

Programming Pathways

Visual Studio 2017

Matthew Dean 2019

Contents

Introduction.....	1
Do this first!	3
But what if I want to locate my database somewhere else?.....	4
So you have a Class Diagram!	5
Pathway 1 – Creating a Simple Class	7
Creating Properties.....	16
Creating Methods.....	19
Pathway 2 – Creating a Simple Data Bound Collection	33
Creating the Properties.....	35
Pathway 3 – Creating the Complex Item Class	57
Creating the Properties.....	60
Creating the Methods	64
Pathway 4 – Creating the Complex Collection Class	116
Creating the Properties.....	118
Creating the Methods	132
Creating the Add Method.....	132
Creating the Delete Method	139
Creating the Update Method	143
Creating the Post Code Filter	150

Introduction

There is a lot of work here to digest so make sure you allow enough time to process the information provided.

One of the big problems with learning to program is that it is an intensely practical skill.

We can understand the terms and concepts but ultimately we need to address the question “can you do it?”

This guide is designed to provide you with a set of strategies to get you started at creating code that does something.

This is not the end of the story on development it is rather more the start of it. Nor are the approaches detailed here the only way of writing code. As with most things in life there is more than one way to “skin a cat”.

This guide takes Test Driven Development (TDD) and uses it as a way of making a start on writing your system.

In TDD you start with small tests. Each test is then used to generate a small portion of your system.

Once you have enough tests you will suddenly find that the system is starting to emerge from these tests.

The good news is that since your system has grown out of your test framework you have a good level of confidence that your system is in fact correct and of a certain level of quality.

This document presents five pathways to building your system.

I assume that you already have in place:

- A smoke and mirrors prototype (along with use case diagrams and description)
- A database (plus ERD and data dictionary)
- A class diagram (Triangulated against your sequence diagram and use case)
- Test plans for your classes

The five pathways are as follows...

- Pathway 1 – Creating a Simple Class
- Pathway 2 – Creating a “Simple” Data Bound Collection
- Pathway 4 – Creating the Complex Item Class
- Pathway 5 – Creating the Complex Collection Class

Pathway 1 – Creating a Simple Class

In this example we see how to create a reasonably simple class from nothing to having a full set of properties along with a simple validation method.

Pathway 2 – Creating a “Simple” Data Bound Collection

Here we build on the previous work to create a data bound collection class. We will see how to create a public list of items that is automatically populated at the class’s instantiation. At the end of this example we will link the middle layer functionality to the presentation layer.

Pathway 4 – Creating the Complex Item Class

In this example we will create a class with a wider range of properties and examine approaches to testing a more complex range of data types.

Pathway 5 – Creating the Complex Collection Class

Here we will repeat much of the work for pathway 2 however we will also create Add, Update and Delete methods based around the ThisAddress property. As each section of the middle layer functionality is created we will link this to the presentation layer.

It is also worth while looking at the two sample applications provided with this document.

The first application is provided for you to try out the examples.

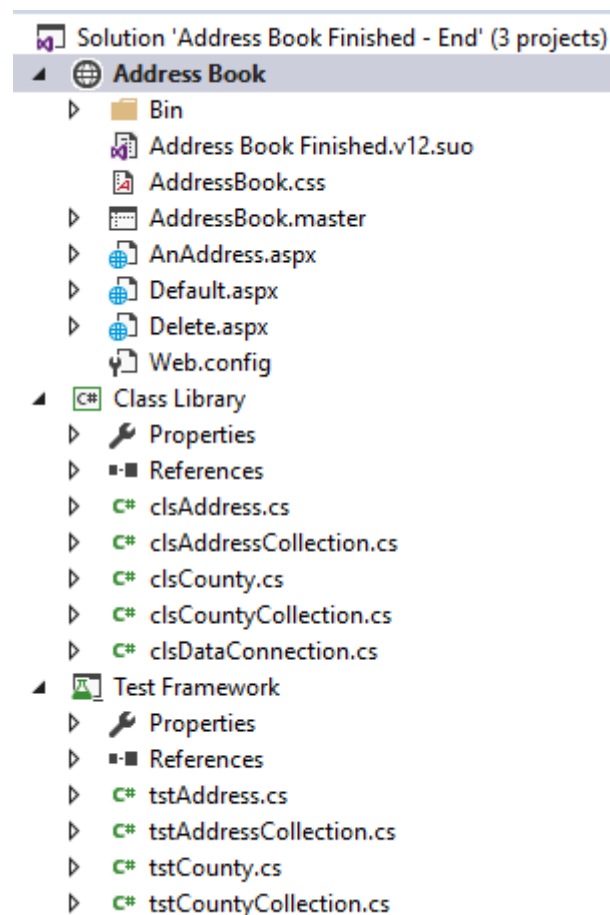
The second example is for you to see what the finished work will look like.

Do this first!

No – I really mean this. If you don't you are going to end up making a lot of unnecessary errors!

On the Programming Pathways there is a finished system called “Address Book Finished – End” stored as a ZIP file. Download the file, unzip it and open it in Visual Studio.

Take a look at how the solution is configured in the Solution Explorer.










In this example we have a presentation layer using web forms called “Address Book”, a class library and a test project.

So here is the question – where is the database file?

You may well be expecting the database to be stored in a sub folder of the web site called App_Data. In a simpler configuration this would be true, however one thing we want to do with this design is provide the opportunity for the data layer and middle layer to be shared across multiple interfaces. So where is the database file?

Look at the extracted files outside of Visual Studio using Windows Explorer...

	Address Book	08/06/2017 16:16	File folder	
	App_Data	08/06/2017 12:26	File folder	
	Class Library	08/06/2017 15:49	File folder	
	Test Framework	08/06/2017 15:50	File folder	
	TestResults	08/06/2017 16:35	File folder	
	Address Book Finished - End.sln	25/09/2015 18:45	Microsoft Visual S...	3 KB
	Address Book Finished - End.v11.suo	25/09/2015 18:45	Visual Studio Solu...	43 KB

Here you will find the App_Data folder for the solution.

When it comes to the stage of adding your database file to the system you will need to do the same.

But what if I want to locate my database somewhere else?

Well you can!

If you examine the code for the DataConnection you will see the constructor...

```
7 references
public clsDataConnection()
{
    GetConnectionString(GetDBName());
}
```

To override the class looking in the default location modify the constructor like the so...

```
7 references
public clsDataConnection()
{
    //GetConnectionString(GetDBName());
    GetConnectionString("I:\\TeamXYZ\\MyDB.mdf");
}
```

This line will force the data connection to look in "I:\\TeamXYZ\\MyDB.mdf"

Note: if one of your team has the database open in Visual Studio it will lock out other people from accessing it. If you need to do something to the database, make sure you tell your team members and make sure you close the connection once done.

So you have a Class Diagram!

Good for you!

The next step is to turn that diagram into some classes and ultimately some code that does the job you want.

The first rule of development is to do the simple stuff first.

Writing a program is like building a house.

You simply cannot put the roof on the house before you have the foundations and walls in place.

Developing a system is often made difficult by trying to perform the final tasks first!

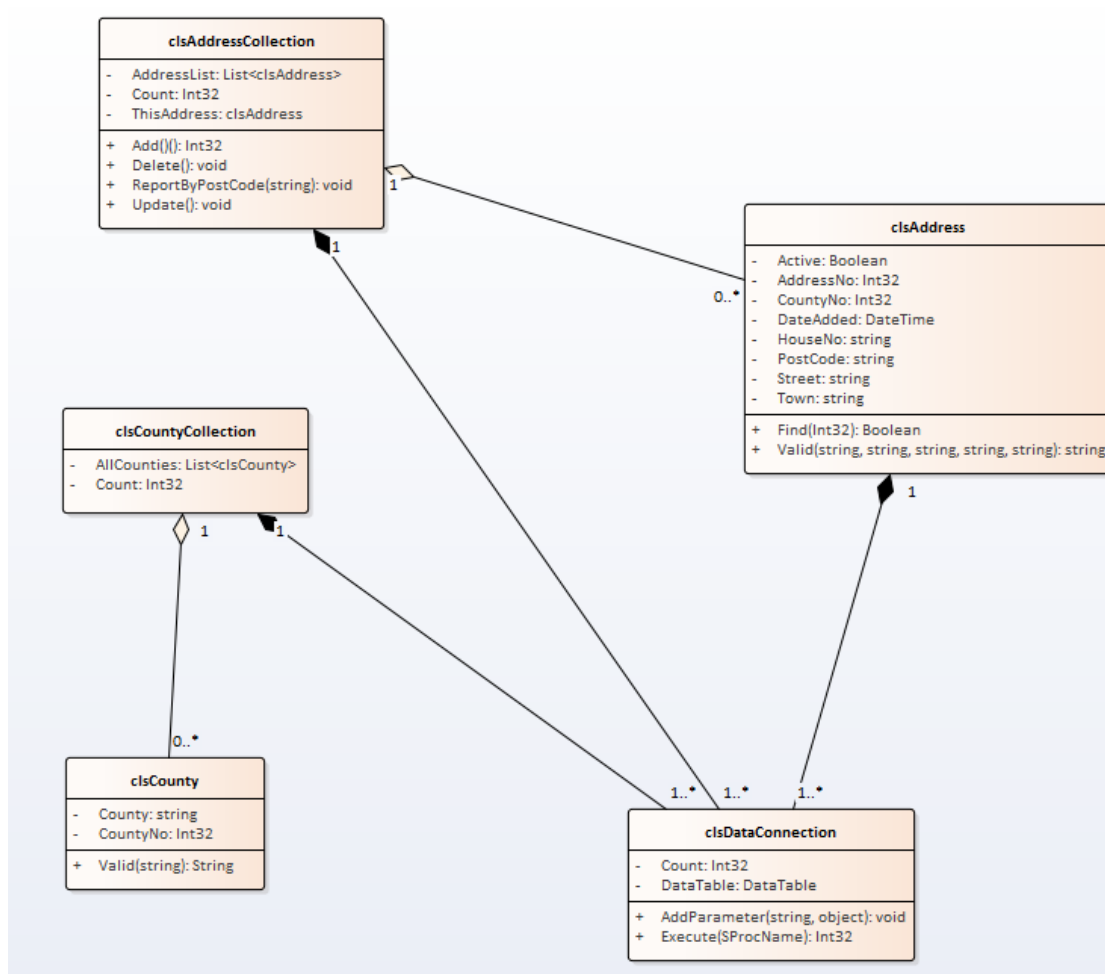
Do the stuff you understand first – the simple stuff.

