Overview

- Today we'll look at:
  - Moving from designing tests to writing tests
  - Formulating unit tests
  - Design decisions for testing our code
  - Running and using our tests

UNIVERSITY OF HULL

Unit testing – Class under test

Remember when we said we should write tests alongside our designs?

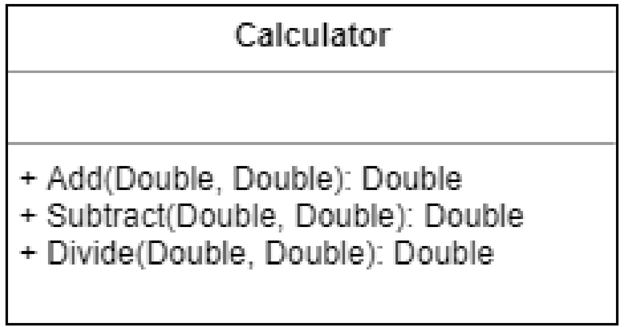
Let’s look at some of the challenges of this for unit testing…

…and consider what that means for us at design time.

Unit testing – Class under test

Here is our (very basic) class that we're looking at.

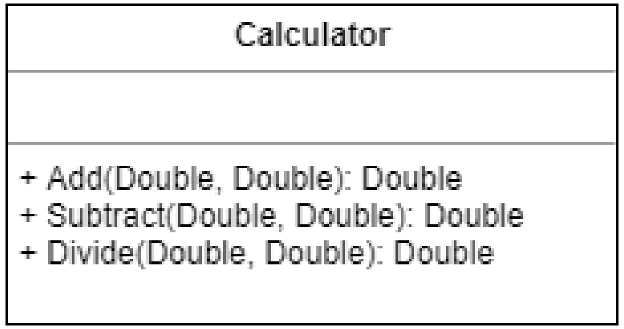
- This would be part of a larger application
- We want to test the units so we want to test individual behaviours ONLY


Calculator
+ Add(Double, Double): Double
+ Subtract(Double, Double): Double
+ Divide(Double, Double): Double

Unit testing – Class under test

We need to make some decisions

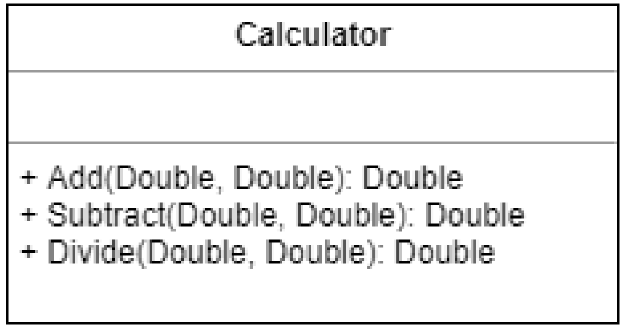
- We know that we need to take two doubles as parameters for these methods
  - We need to determine the order in which these parameters are used
- We also know that each method returns a double
  - We need to determine what it is that is returned. We can assume that we are returning the result of the calculation.


Calculator
+ Add(Double, Double): Double
+ Subtract(Double, Double): Double
+ Divide(Double, Double): Double

Unit testing – Class under test

For Add(Double, Double) we…

- **<u>Don’t</u>** care about the order…
  - 1 + 4 = 5
  - 4 + 1 = 5


Calculator
+ Add(Double, Double): Double
+ Subtract(Double, Double): Double
+ Divide(Double, Double): Double

- **Have known inputs:**

Two values of type Double (let’s call them a and b)

- **And a known output:**

A Double which is the result of the a and b doubles added together.

UNIVERSITY OF HULL

Unit testing

```
[TestClass]  // Decoration for any class that contains unit test methods
public class CalculatorUnitTest  //Convention - name the test class after the class it is testing or the functionality being tested.

{
    [TestMethod]
    public void AddUnitTest()
    {
        Calculator target = new Calculator(); // Arrange
        double a = 1;  // Arrange
        double b = 4;  // Arrange
        double expected = 5;  // Arrange

        double actual = target.Add(a, b);  // Act

        Assert.AreEqual(expected, actual);  // Assert

    }
}
```

$\Big\}$ Test steps

Is this enough?

