Welcome, welcome to episode 19 of Clean Code, part 2. You remember back in part 1, we talked about the three laws of test-driven development, case study where we demonstrated those three laws in operation. And we also talked a little bit about the single assert rule. We'll talk more about that in an upcoming episode. Now for part two, we're going to discuss incremental algorithmics. The idea that we can construct algorithms one tiny step at a time by making a test fail and then making it pass. But there's no time to wait because this episode's already running. for episode 19 part 2 already in progress. Psst! Come here. You want to know a secret? The secret of test driven development? Well here it is. As the tests get more specific, the code gets more generic. the code gets more generic. This is a principle of such profound importance that it requires a story. This is my son Justin. He's currently in New York City working at a startup developing an iPhone domination. But back when Justin was in sixth grade he came to me with some math homework. He had to find the prime factors of some integers. I told him, son, you go to your bedroom, you do the best you can, I'll write you a program that will allow you to check your work when you're done. I wasn't gonna do the work for him but I did want to help him a bit. So off he went to do his homework I said rhythm that that guy from Greece came up with like a zillion years ago like What was his name? Yeah, his name was there at astonish Use the sim of eratosthenes that that'd be cool Yeah, that's pretty much what I had planned to do it so I decided to let the tests drive the solution instead of impose at the moment more people know that language so we begin as usual with a test that tests nothing so we need to first of all create the package and then we're gonna test is going to be the test case of two. Let's copy the first test. And we will say of two, so the prime factors of two, just a list with a two in it. Oh, look, this doesn't compile. And that's because our, our list function doesn't take an argument. So let's make it take an array of there that's nice and this should fail it does it expected list with a two in it but it returned the empty list and i can make this pass by taking the empty list in the of we will say factors.add2. And that should pass. And it does. How nice. Even more fun than making them pass, I think. Changing that 2 to an n was a generalization, wasn't it? And yet with each new test case, the tests become more and more specific. But we're moving the code in the opposite direction. we're making it more and more general. Fascinating. But the next test, the test for 4, will present something more of a challenge. Yes, you're probably right. If we assert that the prime factors of 4 are... Yes, okay, very nice. But there's something strange about this. Look, I've got two if statements that are identical. same predicate. And you know what's weirder about this? I can take that if statement and bring it down out of the first one, and I think this will still pass. Look at that. So I have two if statements in a row with the same predicate. Now, that kind of looks like an unwound loop to me, doesn't it? Well, I don't know what to do about that, so let's just continue on because we got mess here yeah what a mess you're just hacking at it now aren't you you don't have any plan at all do you gee uncle bob i i sure wish you'd implemented my plan because it sure doesn't look like you're getting anywhere like just just try anything to get something to work but i i think you ought to step back and

like you know think it through oh ye of little faith watch this so the next test is going to be five that's going to pass. How about that? Okay, well, assert prime factors of 6, and the prime factors of 6 are a list with 2 and 3 in it, and that passes. Well, we're going to assert that the prime factors of 7 is a list with just a 7 in it, So, uh, that hacked up algorithm you were complaining about seems to be a little bit more general than you thought, doesn't it? Hmm. Nah, you just got lucky. Oh, really? Well, if that was all just luck, then I should have a pretty hard time passing the next test, shouldn't I? Just you watch this. The next test is assert prime factors of 8 is a list with 2, 2, and 2 in it. And, of course, this fails because nothing we've done can put three things in the list. Now, let's play a little game of golf, shall we? What's the fewest keystrokes to make this test pass? Well, what about that? and if to a while that's a generalization isn't it whoa when I sat at my kitchen table back in 2001 and I made that change a chill went down my spine in those days I didn't know the rule But I did know that a while is a more general form of an if, and an if is a degenerate form of a while. So the pieces started to come together for me. Now, watch this. So the next test is for 9. And that's just a list with 3 and 3 in it. Now, of course, this fails. pass? Well, hmm, notice that this is a little engine that factors out twos. Now, it seems to me that what we need is a little engine that factors out threes. So, let's just plop that right in there like so. We'll duplicate that engine, and we'll take all the twos, and we'll Excuse me, Uncle Bob, but it would appear that you have created duplicate code. Indeed I have, Data. Indeed I have. And notice that this little engine that factors out threes has made the code specific to the test. That's not a good idea. So what we'd like to do is eliminate the duplication and make this more general. Now, looky here. This, these two while loops in a row, that looks like an unwound loop, doesn't it? So let's rewind the loop. Let's get rid of our little engine that factors out three, and instead put our little engine that factors out two into a loop. We'll do that by factoring out the two into a variable named divisor, and we will take that initializer of the divisor up out of that if statement, statement into a wait for it