Testing and Test Driven Development

Smoke and Mirrors Prototype

At this point in the module you should have a good idea of what the system is going to do from the point of view of the user.

We have …

Event tables
Use case diagrams / descriptions
An early prototype of the proposed system

At this stage the prototype is smoke and mirrors in that it provides the illusion of functionality without implementing any of it.

Visual Studio Configuration

We should also have Visual Studio set up in such a way that we have a main solution file for our work. Inside the solution file we have sub projects…

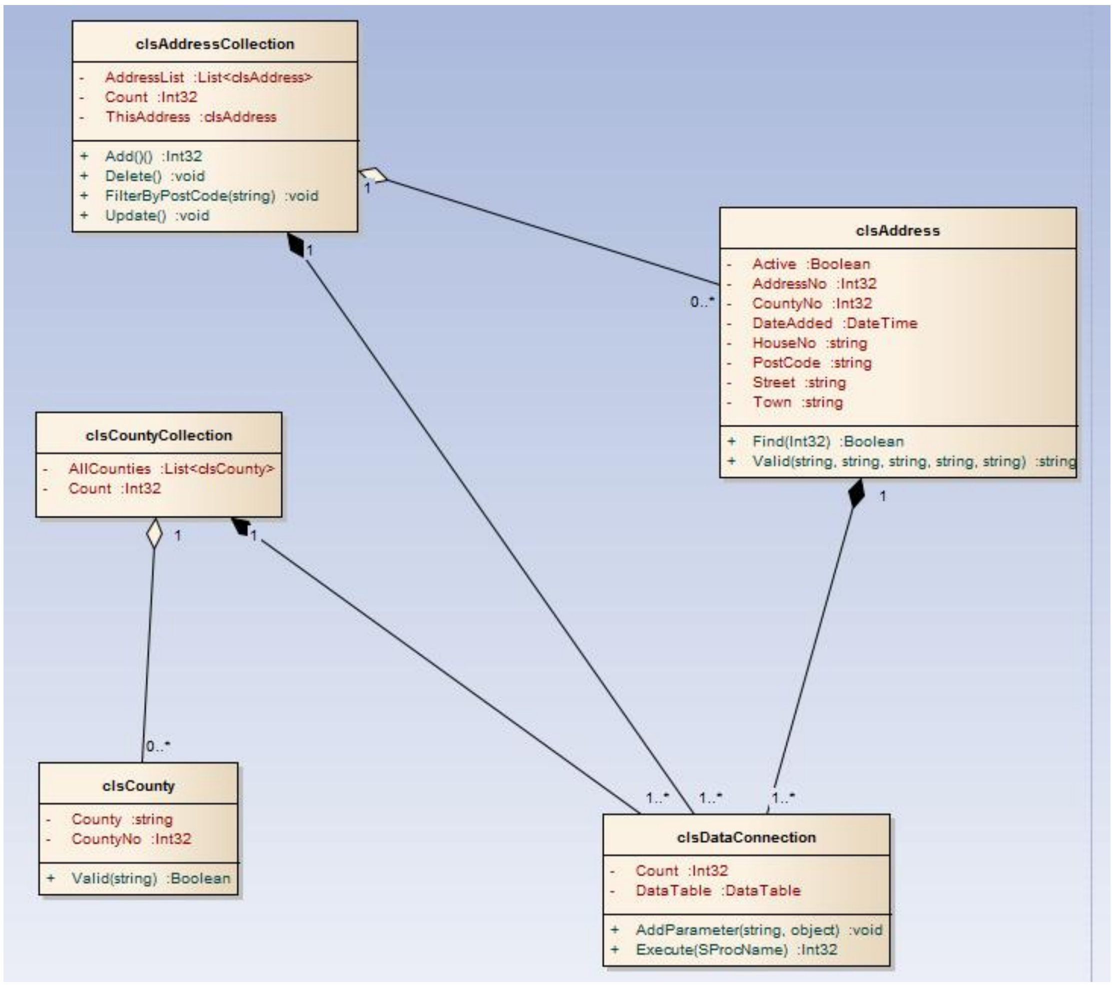
- Project bank web facing front end
- Project bank locally hosted back end
- The class library
- A database file linked to both presentation layers

Other Requirements

Before we get to the stage of writing code we need two more artefacts before continuing…

- A class diagram
- Test plans

Below is the class diagram for the address book which we shall use to get started in test driven development.



The class diagram represents the architectural plans for the house we are about to build.

Like building a house we don't simply start by laying bricks in any haphazard fashion. We need people with the correct skills in place building in such a way that the house will not fall-down shortly after completion.

Test Driven Development

Test Driven Development (TDD) is an approach to software construction which starts with the test for a feature before the feature even exists.

The rules are…

- Create a test that fails
- Run the test and see it fail
- Fix the error in a simplistic way
- See the test pass
- Re-factor to implement the feature correctly

In TDD rather than starting to write the system without any plan or strategy we take small steps which when added together eventually build in to the finished system.

As we create our tests we will create test cases in our code. Every time we run the system all of the test cases are run against our code and we may be reasonably sure that our system is robust due to the constant application of the test framework.

If at some point we get something wrong the test framework will tell us that a test has failed and it should give us a clear indication as to what test has failed and where in the system the problem is.

TDD is a good way of not only building confidence in the systems we create it will also help to build confidence in our own ability as developers.

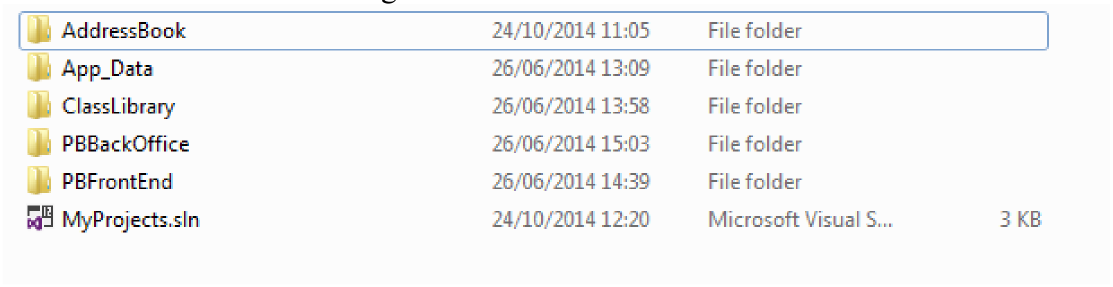
TDD forces us to take lots of small steps in developing the system and small steps are much easier to copy with than making fewer large steps.

Getting Started in TDD

The first step in getting started is to have Visual Studio open and configured correctly such that we are using solutions and projects.

There is a fully configured version available for download from the module web site so that you may work through this example.

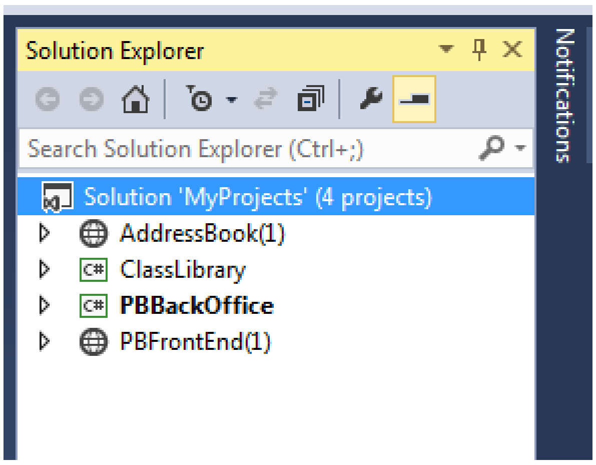
The file contains the following…



AddressBook	24/10/2014 11:05	File folder	
App_Data	26/06/2014 13:09	File folder	
ClassLibrary	26/06/2014 13:58	File folder	
PBBackOffice	26/06/2014 15:03	File folder	
PBFrontEnd	26/06/2014 14:39	File folder	
MyProjects.sln	24/10/2014 12:20	Microsoft Visual S...	3 KB

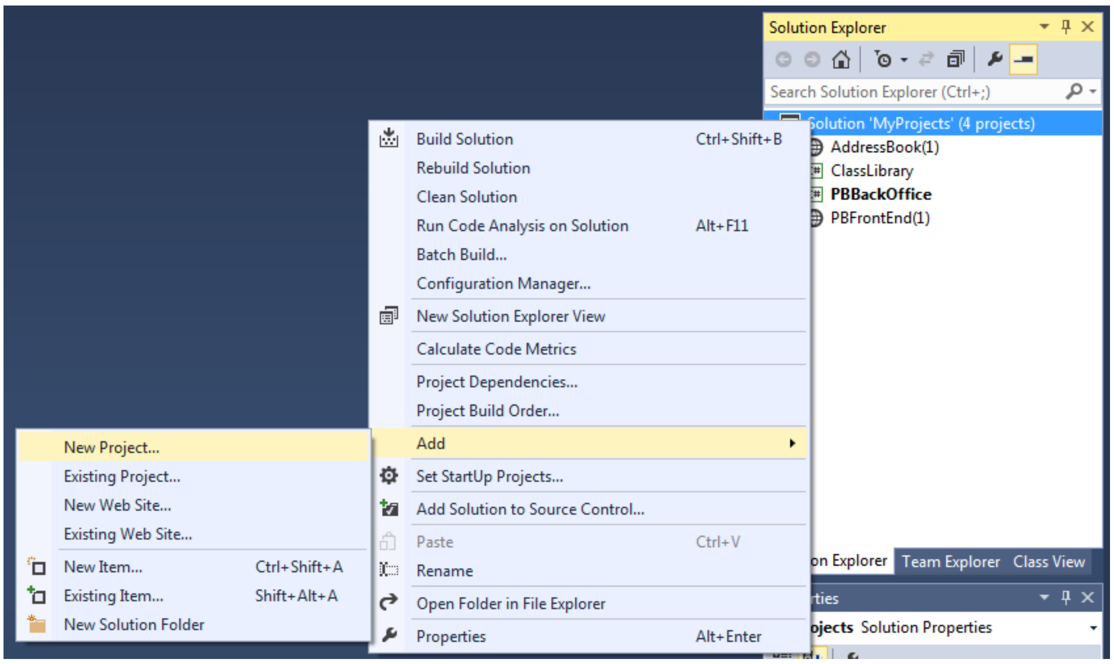
- A smoke and mirrors prototype for the Address Book
- An App_Data folder containing a basic database
- The class library
- A version of the Project Bank back end
- A version of the Project Bank front end

