Hi, I'm Uncle Bob and this is Clean Code. Welcome back to part two of episode six, test-driven development. the origins of the moon in fact and then we got down to business we discussed how and why code rots we learned that code rots because we're afraid to clean it I showed you the test suite for fitness and I told you that if those tests pass we ship that product the tests are the QA process then I did a little cleanup demonstration for an ugly method I found in fitness the tests to clean that code very nicely. The tests eliminated the fear. Then we learned about the three laws of TDD. The first law says that you're not allowed to write any production code until you've first written a failing unit test. The second law says that you can't write more of a unit test than is sufficient to fail, and not compiling is failing. The third law says you aren't allowed to write more production code than is sufficient to pass the currently failing test. Our Mirror Universe visitor nearly had a fit over that. He had loads of objections, some of which we answered last part. We answered them quickly and we temporarily satisfied his ire. We showed how test-driven development shortens debug time, creates reliable low-level documentation, leads to decoupled designs and most of all eliminates the fear of cleaning the code. happened. Yes, Uncle Bob, I see it all now. The logic is impeccable. But that which is logical is not necessarily practical. You have a logical house of cards if you cannot establish that test-driven development is also practical. Then the time has come for Welcome. What we're going to talk about today is the bowling game. This is a very old example. I've been doing it for probably 10 years. It's a good way to demonstrate test-driven development. Let's get into it. Those 10 frames each consist of usually two rolls. You get two tries to knock down 10 pins at the end of the alley. Now, if you look at this particular instance, in the first frame, this guy managed to knock down one pin with his first roll. And then on his second roll, he knocked down four more for a total of five. on his first try. On his second try he knocked down five more for a total of nine. Nine plus five is fourteen. On his third frame he knocked down six pins on the first roll. On the second roll he knocked down the remaining four. This is called a spare. This is the way we draw a spare. A spare is scored as ten plus the which would be a 5. So the score for this frame is 15. 10 plus 5 and 15 plus 14 is 29. He gets a spare in the next frame and then in the fifth frame he rolls a strike. A strike is when you knock down all 10 pins on your first try. The score for a strike is 10 plus the next two balls blue. He rolled a gutter ball followed by a one. That's kind of a heartbreak when you've got a strike. His total for that frame was 11. 11 plus 49 is 60. Plus the one in the next frame gives him a 61. And so on and so on it goes until you get to the 10th frame. 10th frame was a little bit different. In the 10th frame, if you were to roll a one and a four like we did in the first frame, But this guy rolled a spare in the 10th frame, which means he gets one more ball to complete the spare. Had he rolled a strike in the 10th frame, then he would have gotten two balls to complete the strike. This is how you score bowling. These are the rules. Now, our goal for this particular example is to create a class named game. and a method named score the roll method will be called whenever you roll the ball and you will pass in the number of pins you knock down the score method is only called at the end of the game and it will

return the score of the game not to make this demo go a little bit faster we're not going to consider any invalid inputs we're not going to deal with negative rolls we're not going to deal followed by a 7 all those invalid inputs we're going to ignore normally of course we would not but to get this demo over with that's what we're going to do now like all good designers we begin with a brief design session we know we need a class named game which has two methods roll and score but what else do we need what are the other objects in this system well it's pretty easy to think game has 10 frames. You can see the UML here. The UML shows a game having 10 frames. And what is it that frames have? Well, frames have roles, one or two roles. One in the case of a strike, two in every other case. Except, of course, for the 10th frame. The 10th frame has two or three roles. How do we deal in an object-oriented design with a concept that is similar but not exactly the same? we have a subclass. The tenth frame. The tenth frame derives from frame and it contains an extra role. It inherits the one or two roles and has another role beside. So it will have two or three roles exactly right. Now let's think about the score function. What's the algorithm for the score function? Well, it's going to have to loop through all the frames, adding up the scores in the frames. some kind of method in frame so that that loop can ask the frame for its score that's probably the score method what is the algorithm of that method well it's going to have to loop through the rolls but it's harder than that because in the case of a strike or a spare it'll have to look ahead to the next frame in order to complete the game and complete the score so we're gonna us to the next frame. Okay, very good. We have a design. I wouldn't spend much more time on it than this. This kind of gives us the vocabulary of the system, the basic