

Software Engineering for Enterprise Systems

Week 8 – Workshop - Part I

Testing the Application logic of your Enterprise Solution

Task 1: Group Exercise: Test Case Brainstorming (15 minutes)

Refer the program, NumberFun. It includes a C# program developed to find the average of a set of numbers entered as a user input. Read the xml documentation to understand the functionality of each method in the code. As a group, identify a sufficient set of test cases to test the functionality of NumberFun program. Discuss the identified test cases with your tutor.

Task 2: Create and run unit tests for managed code (15 minutes)

This exercise will give you practice at using C# unit test for a simple program. The objective of this exercise is to test a given program as a black box. The black box testing will not focus on the source code and only look at the inputs and outputs.

1. Open the “NumberFun” in your Visual studio environment. Make sure you can run the program.
2. Right-click on the solution NumberFun in Solution Explorer and choose **Add > New Project** and then select the **C# MSTest Unit Test Project for .NET** template.
3. Name the project as “NumberTest”, select the .NETFramework and create the test project. Now The NumberTest project is added to the NumberFun.
4. Now in the NumberTest project, add a reference to the NumberFun project. To do this, go to **Solution Explorer**, select Dependencies under the NumberFun project and then choose Add Reference (or Add Project Reference) from the right-click menu. In the **Reference Manager**, select the solution by expanding the projects and add the NumberFun.
5. Create a Test class to verify your NumberFun program. Rename the **UnitTest1.cs** as **AverageTest**
6. Observe the structure of the test program.

```

1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2  using System;
3
4  namespace NumberTest
5  {
6      [TestClass]
7      public class AverageTest
8      {
9          [TestMethod]
10         public void TestMethod1()
11         {
12             // your test code should go here
13         }
14     }
15 }

```

- The [TestClass] attribute is required on any class that contains unit test methods
 - Each test method that you want Test Explorer to recognize must have the [TestMethod] attribute.
7. Add a using statement of NumberFun to the test class. This will be helpful when calling NumberFun content without using fully qualified names.
 8. Now let's write our first test method. This method will help us to test, A test method must meet the following requirements:
 - It's decorated with the [TestMethod] attribute.
 - It returns void.
 - It cannot have parameters.

This method, will first add only one number to the list and test the average. Refer the following code

```

[TestMethod]
public void TestAverageSingleNumber()
{
    // Arrange
    double expected = 5;
    AverageFun avgFun=new AverageFun();

    // Act
    avgFun.AddNumber(expected);
}

```

```

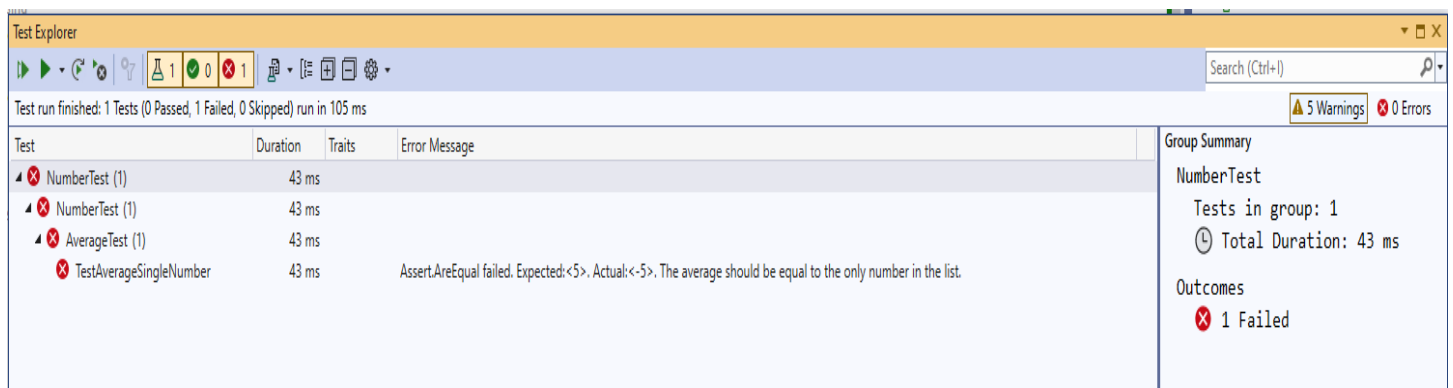
        // Assert
        Assert.AreEqual(expected, avgFun.GetAverage(), "The average should be
equal to the only number in the list.");
    }
}

