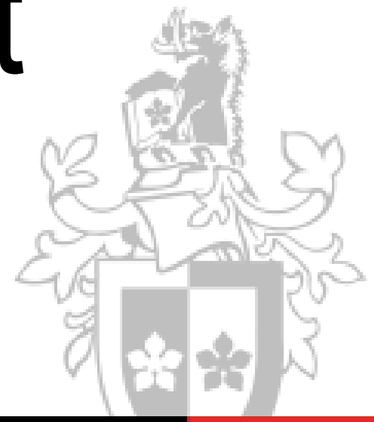
Unit Testing and Test-driven Development

by Andrew Cain and Willem van Straten

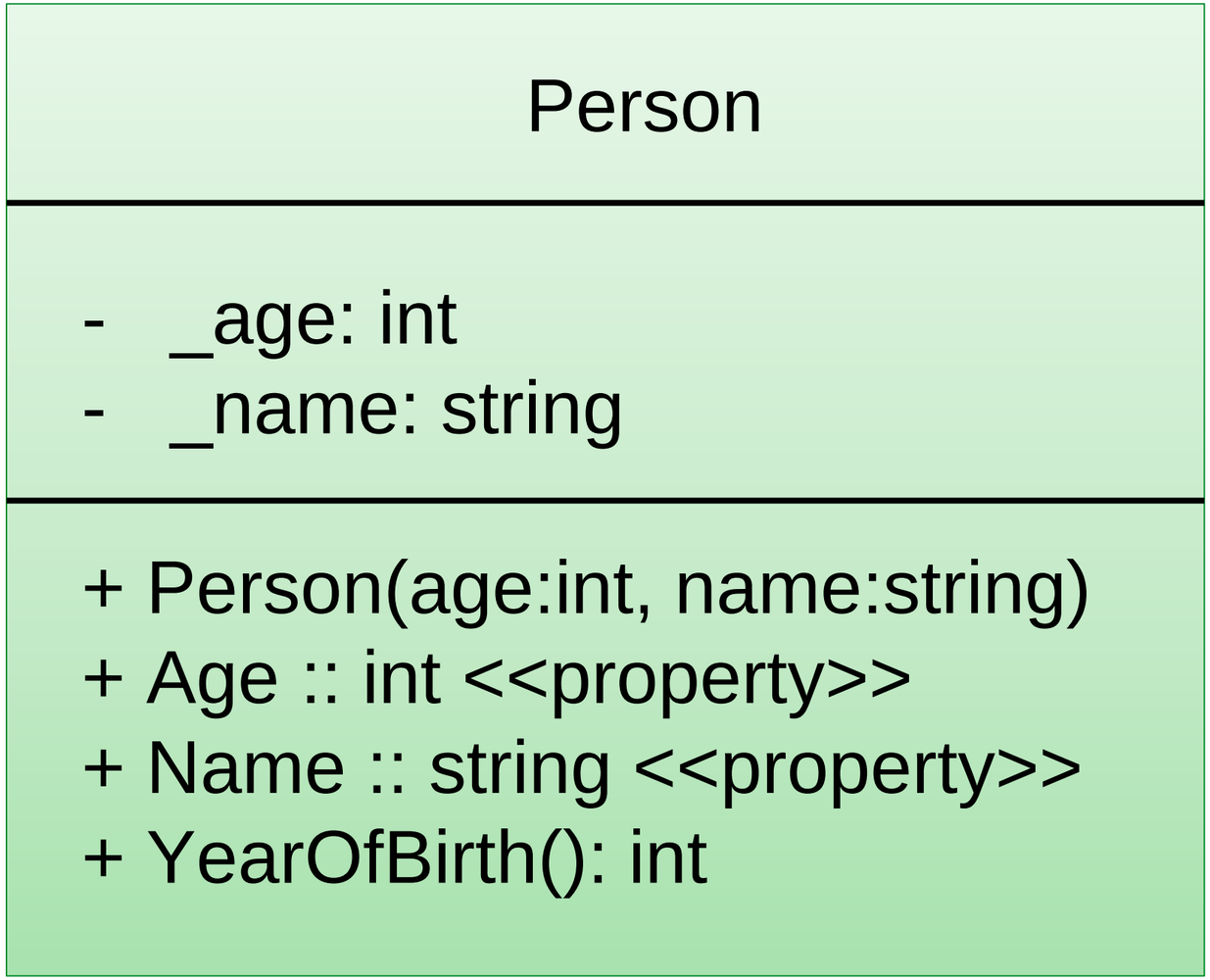


SWINBURNE UNIVERSITY OF TECHNOLOGY

Object Oriented Programming
Object Oriented Programming in C++

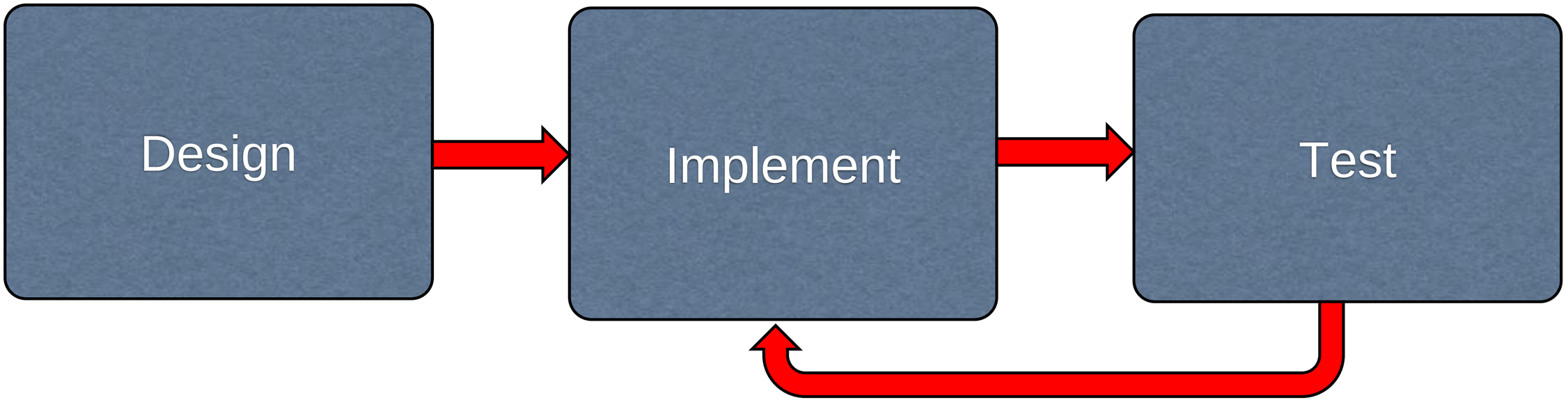
Object oriented programming involves creating objects that know and do things

Let’s have some recaps on C# codes!



