ICT3101 Lab 4

Testing with(out) Dependencies

When generating Unit tests, you will be faced with situations where your code relies on external dependencies (e.g., using third party libraries). However, Unit tests shouldn't test external dependencies. To Unit test, in these situations, we should replace the dependencies somehow.

Up until now, we've been testing a system without clear external dependencies. All our functions are contained within a single class and we're not really reliant on anything else. For this Lab, we're going to introduce (somewhat) external dependencies, and then go through a short introduction on how we can still Unit test despite these dependencies.

Steps:

Basically dependent

1. First, we're going to need to create some pseudo external dependencies, so we can learn to work around them. Let's begin by creating a function for generating "*magic*" numbers (in your Calculator class). It'll get these from reading an external file (representing something generated by an external system).

```
public double GenMagicNum(double input)
    {
        double result = 0;
        int choice = Convert.ToInt16(input);
        //Dependency------------------------------
        FileReader getTheMagic = new FileReader();
        //----------------------------------------
        string[] magicStrings = getTheMagic.Read("MagicNumbers.txt");

        if ((choice >= 0) && (choice < magicStrings.Length))
        {
            result = Convert.ToDouble(magicStrings[choice]);
        }
        result = (result > 0) ? (2 * result) : (-2 * result);
        return result;
    }
```

2. We'll need to create a "MagicNumbers.txt" file to be able to use this (create this text file and add wherever convenient). Furthermore, we need to generate a class called "FileReader". Using something like the following should suffice.

```
public class FileReader
{
        public string[] Read(string path)
        {
            return File.ReadAllLines(path);
        }
}
```

3. Based on this FileReader class, write some text into your "MagicNumbers.txt" file that will work with this "Read" function (one number per line). Then edit the "path" passed into the "Read" function so that it points to wherever you put your file.

4. Now, write a couple of Unit tests for this function and see that they pass. Create a few tests for positive, negative, and exceptional cases, based on your text file and this “Read” function’s logic. (This is where you may need to reconsider the location of the file or generate a path to it in the function.)

Breaking free (dependency injection)

Okay, so we’ve now got a function that we want to Unit test, but it is dependent on another class and, specifically, relies on its “Read” function to work correctly. So, how do we go about testing our function without also, implicitly, testing this dependency? Well, first we need to change the way our function manifests its dependency. Enter, Dependency Injection (DI).

Ideally, loosely coupled design is achieved, in part, by utilizing interfaces for classes. Specifically, in DI, we need interfaces for classes that deal with our external dependencies. In this way, we can have loosely coupled classes; hence, avoid having concrete classes within our functions.

Essentially, DI involves “injecting” dependencies into our classes, or functions, externally with the help of interfaces. We can do this via a variety of methods, some of these are as follows:

- By **parameters** into methods.
  - Pass these in as “new Objects”.
  - Issues: changes the method signature and may require multiple updates in your code.

- By **properties** of classes.
  - Create a class property that we can initialize in the **constructor**, and then for testing we can set it to a fake (mock) property in our test methods.
  - Issues: requires direct access to class properties and cannot be used in some DI **frameworks**.

- By **constructors** *(this is essentially what we’re doing with SpecFlow’s Context Injection)*
  - Create a private class property and set it within a **constructor**, but unlike with DI by **properties**, by passing it into the **constructor** as a **parameter**.
  - To avoid breaking other code we can overload our **constructor**: create one that takes in a **parameter** of our external dependency and another that doesn’t.
  - Issues: adding additional **constructor parameters** could cause messy code and issues elsewhere the constructors are called.

- By **Frameworks**
  - These are containers that map all interfaces to object **constructers**. It will be responsible for initializing objects at runtime. Hence, we can just pass around interfaces and have our **frameworks** instantiate them appropriately for us.
  - Existing DI **frameworks** to try: NInject, StructureMAp, Spring.NET, Autofac, Unity etc.
  - Issues: implementation varies for different **frameworks** and is dependent on the type of application you’re building.
  - **DI Frameworks are <u>beyond the scope</u> of this module** *(but to try, AutoFac is a good start).*

5. We're going to use **parameter** DI for our function. However, in order for any DI to work, we first need to set up an interface to pass around for our dependency. Let's create the following IFileReader interface (the leading 'I' being the standard naming convention here).

```
public interface IFileReader
{
    string[] Read(string path);
}
```

6. Then make sure that our FileReader class implements it: alter FileReader class as follows.

```
public class FileReader : IFileReader
```

7. Now let's apply DI to our GenMagicNum function. Remove the previous FileReader code from the function and use the new IFileReader, that you're now passing in instead (edit the type of the "getTheMagic" variable).