In Visual Studio this translates to …

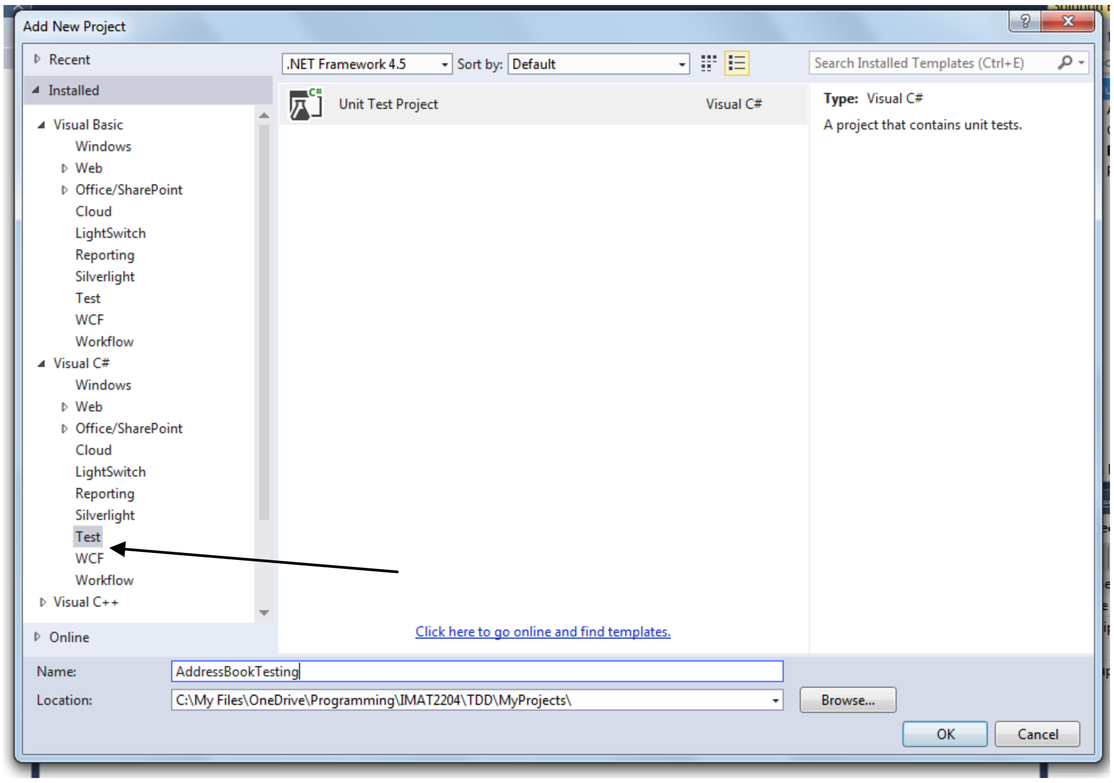


Now that we have the basic components of our system ready we need to create a test project. It is from this test project that we will generate the classes on the class diagram.

Right click on the solution and select Add – New Project…



Select the language as C# and locate the template for Test projects…


Add New Project

Recent
Installed
Visual Basic
Windows
Web
Office/SharePoint
Cloud
LightSwitch
Reporting
Silverlight
Test
WCF
Workflow
Visual C#
Windows
Web
Office/SharePoint
Cloud
LightSwitch
Reporting
Silverlight
Test
WCF
Workflow
Visual C++
Online

.NET Framework 4.5 Sort by: Default

Unit Test Project Visual C#

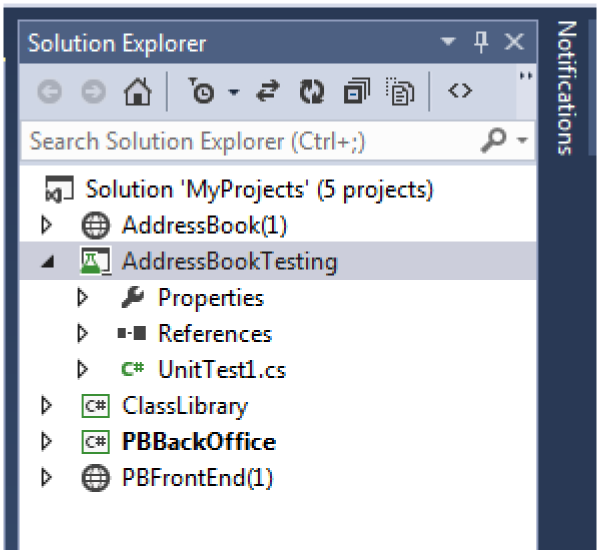
Type: Visual C#
A project that contains unit tests.

Click here to go online and find templates.

Name: AddressBookTesting
Location: C:\My Files\OneDrive\Programming\IMAT2204\TDD\MyProjects\

Set the name of the project to AddressBookTesting and the location to the correct folder for the location of your solution.

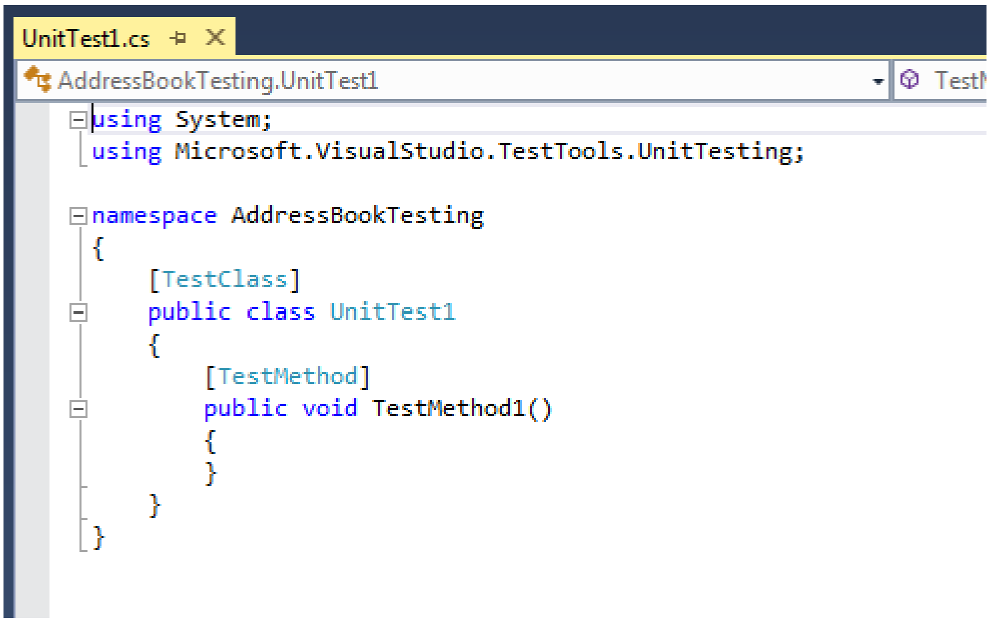
This should create the test project within the solution…


Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'MyProjects' (5 projects)
  AddressBook(1)
  AddressBookTesting
    Properties
    References
    UnitTest1.cs
  ClassLibrary
  PBBackOffice
  PBFrontEnd(1)
```

It will also create a default test class…



```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace AddressBookTesting
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

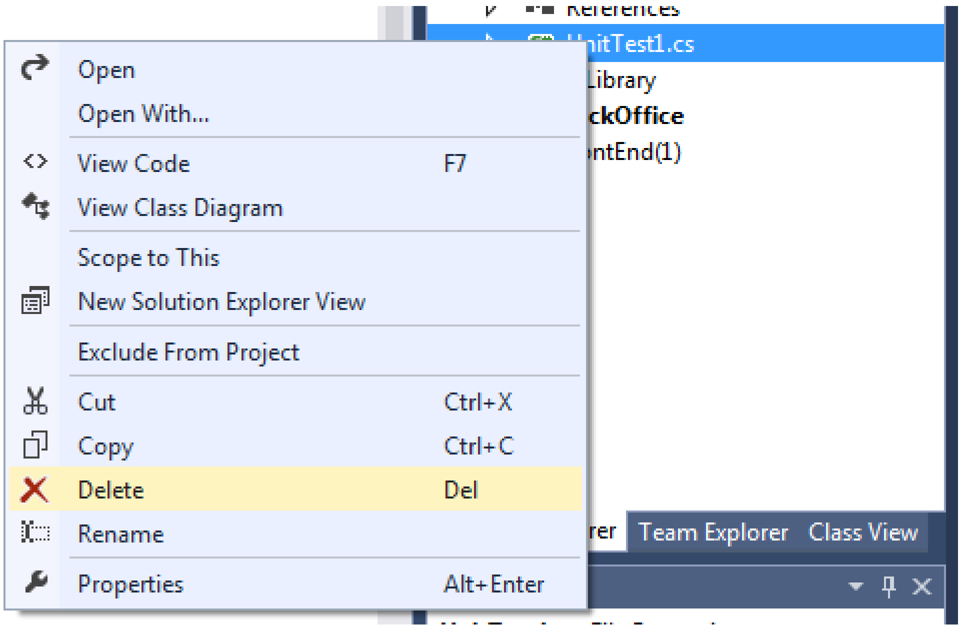
In a moment, we will delete this file and create our own but it’s worth spending a little while looking at the structure of the test class.

Each test class we create must have the text [TestClass] present to indicate to Visual Studio that this is what it is (Visual Studio will mostly do this for you!)