Create the class given and write a main program to run the class created.

Developers use tools and processes to help guide the creation of programs



Getting programs to work correctly can be tricky at the best of times

The right tools and processes will help ensure you get a working product

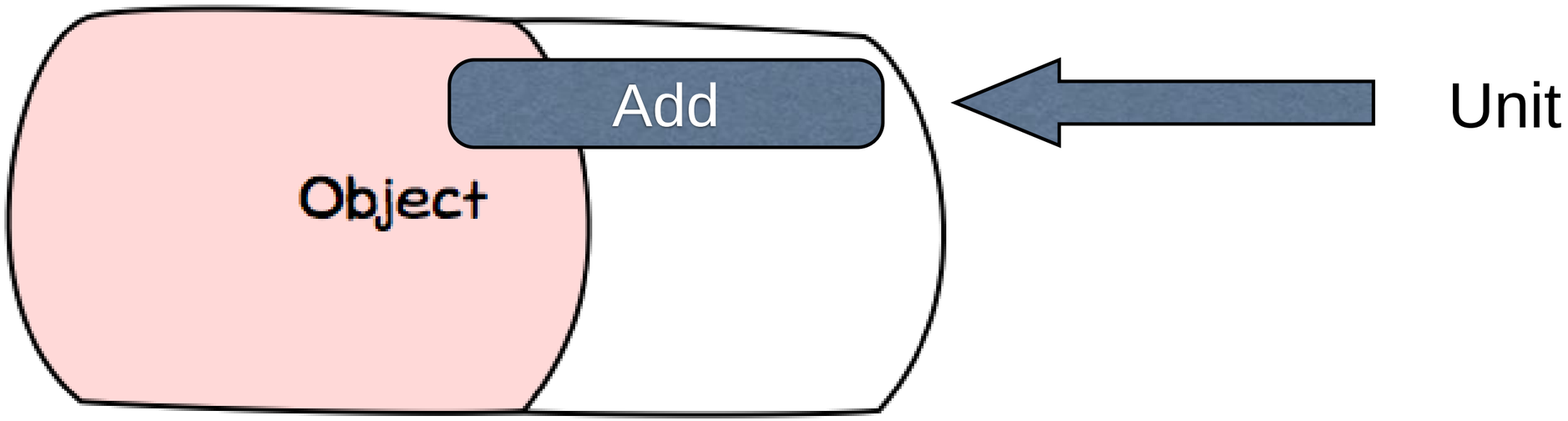
Use Unit Testing tools to help design and build your programs

Verify object functionality
with unit tests

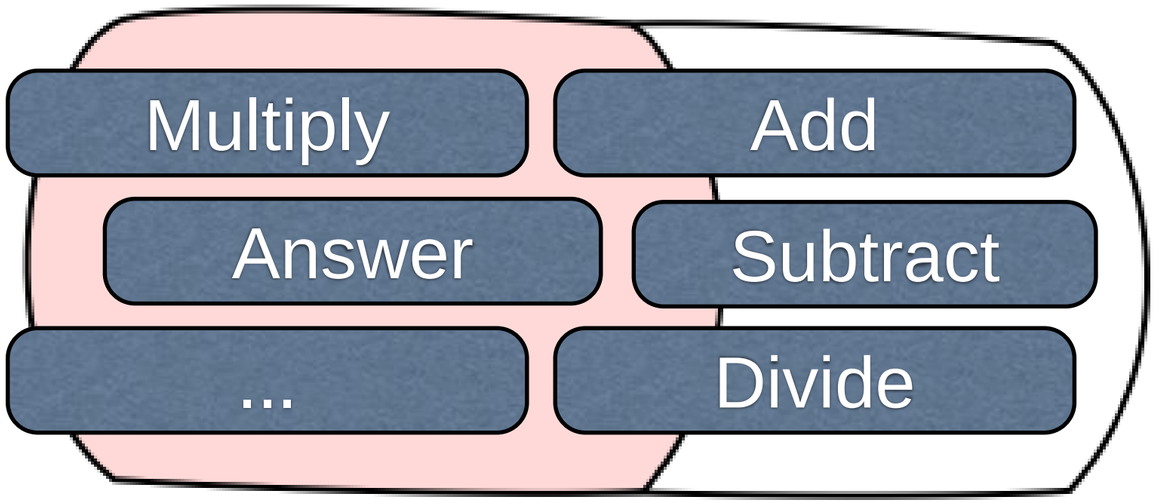
…to verify that our functions work as expected

Begin developing a suite of test cases that can be run each time you work on your logic

