This is something engineering radio broadcast for professional development on the web at S.C. dish radio thought that as a radio brings you relevant and detailed discussions and interviews on software engineering topics every two weeks. Thanks to our audience and the promise listed on our website. OK Welcome listeners to another piece that unsoftened doing radio and today we have Kent back on the mike and talk with him about the history of J. Unit and the future of unit testing so welcome can't thank you very much. It's great to be here. Great to have you on the show would you introduce yourself to all listeners tell us a little bit about your history and your background. Sure. Well I'm a third generation geek my grandfather was a radio geek. My dad was an electrical engineering geek. Turned programmer geek and now. Now I am and now my my oldest daughter carries on the family tradition. So I grew up in an environment with lots of technology around. I started programming when I was about twelve or thirteen maybe maybe eleven. And a programmable calculator my dad brought home. And. Let's see I don't know what kind of highlights the listeners will find interesting. I've done a number of things software patterns were something that I worked on early on. Extreme programming test driven development the X. unit. Architecture which was first implemented in Smalltalk. And then translated into a job by your gammon myself. Responsive design. Well what else what else are you interested in here a lot of the great E.-S. I think I've written eight books so far. Oh wow that's a lot. I think that they're pretty pretty famous in the community by geeks standards Yes OK great. And I think I read on Twitter that you are great fan of the new episodes. Is that correct. Yes So I live on on eight Hector's of mixture of of meadows and forests outside of a little village which it's outside of a slightly larger town in the middle of nowhere in southern Oregon. So I have a lot of farm chores to do and I listen to lots of podcasts including having listened to. I think all of the episodes of software engineering radio. It's great when you're shoveling manure to be able to tune into something to feed your brain while you're while you're taking care of that or feeding the animals or whatever you know cool. I think it's a great honor to have you as a listener you know for all the reasons it's great. Cool. And so can you tell us a little bit about the history of J. Unit. Well things began sure so I my first commercial work was in small talk. And Small Talk had history of testing but not automated testing so you'd make a few changes to your programs and then you'd run a little bit and then you'd make some more changes and it was set up for that kind of very incremental cycle but there there weren't automated tests. I did a series of experiments so I tried four or five different techniques for writing automated tests and then and then one day I was asked to give advise. I sed to a development project early on in my consulting career and I knew I was going to tell them to write automated tests but I didn't have a way for them to write automated test so kind of in a panic. I said let me take this structure that we use for testing in Smalltalk this workspace structure where you'd have some variables that would get initialized and then you'd have expressions. And I said well what if I just turned that naively into an object model. So the workspace is a class. The variables that you use become instance variables the the little snippets of code that you the used to test become methods. So it's just really kind of the naive mapping. And then rather than doing manual testing where we'd say oh print this value out and then you'd make sure that yes printed out A.B.C.. I said well let's have the computer do that. So that's where that's where the assertions came from so that was the origins of the of the architecture and that's got to be in nineteen eighty. You know I kid can't remember. Exe Yeah I don't have a date an exact date for you but that's. Now it's got to be in the one nine hundred ninety two. So that's when the first unit testing framework that I did that was at all successful came from. Fast forward a few years I was living in Zurich and Eric gammon and I would get together and program from time to time. And Java was fairly new at that time. And I wanted to learn it. So we were on the plane going to. And he'd heard me talking about this this testing architecture and I'd heard him talking about how cool Java was and so we just sat there and. And built the first version of J. Unit test for. From the very first which is an interesting bootstrap problem right. How do you test a testing framework so that if it breaks you know that it breaks. So we you know we like those kind of intellectual challenges so by the time we were done with that plane ride we had the first version of Jane and. We gave it to Martin Fowler I mean it but it was so small. Right. J. Unit is so small. How could this

