**Shane McGraw:** Hello. Welcome to today's SEI webcast, "Busting the Myths of Programmer Productivity." My name's Shane McGraw, Outreach Team Lead here at the SEI, and I'd like to thank you for attending. We want to make our discussion today as interactive as possible, so we will address questions throughout today's talk, and you can submit those questions in the YouTube Chat area, and we will get to as many as we can.

Our featured speaker today is Dr. Bill Nichols. Bill's a researcher in the Software Engineering Measurement and Analysis Group here at the SEI. During his time here he has taught the Software Measurement and coached both software developers and managers to use quantitative data and managed software development work.

Now I'd like to turn it over to Dr. Bill Nichols. Bill, good afternoon. All yours.

**William R. Nichols:** Oh. Good afternoon, and thanks for coming. What I want to talk about are some of the things we've learned about programmer productivity. I'll give you little bit of background in why I think it's interesting and why we have something unique to say about this subject. So oh.

So, let's start off with how huge variations in programmer productivity have been documented again, again and again. The oldest ones go back to the 1960s, and they've consistently shown productivity ranges reported from 5 to 1 to 30 to 1. If you want a really, really good summary, just do a web search on Steve McConnell, X10 programmer. He's written some wonderful articles on this and has a very good summary of the evidence, and the point I want to make is that these results have replicated again, again and again, but something was wrong. There's something about all those replications that just wasn't right, and that's what I want to talk about, what we discovered that's really new.

So let's start with why this is important. Why do people care about programmer productivity? I was kind of surprised at the amount of attention the article on IEEE Software got when it first came out, but you think about it, everyone wants to know, where do they stand up? They want benchmarks. I want to be the best or I want to know how I compare to the best. If I'm a manager I want to hire the best. I want to know how I can get the best programmers. If you are in HR or management, you want to know, "How can we train people to make them the best programmers they can be so we can be competitive in this world environment?" You want to be able to compare programmers so you can maintain the best--if you want to keep the best and promote the best, and you want to pay the best. If the best programmers are 10 times better, well, you better be willing to pay to keep them, because they're going to be worth it, and of course, you need to know the planning. For planning purposes, you need to know how long something's going to take. How do you know how long something's going to take and if the programmer productivity varies that much, you better know something about their individual capabilities.

So here are some questions that come up. We have a lot of data. I'm going to talk a lot about data in this presentation, and I hope you can bear with me. I find numbers exciting, but I know not everyone will. I'll do my best to convey some of that enthusiasm.

Do our data replicate the kind of scale of performance ranges that we've seen in these historic data sets? How much does the programmer productivity really vary? That is what is the real range of programmers' capabilities? What factors could explain variation? Is it experience, education? What makes the difference? And as a practical effect, what can you do about it? How big are these differences? What difference will they make and what can we reasonably control?

So background. We got interested in this question because we have a--we just finished a project at SEI called SCOPE, and the idea of SCOPE was we were trying to identify causal relationships in software engineering data. What made this unique was we were looking at observational data. Trying to do experiments, experience randomized, controlled experiments, that is really hard in software engineering. It can be really expensive. It can be impractical. How do you do an experiment with the exact same requirements and get many repetitions? Well, you can't. It's too expensive. You can do it on smaller scales, and some people are doing that, but nothing is large scale.

So with the SCOPE project, we used a technique using a cutting-edge machine-learning technique for causal inference that was sponsored by the National Institutes of Health, and it's being run out of the Center for Causal Discovery, operated by Carnegie Mellon and the University of Pittsburgh. It's actually housed down the street a bit. Now, for data, we used a volume of performance data that we collected while we were teaching a class called the Personal Software Process.

Why is the Personal Software Process data so useful? Well, there are couple reasons. One of them was the structure of the course was well-designed for this kind of purpose, repurposing for this kind of study. The class, in the original form, contained 10 programs. That is we were teaching programmers how to do things like design, personal review. We were teaching some things and we had 10 canned exercises in the form of requirements that the students were expected to perform. Now, they brought their own equipment, they used their own programming language. All they had to do was program the requirements and learn a few things about process as they went through this. But everyone who took the course programmed the same set of requirements in the same order.