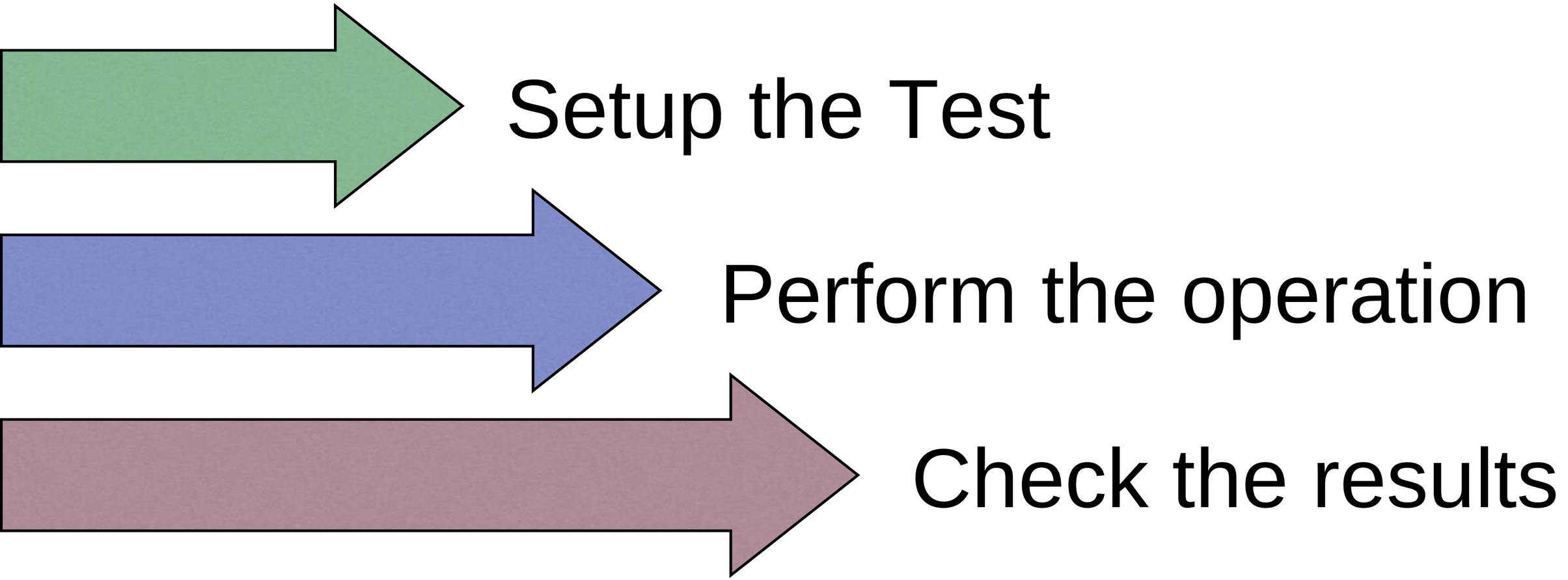
Units represent the smallest testable part of your program



Use many small tests to check as much of the program functionality as possible



Each test checks if that part of the functionality is working correctly



Speed up testing with automated unit testing tools and classes to make testing easier

The *x*Unit framework provides tools to perform unit testing in many languages

*N*Unit	unit-testing framework for all .Net languages such as C#, VC, VB.Net
*J*Unit	…for Java programming language
CppUnit	…for C++ programming language

Create test fixtures that contain unit tests

to indicate that a class contains test methods

```
[TestFixture()]
public class TestCalc
{
   [Test()]
   public void TestPush ()
   {
      RpnCalculator c = new RpnCalculator();
      int actual;

      c.Push(5);
      actual = c.Answer();

      Assert.AreEqual(5, actual, "Test P
   }
}
```

