So, we have written one unit test here, now, let's quickly go back to the implementation of our service. As I told you before, when writing unit tests, we should always look at the method of a test like a black box. So you shouldn't really write line by line of this code, and write unit tests based on this implementation, because its possible that this implementation has a problem, however, in this case because I'm not familiar with the domain of this application I have to rely on this implementation. So let's see what's happening here. When we are iterating over these housekeepers it's possible that some of them don't have an email. In this case, you're going to move onto the next housekeeper, and that means you shouldn't create a statement for them. That means there is a business requirement for this, because if the housekeeper doesn't have an email, we don't want to waste resources generating a statement for someone that is not reachable. So, let's go ahead and write the test for this scenario. Back in our test class, to save time, I'm going to select this method. Duplicate it. So, the scenario is where housekeeper's email is null. So,

HouseKeepersEmailIsNull ShouldNotGenerate Statement. So how do we simulate this scenario? Well, we define a housekeeper object right here. So, before acting we can set the email to null. So, right here. So, this is the arrange part, houseKeeper.Email null. Now, how do we verify that a given method like SaveStatement is not called.

This Verify method that is available with all the mock objects, it has an optional second argument. With this argument we can specify how many times a given method in this case, SaveStatement should be called.

So, as the second argument of the verify method, we pass Times, which is an enumeration, here we have members, like AtLeast AtMost, Between, Exactly, and so on. We're going to use Never. So that means this method should never be called. Let's run this test.

Okay, the test passed, I want to make sure that I'm testing the right thing. So, let's go back to our service, and comment out the line that would make that test pass.

So, here's the line, comment out, back to our test, let's run this one more time, it should fail now. Okay, it failed, beautiful. So, let's remove the comment, now when dealing with strings, we should check for both null as well as white space. So what is missing in this implementation is the check for an email that is an empty string, or an empty string with white space. But I'm not going to modify this code yet, I'm going to write the test, that test should fail. Then, we'll come back, fix the code, and then our test should pass. So, back to our test class. Let's duplicate this last test.

Now the scenario is HouseKeeperEmailIsWhiteSpace. WhiteSpace. So I'm going to set this to white space like this. Again, our Assertion is the same, the SaveStatement method should not be called.

Let's run the test.

Obviously you can see this test failed, so let's go back to the production code, and modify this If statement to something like this. So, if String IsNullOrWhiteSpace, then we pass housekeeper.Email here. Back to our test, let's write one more time. This time it passed. So we have a tiny bug in the code, and it could catch that with this unit test. Now, what if email is an empty string not White Space? Well with this new implementation we shouldn't have a problem.

But let's verify that. So, I'm going to select this code, duplicate it, and this time set email to an empty string. And we should also change the scenario to HouseKeeperEmailIsEmpty. Let's run the test, here's our new test, beautiful. So, if email is null, empty string, or whitespace we are not going to generate a statement, beautiful. Now, back in our service, once we generate a statement, then we should email it to the housekeeper, and that's what we're going to look at in the next lecture.