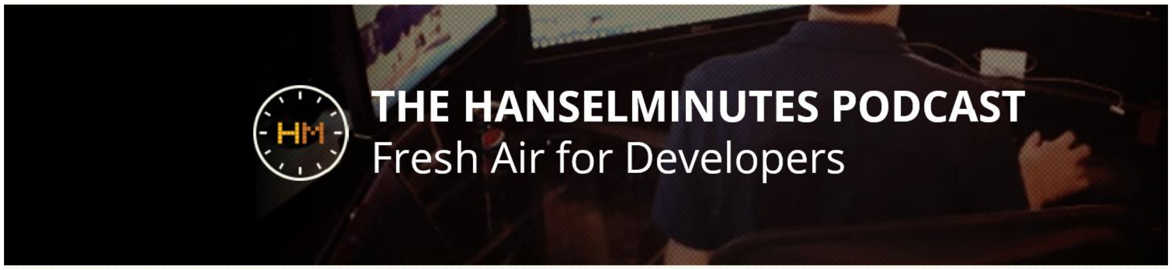

Text transcript of show #396

November 8, 2013

Bugs Considered Harmful with Douglas Crockford

Scott is at the AngleBrackets conference in Las Vegas and sits down with Douglas Crockford. Douglas is the author of "JavaScript: The Good Parts" as well as the discoverer of JSON. What do we need to do to be better developers? Is it better tools? Better attitudes? More discipline?

Our Sponsors

telerik
deliver more than expected

http://www.telerik.com

Lawrence Ryan: From www.hanselminutes.com, it's Hanselminutes, a weekly discussion with web developer and technologist, Scott Hanselman. This is Lawrence Ryan announcing show #396, recorded live Thursday, October 31, 2013. This episode of Hanselminutes is brought to you by Telerik, offering the best in developer tools and support, online at www.telerik.com. And by Franklins.Net, makers of GesturePak, a powerful gesture recording and recognition system for Microsoft Kinect for Windows developers. Details at www.gesturepak.com.

In this episode, Scott talks with Douglas Crockford, author and discover of JSON, about bugs, discipline, and how to write better software.

Scott Hanselman: Hi, this is Scott Hanselman. This is another episode of Hanselminutes. I'm here at the AngleBrackets Conference in the MGM Grand at Las Vegas, and I'm sitting down with Douglas Crockford, distinguished architect at PayPal and the discoverer of JSON. How are you, sir?

Douglas Crockford: I'm great. Thank you.

Scott Hanselman: We were chatting a little bit earlier, and I think I was ranting a little bit. But you made a really fascinating comment about the lack of intellectual curiosity in our industry. Can you talk a little bit about that?

Douglas Crockford: Yeah. Most of the people who do what we do are not aware of where this stuff came from, not aware of their own craft. We have no sense of history, really, in software. Part of that is because it turns over so quickly, but part of it is that we've forgotten all this important knowledge. There's this…I like the way we teach physics. The first course of physics is a history course where we look at Archimedes and Galileo and Newton and everybody, and how each of them enhanced our understanding of how the universe works. It's told in terms of their stories and their personalities, because what they did was important. We don't do anything like that for our own field. It's as if this stuff all happened spontaneously, or there was this inevitability that it couldn't have been any other way. But that's not how it was. The story of how we got to where we are is a really interesting one, and it's worth studying if only to recognize the consequences of the choices that we made which, in some cases, caused us to forego things which may have been better, and which we may still end up having to backtrack to get to.

Scott Hanselman: Is that because there's a large number of developers that did not have a formal Computer Science upbringing, for lack of a better word?

Douglas Crockford: That wouldn't help because the computer scientists don't get this stuff either. Just generally there's too little history in our stuff. We're not completely without history. There are places like the Computer History Museum in Mountain View, California, which is brilliant, and everybody should go there. And there are some collections of material like…one of my favorite websites is BitSavers.org where you can find manuals for all of these old machines that are completely forgotten. It turns out there is brilliance in that stuff, and, even though we don't pay any attention to it, we don't celebrate it the way we should. There are still brilliant ideas in there to be mined. I sometimes think of myself as a software archeologist and I'm trying to uncover the wisdom of the ages. And the ages only happened a few decades ago, but it's still important stuff.