Now, that's important because we taught this course a lot of times. For this data sample--this is not the whole thing--but before we changed the course format somewhat, we had over 3,000 developers writing over 31,000 programs, and that is our data set. That is just huge. By software

engineering standards, this is unique. Nothing like this exists. Over three million lines of code, over--almost an eighth of a million hours of work went into this, and we had almost a quarter of a million defects recorded, so data. We have tons of data on this. We also got along the way demographic information from the students who took the course. That is we got some information on where they were from, their age, their education experience, their number of years of program experience, how many lines of code they wrote with this particular program and so forth.

Now, of these 3,140 students, we selected 494 who completed the course and who wrote the programs in C. Now, there's some reasons we made this particular restriction. Mostly it was because we wanted to do real apples-to-apples comparisons and eliminate some of the confounders, and the programming language can certainly make a big difference in principle. So our first look at this was to take the single language that had the most instances. There are some additional language samples. We have fairly large samples of C+, C#, Java, Virtual Basic, but nothing quite as big as that one sample in C.

We're going to look, for the purposes of this study, only at the programming effort. We're not going to look at the program size, the defects. This one's really scoped to look at the effort. Let's see, and the analysis. We're going--we've applied some of these causal search techniques that I described earlier from the SCOPE project, and we also did something called linear hierarchical models. Think of this sort of like an anova on steroids. We also use structural equation modeling, which is related to using analytic hierarchical models but also--it also helps to control for things like confounding and relationships between the variables, and we use this to measure the effect size and look especially at the variation in the samples.

Now, what did we find? When we looked at this data with causal search, and the causal search is able to actually infer causality directly from the data, there are a lot of assumptions that go into it, which I won't go into here. So you have to do it with some care, but you can actually infer some things about causation directly from the data with certain assumptions and certain conditions, and we found, to no one's surprise, that one of the biggest factors in how long it took to write something in software was, "How big is it?" as the size of the requirements made a big difference. But there was also a variation with students. Yeah. Who wrote it? How big was the requirement? Similar kind of finding with defects. We found that the best predictor of defects was actually the amount of effort it took to write the code and who wrote it. When we looked at the total effort, again, requirement size. "How big was the program?" was the biggest differentiator in how long it took to write it, but it also had something that seemed to relate to the individual student.

So okay. So far this is really aligned with that notion that there's a big variation in programmer productivity. But we found some other things too. We found that the difference wasn't just between programmers. There was as huge difference from a programmer on one assignment to

the next, and you could kind of see this if you looked at the data and used the error terms from something like an anova or a sim. If you don't understand what those mean, don't worry. I'm going to show a few graphs to try to make these things accessible.

So a lot of what we did was to get a follow-up to look more carefully at the sources of variation. Well, I have a couple versions of this chart, and let me just explain what we did. Each of the programmers wrote 10 programs, so what we did was we summed up how much total effort they took for all of these programs. How long did it take to write? We called this the relative effort, so we divided the total effort--we divided the individual effort totaled by the average across all students. So the average student is going to be a 1, 1.0. Basically normalized, and that's what we see in the distribution here. So if you look at here at 100, this is in percents. One hundred percent would be the average, so what you see here is a distribution of how many students took some percentage of the average time to complete the total exercise. A total of 10 exercises, I should say.

The average took 100 percent of the average. Okay. Big standard deviation. So the standard deviation was half of 1. That's pretty big, and when you see a standard deviation that big on 1, you know it's probably not going to be a symmetric distribution. In fact, what we see is a big skew. So this is actually kind of log normal, and we see a peak well below 1, and this frequency of programmers who took longer and longer tails off.

So what you'll notice here is that there are few outliers. This is only a few. Most of these were in a fairly constrained range, so up to about here we are two and a half times the average, if you can see the pointer. There were few of these outliers, so there were a few that give you some support to say, "Yeah, maybe if you take a look at some of these outliers there's a 10 to 1 total range," but that isn't really telling you everything you need to know. Look at the middle path. This is called the interquartile range, just the middle 50 percent. Everyone who finished from 25 percent to 75 percent of the average, that's about a factor of 2: 124 percent versus 63 percent. The middle 50 percent all finished within about a factor of 2. Let's widen that. Let's take the real outliers. Let's talk about 1 in 20 on the fastest and 1 in 20 on the slowest. Even there, the range for the top 5 percent and the bottom 5 percent was only 5.

So what we're seeing here is the overall data just isn't supporting the 10 to 1 very well. Unless you kind of look sideways and look at a couple of extreme outliers way out in this tail, it doesn't make much sense. So we wanted to investigate what's going on. What's going on here?