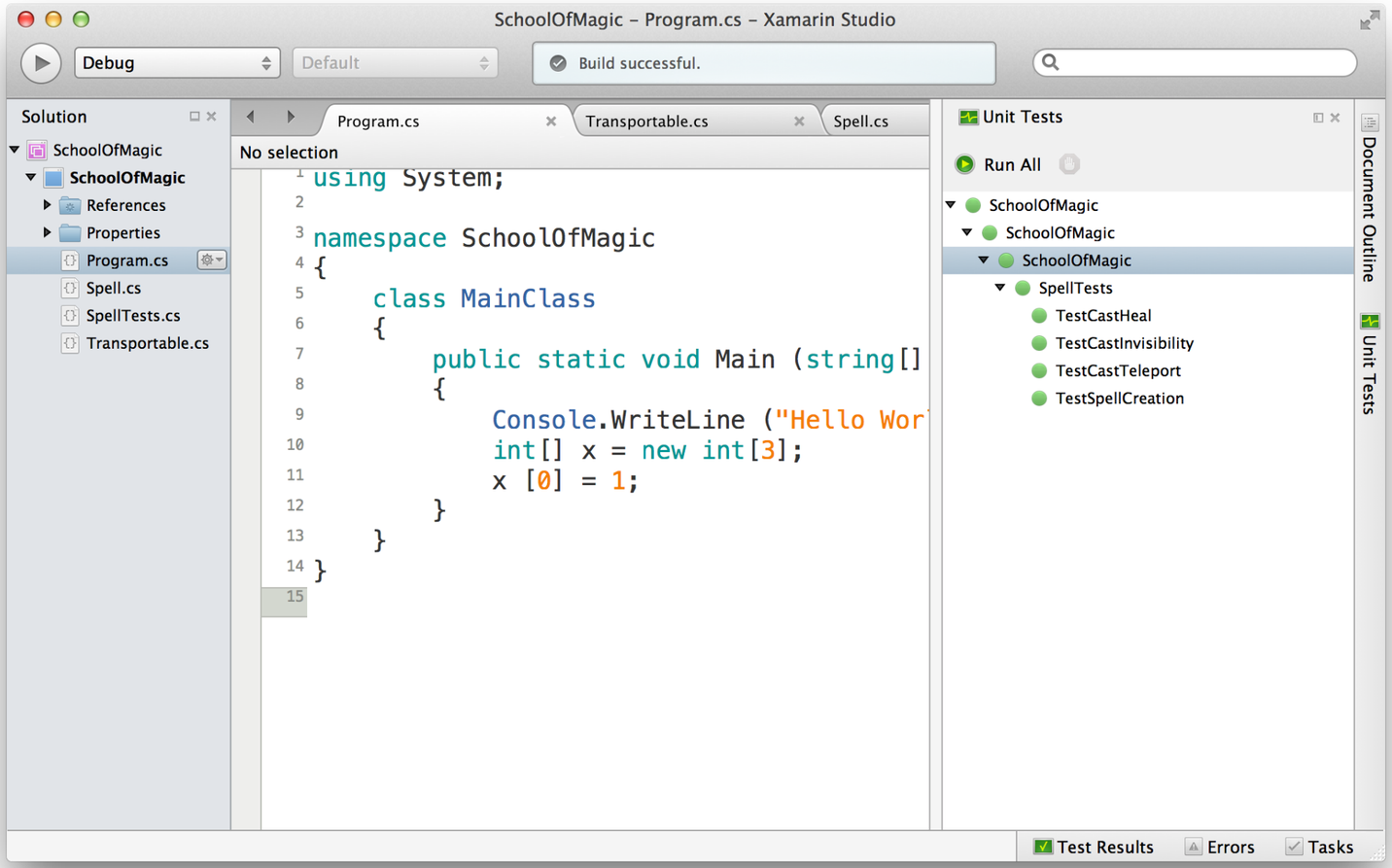
Setup

Perform

Check

to confirm whether the test cases is producing the expected result or not

Use the tools to run all of the tests each time you make changes



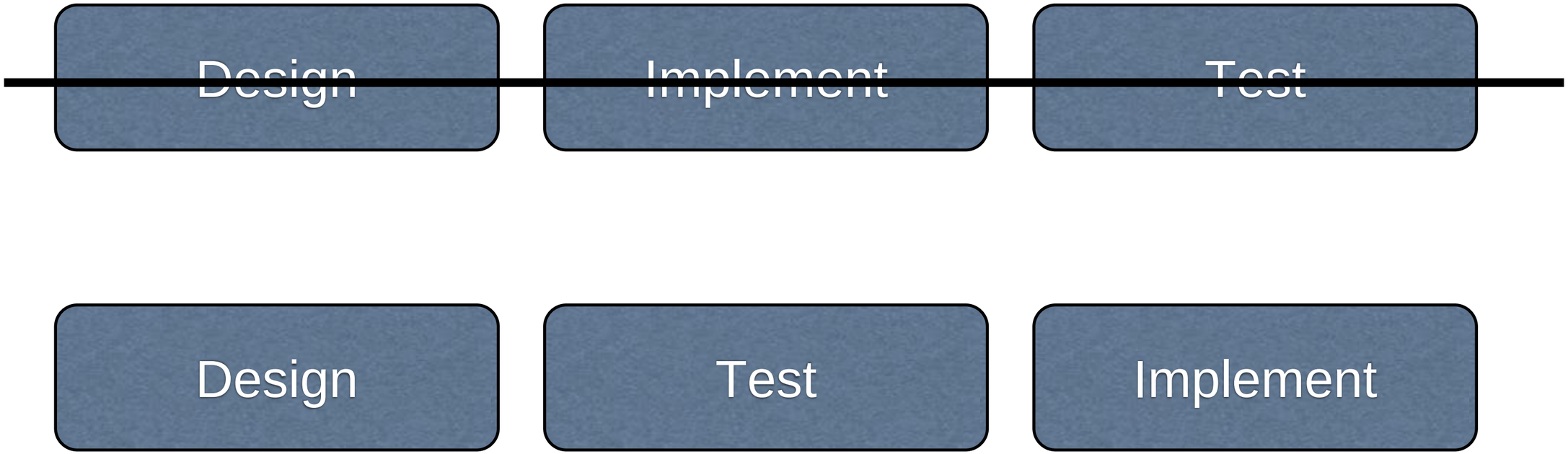
```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Enter two numbers\n");
        int number1;
        int number2;
        number1 = int.Parse(Console.ReadLine());
        number2 = int.Parse(Console.ReadLine());

        MathsHelper helper = new MathsHelper();
        int x = helper.Add(number1, number2);
        Console.WriteLine("\nThe sum of " + number1 +
            " and " + number2 + " is " + x);
        Console.ReadKey();
        int y = helper.Subtract(number1, number2);
        Console.WriteLine("\nThe difference between " +
              number1 + " and" + number2 + "  is " + y);
        Console.ReadKey();
    }
}

public class MathsHelper
{
    public MathsHelper() { }
    public int Add(int a, int b)
    {
        int x = a + b;
        return x;
    }

    public int Subtract(int a, int b)
    {
        int x = a - b;
        return x;
    }
}
```