possibly be valuable. So we gave it to Martin Fowler there in Atlanta. He tried it out immediately and said this is cool. So we gave it out to several more people got really good feedback and that's pretty much how the whole thing started. Well and it's a pretty successful start right today. J. Unit is the the defacto standard for doing automated unit testing in Java right. Correct. So what happens. Since then it's a long time ago. What happened to change unit in the Nikkei. Yes So that was in in ninety six to seven. That we wrote the first version. And so it's gone through. It's now in Version four. It's gone through one big architectural change which was the switch from subclassing a framework sort of metaphor where you subclass something and overrode certain methods to a D.S.L. metaphor which you implement in Java with annotations. So I had seen end unit very early on when it was just a few files on. James new Kirk's machine and seeing the annotations and thought This is a cool idea. And then I realized well a lot of the way that for instance methods used to have to begin with the letters T E S T. Well that's a kind of metadata because we would go and interpret that. You know. J. Unit is a language. It has semantics they're different from Java semantics for an ant for example inheritance runs differently. Than that than it does in Java but. So we were we were using the language. To we created this language in Java that wasn't really job and interpreter. But with annotations now all of a sudden the compiler can give you help again because if somebody's name their test. It just wouldn't run and there'd be no way to tell but with the annotation you know whether it's spelled correctly then there is a lot of language implementation of the gee you know a lot of translations to the languages. What do you think is this movement. Well I think I can interpret it on a couple of levels one level is successful design. I mean it means that there is something common to tests that is that you want to run these isolated chunks you want to be able to compose them together and not have them interfere with each other. And you want to do that all in the source language. I mean there's huge advantages to not having your testing language be your source language but there's also huge huge advantages to keeping the the source language and the test testing language have it in the same in the same language as it is with the with J. Unit. So kind of there's this attractor in the space of design tools and. You know it draws a lot of functionality into that same basic shape today who was implementing the current version. You're still working on it right. Yes yes. So every week. Dave. SAFF who's at Google now and I get together and work on it for for a few hours. You know every week. Yeah pretty much. Wow cool. So you're already planning the next version. Yes We're already working on. We don't have a big new theme I would say though the last version came out with a really cool feature which is kind of working its way through the community that's one of the things the advantages of having a project that last for thirteen years is you learn a little bit the value of patients. So you know it's just if you come up with a new feature and you think it's fantastic five years later if people are using it then yeah that was a really cool feature but you can't you can't tell. After you know if it's only been three years and people still aren't using it then you still don't know whether it was really cool or not so. And that was what we call rules which are kind of let's see what's a simple way to explain it. So when your test runs this chain of objects is created. You know one object will will run the before methods and another object will. Will catch timeouts and another object will report errors Well you can insert using rules you can insert objects into this change so it's kind of a metal object for testing. And so it was a big deal. People are using it for all kinds of cool stuff rather than having to use inheritance to capture the commonality of tests now you can use composition. Which is a whole lot more flexible. So if people are looking for the cool stuff in J. Unit that would be that would be something to have a look at. So you think it's a key feature to do reuse stuff. New Unit testing code. I don't know if it's a key feature because you know when we went thirteen years without it. So but I think it is very useful. Certainly for capturing I mean. I always strive for a kind of declared of expression in my test you should be able to just kind of read a test and it tells a story and rules are a way of setting the stage for a test in a way that's obvious that you can read right there. But you know it at the same time can be quite simple. You know so you can ever rule that creates a temporary folder and then all of the stuff that you do in your test that does file manipulation. When the test finishes the folders automatically deleted. So so in reading the test you can just say oh look there's a temporary folder I'm sure there's

some file manipulation going on here. So rules give in an imperative language the stills gives it this very declarative feel which I think is valuable for tests. OK so that it's not only this kind of reusing stuff. But also to make it more and more expressive. Yes because every test to tell a story. It should have a you know a beginning a middle and then it should have a kind of a flow an R. and it should have a moral right what's the point. What's the point of this test you should be able to read that out of the test so rules give another way of setting the context for test that's a little richer than anything that had happened before and it's introduced in version four point zero eight. I think four point eight. Which is what you can find on get hub. That's where we where we host now after a long and successful run on Source Forge you know OK great. So you also mention these that you were doing on or you. Maybe you're going to test driven development. So I can give you a very brief introduction. What development is. It's this crazy idea that you know the best ideas are the crazy ones. Right. If you have a crazy idea and it works it's really valuable if you have a if you have a fantastic idea and it works. Who cares right because somebody else will have taught it but if you have a crazy idea and it works then you really have something so it's it's this crazy idea that you when you want to code the first thing you do is you write a test that fails that will only succeed when the code that you're imagining is actually in place so it turns