That depends on our requirements…

If we are only ever expecting to have to add two, small, positive, whole numbers then yes this might be enough

Realistically we probably also have to manage other numbers (especially as this is a double type!)

Unit testing

```
[TestClass] // Decoration for any class that contains unit test methods
public class CalculatorUnitTest //Convention - name the test class after the class it is testing or the functionality being tested.

{
    [TestMethod]
    public void AddUnitTest()
    {
        Calculator target = new Calculator(); // Arrange
        for (double a = -10.99; a < 10.99; a += 0.01) // Arrange
        {
            for (double b = -10.99; b < 10.99; b += 0.01) // Arrange
            {
                double expected = a + b; // Arrange
                double actual = target.Add(a, b); // Act
                Assert.AreEqual(expected, actual); // Assert
            }
        }
    }
}
```

Test steps

Is this enough?

We are now testing a range of decimal and whole numbers that are both positive and negative.

Much better but…

- We are reliant on `a + b` to determine our expected result
  - Which is not the end of the world but the more reliance we have on external dependencies the worse our tests are – our tests could have their own errors and require more maintenance
- We are testing lots of different conditions in one test
  - Which makes it hard to know where the fault case lies if the test fails

Unit testing

```
[TestClass]  // Decoration for any class that contains unit test methods
public class CalculatorUnitTest  //Convention - name the test class after the class it is testing or the functionality being tested.

{
    [TestMethod]
    public void AddUnitTest_FirstNegative_WholeNumber()
    {
        // … parameter a is negative, both a and b are whole numbers
    }
    [TestMethod]
    public void AddUnitTest_SecondNegative_WholeNumber()
    {
        // … parameter b is negative, both a and b are whole numbers
    }
    [TestMethod]
    public void AddUnitTest_FirstNegative_DecimalNumber()
    {
        // … parameter a is negative, both a and b are decimal numbers
    }
     // … etc. etc.
}
```

