Hi, I'm Alexis. Hi, I'm Lola. Welcome to another episode of Clean Code. Clean Code. Clean Code. Clean Code. You're back to the beginning of... To begin a journey of... In this episode... It's exciting. By Uncle Bob. Uncle Bob. Uncle Bob. Enjoy. Bye bye. Welcome, welcome to episode 19 of Clean Code. Advanced test driven development, part one. Let me take your hat. Good grief. So, do you remember back in episode 18, we walked through the payroll case study applying the principles of component design. And we began with the principles of component cohesion in order to create a tentative component structure. But we didn't care for that component structure much It had cycles, it had stability issues, it had abstraction problems. And so we fiddled around quite a bit, applying the principles of component cohesion, and eventually we straightened out the mess. But then we found that we had a really absurd number of components. The component count was very high. the components by family and that gave us a strategy for reducing the component count simply by replacing the components with their families. But we decided not to do that, at least not right away, because we liked the tinker toy nature of composing our system out of those little tiny components. So for early development we decided to leave the component count high and then we'd I'm going to keep that promise I've been making to you for the last year. We're going to talk about advanced test-driven development. But this is just part one. You see as I started planning out this episode I realized that I had a lot more to say about this than I could fit into one or two or even three episodes. So we might do four or five, who knows, quite a few episodes on this into the three rules of test-driven development with a stopover at the single assert rule. And then we're going to investigate the incremental nature of driving algorithmic design with tests. And we will uncover this guideline that underpins everything. As the tests get more specific, the code gets more generic. And then finally, we're going to very interesting problem of getting stuck. So hold on to your hats because we're gonna dive into the fascinating world of advanced test-driven development. it produces lots of heat and light which we make good use of but that's not the only purpose for all that energy the energy of that fusion also keeps the Sun from collapsing the pressure created by all that energy in the core of the Sun pushes back against the force of gravity that's furiously burning fuel just to stay the same size. But the fuel in the Sun is limited. The Sun has maybe enough hydrogen to last for 12 billion years, and we've already used up 5 billion of them. So in about 5 or 6 billion years, when the Sun has depleted the hydrogen in its core, the core will collapse and begin to fuse that helium into carbon. we can do that for about another billion years or so. But that's all the fusion the Sun can muster. Its mass isn't sufficient to force carbon to fuse any further. And so, after throwing off its outer layers in a beautiful, if short-lived shroud, called a planetary nebula, the Sun will end its life as a cooling ball of carbon in crystalline form, a diamond in the sky. But stars that are more massive than the Sun have a very different fate. Consider Betelgeuse, the bright red star on Orion's shoulder. This star is probably 10 or 20 times the mass of our Sun, and although it's only 10 million years old, it's already approaching the end of its life. its fuel to hold off the collapse and maintain its size. So although Betelgeuse

is only 10 or 20 times the mass of our Sun, it has to burn through its fuel 10 or 20,000 times faster. Betelgeuse is a red super giant star. Notice that it's a thousand times bigger than our Sun. into helium and has begun to fuse the helium into carbon. How much helium does Betelgeuse have left? We don't know. The helium burning phase for a star like Betelgeuse would last about a million years, but we don't know when it started. So there might not be a lot of helium left. In fact, it's possible that all the helium could already be gone. the core will of course collapse putting pressure on the resulting carbon and Betelgeuse has enough mass to force that carbon to fuse further into magnesium, aluminum, neon and sodium. Now those reactions emit energy too but not very much. The energy produced might hold off the collapse for another thousand years. Then the core collapses again neon from the previous phase to fuse into oxygen. But that only produces enough energy to last for about three years. And so the star begins its collapse again until it's halted by the fusion of oxygen into calcium, argon, sulfur, and most importantly, silicon. This fusion will last just a hundred days Now the star has only a week left to live. The temperatures and pressures in its core mount until it reaches a density of over 100 million grams per cubic centimeter. At that density, silicon can fuse into iron. That's the last gasp for the star. which must ultimately destroy it. For five days, it will furiously burn a full solar mass of silicon into iron. One last chance before the end. But when the silicon is exhausted, the collapse will resume one last time. You see, to fuse iron uses energy. It doesn't produce any energy. So the iron is not going to fuse, and there's no fuel left. that can stave off the inevitable collapse of the star. During the last few moments of