To do that-- to illustrate that point, I made a run chart. So as we go across the x-axis here, we have 10 programs, and what I plotted here now was the relative effort for each program. So the average is always going to be 1. This average in yellow is the--is right around the--well, that's actually the third quartile range, meaning it's going to be at 1, and we can see how this changes how the distribution of the minimum, the minimum time, the maximum time, and the median

here in green, and the 25 and 75 percents, but what you'll notice is that the first four lines here, the minimum through the three-quarters, were all very consistent from one exercise to the next. The real variation is occurring here at the outlier.

So those of you who've programmed more than a little know that there's always that one program you get stuck on. Someone was always getting stuck here, and that's what this is representing. Now, so we can see the range was highly variable, top and bottom range, and the ranges, in fact, showed enormous variation. We were seeing a range from max to mins from anywhere from 20 to even 50. This starts to give us a hint about what we were seeing in all those old experiments. Most of those were done on relatively small samples and programmers seldom did more than one program. So this kind of suggests that they were actually seeing was the maximum and minimum that people were seeing on a single program, but this number is inconsistent with that total I showed you a couple of slides earlier what's going on. I will also point out that the interquartile range was very consistent. It was always in this range of 2, 2 ½.

Well, what is this variation? I'm going to show you a few plots here that are called caterpillar plots, and when I studied these recently, the instructor told me that he had never seen a caterpillar plot used for anything other than a pedagogical purposes to explain the difference between within and between variation. Well, I'm going to use it for that, but I'm also going to use it to show you what that means in real terms.

I'm look--what I did was I took the programmer effort for each program and ranked the programmers. Basically the fastest was a 1, the slowest was a 495, and we did that for each program and each individual. So the rank, the median, their median rank, I plotted along the y-axis. So if someone did a 3, 4 and 5, they got a 5 along the axis. I'm sorry, they got a--their med--start over.

**William R. Nichols:** If someone consistently finished in the top 10, they would have a point along the median axis somewhere around here. If they were consistently slower, their median might be up here, and what we have in these bars here was an estimate of how uncertain that median really was. So someone programs 10 programs. Are they the same? Are they the same rank every time? Well, this particular one--let me pick out one example. This person on one example finished looks like around the top 10. Then on their slowest, they were up here around 300. So this individual, this bar shows this person finished somewhere between 10 and about 300. That's a pretty big range. But on average, this person was, eh, about 200 out of 400, 200 out of 500.

Now, what we found was the top 5 percent of programmers, that is 5 percent who finished consistently or who had a median value in the top 5 percent, actually had a range of performance, so these--this caterpillar plot, if you will, shows what their median, their average median was, and what their range was. So these top 5 percent pretty much finished between 1:100. Think

about that for a minute. The top 5 percent. Five percent of five hundred. Five percent of five hundred is twenty-five, but we were consistently seeing almost all of these finishing anywhere from one to a hundred. The bottom 5 percent also had a range that was a lot wider than you would expect. They typically finished anywhere between 100 and 400. So instead of finishing within 5 percent of the sample, they basically floated--well, some of these are even wider. So they floated between a quarter of the sample and almost half. You see, some of these are pushing down toward the middle of the sample.

Now, what's interesting is let's look at the middling performers. For the middling performers, they were generally finishing anywhere from somewhere around 100 to about 400, and it turned out the average range in where someone would finish in this race--that is if you took this race of 500 people and ran it 10 times, the average person would have had a different place of finish of about 250. That's pretty big. You're not going to get that kind of range if you run a marathon or a sprint. Two hundred and fifty out of five hundred. That's half the pack.

So what we saw here, based on this data, was just this huge variation within the programmer. That is if I'm programmer I run this race 10 times, I could finish anywhere within about half the range of the total pack. That's just enormous and that had never really been seen before. So if you're going to ask me, "Is there really an order of range of magnitude in programmer productivity?" I've got to say that's busted. You were making a mistake of chasing the noise. You weren't looking at where they finished on available so much as you were looking at someone's worst performance versus someone's best performance, and none of them have very consistent performances.

I would say the average programmer range, if we use the 5 percent cutoff, is about a times 5. That's still pretty significant, but it's a lot smaller than times 10 or times 20, the times 30s that we've often seen, and by and large, most of your programmers are going to be within a fairly narrow range of times 2. Now, there will be a few that are consistently faster or consistently slower. No question about that, and caveats on this? No. These are not the most challenging programs. Don't get me wrong. These are not trivial programs to write, but we gave them a lot of instructions. They didn't have to invent anything totally new, and they were exercising features of the language that they were generally familiar with. So we weren't asking them to solve entirely novel problems. These are things that they should've been able to achieve given the instructions.