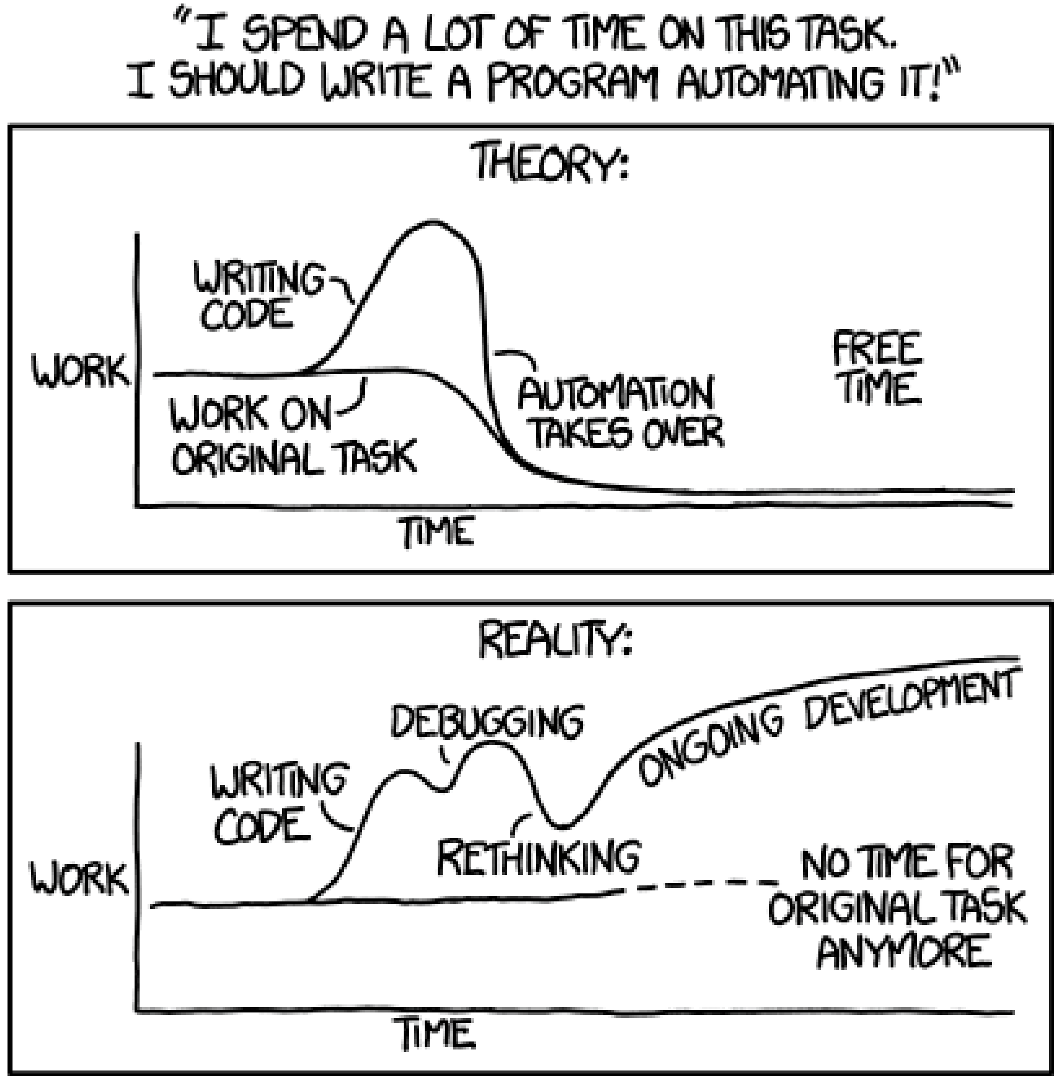
This alternative gives us the benefit of knowing EXACTLY where the fault lies **and** helps us to be absolutely sure that we have covered all test cases

But it will take a lot longer to write and is much harder to maintain

We need to find a good balance:

- In sections of your code where there are safety concerns or hazards are more likely if the code fails, vigorous testing like this is likely to be more necessary
- Otherwise we can save some time and effort, at the risk of some higher chance of faults slipping through

UNIVERSITY OF HULL



Unit testing – Designing Tests

Thinking about these uncertainties we (hopefully) realise that **designing** our tests is critically important

Designing tests is not writing the test code

You should specify the tests before you write the test code

- You are further away from the code
- You can think about things in a higher level of abstraction
- You can track each of your test cases to ensure you have tested all of the possible functionality
- You can map your tests back to requirements, user goals, etc.

UNIVERSITY OF HULL

Unit testing – Designing Tests

Designing our unit tests for Add(Double, Double)

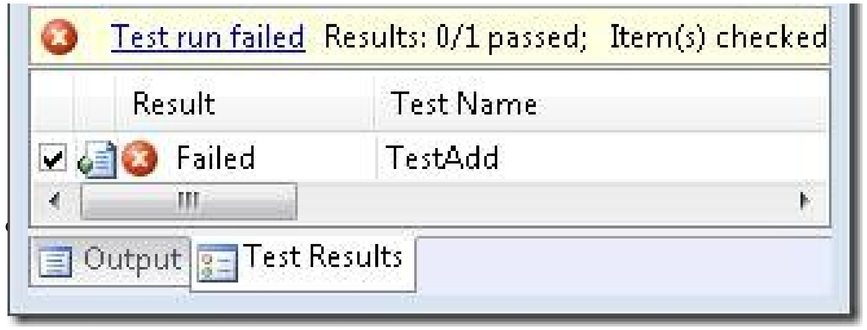
ID	Input a	Input b	Output
1.1	Positive whole number	Positive whole number	Sum of a + b as a positive whole number
1.2	Positive whole number	Negative whole number	Sum of a + b as a positive or negative whole number
1.3	Negative whole number	Positive whole number	Sum of a + b as a positive or negative whole number
1.4	Negative whole number	Negative whole number	Sum of a + b as a negative whole number
1.5	Positive number with fraction	Positive number with fraction	Sum of a + b as a positive number with or without a fraction
1.6	Etc.	Etc.	Etc.