Betelgeuse's life, its internal structure will resemble an onion. It will have an outer layer of hydrogen that's not fusing. Within that there will be a shell of hydrogen fusing into helium, and below that another shell of helium fusing into carbon, silicon into iron and at the very center there will be a nugget of iron growing rapidly. The iron will be held up against collapse by the degeneracy pressure of its electrons but as we found out in a previous episode that pressure has a limit the Chandrasekhar limit and when the Chandrasekhar limit is passed the iron core will collapse its outer edge will be moving inwards its internal temperatures will skyrocket until inverse beta decay is possible. The electrons and protons within its body will fuse into neutrons, and the degeneracy pressure of those neutrons will be sufficient to call an abrupt halt to the collapse. The interrushing material moving at 20% the speed of light will bounce off that hard core of neutrons and begin to move back outwards at 20% the speed of light, neutrinos pouring off that neutron core because of the inverse beta decay. The resulting shockwave tears through the star. The overpressures are such that any element will fuse into any other. Gold, uranium, mercury, platinum, all of the elements we know of are made in this process. And then the star blows its guts up all over the sky, scattering those elements far and wide, you and me to exist. You don't want to be too close to one of these events, and 50 light years is probably too close. At that range, the incident gamma radiation would probably deplete the ozone layer and expose us to the ravages of the sun's ultraviolet radiation. It wouldn't be good. But don't worry, Betelgeuse is 700 light years away. close enough to put on one heck of a

really good show. What kind of show? Well, for several months, you'd be able to see it during the daytime, and it would be bright enough to maybe read by during the night, much brighter than the normal full moon. We'd have a year where the sky would never be darker than dusk, but then it would fade, and we'd look at Orion and see that it was missing its shoulder. Gold in my wedding ring, the iron in your blood, the carbon in our wires, the silicon in our chips, the uranium in our power plants, the carbon in our proteins, all of these elements were made when a giant star blew itself to smithereens and scattered its guts all over the sky. Nothing else that we know of can cram this many protons together into a nucleus. out of the ashes of dead stars. When is Betelgeuse gonna blow? Well, sometime in the next million years. Anytime in the next million years. Um, could be today. In episode 6, we learned about the three laws of test-driven development. Do you remember what they are? Oh, I know, I know. I was just talking to Wendy about this yesterday. The first law is you're not allowed to write any production code until you've got a unit test failing first, right? Yeah, that's right. I mean, this is where we get the phrase test first from. Before you write any production code, you must first write a test for that code, test will fail. Because the production code's not written yet, right? Yes, Danny, that's right. Very good. The test fails because the code that could make it pass hasn't been written yet. So, now who remembers the second law? I mean, that's really stupid. Write, or more formally, don't write more of a test than is sufficient to fail, even if that failure is just the failure to compile. What that means is, is that as soon as a test fails, for any reason at all, you have to stop writing that test and start writing production code that will make it pass. I know the last rule. It's like, um, uh, you stop writing the production code as soon as the test passes. Right. Once a test fails or fails to compile, you write just enough production code to make that test pass, and then you go back to writing more of the test. Following these three laws will lock you into a cycle that is perhaps 30 seconds long. You will first write a test until that test fails. Then you will write production code until that test passes. And you will repeat this cycle until you are done. That's right, Spock. But it's not quite that simple. Remember that we also have to follow the red-green refactor cycle. That is correct. First we make the test fail. make the test pass. That is the green part. And finally, we refactor in order to keep things clean. Right, we follow the three laws of TDD, but we also refactor frequently. The idea is that it's hard to focus on more than one thing at a time. So the first thing we focus on is defining the the test pass and then we focus on cleaning up the mess we just made that's the refactor step and remember refactoring never appears on a schedule mister right refactoring is the kind of thing you do all the time like washing your hands after going to the bathroom you never ignore it you never let it slide you also don't put it on a schedule or a plan you just do it all the as a matter of course. But tell me, Uncle Bob, do you really practice all these rules as dogmatically as, say, a doctor scrubbing before surgery? Pretty much, yeah. Oh, you might see me break a rule from time to time as a matter of pragmatics, and we'll talk about that in an upcoming episode. But in general, I take these disciplines very seriously. This is my profession. This is what I do. So, let's drive this home with