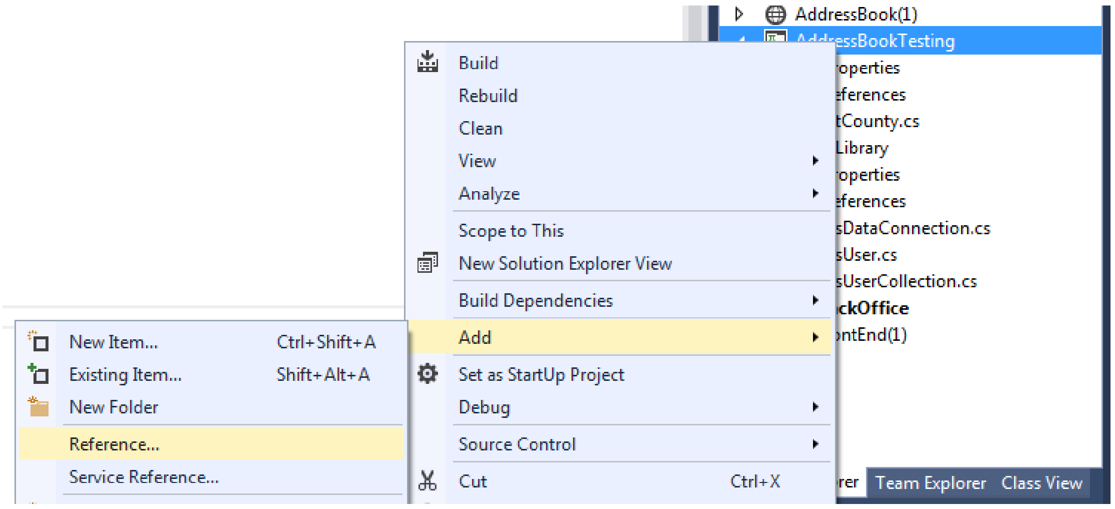
Each test we create will be set up as a function. Each function will test a single aspect of the class in question.

Each function needs to be tagged as [TestMethod] Visual Studio won’t do this for you so you will need to remember to type it otherwise your test will be ignored.

Delete the default test class by right clicking it in the solution explorer…

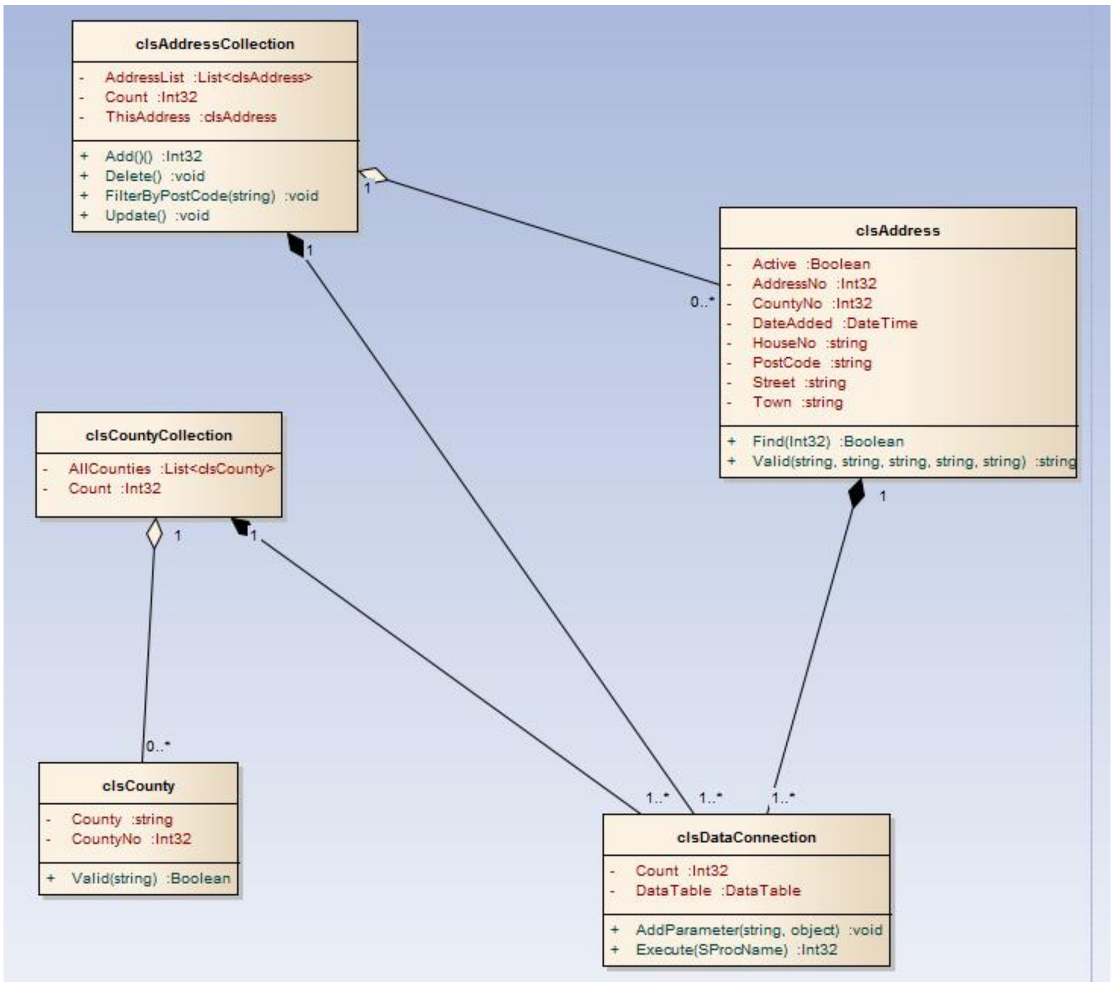


The last step is to link in the class library to the test project since we want our classes to be created in the class library not in the test project.

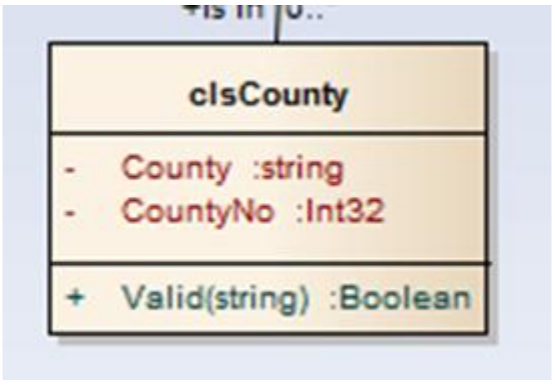


Now we will create our first test class so where to start?

The first place to start is the class diagram. We need to decide which class we want to create first…



There are no hard and fast rules at this point but starting with something simple is a good guide. That being the case we shall start with the most basic class on the diagram clsCounty.

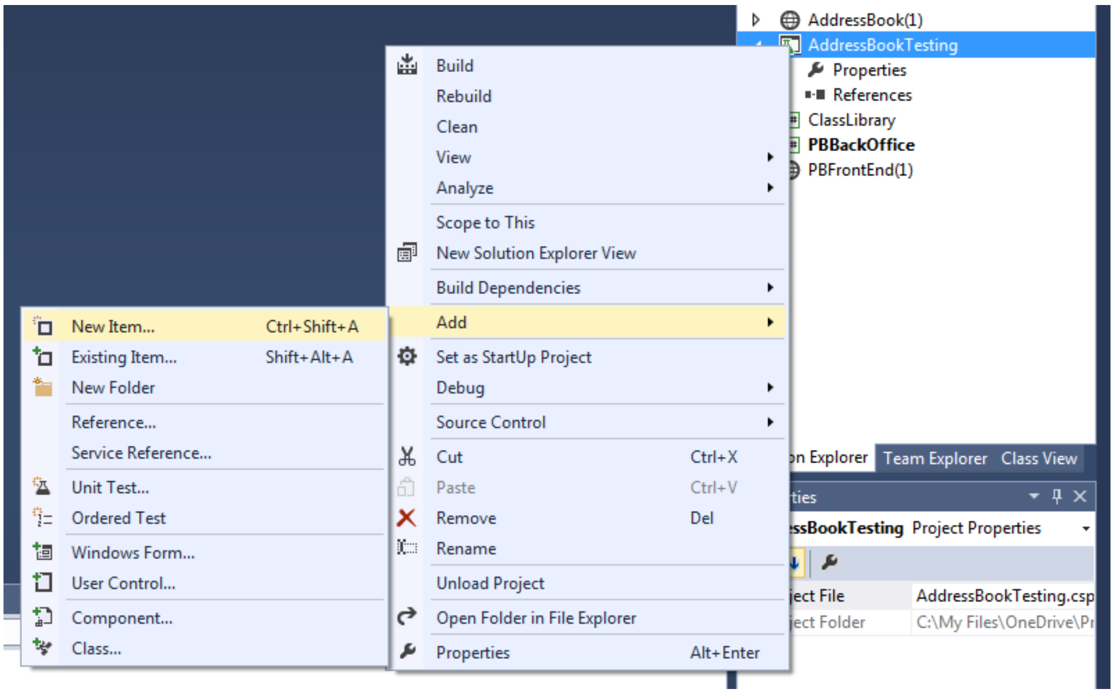


It has one operation and only two attributes.