What else did we find? Well, surprisingly, experience didn't matter. We like to joke, you know, do you have 15 years of experience or do you have that 1 year of experience 15 years over? That is you do this again and again more and more and you just don't get any better. It seems that people's performances as far as productivity on these types of programs, plateaus fairly early in their career. There might've been a small negative effect with experience, but my best guess is that was probably a bias from the best programmers getting promoted out into management. We

found no evidence of effect with experience writing code, nor did we find it with the number, the amount of code that they had written. It just didn't seem to make an effect. We looked at the highest degree obtained. Were they an undergraduate? Did they have a graduate degree, a PhD? No difference.

Now, someone, we thought about, well, these are kind of mathematically, statistically inclined programs. Maybe that has an effect. So we looked at, "Is there any correlation with their exposure and experience with college-level statistics?" Again, no effect. Just didn't matter.

So what we had here, and I'll talk a little bit more about the math, is what we call the intracluster correlation. That means how much does a group look like itself? The stronger the intracluster correlation, the more individualistic that group is, and in our case, the group was 1. If I have an intracluster correlation of 1, my work is completely predictable. What we found here was about 40 percent of the total variation was actually within the developer. Not between the developers, but actually within the developer, and that accounts for that big difference between some of those historic measures and what we were seeing, and that's a whole--that's an entirely new observation. To my knowledge, no one had ever really measured that or even thought about it much before in terms of running software engineering experiments.

Now, there is still a significant gap in the min-max range, but on any given point that min-max range would be huge. Overall the min-max range is really going to be kind of narrower, and we already said, it's going to be maybe about a times 5 factor between your best programmers and your worst. Within the middle of the pack it's going to be about 2. It's going to be about 2 between the middle and the best, and about 2 between the middle and the worst. That's just about where it comes out. It tells us that if you just take a measure one time of someone's productivity, any number of things can go wrong, and those of you who've done programming have a pretty good idea of the kinds of things that can go wrong when you're writing a program. That bug that you either see right away or that you obsess over because you just can't see it. You don't find that bug until someone comes over, looks over your shoulder and says, "Hey, did you realize that you forgot to put the decimal point on that 2 and it's not being treated as a floating point?" Amazing how often that happens.

**Shane McGraw:** Bill, can we work in some audience questions here?

**William R. Nichols:** Sure. You know, my Chat window has vanished again, and I don't know how to get it back.

**Shane McGraw:** Okay. No problem. I'll just read them off for you and we'll take couple minutes and we'll turn back to you. So first of all, I just wanted to thank, again, the great attendance we got today. We have a worldwide audience. Saw people from Mexico, Costa Rica, Germany, Chile, Ireland. So we just want to thank everybody for taking an hour to spend with

us here today. So first question, a question and comment from David asking, "How about the productivity in the sense of what programmers can do and what they enable?" and his comment was, "One programmer can design complex data structures and write hard-to-implement, low-level functions that manipulate them. Others write programs that can use these functions and data structures. The first group makes the second productive."

**William R. Nichols:** That's a great question, you know, and our experiment, our data, doesn't really speak to that problem. We looked at specifically was how they can program a single program that would--what I would say, I think that was David. I'm sorry, I don't have my Chat window in front of me, is you've hit on exactly the point. When you find the good programmers, what sets them apart is not going to be how fast they can do normal programs. It's going to be their ability to solve harder problems, and that really talks about how you should be using those programmers and how you can identify them. You're not going to find them by saying they do these programs faster than others. It's going to be they can do these really hard programs faster than others.

**Shane McGraw:** Okay. Great. Next one we got, Bill, is, "Did you observe a single programmer or a group of people collaborating? A very productive programmer is not always the best team player."

**William R. Nichols:** That is very true. This was set up in a controlled environment where programmers were working alone. Now, they wee allowed to do some collaboration, especially if they had a problem or something, but this was really not designed as a collaboration experiment, and I agree completely. If you're going to talk about leveraging programmer productivity on real projects, you need to find people who can work in a group, and among the things that we taught were things like how to do peer reviews, and we taught a course on how to do peer inspections. Those had some of the highest leverage of anything, and if you haven't seen--I'm sure you guys have seen programmers in action. The people that are able to do these reviews are worth their weight in gold. The people who are able to accept the results of these reviews without their ego getting in the way, those are going to be the team players, and that's really part of what's going to make real teams effective, helping each other, and I've got more to say about that in a little bit.