components. So now we're going to do test-driven development. And in test-driven development, we do something interesting. We ignore the design. Well, not quite. We don't ignore it, but we don't follow it. We use it as here you see on the screen the beginnings of a test bowling test and I believe you can see that it executes it executes with a function named nothing well that's because I always begin this way I always start a project with something executing so I'm already running even Okay, what test will I have to write? Now remember the first law of test-driven development. You are not allowed to write any production code until you have a failing unit test. I know the production code I'd like to write. I'd like to write public class game, but I'm not allowed to. I have to write a unit test first. So now I have to play this strange game with myself. that will force me to write the code I know I want to write. It's time for the red-green refactor cycle. And I am in the red phase. I must write a failing unit test. All right. What test must I write that will force me to write public class game? game. Game g equals new game. Oh, heavens. Thank you, IDE. Helpful IDE. That's enough of that. Heavens to Murgatroyd, that doesn't compile. Well, I must make it compile. So now, I'm going to create the class game. Yes, in the current package. There it is, the class game. Look, the test passes. It's time for me to go to the green phase. I am now green. And in the green phase, it's time for me to refactor. Is there anything I can refactor here? No. Nothing to refactor. Very good. Then I am done with this test. I am passing. Lovely. Back to the red phase. The next test. want to write that roll method. All right. Test

can roll. Oh, heavens, I need a game. Game g equals new game. Lovely. g dot roll. What should I roll? Well, let's roll a zero. That doesn't compile. ha look at that roll and it should take a pins this is lovely and i believe that that will pass because i i'm not actually testing anything oh time for me to refactor i've got a passing test uh what's three factor oh goodness i've got duplicate code here duplicate initialize it in the setup method which of course I don't have but this will create I'll keep it private lovely so now I've got a nice setup method it's going to create the the the game it'll put it in the public and the private field G can create game is now empty because well it doesn't do anything and I think I can get rid of that line of code in can roll and all of this should It does. I think I can get rid of that empty test now. This is common by the way, you write a test just to delete it later. Lovely. Next most interesting test case. Back to red. Well what can I make fail now? Oh, I got to write the score function. until I roll a complete game. Okay, so here we get into some of the technicalities of test-driven development. When you must write real code, you write the simplest real code you can. In this case, I'm going to have to roll a complete game. What is the simplest, most degenerate complete game that I can roll? A gutter game. Let's roll the gutter game. Okay, 4 int i equals 0, i less than how many balls in a gutter game? 20. 20, mm-hmm. i plus plus, g dot roll 0, that will roll 20 zeros, that is a gutter game. And now we will assert that the score is 0. Oh heavens, that doesn't compile. pass. Excellent. I will make this compile. It should return an int. I'll have it return a negative one because I still want to see it fail. It should fail. Yes. And now I will make it pass by returning a zero. This is stupid, but it's also easy. And I have now seen my test both pass I know my test works. It cost me nothing, but I now know that my test will fail if I don't get a zero. Ah, well, that was nice. What is the next most interesting test case? Or is there any refactoring to do? Mmm, no. Next most interesting test case. Well, I've got a gutter game. The next most complicated game. But the next game that might be interesting might be All Ones. That has a trivial score, a score of 20. So let's test All Ones. Wow, um, gee, I think I can just take this code here and cut it and paste it. Duplicate code, I have duplicate code here. Oh, alright, I'll refactor it in a minute. Hold on, I'm still in the red phase. Um, I'm gonna roll 20 ones and the score for that game will be 20. fail. It does. Expected 20 was zero. Now I must make it pass. How am I going to make this pass? Well, I could return a 20, but that's not going to make the first test pass. Now the first test will fail, even though the second test passes. This is not good enough. I think I'm actually like let's just add up all the pins in role so we'll make a private int score equals zero that's a nice variable inside of role we will say score plus equals pins and inside of score we will say return pins no return score lovely I I'm a programmer. Okay. By the way, how many of you guys are programmers because you got something to run once and you want that feeling again? I just got that feeling. So this seems to work. I mean, it's not the best algorithm I've ever seen. Seems to work. Let's go to the refactoring phase. Anything wrong with this code? Oh, duplicate code. that for loop is duplicated. Heavens, how did we allow that in this code? What a mess, it's duplicate code. All right, so I believe that I can extract a function that will contain this for loop. I'd like to