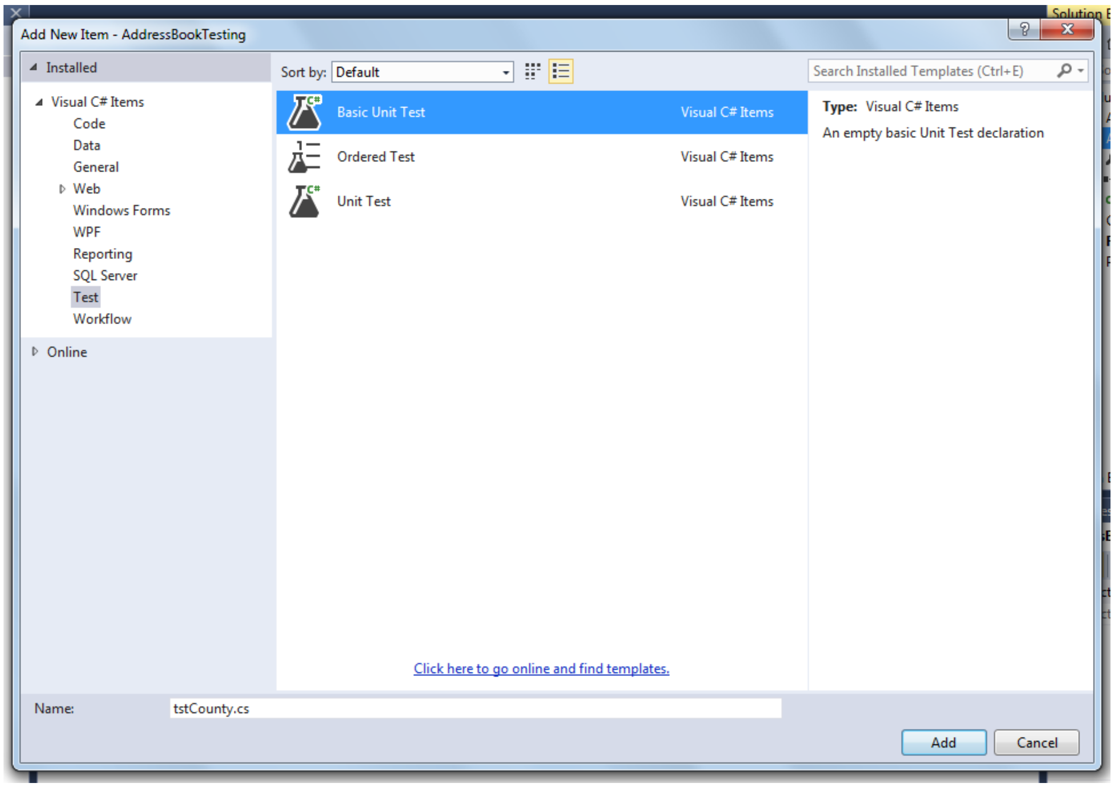
To create this class, we will create a corresponding test class called tstCounty.

Each class on the diagram will have its own test class and we will differentiate the test class from the actual class using the prefix tst.

Right click on the test project and select add – new item



Locate unit test under the available items for C#...


Add New Item - AddressBookTesting
Installed
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
SQL Server
Test
Workflow
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Basic Unit Test	Visual C# Items
Ordered Test	Visual C# Items
Unit Test	Visual C# Items
Click here to go online and find templates.
Add
Cancel

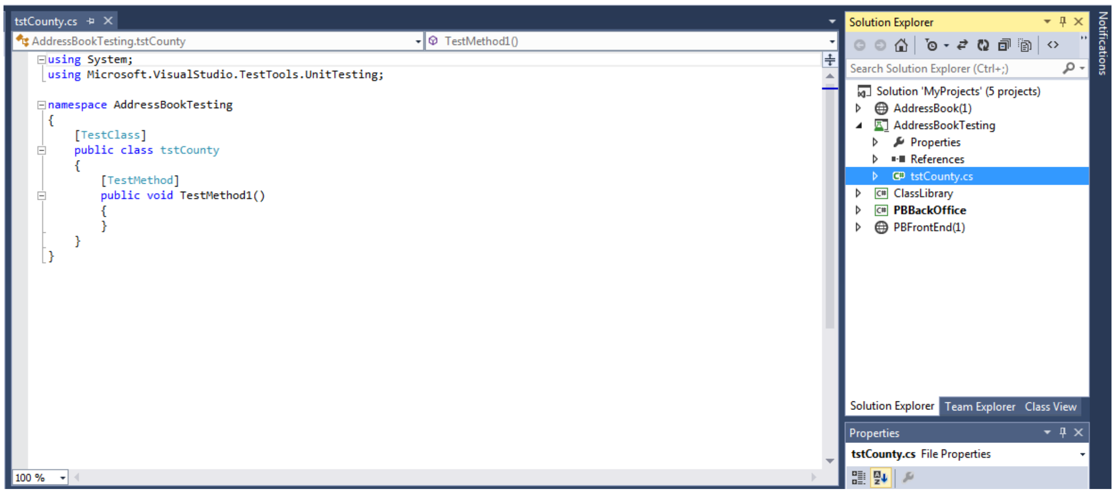
Type: Visual C# Items

An empty basic Unit Test declaration

Name: tstCounty.cs

Set the name of the test class to tstCounty and press add.

This will create the test class in your project with the default code in the main window…



```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace AddressBookTesting
{
    [TestClass]
    public class tstCounty
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

The First Test

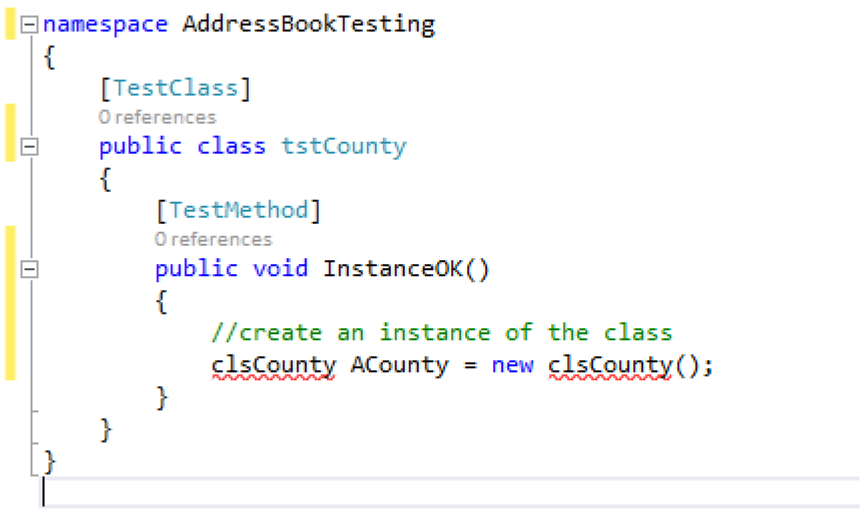
Where do we start then? Remember before we write any code in our class we need to create a test that fails.

What then is the first test?

Pretty much every time in TDD the first test is to try and use the class before it even exists.

We will create a test called InstantiationOk which tests to see if we may create an object based on the class definition.

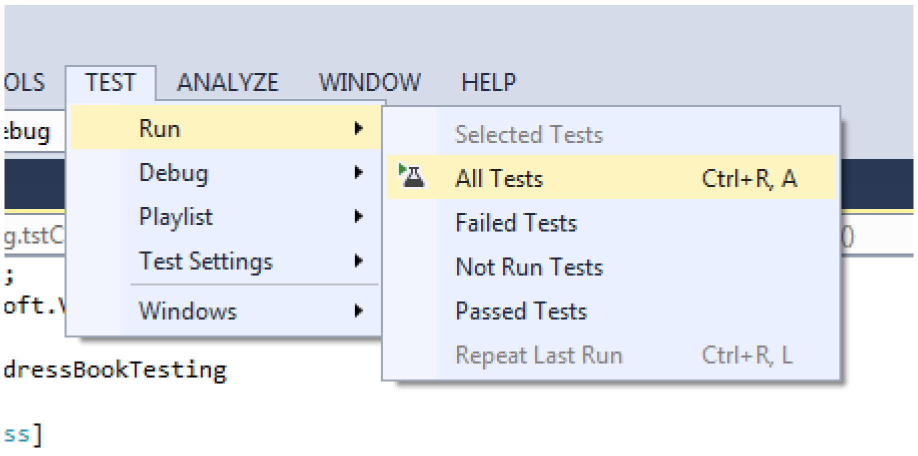
Modify the default code so that we have a test case to test this…



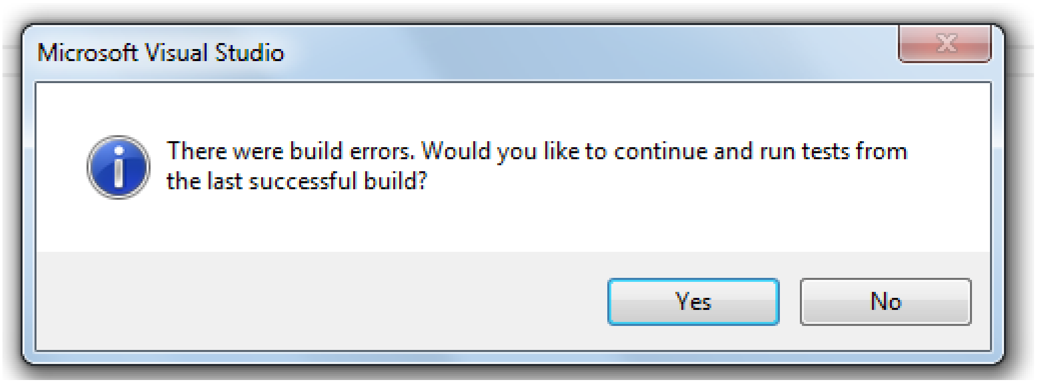
```
namespace AddressBookTesting
{
    [TestClass]
    0 references
    public class tstCounty
    {
        [TestMethod]
        0 references
        public void InstanceOK()
        {
            //create an instance of the class
            clsCounty ACounty = new clsCounty();
        }
    }
}
```

We should be able to see even at this point the test is going to fail since we have yet to create the class clsCounty!

To run our tests, select Test – Run – All Tests



As expected the test fails…


Microsoft Visual Studio

There were build errors. Would you like to continue and run tests from the last successful build?

Press no.

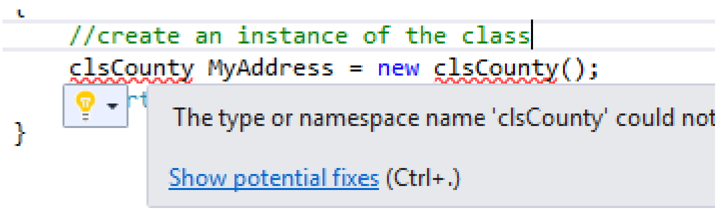
This is the starting point of TDD we always start with a test that fails.

So, what next?

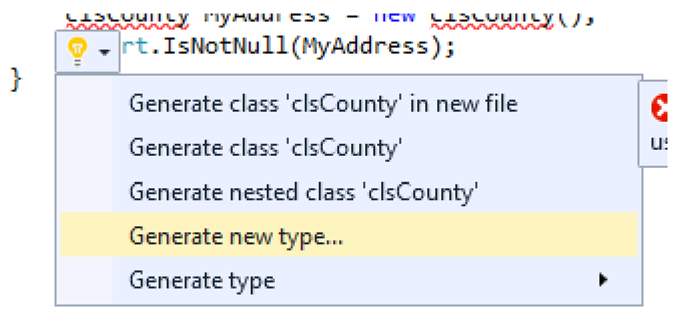
Now we fix the problem.

It is important to appreciate that this fix may not be an elegant solution. We just want to see the test pass by any means possible.

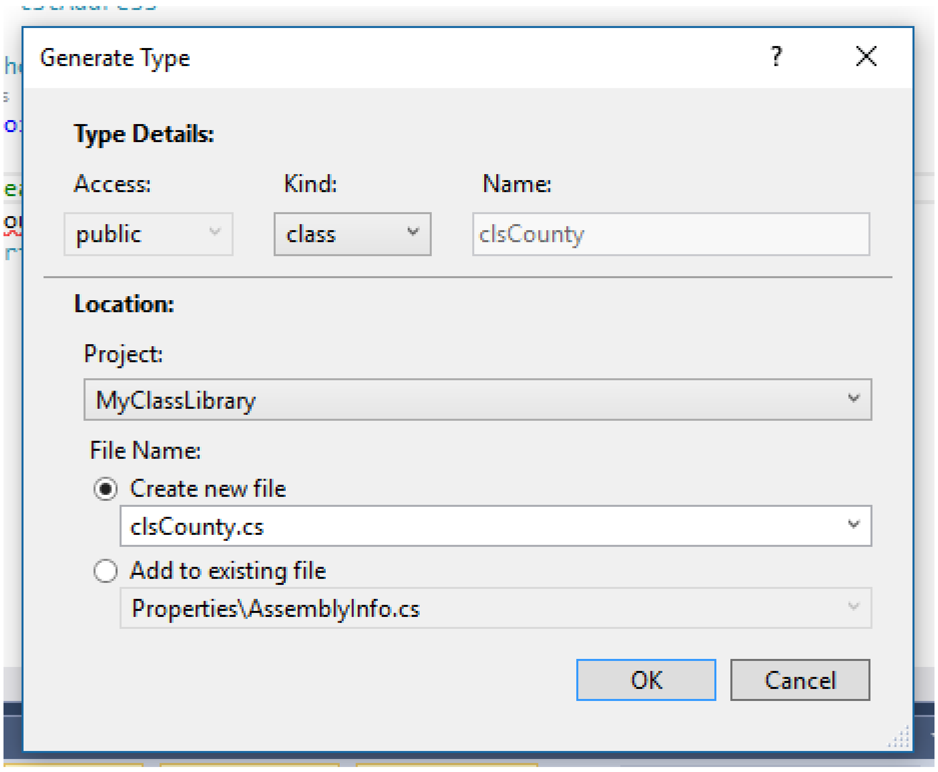
Hold the mouse over the red underlining and click show potential fixes…