This is a very simplistic view

– you should refer to the last lectures in the series to see all of the elements that make up a test

Unit testing – Class under test

- Now we have one or more test cases that fail…



Test run failed Results: 0/1 passed; Item(s) checked

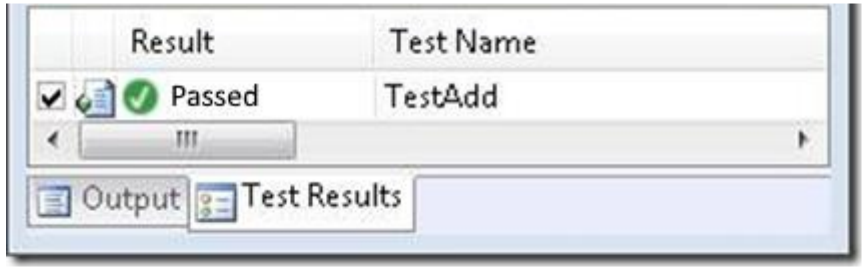
	Result	Test Name
✓	Failed	TestAdd

Output | Test Results

- We write our code…

```
public class Calculator
{
     public double Add(double a, double b)
     {
         return a + b;
     }
}
```

- And confirm that our code is fit for purpose when it passes all of the tests for this unit…



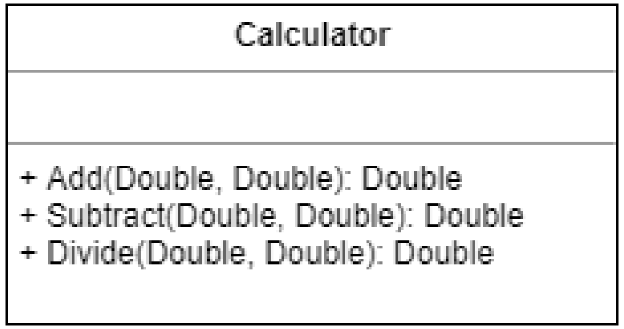
	Result	Test Name
✓	Passed	TestAdd

Output | Test Results

Unit testing – Class under test

We need to make some decisions

- We know that we need to take two doubles as parameters for these methods
- For Subtract(Double, Double) we **do** care about the order…
  - 1 - 4 = -3
  - 4 - 1 = 3
- UML doesn’t tell us how to manage our behaviours based on the parameters it defines so we need to make a decision and it would be a good idea to document it.
- In this case lets make the decision that the second parameter is subtracted from the first
  - Which is convention – but you can’t always rely on convention


Calculator
+ Add(Double, Double): Double
+ Subtract(Double, Double): Double
+ Divide(Double, Double): Double

UNIVERSITY OF HULL

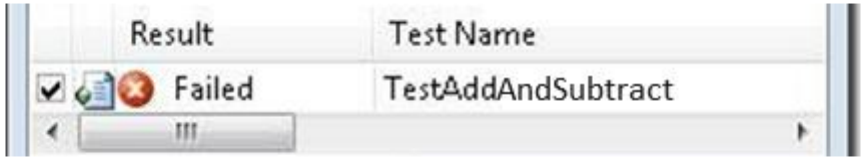
Unit testing

```
[TestClass] // Decoration for any class that contains unit test methods
public class CalculatorUnitTest //Convention - name the test class after the class it is testing or the functionality being tested.

{
    [TestMethod]
    public void AddAndSubtractUnitTest()
    {
        Calculator target = new Calculator(); // Arrange
        for (double a = -10.99; a < 10.99; a += 0.01) // Arrange
        {
            for (double b = -10.99; b < 10.99; b += 0.01) // Arrange
            {
                double expected = a + b; // Arrange
                double actual = target.Add(a, b); // Act
                Assert.AreEqual(expected, actual); // Assert

  →             expected = a - b; // Arrange
  →             actual = target.Subtract(a, b); // Act
  →             Assert.AreEqual(expected, actual); // Assert
            }
        }
    }
}
```

$\}$ Test steps

If we assume that we are ok with testing lots of cases in one test, and relying on + and -, is this ok?