Once you have achieved that, the rest of the development will make more sense.

In this guide we will look at some approaches to programming that break the problem down into small digestible chunks or bricks.

Once we have enough bricks created we will be in the position to assemble them into something looking like a system.

The first clue as to what the bricks look like is the class diagram.



I assume you have your own class diagram with similar elements present.

You will have

- Individual classes
- Attributes
- Operations
- Relationships indicating
 - Aggregation
 - Composition
 - Collections

There is a lot of development required here.

Pathway 1 – Creating a Simple Class

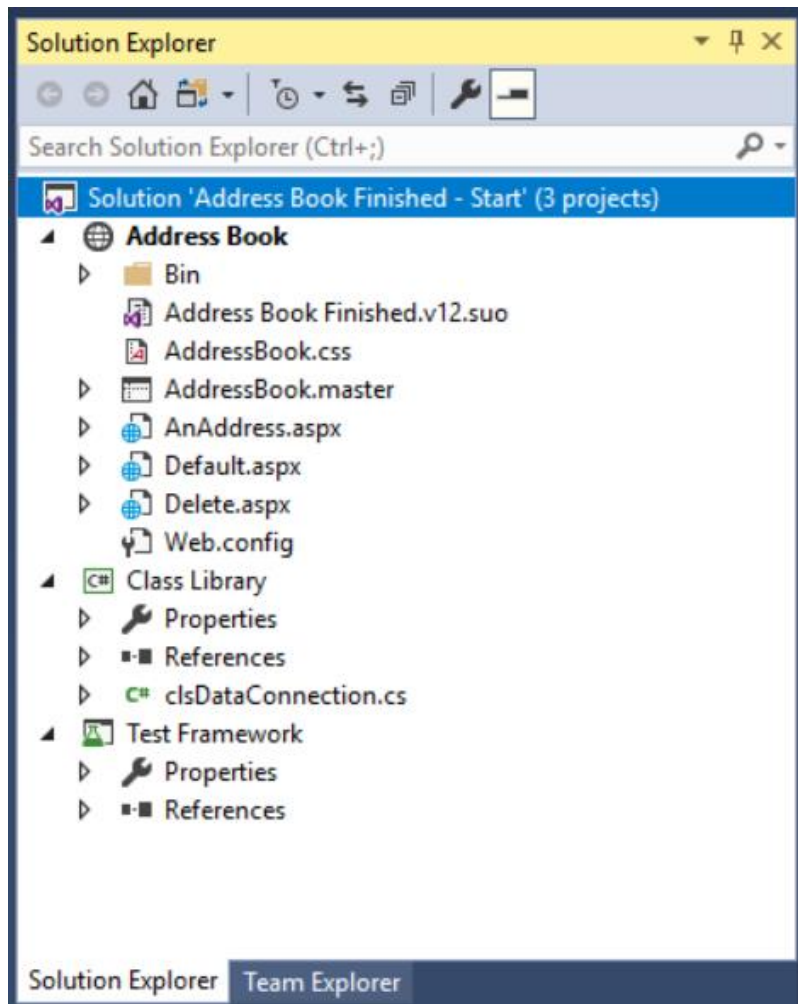
So as stated above let's start with the simplest thing that we can possibly do.

Let's create a test to ensure that an instance of a class may be created.

I am assuming the Visual Studio is configured such that you have solution containing

- The project you are creating
- The class library where the projects classes will live
- The test framework where the test classes will live

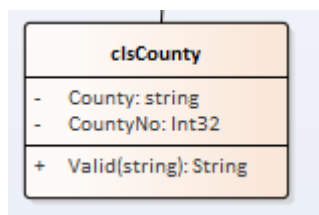
I am also assuming that these have all been linked together correctly.



The next step is to decide on which class we are going to implement.

We could in theory pick any of them but as I said above it is best to start with something nice and simple.