```
    //create an instance of the class
    clsCounty MyAddress = new clsCounty();
}
```
The type or namespace name 'clsCounty' could not

Then select generate new type…


```
    rt.IsNotNull(MyAddress);
}
```
Generate class 'clsCounty' in new file
Generate class 'clsCounty'
Generate nested class 'clsCounty'
Generate new type...
Generate type

This will allow us to create the class file we want to use…


Generate Type

Type Details:

Access: public

Kind: class

Name: clsCounty

Location:

Project: MyClassLibrary

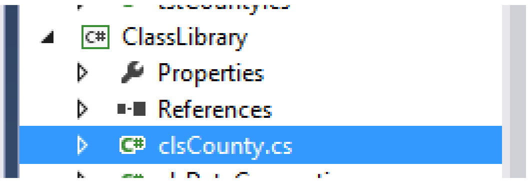
File Name:

(●) Create new file: clsCounty.cs

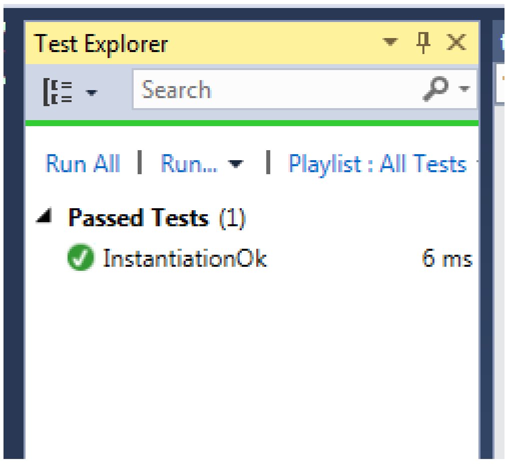
( ) Add to existing file: Properties\AssemblyInfo.cs

Mostly you may leave the default setting as they are however the one setting to watch is the name of the project the class will be created in and the name of the class file.

Press OK and the class file should be created in the class library.


ClassLibrary
Properties
References
clsCounty.cs

Now run all tests again and it should pass…


Test Explorer

Passed Tests (1)

InstantiationOk 6 ms

The test explorer to the left of the screen will always tell us if a test has gone wrong.

It is also important to note what we have done here.

TDD is often about making small steps.

Creating a test for the presence of a class watching it fail then fixing the problem is a very small but important step.

Since this test is going to be run every time we run our tests we can be reasonably confident that the name of our class is correct and that it exists. If either point changes, the test framework will tell us.

Our steps may not be always this small but the fact that we can make small steps makes code development much more manageable.

Testing the Properties

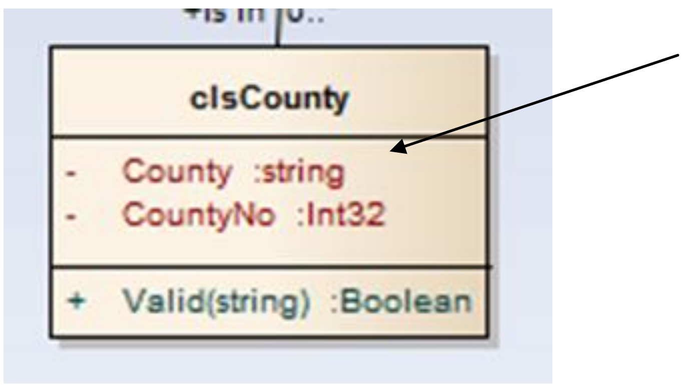
The next step in TDD is creating the properties for the class.

As before we will start with a test that fails, watch it fail and then fix the problem.

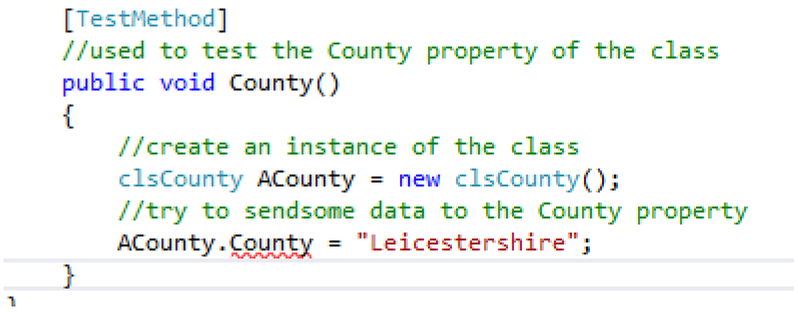
Create the following test method…

```
[TestMethod]
//used to test the County property of the class
public void County()
{

}
```

We will now test the County attribute from the class diagram…



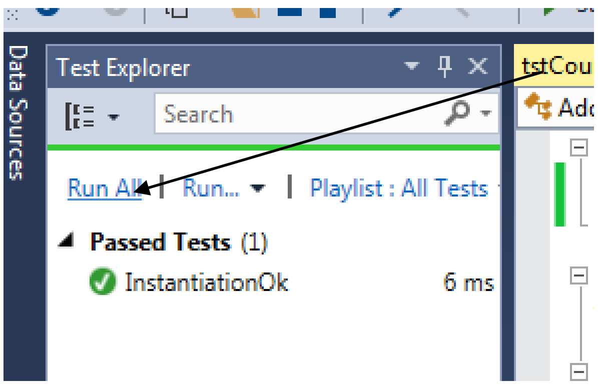
Create the test like so…



```
[TestMethod]
//used to test the County property of the class
public void County()
{
    //create an instance of the class
    clsCounty ACounty = new clsCounty();
    //try to sendsome data to the County property
    ACounty.County = "Leicestershire";
}
```

We should be able to see instantly that there is going to be a problem.

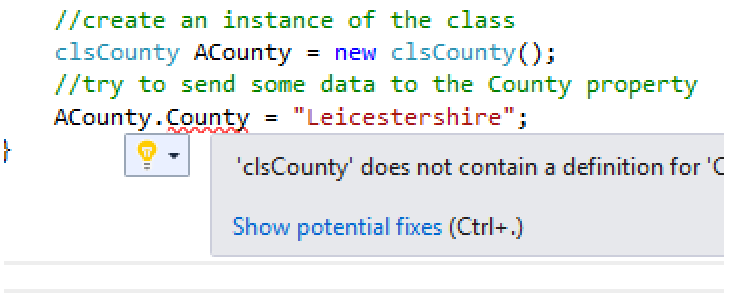
Run all tests from the test explorer…



Watch the test fail!

Now we fix the failing test.

Hold the mouse over the property underlined in red…

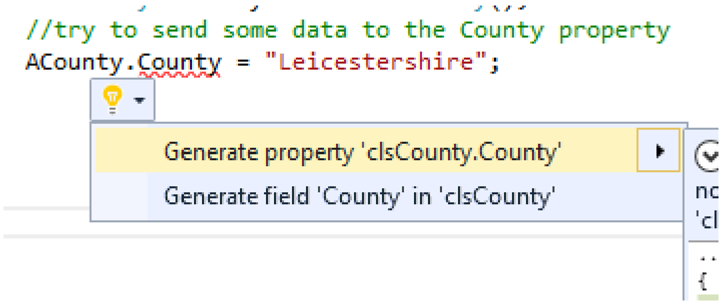


