLID principles of object-oriented design with Robert C. Martin.

Scott Hanselman: Hi this is Scott Hanselman and this is another episode of Hanselminutes, and we're sitting down today with 'Uncle Bob', Robert C. Martin, the Founder, CEO and President of Object Mentor Incorporated but known colloquially to the software community as 'Uncle Bob', thank you sir for chatting with me today.

Robert C. Martin: Oh, it's my pleasure.

Scott Hanselman: One of the things that I'm hearing talk about a lot in the community that I kind of run in, the .NET community, is this solid principles, these principles that you outlined in your discussion of the principles of object oriented design and it's an acronym of acronyms that listed out five kind of basic principles that add up to solid, and when someone who's maybe not a software development wonks sits down and looks at these things, it's like patterns and more patterns and principles and it's just very, it could be looked out as a little academic and I could see where some people's eyes would blur when reading this kind of stuffs. So, how do we make this a more accessible? How do you demystify the solid principles?

Robert C. Martin: Yeah, I confess that the names of the principles were something of a conceit, I had spent a lot of time studying physics and astronomy and in those disciplines you find principles like The Heisenberg Uncertainty Principle or The Poly-Exclusion Principle and so I rather like the idea of three words, with the last word being principle. That was spurred on even more by a book that was written in 1992 by James Coplien affectionately known as Cope who coined the word the Liscov Substitution Principle which is the third in the solid list. So to demystify them, let's take the first one, The Single Responsibility Principle, it's a nice word, the flows real well SRP, Single Responsibility Principle, what does it mean? It means that a software module should have one reason to change, then that's what I call a responsibility, a reason to change. So for example, take a payroll application, if there's an employee class in that payroll application, you could imagine that it might have methods for calculate pay or perhaps another method for print a report, perhaps another method in the employee object for save me to the

database, and what's unfortunate about these three methods existing in the same class is that they have all three completely different reasons to change. The payroll, the calculate pay will change if the accountants decides on a new way of calculating pay. The report generator will change if the people who consume the reports want the format of the reports to change. The save function will change if the DBA's decide that we need to change the database's schema. That means that this one class has three different reasons to change and there are probably many other classes that depend upon it and so as it changes those depending classes also suffer through change, they'll be effective or impacted by those changes. So the Responsibility Principle simply says, "Find one reason to change and take everything else out of the class." So that you separate the things that change for different reasons, you group together the things to change for the same reasons. This is somewhat out of the norm for object oriented design, early object oriented design principles had as grouping together functions that operated on the same data structures so that the methods of a class would all manipulate the same variables of that class, but if those methods change for different reasons then they really belong in separate classes.

Scott Hanselman: That's the definitely flips things on its head because I remember when I got into C++, I was moving from C and we were already trying to write what we called Object Oriented C. Having an associated group of static methods act on a structure that was a C structure and we lied to ourselves and said, it was OO and then moving that to C++ we would take you know book objects and people objects and then give them all sorts of bits of functionality. This almost inverts that whole concept.

Robert C. Martin: Well it does, in fact it drives it somewhat closer to the thing that you were kidding yourself about because in C you could take a list of, a data structure in a .H file and then you could take a list of functions that manipulated it and put that in a one C: file but you could take another list of functions that still manipulated it and put it in another .C file, separate them, so that you could, in that case, have functions that did reports, functions that did database manipulation, functions that did calculations, manipulating the same data structure but in completely separate .O files and now it'll allow you to separate them into different dll's nowadays which probably couldn't do back then, and deploy them independently which is one of the main goals here.

Scott Hanselman: Now, one of the things that I was taught in school was that naming your objects was very, very important. My particular instructor just pounded that into my head, that if you can't name it and feel good about the name and feel that the name says what it is, then you need a better name. It seems like the Single Responsibility Principle would cause an explosion of classes and I would suddenly



be faced with difficult naming decisions like what exactly is this now that, now that it is not the object that I expected that it's going to be?