call that function roll many. In order to extract it, that zero, turn that into a variable named pins. I'm going to take the variable out of the loop. So now the loop is parametric. And then I will extract the method, call it roll many. It will take n and pins as arguments. It has detected that a similar batch of code is in please. Lovely. Here's the roll many call. There's also another one here in all ones. Isn't that pretty? I'm going to continue to refactor this by removing those variables now. I only needed them in order to extract the method. A gutter game. Roll 20 zeros. Score is zero. That's clear. All ones. Roll 20 ones. Score should be 20. That's pretty clear. I like this. I'm going to take this roll Oh, look at that can roll function up there. That's really not needed anymore. I'm going to get rid of it. Everything should still pass. Lovely. Isn't that wonderful? Okay. Next most interesting test case. I've got gutter game. I've got all one. I could do all twos, but I know it'll pass. And we don't usually write tests that we know will pass. that'll pass all fours that'll pass all fives won't oh because all fives would mean spares ah so the next test I'm going to write is gonna be about spares what's the simplest spare cast test I could I could enter here one spare one spare followed by gutter balls that's pretty simple all right one spare that's a spare ugly comment I've got an ugly comment there I'm gonna have to refactor it but not yet not yet I need one ball after the spare how about a three just for grins and then the rest will be gutter balls how many gutter balls do we need 17 and the score for this game is going to um 13 plus 3 16. yeah that three gets counted twice that's the bonus for a spare uh g dot score yeah okay uh pass or fail oh it's going to fail failed expected 16 was 13. i need to make a pass how do i make it pass I'm calculating a score here. If I get two fives in a row. No, no, not two fives in a row. If two balls together add up to ten. But I only got one ball here. Okay, okay. If pins plus last pins. That would be a static variable. I'll load it later. If that equals ten. That's the spare. Now, this should be bothering you at some very deep level. Your sphincter muscles should all be tightening. I'm adding some horrible flag here. This is bad. I don't want to add a flag. I don't want to have this static variable. So whenever the code asks you to do something horrible like this, you have to back away. Something must be wrong with the design. design. There's only two lines of code. Yes, but something must be wrong with the design. Actually there is. There's a design principle violated here. It's a design principle everyone knows. It's one of the first ones you learn. It's very popular. Anybody know what it is? Well I'll tell you what it is. Which of these functions by its name implies that it calculates Oh, the score function. Which of them actually calculates the score? Oh, the roll function. Misplaced responsibility is the name of this design principle or this design smell. Misplaced responsibility. You go to the function that says it does something and the function doesn't do it. How many times have you been caught by this? You've got to change some behavior in the system. function that says it does it and that function doesn't do it. Where is it in the system? It's somewhere but it's not in the function that says it does it. So where is it? The programmers get clever. They know how the system works. They hide the functionality in places that that it's not expected and then you have to figure out where the heck it is. We're not going to do that. We're gonna put the functionality in the

function that says it does it. We're gonna get score to score of course what that means is that i've got to save all the rolls i can't add up the score in rolls oh goodness i have some work to do what must i do uh i'm gonna refactor but wait i've got a failing test no i don't i'm gonna go back to my test and i'm going to ignore it tests pass now. There's a little reminder here that tells me that I ignored the test, but my tests are passing now. I can refactor. Let's us refactor. I want to save all these pins. I want to put them someplace nice. I'm going to put them in an array. Let's do this. Let's do roles in a game of bowling 18 19 20 21 21 and I'm gonna need some index for this array private and current role equals 0 good inside the role function I will say role sub current role plus plus equals pins proving once and for all score function I'm going to iterate over the array, ah yes, I'm going to say score plus equals role sub i. And then what I should do is just return the score. A little bit more refactoring, I don't really think I need this score variable anymore, I think actually Hmm, pass or fail? Pass. I have refactored. I have refactored the production code. The algorithm is entirely different. And now I don't have misplaced responsibility. Now the score function actually calculates the score. All right, back to red. Let's get the ignore out of there. This should still fail. it does for the same reason as before too so we haven't changed any behavior at all we've just moved where it went okay now where do we get this how do we get this to pass um okay um what i'd like to do is something like this if um uh roll sub i plus roll sub i plus one equals 10. That would be the spare. I see some of you shaking your heads. You don't agree with me. You don't think that's the spare. Oh, I see, because it might not be on frame. That's right. This test would pass if the last ball of the first frame and the first ball of the second frame added up to 10. That's not a spare. Oh, but