Scott Hanselman: What should we do, though, in teaching the next generation? I mean, I have young children, and they're playing on a Raspberry Pi. Should I get them a Commodore 64 or teach them about punch cards?

Douglas Crockford: I think it is important to teach them about punch cards because there are consequences in that system that came over. I would teach them how typewriters used to work because there is an enormous amount of typewriters which survive in our modern machines, and it's all things which were a good idea at one time, a long time ago, back when typewriters were driven entirely by the energy that you could transmit through your fingertips.

Scott Hanselman: Mm hmm.

Douglas Crockford: And there were design decisions that were made then which are completely unnecessary now, but we still keep it in our systems, and we still see it in ASCII and Unicode.

Scott Hanselman: Yeah, why a code for a certain letter is a code. There were physical reasons for that, and that remains today.

Douglas Crockford: Yeah, parts of ASCII were optimized for teletype machines in order to make the decoding with signals that would drive the little doo-dah that had the letters on it.

Scott Hanselman: The little striking ball.

Douglas Crockford: It was a cylinder.

Scott Hanselman: It was a cylinder? I'm thinking of the Selectric, I guess.

Douglas Crockford: The Selectric…there's a parallel story. So, IBM was using EBCDIC and that was a completely different code set. But, yeah. That stuff's important.

Scott Hanselman: I always find it funny, the whole carriage return/line feed arguments as I guess Windows has picked one and Linux has picked another, and Mac has picked another. And all of this was whether or not we could feed the paper and then return the carriage.

Douglas Crockford: Right. That comes directly out of the teletype, that one code would return the carriage and the other would roll the paper, and the old teletypes were so slow that they really needed two codes because it took a while for the carriage to get returned, and, while that was happening, you could then roll the paper.

Scott Hanselman: Aaah.

Douglas Crockford: So having it as two codes allowed you to put in that delay without actually having to put any logic in.

Scott Hanselman: See, I find that stuff fascinating, but are computer scientists who are being taught by professors in school, or whether they are on-the-job trained, not being taught this because there's no perceived value?

Douglas Crockford: Perhaps. But then we see it go wrong in other places. Like what should be the code and a line? There was an argument about that many years ago, and some people said it should be carriage return. Other people said, no, it should be line feed, and some people (probably not the smartest guys in the room) said it should be both because that's how it's always worked – you know, the traditionalists. The carriage return guys are looking at it from the human point of view: when I want to end it, I hit the return key. The line feed guys were thinking of it from the printer's point of view: when the printer wants to go to the next line, it does a line feed. And the committee couldn't agree, so the way the internet works today is you have to have both. That's the mutually disagreeable compromise, that the guys who wanted a carriage return didn't want the line feed guys to get it because that meant they were wrong, and vice versa. This way, everybody's wrong.

Scott Hanselman: And, to this day, this remains the #1 issue that hits me on a day-to-day basis when checking in and out from Git, of repositories that are on a different system than mine. And I'm thinking about this several times a week.

Douglas Crockford: Yeah, and it's completely unnecessary, but it's because we didn't understand this thing. So going forward, understanding mistakes that we made historically gives us a way to think about how to make things correct going forward except we tend not to learn the history, and so we tend to repeat the mistakes.

Scott Hanselman: Sometimes we learn early programming languages in school, and they think that that's historical context. They'll say, "Well, here. We're going to learn Assembler," or, "We're going to learn OCaml or Haskell," and then that's going to be our entrée into some newer…whatever the new shiny programming language is. Do you think we need to connect software to hardware more? Should we be talking about soldering and physics and electrons at that level and then work our way up to software?

Douglas Crockford: I don't know that we need to go that far down, but I think it is important to understand how instruction sets work because ultimately that's where software is grounded. Everything else is transformations or abstractions on top of that. Many of our languages were very strongly influenced by the kinds of machines they were going to be resting upon. But modern machines are architecturally quite different from those, but our languages are still in the old paradigm. There are a lot of things which we do in our languages which, I think, work against what should be our primary goal of writing software that works well, that has no bugs in it. And very often I see programmers pursuing other goals which are incompatible with that one, and so that becomes subordinate rather than primary.