We can now test both Add() and Subtract() in one test!



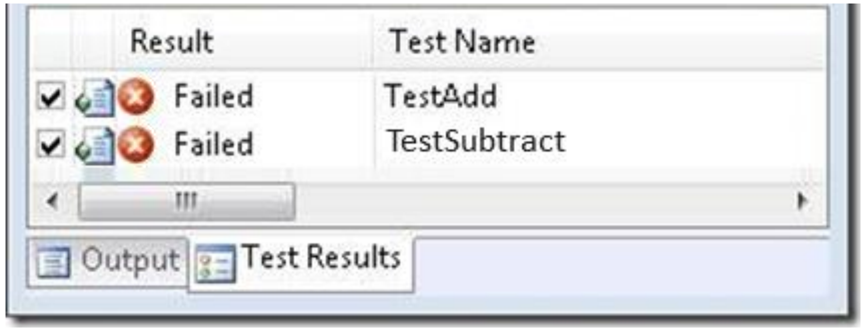
Result	Test Name
Failed	TestAddAndSubtract

But now it's very difficult to know where the fault is if we get a test fail

Our Unit Test is now testing more than one unit which is considered bad

Unit testing – Class under test

- So we should have one or more test cases that fail for one unit each…



Result	Test Name
Failed	TestAdd
Failed	TestSubtract

…and we continue to write our code to make those tests pass…

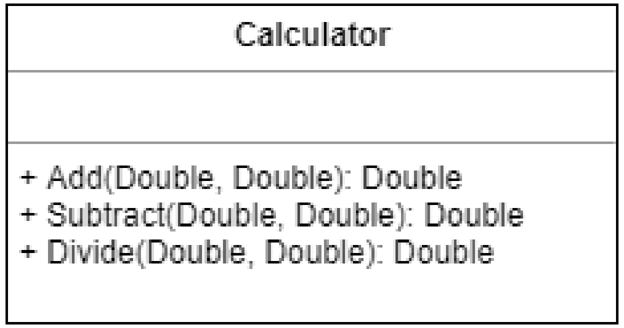
```
public class Calculator
{
    public double Add(double a, double b)
    {
        return a + b;
    }

    public double Subtract(double a, double b)
    {
        return a - b;
    }
}
```

Assertions

- Assertions are a "language" of testing - constraints that you place on the output.
  - Assert class
    - Verifies conditions in unit tests using true/false propositions. E.g:
      - AreEqual(Double, Double, Double, String) ***– So far this is the only one we've seen used***
      - IsInstanceOfType(Object, Type) – Type checking
      - IsNotNull(Object) – Checking for instances
  - CollectionAssert class. E.g:
    - Verifies true/false propositions associated with collections in unit tests.
      - AreEqual(ICollection, ICollection) – Collection (e.g. list) equality comparer
      - AllItemsAreInstancesOfType(ICollection, Type) – Collection type checking
  - StringAssert class. E.g:
    - Verifies true/false propositions associated with strings in unit tests.
      - EndsWith(String, String)

UNIVERSITY OF HULL

Unit testing – Class under test


Calculator
+ Add(Double, Double): Double
+ Subtract(Double, Double): Double
+ Divide(Double, Double): Double

We need to make some decisions

- We know that we need to take two doubles as parameters for these methods
- For Divide(Double, Double) we **do** care about the order…
  - 1 / 4 = 0.25
  - 4 / 1 = 4
- UML doesn’t tell us how to manage our behaviours based on the parameters it defines so we need to make a decision and it would be a good idea to document it.
- In this case lets make the decision that the first parameter is divided by the second
  - Which is convention – but you can’t always rely on convention

But what have we forgotten so far? Something that is a KEY part of testing…

- Testing error conditions
- Testing for the unexpected (edge cases)

Unit testing – Error Conditions

What happens when something uses your code in a way it wasn't designed to be used?