maybe we could do this. We found everybody who did that years ago and we shot them all. No, this is not how we're going to do this. Actually, I think we need to refactor. Oh, but I can't refactor because my tests are failing. Back to the test. Ignore. Test should now pass. Why do I want to refactor? I want to refactor this loop so that it loops through the rolls one frame at a time, not one ball at a time. If I do that, then I could write that if statement. And that if statement would always be on frame. So let me rewrite the if statement, the for loop, excuse me. Let's see, int frame equals zero, frame less than 10. the game. Frame plus plus. Good. And I think what I'll do is I will. Oh, I don't have that I variable, do I? Well, I need to create it. Good. Okay, so now I am looping through the array. One frame at a time. I'm adding up the two roles in the frame. Oh, I I need to increment I, don't I? I plus equals two. So I'm actually walking through the array two balls at a time. One frame, two balls. Yes, I know. Strikes are different. But we haven't got a strike test yet. So this is good. Pass or fail? Pass. Another refactoring success. Lovely. All right. So. back to the red, fails, expected 16 was 13, same result as before, excellent, okay, now, can I make that spare case pass, well, I think that if statement will work now, if, equals 10. Ugly comment. And I should probably put an else here. What should I put in here? This is the spare case now. Score plus equals... Well, the score for a spare is 10 next ball you roll rolls sub I plus 2 and what should we increment I by well there's two balls in this frame pass or fail to do this is

pretty ugly code I mean I I no no no we need a better name than that let's see what is this I variable it is the index of the first ball in the frame well that's what it is but that name is awful so let's back it up index Mmm, first in frame. Ooh, first in frame. I like that. First in frame, that's pretty nice. Okay. This comment, I need to get rid of this ugly comment, which is easy to do. I'm gonna take that expression right there and I'm going to remove it as a method named is spare. Oh, isn't that nice? If is spare. rid of the ugly comment because it's superfluous. This should still pass. Yes, it still passes. I can go to my tests. Oh look, another ugly comment. Well, I can refactor my tests, which by the way we always do. We treat the tests just as though it were production code. We do not treat tests as though they were second-class citizens. Tests are as important as production code and you keep your test clean. The name of this function? Roll spare. There, that's nice. And I no longer need that comment. It's entirely redundant. I believe that will still pass. Yes, it still passes. Hallelujah. All right. I'm going to take this function. I'm going to move it up a little bit where all the utilities are. Good. So now it's up here with roll many. case well we did one spare let's do one strike test one strike G roll 10 ugly comment that is our strike though now we need two more balls after the strike G How many gutter balls do we need? Oh, hello Twinkles. Hello Twinkles. Twinkles has come to visit me. Yes, she is. She's a good little worm animal. That is the worm animal. This is neither red, green, or blue. This is some other color. It's called making your little dog happy. Yes. Oh yes. Good dog, Twinkle. That's a good dog. Yes, yes, yes. Good dog. Good dog. Okay. Yeah, that's a good little worm. This is a good little worm dog. Yes. All right. You want to sit on my lap? Go ahead. Sit on my lap. Or do you want to get down? All right. Down you get. Thank you. Thank you for the visit. Bye-bye. Okay. We need to finish this by rolling gutter balls. Roll many. How many gutter balls? Well, there's eight frames left. 16. And what is the score of this game? It's 10 plus the next two balls. Actually that 7 there, that gets counted twice. So that would be 10 plus 14, 24. Pass or fail? I've got my red hat on so it better fail. Fail! Yeah, it's expected 24 but was 17. Oh! Green. Well, we've got a kind of design pattern cooking here, don't we? We've got this if statement. I think I could put another if statement in. If role sub first in frame equals 10. Well, that would be a strike. Ugly comment. Better put an else in front of that if. Good. Well, the score for a strike is 10 plus the next two balls. 10 plus roll sub first in frame plus 1. That would be the next ball. Plus rolls sub first in frame plus 2. That would be the next ball. And what do we increment first in frame by? Oh. There's only one ball in a strike frame. First in flame. Plus plus. Pass or fail? Pass. Refactor. Heavens. Terrible, terrible stuff here. We've got an ugly comment. Well, we can get rid of that. That's going to be a is strike method. Excellent. Don't need that comment anymore. Horrible, horrible expression. Too much stuff going on. Let's take that. Take that and extract it as next two balls for strike. Yeah, that's good. Next two balls for strike. Okay, fine, fine, fine. Oh, since we've done that, we should probably do this. That would be next ball for spare. That makes things nice and symmetrical. No, don't do that for me, please. Now, this looks completely out of place. very large so let's take that and make it two balls in frame no don't do anything else