the work flow of of programming backwards when you think is the most important point when you do test driven development then then you think about the design for drive the implementation from the viewpoint of the user of the class for example because you read the test first. Or is it something else. It hears what happened to my head after I'd been doing T.D.D. for a couple of years and that is I would imagine some functionality. And my next thought was How am I going to test that always before I would imagine some functionality and then I would think. You know I could subclass this and extract data as an object and replace it in a bump on the SOL about implementation. And now when I think about functionality My next thought is how my going to test that what it does for my thinking is that trying to imagine the implementation and how I will know that it's correct at the same time because I make mistakes. Is is a big job sometimes it's too big for me to hold in my head. If I can divide that into two parts and handle the how am I going to test this part first. Then I have. Smaller less stressful easier to focus on problem to deal with namely how my going to test this. Oh well if you can't write a test for you've got no business programming it which is true for most programs. Now when I'm being very exploratory I don't worry about this stuff you know if I just want to say can I get a program that does X. at all then I'll just slink code and check it manually. Then once I discover Yes this is possible then it's a good time to switch gears and say OK now you know let me go back and let me implement this a little at a time and figure out how I'm going to test this for many people I talk with him. They still think that. Doing the test before the actual code is really a crazy idea and there isn't some maybe means something there. If fraid of doing this kind of stuff. What is your observation of the adoption off test of new development in practice in industry in the wild. Well I'm really curious like what could go wrong. Your head's not going to explode like it. So do you do T.D.D. Yeah so what do you say when somebody says Oh it isn't going to work to me I say it helps me just go forward in small steps and keeps me going. Sustainable pace in small steps and just helps me thinking about what I need to do and how can a person be implemented stuff. Yeah that's the there's a sense that I I have that's my experience of it as well. Now it is I thinks some of the barriers are probably social because when I went to computer science school you know back when we had to dig our own sand for our own transistors. The good day students got to be programmers in the B. students had to be testers. So part of the point I had to overcome was just. And of this this social status. Oh I'm a prig. You know I don't have to write tests. And and like I had to overcome that you know that I was somehow lowering myself to write this which isn't true. I mean it makes no sense but that is the human brain. We're talking about so. So I you know I don't know if that's part of what the people you're talking to are encountering but that was certainly certainly a barrier for me to think. Yeah I should spend thirty forty percent of my time recognizably writing tests. Now when I'm writing test some not just writing tests I'm making A.P.I. decisions. I'm making analysis decisions and the tests just happen to be the notation that I use to record those decisions. But at the end I still do have a

bunch of tests that actually run. Do you think you convinced most of the people you talk to. I don't know because I don't I'm not I don't try to convince people anymore. I think I spent probably ten years really trying to convince people here programming you should do this you should do that and I don't do that anymore. I'm always working to improve my own practice I eager to share what I learn. And I try and listen and understand what other people learn in their experience as well. So I don't like. I don't really keep score by how many people do T.D.D.. If the practice of software development is improving overall then I think that's a great thing. And if I have some small influence on that then I think that's a great thing to do well. So yes a question about do people like how widespread is is T.T.. I think it's still a very small percentage of people that. They are like won't write a line of code unless they have a test that's failing. You know under for production coding. But I think there are a very large number of people that have been exposed to tests and the potential value of tests. But I still go places and people say oh yeah we did a bunch of tests but then the test stopped working. So we threw him out which just seems bizarre to me. I mean like Aristotle would be shocked at the logic just doesn't add up this test says if the test is running my programs are running and if the test is not running then my program is not running. And the test stops running and your next act is to delete the test or just stop running it or ignore the test report that you get like wow that means your program is not running but somehow I mean there's a lot of other pressures on people. Then get your program running I guess that's the that's the conclusion that I can draw from that but it's kind of it's kind of it's too bad I think there's. This potential value there people could produce more value as programmers if they trusted the tests and paid more attention to them but you know there's a lot of other things going on in software development than coding then say maybe test philosophies. For example. Be a view driven development and instead of doing test new movement. What do you think about it. Well. So the the big picture. I think is very positive programmers actively taking responsibility for the quality