**Shane McGraw:** Okay. One more in the queue, Bill, and then we'll be caught up and we'll turn it back to you to proceed, but from Mark asking, "Is relative effort always based on normalizing across all 10 assignments or is analysis normalizing across each assignment?"

**William R. Nichols:** Okay. Good question. In the first graph I showed you--let me see if I can get that back here--this one is across all 10 assignments. So this was the total. So basically this is the summed total over 10 assignments. When I got here and I showed the spread here, each assignment goes to one. So this--what we saw was a difference here was this gave you an

average tendency of the programmer. Gave you a good measure of the average, but that gave you no sense of how uncertain that average was. By normalizing to the individual assignment and getting to the spread on this chart, this gives you an idea on the individual program how variable their performance was.

So on the first chart we were normalizing the total effort, and on these next two charts we were normalizing by individual program. So we weren't trying to measure the size of the program by anything specifically like lines of code or function points. It was we're going to compare everyone on one program at a time, and if we add everything up it's going to be comparing everyone to the whole basket of programs at a time. Okay?

**Shane McGraw:** Yep. All caught up. Thank you.

**William R. Nichols:** Right. Thank you. So.

So what I took away from this is that we were seeing historically some of these huge measures, these huge variations in programmer productivity because, well, you could get whatever min-max ratio you wanted just by varying your sample size. If you waited long enough, you'll find someone who got stuck on a program, and it really means you were measuring for the max how long someone was willing to wait before quitting and they dropped out of the experiment. That's what you were really measuring.

So all those other studies, they replicate. This is not example of a replication crisis. These studies replicated but they were chasing noise. They were actually measuring the wrong thing. Now, there are some limitations to this study. We already touched on a couple of them. First of all, we only looked at the C programmers. That's only 15 percent of this sample. As I said, we have some additional large samples with Visual Basic, with C++, with C#. We haven't looked at those in detail yet but when we've looked at them they've generally looked the same. But I'll tell you in a little bit where to find the data. If you're a data scientist or a software engineer and you're inclined to look at this data, I've put it all out there under the Creative Commons license and you are perfectly free to use it. As long as you do the attributions, have fun.

Now, we have the issue these programs might not be representative. They're mostly mathematically and numerically oriented. So that means, oh, maybe some programmers are just going to be more naturally suited or more experienced in this than others. They tend to be fairly small. There's only a range of about two and a half in the program's size. They're simple and well-defined. Well, that was a subject of one of the earlier comments. These were not particularly challenging programs but they were representative of normal programmer work. So what we can say is this applies where normal programming effort. For normal programs, you just shouldn't see that much variation among professional programmers. If there are ways to distinguish them, you're going to see those in other ways. You're going to see some

programmers who can do the most challenging problems. You're going to see the Linus Torvalds of the world who are solving the bigger, the harder problems, and you may--and those are the people that you could potentially use as enablers to make teams effective.

What about dropout bias? We only measured people--in this study--people who completed all 10 programs. Maybe we would say, "Well, if you include the people who dropped out, you might see much bigger ranges." Well, I started to take a look at that and I don't think that holds up. I think that the bias--the dropouts tended not to bias these results, at least not very much. But to be comparable to earlier studies, the earlier studies didn't include people who dropped out either. All the prior studies were done with people who finished, so apples-to-apples comparisons, that's where we are.

What about selection bias on entry? These were highly selected, that we're only getting the best programmers. Well, I don't think that's very likely. Most of the people who took this course were sent by their company, and a lot of the people who took the course actually weren't professional programmers, they were people who were managing the programmers. So maybe 5 or 10 percent of the people who took this class were actually either managers or they were on the process team. They were people who had to work with the developers. Sometimes we had requirements people taking this course. They often did have trouble completing it, but that wasn't real--the bottom line is we had a really good cross-section of people in the class, and I don't think entry, I don't think the selection coming into class, was a big factor.

We did have a lot of reuse of artifacts, so there could--that is these programs often build on one another, so what we might've actually been measuring at some level was the--how well some developers were at reusing what they'd already built. But if that were the case, you'd expect to see more of them in a narrower range, and that's not what we found. We found this huge--we still found this huge variation, and we still--but finally, we do find some significant. They aren't huge, but we still find differences in the programmer productivity.

