Now we are ready to unit test this method. I've got a question for you. How many tests do we need for this method. I'm going to collapse this implementation, so you think of this method as a black box. So how many tests do we need here?

Well, in this service we are getting the list of videos from our repository. What are the possibilities here? one possibility is that our repository returns an empty list. In that case, what should we get from this service, we should get an empty string.

Another possibility is that our repository is going to return a list with one or more video objects. So, imagine we have an array, with multiple video objects, 1, 2, 3, and each video object has an ID. So then, this method is going to get the ID of these video objects, and join them using a comma.

So here we should get a string like this, 1, 2, 3, where these are video ID's, okay? So let's go ahead and write these tests. So, back

in our unit testing project. In the Mocking folder we have VideoServiceTests, beautiful. So, let's create a new test method public void. GetUnprocessedVideos, as Csv.

What is the first scenario? No UnprocessedVideos. So NoUnprocessedVideos. Now, this scenario is a little bit confusing because we have double negative. It's better to change this to

All videos are processed. AllVideos AreProcessed. Then, this method should return an empty string so in the arrange part, we need to create a mock object

So, let's go back to our set up method, here we need to create a new mock object, let's call that repository and set this to new Mock of IVideoRepository. Now we don't have this field, so let's create this here.

Okay. So here's our new field, Mock up IvideoRepository. Now we're going to pass that as a second argument with a constructor of video service.

So\_repository.object.

Now, back in our test method, in the arrange part, we need to program our mock. So \_Repository.Setup, here we pass a lambda expression, repository goes to repository.Get UnprocessedVideosReturns, you're going to return an empty list. So new List of Video like this. And let's import video on the top. Next, we're going to call video service.GetUnprocessedVideos. Get the result, and Assert That result Is Equal to To an empty screen (?). Let's run this test.

Beautiful, it's passing. Now, the second test.

So, to save time, I'm going to duplicate this test. Now the second scenario is we have a few unprocessed videos. So, AFewUnprocessedVideos. So this method should return a string with the ID of unprocessed videos. So, return a string, with ID of UnprocessedVideos. Okay? So here in the arrange part, I'm going to initialize this list, with a few video objects.

So a new Video, let's create a video with ID 1 we don't need to set other properties here because they just create extra noise in the code. We don't need these parenthesis either. Now let's duplicate this and change the ID to 2 and 3. Finally, we're going to Assert that result is a string like this. 1,2,3, let's run the test.

Still passing beautiful, I want to make sure that I didn't make any mistakes here, so I'm going to go back to our video service. and comment out the line that would make these two tests pass. So, this is the line, if I change this to, let's say return 1, then both our tests

should fail.

So, here I'm going to put this cursor on the class name, and run all the tests in this class. Okay, you can see two of them failed, both of them

are ForgetUnprocessedVideos method, beautiful.

So, this verifies that we have implemented our tests properly, so let's go back to our video service and bring back the original code. And we don't need this column anymore. Now one final thing before we finish this lecture. Let's go back to our test class. Look at the initialization of this video service.

We're passing these two objects as dependencies, and this is not terribly bad, but in a larger, more enterprise like application, you may end up with a constructor with quite a few parameters. And your setup code is going to be complex. If you have this situation in your code, that can mean that perhaps that class is doing too many things, perhaps it has too many dependencies, maybe some of the methods in that class belong to a different class.

So when you move those methods to a different class, some of the dependencies will automatically go away. So that's one thing you may want to look at.

Another reason for having a constructor with two many parameters is

when you're mocking everything, again, you should reserve mocks mostly dealing with external dependencies. The only exception is if you have complex methods with complex execution paths, then you want to test them in isolation without their dependencies.

Because otherwise your test methods may get really complex and bulky. And the last thing you may want to look at is where those dependencies are used. In this case we have used file reader only in this method, ReadVideoTitle, and this repository only in this other method. GetUnprocessedVideos. Right? If a dependency is used only in a single method from a design perspective, it's better to pass the dependency as a parameter to that method. For example, in our video service this FileReader is used only here.

No where else in this class. So potentially you can pass this as a parameter here. So IFileReader. This way you are going to end up with a bulky constructor with too many parameters. But as I explained before, this feature may not be supported with the dependency injection framework you're using, so you're only option might be constructor injection. Okay, let's revert this. We are done with this exercise, next, we're going to look at another exercise.