of their work. I think that's that's a good thing and whatever you call it kind of. That's a second order effect. Something. I didn't communicate very effectively in my first discussions of T.D.D. is the importance of working at the testing of various scales. So T.D.D. is not a unit testing philosophy. I write tests at whatever scale I need them to be to help me make my next step of progress. So sometimes they're very. You know they're somebody else would call a functional test. So for example forty percent of the tests of J. Unit work through the public A.P.I. sixty percent of them are working on lower level objects. The the public A.P.I. is is quite good for testing. Probably because we've written so many tests so I don't know if those. Proportions are are. I don't want to claim those proportions are anything more than one data point like should you have forty sixty should you have ten ninety or ninety ten I really don't know but just this idea of moving. Part of the skill of T.D.D. is learning to move between scales. Right. So I write a test. That my customer says Oh blah blah you know this scenario should result in a five. So you write a test that says this scenario should result in a five and then you're down in the Deep in the intestines of your program and you're thinking Oh I see this object when given a five and a seven should return the five. Well that's a good place to write a test because that's another piece of the story that needs to be told. But you know is that acceptance test driven development or is that you know I think that. Directing rigid walls between the styles is actually a mistake. Like this. The scales as a programmer I want to understand all those. Scales. Tests help me understand so I write tests at all those scales. Does that it's your question I think so. And it also leads me to another one about scaling unit testing. And so projects having thousands and thousands of unit testing and they had problems. Let's say dealing with all those huge amounts of unique testing. When you think about this is going to see does unit testing merely scale and that's in the on the small level not the acceptance testing level but on the small level you have thousands of classes and huge amounts of unit testing maybe two or three times as much unit testing code as your actual real production code. What is your experience with that. Well I would say that's two to three X. testing code to production code the production code have to be pretty complicated but that's well within the realm of normal. That the guy I learned the most about testing from was a compiler writer and and he had five lines of tests for every line of compiler that he wrote and he was the most productive guy that I'd ever seen. So that was a real inspiration to me.

But he's working on a compiler which is you know extremely complicated inside. So if it was one to one or five to one it wouldn't that wouldn't bother me but you still think that you need some kind of let's say a tester of new element or testing automatic tests on different levels of scale. So you have these kind of very small test for for very small units some A.P.I. is to acceptance level stuff. You know it's SAF claims that. David SAS the other current committer on J. and he claims that test tends to migrate either to the lowest level of objects or the highest. And he was talking about tests for Eclipse applications and so on. You know I can I can believe that I don't know if that I've really noticed that in my experience I think. I think that I have some tests in kind of the middle of will that that I'm happy to have and that don't cost me too much already mentioned that you. For example if you would like to figure out if that's possible to implement X. you don't write a test very if you just would like to figure out is it possible to implement this. There are the situations where you think. You do not test driven. Development the other way round. So I remember a tweet on Twitter saying something like good developers know when to test and when not to test can explain it a little bit. Well. So the tests you've got costs and benefits and. Some of the benefits are short term and some of the benefits are long term some the costs are short term some are long term. And different situations have different profiles of of short and long term. So for example I threw together a site called Poker workout dot com because I was learning to play poker and I was having trouble doing the pattern matching just figuring out. I have seven cards. What kind of hand. Do I have to have a pair do I have two pairs so first I just wanted to know if I had a program like this would it be fun to use and would I you know would my recognition skills improve. So I just banged it together first version in small talk then I translate it into Java Script and I didn't have tests. But I discovered yes this is fun to play so then I wanted to expand it to. You know more situations and I had to make the Demain model richer and to cover more of poker and when I did that then I started writing tests and when I started writing tests I discovered you know the the the design was. The chunks were too big so I had to pull pieces out and then I could write test more easily for them but there's this there's this transition period. At first read I could easily have said. Is this fun useful at all and the answer could be no. So all the long term benefits of having carefully written tests for that are gone. Because the you know the half life of the program is so short. Once I say Ah yes this is good and A Yes I want to maintain it and yes I'm going to live with this code for a while. Then the long term benefits kick in. They become you know more probable that I'll be able to reap benefits long term. So then I need to kind of switch gears. But overall