Scott Hanselman: So, you're saying that our head is not right. Our priorities are wrong. Our goal is to write software that has fewer mistakes.

Douglas Crockford: It should be, but all of our tools tend to be working against us. Let me tell you about a bug I made.

Scott Hanselman: Okay.

Douglas Crockford: I wrote a JSON library in Java many years ago as a reference implementation. In that library, I had a variable called `Index` which was counting the number of characters into the text that was being parsed. It was used primarily for error messages; so, if there was a syntax error it would say, "At this location, we found a problem." I made it an `int` because that seemed to be the right thing. An `int` allows you to have a text that is up to 2 gigabytes in size which, at that time, was still a pretty big disk drive. Today, it's not. Today, it's RAM, right, which is wonderful. Moore's Law has been working for us. But at the time, I couldn't imagine anyone would ever have a text bigger than 2 gigabytes. Last year, someone wrote to me and said that they had a text which was considerably bigger than that and that they had a syntax error past 2 megabytes, and so my package misreported where the thing was.

Scott Hanselman: So they had a JavaScript file that was larger than 2 gigabytes?

Douglas Crockford: They had a JSON file.

Scott Hanselman: A JSON file of more than 2 gigabytes?

Douglas Crockford: Yeah. Which amazed me because, when I designed JSON, I didn't expect there to ever be a JSON text bigger than a couple of K. Now I intended it as a message-passing thing to go over the network, something highly interactive. You can't anticipate all the future uses of the things that you do.

Scott Hanselman: Right. But, if you had set it to 2 terabytes, 5 years from now someone would send you about their multi-terabyte JSON file.

Douglas Crockford: Yeah, I'm not thinking that's going to be such a problem.

(laughter)

I asked the guy how long did it actually take to parse that, and it took a while.

Scott Hanselman: Okay.

Douglas Crockford: So Moore's Law is not working that well. I don't think I have to worry about that. If I had made it a `long`, it wouldn't have been a problem. So it's nice that it took over a decade for this bug in my program to be revealed, but, looking at why it was a bug, it was because Java gave me a choice of six types that I could choose, and by giving me that choice I could choose the wrong one. In this case, I did. So this is a trap that the language is setting for you. The bargain it's suggesting is you can save memory by picking the smallest possible type that will contain your value, and the flip side of that bargain is that – and if you get it wrong, if you pick one that's too small, then disaster happens. There's no reason to be making that trade-off. Many years ago, it did. If you were programming the Atari 2600, the VCS, that machine had 128 bytes of RAM in it and that included the stack. It included all the buffers, the OS – everything fit in 128 bytes. So on that machine, you really paid attention to how big a variable was. But we've have a lot, many, many doublings of memory capacity since then. We know have gigabyte memories – it is not worth our time to try to figure out what the right type for a variable is.

Scott Hanselman: But every time we sit down to write even the most trivial of programs, we type `int`, and I go, "Do I need an `int`?" And then the processor power that requires my brain to make that decision…

Douglas Crockford: Yeah, should that be a `short`? Should that be a `byte`?

Scott Hanselman: Or maybe a `uInt`?

Douglas Crockford: Yeah, I can get one more extra bit if it's unsigned – which is a total waste of time. Just measuring our own productivity, there's no reason to be doing that. And then it gets worse because there's this opportunity to have overflow. The way numbers overflow has a lot of history to it. There are a number of models for what should happen if you try to put too big a number into a variable. One says you should get an interrupt, you should throw an exception, which would be reasonable. Another says you should get NaN or some other signal that says that's not an adequate value, and that's okay, too. Some systems, it saturates. It gets clipped to a maximum possible value. That's really good for signal processing and other applications. Instead, we do the worst possible thing which is we flip around to a maximally negative number but, because it depends on other aspects of the computation, we can't even predict what that number is going to be, so you can't test specifically for that value. It's the worst possible thing. And the reason we do it that way is because, in the 50s when they were starting to figure out how to build CPUs, someone figured out that, if we put in a complement and use a complementary notation, then we don't have to implement subtract. When we want to subtract, we'll simply complement and add and ignore the overflow. And that works.

Scott Hanselman: So someone micro-optimized?