For example:

- What if you were passed two values that were very, very large, and they you were expected to return them added together?

```
double c = Add(Double.MaxValue, Double.MaxValue);
```

- What if you were passed two values that were very, very small, and they you were expected to subtract them?

```
double c = Subtract(Double.MinValue, Double.MinValue);
```

- What if you were passed a divisor value that can be a double but can not be a divisor?

```
double c = Divide(4, 0);
```

It is **much** easier to think about these error conditions in isolation from the rest of your code, which is one reason why designing and writing your tests before your code is such a good idea.

Otherwise you easily fall into the trap of testing only the code you have written in the way you expect it to be used.

Unit testing – Checking for Exceptions

```
public class UnitTest1
{
    // ...

    [TestMethod]
    // Identify a particular expected exception
    // (test passes if thrown, otherwise fails)
 → [ExpectedException(typeof(System.DivideByZeroException ))]
    public void DivideByZeroTest()
    {
        Calculator calcObject = new Calculator();
        double result = calcObject.Divide(15, 0);
    }
}
```

OR

```
public class UnitTest1
{
    // ...

    [TestMethod]
    public void DivideByZeroTest()
    {
        Calculator calcObject = new Calculator();
        try
        {
            double result = calcObject.Divide(15, 0);
        }
   →    catch(System.DivideByZeroException e)
        {
            return; // Return will pass the test as nothing failed
        }
        Assert.Fail(); // Exception did not occur so force a fail
    }
}
```

UNIVERSITY OF HULL

Unit testing – Code Reuse

```
[TestClass]
public class CalculatorUnitTest
{
    [TestMethod]
    public void AddUnitTest()
    {
        Calculator target = new Calculator(); // Arrange
        double a = 1; // Arrange
        double b = 4; // Arrange
        double expected = 5; // Arrange

        double actual = target.Add(a, b); // Act

        Assert.AreEqual(expected, actual); // Assert
    }

    [TestMethod]
    public void SubtractUnitTest()
    {
        Calculator target = new Calculator(); // Arrange
        double a = 1; // Arrange
        double b = 4; // Arrange
        double expected = -3; // Arrange

        double actual = target.Subtract(a, b); // Act

        Assert.AreEqual(expected, actual); // Assert
    }
}
```

It looks like we're doing the same thing multiple times.

A good design principle for programming is to minimise repetition as it means making changes is far more difficult.

It is also harder to verify that each instance of the code is written correctly if you have to verify lots of times.

Copy-pasting code is also the devil so…

Where possible we should write once, use often.

Unit testing – Code Reuse

```
[TestClass]
public class CalculatorUnitTest
{
    Calculator target = new Calculator(); // Arrange

    [TestMethod]
    public void AddUnitTest()
    {
        double a = 1; // Arrange
        double b = 4; // Arrange
        double expected = 5;  // Arrange

        double actual = target.Add(a, b); // Act

        Assert.AreEqual(expected, actual); // Assert
    }

    [TestMethod]
    public void SubtractUnitTest()
    {
        double a = 1; // Arrange
        double b = 4; // Arrange
        double expected = -3;  // Arrange

        double actual = target.Subtract(a, b); // Act

        Assert.AreEqual(expected, actual); // Assert
    }
}
```

We could take out this line and put it inside our class.