that's what I'm thinking about what are the costs and benefits short term and long term. So when I started. Jane and Max. Which isn't clips plug in writing tests for clips plugins or is notoriously difficult. You know and I I wrote a book about it with Eric and I still find it a real challenge sometimes is the A.P.I. is is extremely powerful and flexible but it's not set up for isolated testing very well. So when I when I started Jane X. I didn't have automated tests. Not because I knew I was going to live with this code for a while because I just knew that Max was a great idea and I wanted it for myself and I maintain it for myself if no one else. But the cost of writing those tasks was going to be extremely high especially the first few of a. And then I wrote so for the first same month that I was developing Macs I didn't have any automated tests I did all my testing manually at some point the complexity of the code got to where I was started worrying about if I change this. I'm going to break something else and that's when I had to as we say in English bite the bullet. You know do something difficult to spite of its difficulty. And just start writing tests now. Now I have nine I was going to say lots I have sufficient tests automated tests for Macs. But in that exploratory phase because the short term costs were so high. Even though I knew I was going to reap the long term benefits I still didn't write automated tests for that. And again I had to get become aware of this balance. I'm waving my arms around here which doesn't really help you understand. I become where aware of where this balance shifted to where the costs and benefits short term and long term now said I Yes that you know the previous feature I did without an automated test and this feature I'm going to do with an animated test even if I have to pay a substantial price to write that test and and I was happy with how that worked out so I think that's what I'm thinking. Yes it's very interesting because also people that are much more dogmatic about

unit testing interesting development and they say before you implement a single line of whatever it is due to student development because it helps you thinking about the piece of cool designing the piece of cool whatever. They're very much more dogmatic and then you are interesting to see different opinions about this. Well I think it's worth being dogmatic as as a learning tool. Right. What if I just said I'm always going to write tests for everything. And then you discover. Oh I'm glad I did this here. I'm sorry that I did it. So I won't do it in you know what's the commonality in the experiences where I wished that I hadn't written test. What's the commonality in the experiences where I'm glad I wrote the tests then let me and I'll use that to inform my behavior going forward. Some people are actually writing in their books that or discussing it that people aren't today writing too many unit tests. But what is your experience or your opinion. Oh let's let that be a big problem before we try and fix it. OK I you know this. It's kind of the it's kind of one of these three bears things where you know they're certainly too little and there's maybe too much in there someplace in the middle. That's just right. Do you have do you have Goldilocks in the three bears in Germany or do you know the story. Now I'm not sure. So little girl goes into the bear's house and she eats the papa's porridge and it's too hot and she eats the mama's porridge and it's too cold and then she eats the baby's baby bear's porridge and it's just right so when I say Goldilocks I mean you know ye yes you could have too much you could have two little and someplace in the middle is right but I don't think people really know what too much really is. I would want to examine both of the tests and the activities of a team that felt like they had too many unit test it. Understand more about that I've never achieved it. And I've tried. What do you think what makes a good test a good test and what makes better test a better human test. I'm pretty sure you have seen a lot of good in a little bit. Sure. Well I think every test has got to tell a story that is somebody coming on later and reading it should be able to understand something important about the program. So my first filter is is kind of a human communication filter. Tests should have. In in medicine they call a differential diagnosis where they say I'm going to order this test. And based on the results of this you know whatever blood test. I will be able to rule out a bunch of stuff and confirm some other things. So every test should have this kind of maybe this is an information theory. Thing. Should be able to differentiate good programs from bad programs. If you have a test and it doesn't do anything to advance your understanding of good programs and bad programs and that's probably a useless test that if you took the space of all possible programs to solve your problem. You know almost all of them won't. And a few of them will a test should should should lop off a big portion of that space and say nope all any program that doesn't satisfy this test is definitely not going to solve the real problem. So there's a part of that and then there's a sense of redundancy leaf you have a bunch of tests that tell you exactly the same thing. Then I would look to see which of them adds the least value and delete them but they have to they have to really cover exactly the same cases you think that all the other means I'm principles that we all know for general programming also play for for the test code like these things like dry and whatever dry in particular I don't subscribe to for test code because I want my tests to read like a story. So one of the things the features in J. Unit is that you can if you have a test