Douglas Crockford: Well, at the time, it was a reasonable optimization because gates were really expensive, and also every gate increased the unreliability of the machine because they were built out of tubes. As much as you could simplify the architecture, that made it more viable and have more up time. So it was a good idea at the time. But then it got baked into FORTRAN, and there was an expectation that, in order to be portable, we want to keep this crazy thing going on, and then that got into C where it's even worse, where people expect to get a free modulo operation if they intentionally overflow a variable. So we're stuck. It's in the architecture. That compounds the trap of having too many `ints` to choose from. In JavaScript, we only have one number type which is great. I think JavaScript is actually smarter than most of the other languages in doing that. Because there's only one number type, you waste no time thinking about what type to use and you can't ever pick the wrong one. And yet, JavaScript is under tremendous pressure from the community to add `ints` to it because people want this behavior, because they have – prior to their experience with JavaScript – maybe never known another language in which you didn't have the option of having `ints`. So they want to dumb it down. It's difficult for us to see that JavaScript is actually the future. It's where we want to be going. The thing that JavaScript got wrong was that it chose the wrong one number type. It chose binary floating point; it should have picked the decimal floating point.

The idea that it was just one that was big enough to hold all the numbers that you could ever do was correct.

Scott Hanselman: It's fairly forward thinking given the context.

Douglas Crockford: It was. And it was done in that language because the language was originally intended for beginners. In order to make it easy for them, let's just give them one kind of number and that's all they do. The original BASIC did the same thing, so it's not a new idea. Microsoft kind of screwed BASIC up by adding `ints` of various forms eventually, and so it made it possible to waste time micro-optimizing and it also made it possible to introduce more errors. All of our other languages do that, too.

Scott Hanselman: It's interesting, though, whenever I think of things I don't like about JavaScript, numbers are not my problem. Dates are where my pain lies in JavaScript.

Douglas Crockford: JavaScript's dates were based on Java's dates. Yeah. I'm not aware of anybody who's ever gotten dates right.

Scott Hanselman: It's all the…it's the serializing. It's the string formats for dates that gets so frustrating with time zones and the **z** in the middle and the **t** and getting all of that just so.

Douglas Crockford: Well, that's not JavaScript's fault. That's ISO's fault.

Scott Hanselman: (laughter) That's a valid point.

Douglas Crockford: JavaScript is just implementing what ISO did, and I think for its problems, the ISO format is a big step forward because it's at least a single format that we can all argue about.

Scott Hanselman: That's a good point. So, you're saying that the tools are kind of working against us. Is it we that need to have more discipline?

Douglas Crockford: We do. One thing we can do is demand programming languages which have fewer affordances for going off the rails. We tend as a community to do the opposite. We want more ways of things to get wrong.

Scott Hanselman: You're saying that you want the programming language to give us less rope?

Douglas Crockford: Yeah. I want it to be simpler. I want it to be more constrained. I want it to be harder to get things wrong. I want it to be simple, because the biggest problem we're dealing with is managing complexity. That's most of what we do. Complexity is complicated, and so having complexity in our languages actually doesn't help us. There's no conservation of complexity in that, if we put more complexity in the language, then our programs get simpler?

Scott Hanselman: (chuckles)

Douglas Crockford: That doesn't happen.

Scott Hanselman: Maybe we'll just put a complement in, and it will flip the whole thing over and then it will just become simple.

Douglas Crockford: Yeah, unfortunately….

Scott Hanselman: It doesn't work that way.

Douglas Crockford: Simple isn't the complement of complicated. I'm a minimalist in that respect, so I want our languages to be as small as possible. Having a lot of features in the language is really attractive to us emotionally but doesn't actually help us. One of the things that a lot of programmers get wrong is they think their job is to identify every feature of the language, master it, and then demonstrate that mastery in everything they write. And that turns out to be an extremely bad practice. What I would recommend instead is: figure out what parts of the language are most reliable, are most consistent with being able to write correctly, and use only that.

Scott Hanselman: That is really interesting, because I'm thinking to myself about the thing that feeds programmers that makes us move forward: new features bolted onto the side of a language, whether it be LINQ in C# or new ways that…there are even new features that are being added to JavaScript…and then they say, "Ah! Now I have another tool in my tool belt!" we like to say, but we never seem to celebrate the removal of tools from our tool belt.