wait for it Wow and now all we have to do is increment that divisor and that puts our little engine into a loop now will it Dudley. Well, let's clean it, shall we? So, we can clean up this inner while loop. Let's see. I think we can take that and make it a 4 instead of a while. It doesn't really have an initializer. But it does have an incrementer, which is just this, really. We'll take that and plop it in there. going to work, get rid of that semicolon, and if I run that, I think it'll work, and of course, that means I can get rid of these horrible braces. My mission to destroy all braces is succeeding. I think I can get rid of that for loop, and in this case, we definitely have an initializer, that and we get rid of that semicolon and I think that's still going to work yeah so we can get rid of more braces and that should still work and now looky here this for loop will only terminate when n is equal to 1 and so I condition of an unwound loop, which I have now wound, and that should still pass. So, I mean, holy cow, look at that. We got three nice little lines of code in this algorithm, and all the tests are passing. Okay, Uncle Bob, but you're not done yet, are you? I mean, what's the next test case? You want another test case? You want another test case? Well, I'll give you another

test case. How about this one? Cert prime factors of 2 times 2 times 3 times 3 times 5 times 7 times 11 times 11 times 13. There you go. There's a bunch of prime numbers. 2, 2, 3, 3, 5, 7, 11, 11, 13. There, that's going to be a list of prime factors. Now what do you think? Pass or fail, guys? What do you think? Pass. And we're done. Those three lines that you just saw the prime factors of an integer that there is. If you look closely, you'll realize that it is the sieve of Eratosthenes. Oh, there's a minor improvement you can make. You could terminate the loop at the square root of n, and that makes it a little bit faster for big primes. But overall, those three lines, that's the algorithm. If you were to play that session over again, watch what happens to that first if statement, than one statement. It's a while loop now, but it didn't start out as a while loop. We laid the seeds for that while loop by writing that hacky if statement. In fact, all the hacky stuff we did in the first few steps, it's still all there. It's just better now. It wasn't wasted. It wasn't wrong. It was just too specific. Isn't that cool? to ask you a question. Where did that algorithm come from? It came the way the solutions to all problems come. Through a sequence of incremental generalizations. And so what do we say, boys? As the tests get more specific, the code gets more generic. The code gets more generic. If you've been doing test-driven development for any length of time, then you've probably experienced getting stuck. is a technical term that means there's nothing incremental you can do to pass the currently failing test. Getting that test to pass forces you to write a whole big bunch of production code. In extreme cases you just have to write the whole damned algorithm. Yeah, man, I mean Motor yard or was it engine garden? I forget but like this one time man I wrote this Humongous test. I mean it was a monster Pages and pages of test code and then and then to make that pass Like I had to write mountains of code Yeah, I get stuck all the time too Sometimes I'll write some simple little test, and the only way to make it pass is to just write the whole program. What's up with that? Oh yeah, I mean, that happens to me all the time. In fact, it happens so much, I just gave up writing tests first. Now I write them all after I'm done. Yeah, getting stuck can be frustrating, and if it happens often enough, it can get very discouraging, you might actually give up on test-driven development altogether. Or maybe you'll write your tests after the fact, or maybe you won't write any tests at all. But there's a better way. You see, getting stuck is a symptom of a problem. Now that problem could be that you wrote the wrong test. Or maybe you're making the production code too specific. Or maybe it's both. this is to demonstrate it so let me show you the word wrap problem now the word wrap problem is the writing of this function wrap which takes an input string and a width and returns an output string the nature of the input string is sort of like this the Gettysburg Address it's a sequence There's no punctuation, there's no multiple spaces, just words separated by spaces. And the other argument is the width argument. The output string will be the input string, except that line ends will be inserted at strategic places, so that no line is longer than the width. Now, those strategic places. to choose the most appropriate space to replace with a line end unless of course you run across a word that is longer than the width and in that case you will have to split the word now first I'm gonna

do this wrong and get us stuck and then we'll back up and we'll do it right we begin as wrapper package, let's say. And a class named wrapper test. And a test function that does nothing. And we wire it into our environment. And then we make sure it runs. Like so. All right, let's pose our first test. First of all, let's change this to should wrap. And then we're going to pose an assertion here. Two words separated by a space. So the outcome of that would be return word. Word, word, comma, four. Okay, so that should wrap to words. Now, we don't have a wrap function, so let's make one. There it is. It should return a string. S is good, but I should be width. And have it return null. Of course, the assert needs that on it. a static import so that it's not so ugly, and now this ought to fail. And it does, of course. But we can make it pass by doing this clever trick. S dot replace all space with line end. I think that'll pass. Ah, we're done. Illogical, Uncle Bob. You did not begin with a degenerate test case. Right. I went for the gold, didn't I? That's always a mistake. A