set up in a superclass it will be run before the test is run. So you can you know you can extract out some common set up code between several different test classes. You do that. Three or four layers and now. Now the test can be very terse right the test just says. You know some objects some message. Here's a return value well but to understand what's really going on there to understand the story. You have to look at the class and it's set up and I have to look at the superclass and it's set up and I have to look at it. Superclass and set up but time I'm done with that right. The. You know it's it's like telling someone the punchline to the joke without tell I'm telling them the whole the whole set up. You know it was the priest. You know it's not funny because you don't you don't have the whole set up now. I hope I haven't offended any priests who listen to the podcast. But do you observe or do you observe any other really strange phenomena in testing was a couple of really weird ones. So there's one that I that I can explain that's really useful and there's one that's just weird. So the the useful one is that tests don't fail at random. If you take a population of tests. And you run them while doing normal development. If a test fails. It's very it's much more likely to fail

soon thereafter. So for most tests. Once they start passing they'll never again fail in this is based on looking at hundreds of millions of test runs representing like fifty person years worth of development. So once a test starts working it's just going to keep. If it ever fails then it's very likely to fail much more likely to fail the next time you run it. So we use that in J. and Max which is a skin us testing tool for. For clips and Java. To prioritize tests so that you run the tests that are most likely to fail first so that while your attention is really on the tests that you're running you get most of the feedback. Because really if you could prove and you I suppose you could with you know a coverage tester and some static analysis. You could prove that this if this test succeeded last time it'll succeed this time. Then you don't even have to run it at all. The this recent failure phenomenon is a here a stick. That's a lot cheaper to calculate because you just have to have a little bit of memory about what's happened in the past and. It lets you also prioritize the tests and say well let me run these first since they're seemingly fragile. You know brand new tests run first tests that have failed recently run for. So that that's true and useful phenomenon that I may be wouldn't have guessed at but makes sense. The other one is that if you look at any test suite. The time it takes to run the tests. Some tests are shorter and some tests are longer the distribution of that is a power law. So you'll have lots and lots of tests that will run very quickly. And a few tests that will take a very long time to run. If you create a histogram plotted on a log log graph you get this really ridiculously straight line. So I don't I can't explain why that is true but I've looked I've seen it in nuff datasets to believe that it's pretty much universal when you in such cases when you split it into two different lenses suite so as saying on the long running tests less frequently or only during the night. He builds a so to continue running the old tests. Well I think there's that would be certainly be one thing you could do and you would. What that power law distribution says is that most of the tests can be run in a small fraction of the time. Because the outliers are so big. So that you could split it up that way and indeed when J. and Max runs it runs the short running tests first what I haven't confirmed and I would love to have more data on is whether long running tests are more likely to fail or kind of the the worst the worst outcome would be the longer a test takes to run the more information that it generates and and the best would be if if there's no correlation. So if you want confidence that the code is running all you have to do is run the the short running tests and you get most of the confidence. Can you tell us a little bit more about. So you makes you know you mention it I think so. Time can tell us a little bit more what is it exactly sure. So it's it's a continuous tester for Eclipse. So if you're in a project. Java project in Eclipse you had at the source code you save it all the tests for the project are run automatically. And they the metaphor is is it's just like the compiler. So you make a change to your source code. The compiler runs you get feedback right away. There's not a separate testing. There's not a separate compiler tool. Do you go and execute in Eclipse and with Macs there's not a separate testing tool there's not a separate window you go to you just get the feedback right away. So it came out of my frustration as I ran tests. The more I did T.D.D. The more I ran the tests the more frequently. I ran the tests. So take the J.. Unit test suite for example it takes about ten seconds to execute. If you're running that twenty times an hour which is probably a conservative estimate and you wait till the end every time you run it. You know you can you can just do the math and say you know depending on what my billing rate is here. Here's how much money per hour. I'm spending on on just waiting for the tests to complete. So I wanted to cut that down. Well this test prioritization trick. And the integration into the the the tests appear. Failed tests appear like compilers so it shows up in the you know right in your source code you don't have to show switch context when you get test results back. So the combination of that means that I only have to wait a couple of seconds before I say yeah that's good enough. Now I can get back to coding. Well. So you know you take the the. Every time I do the numbers it seems ridiculous so I'll leave that as an exercise to your readers to figure out how long they're spending waiting for tests. And how much money they could save by having test results in in say ten percent of that time. It turns out to be a substantial number so that part of it part of the My motivation was frustration with just I was spending more my time waiting for test results and I wanted them part of my motivational as I wanted a way to make a living programming. You know I can. Yeah I can fly around give talks do consulting and so on but you know