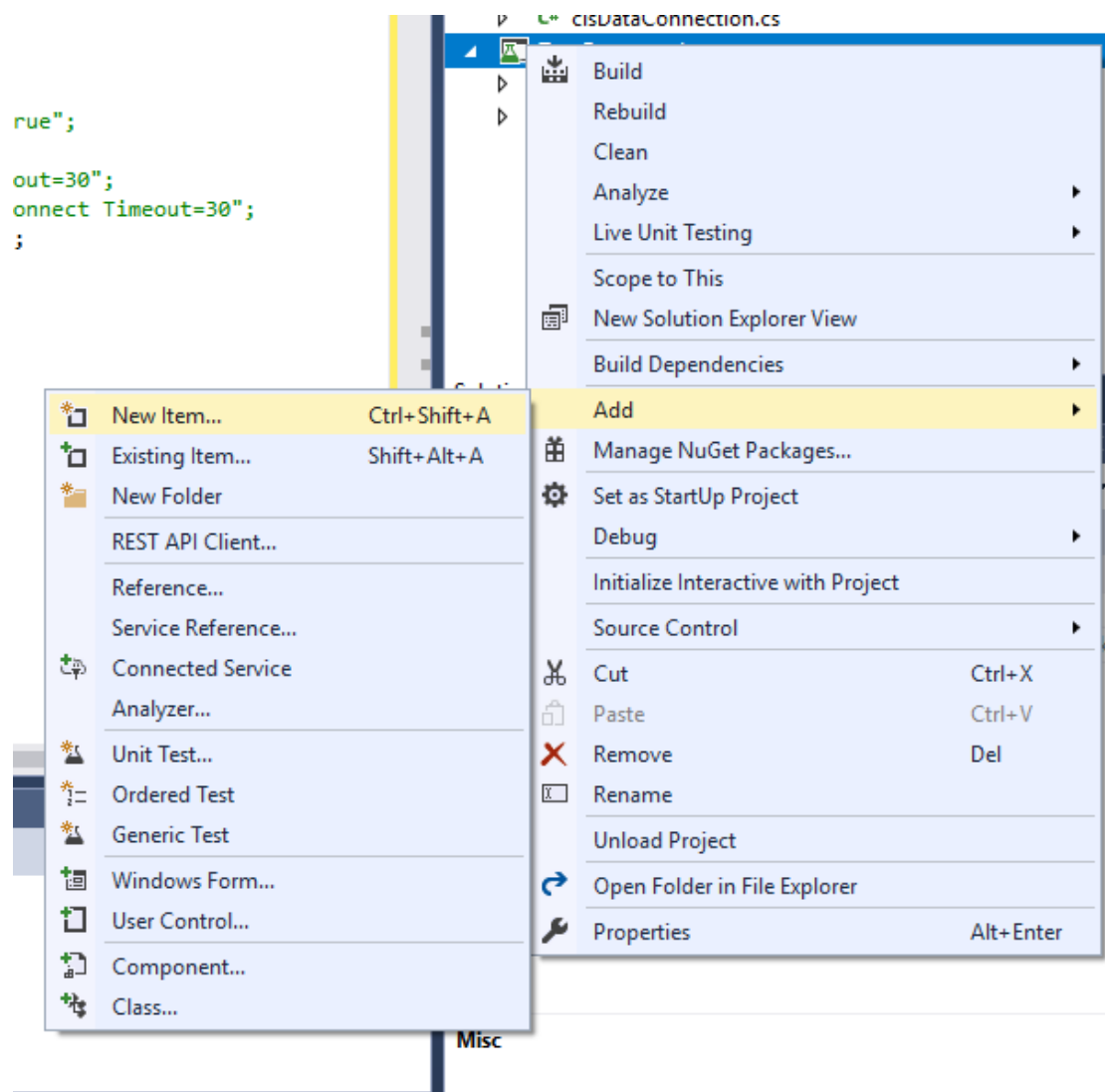
So we will create the class clsCounty



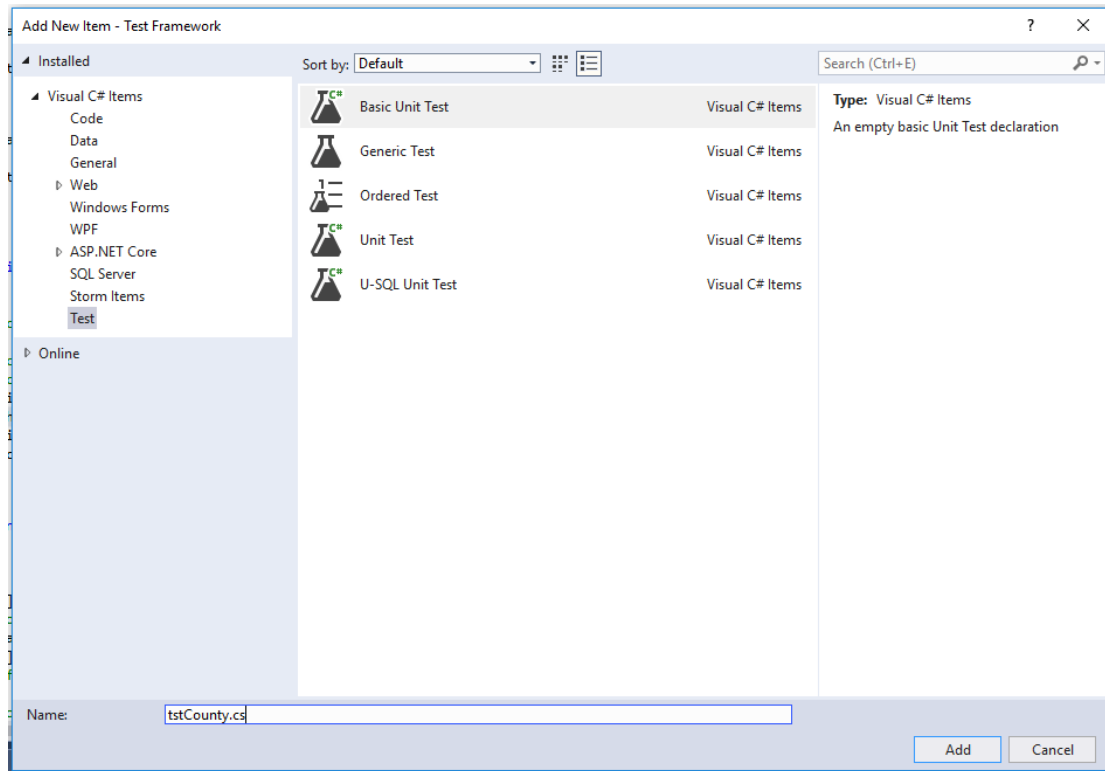
It has two attributes and one operation.

To create the test class for this class we need to create a test class called tstCounty

To create the test, right click on the test framework select Add – New Item...



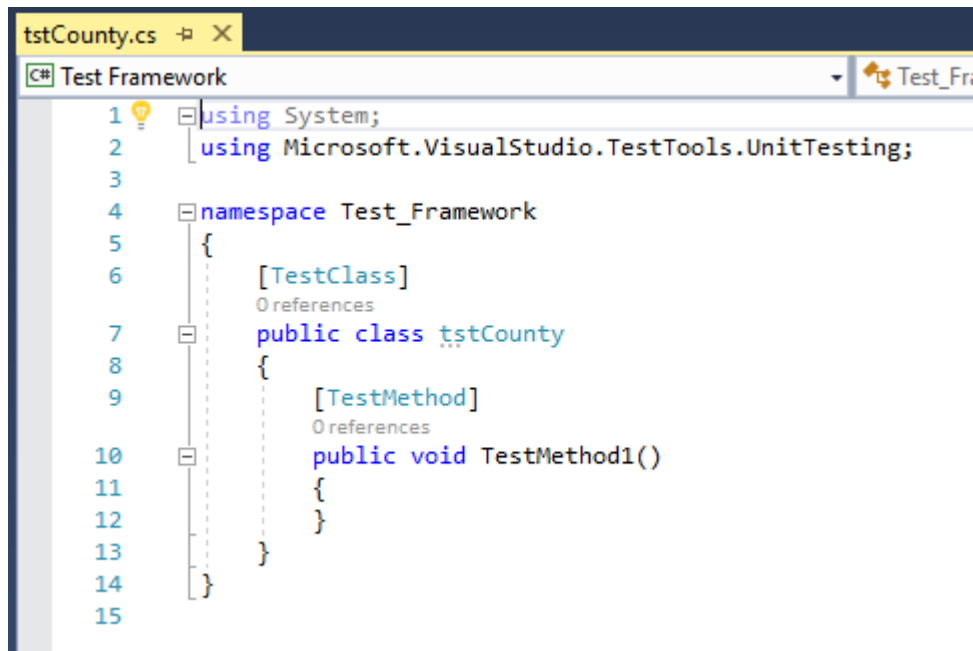
From the list of options, select basic unit test and set the name of the test class to tstCounty.



The idea is that for each class that we create from the class diagram we will have a corresponding test class to contain the testing for that class.

So tstCounty contains the testing for clsCounty!

You should see something like this...



We will now create the first test.

There are three rules for TDD

1. Create a test that fails
2. Fix the test (in some simplistic way)
3. Re-factor the code to make it work “properly”

The first test we will create is to see if we can make a valid instance of the class `clsCounty`.

The first test method should look something like this...

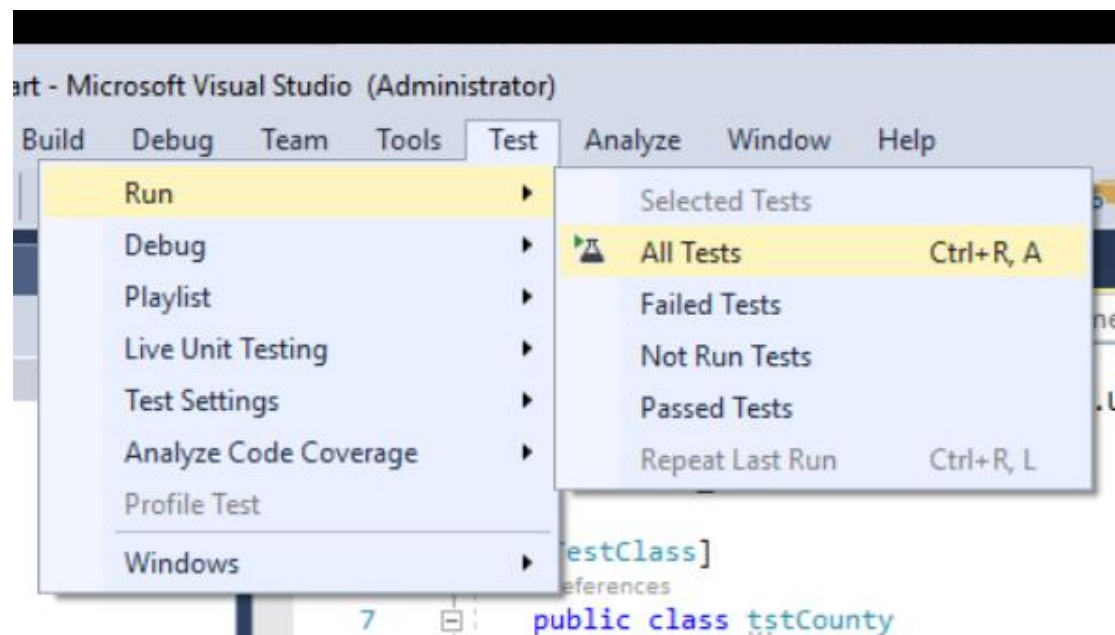
```
[TestMethod]
public void InstanceOK()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //test to see that it exists
    Assert.IsNotNull(ACounty);
}
```

Things to note at this stage

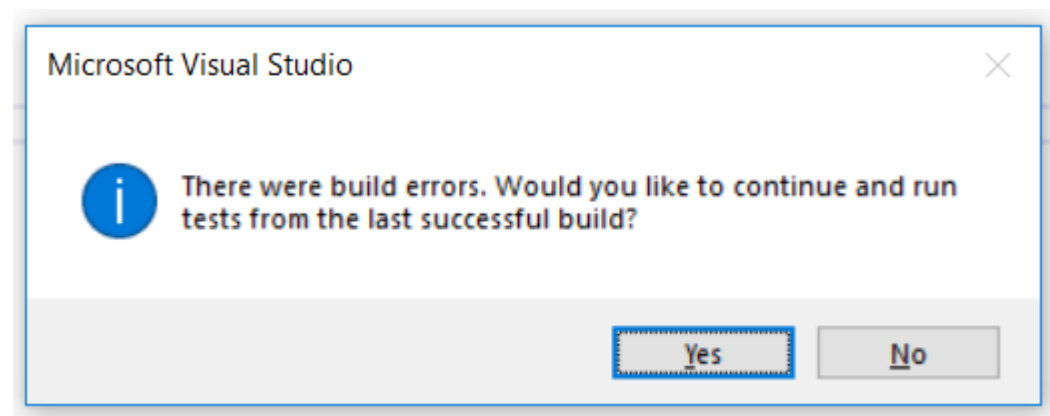
- The fact that the class hasn't been created is irrelevant
- The red underlining is not important
- It is a good idea to name your tests in a way that makes sense. Doing this ensures that when things go wrong (and they will) you have some idea as to where the problem is.

The next step is to run the test and see it fail.

From the main menu select, Test – Run – All Tests...



In this case you should (obviously) get a compilation error due to the red underlining...



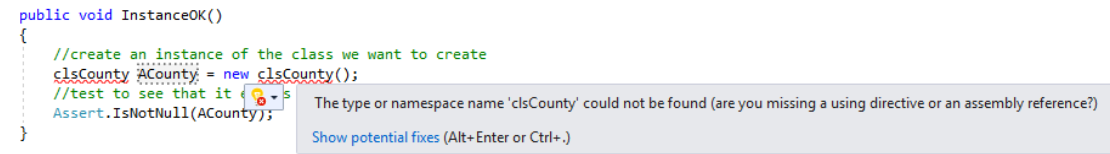
(Press No!)