thank you okay that looks pretty good let's go to the tests by the way I think this still passes yes it still passes oh goodness we've got horrible code in the tests there's this strike method let's take that out and the horrible comment you know how I feel about comments you saw that in a previous episode hmm this should pass hurrah I'm gonna take that roll strike method and move it up with all the other Next most interesting test case. Well, we've done strikes, we've done spares. We should probably think about multiple strikes or spares. And in fact, we should think about the 10th frame, because we've completely ignored the 10th frame so far. So there's an obvious choice here. Let's do a perfect game. Perfect game is a whole bunch of strikes. Everybody knows the score. Let's see. Perfect game. roll many how many strikes in a perfect game 9, 10, 11, 12 12 12 tens that's the perfect game and the score for that game is everybody knows 300 okay pass or fail should fail right pass wait I've got my red hat on that should have failed passed? Why did it pass? Well, let's look at the code. Four. Every frame. All ten. If the frame is a strike, the score is ten plus the next two balls. Otherwise, if the frame is a spare, then the score is ten plus the next ball. Otherwise, the score is the two balls in the frame. That reads like the rules of bowling. Okay. Break the suspense. We're done. This is the algorithm. The algorithm is a for loop and two if statements. The algorithm is roughly 14 lines of code. We could probably shorten it a little bit, but this is the algorithm. It's actually absurdly simple. Why didn't we know this going into it? I mean, do you remember that? We were going to do that. We were going to do that. Imagine if we had done that. We could have drawn this on the board. We could have said, okay, I'll do the game class. You do the frame class. You over there do the 10th frame class. You do the roll class. We'll all meet back here next Tuesday, and we'll integrate with 400 lines of code, and we'll make it work. And we would have, too. We would have made it work. Without ever knowing that this was a for loop and two if statements, of code. We snuck up on this problem. We snuck up on it by doing a set of steps, each one too simple to think about, each one so stupid it wasn't worthy of much intelligent thought. And at the end we had solved the problem and we didn't even know we had solved the problem. Does this always You will go sit down and do tests, and lo and behold, the tests will follow the design, and everyone will be happy, their tests and their design are in concert. But it happens often enough that you do your design, and of course you always should, but then you start writing tests, and the tests take you in a completely different direction. I did this many years ago, 10 years ago, maybe a little more. in a hotel room during a conference. I believe it was a C++ world. And we did exactly this. We actually drew this diagram on a napkin. And then we started doing test-driven development. It was one of our first times doing it. And we struggled around a lot. We tried to follow the design. We tried to write the tests for the role class. But we found there were no tests. Nothing, no behavior that you could test. So we backed up. We went to the frame class. There must be tests there we could write. but we found there weren't any behaviors for the frame class that we could identify. No tests we could write. So we backed up one more time to the game class, and then we found behaviors. And we began to implement one test at a time, the

way you saw me just do. And slowly that for loop with the two if statements started to peek out of the mass of code we were producing. And I looked at it at some point and I said, my God, Bob, that reads like the rules of bowling. into place and we sat there in awe that the design we had written on that napkin had not come to fulfillment, but instead a much simpler algorithm had shown up. It still sends chills down my spine today. So there's the bowling game. That's the famous demo that has been done so many times Thank you. Yes, Uncle Bob, all very convincing. But as we all know, there are many logical objections to test-driven development. It is now time for you to answer those objections. Sure, go ahead. Hit me with your best shot. code Uncle Bob. By following test-driven development you write much more code than normal and therefore test-driven development must slow you down. No, test-driven development makes you go fast. First of all, you spend a lot less time debugging. I mean, think of how much time you could save if you didn't have to hunt bug after bug after bug. But that's actually small potatoes compared clean code base. Test-driven development allows you to clean your code so you don't suffer the impediment of constantly dealing with everybody's mess. If you want to go fast, practice test-driven development. Logical Uncle Bob, but unsubstantiated. Verifying these facts will require trials and experiments. And that brings me to my next objection. There are Uncle Bob, who either through ignorance or disagreement, will never allow their programmers to practice test-driven development. But why should your boss have to know? I mean, do you ask your boss if you can use semicolons? Do you ask your boss if you should indent your if statements? When's the last time you asked your boss if you could go to the bathroom? Test-driven development is a personal practice. and still they will object. If your boss questions you about test room development, tell him you do it in order to go faster. If he objects, tell him that you'll have to double all your estimates. If he continues to object, then tell him to mind his own business