Example 2

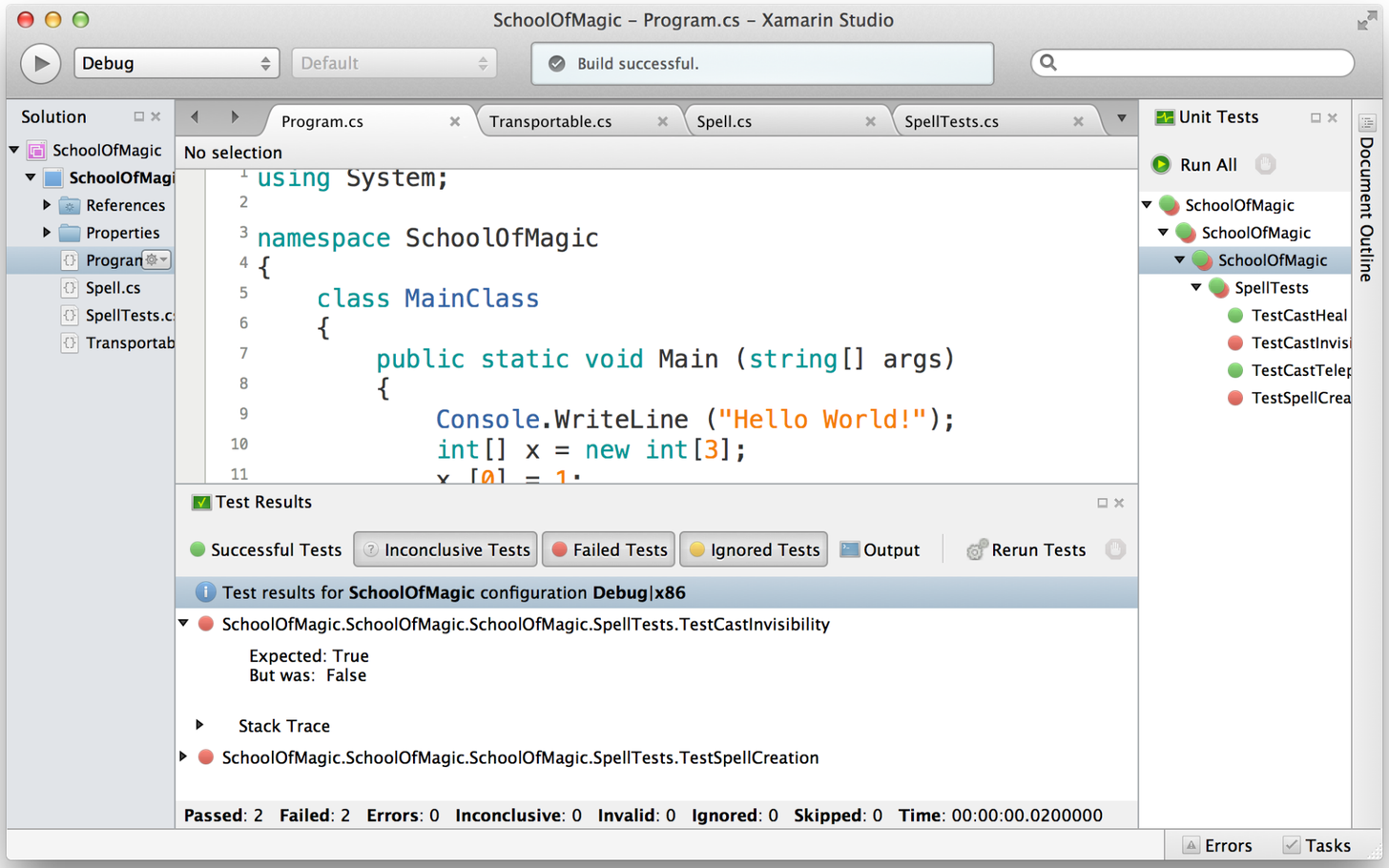
```
[TestFixture]
public class TestClass
{
    [TestCase]
    public void AddTest()
    {
        MathsHelper helper = new MathsHelper();
        int result = helper.Add(20, 10);
        Assert.AreEqual(30, result);
    }

    [TestCase]
    public void SubtractTest()
    {
        MathsHelper helper = new MathsHelper();
        int result = helper.Subtract(20, 10);
        Assert.AreEqual(10, result);
    }
}
```

Step it up with test driven development

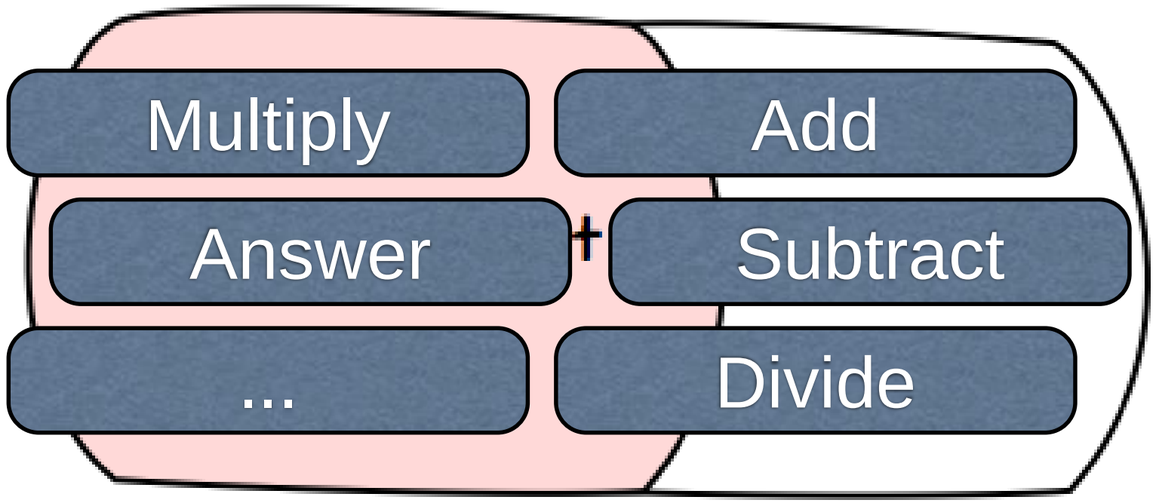
Write the tests first!



Add features only where the tests fail: create a test, watch it fail, make it work!