Error List		
<div> <div>Entire Solution</div> <div>2 Errors</div> <div>0 Warnings</div> <div>0 of 1 Message</div> <div>Build + IntelliSense</div> </div>		
	Code	Description
	CS0246	The type or namespace name 'clsCounty' could not be found (are you missing a using directive or an assembly reference?)
	CS0246	The type or namespace name 'clsCounty' could not be found (are you missing a using directive or an assembly reference?)
		Test Framework
		Test Framework

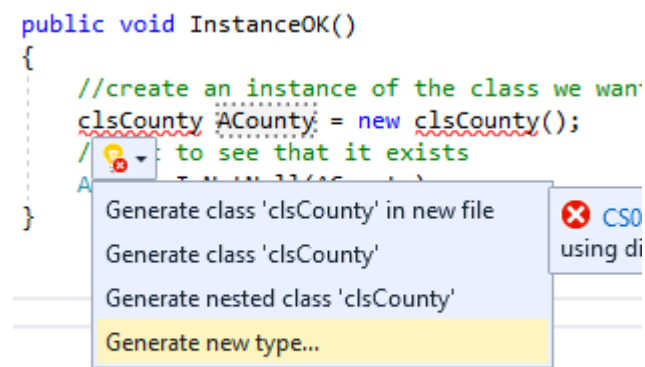
OK – the next step is to fix the problem.

We need to make sure that the class actually exists in our class library.

Hold the mouse over the class name with the red underlining and you will see the following message pop up...

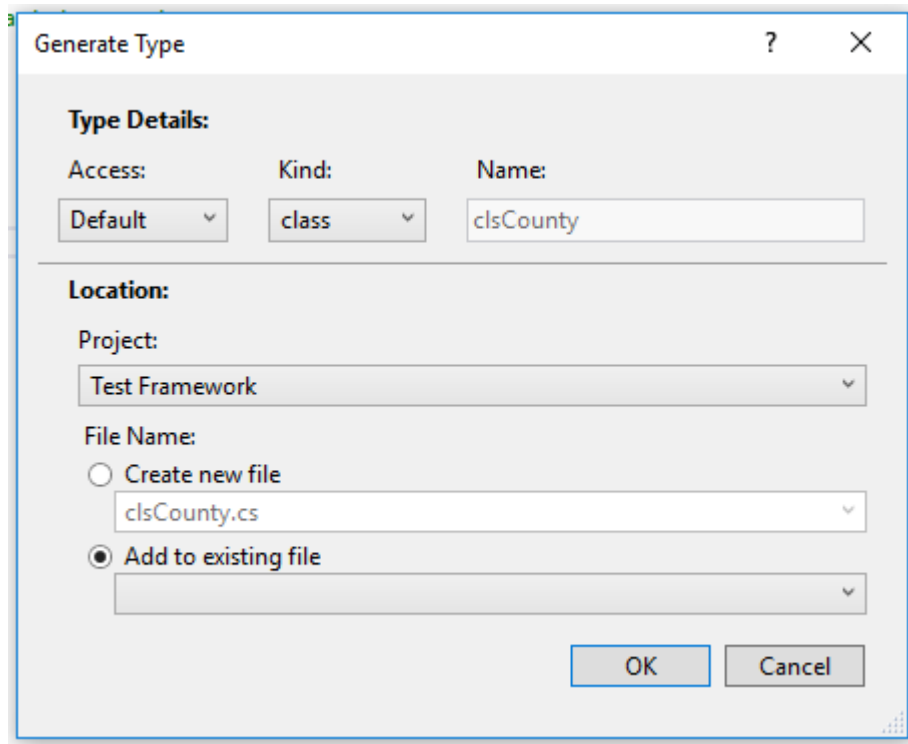


Click on “show potential fixes” you should see the following options



From this menu select Generate new type.

You should see the following window...



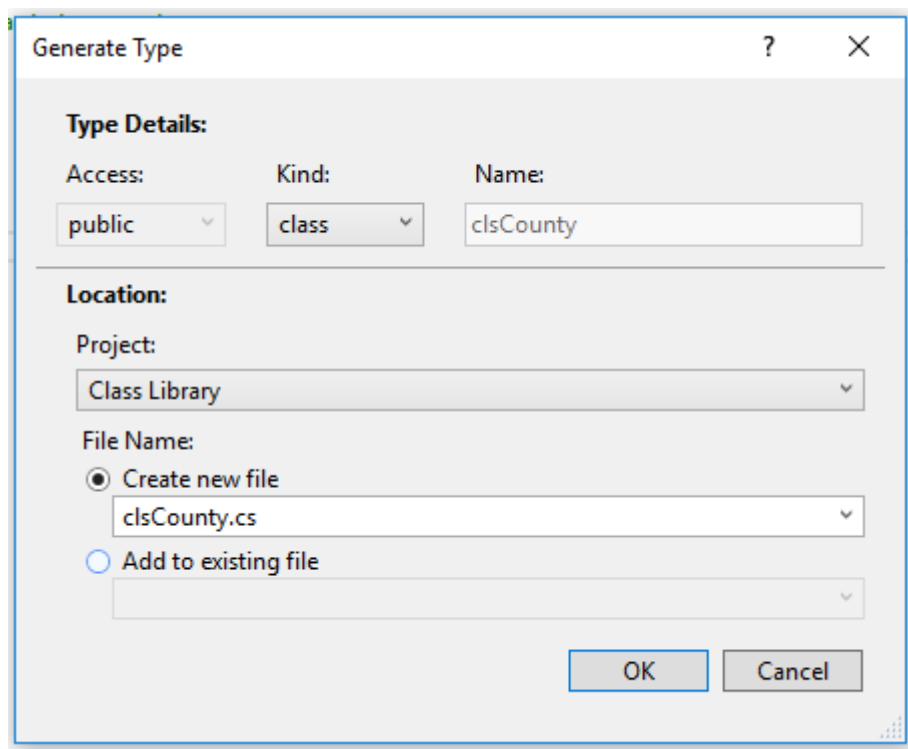
The 'Generate Type' dialog box is shown with the following settings:

- Type Details:**
 - Access: Default
 - Kind: class
 - Name: clsCounty
- Location:**
 - Project: Test Framework
 - File Name:
 - ☐ Create new file (selected)
 - clsCounty.cs
 - ☒ Add to existing file

Buttons: OK, Cancel

You really need to pay attention to the settings on this window as getting it wrong will result in your class being created in the wrong location.

Change project location to your class library, not the test framework and make sure that “Create new file” is selected like so...



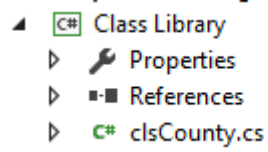
The 'Generate Type' dialog box is shown with the following settings:

- Type Details:**
 - Access: public
 - Kind: class
 - Name: clsCounty
- Location:**
 - Project: Class Library
 - File Name:
 - ☒ Create new file (selected)
 - clsCounty.cs
 - ☐ Add to existing file

Buttons: OK, Cancel

Then press OK.

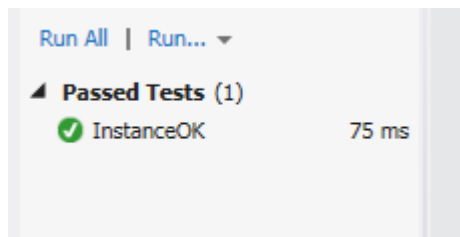
You will know that you have done this correctly because your class file will be visible in your class library...



The red underlining should also disappear...

```
[TestMethod]
public void InstanceOK()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //test to see that it exists
    Assert.IsNotNull(ACounty);
}
```

Run your tests and you should see the test pass...



This part of the process may seem trivial however it really isn't.

You have just dug the first hole in building the foundations of your house. Without that first hole in the ground nothing else will happen in the construction process.

We could now go in several different directions with the development of the system.

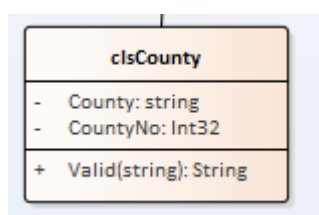
We could concentrate on this class and get that finished i.e. dig the hole deeper until it is done. Or we could make a start on all of the other classes, i.e. make a start on all of the other holes for the foundation.

For this example we will concentrate on making this hole deeper so that we end up with a (mostly) completed class file.

Creating Properties

So what next?

Looking at the class diagram we may as well go for the two properties...



So let's write the test immediately following the test we have already created.

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

namespace Test_Framework
{
    [TestClass]
    0 references
    public class tstCounty
    {
        [TestMethod]
        ✓ | 0 references
        public void InstanceOK()
        {
            //create an instance of the class we want to create
            clsCounty ACounty = new clsCounty();
            //test to see that it exist
            Assert.IsNotNull(ACounty);
        }

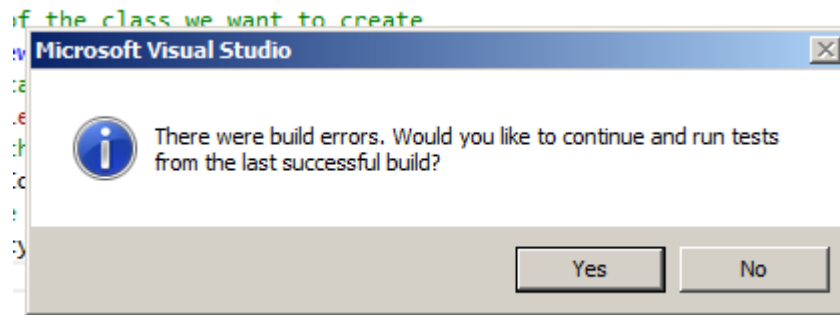
        //create new test here !!!!!!!!!!!
    }
}
```

Here is the next test...