Robert C. Martin: Yeah, and in fact the exact opposite is true, because the more smaller objects you have, the more focused they are and the more easier it is to think up of a name for them. We get into naming dilemmas when we have an object that does so much that it's impossible to name it except to give it some broad rush name like employee manager, but if you separate out a bunch of very small objects, it's very simple to come up with name, if their focused. So you'll going to have the employee report generator and the employee database saver and the employee payroll calculator and those names are very specific to what those classes do.

Scott Hanselman: So if we're grouping things based on their reason to change, which isn't quite data and it isn't quite methods it's more their domain it's more the space that they work within. What are the names, I mean traditional object oriented design, the way that I was taught pushes us into these manager objects, master objects, service objects, these generic terms that mean I know too much. What kinds of names or objects am I going to end up with while I'm applying the Single Responsibility Principle appropriately?

Robert C. Martin: Well, you'll wind up with longer names generally but very focused. So you'll create a class and that class will do one thing, for example the class that saves the employee to the database, you might call this the employee gateway using Martin Fowler's gateway pattern or you might call it the The employee DB, bad name, but some people like to put DB on their database controllers. Whatever conventions you used, it becomes clear that this class relates employees to the database and does not do calculations and does not do report generations. If you had a class that puts an employee record up on the screen and perhaps in HTML, you might call this, the Employee HTML Renderer or something like that. Very precise names for many small classes as opposed to very generic names for a few large classes, and by the way, this practice goes right down to the function level as well, we want to have lots of function levels not few big functions and by little, I mean five, six, seven lines long. Functions should be, should follow the same rule as "The Single Responsibility Principle."

Scott Hanselman: And that applies to methods too, you're saying, so a class should have only one reason and the method should have only one reason to change.

Robert C. Martin: Exactly, yeah. A method should be extremely focused, if it's got an IF statement in it, it probably shouldn't have two or three,

if it's got a Y looped in it probably shouldn't have two or three, and very, very small focused methods.

Scott Hanselman: Yeah.

Robert C. Martin: What you get with that is, it becomes very easy to name them because you can choose very long names, because they're only called once, and those names can be extremely, extremely informative and then you wind up with lots of little bits of code underneath well-named functions. I call this "Cleaning up the room." Remember when you were a teen-ager or younger, your filing system in your room was to dump everything on the floor, and you knew where everything was, and it was fine, and your clothes would be on the floor and your toys and your books and whatever they'd be on the floor and you knew it all was but your mother would comment and say, "No, you got to clean-up this room."

Scott Hanselman: Much like my Windows desktop.

Robert C. Martin: And eventually out of frustration, she would clean it for you, and she would taught you, "Put things in certain places, know where they go." And maybe if she was of the old school she recited that old, old saw, "a place for everything and everything in its place." That's the same kind of thing we get with The Single Responsibility Principle, we divide things up into small little bits, we name the place that they go with extremely informative names so that our code, our entire system follows the rule, a place for everything and everything in its place.

Scott Hanselman: Well, how long is this going to, if I go and apply this today and start thinking about this, how Draconian should I be about these kinds of things? How ruthless should I be when I make these decisions? I think it would apply a very different feel, it would make my, it would make my programming day feel different, for lack of a better way to rephrase that...

Robert C. Martin: Oh yes, oh yes. It'd probably make your programming day feel different, because what you're doing is imposing a great deal of structure on something that otherwise would have been somewhat amorphous. Imagine a function that is 50 lines long and the left edge looks like the side of the cross-cut saw, deeply indented and it goes in and out finding your way around in there is fairly difficult and then remove chunks of that, split it up into 10 functions, 5 lines long each one has one level of indent and name each one of them and you wind up with a structure that is readable and understandable and you don't have to dig your way deep inside of the 6th or the 7th indent to figure out what's going on, you can read the function above it and say, "Oh, that's what's happening here." That's a more powerful way of working. Some people get afraid that, "Well that's going to proliferate all functions like crazy, I'm going to



have too many functions and too many classes and I'm going to get lost in this sea of classes and functions." and actually the opposite is true, if you don't do this you are all already lost in a sea of heavily indented code that has no names and no structures. Once you put the name and the structure a part of it, it gets a lot easier to find your way around.