```

9. Now let's **Build and run the test**. Navigate to Test > Test Explorer (or Test > Windows > Test Explorer) . Then Choose Run All to run the test.

While the test is running, the status bar at the top of the Test Explorer window is animated. At the end of the test run, the bar turns green if all the test methods pass, or red if any of the tests fail. In this case, the test fails.

Select the method in Test Explorer to view the details at the bottom of the window.



- 10. Fix your code:** The test result contains a message that describes the failure. You might need to drill down to see this message. For the AreEqual method, the message displays what was expected and what was actually received. What do you see in the message and why?
- 11.** Go back to your code's "GetAverage()" method and try to find the error to fix the miscalculation. What is the error that you specified.
- 12.** Re-run your test program to see whether you can successfully run the unit tests.
- 13.** Now refer the above example and develop a test method to insert multiple numbers and test the average

```

14. // Arrange
15. //create an instance
16. AverageFun avgFun = new AverageFun();
17.
18. //Add three numbers to the list
19.
20. // Act
21. // call the getAverage() Method
22.
23.
24. // Assert
25. //Write the Asset.AreEqual method

```

14. You may notice that the /create an instance (AverageFun avgFun = new AverageFun()) is now being repeated in multiple methods. Since we need an object from this class we can make this instantiation before start your test methods.

```
private AverageFun avgFun;

[TestInitialize]
public void TestInitialize()
{
    // Instantiate the AverageFun object before each test
    avgFun = new AverageFun();
}
```

TestInitialize is called right before your test is started. The TestInitialize is similar to the class constructor but is usually more suitable for long or async initializations. The TestInitialize is always called after the constructor and called for each test. This is usually used when we want to run a function/pre configuration before execution of a test.

15. Extend test your methods to Test the functionality of adding Zero numbers (An empty list.

16. Write another test method to test the functionality of Reset(). Hint: you can still use the Assert.AreEqual() method for this.

Task 3: Brainstorm Test cases for FleetPro Application (15 minutes)

Go to your FleetPro Application developed in Week 8, which includes the database integration. Carefully review the different layers and their respective classes to understand the testing requirements of the project. As you explore each layer, brainstorm and identify potential test cases. Create a list of key functionalities that need to be tested across the application layer

Task 4: Develop the test methods for FleetPro (30 Minutes)


We are going to prepare a set of MSTest methods to test the functionality in each layer in your system.

4.1 Create your Unit Test Project.

Open your **FleetProLayers** solution in Visual Studio. Add a new **MSTest Test Project C#** and name it **FleetProTests**. This will be your dedicated project for writing unit tests.

4.2 Add Project References

In your test project, you need to add the FleetProLayers as a reference. This will link the two projects for testing.

- Right-click on **FleetProTests** > **Add** > **Project Reference**
- Tick  **FleetProLayers** to link the main project for testing.

4.3 Install Moq Library (for Mocking Dependencies)

In **unit testing**, we want to test **only the logic of the class under test**, not the behavior of its dependencies. If we test them directly, we're doing **integration testing**, not **unit testing** and we can't control the test environment well.

Moq allows us to create **fake (mocked) versions of dependencies** where we can perform the following.