```
[TestMethod]
public void CountyPropertyOK()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create some test data to assign to the property
    string SomeCounty = "Leicestershire";
    //assign the data to the property
    ACounty.County = SomeCounty;
    //test to see that the two values are the same
    Assert.AreEqual(ACounty.County, SomeCounty);
}
```

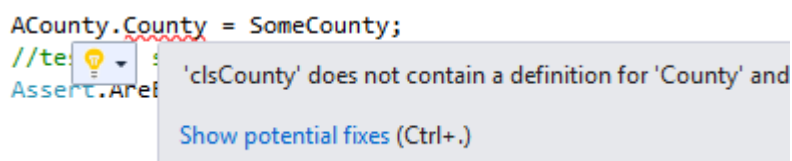
From the red underlining we should be able to see that there is going to be a problem.

Try running the test to confirm that it doesn't compile...

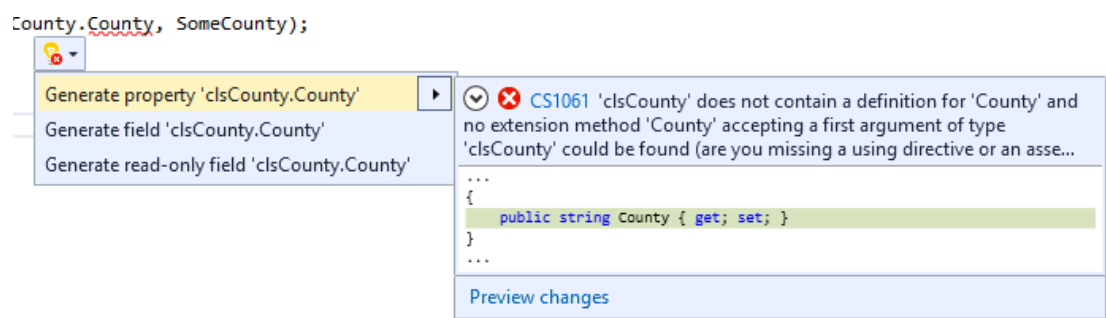


So the next step is to create the County property.

As before hold the mouse over the red underlining and select show potential fixes...



Followed by the option to Generate the property...



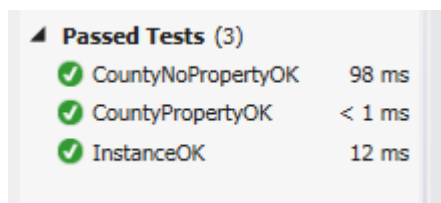
This should result in your tests passing.

Create a similar test for the CountyNo property like so...

```
[TestMethod]
public void CountyNoPropertyOK()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create some test data to assign to the property
    Int32 CountyNo = 1;
    //assign the data to the property
    ACounty.CountyNo = CountyNo;
    //test to see that the two values are the same
    Assert.AreEqual(ACounty.CountyNo, CountyNo);
}
```

Now follow the same procedure. Run the test watch it fail and then put in the fix.

You should see all tests pass.



▲ Passed Tests (3)	
✓ CountyNoPropertyOK	98 ms
✓ CountyPropertyOK	< 1 ms
✓ InstanceOK	12 ms

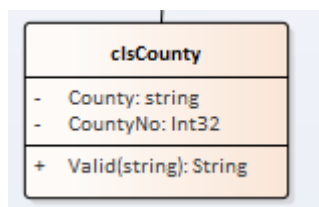
Again this may not seem like much but we are slightly closer to the finished system than we were a few tests ago.

The other important thing is that in writing the tests we can be confident that the code is correct.

Ideally the code for the system should grow gradually out of the tests that we write.

Creating Methods

So what's left?



We have one operation to think about which will be needed to validate any new county values we want to enter into the system.

The method will accept a single parameter representing a new county to be validated. The method will return a string reporting the nature of the error.

We could use the method like this...

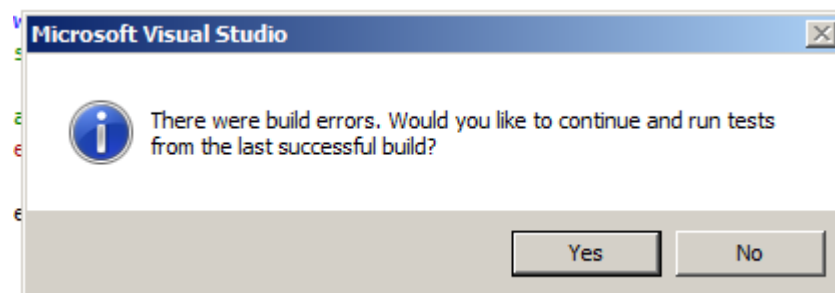
```
clsCounty ACounty = new clsCounty();
string NewCounty = txtCounty.Text;
if (clsCounty.Valid(NewCounty) == "")
{
    //do something with the data
}
else
{
    //display an error message
}
```

So the first step is to write a test method that checks to see that the method actually exists.

```
[TestMethod]
public void ValidMethodOK()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create a string variable to store the result of the validation
    String Error = "";
    //create some test data to test the method
    string SomeCounty = "Leicestershire";
    //invoke the method
    Error = ACounty.Valid(SomeCounty);
    //test to see that the result is OK i.e there was no error message returned
    Assert.AreEqual(Error, "");
}
```

In this case we are sending the method some valid data so we would expect the result of the validation to be true, that is a blank string.

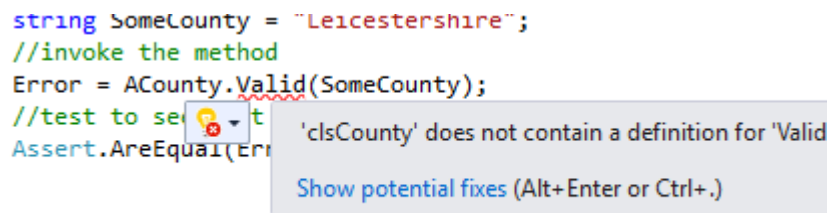
As before because we have yet to write the method so the test will fail due to the red underlining...



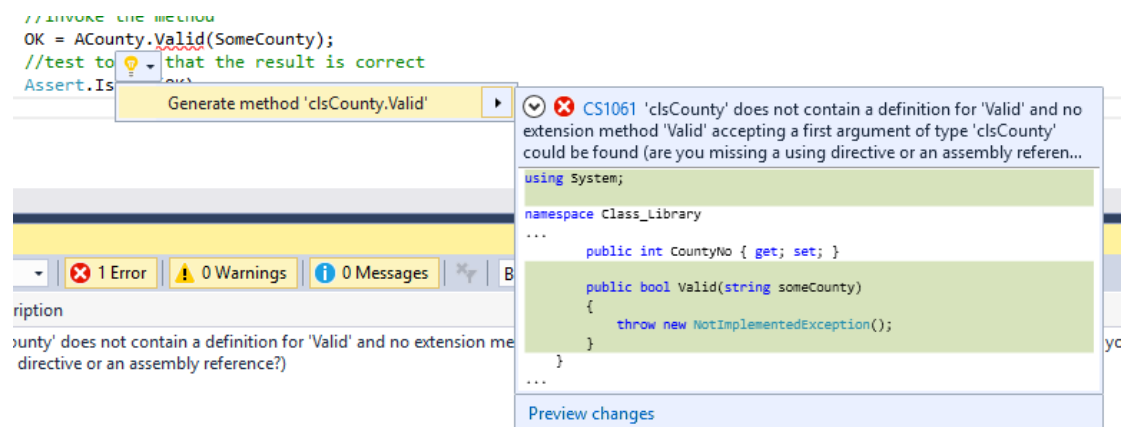
To fix this we need to create the method stub.

The procedure is as above.

Hold the mouse over the red underlining...

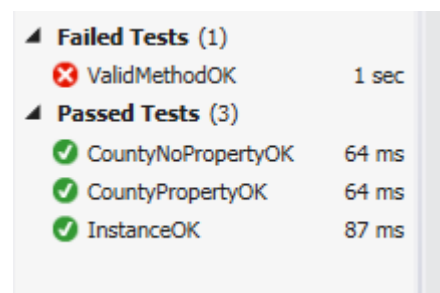


Select “Show potential fixes”...



Generate the method.

Run the test again and you should see it fail! (Now what!?!)



The problem is that for the method to work correctly we need to write some actual code.

If we open the class file `clsCounty` and take a look at the auto generated code we see the following...

```

namespace Class_Library
{
    public class clsCounty
    {
        public string County { get; set; }

        public int CountyNo { get; set; }

        public bool Valid(string SomeCounty)
        {
            throw new NotImplementedException();
        }
    }
}

```

The code stating `throw new NotImplementedException();` is resulting in the program always giving us an error.

First we need to get rid of the line as we don't need it.

Next we need to add some of our own code to implement a partial fix to make the test work.

What we are checking for is if a non blank value has been supplied to the function via the parameter we want the function to return an error message. If the name of the county is blank then we need to return a blank string.

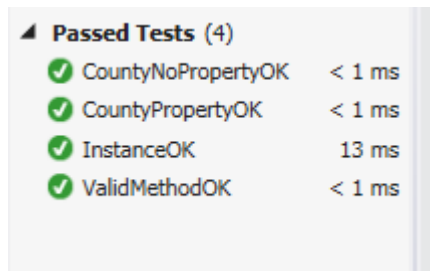
The following code should do the trick...

```

1 reference | 0/1 passing
public string Valid(string someCounty)
{
    //if the name of the county is not blank
    if (someCounty != "")
    {
        //return a blank string
        return "";
    }
    else
    {
        //otherwise return an error message
        return "The county name may not be blank!";
    }
}

```

Run the test and it should pass.



Passed Tests (4)		
✓	CountyNoPropertyOK	< 1 ms
✓	CountyPropertyOK	< 1 ms
✓	InstanceOK	13 ms
✓	ValidMethodOK	< 1 ms

We are now a bit further on in building our system with a good level of confidence that we are doing it right.

The next step is to spend a bit of time creating a more structured approach to the test data.

To do this we will use the following test plan pro forma.

Description of Item to Be Tested:

--

Required Field **Y / N**

Test Type	Test Data	Expected Result	Actual Result
Extreme Min			
Min -1			
Min (Boundary)			
Min +1			
Max -1			
Max (Boundary)			
Max +1			
Mid			
Extreme Max			
Invalid data type			
Other tests			

Additional Notes / Instructions:

--

In order to test the parameter for the county validation we will need to think about appropriate tests so that we know it is working correctly.

We will also need to do this with an eye to the database design. This information may be obtained from the data dictionary or from the database implementation.

In this example we will use an existing database implementation.