a simple example. Let's write a program that takes a name and rewrites that name last name first. We'll call this program the name inverter. I always start this way. name inverter test and then inside the name inverter test I create a test named nothing and I make sure that that test is wired up nicely and I make sure that the test passes let's see if it does come come pass the test yes there we are that I have an execution environment and that the execution environment runs because I always like to begin from executing. Now, by the first law, we're not allowed to write any production code until we first write a failing unit test. That unit test ought to be pretty simple, something really degenerate, like an invalid input. So let's see what our name inverter does when we pass it a null. So, we should just write a test that returns an empty string when passed a null. Given null returns empty string. Okay. And that test should be simple. It's an assert equals of an empty string calling invert name with a null. Now, of course, there's no invert name function yet, so our test doesn't even compile, and by the second law, that means I've got to stop writing the test and start writing production code. We can make this compile by creating a new method named invert name right here in the test. Of course, it returns a string, and it takes a string argument named name, and we'll have so and then oh assert equals needs to be assert dot doesn't it so let's do that and now let's run the test don't worry we're not going to leave that function in the test we'll move it out a little bit later right now however the test still fails so we're still stuck null into an empty string. This makes the test pass and satisfies the second law. So now the third law kicks in and forces us to stop writing production code and go back to the unit test. That's the cycle of the three laws force us into. It's clean, it's quick, it's easy, and it's a lot of fun. But we've got some more cases to write so we better get back to it the next test simply ensures that given an empty string it returns an empty string and of course this test is going to pass right away lovely since this test passes we're still stuck on the first law we don't have a failing unit test so So we're going to have to write another one that fails. But now there's duplication in the tests. Those assert equals statements are just too similar. And they're wordy as well. Just when do you plan on refactoring them? Well, there's no time like the present, so let's refactor now. In fact, that's part of the cycle. Remember, red, green, refactor. fail, we make it pass, then we clean everything up. Okay, so first, let's extract the inverted name into a variable. And then let's extract the original name into a variable. And this makes this assert equals statement parametric. Now we can extract the assert passing in the inverted name and the original name. Let's swap those. So it's original and then inverted. Good. And oh, look, my IDE has found that I can replace another one. Yes, go ahead and do that with all of them. And look, isn't that nice? It's now a calling assert inverted here and assert inverted here. And we can now get rid of these variables because they're anymore and then that's very nice now the tests are much cleaner than before in fact I think I'll take that a certain verted and put it up there and of course the tests still pass now if you're working in Java or C sharp then you've got an IDE that does all these fancy things if you're working in other You keep right on doing these refactorings. Okay, so here, let's close this window. It's kind of in the way. Now,

our next degenerate test is given a simple name. We should return a simple name. What is a simple name? Well, assert inverted Bob. No, no, how about this? that's better, should return name. There, good. A one word name returns a one word name. Of course this fails. There, good. It fails. To make this pass, we're going to want to put some horrible if statement in here. If name is not equal to null and name.length greater than zero then return the name and that should pass now it passes nicely it's a little bit ugly though so let's see if we can invert this if condition to make the the if statement look a little bit more like a guard Whoa man, I mean, like, this is so boring. I mean, you keep on writing all these tests and I completely miss the point. We're not inverting any names here, man. Like, when are you going to invert some names? Well, I'm sorry, Ruby, if we're boring you here. But this is the cycle. This is what we do. simplest, most degenerate test cases. We only ratchet up the complexity very gradually. We never try to go for the gold in the first test case. We go gently and softly. But since you mention it, the next test we're going to write is going to invert a name, although it's going to be the simplest possible inversion we can do. Okay, so now let's assume that we've got a regular two uh, given first last, and it should return last first with a comma in there. So we will assert that if we pass in first space last, we will get back last comma space first. Now that's a good test. Uh, I assume it's going to fail, should fail, right? Because we're not this pass, we're going to want to modify this return statement, which means we probably want an else here. And we're going to want to split that name up by space. So let's do this string two, then we can return the swapped names. String.format. Good. Percent