Douglas Crockford: And removal is really hard because the problem with the things that you want to remove isn't that they're useless; it's that they're dangerous. And not dangerous in that they're going to blow your hands off but just they increase maybe slightly the likelihood that you're going to make mistakes. It's easy to get into arguments about that stuff. "I used one of those dangerous features but, in this particular instance, I think I'm using it correctly and it is not an error," and that's likely true, but I think it is still better that you had not done that and written it the simpler, more reliable way instead.

Scott Hanselman: Are these like known factual things where one could say, "I have removed a feature from a language," or "I have made this language minimal" and this is mathematically true that it is better, or are you describing a language that is a very opinionated language where lots of people with opinions, the internet, are going to say, "No, that's nonsense! That guy's wrong!"

Douglas Crockford: Well, these arguments are almost always exclusively emotional, and I used to make those same arguments. The thing that changed me was developing JSLint which a linting program for JavaScript. That program turns out to be a lot smarter than I am about programming languages because my intuitions were all wrong. I didn't develop that program intending to enforce my opinions on other people. It told me what my opinions should be, that, if I want to be able to statically analyze a program and help determine the presence of bugs in it, I have to avoid certain features because those features will tend to mask bugs or are undecidable in static analysis as to whether they're being used correctly or not. So, whenever I found there was a feature which has one of these confusing aspects to it where it's easily misused and if there is a different feature which does the same thing but doesn't have that ambiguity, then use a good one and never use the bad one, and then you don't have to decide it.

Scott Hanselman: Mm hmm. At the beginning, before we talked, you said that you were not the inventor of JSON but rather the discoverer, and then you described yourself as a software archeologist. When someone is talking about Michelangelo pulled David out of the stone – like he was inside of this rock and I had to just let him out – it uses a different part of the brain to discover that the program wants to be a certain way than creating the program from your own hands.

Douglas Crockford: I've used the Michelangelo metaphor in describing what JSLint does…

Scott Hanselman: Oh, yeah?

Douglas Crockford: …and how I came about it. Yeah.

Scott Hanselman: Wow.

Douglas Crockford: There is an ideal language trapped inside of JavaScript…

Scott Hanselman: (laughter)

Douglas Crockford: …and I just had to chip away the bad parts.

Scott Hanselman: Is that using a different part of the brain? Is there a whole class of developer, a whole generation as it were, that's maybe not thinking about it right?

Douglas Crockford: Yeah, we tend to be very emotional about syntax in the same way that the fashion industry is emotional about fashion. They'll argue about should there be pleats or not, should there be cuffs or not, things which make no sense to us, right? How high should heels be?

Scott Hanselman: Software pleats. I'm trying to think of different language features that I would describe as software pleats.

Douglas Crockford: We are so into that so, that part of the brain, we don't use it for fashion. We use it for language features.

Scott Hanselman: Because we have no fashion sense as we sit here in our sneakers and our T-shirts.

Douglas Crockford: Exactly. We think that, because we are the ambassadors to the computer, which is this ultra-rational machinery, we imagine ourselves also to be ultra-rational and we're not. We are deeply emotional about things like syntax, and that's one of the reasons why JSLint makes people cry, because it's telling you that you can improve your program by doing these things. Our #1 objective, because we want to be writing good programs that are free of error, we should go, "Great! I needed that advice." But instead, no, we become really defensive and emotional and say, "No! You can't tell me to do that. You have no right. I have a right to use this feature." Someone once complained to me that he should be able to write his programs in excrement if he wanted to in order to express himself as an artist. So he was telling me that he has a right to write programs which are literally crap.

Scott Hanselman: (laughter)

Douglas Crockford: I think we should have a higher purpose than that.

Scott Hanselman: I've said before that I felt that sometimes programmers – and I have a tendency, and it's probably not a good habit to refer to younger programmers because, now that I'm becoming middle-aged, I'm thinking of myself having moved from one generation to another – have forgotten why it is exactly that we are writing software. It's not necessarily an intellectual pursuit. It's not necessarily the pursuit of poetry or haiku; I know that some Rubyists consider themselves to be trying to write a perfect program that is haiku. Ultimately, we're trying to solve a business problem, presumably, or for academics, try to solve an academic problem. So correctness seems to be the thing we should be striving for.