good test-driven developer always approaches the problem from the outside in, testing the most degenerate things first, all the error conditions and boundary conditions, all the simple stuff around the periphery of the problem before he goes for the guts of the algorithm. you stuck. Oh yeah, just you watch. So for our next test, let's do a sentence with a space that does not need to wrap. A sentence like a dog. And we will wrap that. A dog. And make sure fail. Now, how do we make that pass? I suggest you be very careful, because it looks to me as though you're about to get stuck. Now, wait a minute, Sherlock, wait a minute. Don't get all hot and bothered yet, because I think there's a way out of this. every space with a line end. But this line here, a dog, does not need that space replaced. And why not? Well, because it's too short. So look, let's check the length of the line. And if the length of the line is greater than the width, well then we'll replace all the that will work. Yes, that works. See, we got out of the problem. Captain, we are being surrounded by a web of specific production code. Our generalization factor has been lost. If we do not back out of this situation rapidly, we will find ourselves quite stuck. Scotty, give me full reverse power now! Okay, hold on you hold on you know let's just try the next test okay let's do a sentence where some of the spaces break and some don't so a dog with a bone wrap warp a dog with a bone six okay so one two three four five six yeah that one should break one two six. Well, that one should break, huh? Good. Now, look, this should fail, right? Ah, it fails. Okay. Now, how the heck do we make this pass? Emergency reverse! Emergency reverse! Scotty, I need that power! Where is that power? Too late, Captain. The web is complete. escape is now impossible. Oh well now you've gone and done it! You've gotten yourself well and truly stuck! It should be clear by now that the only way to get these tests to pass is to just rewrite the algorithm. I think we're going to have to throw away everything we've done and write the whole thing over from scratch stuck. Yeah, yeah, stuck. Okay, now let's do this right. Let's start over. And we'll write the most degenerate test cases we can. We will very gradually climb the ladder of complexity one little test case at a time it pass by generalizing the production code rather than doing some specific fix just to get the test to pass okay so we just did this wrong didn't we let's get rid of these tests let's get rid of this implementation return it

back to the way it was no good let's close that window we really don't need that all right so that the wrap of no hmm with I don't care what that width is should always return a empty string let's make that an assert dot right okay yes all right and very nice very nice and let's try this search equals empty string again another degenerate test case in this case it'll be that empty string in doesn't matter what the width is regenerate test case, just a single character, that's it, rep x comma 1, okay, and I think this is actually going to fail, isn't it, let's see, yeah, that fails, although, boy, it's really easy to make this pass, isn't it, because all I have to do is return s, and in every case, I think that's we're gonna need how about this if s is equal to null then return empty string otherwise return s clean up those tests mister aye aye sir they are a little bit ugly aren't they so let's do this let's pull out this which is the expected let's call that okay expected that's fine and let's pull out this that's the input string s fine and let's pull out that that's the width yes fine width okay good and now let's take and let's take the width and the string and move them before expected so that it reads a little better. Okay, and yes, you may replace all of them. Thank you. Very good. And now, of course, we can inline the variables that we use just to hold the places. and now we see that the should wrap function reads much better. Assert wraps null comma one is the empty string. Assert wraps empty string one is the empty string. Assert wraps x one is x. So it reads in a more intuitive order. So now let's assert wrap something else. The first I think is that one. And it should break like so. So we're breaking a word that's longer than the width. I believe that will fail. And it does. So how do we make that pass? Well, we're going to return s if s.length is less than or equal to width. But if it's not, well, then we've got to split it. And we can split it pretty easily. Return s.substring of 0, width plus a line end plus s.substring of just plain width. the line end and the tail. And if I've done this right, it'll pass. Ah, yes. Excellent. This is an absurd procedure and counter to all established in conventional wisdom. Every good architect knows that you solve the riskiest parts of the problem first. You don't dance around outside the problem solving trivialities. You dive into the guts of the first otherwise they'll come to bite you in the end. I completely agree that we need to mitigate the greatest risks first. The trick here is to figure out what the greatest risks are. The greatest risk to test-driven development is getting stuck. We mitigate that risk by assiduously avoiding the behaviors that lead to it, specifically the behaviors that leap into the complexity of the to gradually increasing and incrementally stepping into that complexity. So, is it time for some spaces yet Uncle Bob? I want to see some spaces! No, it's not time for spaces yet Jerry. Right now there's a simpler test for us to pass. It's the test of multiple lines. We can test this case by choosing an input that has no No spaces, but can be broken more than once. Three X's ought to do it with a width of one. That'll force it to break in two places. X, return, X, return, X. And that, of course, should fail. And it does. Now, how to get this to pass? Well, let's play a little game of golf. What is the quickest, shortest, fewest keystrokes? And that will be... to put that last call to substring inside a call to wrap. And I believe that will pass. Yes! Whoa, man, that, like, blows my mind! Like, I thought you were going to write a really big loop, but that's really cool! may be but