Scott Hanselman: Okay, alright. So that's the S in Solid then, so then the...

Robert C. Martin: Yeah, we got through one.

Scott Hanselman: We got through one, but this is good though, this important because I think this is the closest thing to commandments and I don't commandments in the sense that you know, "Uncle Bob and family went up on high and came down with this information," but this is guiding principles that are good basic stuff that aren't really arguable. "Thou shalt not steal." is pretty ambiguous and you can always find reasons to, there's always exceptions, right? I'm sure that there are reasons to make a multiple responsibility class but then you can certainly debate that until the cows come home.

Robert C. Martin: Sure, you can think of these as engineering principle as opposed to commandments.

Scott Hanselman: Yeah, exactly. So the O, Open/Closed Principle.

Robert C. Martin: Yes, this is a principle that was coined in 1988, I believe by Bertrand Meyer in wonderful book of Object Oriented Software Construction which was one of the early terrific books on object oriented design. This principles states what sounds like an oxymoron, sounds like a contradiction in terms. It says that, "Modules should be open for extension but closed for modification." And that means that you should be able to change what a module does, you should be able to extend its behavior without modifying any code in the module, whatever. Imagine that you have a module and you set the READ ONLY bit, you're not allowed to change it and yet you can still modify what this module does by using some other mechanism and that's what's the Open/Closed Principle is all about. Obviously the mechanism we use is polymorphism, we create abstract interfaces so that we can have the module that we want closed calling an abstract interface and then we can extend what that module does by creating derivatives of the abstract interface. The benefit here is that it allows us to create systems whose behavior can be extended without modifying the core of code. We can add new features by adding new code, not by exchanging old existing code and that has obvious benefits. If you could add a whole new suite of features and never touch the core of the code, just add new code, you would not break the core of the core, you would not be putting it in any kind of risk. So the Open/Closed Principle is a

principle of attitude, we want you to think about how you can separate the behaviors of the system so that you can keep those behaviors that are utterly intrinsic behind a wall that can't be touched and then all of the behaviors that are variable and transient and change a lot, you put on the other side of the wall and you allow them to change frequently as you want and you make sure that all of the dependencies that cross that boundary point inwards towards the code that can't be changed. The beauty of the inheritance relationship is that it points at the base class, the base class changes less frequently than the derivatives and so we try to apply that with object oriented techniques, using the Open/Closed Principle, to create this two-sided structure, on the one side, you have all the base classes where nothing changes, and on the right side you have all the derivatives where things are very, very volatile and all the dependencies point towards the base class.

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 in .NET? We want to share with you Genome, specifically designed for developing .NET enterprise applications. Genome is a mature LINQ integrated ORM tool, it's been employed in numerous large scale projects over the last six years. Genome was created for the .NET platform as opposed to being a port from JAVA, and it's thrived on platform innovations since .NET 1.0. Genome has supported LINQ since its CTP release in May of 2006. It offers several unique features such as encapsulation and reuse of LINQ queries and expressions. You can really, fully, harness the power of LINQ while benefiting from your database platform's unique features. Compose complex link queries, decompose the query logic in your domain model, LINQ supports all the major database platforms you find in enterprise environments like SQL Server, but 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 development time at tinyurl.com/trygenome, G-E-N-O-M-E, where you can also download a free and fully functional trail version. I hope you enjoy it.

Now these principles don't dictate language but they do kind of there's an, it feels like I think, traditional object oriented languages. Do we use traditional, is this a Polymorphism Principle or can we do strategy patterns and have a different pluggable interfaces to achieve the Open/Closed Principle or does this dictate, mark your methods virtual?