```
CREATE TABLE [dbo].[tblCounty] (  
    [CountyNo] INT IDENTITY (1, 1) NOT NULL,  
    [County] VARCHAR (50) NOT NULL,  
    PRIMARY KEY CLUSTERED ([CountyNo] ASC)  
);
```

From this we can see that the County field may not be empty (NOT NULL) and that the name of the county has a maximum length of 50 characters (VARCHAR(50)).

The test data might look something like this...

Description of Item to Be Tested:

SomeCounty parameter of the Valid method of clsCounty.
--

Required Field **Y**

Test Type	Test Data	Expected Result	Actual Result
Extreme Min	NA	NA	
Min -1	Blank string	Fail	
Min (Boundary)	One character 'a'	Pass	
Min +1	Two characters 'aa'	Pass	
Max -1	49 characters 0123456789012345678901234567890123456789012345678	Pass	
Max (Boundary)	50 characters 01234567890123456789012345678901234567890123456789	Pass	
Max +1	51 characters 012345678901234567890123456789012345678901234567890	Fail	
Mid	25 characters 0123456789012345678901234	Pass	
Extreme Max	500 characters 01234567890123456789012345678901234567890123456789 01234567890123456789012345678901234567890123456789 01234567890123456789012345678901234567890123456789 01234567890123456789012345678901234567890123456789 01234567890123456789012345678901234567890123456789 01234567890123456789012345678901234567890123456789 01234567890123456789012345678901234567890123456789 01234567890123456789012345678901234567890123456789 01234567890123456789012345678901234567890123456789 01234567890123456789012345678901234567890123456789	Fail	
Invalid data type	NA		
Other tests	NA		

Additional Notes / Instructions:

--

--

We now need to write the test methods to apply the above test plan.

So the tests we need to write are as follows...

Min -1	Blank string	Fail
--------	--------------	------

```
[TestMethod]
0 references
public void CountyMinLessOne()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create a string variable to store the result of the validation
    String Error = "";
    //create some test data to test the method
    string SomeCounty = "";
    //invoke the method
    Error = ACounty.Valid(SomeCounty);
    //test to see that the result is NOT OK i.e there should be an error message
    Assert.AreNotEqual(Error, "");
}
```

With the code we created for the previous test we get lucky as it also addresses this new test too.

Min (Boundary)	One character 'a'	Pass
-------------------	-------------------	------

```
[TestMethod]
0 references
public void CountyMinBoundary()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create a string variable to store the result of the validation
    String Error = "";
    //create some test data to test the method
    string SomeCounty = "a";
    //invoke the method
    Error = ACounty.Valid(SomeCounty);
    //test to see that the result is OK i.e there was no error message returned
    Assert.AreEqual(Error, "");
}
```

Same with this test – our existing code should address it too.

Min +1	Two characters 'aa'	Pass
--------	---------------------	------

```
[TestMethod]
0 references
public void CountyMinPlusOne()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create a string variable to store the result of the validation
    String Error = "";
    //create some test data to test the method
    string SomeCounty = "aa";
    //invoke the method
    Error = ACounty.Valid(SomeCounty);
    //test to see that the result is OK i.e there was no error message returned
    Assert.AreEqual(Error, "");
}
```

Max -1	49 characters 0123456789012345678901234567890123456789012345678	Pass
--------	--	------

```
[TestMethod]
0 references
public void CountyMaxLessOne()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create a string variable to store the result of the validation
    String Error = "";
    //create some test data to test the method
    string SomeCounty = "0123456789012345678901234567890123456789012345678";
    //invoke the method
    Error = ACounty.Valid(SomeCounty);
    //test to see that the result is OK i.e there was no error message returned
    Assert.AreEqual(Error, "");
}
```

This one should be OK too.

Max (Boundary)	50 characters 01234567890123456789012345678901234567890123456789	Pass
-------------------	---	------

```
[TestMethod]
0 references
public void CountyMaxBoundary()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create a string variable to store the result of the validation
    String Error = "";
    //create some test data to test the method
    string SomeCounty = "01234567890123456789012345678901234567890123456789";
    //invoke the method
    Error = ACounty.Valid(SomeCounty);
    //test to see that the result is OK i.e there was no error message returned
    Assert.AreEqual(Error, "");
}
```

And this one!

Max +1	51 characters 0123456789012345678901234567890123456789012345678901234567890	Fail
-----------	--	------

This test is not accounted for by our existing code – it should give the wrong result.

```
[TestMethod]
0 references
public void CountyMaxPlusOne()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create a string variable to store the result of the validation
    String Error = "";
    //create some test data to test the method
    string SomeCounty = "012345678901234567890123456789012345678901234567890";
    //invoke the method
    Error = ACounty.Valid(SomeCounty);
    //test to see that the result is NOT OK i.e there should be an error message
    Assert.AreNotEqual(Error, "");
}
```

▲ Failed Tests (1)		
✖	CountyMaxPlusOne	120 ms
▲ Passed Tests (9)		
✓	CountyMaxBoundary	< 1 ms
✓	CountyMaxLessOne	< 1 ms
✓	CountyMinBoundary	< 1 ms
✓	CountyMinLessOne	< 1 ms
✓	CountyMinPlusOne	< 1 ms
✓	CountyNoPropertyOK	< 1 ms
✓	CountyPropertyOK	< 1 ms
✓	InstanceOK	12 ms
✓	ValidMethodOK	1 ms

So we need to modify the validation code to take into account for data longer than 50 characters.

Let's modify the validation code so that it looks like this...

```
7 references | 7/7 passing
public string Valid(string someCounty)
{
    //string variable to store the error message
    string Error = "";
    //if the name of the county is more than 50 characters
    if (someCounty.Length > 50)
    {
        //return an error message
        Error = "The county name cannot have more than 50 characters";
    }
    if (someCounty.Length == 0)
    {
        //return an error message
        Error = "The county name may not be blank!";
    }
    return Error;
}
```

Run the tests – do they all pass?

Yes they do.

Passed Tests (10)	
✓ CountyMaxBoundary	< 1 ms
✓ CountyMaxLessOne	< 1 ms
✓ CountyMaxPlusOne	< 1 ms
✓ CountyMinBoundary	< 1 ms
✓ CountyMinLessOne	< 1 ms
✓ CountyMinPlusOne	< 1 ms
✓ CountyNoPropertyOK	< 1 ms
✓ CountyPropertyOK	< 1 ms
✓ InstanceOK	12 ms
✓ ValidMethodOK	< 1 ms

Mid	25 characters 0123456789012345678901234	Pass
-----	--	------

What about the mid range test?

```
[TestMethod]
0 references
public void CountyMid()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create a string variable to store the result of the validation
    String Error = "";
    //create some test data to test the method
    string SomeCounty = "0123456789012345678901234";
    //invoke the method
    Error = ACounty.Valid(SomeCounty);
    //test to see that the result is OK i.e there was no error message returned
    Assert.AreEqual(Error, "");
}
```

All fine.

What about the extreme max test?

Extreme Max	500 characters 012345678901234567890123456789012345678901234567890123456789 012345678901234567890123456789012345678901234567890123456789 012345678901234567890123456789012345678901234567890123456789 012345678901234567890123456789012345678901234567890123456789 012345678901234567890123456789012345678901234567890123456789 012345678901234567890123456789012345678901234567890123456789 012345678901234567890123456789012345678901234567890123456789 012345678901234567890123456789012345678901234567890123456789 012345678901234567890123456789012345678901234567890123456789 012345678901234567890123456789012345678901234567890123456789	Fail
----------------	--	------

Well...

Before we create this test we need to think about how we are creating our test data. There are no hard and fast rules on how we do this. One thing we want to do is make it as easy as possible.

We could in the test data above type a string of 500 characters. Or we could make the system work on our behalf like so...

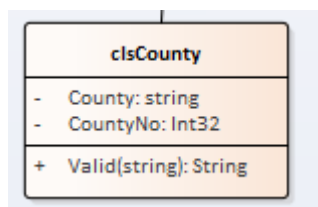
```
[TestMethod]
0 references
public void CountyExtremeMax()
{
    //create an instance of the class we want to create
    clsCounty ACounty = new clsCounty();
    //create a string variable to store the result of the validation
    String Error = "";
    //create some test data to test the method
    string SomeCounty = "";
    //pad the string with characters
    SomeCounty = SomeCounty.PadRight(500, 'a');
    //invoke the method
    Error = ACounty.Valid(SomeCounty);
    //test to see that the result is NOT OK i.e there should be an error message
    Assert.AreNotEqual(Error, "");
}
```

PadRight is a built in method that allows us to pad out an existing string with a set number of a specific character.

Having created the automated tests for the class we can be pretty sure that the component we have built so far is functioning correctly. Also as the system grows and we run our tests we get a constant check to make sure that we don't break our system further down the line.

The final comment on clsCounty is that we won't be creating tests for the CountyNo property.

Why?

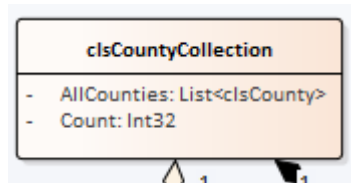


Being a primary key value created by the database we will assume that the database will handle this.

Pathway 2 – Creating a Simple Data Bound Collection

Let's now have a go at something more complex.

The next class we will look at is clsCountyCollection.



This class is associated with `clsCounty` so it makes sense to complete the pairing. The class is also an example of a collection class so we have the opportunity to think about some of the issues this creates.

Before we do anything complicated let's get back to basics.

First we need to create a test class for `clsCountyCollection`, i.e. `tstCountyCollection`...