```
    //create an instance of the class
    clsCounty ACounty = new clsCounty();
    //try to send some data to the County property
    ACounty.County = "Leicestershire";
}
```

'clsCounty' does not contain a definition for 'C

Show potential fixes (Ctrl+.)

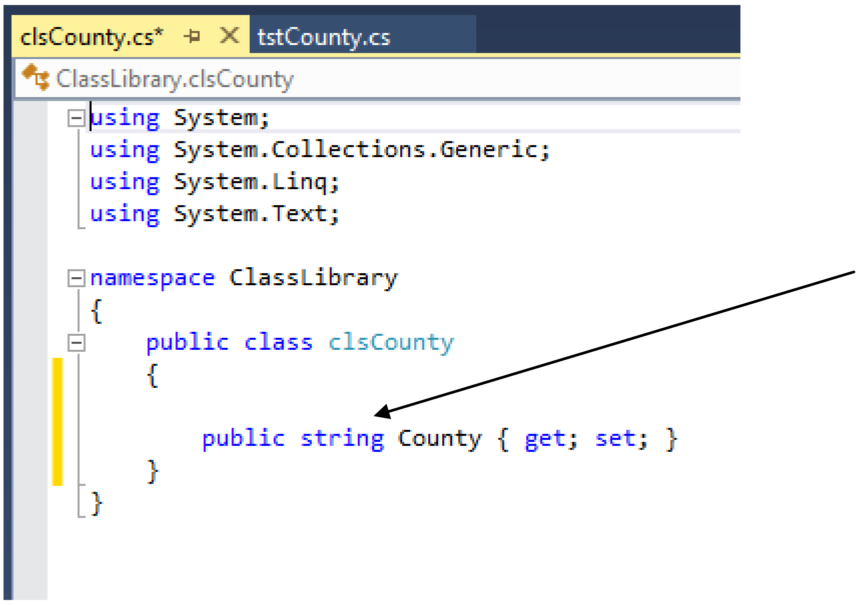
Select show potential fixes followed by generate property…



```
//try to send some data to the County property
ACounty.County = "Leicestershire";
```

Generate property 'clsCounty.County'
Generate field 'County' in 'clsCounty'

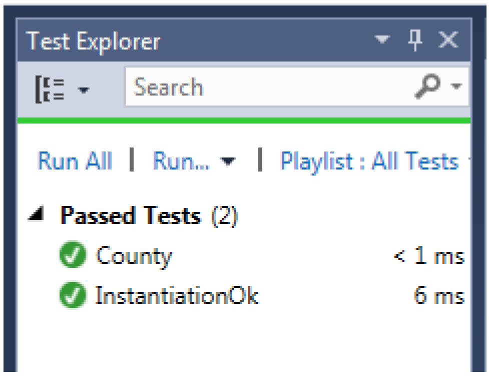
Examine the code for clsCounty and you will see that the property has been created for you…



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ClassLibrary
{
    public class clsCounty
    {
        public string County { get; set; }
    }
}
```

It has inspected how you are using the property and come up with the correct data type for the property.

Run all tests again and you should see that our two tests are green…



```
Passed Tests (2)
  County              < 1 ms
  InstantiationOk       6 ms
```

The next step with this test (and all tests really) is to re-factor.

Refactoring requires us to think more deeply about the test and make sure that it is a thorough as possible.

In the case of testing a property like this it has both a getter and a setter. Meaning that it is a read write property.

We need to come up with a test that examines both these aspects.

The following code will do this…

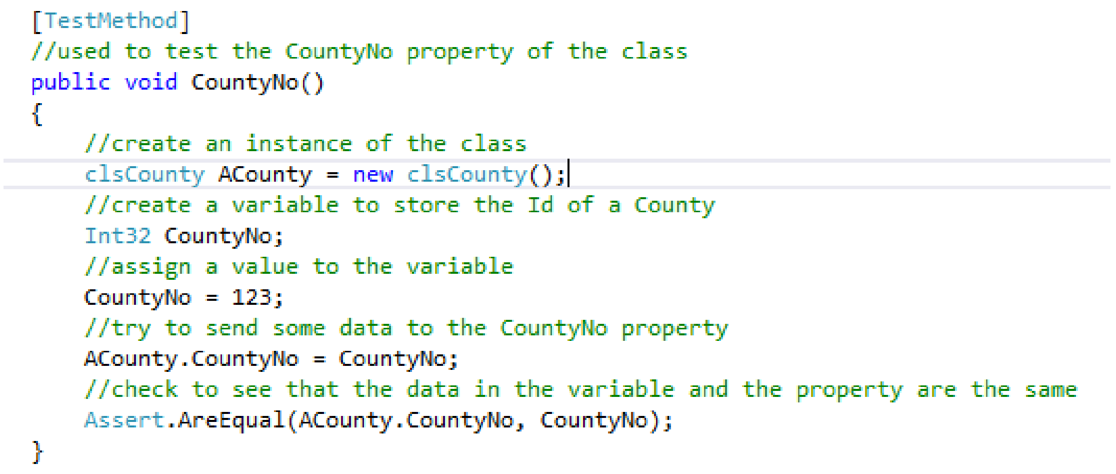
```
[TestMethod]
//used to test the County property of the class
public void County()
{
    //create an instance of the class
    clsCounty ACounty = new clsCounty();
    //create a variable to store the name of a county
    string SomeCounty;
    //assign a county to the variable
    SomeCounty = "Leicestershire";
    //try to send some data to the County property
    ACounty.County = SomeCounty;
    //check to see that the data in the variable and the property are the same
    Assert.AreEqual(ACounty.County, SomeCounty);
}
```

Modify the test case and make sure that it still passes the test.

Creating the second property (CountyNo) is a similar procedure as for County.

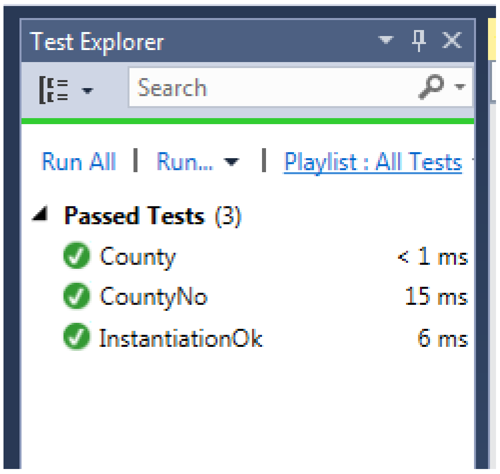
- Test the presence of the property
- Test both read and write aspects of the property

You should end up with a test case like this…



```
[TestMethod]
//used to test the CountyNo property of the class
public void CountyNo()
{
    //create an instance of the class
    clsCounty ACounty = new clsCounty();
    //create a variable to store the Id of a County
    Int32 CountyNo;
    //assign a value to the variable
    CountyNo = 123;
    //try to send some data to the CountyNo property
    ACounty.CountyNo = CountyNo;
    //check to see that the data in the variable and the property are the same
    Assert.AreEqual(ACounty.CountyNo, CountyNo);
}
```

Which should produce the following results…



Passed Tests (3)

County	< 1 ms
CountyNo	15 ms
InstantiationOk	6 ms

Applying the Test Plan

One document we looked at last year in IMAT1604 was the test plan.

Test plans are important documents when it comes to TDD as it is from these that we will generate many of our test cases.

There is a big problem with test plans.

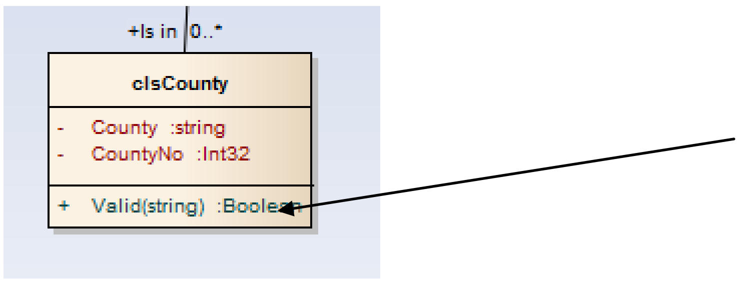
They are boring to create and even more tedious to enter manually.

This means that testing is often carried out at a very surface level if at all.

TDD gets past this by automating the test process. What we need to do as developers is create the test plans and then write test functions that automate the whole process.

A very good place to illustrate this is to create a validation method for the class clsCounty.

We will now implement the Valid method for the class.



This method will test the name of the county to make sure that it conforms to the storage requirements. It will return true or false based on if the data is correct or not.

We will make the following assumptions about the county name.

- It may not be left blank i.e. more than zero characters
- In the database, it will have a maximum field size of 20 characters

It is important to appreciate that we may state these limitations even in the absence of a table in the database!