What does this tell us? That min-max ratio is kind of an artifact and probably not the best way to think about the developer productivity. Really like to kind of think about how do they compare to a mean or an average? Because when you're making program plans, when you're planning a project, you want to--you really are going to be working with benchmarks and averages, and that min-max ratio is really driven by a few stragglers at the low end. So that typically isn't very illuminating. I would really try to get more about look at average values, at median values, and how people deviate from those.

One of the things that I will point out is that it's going to be really hard to get enough top programmers to affect a large program. Now, this isn't to say that you don't want some of those top programmers as enablers, but don't think that just getting a bunch of super programmers is going to make your project go all that much faster, because even the best programmers had a

pretty wide range in their productivity. If I'm running a project, I would caution, "Don't overreact to short-term variation. You're going to get a lot of variation." This data's showing us that everything we've been told about the unpredictability of software is true to the extent that it's very hard to predict when any particular program will get done. I can give you a lot of assurance on a basket of programs, on a number of them, the statistical average, but any one of them is going to have a lot of variation. But those outliers have some information.

When we were teaching this course, one of the things we often found was that we could walk around the room and identify people who were stuck, and we'd usually try to give them a little bit of help. They were usually stuck on some problem, and it was basically something where they were getting a mental block and just couldn't see it, and those are the kinds of things where if you recognize outliers, that's where peers in a team environment can really help each other and make the entire thing work better. The other thing is that short-term variation is really large. A lot of people like to put in process changes and just measure the results and say, "Well, that worked," or, "That didn't work," and the real world is a lot more vague. The short-term variation can be so huge you really do have to get a lot of data before you get a good sense on whether some change is really helping or not.

So part of my lesson here is don't just focus on productivity. If you want to improve overall performance, look at a lot of global factors, like environment, design, tooling, reviews, quality. These matter and they're controllable. I mentioned earlier about inspections. We found those have an enormous impact on the late stages of a product, of a project. Putting in peer inspections makes the entire project must more predictable because you find a lot--people find each other's defects really well.

Environment. If you want people to be productive, they need to be in an area where they can focus, where it's quiet, where they don't have distractions. I know a few of you are--more than a few of you probably think you've never been more productive in your careers than you have been at COVID unless you have small children or dogs around the house.

Some future work I will talk about is interestingly this relative effort was actually one of the places where the individual varied the most. We found people were even more consistent in how many lines of code they could type, they could write, per hour. The upper limit there is how fast you can type, but the total relative effort actually had more variation within programmers than some of these others.

Defect density was even less specific to individuals. All the way down to estimations, we found that estimation there was almost no variation. So it's kind of interesting that there're all these other effects and they all seem to have a different mix of how much is the variation between programmers and how much is within? So if we--again, if you want to look at the data, if you really want to look at the data, down here--I'm trying to get my pointer. So there we go. On the

IEE Data Port, we put the entire data set. Not just the one in this study, but the entire data set is up there, along with descriptions. So if you'd like to look at the data, if you'd like to use it, go ahead. If you have questions on how to use it, please feel free to ask, because we're more than anxious. This is a unique data set. I think it would be a great data set to start using in statistics or software engineering courses for training purposes. Can actually--it's one of the few examples where we have really good quality large volume of data, and you can talk about the issues of what are the potential biases? What can you actually infer from these? I think it'd be a really great teaching tool.

And here are some more links on how to get the data and the descriptions, and I do want to thank all the people who are responsible for recording this data. This was from--oh, this must've been around 2006-2007, when we all went to Disneyland, and off here on the side is Watts Humphrey, who created the Personal Software Process, and he's been near and dear in our hearts for a long time. We miss him.

A bunch of references, and that's the end of this presentation. I hope it wasn't too data-intense, but I really do get charged up by looking at the data, and I hope some of you do too.

**Shane McGraw:** Hey, great talk, Bill. We got lots of questions and comments though.

**William R. Nichols:** Great.

**Shane McGraw:** Let me know when you're ready and I'll start firing them off.

**William R. Nichols:** Any time.

**Shane McGraw:** Okay. From Juani asking, "Are these normal programmers, or does the self-reflection process weed out the poor programmers?"