1. **Isolation:** By using Moq, you ensure that the tests are isolated from external factors, which helps in identifying issues within the class under test more easily.
2. **Behavior Verification:** Moq not only allows you to control return values but also to verify that specific methods were called with expected parameters, ensuring that the interactions between your class and its dependencies are as expected.
3. **Flexibility:** Moq provides flexibility in setting up different scenarios, such as simulating exceptions or configuring complex return values, which can be very useful for thorough testing.
4. **Readability:** Using Moq can make your tests more readable and maintainable by clearly defining the behavior of dependencies within the test itself

Moq framework in .net

Moq is a powerful and widely used mocking library in .NET that allows developers to create fake versions of interfaces and classes for unit testing purposes. In layered architectures, classes often depend on other services or repositories, making it difficult to test them in isolation. Moq helps overcome this by enabling developers to simulate the behavior of dependencies, define return values for method calls, track method invocations, and simulate exceptions—all without relying on actual implementations like databases or APIs. This ensures that unit tests remain fast, predictable, and focused solely on the logic of the class under test.

Watch

<https://learn.microsoft.com/en-us/shows/visual-studio-toolbox/unit-testing-moq-framework>

To install Moq to your test project, Right-click on Tests project > Manage NuGet Packages and install Moq

4.4 Organize your Test project

We are going to create three test classes for each layer in our application. Create three folders in your Test project and name them as Application, BusinessLogic, DataAccess.

You are going to create the following test classes and test the functionality of your code.

Test Class	Layer	Target Class
VehicleAppServiceTests.cs	Application Layer	VehicleAppService
VehicleServiceTests.cs	Business Logic Layer	VehicleService
InMemoryVehicleDBTests.cs	Data Access Layer	InMemoryVehicleDB

4.5 Implement your first test class VehicleAppServiceTests.cs by implementing the following steps

You may refer to the following steps

1. Make the class as a test class by marking it as a container for test methods. Use [TestClass]
2. Implement the [TestInitialize] It runs before each test. It should create a mock object of IVehicleService and injects it into a new instance of VehicleAppService. This ensures each test runs in isolation with a controlled, fake service layer.

```
private Mock<IVehicleService> _mockVehicleService;
private VehicleAppService _vehicleAppService;

[TestInitialize]
public void Setup()
{
    _mockVehicleService = new Mock<IVehicleService>();
    _vehicleAppService = new
    VehicleAppService(_mockVehicleService.Object);
}
```

3. Generate the test method GetAllVehicles_TestAllVehicles(). Implement the three steps, Arrange, Act and Assert. Each step should perform the following

Arrange: Sets up the mocked service to return a fake list of vehicles.

Act: Calls the GetAllVehicles() method on the application service.

Assert: Verifies that the result contains one item with the expected model.

```
// Arrange
var vehicles = new List<Vehicle> { new Vehicle { Id = 1, Model = "Model X" } };
_mockVehicleService.Setup(v => v.GetAll()).Returns(vehicles);

// Act
var result = _vehicleAppService.GetAllVehicles();

// Assert
Assert.AreEqual(1, result.Count);
Assert.AreEqual("Model X", result[0].Model);
```

4. Now develop the next Test method GetVehicle_TestReturnCorrectVehicle(). Follow the same Arrange, Act, Assert cycle

Arrange: Configures the mock to return a specific Vehicle when queried by ID.

Act: Calls GetVehicle(1) on the app service.

Assert: Confirms the returned object is not null and matches the expected model.

```
// Arrange
var vehicle = new Vehicle { Id = 1, Model = "Model S" };
_mockVehicleService.Setup(v => v.GetById(1)).Returns(vehicle);

// Act
var result = _vehicleAppService.GetVehicle(1);

// Assert
Assert.IsNotNull(result);
Assert.AreEqual("Model S", result.Model);
```

4.5 Run your Tests to observe the output

- Build your solution and make sure your project builds successfully.
- Open the Test Explorer from Test menu and click on “Run All Tests” to run the tests.
- Observe the outputs

4.6 Extend your project to develop other test classes

Now Develop your test classes: “VehicleServiceTests and InMemoryVehicleDBTests” to test the functionality of Business logic layer and the data access layer as well. Run your project to observe the output.

4.7 Test your database class

Expand your project to test the functionality of “EFVehicleRepository.cs” to test the correct use of database operations

----- **End of Workshop** -----