Add tests to expand the program's functionality

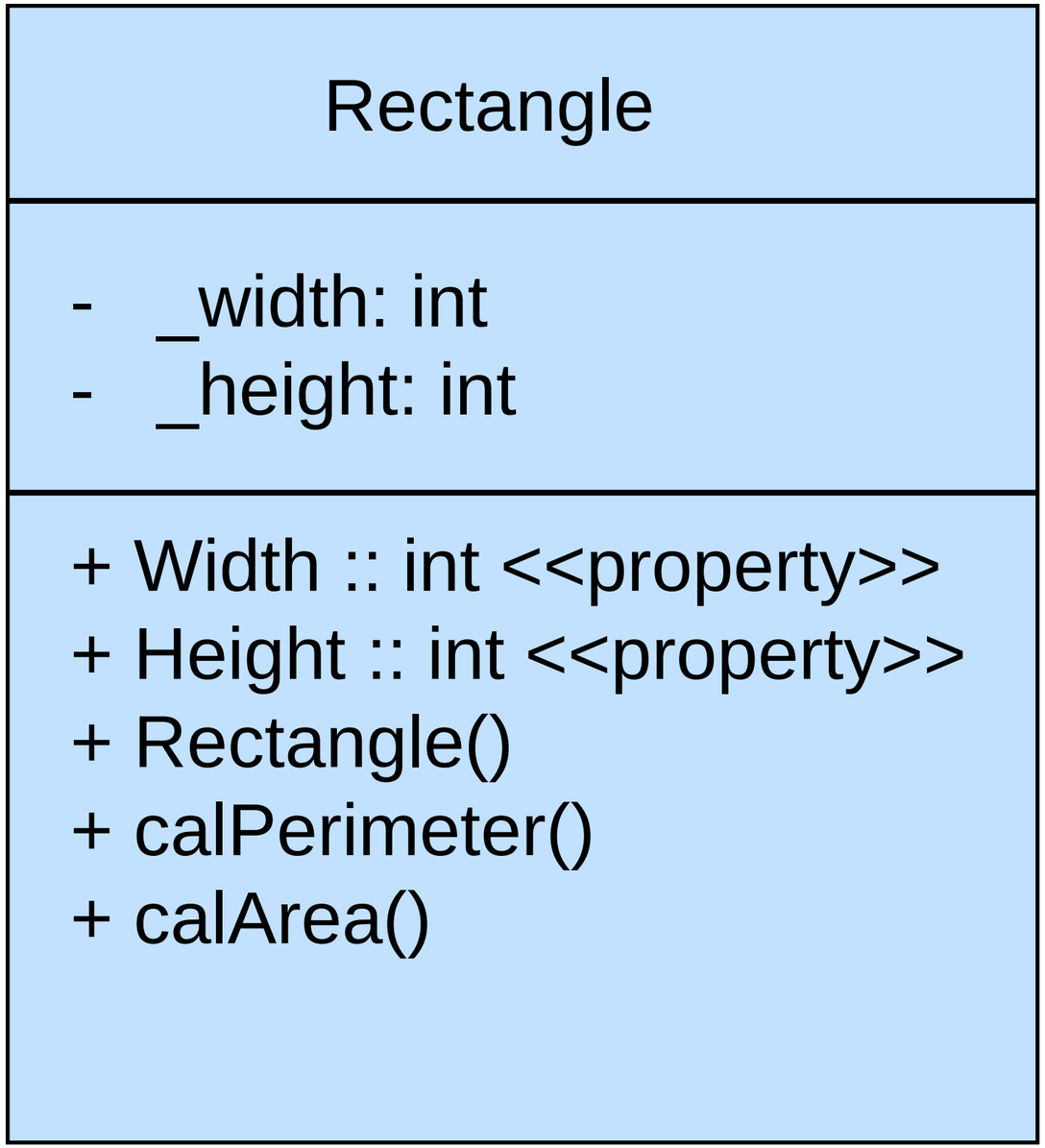


Use Unit Testing tools to help design and build your programs

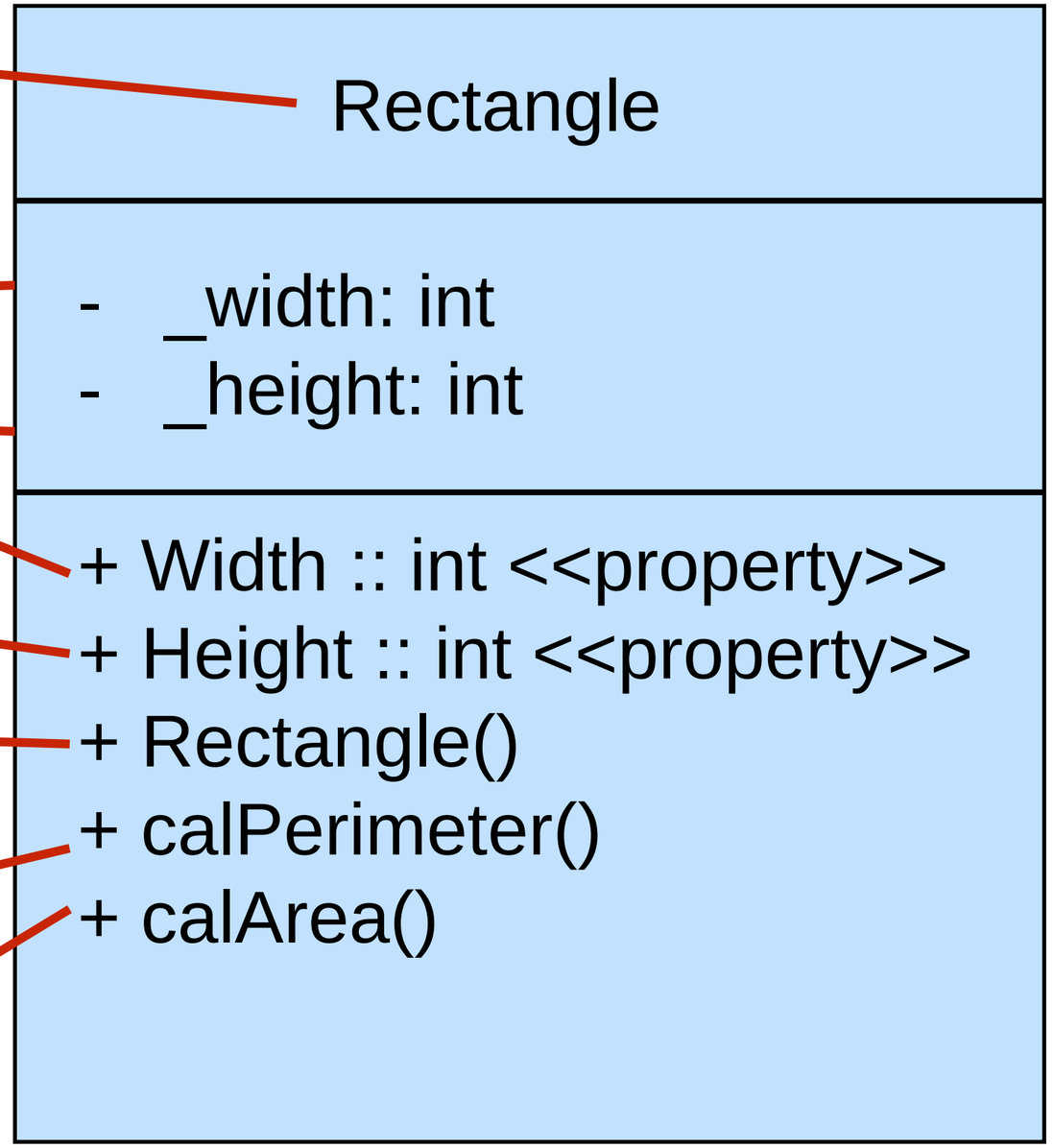
Build the program right with Unit Testing and Test Driven Development

Let’s try out an example!