s, comma, space, percent s. We'll just swap the two and put the comma in that way. Names sub one and names sub That's very specific to this test. It's not general. I'd like to change this. So what I think I'm going to do is invert the if condition here and then say if the names.length is 1, then we return the name. Otherwise, we're going to swap it. This makes 1 the special case, everything other than one, the generic case. I wonder if that passes. And of course it does. Yeah, yeah, but Uncle Bob, aren't you wasting time with all that cleanup you're doing? I mean, you're always changing code that already works. Isn't that like a rework? Isn't that bad? Bad? No, this isn't bad. This is a good thing. It's hard to do more than one thing at a time. It's hard to focus on making the code work and keeping it clean. So what I like to do is do these things at separate times. I like to first make the code work and then clean it up. And by the way, this doesn't make you go slow because it lets you keep the code clean all the time. And boy, the only way to go fast is to keep the code clean. That split statement reminds me that we have another degenerate test to write. Given a simple name with spaces, we should return the simple name without spaces. Now, we can do that like so. name. And I expect that to fail. And it does. And of course we can make that pass by simply trimming the name. Like so. Excellent. But gosh, Uncle Bob, it looks like you did that out of order. I mean, shouldn't you have cases first? Well, ideally yes, but the next time I write tests in the ideal order will be the first. Look, I often discover that I missed a degenerate test while I'm writing some other test and when that happens to me then I just make sure that the next tests I do are

the degenerate cases. I'm trying as hard as I can not to go too complex too of degenerate tests, what would happen if there was more than one space between the first and last name? Let's write a test to cover that. Let's see, given first last with extra spaces, return last first. Now we can implement that test simply. A bunch of spaces first, and that should return last comma first. This, I expect, will fail. And, of course, it does. Good. Now, how do we make this pass? Well, what we'd like to do is split on more than one space, and, in fact, more than one white space, which we can do by putting a regular expression in here. on more than one white space, I think this will pass. Yeah, good. Excellent. And that lets us do a little refactoring. Now, one of the things I don't like about this solution is changing the existing state of the name variable. This is always bad form when you overwrite a variable like this. So we can break the dependence on name by changing this to names subzero. that's gonna be the same thing that passes yes it does good and so I don't need to change the state of name anymore which means I can get rid of that and just do a trim there yeah that's nice and so that means I can probably put a nice little else in here because that looks funny hanging out there without an just for nice symmetry sake. There. There. That's much better, isn't it? Son, I think you could have put a herd of stampeding longhorn cattle to sleep with that little show you just put on. You think maybe we could do something a little bit more challenging? mister in it or a missus or a miss or a miss you know what they call them things don't you son they call them honorifics writing the test for a simple honorific isn't hard we're just going to ignore the honorific um and that means we're going to do um um mr first last and that should turn into Last comma first. Now, I expect that that will fail. And it does. So, how do we make this pass? Yeah, and so I suggest that in the case where there are two or more words, if the first word is Mr., then we simply ignore it, and then everything should work as before. makes sense. The problem with it is that it's difficult to ignore the first string. So if we had a list of strings instead of an array of strings, we could remove the first element. Okay, so let's take the output of the split function and we'll put it into arrays.asList and this of course is going to have to turn into a list and that would be a list of string oh and of course you can't do that can you and it should be size and yeah that would be names dot get zero right and names dot get one and dot get zero okay and I think all the compiler bugs are gone so it fails okay fine it fails but it fails for the same reason as it failed before so now let's is greater than one and names dot get zero dot equals mister then what we want to do is remove names dot remove zero we'll remove the first element and that list remove oh yeah right right right as list returns a list that is backed by the original array and you cannot remove from an array so we we can't use as list here let's let's do this new array list of string there yeah I'm gonna have to Oh, and I still need the arrays.as list. Jesus, this is awful, isn't it? Okay, and that seems to compile, and oh, it passes. Well, okay, but we've got a really hideous mess here, so let's take this hideous mess right there and split names. There. That's at least not so horrible. And here's another little horror scene here. Let's pull that out into a method named isHonorific. Yeah, I think that works. Let's see if it passes. Yes, it passes. Wonderful. like the fact that names gets passed into is honorific.