Robert C. Martin: You can, you can do this with the standard virtual methods in C# or in Java or C++ or if you wanted to, you can use pointers to function in C or you could use cleverly crafted switch statements in Fortran or computer GoTo's...



Scott Hanselman: Okay, so the principle is the important thing.

Robert C. Martin: Or in other languages, there are other many, many ways to achieve this. Even the template languages, C++ and to a lesser extent Java, give you some capabilities for doing this. The basic idea is very simple, keep the things that change frequently away from the things that don't change, and then make sure that if they depend on each other, the things that change frequently depend on the things that don't change frequently.

Scott Hanselman: Okay, so...

Robert C. Martin: That's really the Open/Closed Principle and you can do that with Polymorphism, you can do it with textual substitution, there's a whole bunch of ways to do that.

Scott Hanselman: So the Open/Closed Principle kind of segways very neatly into the crazily named or the interestingly named Liscov Substitution Principle, that's the one that you make you sound extra smart if you throw it around the office.

Robert C. Martin: Yes, right. That was the one I like because it's so much like the Poly-Exclusion Principle.

Scott Hanselman: Yeah, you can't help but sound intelligent saying these things around the water cooler.

Robert C. Martin: Liscov Substitution Principle. Yes, yeah. Very erudite.

Scott Hanselman: Now does this one, this seems to be a little closer to, this principle seems to expect virtual methods to work in certain way. It seems to expect the language to work a certain way more than the other, the previous two principles we've covered so far.

Robert C. Martin: It's certainly has a very strong connection to inheritance and virtual functions, although it applies in languages like Ruby or Python which don't have traditional inheritance of that nature. The idea behind this principle is pretty simple, although the implications are very far reaching. If you have an expectation of some object or some entity and there are several possible sub-entities, we'll call them sub-types that could implement that original entity. The caller of the original entity should not be surprised by anything that happens if one of the sub-entities is substituted. So, the simple way to think about this is, if you're used to driving a Chevrolet you shouldn't be too surprised when you got into a Volkswagen, you'd still be able to drive it. That's the basic idea, there's an interface, you can use that interface, lots of things implement that interface one way or another and none of them should surprise you.

Now the canonical example is the rectangle square problem, and this is a huge philosophical debate it goes on all the time in OO circles or at least it used to I think it's pretty well resolved now but given a rectangle-base class and given a, the need for a square, should the square class derive from the rectangle? At first blush, this seems like it's obvious, of course the square should derive from the rectangle, a square is a rectangle, and the word 'is a' becomes very important.

Scott Hanselman: 'Is a', exactly.

Robert C. Martin: The word square is a rectangle. Well then you cite some interesting dilemmas, once you create that relationship then you start to ask yourself some funny questions like, "Wait a minute, how many variables does a rectangle have?" Well, that's two height and width. How many variables does the square need? It only needs one. Well how many is it going to inherit? It's going to inherit 2. Or is something wrong there, it's getting too many variables from the base class. There's a dilemma here and it sounds right that square should derive from a rectangle but if it does, it's going to be wasteful of memory, now memory is cheap we can ignore that. Okay, there are methods inside a rectangle, methods like set height and set width and when we derive square from rectangle, we're going to inherit those methods. What does that height mean on a square and how is it different from the other method that's inherited by a square which is set width. There's a naming problem and the user of square is going to be faced with these names set height and set width, they don't make any sense on a square. Now, we can make them work by causing the set height function to also set the width and to cause the set width function to also set the height but the main issue is still messed up, there's something wrong and then we force the failure, there is a failure mode -that can occur here. The failure mode is very simple. You've got some guy up there who's some class who uses rectangles and the programmer of this class has made an assumption, when you change the height of a rectangle, the width never changes, purposely reasonable assumption if you're dealing with a rectangle but now because I have derived a square from rectangle, I can pass in the square, he's going to call set width and the height is going to change out from underneath him and because he didn't expect that he'll have some statement that blows up, it'll corrupt the heap and the billion instructions later, the whole system will crash and he'll get his logic analyzers, they'll debug this thing for 2 weeks and eventually he'll find out that, "Oh my God somebody passed me a square and I was expecting a rectangle." and as a result of that he would put an IF statement in his code and this statement will read, "If this rectangle is really an instance of square." And as he's done that he has hung a dependency on the derivative of a rectangle. He has mentioned the name of the square and that dependency...