```
tstCountyCollection.cs  [X]
Test_Framework.tstCountyCollection  [TestMet]

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

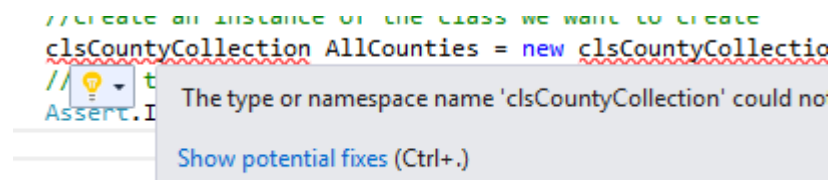
namespace Test_Framework
{
    [TestClass]
    public class tstCountyCollection
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

And as before we need to test that we are able to create an instance of `clsCountyCollection` like so...

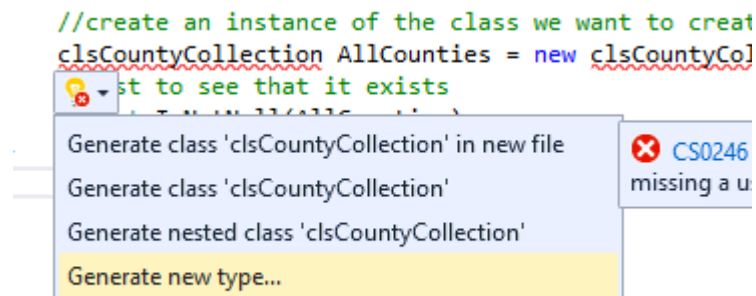
```
[TestMethod]
public void InstanceOK()
{
    //create an instance of the class we want to create
    clsCountyCollection AllCounties = new clsCountyCollection();
    //test to see that it exists
    Assert.IsNotNull(AllCounties);
}
```

The test will fail – as it should.

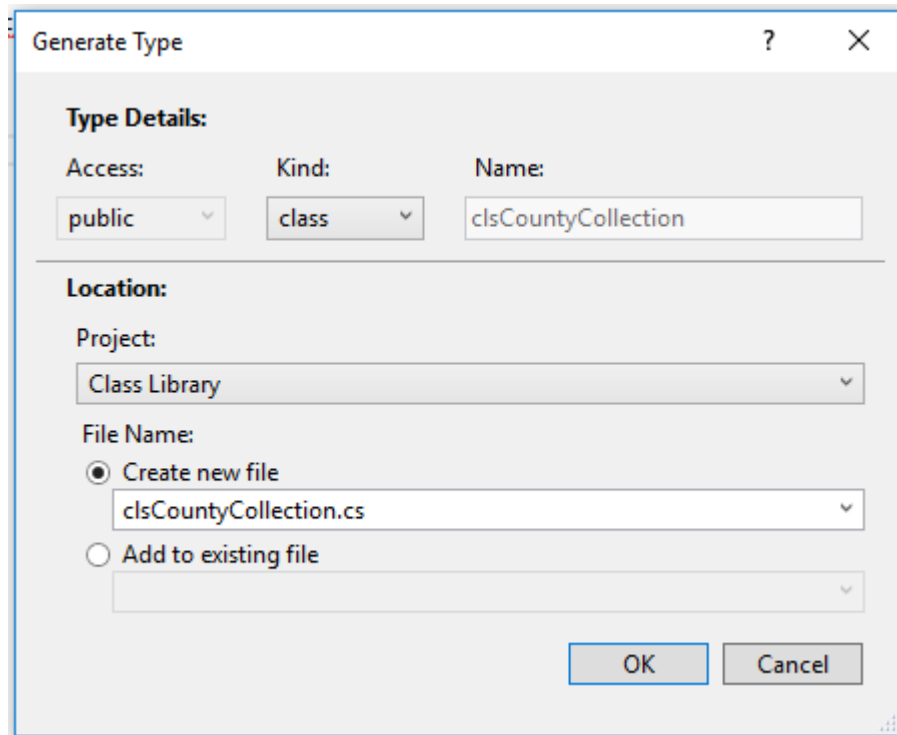
As above create the class in the class library...



Followed by...



Then make sure you set the next screen up correctly...



The 'Generate Type' dialog box is shown with the following settings:

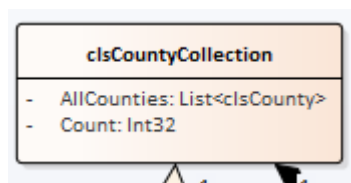
- Type Details:**
 - Access: public
 - Kind: class
 - Name: clsCountyCollection
- Location:**
 - Project: Class Library
 - File Name:
 - ☒ Create new file: clsCountyCollection.cs
 - ☐ Add to existing file: (empty)

Buttons: OK, Cancel

The test should now pass.

Creating the Properties

Next we want to create the properties...



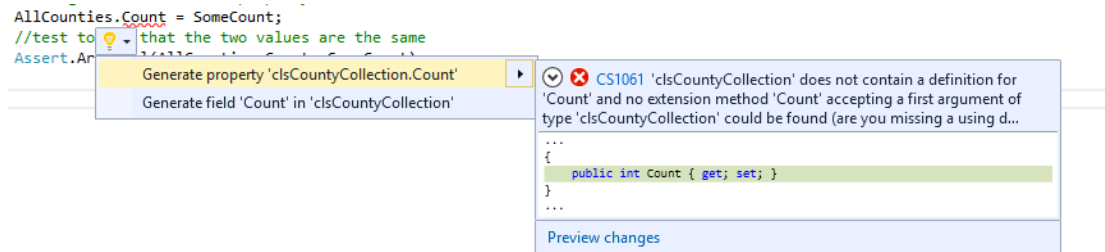
Count is the simplest one so let's do that first.

```

[TestMethod]
public void CountPropertyOK()
{
    //create an instance of the class we want to create
    clsCountyCollection AllCounties = new clsCountyCollection();
    //create some test data to assign to the property
    Int32 SomeCount = 1;
    //assign the data to the property
    AllCounties.Count = SomeCount;
    //test to see that the two values are the same
    Assert.AreEqual(AllCounties.Count, SomeCount);
}
  
```

As usual the test will fail since the property has not been created.

So let's do that now. As above show potential fixes followed by generate property...



Should do the trick!

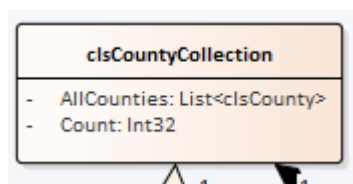
The important thing to appreciate with TDD is that it doesn't actually matter if our tests are initially rubbish!

With all of the code we write there needs to be a constant process of review called re-factoring.

In re-factoring we are constantly asking the question "is there a better way of doing this?"

In this next example we will see an example of this.

The next property we will create is the AllCounties array list.



When the class is completed this list will draw its data from the underlying database. The count property will also need to tell us exactly how many records there are in the list.

Linking the class to the database is a big jump even for an experienced programmer. Also making a big jump may sacrifice the quality of testing applied.

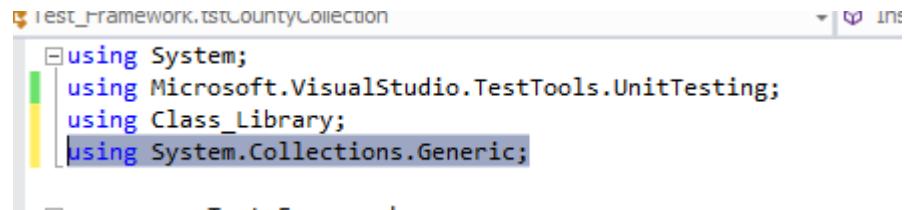
If we take small steps to solve the problem then we can test each step as we go and we won't get overwhelmed by the scale of the problem.

First off let's treat the property like any other property, ignoring the fact that it is going to be linked to the database.

The following code does this and it is worth spending a little while understanding what is going on.

Firstly we need to import the library code from the .NET framework allowing us to work with collections.

At the top of the test class add the following line of code...



```

test_Framework.tstCountyCollection
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Class_Library;
using System.Collections.Generic;

```

This will import the required features into the class.

Next we need to write the test method like so...

```

[TestMethod]
public void AllCountiesOK()
{
    //create an instance of the class we want to create
    clsCountyCollection Counties = new clsCountyCollection();
    //create some test data to assign to the property
    //in this case the data needs to be a list of objects
    List<clsCounty> TestList = new List<clsCounty>();
    //add an item to the list
    //create the item of test data
    clsCounty TestItem = new clsCounty();
    //set its properties
    TestItem.CountyNo = 1;
    TestItem.County = "Leicestershire";
    //add the item to the test list
    TestList.Add(TestItem);
    //assign the data to the property
    Counties.AllCounties = TestList;
    //test to see that the two values are the same
    Assert.AreEqual(Counties.AllCounties, TestList);
}

```

The test method will fail as we don't yet have the property AllCounties.

Create the property exactly as you did in the previous examples.

It is important to understand what we have done here and how it works.

Firstly we create an instance of the class we want to test – nothing new here.

```
//create an instance of the class we want to create
clsCountyCollection Counties = new clsCountyCollection();
```

In this example creating the test data is a little more complicated.

```
//create some test data to assign to the property
//in this case the data needs to be a list of objects
List<clsCounty> TestList = new List<clsCounty>();
//add an item to the list
//create the item of test data
clsCounty TestItem = new clsCounty();
//set its properties
TestItem.CountyNo = 1;
TestItem.County = "Leicestershire";
//add the item to the test list
TestList.Add(TestItem);
```

Since we are testing a property that contains a list of data we need to test it with another list, in this case TestList.