Douglas Crockford: Correctness and maintainability. And anything that we do that works against those is counterproductive, at least.

Scott Hanselman: Why do we spend so much time on opinions and white space and where the curly braces go and the semicolons?

Douglas Crockford: Because we literally don't know what we're doing.

Scott Hanselman: As an industry…

Douglas Crockford: As a community. Yeah. We are very smart people. You have to be smart in order to do this work. We'll give ourselves points for smartness and give ourselves passes on everything else.

Scott Hanselman: What's our takeaway? How do I program better? Do I need to change the way I think?

Douglas Crockford: Yeah. You need to keep in mind that approaching perfection is the most important thing you can do, and that perfection is hard particularly for deeply flawed human beings like us. I am a deeply flawed human being, but I'm a programmer, and I'm pretty good at it. But it's not a natural thing, and I do it because I enjoy it. I find it pleasurable, but it requires a lot of discipline. One of the things that we tend to do is ignore our mistakes. We'll go down into the debugger and we go down into this cold, hurtful place where we do that work – I call it the abyss. Normal people cannot do that. They cannot go down there with the confidence that they're going to come back alive. We do it all the time.

Scott Hanselman: Yeah.

Douglas Crockford: And when we come up, there's this exhilaration. "I did it! I went down there, I found the bug, I killed it, and I survived. And I'm back!" In this euphoria, we have an amnesia. We forget that we did that. So, as a result, we tend not to learn from our mistakes. "Oh! I did that again." Hit on the head, "I forgot." It's going to happen again. We do this all the time.

Scott Hanselman: Yeah. Yeah. I've talked about developers having 20 years of experience but it's unfortunate if it's the same year 20 times.

Douglas Crockford: We do that a lot. There's a lot of Groundhog's Day in the way that we work. One thing you can do is, every time you make a mistake, write it down. Keep a bug journal.

Scott Hanselman: Really?

Douglas Crockford: Yeah. You need to develop self-awareness of how you make mistakes.

Scott Hanselman: You set that up as a bug journal?

Douglas Crockford: Yeah.

Scott Hanselman: I like that. What would I write about the bug? How would I describe it? The algorithmic issue that came up?

Douglas Crockford: Yeah. Donald Knuth wrote a brilliant article about the bugs in TeX. He kept a bug journal as he was developing that program, one of the biggest programs he wrote in his career. One of our best programmers. He made lots of mistakes, and he categorized them. He developed letter codes indicating the different kinds of things. The typos were distinguished from the algorithm errors and so on. He sorted it and produces the results. It's a really good article. The important thing isn't his findings about what kind of bugs were common, it's that that as an exercise is so useful because we tend to not learn from our mistakes, and the way you learn from them is by paying attention to them.

Scott Hanselman: Yeah. You can't cut something unless you've been measuring the whole time.

Douglas Crockford: Oh, that's the other thing we do. We do a lot of cutting without measuring. One of the things that drives us instead of correctness is performance.

Scott Hanselman: Micro benchmarks for all!

Douglas Crockford: Performance is important, but it's less important. It's more important that it be correct. So there's this thing that we tend to do which is micro-optimizing, trying to cut time out of code even in code where it absolutely doesn't make any difference.

Scott Hanselman: Right.

Douglas Crockford: Our machines are so fast that, unless you're in the middle loop of something that's going to take a long time, optimizing is simply a waste of time. Generally, in any project, we have a limited amount of time in which we should do optimization. We need to be really careful where we do that optimization. You need to optimize your optimizing.

Scott Hanselman: Yeah. It's amazing how many times perfectly rational people will put a for loop from 1-10,000 around a piece of code that just doesn't matter when the network latency is an order of magnitude more than anything you could ever do to that for loop.