Scott Hanselman: Right.

Robert C. Martin: Violates the Open/Closed Principle. So you wind up with IF statements scattered around your code because of these violations of the Substitution Principle, you couldn't even see it first, and then these statements hang dependencies on nasty derivatives that then violate the Open/Closed Principle, makes it very difficult for you to modify a square without changing all of these things that now depend on rectangles.

Scott Hanselman: So that's, that makes sense but let me push, what would you do, I mean, a pick a side

Robert C. Martin: What would I do with the square-rectangle problem?

Scott Hanselman: Yeah.

Robert C. Martin: They aren't related at all. Squares and rectangles aren't related in the code, but a square in geometry is related to a rectangle in geometry, but these two pieces of code are not pieces of geometry, they're pieces of code, they have completely separate behavior. So I wouldn't have any relationship between them at all or, at most perhaps, they would have a common parent which might be shapes.

Scott Hanselman: Yeah, that's interesting. When you first said "Is a", of course the object oriented light went on but then later on you said "is related to." I think it's comforting sometimes to try to make, as we as programmers try to make order out of chaos to go and say, "Oh a cat is a mammal, is an animal, this is a nice, clean hierarchy. Let's go and start naming things." And we try to simulate these hierarchies when you know, "Maybe a cat is different, you know. Suspend this belief, it's not a mammal, it's something else."

Robert C. Martin: The word "ISA" crept into our vocabulary and by the way that's one word ISA, it crept into our vocabulary through a circuitous route and became very important in object oriented circles, but it didn't start out that way. It crept in the 80's through the AI crowd who had created these wonderful knowledge nets, you might remember this, all the hype about Artificial Intelligence in the late 80's, early 90's, and then created these structures that would walk knowledge nets these inference engines, and the relationships between the entities and the knowledge nets with things like, is like a, tastes like a, smells like a ISA, all of these -A relationships is a, like a, has a and when the AI crowd lost it's funding, and all those funding drives, they kind of looked over and said, "Oh, there's this other stuff, it's kind of cool. Look, there's these relationships like has on and is on, it's real similar, we ought to just move in." And they kind of did and the vocabulary transitioned over.

That's interesting, it's also a little unfortunate because inheritance is not ISA. Inheritance, if you look at it with a very jaded eye, inheritance is the declaration of methods and variables in a subscope and it has nothing to do with ISA whatever, and the notion of ISA can be very confounding. Simple example, an integer is a real number and a real number is a complex number. You could draw that in your UML, it'd very simple with all the inheritance there and so forth, but think about trying to compile it. An integer we would hope would be 16 or maybe 64 bits but if it were to derive from the real number, a real number has two integers in it, a mantissa and a characteristics, the exponent and they used, they implied binary points inside them to make these floating-point number. The floating-point number, the real number, derives from complex but the complex has two real numbers in it, the imaginary and the real part. If you were to think about writing that in C++ or in Java, you would write a structure that could not be compiled because it has infinite thought. Makes perfect sense in English makes no sense at all in software.

Scott Hanselman: Which, seems to be the most of the things that I work. It all seems so clean and then you start to model it, it makes me think about those, these new modeling languages that people are coming out with to, you know, model everything.

Robert C. Martin: Oh yeah.

Scott Hanselman: What do you think about those grand designs to model, all we need is a new modeling language and then it will be okay?