It really ought to just be a name. So let's change is honorific to string word. Yeah, all right. And then it should be here, that should be word. And then up here, when we say is honorific, we should say dot get zero. And I think that will still pass. Yes. Okay. So, we survived that one. Good. Good work, mister. But you're going to need more than just mister. Yeah, that's probably true. So, let's do this. Let's duplicate that test. We'll add a Mrs. case. And we'll change the name of the test to Ignore Honorifics. believe that's gonna fail and it does. Fascinating! I note you are in violation of the single assert rule. Was this intentional? Ah yes! The single assert rule! The rule that says that every unit test should have one and only one assert. This is sometimes called the triple A rule for unit test should be broken up into those three parts. A part that arranges, another part that acts, and a third part that does the assertion. The arrange operation creates the data and context for the test. This is usually done in the setup function or at least partially done in the setup function. In our case there's no setup or arranging to do so we can ignore that. The act operation is when tested. In our case that's going to be the invert name function. The assert operation verifies that the function being tested did what it was supposed to do. This is typically done with some kind of assertion. That assertion is a logical are represented by several calls to assert. For example, this is a single logical assertion that n is even and greater than 10. The goal of the single assertion rule or the triple A rule is not to keep you from writing more than one physical assert. It's to make sure that when We want arrange, act, assert, not arrange, act, assert, act, assert, act, assert. Well, yes, obviously, but, well, then why? Because we want every action tested independently. We don't want the input of one test to be the output of the previous test. Yeah, cool, but why? Well, you see, this is going to get into the whole topic of test independence and isolation. Excuse me, mister, but I'm still waiting for my missus. Yeah, right, right, right. Let's talk about this in another episode when we talk about test independence and isolation. Right now, we've got to get done with this. To make this pass, we want to modify the isOnOrific method, probably to use regular expressions. We'd like to have Mr. or Mrs. in here, something like this. Of course, those dots are not quite right, are they? So we have to kind of do that, don't we? And I think that will pass. Oh no, it doesn't pass. Heavens, heavens. Oh, that's right. Matches. There we go. Okay there, smarty pants. But what you gonna do about all those fancy-dancy guys like doctors and lawyers and scientists and dentists who like to put things after their names? You know, letters like Ph.D. or M.S. or B.A. or all them post-nominals. huh? All right, let's, um, let's write a test. Um, let's see. Postnominals, uh, stay at end. Okay, and, uh, let's write one for the senior case. So, um, first, last, uh, senior, uh, should return, good and I expect that that'll fail of course it does now since we're already pulling off the honorifics then at this point here if we have more than two if names dot size is greater than two then there must be post nominals oh let's create a string for that string post that nominal equals empty string good okay and then inside the if we will say that the post nominal equals names And now we just have to change the format statement to add the post nominal. Just like that. And I think that'll pass. Oh, heavens, it fails. But not the post nominal test. Look at

that. The post nominal test is passing. It's all the other first last name tests that are passing. And it's failing because, you know, You get these errors, right? And expected last first and actual last first. Oh, do you see that space there? See that space there? These look identical, but they're not. There's a space at the end. And that space comes from right there. Because if there's no post nominal, I stick that space in. So I think I can fix that with trim. With a star in it. Yeah, look at that. Okay, okay, but what about them city slicker guys who like to have more than one of those doo whoppers at the end of their name? You know like they've got a Bachelor of Science and a PhD? You know what that means, don't you? That means their BS is piled higher and deeper. That's a good one! Yeah, okay, look, this is an easy test to add. here, assert inverted that first last BS PhD, just if you want to be happy, probably I'd be like that, huh, and that should turn into last first BS PhD, right, just like so, and I expect that'll Uncle Bob. I know, Spock, I know. Single assert rule, right? Look, we're going to talk about that in another episode. Right now, let's just get this test to pass. So, to get this to pass, we're going to have to do some funky work here to get that post-nominal. Let's do this. Let's pull that out as a method called getPostNominals. Good. Okay. And now I've got this function. What I'm going to have to do here is not just get the first postNominal. I'm going to have to get all of them, So let's do this. List of string, and we'll call this postnominals. And we will say that postnominals is equal to names.substring, sublist, of 2 to the end, which is names.size. and that creates the sub list of post nominals now what I have to do is join them all together into a string so a string post nominal equals that okay and then for string post nominal in post nominal PN plus space. And we should be able to return post nominal. And, gee, I wonder if that'll work. That was a lot of code. Oh, man, am I a programmer or what? Whoa. Gosh, Uncle Bob, do you think it all works now? I mean, what about a really complicated name with spaces in it and post nominals and honorifics and all that stuff? we test it all yeah kind of integration test to see if everything we've done works together that sounds like a great idea let's do that okay so let's write that integration test test integration cool a certain verted room bunch of and more spaces Martin spaces the third spaces Esquire yeah more spaces and that should turn into Martin comma Robert the third Esquire just like so there's our passes. Oh, refactor, Uncle Bob, refactor. Yes, Danny, I think we will because refactoring is the reward for making tests pass. So let's refactor this code because right now it's a mess. of removing the honorifics. That happens right there. Let's just pull that onto a method named remove. Honorifics. How about that? Yeah. And that's still going to pass, I bet you. Sure it does. And, oh, heck, I think we could do the same here with this string format business down here. That's a mess. Let's just take that. there and we will say format name. Oh, that's good. I don't like the fact the post nominals in there. Let's undo that. And let's try this a different way. Let's say trick that into a variable named the name. Now formatted name. and that should still pass, of course. It does. And then we ought to be able to take this whole wad of code here and pull that out into format name. And, of course, we don't need the variable anymore. That was just there to hold the place. Let's see. That should still pass. It does.