I live in paradise so I'd rather not do that and I love programming. So you know is there a way that I can make my living programming so it was the combination of. The theoretical work on tests and the behavior of real test suites in the wild. My personal motivation wanting. To get more out of programming and spend less time waiting and this desire to to create a way of supporting myself and my family and college tuitions and paying for goat feed and so on through an activity that I still love to engage in and so that's genomics cool and doing need to write special tests doesn't just my general do you know this version three for what it gets so it's one way to look at it is it's a replacement for the the gene test runner that comes bundled with Eclipse. OK So so no special code in my test for that. No Not no not at all. It just gives you the results with less waiting. Oh great work again Janet Max dot com OK does not sound like open source it's some kind of you need a point. Yes So it's one hundred dollars a year. OK And every time I do the math to figure it out. It pays for itself about every two days into in time saves great. So we talked a lot about testing in the past tense in the history. Let's not try to take a look into the future. What do you expect. Well the things that I see. I think Jayna maxes is kind of my take on where the future is going that tests will become as important to programming as the as the compiler is when the tests are giving you value every minute then you're willing to invest quite a bit more in creating them. I think there's a big trend coming in design for testability So a system like Eclipse. I think is beautifully designed. But it's difficult to write tests that are. Concise and run quickly for Eclipse functionality. There are there's certainly another way to design exactly that same functionality. For which it would be easy to write unit tests that would execute quickly and be isolated from from you know extraneous failures and so on. But like exactly how do you do that. I think is there's a whole lot of learning to be done there. But as the value of tests increases. Then the value of learning about those design skills. Goes up at the same time and I think that they'll be a virtuous cycle of. More tests driving the need for more testable designs which make the tests less costly and more valuable. Which leads to more tests and does it. So I think that's that's something that I expect and then just running tests more frequently more of the tests running more of the time so kind of the next stage for something like Max if you wanted to get more feedback per clock second of the of the cusp of the programmer's experience. You'll need to start executing tests in parallel on a massive parallel any other pearls of wisdom we would like to share with the listeners is less questioned. Now I really I I have grown. OK so here's my pearl of wisdom which is be distrustful of pearls of wisdom. I don't trust the simple morals that people come up with at the end of their stories. I'm very happy to listen to the stories. But. When then they when they conclude you know therefore you should do things just like I did them. I think it misses a lot of context. So listen to stories and tell stories there that's my that's my moral which isn't a story. So you can just disregard it. But then I have to ask you another maybe different difficult question what do you think will suffer engineering look like interest the big the big trend is towards more frequent deployment and that drives everything else all the social changes that need to happen all the technical changes that need to happen. The changes in practice in language in infrastructure. You know if you're deploying fifty times a day in a database has to look different your deployment status look different year the current separation of ops and Dev just asked to go away. Marketing Sales business models everything changes when you're deploying very very frequently and that's a trend that is gathering momentum and I don't see any place where that's going to stop great maybe we should we should spend another piece talking about continues deployment. Oh I'd love to. Yeah great. So with this thank you very very you very much for your time. So it's my pleasure thank you thank you very much like to see the light. Thanks for listening to something new and radio something new radio is an educational program brought to you by host let you know if you want more information about the podcast and all the other episodes visit our website at se the radio thought that if you want to support us. You can donate to the radio team by a left side or you can advertise for us your radio for example by clicking on the big Reddit delicious links and this left up to contact the team. Please send email to team at S. he dashed radio talk for this specific to an episode. Please use the comments facility on the website so other people can react to your comments this episode if I see radio as well as all the episodes are licensed under the Creative

Commons two point five license. Please see the website for details. Thanks to the crowd and the pulse music. Well the music used in this show the song is called.