Robert C. Martin: Yes we, it's we've always been on a hunt for a new modeling language, we always will be and that's a fine thing I think. We want to continue to hunt for a grand unifying language, there is no such thing, obviously.

Scott Hanselman: Yeah.

Robert C. Martin: There's a, you've heard about the MDA people, the Model Driven Architecture folks, I think this is a great effort. I wish them a lot of luck. I have no confidence, whatever that they're going to find a way to create a modeling language that is so good that you won't need programmers anymore. Programming is the profession of managing details, detail and information system and there's no way to get rid all that detail. Any modeling language is going to have to be able to deal with nasty, risky, rotten, lousy detail and it's programmers who drive the language to do that kind of mastery. So if it's a language that uses UML or language that uses some other wonderful graphic tool or a language of some other kind, it's still going to be dominated by massive amounts of detail and require programmers to manage it.



Scott Hanselman: Yeah, and at some point, someone will do a switch statement and then they'll spackle over it and pretend it didn't happen and then it becomes legacy code.

Robert C. Martin: Wow, yeah, have you read Feathers' book on legacy code?

Scott Hanselman: I have, actually, Carl Franklin, had him on .NET Rocks, I think at OrDev.

Robert C. Martin: Oh good, yes, at OrDev. Michael's definition of legacy code is code that's not under test.

Scott Hanselman: Exactly.

Robert C. Martin: If you don't have tests for it it's already legacy code.

Scott Hanselman: Yeah, it's interesting going back and finding code that I wrote not 10 years ago which had no tests but 5 years ago which did have piles of NUnit tests. The only thing that I feel when I see my old legacy code is well, not legacy code but my older code is, "Gosh, I should have written more tests."

Robert C. Martin: Yeah, well...

Scott Hanselman: That's the one thing I keep coming back to like, "Oh wow, there's tests, that's great! Test to that .BAT. Oh, there's not enough tests." I continually have that feeling.

Robert C. Martin: This is something that the community owes a great debt to Kent Beck and Ward Cunningham for seeing the whole notion of test driven development and the drive towards massive amounts of unit testing and acceptance testing and it makes a huge difference, it's an enormous difference in the quality of code and the ability to get into code and manipulate it and change it. When you have tests, you're not afraid. If you've got a big wad of code out there and there's no test, you're scared to death to touch it and you know you're going to break things in there but if you have tests, you can reach into it and fiddle with it and pull on it and push at it and tweak it, run the tests on, I passed, I didn't break anything, as long as the tests have enough coverage.

Scott Hanselman: You know there's an interesting discussion I got into with some people about dynamic languages versus statically typed languages and an individual said that the compiler was just a unit test. Once you accepted the fact that the compiler was just a unit test, then the whole argument of dynamically typed versus statically typed didn't matter anymore.

Robert C. Martin: The argument between of static versus dynamic is an interesting one because the static typing creates a set of very rigid

dependencies that tends to make the code inflexible but then it's really hard to segregate your code into a nice set of flexible modules. Whereas with dynamic languages, it is really easy to keep the code as flexible as you want and it's also really easy to make a horrid mess. So you wind up with this double-edged sword, which one would you like? Would you like to be highly disciplined and ordered but you have to live within a rigid framework or do you want to be a little lackadaisical and do some nice flexible things and I, my preference of late is towards the dynamic language. I've gotten very pleased with languages like Ruby, I enjoy them a great deal. I maintain my discipline by writing gobs and gobs of text although I probably still write much more Java than Ruby.

Scott Hanselman: Well let's pound through the last two principles here, as we get towards the end of the show. We've still got to talk about Dependency Inversion and Interface Segregation. If we do it in order, it's S-O-L-D-I, we spell out solid, why is it not spelled out? So you actually never say SOLID, in your article?

Robert C. Martin: No, the word SOLID actually came up later. Michael Feathers wrote me a quick email saying, "You know if you reorder these, you could spell the word solid."