This is convention for most normal object oriented classes that we write but **it doesn't work here**

Each test method must exist in isolation

- The SubtractUnitTest() would be supplied with the same Calculator instance as AddUnitTest(). If the first method to use the instance changed the object then it could affect the test conditions.
- We can't allow this to happen in our tests as it may cause unexpected test behaviour.
- We also can't rely on our tests being run in a particular order

UNIVERSITY OF HULL

Unit testing – Code Reuse

```
[TestClass]
public class CalculatorUnitTest
{
    Calculator target;

    [TestInitialise]
    public void TestInit()
    {
        this.target = new Calculator();
    }

    [TestMethod]
    public void AddUnitTest()
    {
        double a = 1; // Arrange
        double b = 4; // Arrange
        double expected = 5; // Arrange

        double actual = target.Add(a, b); // Act

        Assert.AreEqual(expected, actual); // Assert
    }

    [TestMethod]
    public void SubtractUnitTest()
    {
        double a = 1; // Arrange
        double b = 4; // Arrange
        double expected = -3; // Arrange

        double actual = target.Subtract(a, b); // Act

        Assert.AreEqual(expected, actual); // Assert
    }
}
```

We can define an uninstanced variable of type Calculator

Then define a new method, decorated with the TestInitialise attribute

- This attribute ensures that the method is run before each test method in the class

And then create a new instance per test

Now they are completely independent instances used in each test method

UNIVERSITY OF HULL

Unit testing – Cleanup

```
[TestClass]
public class CalculatorUnitTest
{
    Calculator target;

    [TestInitialise]
    public void TestInit()
    {
        this.target = new Calculator();
    }

    [TestMethod]
    public void AddUnitTest()
    {
        // do a test…
    }

    [TestMethod]
    public void SubtractUnitTest()
    {
        // do a test…
    }

    [TestCleanup]
    public void TestCleanup()
    {
        this.target = null;
    }
}
```

We want to be absolutely sure our instances are independent for each test.

There may be situations where this isn't inherently realistic.

We can use the TestCleanup attribute to run code in the associated class after each test.

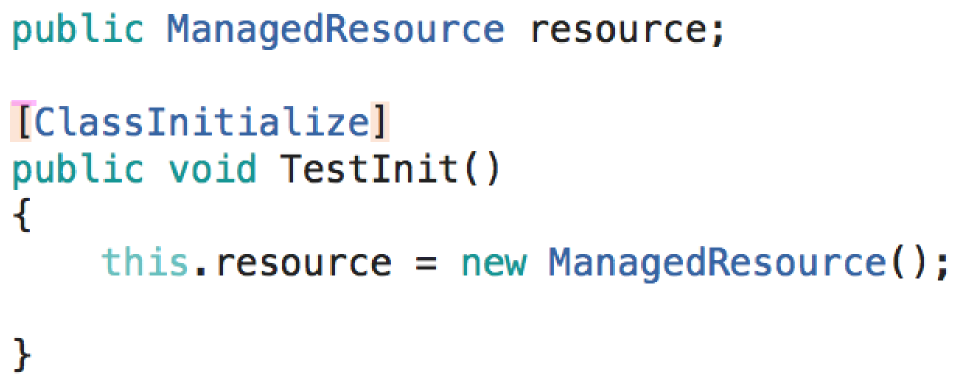
- Again this prevents a situation where we would otherwise need to replicate code at the end of each test.

In this case the cleanup isn't completely necessary but we can use it to be absolutely sure that the same instances aren't used in multiple tests.

More Fixtures

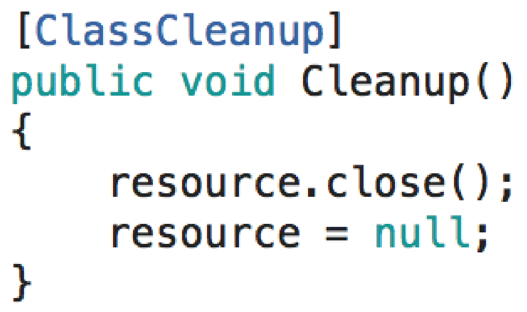
These methods run once for the test class

- [ClassInitialize] Identifies a method that contains code that must be used before any of the tests in the test class have run and to allocate resources to be used by the test class. This class cannot be inherited.



