One of the questions that I get a lot is Mosh, how should I test my private or protected methods? The answer is, don't test them, you shouldn't.

Because these methods are implementation detailed. What do I mean by that? Think of a DVD player. A DVD player has a few buttons on the outside, but a complex logic board on the inside. What you see on the outside is what we call a public API, or the public interface of a DVD player.

What is inside this black box that we don't see, is what we call implementation details. An implementation can change from one version to another. But the buttons on the outside they stay the same. In object oriented programming, we want to design our classes like a DVD player, so we want to have very few public members, and these members represent the public API, or the public interface of our classes. Private and protected members represent the implementation detail. But they can change easily as you refactor and restructure your code. There are tools out there that allow you to write tests against private members. Or you can make a private member public and test it, but you shouldn't do this, because when you write tests that use private members, your tests are coupled with an implementation detail.

And as you change this implementation, these tests are going to break. They get in the way and they slow you down. Let me show you this concept in action. So, back in our error logger class, buried the log method, erase this ErrorLogged event.

Now for this demo, I want to extract this line and put it in a

protected virtual void method. So, you probably see this convention before. Protected virtual void OnErrorLine.

So this method can take a parameter, like guid, which would be the error ID, and this is responsible for raising the event. So, I'm going to cut this line, and put it here.

And replace this new guid with the error ID. Now here, I'm going to call OnErrorLogged, and guide it this new guid. So this is a convention that you have probably see the applications out there.

This method we have here, this is implementation detail.

It can change from one version to another. So we shouldn't write tests against this method, instead we should write tests of our log method. So let me show you what happens when you write tests for this method. Before going any further let's go back to our error logger test, here we have a few tests, and all of them are testing the log method. Let's run this test and make sure all of them are passing. So we can see all these tests are passing, beautiful, now, let me show you what you shouldn't do. I'm going to write a test for this new protected method, so test, public void, OnErrorLogged, WhenCalled, it should Raise Event. Now I'm going to go back to our error logger class, and cheat by changing this access modifier from protected to public, this is bad right from the get go, because this is about implementation detail. You shouldn't leak these to the outside. This is like designing a DVD player that exposes the internal chips to the outside. So, back to our test method, I'm going to create a new error logger, ErrorLogger, call OnError Logged, give it a new guid, and make some assertion, and as I told you in the last lecture to test and method that raises an event, first you need to subscribe to that event, and then make an assertion, like here.

But to save time, I don't want to repeat this, so I just want to write a simple assertion that always passes. Assert that true. It doesn't really matter. Now,

I'm going to run this test obviously it passes, okay?

Now, let's see what happens when we refactor our code and change

our implementation. So back to our error logger, let's say tomorrow we decide to change the signature of this method, and instead of passing the error ID here, storing the error ID, in a private field. For example, we may introduce a field here, private, guid, underline error writing.

So in our log method, before calling OnErrorLogged, instead of passing this as an argument, we're going to set errorId with a new guid, and

here we're going to remove this parameter, and simply read a private field.

So we have changed our implementation, we have refactored our code, we have

restructured it, right? Now guess what? Our test broke, let's have a look. We have to change this test and remove this argument. Now this is a simplified application, and we have only a single test for this method, but in a real world application you might have ten or twenty tests for a given method. Now as you refactor the code, as you change the implementation all those tests are going to break, at least this test project is not going to comply. You have to go through each test method and make the necessary changes, just to make sure that the test project complies and then you have to run the test to see if they pass, alright?

So, I'm going to remove this, now back in ErrorLogger, let's say tomorrow I decide to refactor this code and totally drop this method. So I'm going to decide to directly raise the event inside the log method. So cut this from here, and raise the event like this. Now, I'm going to delete this method.

So I changed the implementation. Now guess what? Our test broke again. Look, We're testing a method that no longer exists, so this is the problem with testing private or protected members. They are about implementation detail, and they can change easily, and you (?) write tests for these methods, our tests become very fragile, and they break often. Now let me show you something.

I'm going to delete this test, so in the last lecture. We wrote this test for the log method to make sure it raises the right event. This is the right way to test this class, we're testing only the public API, or the public interface. Now let's take a look at the current implementation.

So currently, you are raising the event like this. If tomorrow I

decide to extract this line, and put it inside the protected virtual method, my test will still pass. Let's do it one more time.

So, protected virtual void OnErrorLogged.

So I'm going to cut this line from here, and put it here, I changed the implementation but I kept the public API, or the public interface the same. Now, if we go into our tests, we're going to make sure to call this method OnErrorLogged. Now let's go back to our tests, now that my test is broken, let's run them all and see. Look, they are all passing. Beautiful.

Now back to our error logger, let's say tomorrow we decide to change this implementation, so instead of using this private field, we may prefer to pass an argument to this method. So we pass guid, .NewGuid. Of course we add a parameter Guid error Id and use that here. Again, we change the implementation but none of our test is broken, because we are testing the public API, the public interface, which should be small and lightweight. Now what if in your applications you have a public that calls a private method and that private method also calls another private method, perhaps you have a chain of private methods, and each private method has five to ten lines of logic, in that case if you want to write all the tests for the public method, your tests are going to get so complicated.

Maybe you have so many execution paths. If that's the case, that could be a symptom of a design spell in your application.

So chances are those fat private methods shouldn't really be private methods in that class. Maybe they belong in a separate class, and there they can be public methods that can be reused in other cases. In that case you move those private methods to a new class, make them public, and properly unit test that class.