Scott Hanselman: SOLDI, didn't quite roll off the tongue.

Robert C. Martin: No but I was looking for an acronym, he just found it for me and then we've since found some pictures. So if we do it in order, if we do it in the original order, then the next rule is the Dependency Inversion Principle.

Scott Hanselman: Right.

Robert C. Martin: And this is almost a restatement of the Open/Closed Principle except from a 90degree rotation. The Open/Closed Principle states the goal, make things so that you don't have to modify them and you can extend their behavior. The Dependency Inversion Principle says, "Don't depend on anything concrete, depend only on things that are abstract." Now if you think about that, those two are roughly the same. The Open/Closed Principle is achieved by having derivatives depending on base classes and making sure that all of your volatile code is in the derivatives and your non-volatile code is in the base class. Dependency Inversion Principle just says that the opposite way around, "Okay, make sure that all of your dependencies point at things that are abstract." The Dependency Inversion Principle goes to a pretty far extreme, it goes on to say, "Don't depend on anything concrete, ever. If you call a function, it should be an abstract function, if you're overwriting a function, make sure it's pure. If you're calling an object or you're holding reference to an object, make sure it's a reference to an abstract base



class of the object. Never touch anything concrete." Obviously, you can't do that, there are some code that....

Scott Hanselman: Yeah, I mean that's pretty, I keep saying Draconian but sometimes people take this all these principles and they wield them like a stick.

Robert C. Martin: Well, yeah, then that'd be silly, again they're engineering principles.

Scott Hanselman: Right, it's an interesting that I mean, you work with groups and you go in and consult with groups but when you're a member of one of these groups and someone comes in and drops some principles on you that maybe, you're unfamiliar with and then leaves. The people that drank the deepest of the Kool-Aid then start to have these principles, basically, they, "Thou shalt not kill. Remember the commandments." At every turn. How many grains of salt should we use when we're applying these principles? How important are these? Are they, is one more important than the other or are they all equally?

Robert C. Martin: No, I think they're all very important although if I had to pick one to, if I had to pick an order, I'd probably say Single Responsibility, Dependency Inversion, Open/Closed, Liskov's, Interface.

Scott Hanselman: Okay.

Robert C. Martin: That was completely off the top but, okay.

Scott Hanselman: Sure.

Robert C. Martin: Dependency Inversion is very mechanistic, it says, "Make sure that you depend on the direction of abstraction. There are certain parts of the code that must violate that but then I have a rule about those certain parts that must violate it, they belong on the right side. So remember, we had this, when we talked about the Open/Closed Principle, we had these two sides, the volatile side and the non-volatile side and the things that violate the Dependency Inversion Principle all belong on the volatile side, the things that change frequently, anything that we don't want to change, anything that's part of the core abstraction of our system, we furiously defend for the Dependency Inversion Principle. I call the other side the side related to Main, the volatile part, that's the side related to Main. Main is the most concrete of our functions and it will create all of our instances and all of the factories and all of the concrete classes for us and it will, then hand off to the abstract part, a set of abstract interfaces and the abstract core will manipulate it as though it were in this fantasy world where everything was abstract. That boundary line that I draw between the volatile

and the non-volatile, actually repeats itself several times in every system so you'll find lots of those boundary lines all through. On the one side, you've got flagrant violations of Dependency Inversion because somebody's got to create these objects and then on the other side, you've got really strong defense of the Dependency Inversion Principle.

Scott Hanselman: Yeah, I mean at some point you've got to draw the line, I mean the framework itself, that does the "newing" has to break some principles.

Robert C. Martin: So we make sure that those go on into a certain place.

Scott Hanselman: Right.

Robert C. Martin: You don't call new things in the normal part of the code, you call new inside of factories or inside of prototype methods or somewhere that's related to the volatile part of the system. We don't want to see any new in the non-volatile part of the system.