because after programmers behaving as professionals? I've not thought it possible. My next objection, Uncle Bob, is that refactoring is rework. It would be better to write the code correctly the first time. Well, when you figure out how to do that, you let me know, because I'd be real interested. Meanwhile, every creative effort on the planet is done iteratively. Producers don't paint the perfect portrait the first time. Songwriters don't write the perfect song the first time. Journalists don't write the perfect article the first time. And programmers don't write perfect code the first time either. When you were in third grade, your teacher probably asked you to write a story. And then she told you how to do it. She said, first write an outline, then write a rough draft, and then maybe the final copy. We are taught from a very early age that creative work requires iteration and rework. And then, when we become a programmer, we can alter reality and write everything correctly the first time. I mean, that's laughable, right? Of course it's going to take several passes to get the code right. and rework to do things well, of course we're not going to write it right the first time. Yes, Uncle Bob, I entirely agree. All constructions of logic must be approached iteratively. But now, tell me this, Uncle Bob. Who tests the tests? We have two streams of code, the tests and the production code. production code, but the production code tests the tests. I see. The two streams

of code embrace each other in a mutuality of testing. Fascinating. But now, Uncle Bob, I have been told that a single change to production code can cause many hundreds of tests to break. Well, only if they're poorly designed. Design your tests well, and if you discover better designs, refactor your tests. Tests are just as important as production code. Maybe they're even more important. So treat your tests the way you would treat your production code. Don't allow them to become a mess. Remember, if a single change to the production code causes a whole bunch of tests to break, wrong with your tests. Your tests are too coupled to the production code. So find some way to introduce some abstractions and other decouplings into the tests so that when you change the production code in one place, you don't get a whole bunch of tests failing. A design for tests. of bugs. But tests cannot prove the absence of bugs. A suite of tests will not prove that this program is correct. Well, look, our goal is not to prove the software correct. You're right, that's infeasible. Our goal is to create a parachute that eliminates the fear of making change. We may never achieve 100% certainty, but 99.999 is pretty damn good. The point is well taken. Absolute certainty is desirable, but seldom achievable. Uncle Bob, test-driven development is a dogma, a discipline. It encourages people to follow rules instead of using their minds and thinking. Well, yeah, that's true. I mean, that's what disciplines are. The reason we follow disciplines is so that we don't have to keep making the same decisions over and over again. Now that's both a strength and a weakness. I mean, we want a certain amount of dogma because, in the heat of battle, we want to be able to fall back on our training and move effectively and efficiently without constant questioning. a step back and re-evaluating your dogmas and disciplines. You might make significant changes. You might make subtle changes. Either way, it's both appropriate and it's necessary. Consider CPR. We learn the dogma and the discipline of CPR so that in an emergency, we don't have to think about it. We can just get on with the business of saving someone's life. CPR is constantly being revised. If you get certified at CPR, you have to re-qualify every few years so that you're kept up to date on the changes. I have indeed noted that humans often require discipline. And it's also true that discipline does not preclude thought. So now, Uncle Bob, to gain this suite of tests, It is the tests that matter, not when we write them. Balderdash! If you write your tests at the end, they're not going to be complete, and you're not going to trust them. This is just simple human nature. Humans consider things that come first to be important and things that come at the end to be less important and somehow optional. That's why they're at the end, so that we can leave them out if we have to. When you write tests first, you trust those tests, because every line of production code is tested. Tests come first because they're important, more important perhaps than even the production code, because the tests are what makes the production code flexible. Without the tests, the production code rots. things they consider optional or undesirable. If the tests are critical, the tests must come first. So, Uncle Bob, there is a vast amount of code already written and it does not have a suite of tests. What do we do about that, Uncle Bob? In his wonderful book, Working Effectively Legacy Code as code without tests. It's hard to get Legacy Code tested because it wasn't