Create the class
Rectangle
in C#



```
public class Rectangle
{
    private int _width;
    private int _height;

    public int Width{
        get{ return _width; }
        set{ _width = value; }
    }

    public int Height{
        get{ return _height; }
        set{ _height = value; }
    }

    public Rectangle ()
    {
        _width = 0;
        _height = 0;
    }

    public int calPerimeter(){
        return ((_width * 2) + (_height * 2));
    }

    public int calArea(){
        return _width * _height;
    }
}
```



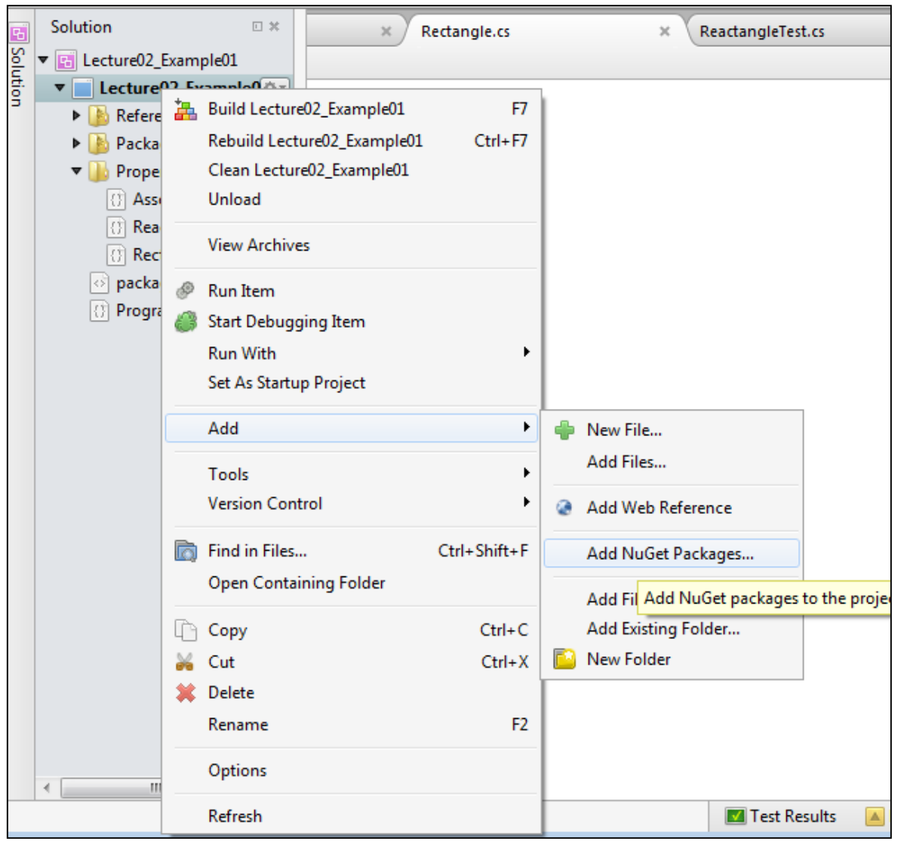
Let’s create two ***unit test*** for the class Rectangle!

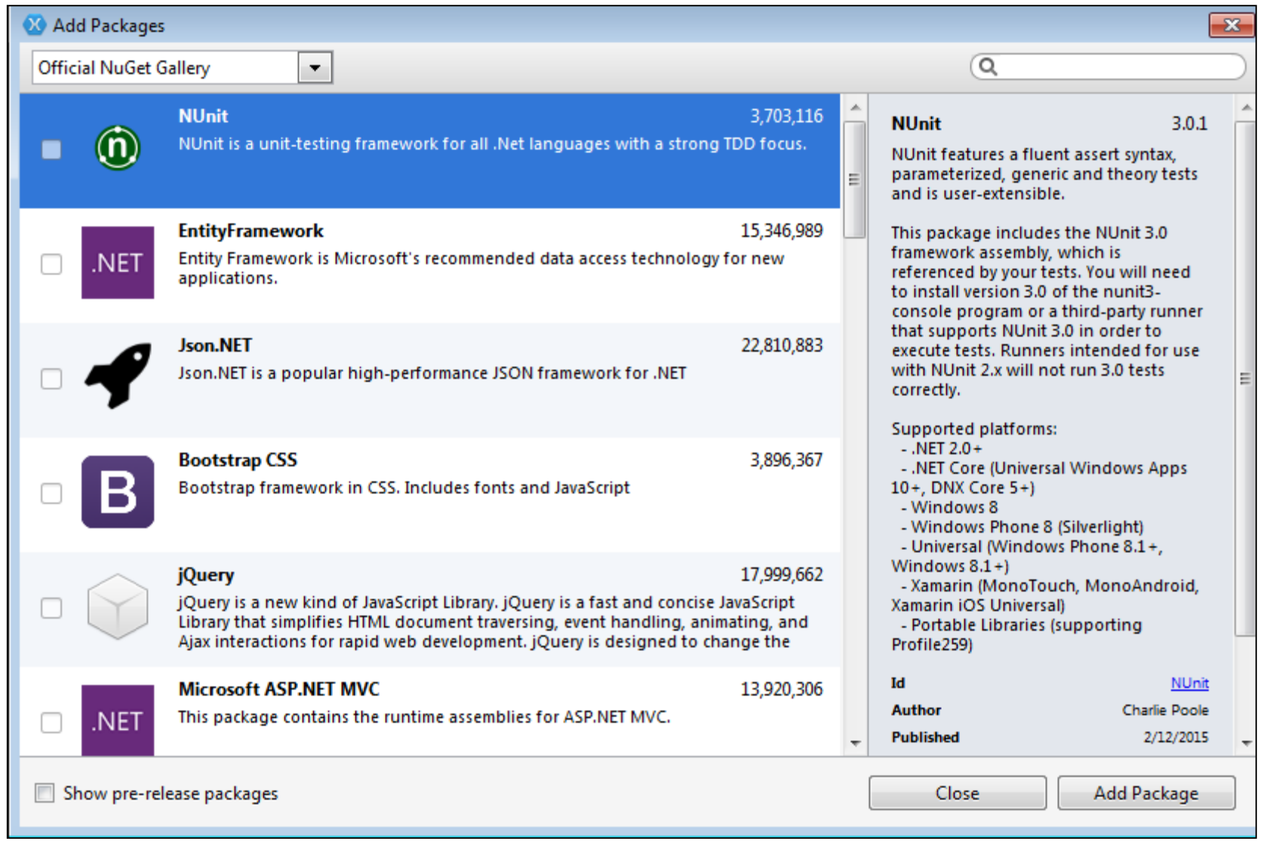
1) To test the calPerimeter Method

2) To test the calArea Method

Adding the NUnit Package

1. Right click your project, choose “Add”, and choose “Add NUGet packages…”
2. Select “Nunit” and then click Add Package.