**William R. Nichols:** I'm not entirely sure how to answer that. I believe these are just normal programmers. There's going to be some weeding out of people who just can't do the work, probably never get to here, but I think those are the people who drop out of the field too. We had a very wide range of experience. Most of them were in the one-year to six-year range. We had people with 15, with 30 years of experience, and it did tend to drop off with time. But, you know, the programmer pool was growing too. My best sense of this is that this is representative of normal working programmers. That is this was not student data. This was not data from college students. We have lots more data that was recorded in college courses, but this is actually from working software engineers, and as I said, most of these were referred by their company. We had entire teams come and take this class at a time.

**Shane McGraw:** And then you had mentioned at one point in the talk, Bill, Creative Commons license. Was that referring to the PSP course that's available?

**William R. Nichols:** Oh, no, but that is true. The PSP course has been placed under Creative Commons, and I think I put some links to that course at the back of this presentation. That gives you all the context you need to understand what this course is teaching, gives you the actual programming exercises that we used and so forth. The data, the actual data, is also placed--it's free to use. It's basically--the data's basically free to use as long as you do attribution, and the suggested citation.

**Shane McGraw:** So what we'll do is we'll make Bill's slides available. We'll send, in the follow-up email that we send out, or by tomorrow morning, we'll make a PDF of the slides. We'll have the references so everybody can get to those materials.

Next question from Manuel asking, "Did you allow the developers to use IDEs, automatic syntax checking?"

**William R. Nichols:** Great question. We told people to bring the environment they were comfortable with. If they were using an IDE, we asked them to turn off syntax checking so that that should not be as big, that should not be a big variation here. The reason we asked them to turn off syntax checking was because the course was really designed to teach them quality and it was harder to teach, we thought it would be harder to teach them how to review code, if there were fewer defects in the code. Honest truth, as time went on, everyone was using IDEs and it became really hard to get people to turn off syntax checking. When they started using syntax checking a couple years later after this data, all that changed was the number of defects we found in compile went down. It didn't change the--it really didn't change number of defects that were escaping into tests, so the IDs didn't make as big a difference in our observation as we thought. In principal, if we can dig that out of some of the later data sets, we might be able to measure the effective IDEs, but that'll be a harder thing to get because we didn't record it.

**Shane McGraw:** Okay. Now we got a couple questions and comments from Mark Paul, just so you have a context.

**William R. Nichols:** Oh, Mark. Hi, Mark.

**Shane McGraw:** Yeah, who we're speaking to here, and Mark's first question was, "Did you read my PSP analysis, which also broke out the data by programming language and included analysis that dropped outliers?"

**William R. Nichols:** You bet. I did read your thesis. I read your thesis--

**Shane McGraw:** Okay. And I'll go into his comments. Whenever you're ready, Bill. I'll let you--

**William R. Nichols:** Go right ahead.

**Shane McGraw:** Okay. So the questions, then the comments following up, were, "The thing they did not do was normalize by average effort. I also did mixed models. I got different results than you on--

**William R. Nichols:** Yep.

**Shane McGraw:** --several items," and he says, "There was a consistent 5 to 1 difference between top 10 and bottom 10 quartiles, for example, 10 times for top and bottom vessels," and the last one was, "I agree that the min-max is a bad technique, and outliers may be influencing top and bottom."

**William R. Nichols:** I'll have to take a more careful look at some of the things you wrote the-- but I'm glad to see you're there, Mark, because we definitely, we were definitely looking at your thesis back in the day when we were looking at some of this data, because there was a lot of really good work in it.

**Shane McGraw:** Next one from Manuel asking, "Have you considered using TSP to get the team effect?" and TSP, team software process.

**William R. Nichols:** Great question. The TSP data's a lot harder to analyze because you don't have a lot of these artificial constraints. We are looking at some of the TSP data, but we can't do the same kind of comparisons because they weren't programming the same things. Now, we do have some data, I do have some data published out there. I don't have the link on these slides, but I can show you some individual TSP data sets that I prepared for another presentation when we were looking at static analysis, and if you like I can add that link to these slides so that you can download those from the Carnegie Mellon GitHub. Not GitHub, KiltHub.

**Shane McGraw:** Great. So a question from Lonnie and a comment from Paul, so I'll group them together. Lonnie asks, "So faster programmers don't have a lower defect density?" and Paul's comment was, "Teaching the course to pairs of programmers using pair programming would be interesting."

