So, look at these 3 tests that we wrote in the last lecture. Here's one example.

This test is not too bad, it's fairly short, but I'm not 100 percent happy with this implementation. We can take this to the next level and make it cleaner and more maintainable. Look, this arrange part is 3 lines of code and we have kind of repeated this in a few different places.

The only difference is the arguments that we pass to the returns method. Now in contrast, look at how we implemented this scenario. Where housekeeper's email is empty string or white space. or null. Our arrange part is only one line of code. So the first thing I want to improve here is change the implementation of these test methods, and instead of programming the mock here, I want to program the mock in our SetUp method, and then here we can simply set statementFileName to let's say null for this test case.

But currently, this doesn't work because we defined this field as readonly. So, first let's move these few lines inside our SetUp method.

Cut. Okay, here's the SetUp method. So this is where we're initializing the statement generator object. So, right after I want to paste this code and then add a line break. So these two lines, highly related, they're purely for programming our statement generator Mock

Now, instead of returning null here, I'm going to return statementFileName. Okay? And here in the SetUp method, I want to set the initial values for the happy paths.

And then in test cases where we're testing something negative, we will modify that happy path. So here, we set statementFileName to an actual

value like fileName, and then we can overwrite this in those three negative tests. You can set it to null, empty string, or white space, Now let's get rid of this compilation issue. So, we need to remove

the read only modifier here. And with this new implementation, we should initialize this field inside the SetUp method, because we want to make sure that before each test we have a clean state. So, let's delete this initialization here, okay now, back to our negative unit tests. So, towards the end here, okay, here's one, send statementEmails where StatementFileName is empty string.

So we can simplify these three lines and instead add statementFileName, set it to an empty string. Here is another one. We set statementFileName to white space, look, test is already cleaner.

And here's the other negative test case. Let's run these tests, so I'm going to put the cursor on the class name, and run all the tests, okay.

These three tests, they all broke. Why? Let me explain. So,

look at the SetUp method.

Here we set statementFileName, to this string, statement fileName, and then programmed this Mock to return this value. So even though we overwrite the value of this field in our test methods, this mock is programmed ahead of time to return a fileName. So we are not testing the scenario where our statement generator cannot generate a statement file. How do we solve this issue? It's very easy. In the last lecture I told you that this Returns method has two overloads. One that takes an extra value, like a string.

And the other that takes a funk, a delegate, a lambda expression. We use that for lazy evaluation. So, if instead of passing this private field here, we pass. a lambda expression, the lambda expression, a function with no arguments, goes to this private field. Here we have lazy evaluation. So, with this technique, we'll be able to change the value of these fields in our test methods, and this expression will be evaluated later. Now, let's run all the tests, so I put the cursor back on the class name, and run the tests, you can see they're all passing, beautiful. Now, let's have one final look at the tests we wrote in the last lecture. So these three negative tests we can see their Assertion is exactly the same.

You're verifying that this method is not called and we're using it IsAny string and Times.Never. And this is repeated in the last three tests.

So when writing unit tests, it's very helpful to extract some private helper methods, to make your tests more readable. For example, here, we can select this code, and extract it into a private method, so alt and enter, refactor this, extract method,

We can call this VerifyEmail NotSent. It's very descriptive. Next, look at this test method, only three lines of code. It can't be simpler than this, we set the statement file name to null, act and verify that EmailIsNotSent. Now, I'm going to move this to the bottom of this class, because we put. all the helper methods down in the bottom. And modify these two occurrences, to call VerifyEmailNotSent. And one more here.

Okay. Much, much better. Now, potentially in the other test case, where we want to assert that email is sent, again, again we can extract this into helper method called VerifyEmailSent. And this way we'll have symmetry in our code. So, let's do this too. Alt and enter. Refactor. Extract method. VerifyEmailSent. Much better. And also we don't need

these few lines anymore, because we programmed our statement generator mock to return a valid file name inside the SetUp method, so let's delete these two lines, and move this helper method down the bottom. Like this. So this is what you call proper unit test.

Very, very short, very clean. In one of the companies I worked at a few years ago, we had unit tests that were 50 lines of code, that's not a unit test. That kind of unit test slows you down, it breaks easily, and when it breaks it takes you more time to figure out what is going on.