designed to be testable. In order to test it, you have to make a series of small decoupling design changes. But without the test, you've got no way of telling if your design changes have broken something or not. This leaves you in a catch-22. refactor without tests. Michael's book is full of good ideas for how to break through this logjam, but the upshot is simply this. Find some small part of the legacy code that you can test without making big changes. Then use those tests to extend the design changes more safely. I wouldn't start a big project to do this. factoring or big testing projects. Instead, I would wait until a new feature was needed in some part of the legacy code, and then I would change a little bit of design and add a few tests in the affected area. Month by month, year by year, this slow and gradual process will eventually spread tests throughout the bulk of the legacy code. Oh, there's some parts of the legacy that you'll never get tested. important volatile bits will almost certainly all get tested. And of course all new code you write, you write test first. And in those situations where you have an option either to modify old code or write a new module, write the new module so that you can write it test first. Always prefer test first. We're going to talk much more about legacy code in a future episode, so watch for that one. Uncle Bob, how do you test graphical user interfaces? Well, for the most part, you don't. I mean, GUIs are the kinds of thing that you fiddle into place. And you move a text box a few pixels to the right or the left. You change the shade of green on a particular button. You change a radio button into a text box. You alter the font of a line of text, or you change it from bold to italic. This kind of fiddling is hard to test drive because you're not sure what you want to see on the screen. So you experiment and you fiddle and you mess around and massage until you like what you see. However, this only applies to that last thin outer layer of the GUI where all the formatting stuff is. There is code one layer back from that that you can and should test. For example, the decision to gray out an inactive button should be tested. I don't necessarily need to test that the button itself was rendered in gray. I just need to test that the Boolean that controls that state was set properly. The format of a currency or a date can be tested. The elements of a pull-down list can be enumerated and tested. tested even if the final format cannot. We do this by separating the GUI into two components, the presenter and the view. The presenter interrogates the business objects and then constructs data structures that the view then renders. The presenter knows which buttons are active and not active so it sets the appropriate booleans in those data structures. The presenter of the pull-down items and it puts them into the data structures. The presenter understands how to format dates and currencies and other items and it puts the formatted strings in the data structures and of course the presenter can be tested. Some folks like to write Vue tests and if they do they typically write them after the the position's just right, the color's just right, and then they'll write a set of tests that make sure that those things don't change. I don't usually do this personally, but there are many folks who do. How then do you test databases? For the most part, you don't. Databases are black boxes that you usually don't have to test. What you do want to test is that your schema and queries work as planned. The first step is to separate the application from the database using the layering technique

that we talked about in episodes 4 and 5. That will allow you to substitute mocks and stubs for the database and then test the application without the database actually in place. Then you can test the database independently from the application. You can test, for example, test that the generated queries behave properly by loading a database with a minimal number of rows and then executing those queries. Some programmers complain that testing is not part of their job description. They are programmers, they assert, not testers. They need to grow up. Some programmers Would they like some bread and cheese with that wine? Those last two answers appeared to be sarcastic humor, indicating that the objections were not worthy of a direct answer, a conclusion that I concur with. Uncle Bob, this has been a fascinating session. You have answered my objections with logic that, on the surface, appears sound. and I will have to change my conclusion about test-driven development. Uncle Bob, I shall consider it. In the past I have made the statement that test-driven development may be a prerequisite for professional behavior. the feelings of some good friends of mine, some friends whom I hold in high regard. So, in this segment, allow me to explain my reasoning. Software is a sensitive discipline. By that I mean that there are single bits within the binary of an application that, if set the wrong way, will crash the system hard. Very few systems are that sensitive. We could go out to a bridge or an overpass and start removing bolts. That bridge probably wouldn't fall down right away. There is no single nail in your house that you could pull out and cause the house to fall down. There is no single cell in your body which, if you kill, kills you. it. There is another discipline that is just as sensitive as software. Accounting. Accounting is sensitive to single-digit errors. If I make a mistake on a single digit on just the right spreadsheet at just the right time and place I can take the company down and send the executives developed a discipline called double entry bookkeeping. It now carries the force of law as part of the generally accepted accounting principles. The discipline is remarkably simple. Every transaction is entered twice. Once on the debit side, once on the credit side. The two follow completely separate