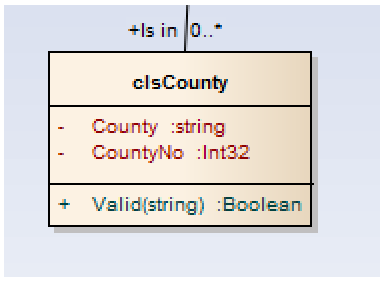
This should give us test cases along the following lines…

Nature of test	Test data	Expected result
Extreme min	NA	NA
Min less one	0 characters	Fail
Min	1 character	Success
Min plus one	2 characters	Success
Mid	10 characters	Success
Max	20 characters	Success
Max less one	19 characters	Success
Max plus one	21 characters	Fail
Extreme max	500 characters	Fail

Translating the Tests into TDD

As before we shall start simple.

From the class diagram, we need to test to see if our test method actually exists…

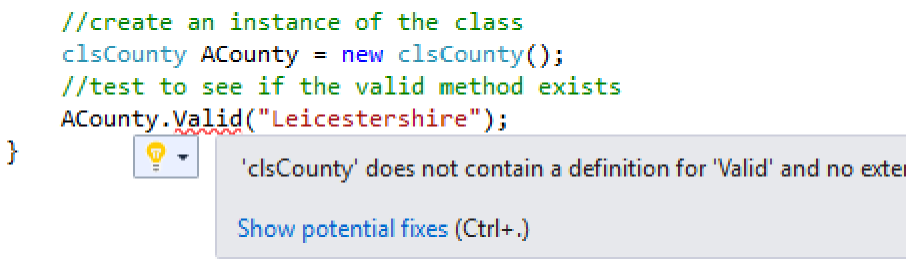


The test to do this will look something like this…

```
[TestMethod]
//used to test the presence of the Valid Method
public void Valid()
{
    //create an instance of the class
    clsCounty ACounty = new clsCounty();
    //test to see if the valid method exists
    ACounty.Valid("Leicestershire");
}
```

As we should expect the test will fail.

Hold the mouse over the method…

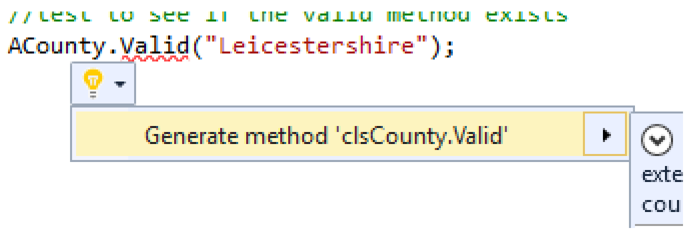


```
        //create an instance of the class
        clsCounty ACounty = new clsCounty();
        //test to see if the valid method exists
        ACounty.Valid("Leicestershire");
    }
```

'clsCounty' does not contain a definition for 'Valid' and no exte

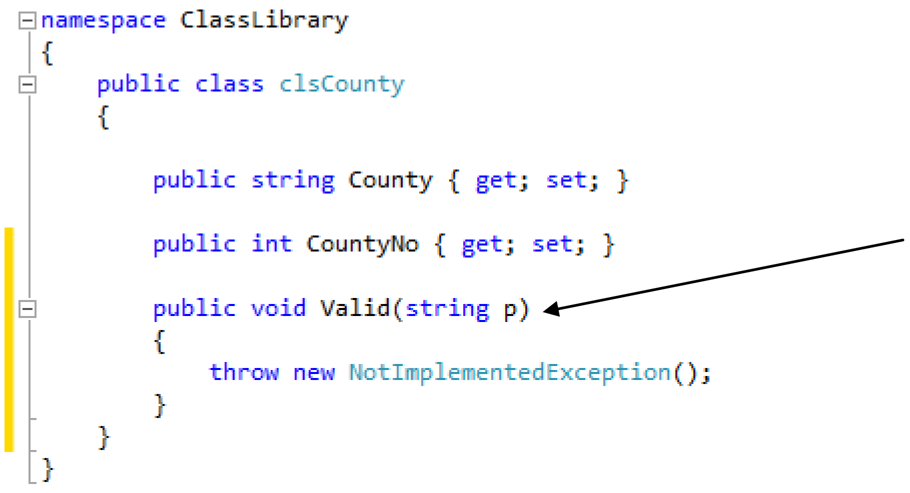
Show potential fixes (Ctrl+.)

Select show potential fixes then generate method



```
//test to see if the valid method exists
ACounty.Valid("Leicestershire");
```

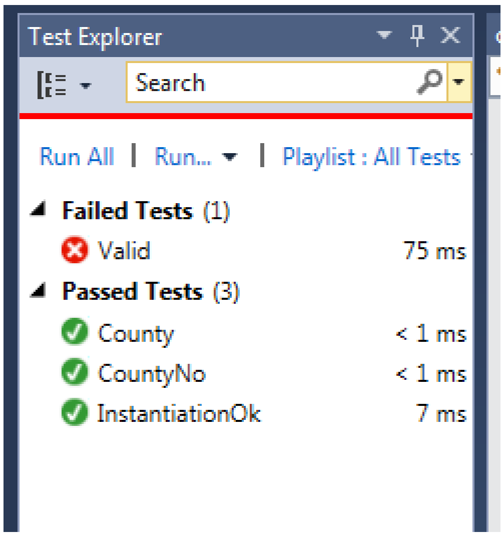
This will create the function in the class for your method…



```
namespace ClassLibrary
{
    public class clsCounty
    {

        public string County { get; set; }

        public int CountyNo { get; set; }

        public void Valid(string p)
        {
            throw new NotImplementedException();
        }
    }
}
```

Run all tests to see what happens.

We should see a failed test…



```
Failed Tests (1)
    Valid                75 ms
Passed Tests (3)
    County               < 1 ms
    CountyNo             < 1 ms
    InstantiationOk      7 ms
```

So, what is the problem?

The problem is the auto generated code has some things that we need to adjust ourselves.

```
public void Valid(string p)
{
    throw new NotImplementedException();
}
```

Firstly, the parameter "p" isn't a terribly good name. Secondly the one line of code…

```
throw new NotImplementedException();
```

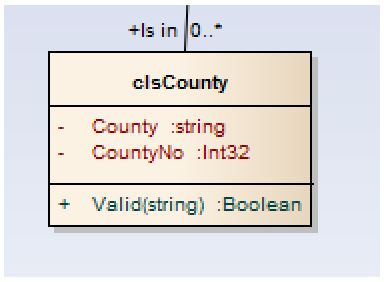
Will always result in the failure of this test.

Lastly the return data type of the function needs to be Boolean not void!

Modify the function like so…

```
public Boolean Valid(string County)
{
    return true;
}
```

This matches the definition in the class diagram…

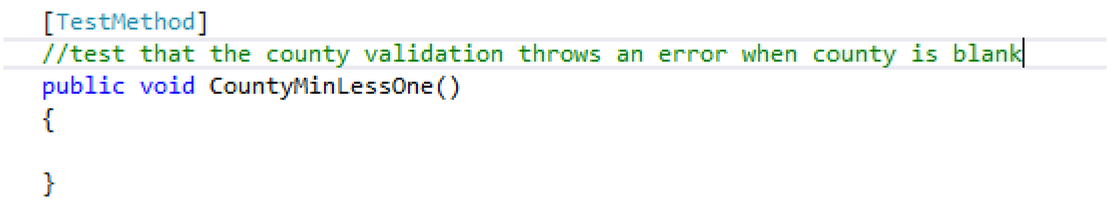


We should now be back in the green with our testing and ready to go onto the next step we are now able to apply the test plan.

The first test we will create is the minimum less one test

Nature of test	Test data	Expected result
Extreme min	NA	NA
*Min less one	0 characters	Fail*
Min	1 character	Success
Min plus one	2 characters	Success
Mid	10 characters	Success
Max	20 characters	Success
Max less one	19 characters	Success
Max plus one	21 characters	Fail
Extreme max	500 characters	Fail

In the test class create the following test function…

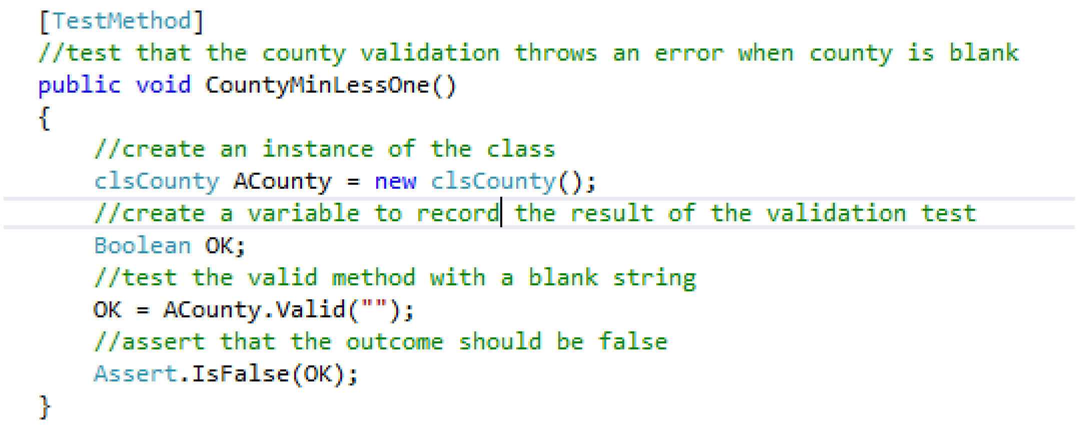


```
[TestMethod]
//test that the county validation throws an error when county is blank
public void CountyMinLessOne()
{

}
```

To write the test code for this we need to do the following.