Scott Hanselman: That's a very good way to phrase it that makes a lot of sense. So then we, yourself described least important or least important among its peers here, the Interface Segregation Principle.

Robert C. Martin: Yes, this is a, it applies to very fat classes, classes that have hundreds of methods and many, many clients and you get into a situation where you'll have populations of the clients that use subsets of the methods of this class. So to make this simple, let's you've got a big class, call it FAT and it's got a bunch of methods in it and there's a group of clients over here that calls the first three methods and another group of clients that we call the last three methods. Now imagine that one of the first three methods changes its signature for some reason, we've got to add an argument to it.

Scott Hanselman: Okay.

Robert C. Martin: The population of clients that does not care about that method is still forced to be, to change because the class itself changed. This was really important in C++ where a #include was involve and the #include caused a recompile of everybody that depended on it. It's also important in Java and C#, although not quite as important, some people play a funny game where they change the signature of a method and then the other files that don't care about it, they just don't bother to recompile them we've found everybody who did that and took them out in the back. In general, if a source file changes, you have to recompile everybody who depends on that source file even if nothing they really cared about changed. Now the way we can avoid that from happening is by taking each of the clients and



creating an interface that contains only the methods that they care about and if we had 10 clients then we create 10 interfaces and each of those interfaces would have a different set of methods in it and all of the methods in there would be abstract and then the FAT class would inherit from all 10 of the interfaces. And if there were duplications in the methods, they would just be grouped together through that interface and once you've done that, no class depends on any more methods that it ever needs it simply depends on only the methods that it needs and if a change happens, it changes to the interface, the clients of that interface, obviously have to be recompiled but nobody else does and so all of the other clients are safe.

Scott Hanselman: There's an elegant pragmatism to that. For some reason I always feel like, when I say pragmatic, when I'm doing something, "Oh, I have to make it pragmatic decision." I always feel like that's a difficult decision or the wrong decision and I just apply the word pragmatic to make myself feel better about it but for something to be both elegant and pragmatic feels really, really good, that makes a lot of sense.

Robert C. Martin: Oh and generally, you don't want to depend on something that you don't use. Here you've got a class with a whole bunch of methods in it and you don't call most items and yet by hanging a dependency on them you're depending on those methods.

Scott Hanselman: The number of interfaces, the sheer number of interfaces, the specialization, applying the Single Responsibility Principle, not just to classes but also to methods and to interfaces will definitely cause a relative proliferation of new things, within new interfaces, new methods, new classes but like you said, they'll each be specialized and basic and simple. They'll be more of them but they will be specialized.

Robert C. Martin: That's certainly true, especially in languages like C# and Java and C++.

Scott Hanselman: Right, which is true.

Robert C. Martin: Now think of a language like Ruby, in a language like Ruby, you never depend on anything more than the method you're calling. The methods declared in a class are not part of what you depend upon when you send a message to an object. In fact, when you send a message to an object, you don't know what class it is and so dynamic languages automatically obey the Interface Segregation Principle and obey it in the most extreme way possible. There is no way for them to depend on any more than the methods that they actually call.

Scott Hanselman: Well that is an excellent way to end our discussion of the SOLID Principles or demystification. Uncle Bob, I really appreciate you

taking this much time as you did to sit down and chat with me about these principles and folks can learn about them at objectmentor.com, you can poke around the site there. There's also blog.objectmentor.com and if you go up there, you could see that Uncle Bob, as well as his cohorts Dean, Brett, Michael and Bob are all on Twitter so you can drink in as much of the wit and wisdom of Uncle Bob as you can possibly take on one sitting. Well, thanks so much for your work in the last many, many years, you've been in the business for so long and we look to you and your cohorts with a lot of respect and we really appreciate everything that you offer to the community.

Robert C. Martin: Well, thank you. I appreciate that and this has been a lot of fun, so maybe we'll do it again sometime.

Scott Hanselman: All right, well this has been another episode of Hanselminutes and we'll see you again next week.