Douglas Crockford: Yeah, the first law of optimization is look at where the cost is. For us, that usually means where is the time being spent. Optimizing anything which isn't the principal place where the time is being spent is a waste of time. But we do that all the time, and it works against us in that doing those kinds of optimizations, you remove generality from the code. By removing that generality, you make it harder to test; you make it harder to maintain; you make it harder to be confident that it's correct. So we're actually working against what should be our principal goal. Sometimes you have to optimize and sometimes you have to do terrible things, but you should measure carefully before you do that to be sure that, in fact, you even have to.

Scott Hanselman: When we were speaking earlier, you told me a story about the `go to` statement and how developers argue and make their arguments understood and how ineffective arguments can potentially have decade-long ramifications. Could you talk about the `go to` statement?

Douglas Crockford: Yeah. So, in '68, Dijkstra wrote a letter to the communications of the ACM which was titled *Go To Statement Considered Harmful*, and in it he describes `go to` statements as being disastrous and that we should get rid of them. It turned out that getting rid of them was a good idea, but they weren't disastrous, that there were good programs that had `go to`'s in them that worked fine. So his charge that, simply by having `go to`'s, that that was causing destruction was not true. And so there were a lot of people who were emotionally invested in the `go to` statement who thought they were being completely logical about their defense of the statement, and they looked at Dijkstra's claim of destruction and said, "Well, that's obviously not true. Therefore, his conclusion is obviously not true." Dijkstra was trying to change a paradigm. In retrospect, it doesn't look like a paradigm shift because we're just getting rid of a statement which, at that point in time, we didn't need any more. There was a theory that, if you add `else` and `while` to a programming language, most of the needs for `go to` go away. But not all of them said the defenders of the `go to`. They come up with lots of other arguments like, "You're saying that we cannot tell when `go to` is appropriate or not, so you're going to take it away from us. That is an insult to us, that you're challenging our competence and intelligence, and you're wrong about that because we are competent and intelligent people." They made arguments about tradition; "We've always had `go to` in our language. We're comfortable with it. It works for us. We're making livings writing with `go to`. You cannot take it away from us. You can't destroy our livelihoods." They made arguments about personal expression: "We're artists, and we express ourselves with our `go to's`, and you can't take that away from us." They made the argument that, "Nobody wants the `go to` statement taken away," which was ridiculous because the reason they said that was because someone had suggested that very thing. So saying, "Nobody wants that" is obviously wrong. There were some other guys who weren't quite as mathematically challenged who said instead, "The majority of us want to keep the `go to`," which is less obviously wrong, but they said that without any data to support it. And they continued saying that even when they were no longer the majority. And all of that completely missed the point, that it doesn't matter what the majority wants. The argument should have been: We're trying to write programs that have fewer errors in them, and it's suggested that, by getting rid of this statement, we'll accomplish that. Is that something we want to do? Is that a true claim or not?" That's what the argument should have been. Instead, it was about all those other things which are ultimately all emotional.

Scott Hanselman: Right. I think that that's an important lesson for everyone to remember, that you are not your code. And, if you take an argument about someone's code, even a friend or someone you work with, and it starts with something like disastrous, a big specious declarative statement about how your code sucks, then it's going to go completely off the rails.

Douglas Crockford: Yeah. Dijkstra started the argument in a really bad way. Another thing he said was, "We've observed that programmers who use lots of `go to's` are inferior to programmers who don't." What he could have said was, "Programs that have lots of `go to's` are inferior," but he didn't. He made it personal in the first sentence of his letter.

Scott Hanselman: Wow.

Douglas Crockford: It's just really arrogant, right?

Scott Hanselman: Yeah.

Douglas Crockford: Alan Kay said that "Arrogance is measured in nano-Dijkstras."

Scott Hanselman: (chuckles)

Douglas Crockford: We should love the coder but hate the code, and that should have been his argument but he didn't. He made it personal, too. So maybe it's not surprising that they countered with personal arguments.

Scott Hanselman: Yeah. I am not my code. I try very hard to make sure that other people do not feel that they are their code either, but it's hard. In the world of pull requests and Twitter, it's very easy to take these things personally.

Well, thanks so much for chatting with me today. I'll put in the show notes links to your Google+ and your website, and things that you've worked on as well JSLint. Thank you so much.

Douglas Crockford: Thank you.

Scott Hanselman: This has been another episode of Hanselminutes. We'll see you again next week.