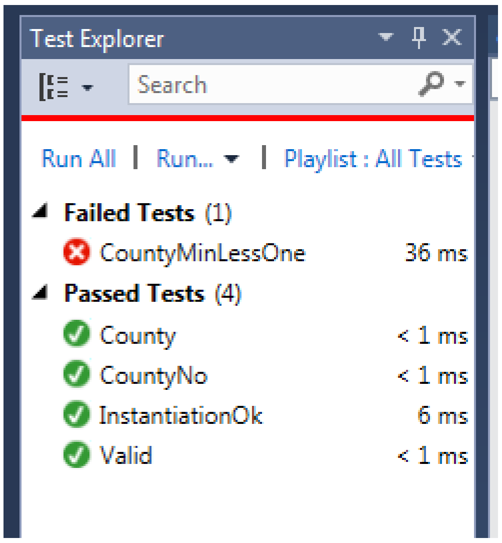
- Create an instance of the class
- Send a blank string to the Valid method
- Test to see that the valid method returns a false value

The following code should do this…



```
[TestMethod]
//test that the county validation throws an error when county is blank
public void CountyMinLessOne()
{
    //create an instance of the class
    clsCounty ACounty = new clsCounty();
    //create a variable to record the result of the validation test
    Boolean OK;
    //test the valid method with a blank string
    OK = ACounty.Valid("");
    //assert that the outcome should be false
    Assert.IsFalse(OK);
}
```

Run the test and it should fail.



```
Failed Tests (1)
    CountyMinLessOne        36 ms
Passed Tests (4)
    County                  < 1 ms
    CountyNo                < 1 ms
    InstantiationOk           6 ms
    Valid                   < 1 ms
```

The problem?

The valid function currently contains no code for such validation…

```
public Boolean Valid(string County)
{
    return true;
}
```

We need to fix this such that we get our test results into the green and out of the red!

To do this we will complete some of the code for the validation function so that it works correctly.

```
public Boolean Valid(string County)
{
    //var to record any errors found in County name assuming all is OK
    Boolean OK = true;
    //test to see if the county has zero characters
    if (County.Length == 0)
    {
        //set OK to false
        OK = false;
    }
    //return the results of all tests
    return OK;
}
```

Run the test again and we should be back into the green.

The next stage is to implement the rest of the test plan as a set of test functions.

The interesting thing about TDD is that we are really most interested in tests that fail from the start.

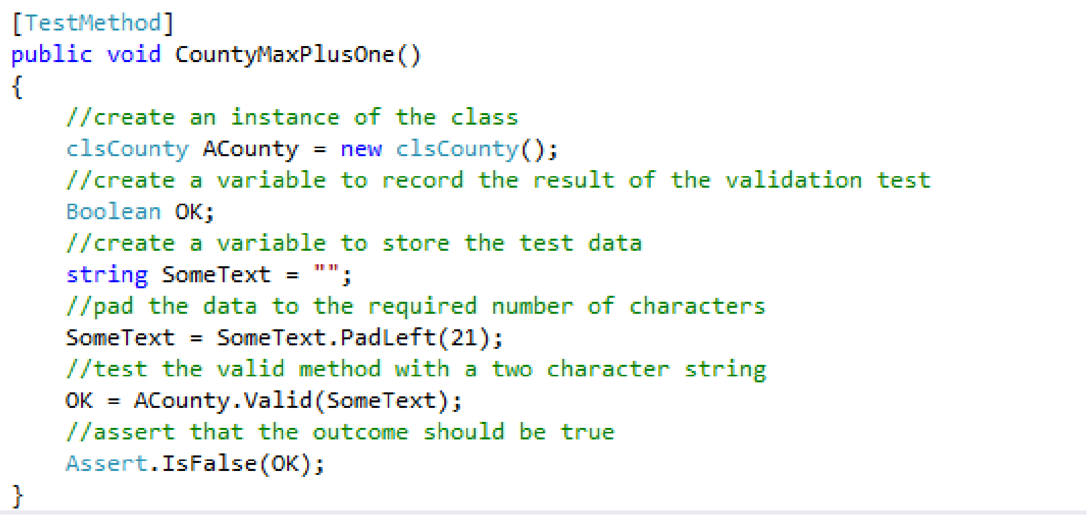
There is no point writing tests for the following test cases…

Nature of test	**Test data**	**Expected result**
Extreme min	NA	NA
Min less one	0 characters	Fail
*Min	1 character	Success*
*Min plus one	2 characters	Success*
*Mid	10 characters	Success*
*Max	20 characters	Success*
*Max less one	19 characters	Success*
Max plus one	21 characters	Fail
Extreme max	500 characters	Fail

So, let's concentrate on the tests that fail!

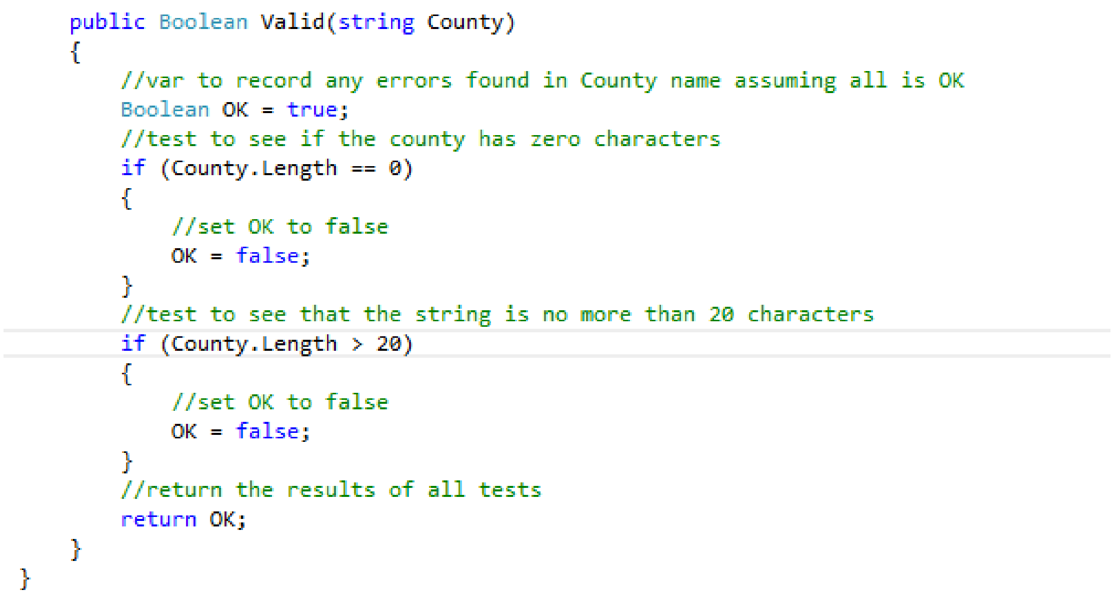
Below are the test functions for the remaining test cases. Remember to run each test at a time – be systematic with this.

Max plus one	21 characters	Fail



```
[TestMethod]
public void CountyMaxPlusOne()
{
    //create an instance of the class
    clsCounty ACounty = new clsCounty();
    //create a variable to record the result of the validation test
    Boolean OK;
    //create a variable to store the test data
    string SomeText = "";
    //pad the data to the required number of characters
    SomeText = SomeText.PadLeft(21);
    //test the valid method with a two character string
    OK = ACounty.Valid(SomeText);
    //assert that the outcome should be true
    Assert.IsFalse(OK);
}
```

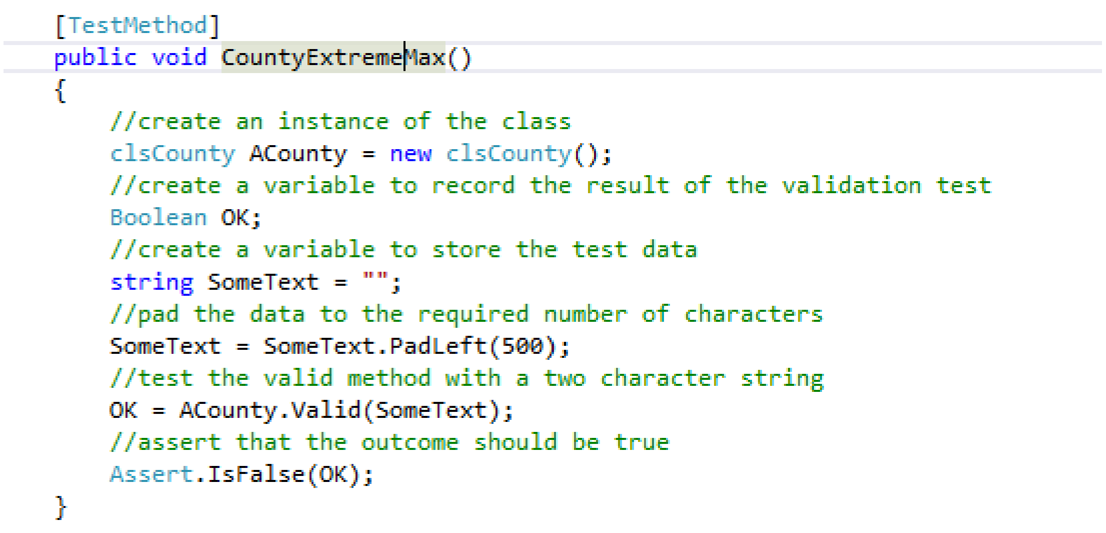
This test should fail. You will need to write suitable validation in the valid method to get past this stage like the following…



```
    public Boolean Valid(string County)
    {
        //var to record any errors found in County name assuming all is OK
        Boolean OK = true;
        //test to see if the county has zero characters
        if (County.Length == 0)
        {
            //set OK to false
            OK = false;
        }
        //test to see that the string is no more than 20 characters
        if (County.Length > 20)
        {
            //set OK to false
            OK = false;
        }
        //return the results of all tests
        return OK;
    }
}
```

Extreme max	500 characters	Fail

In this example, we could create a test case like this…



```
[TestMethod]
public void CountyExtremeMax()
{
    //create an instance of the class
    clsCounty ACounty = new clsCounty();
    //create a variable to record the result of the validation test
    Boolean OK;
    //create a variable to store the test data
    string SomeText = "";
    //pad the data to the required number of characters
    SomeText = SomeText.PadLeft(500);
    //test the valid method with a two character string
    OK = ACounty.Valid(SomeText);
    //assert that the outcome should be true
    Assert.IsFalse(OK);
}
```

Or we could ignore this test case based on the assumption that if we can spot a string of 21 characters we should be able to spot a string of 500!