Alright, now we're ready to write interaction tests for this method. But before we get started, I just want to highlight something. Earlier in the course, I talked about application services. I told you that application services are responsible for orchestration. They do high level jobs. And here this class is a perfect example of an application service.

Because this method is orchestrating a few different things. It's getting housekeeper records from the database, and for each housekeeper is generating a statement, and also emailing that statement to the Housekeeper. So this is all about orchestration.

So I would rather to rename this class from HouseKeeperHelper, to HouseKeeperService. So it's clear this is an application service. So let's rename this HouseKeeperService. So, as part of this unit testing journey, we started with a helper class with a bunch of static methods. We moved away from that design, now we have a proper application service, HouseKeeperService. And we also have better separation of concerns.

So earlier, all this code for sending emails or generating statements was implemented as private static methods in this class. This violates the single responsibility principle. Because sending emails is an entirely different concern. It shouldn't be implemented as a private static method in this class. The same is true for generating statements. So with this design, we put the code where it belongs.

We have proper separation of concerns. And also our class is loosely coupled and testable. Because we are programming against interfaces.

So now, let's create our first interaction test. Back in our unit testing project, inside the Mocking folder, let's add a new class, HouseKeeper ServiceTests. We add the TestFixture attribute, and create our first unit test. public void, the method under test is send. StatementEmails. So, what scenario do we want to test? We want to make sure that when we call this method it's going to generate statements for our housekeepers.

So, WhenCalledShouldGenerate Statements. Now, let's quickly look at the implementation of this method. Here, we need to get the housekeepers from the database. So, we need to mock this unitOfWork, and return a list of housekeepers in memory.

So, var unitOfWork we'll set this to a new Mock, of IUnitOfWork and here we call UnitOfWork.SetUp when we call the unit of work. Query of Housekeeper we should return a new list ofHousekeeper Objects, let's import the list class. And we initialize it here, now we have a compilation issue, because this query method defined a unit of work returns an IQueryable object. So, after the initialization of the list, we need to call as queryable. And the compilation error is gone.

Now here we create a new Housekeeper object. Let's set it's properties, so we have

four different properties here, and three of them are strings. So, let's set Email to a, FullName, to b, Oid is an identifier we set that to 1, and finally StatementEmailBody to c. Simple values. So this is our UnitOfWork. Now we need to create our InstanceOfOurService So, service is a new HouseKeeperService, the first dependency is our unitOfWork, so we pass unitofWork with object.

But we have three more dependencies, so let's quickly create a few more mock objects. The second one is statementGenerator we set this to a new Mock ofIStatementGenerator the third one is emailSender, so email Sender set it to a new Mock, of I EmailSender. And the last one is our messageBox. So message box is also a new Mock. of IMessageBox. XtraMessageBox, more accurate. So now with these dependencies, we can create our service.

So, let's pass statementGenerator.object emailSender.object and finally, messageBox.Object. We can see our code is going beyond the screen, so let's put each argument on a separate line and make this code more readable. So here we have a service. Now, we need to act. So, service, .SendStatementEmails. We need to pass a DateTime, let's just send an arbitrary date. And finally, we need to verify that the right method in statementGenerator was called. So, statement Generator.Verify sg goes to sg.SaveStatement.

Now what argument should we pass here? The first argument is Id of our HouseKeeper, I'm going to pass 1, the second argument is HouseKeeper's name, we use b here, so we should pass b here as well.

And the third argument is the statement date. And that's the same date we use right here.

So, let's pass that here as well, and we are missing 1 closing parenthesis, got it. Okay, so here's our first unit test, it's very fat, it's bulky and noisy, we're going to refactor this later, but first let's make sure that it passes. So, okay, we have compilation error here, in EmailSender class, looks like we're missing the reference to the file class. So, let's import the name space. Done. Back in our unit test class, let's run this test,

Okay, here's our test, and it successfully passed. Beautiful. In the next lecture, we're going to refactor this code, and make it cleaner.