Wonderful. and let's see format multi element name and now we can pull this whole thing out into format name look at that i'm kind of liking this um now wouldn't it be cool if remove honorifics here and have it return a list of string and just return names. Now, I don't think that changed anything and break anything. No, it didn't. So now I'm going to enjoy this. Let's do this. We'll take that and we'll put it right in there. How about that? Yeah! Now we're gonna take this and we're gonna put that in there. Oh boy, am I enjoying this. Yeah! Okay, that's cool. So now I think I can get rid of these horrible braces. Mission to destroy all braces. Passes. Excellent. Excellent. Good. a name right this is cool I mean that's like really simple and formatting a name when you do that if the name size is one you just return the first element otherwise you format the multi element name how do you format a multi element name well you got to get all the post nominals boy I don't like this business here you know what let's do this let's take that get post nominals and inline Yep, inline it right there. And now let's take this whole horrible thing and extract that as a method named getPostNominals. Oh, that's interesting. Reformat this. Okay. That's nice. So that's the postNominal there. Good. Let's continue this. First name equals names dot get zero. String last name equals names dot get one. This should be last name. This should be first name. I think that's nice. A bunch of explanatory variables. Oh, yes, very pretty. Okay. How do you get the post-nominals? Well, you start with a string that's a blank. If there's more than two in there, then you're going to really get them. So you get the post-nominals out of this string. We don't need that separation of an assignment and initialization. Ooh, this is awful. just post nominal. Let's close the post nominal string. Yeah, that's fine. Post nominal string. And then we take the this, which is a post nominal. And we can just return the post nominal string, can't we? I mean, there's no reason to do any of that. And yes, let's take this up out of there, and we really don't need that, do we, okay, and, oh, cross your fingers, nothing broken, that's good, okay, um, this is much better, let's see, how do you remove the honorifics, oh, that's about right, isn't it, uh, how do you check an honorific, like so, how do you split the names, like so, kind of ugly, oh, look at those blank lines, ah, blank lines, terrible, it's all in the test. Let's get it out of the test, huh? Let's take this and get it out of the test. So we're going to go to invert name here. That's our top level function. And we're going to pull down a refactoring named extract class. Right. And the name for the new class will Well, obviously, we're going to put invert name in there and format name and format multi-element name and get post-nominal. And, gee, I think we're going to do all those, aren't we? Yeah. Anything else? I don't think so. Okay. Refactor that. Pull out the name inverter. And the name inverter has been pulled out. Let's see. Where's our tests? Our tests. assert inverted assert inverted calls oh look name inverted on invert name look what happened up here um you know that probably ought to be in a setup method let's let's do that let's create a an ant before called public void setup and I know, I know, I know. Okay, and now we probably don't need that. And, oh yes, not final. Just private. Okay, good. I gotta believe that's gonna pass. Oh, heavens cannot find symbol arrays. Well, silly people. Do your doggone imports. There we go. we have the

name inverter. We don't need a default constructor. Who put that in there? It's still going to pass. And the name inverter has the invert name function and the format name function and the format multi-element name and the get post nominals function and the remove honorifics function. This is wonderful. Look at that. All that nice, lovely code just sitting in there. Hmm. Well, I think we've gotten our reward. We've done the refactoring we needed to do, and we've gotten our reward for making the tests pass. Look at those tests. Are they pretty or what? Simple little test. Oh, oh, oh, look at that. It did not take those functions out of there. They're not called anymore. It just kind of copied them out. Okay. we'll take them out good everything should still work it does oh my yes yes yes pretty little tests pretty code very nice and that is how you follow the three laws of test-driven development but we've just scratched the surface So let's talk about specifics, generics, and incremental algorithmics. Wow, look at the time. Man, it's been an hour already, and I'm only half done with this episode. But we're going to have to cut it here. Cut, cut. I'll continue this in episode 19, part 2. That'll be the next episode coming up. when I refactored the name inverter. I'll bet some of you were snickering back there going, you forgot this or that. Well, I'll tell you what, man, I've got 20 green bands. And I'm going to give them away to the first 20 people who post one of those mistakes at our discussion group, cleancodediscussionatgooglegroups.com. You see the URL down there right now, right? and loads of stuff to talk about. We've got incremental algorithmics, that'll be the next episode, part two of this one actually. And then we've got clean tests, and testing anti-patterns, and clean design of tests, and testing GUIs and databases, and all kinds of stuff to talk about. So you're not gonna wanna miss the next exciting episode of Clean Code, episode 19, part two, Algoritmico-mental algorithmics. you