mathematical pathways until they combine in a magical subtraction on then somebody made an error somewhere into one of those transactions. Of course this process is not perfect. It is possible that you could make complementary errors that yield a false zero on the balance sheet, but the odds against that are appalling. For that reason, double entry bookkeeping is considered appropriate due diligence for practicing accountants. bookkeeping. Everything is said twice. Once on the test side, once on the production code side. They follow separate but complementary execution pathways and they meet at a successful execution, a green bar. Is our source code somehow less important than the accountant's spreadsheets? Do our employers depend less on the accuracy of the software than they do on their financial programmers somehow less severe or less expensive than the errors made by accountants? The answer to all these questions is no. So then, how can we programmers treat our work with any less respect than accountants treat theirs? Shouldn't we programmers, as professionals, have a due diligence that is at least as stringent By the way, accountants do not enter all the debits first and all the credits last. They instead enter them all at the same time, keeping the

balance sheet always zero. Writing your tests after you write your code is a lot like entering all the debits last. You'd never know which entry caused the balance sheet to go out of balance. When a team of professional developers releases their code to QA, what should they expect QA to find? Nothing. Of course, sometimes QA will find some defects. But then the team of professional developers should be aghast and chagrined. The goal of a professional development team is that QA will find nothing. This is just basic software ethics. You don't ship code you don't know works. Shipping code you're not sure of is irresponsible and unprofessional. I know of no better way to know that your code works and to make sure that QA finds nothing than to practice the three laws of test-driven development. How much of your code should be covered by tests? Should it be 50% or 60% or maybe 80%? Frankly, I think the question's absurd. I mean, what's the point of covering any less than 100%? After all, if you only cover 80% of your code, you don't know if 20% works. There's no way that's acceptable. you realistically cover 100% of your code with tests? Well, probably not. But that doesn't mean you should accept a lesser goal. 100% is the goal. No lesser number even makes sense as a goal. The fact that you can't attain it doesn't mean you shouldn't push as hard as possible to get to it. sound pretty good to you, but that doesn't mean I don't try to push it higher. I do. You will certainly have to ship with a number that is somewhat less than a hundred percent, but you should never be satisfied with a number less than a hundred percent. Vienna, Austria. Among other things, he was responsible for two free labor and delivery clinics where poor women could come to have their babies and receive care. In that first clinic, sometimes one out of every six women would die of a horrible disease called childbed fever. In the second clinic, however, no more than one in 12 ever died. The primary difference between the two clinics was that the first was a training facility for young doctors. The second was staffed by midwives. Could the difference in mortality rate be due to the doctors? Semmelweis noted that the doctors frequently conducted autopsies before going to the first clinic to conduct internal examinations on the women. the blood and tissue from those dead bodies, he called it cadaverous material, might be getting into the women and causing those fevers. So he instituted a policy of hand-washing. Before doctors could visit patients in the first clinic, he had them wash their hands in chlorine bleach and wonder of wonders the death rate dropped to less than one in a hundred. He same policy for the midwives and the death rate dropped in that clinic too. He started washing the instruments that were brought into the clinic and the death rate dropped even more. Semmelweis had discovered the miracle of antiseptic procedure. Of course the medical community adopted these procedures at once and hailed Semmelweis as a hero. Not! Actually they rejected his outright. In their view, disease was not caused by tiny little particles on the hands of physicians. Disease was caused by an imbalance of humors and bad vapors which could be corrected by proper bloodletting techniques. Wash our hands? Preposterous! We are gentlemen, sir. Our hands are clean already. it would take to wash our hands all the time? Why, we'd hardly ever get any code written at all. It took 60 years for Semmelweis' ideas to be accepted by the medical community. He died a broken and dispirited man. He saved

thousands upon thousands of lives and yet he was rewarded Even today, medical professionals, including doctors, fail to heed Semmelweis' advice. This very day, patients in hospitals around the world will die because careless caretakers were too busy to wash their hands. Fascinating. A totally parochial attitude. So, what have we learned? Well, first we learned that code rots because we're afraid to clean it. And that to keep a system clean, we need to somehow eliminate the fear of change. And that only a suite of tests that you trust with your life can eliminate that fear. Number one, you're not allowed to write any production code until you have first written a failing unit test.