**William R. Nichols:** That might be interesting. We never tried that, although there were some people who used PSP to measure pair programming, and they were finding some pretty decent results. I think, if I remember, the pair programming results were getting about a 30 percent defect yield pretty consistently. Sometimes better. We have one fellow from Hill Air Force

Base that's saying that they were getting spectacular results with peer programming. I think if you were teaching that as part of the course it might be very interesting. The difference is the PSP was really relying on a checklist-based review, and I don't think the peer program is going to work quite the same way. So it's going to take some more thought, but all the courses out there, and if you have some bright ideas, I do invite you to try it out and see what works.

**Shane McGraw:** Okay. From Sandish asking, "Was there any architectural qualitative analysis of code? Evolvability, scalability, modularity, design of the code itself? And that's the last one we have in the queue, so while you respond to that, any other questions, feel free to get them in there, folks.

**William R. Nichols:** Okay. The answer to that one is "No." This one did not take into account the architectural properties of the code. The course wasn't really designed for that and I won't go--I won't take it beyond its capabilities. Those are certainly good questions. Those are very relevant, and it's, again, one of the limitations of this study. It's really based on primarily the developer portion of having a set of requirements due and just getting out some code that satisfies those requirements.

**Shane McGraw:** And a question from Baboo just came in, "Was the code considered in the analysis?'

**William R. Nichols:** I'm not quite sure what the--oh. In this analysis, I did not consider the code. I'm not sure in what sense you are looking for. We did not save the source code in this version of the course. We did have, we do have data on things like the programming language and the amount of code written, and we have some other studies that look at that. But later data sets do actually include the code, but that's a much harder analysis and we've never used it specifically.

**Shane McGraw:** Okay. So we're caught up in the queue. I'm going to give everybody about 30 seconds to see if there's any other questions. While we're waiting for that, Bill, just a reminder to everyone that our next webcast will be on December 15th, and we are going to have a roundtable with the CMMC model architects, so everybody will be emailed a registration link for that, but that will be December 15th. Look for that.

So we do have a follow-up question from Manuel asking, "Was there any way to detect if domain showed a difference in the results?"

**William R. Nichols:** Domain. I don't think we have a--we don't have a whole lot of information on the domain they came in. I'm going to have to check the data. I don't think we can answer that directly.

**Shane McGraw:** Developer domain?

**William R. Nichols:** Yeah, that's kind of what I was thinking the question might be, the developer domain. We do know that math background and statistics background didn't make much effect. It certainly is plausible that people who work in a numeric intensive environment would have an advantage on some of the programs, but there were couple of programs that were more text based. I will say that when I took the course, I struggled a lot more with the line counting programs, because there was a lot of text manipulation going onto those that I just wasn't as familiar with, but I knew the numeric parts of the language inside out.

**Shane McGraw:** You know what, and I did miss one earlier from Paul that I thought was a good question, talking about--it said, "It would be interesting to know where the students were taking the class. Were they sleeping at home, hotel they were staying in?" So I'll let you comment to that.

**William R. Nichols:** Okay. That's a good point. I hadn't thought of trying to do that. The advantage of the--we taught this course in a couple of different formats, and one of them was we would teach this at SEI, and for those, most of the people taking the course were staying at hotels, although there were some locals who could come in and take it as well. But we often went out and taught this course at the customer site. In that case, the majority of the people taking the course typically would go home at night. My experience as a teacher says that the people who were in the hotels had a lot more time to focus, and the biggest effect there was they were the best at actually getting done. The biggest challenge teaching this course was getting the programmers one, because you saw those outliers. Everyone had a program where they had trouble, and if you didn't get it done it was hard to keep up and finish the course, and that was one of the things that drove dropouts was just--people had to be able to keep up with the class or it was hard to finish, and I think the people in the hotels really had an advantage there because they didn't have as many distractions.

**Shane McGraw:** Great. That sounds like a good one to wrap up on. Bill, thank you very much for sharing your expertise with us today. Appreciate your time.

**William R. Nichols:** You're welcome.

**Shane McGraw:** And lastly, we'd like to thank everyone, again, for attending. Again, we had a worldwide audience. We thank you for the different time zones and people taking the hour to spend with the SEI today. Upon exiting, please hit the "Like" button below and share the archive if you found value. Also, you can subscribe to our YouTube channel by clicking the SEI seal in the lower, right corner of the video window, and lastly, I mentioned, our next webcast will be December 15th, where we have a roundtable with the model architects of the CMMC.

Any questions from today's event please send to info@sei.cmu.edu. Thanks, everyone. Have a great day.