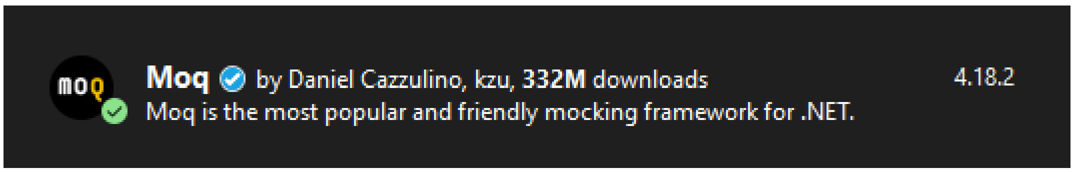
```
public double GenMagicNum(double input, IFileReader fileReader)
```

8. So now, when we call our function, we need to pass in a new FileReader. Alter your code (including tests) accordingly and test that everything still works – *this is why we write tests, after all*. [Bonus: does your push to Git Actions still build correctly? Can you fix this?]

Counterfeit constructions

Okay, so now that we've written our function in a DI compliant way, we can start to write tests that ignore the dependency on the FileReader class completely. This will allow us to (actually) Unit test our function. To achieve this, we need to create some *fakes* that replace our dependencies, we're going to do this using *Mock objects*.

9. There are several Mocking Frameworks available: Moq, NSubstitude, FakeItEasy, Rhino Mocks. We're going to use Moq. Install the following NuGet package.


Moq by Daniel Cazzulino, kzu, 332M downloads 4.18.2
Moq is the most popular and friendly mocking framework for .NET.

10. Let's now create a new "AdditionalCalculatorTests.cs" TestFixture in our Unit Tests project. In this class we're going to use Moq. Firstly, we need to copy the previous GenMagicNum tests in here.

11. As before let's create a private property for Calculator, but also add a Mock IFileReader.

```
private Calculator _calculator;
private Mock<IFileReader> _mockFileReader;
```

This command is creating a mock wrapper of our interface IFileReader.

12. Next, we'll add a Setup method as follows:

```
[SetUp]
public void Setup()
{
        _mockFileReader = new Mock<IFileReader>();
        _mockFileReader.Setup(fr =>
                fr.Read("MagicNumbers.txt")).Returns(new string[2]{"42","42"});
        _calculator = new Calculator();
}
```

13. So, what's happening here is that we're setting up our new mock IFileReader and then defining its "Read" function such that when it receives the string "`MagicNumbers.txt`" argument it will return the precise string array that we have defined: "`new string[2]{"42","42"}`". (Define this array to your intended file content.)

14. Now, for the final step in mocking magic! For our tests that are passing in a FileReader, that we copied over to AdditionalCalculatorTests, we now want to edit them to pass in our "`_mockFileReader`". However, because this is just a wrapper for the interface, we need to instead pass in the instantiated Object, that is: "`_mockFileReader.Object`". Finally, rerun the tests and see that they work with our mock. And voila, that's how to Mock a killing bird… or something like that.

A note on Integration testing:

Although we're using **Mocking** here to avoid unintentional Integration testing, a similar process can be used when you want to do integration testing of some modules when some other dependent modules aren't ready yet. E.g., in bottom-up or top-down integration tests you use **Drivers** and **Stubs**, where appropriate, before you can start adding more completed modules into the tests—*fake it till you make it*.

- **Drivers** are essentially a simple (fake) replacement for a calling program (calling our other modules, perhaps in specific orders).
  - The purpose is to allow the testing of lower levels of code when the upper levels aren't yet developed.
- **Stubs** are also simple replacements but, in contrast, for called programs.
  - The purpose is to allow the testing of upper levels of code when the lower levels aren't yet developed.

*Mocks Vs **Stubs***

Although similar in the way they are used for Dependency substitution (they are both considered *Test Doubles*), **Mocks** and **Stubs** are different:

- **Mocks** are "dynamic wrappers" for dependencies in tests. They are created with knowledge of the exact function's calls and in what sequence dependencies should be received from a system under test, to deem it working correctly.
- **Stubs** are user programmed classes that mimic a dependency's behaviour, but they achieve this with significant shortcuts as they may only do the minimum to satisfy the interface they implement.

Nice short Video on Integration Testing: https://www.youtube.com/watch?v=QYCaaNz8emY

—The end of steps/questions—