Now, you can start to create the Test class!

Remember to include **`using Nunit.Framework;`**
as one of your directives at the beginning of the class.

```
1  using System;
2  using NUnit.Framework;
3
4
5  namespace Lecture02_Example01
6  {
7      [TestFixture]
8      public class ReactangleTest
9      {
10         [Test]
11         public void TestPerimeter ()
12         {
13             Rectangle rect1 = new Rectangle ();
14             rect1.Width = 5;
15             rect1.Height = 4;
16             Assert.AreEqual (18, rect1.calPerimeter ());
17         }
18
19         [Test]
20         public void TestArea(){
21             Rectangle rect2 = new Rectangle();
22             rect2.Width = 5;
23             rect2.Height = 4;
24             Assert.AreEqual (20, rect2.calArea ());
25         }
26     }
27 }
28
```

Indicates that this class contains test cases

Indicates a unit test begins

Indicates a unit test begins

Check whether the actual value is equivalent with the expected value.

`Assert.AreEqual(actual, expected);`

Checking for conditions:

- `Assert.IsTrue();`
- `Assert.IsFalse();`

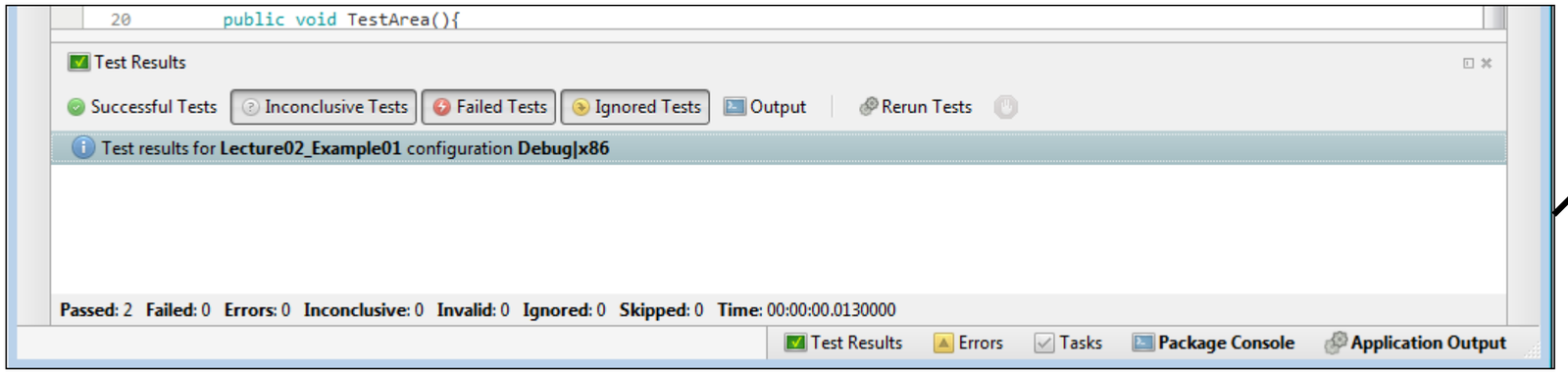
Additional Info on Assertion

An **assertion** (to assert something) means that the statement must be true, otherwise the test **fails**.

Nunit provides a number of classes that can help you make assertions in your unit test.

- The **Assert** class provides basic abilities to test if things are true, false, the same, etc.
  - `Assert.IsTrue(), Assert.IsFalse(), …`
- **StringAssert** provides methods that make it easier to make assertions about strings.
  - `StringAssert.AreEqualIgnoringCase(expected,actual), …`
- **CollectionAssert** makes it easier to test arrays and other collections of objects.
  - `CollectionAssert.Contains(collection, actual_object), …`

Let's run the unit tests!



Now, we can move on to create the main program after the unit tests are success!

In the main program, create a Rectangle object called rect3 and request user input for the width and height and display the rectangle details (width, height, perimeter & area).

```
class MainClass
{
    public static void Main (string[] args)
    {
        Rectangle rect3 = new Rectangle ();
        Console.WriteLine ("Enter the width:");
        rect3.Width = int.Parse(Console.ReadLine ());
        Console.WriteLine ("Enter the height:");
        rect3.Height = int.Parse(Console.ReadLine ());

        Console.WriteLine ("\nRectangle Details");
        Console.WriteLine ("==================");
        Console.WriteLine ("Width: " + rect3.Width);
        Console.WriteLine("Height:" + rect3.Height);
        Console.WriteLine ("Perimeter: " + rect3.calPerimeter ());
        Console.WriteLine ("Area: " + rect3.calArea ());
        Console.ReadLine ();

    }
}
```

Creating rect3 object from Rectangle class

Accepting user input for width and height using Console
`int.Parse()` is to convert the input string into integer

Display the rect3 object details accordingly

Program Output

```
Enter the width:
5
Enter the height:
4

Rectangle Details
==================
Width: 5
Height:4
Perimeter: 18
Area: 20
```

So, Test Driven Development

Any Questions?

This Week's Tasks

Pass Task 5 - Shape Drawer

Pass Task 6 - Unit Testing Shape

Pass Task 7 - Unit Testing the BankAccount (Assessed Task)

Pass Task 8 - Documenting the BankAccount Class (Assessed Task)