the algorithm is not in tail recursive form given sufficient input it will exceed the capacity of the stack yeah Spock that's true and Java doesn't do tail call optimization anyway so it's kind of a moot point but you'd really need a very large input to blow the stack here and for the purposes of this demonstration I think we can safely ignore it remember the old adage you know First, make it work, then make it write, and then make it small and fast and memory efficient. Is it time for a space yet, Uncle Bob? I mean, I really want to see you breaking a space. Yes, Jerry, it's time for a space now. The next test is going to test the case where we have to break at the last character of a word is a space. We can state this problem as a test by writing a sentence that has a space in it right after the breakpoint. So that one would be right before that space. This should wrap to x return x, of course, and one would expect this to fail. It does fail, and the reason it fails expected x return x, but what it returned was x return blank line return x. Now, the reason it did that is because this tail right here has a space at the beginning of it. We can get rid of that space by playing a little golf. We can trim it right there, and that will eliminate that space Excellent. Oh wow! We're done? I mean, you've handled spaces with just a trim? I'm just so impressed! Well that's a bit premature, Danny. We're not done yet, I don't think, anyway. There's at least one more test case to examine here. So, in that case, what we want to do is find the last space after the width, and then back up to it and break there. We can force that situation with a test like this. So that 3 will put us in the middle of the second word, the second two x's, and then to create that x return xx now i believe this will fail of course it does now to make this pass what we're going to have to do is right here we're going to have to put some braces in you the last instance of a space. And that's going to be equal to s.lastIndexOf space, and we will start that search at the width. So that'll search backwards until it finds the last space prior to the width. Now, if that's not found, for negative, then we want the breakpoint to be just width. And now what we should do is change all these widths in the substring statements to breakpoints. And this one too. But of course not this one because that's the argument to wrap. And if I've done this correctly, excellent! Excellent! Now, Let's try something. I have in my paste buffer an acceptance test based on the Gettysburg Address. Why not? And you can see, here it is, four score and seven years ago, blah, blah, blah. We will break it at seven, and we will find all the necessary places to break it. And what do you think? Is that going to pass or fail? Well, pass. Are we done Uncle Bob? Are we done? I mean, this is just so exciting! Yes, Danny, we're done. That is the Word Wrap algorithm. Fascinating. It would appear that if you write the tests in the correct order, and continuously generalize the production code, the algorithm all but writes itself. Consider this Venn diagram. The big rectangle on the outside represents every possible behavior of every possible program. Now we're doing test-driven development and as we begin we all the behaviors allowed by the test. Now the test is very small so it doesn't do much constraining so that gray shape is very large. The dashed circle that you see represents the set of desired behaviors. These are our requirements. This is what the tests are trying to force us to turn our program into. And of course the production code initially fails even So that

little dot off to the left represents the production code outside of the behavior of the tests. When we make that first failing test pass, we bring the production code's behavior inside the bounds of the constraints of the test. Now that new behavior barely intersects with the desired goals at all. behaviors. We make that test pass by generalizing the production code. But this does much more than just make the test pass. It expands the behavior of the production code. And so we write a new test which further constrains the behavior of the program. And then we make that test pass by generalizing the program. And then we do it again, another test which further constrains generalizes and we continue this and so it goes. Step by step we tighten the constraints of the tests. Step by step we expand the behavior of the program by generalizing it, gradually approaching the desired goal until finally that goal is met. But look at the tests. When the goal is met, tests. Tests cannot fully constrain a program. They can add constraints but they can't specify its final behavior. Tests can only prove a program wrong. They can never prove a program right. And if this is so, then how did we meet the desired goal of the program? The desired behavior is desired behavior is and so you gradually increase the generality of the code until that desired behavior is achieved and this takes patience it takes creativity it takes endurance care and drive in short it takes a programmer but not just any kind of programmer it takes a craftsman to do something but boy we've just scratched the surface of this topic there's a whole bunch of stuff we have to talk about this fascinating discipline of test-driven development and then after that we've got design patterns we've got acceptance testing professionalism functional programming you're not gonna want to miss the next exciting episode of clean code episode 20 clean tests Good dogs, good dogs, let's go. Oh, the Zappos man came. Good dogs, good dogs. And we, this makes the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the, the you