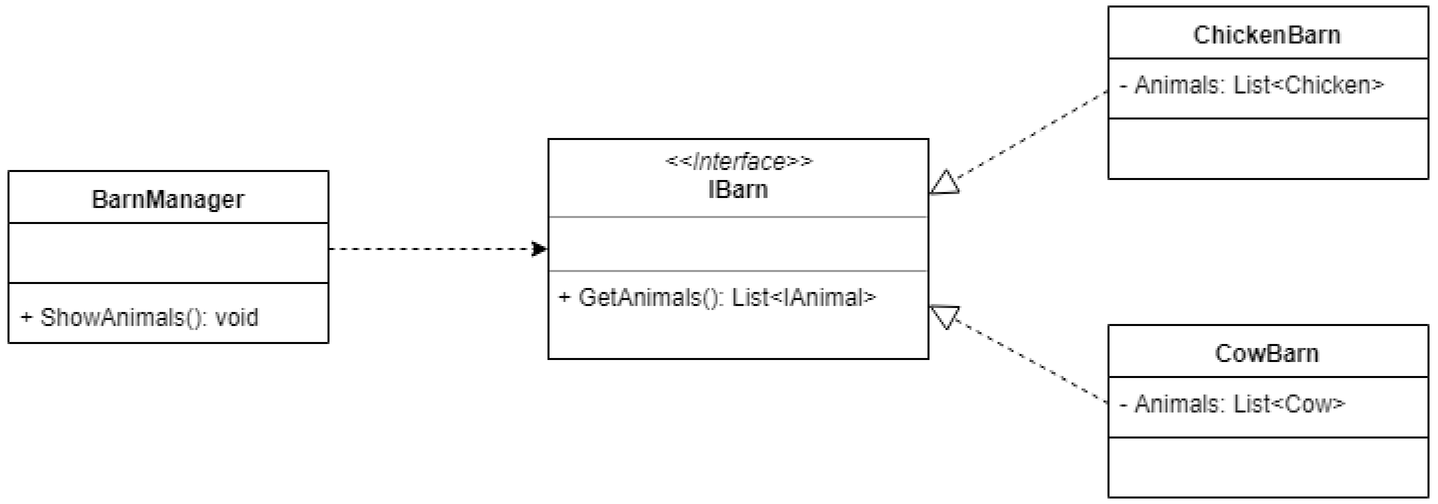
```
public ManagedResource resource;

[ClassInitialize]
public void TestInit()
{
    this.resource = new ManagedResource();

}
```

- [ClassCleanup] identifies a method that contains code to be used after all the tests in the test class have run and to free resources obtained by the test class. This class cannot be inherited.



```
[ClassCleanup]
public void Cleanup()
{
    resource.close();
    resource = null;
}
```

Dependencies



Life gets harder when we have dependency.

Here, BarnManager is dependent on a type that implements the IBarn interface in order to do its work

If we test and include the dependencies then we are doing integration tests.

If we want core unit testing we don’t want our unit tests to test other units - so how can we prevent this dependency?

UNIVERSITY OF HULL

Mocking

- **Test Dummy**
  - We write our own object which will be used for testing. We make sure that object always returns the same thing.
  - Just an object needed for execution of the class, but no control observation needed
- **Test Stub**
  - Typically an override of the behaviour
  - Often made automatically at test runtime
  - Enables us to control what values are returned when the class under test calls methods on a collaborating object
- **Test Shim**
  - For behaviours which cant be overridden
  - Actually modifies the underlying code that is going to be used at runtime
  - Useful when you don’t know what the underlying behaviour is
    - e.g. you are using a system library type like Random and calling Random.Next(Int32)
- **Test Mock**
  - Enables us to observe what method calls (including any parameters) that are made to a collaborating object from the class under test
  - In this way we can test that methods are called as part of our test.
  - Useful if the purpose of a piece of code is to call other methods in a specific order.

UNIVERSITY
OF HULL

C# Unit Testing Frameworks

- **MSTest**
  - https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest
- **NUnit**
  - https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit
- **xUnit**
  - https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test
- **Moq**
  - https://github.com/Moq/moq4/wiki/Quickstart