Shim and InstanceBehavior fallthrough to isolate part of the SUT

2012, May 10 *3 mins read*

I've dealt in a previous post with the new Shim library in Vs11 that permits you to test "difficult to test code" and **I showed a really simple example on how to use Shim to isolate the call to DateTime.Now** to simulate passing time in a Unit Test. Now I want to change a little bit the perspective of the test, in the test showed in previous post I simply exercise the sut calling Execute() a couple of time, simulating the time that pass between the two calls. Here is the test

```
1     [Fact]
2     public void Verify_do_not_execute_task_if_interval_is_not_elapsed(
3     {
4         using (ShimsContext.Create())
5         {
6             Int32 callCount = 0;
7             PerformHeavyTask sut = new PerformHeavyTask(10, () => call
8             DateTime startDate = new DateTime(2012, 1, 1, 12, 00, 00);
9             ShimDateTime.NowGet = () =>  startDate;
10            sut.Execute();
11            ShimDateTime.NowGet = () => startDate.AddMinutes(9);
12            sut.Execute();
13            Assert.Equal(1, callCount);
14        }
15    }
```

I can also change the point of view and **write a test that uses a shim to isolate the SUT** , this is a less common scenario but it can be also really interesting, because it shows you how you can write simple White box Unit Tests isolating part of the SUT. The term **White Box is used because this kind of Unit Test are created with a full knowledge of the internal structure of the SUT** , in my situation I have a private method called CanExecute() that return true/false based on the interval of time passed from the last execution and since it is private it makes difficult for me to test the SUT.

```
1     private Boolean CanExecute() {
2         return DateTime.Now.Subtract(lastExecutionTime)
3            .TotalMinutes >= intervalInMinutes;
4     }
```

But I can create a Shime on the SUT and isolate calls to the CanExecute(), making it return the value I need for the test, here is an example

```
1     [Fact]
2     public void Verify_can_execute_is_honored()
3     {
4         using (ShimsContext.Create())
5         {
6             Int32 callCount = 0;
7             PerformHeavyTask sut = new PerformHeavyTask(10, () => call
8             ShimPerformHeavyTask shimSut = new ShimPerformHeavyTask(su
9             shimSut.InstanceBehavior = ShimBehaviors.Fallthrough;
10            shimSut.CanExecute = () => false;
11            sut.Execute();
12            Assert.Equal(0, callCount);
13        }
14    }
```

To write this test I've added another fake assembly on the assembly that contains the PerformHeavyTask class to create shim for the SUT. This test basically create a **ShimPerformHeavyTask** (a shim of my SUT) passing an existing instance to the SUT to the constructor of the shim, then I set the InstanceBehavior to * **ShimBehaviors.Fallthrough** *to indicate to the Shim Library to call original SUT method if the method was not isolated. At this point I can simply **isolate the call to the CanExecute() private and non-virtual method** , specifying to the shim to return the value false, then I call the Execute() method and verify that the heavy task is not executed.

This test shows how to create a shim of the SUT to isolate calls to its private methods, thanks to the Fallthrough behavior ; if you forget to change the InstanceBehavior the test will fail with an exception of type *ShimNotImplementedException*, because the default behavior for a Shim is to throw an exception for any method that is not intercepted. Thanks to shim library you can simply isolate every part of the SUT, making easier to write Unit Test for classes written with no TDD and no Unit Testing in mind.

Gian Maria.

Testing

shim, Testing, VS11

Leave comment in GitHub discussion page
alkampfergit/personal-blog/discussions

←Previous
Getting the list of Type associated to a given export in MEF

Misusing an ORM
Next→