First we create an instance of the test list

```
List<clsCounty> TestList = new List<clsCounty>();
```

Now we need to add an item to the test list so that we have one entry in the list like so...

```
//create the item of test data
clsCounty TestItem = new clsCounty();
//set its properties
TestItem.CountyNo = 1;
TestItem.County = "Leicestershire";
//add the item to the test list
TestList.Add(TestItem);
```

Once we have created the test data we may now assign the test list to the property we want to test...

```
//assign the data to the property
Counties.AllCounties = TestList;
```

Lastly we test to see that the property and the original test data match as we have done in previous examples...

```
//test to see that the two values are the same
Assert.AreEqual(Counties.AllCounties, TestList);
```

And after all this the test passes.

There is however one major problem. The test passes but the functionality isn't behaving how we want it to.

This test proves that we may create a list in code and assign it to the property.

What we really want is for that list to be taken from the database and that data used to populate the list.

We also want the count property to keep track of how many items are in the list regardless of where the data is coming from.

So the question is where next?

We could write the code to link to the database but to be honest that's sounds too hard.

I am going to opt to make the count property work correctly as it is an easy win.

The first step is to create a test that fails which tests that the number of items in the list is the same as the value of the count property.

The following code which is a variation on the previous example does this.

```
[TestMethod]
public void CountMatchesList()
{
    //create an instance of the class we want to create
    clsCountyCollection Counties = new clsCountyCollection();
    //create some test data to assign to the property
    //in this case the data needs to be a list of objects
    List<clsCounty> TestList = new List<clsCounty>();
    //add an item to the list
    //create the item of test data
    clsCounty TestItem = new clsCounty();
    //set its properties
    TestItem.CountyNo = 1;
    TestItem.County = "Leicestershire";
    //add the item to the test list
    TestList.Add(TestItem);
    //assign the data to the property
    Counties.AllCounties = TestList;
    //test to see that the two values are the same
    Assert.AreEqual(Counties.Count, TestList.Count);
}
```

What we are doing is creating a list of test data with one county in it.

The count of this list is one.

This list is then used to set the list of the object Counties making them the same.

If everything is working correctly the Count property of Counties.Count should also be equal to 1 as it is using the same list.

We know that it is the same list because the previous test is checking it.

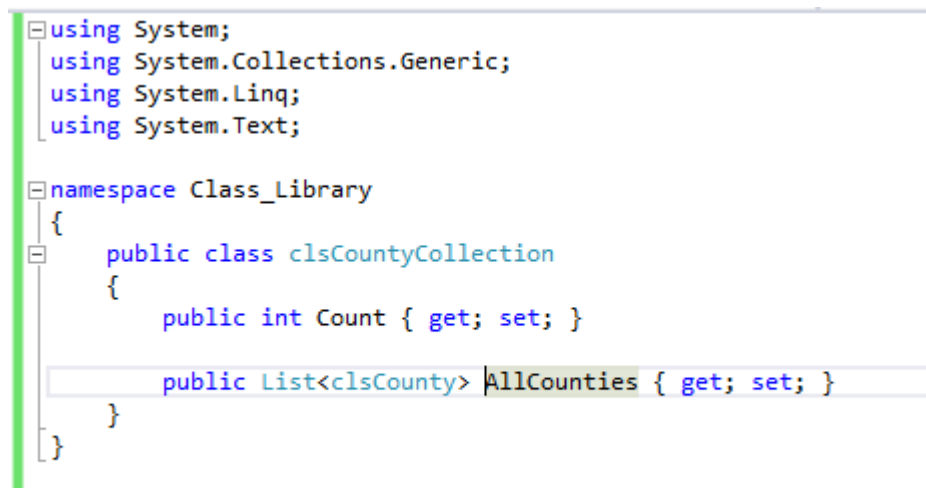
The problem is that when we run the test it fails.



▲ Failed Tests (1)	
✖ CountMatchesList	1 sec
▲ Passed Tests (15)	
✔ AllCountiesOK	1 ms
✔ CountPropertyOK	< 1 ms
✔ CountyExtremeMax	< 1 ms
✔ CountyMaxBoundary	< 1 ms

So where is the problem?

If we look at the code for clsCountyCollection we see the following auto generated code...



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

namespace Class_Library
{
    public class clsCountyCollection
    {
        public int Count { get; set; }

        public List<clsCounty> AllCounties { get; set; }
    }
}
```

Let's flesh this out to get the count property working correctly.

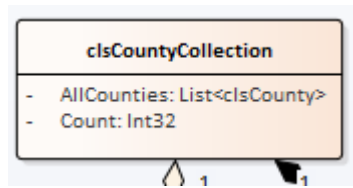
In object oriented programming we need to make sure our classes are correctly encapsulated.

What this means in English is that the inner workings of the object need to be hidden from other objects. Other objects will access the methods and properties of the object but have no access to the inner workings of the object.

To hide the inner workings of the class we use private data members to store the actual data.

The data in these private data members is made public via the public properties and methods of the class.

For the class we are creating clsCountyCollection we have two public properties



AllCounties & Count

In the code for the class we will need to create private data which does the job of actually storing the data for these properties.

The following code creates the first data member...

```
public class clsCountyCollection
{
    //private data member for the allCounties list
    private List<clsCounty> mAllCounties = new List<clsCounty>();
}
```

Note how we use the prefix “m” to remind us that it is a data member.

We now need to modify the auto generated code to make it does what we want it to do.

The first step is to link the public property AllCounties to the private data member allCounties like so...

```
//public property for Count
public int Count
{
    get
    {
        //return the count property of the private list
        return mAllCounties.Count;
    }
    set
    {
        //we will look at this in a moment!
    }
}

//public property for allCounties
public List<clsCounty> AllCounties
{
    //getter sends data to requesting code
    get
    {
        //return the private data member
        return mAllCounties;
    }
    //setter accepts data from other objects
    set
    {
        //assign the incoming value to the private data member
        mAllCounties = value;
    }
}
```

Run the tests and you should see no improvement from before. However it shouldn't be any worse!

```

//public property for Count
public int Count
{
    get
    {
        //return the count property of the private list
        return mAllCounties.Count;
    }
    set
    {
        //we will look at this in a moment!
    }
}

//public property for allCounties
public List<clsCounty> AllCounties
{
    //getter sends data to requesting code
    get
    {
        //return the private data member
        return mAllCounties;
    }
    //setter accepts data from other objects
    set
    {
        //assign the incoming value to the private data member
        mAllCounties = value;
    }
}

```

Notice how we simply return the Count property of the private list mAllCounties.Count.

This means that however many items there are in the list the count property should always report back the correct value.

Run the tests to see if everything passes.

There should be a problem.

The test CountMatchesList should work fine.

However the test CountPropertyOK is now failing.

This is going to happen a lot as we create half baked tests and refine them.

As our testing improves it will expose previous problems we hadn't spotted.

So what is the problem with CountPropertyOK?

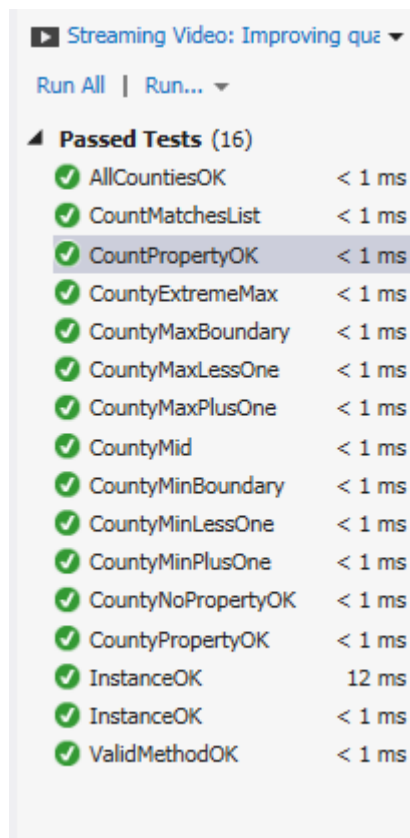
Let's look at the code.

```
[TestMethod]
public void CountPropertyOK()
{
    //create an instance of the class we want to create
    clsCountyCollection AllCounties = new clsCountyCollection();
    //create some test data to assign to the property
    Int32 SomeCount = 1;
    //assign the data to the property
    AllCounties.Count = SomeCount;
    //test to see that the two values are the same
    Assert.AreEqual(AllCounties.Count, SomeCount);
}
```

The issue with this test is that we are setting the value of SomeCount to 1. This assumes that there is one record in the list. The problem is that we are not adding any records in this test. This is good as it proves that the test framework is working correctly.

Change the code and all tests should now pass...

```
[TestMethod]
public void CountPropertyOK()
{
    //create an instance of the class we want to create
    clsCountyCollection AllCounties = new clsCountyCollection();
    //create some test data to assign to the property
    Int32 SomeCount = 0;
    //assign the data to the property
    AllCounties.Count = SomeCount;
    //test to see that the two values are the same
    Assert.AreEqual(AllCounties.Count, SomeCount);
}
```



We may be reasonably sure that the count property is working as planned but we still need to get that database connectivity working.

So where do we go to next?

We need to think a little about how this class is going to work when fully implemented. Based on the testing created so far we are creating a list of counties and assigning that list to the collection. This isn't quite right.

What we really want the collection class to do is to open the table in the database and populate the list automatically when we create a new instance of the class.

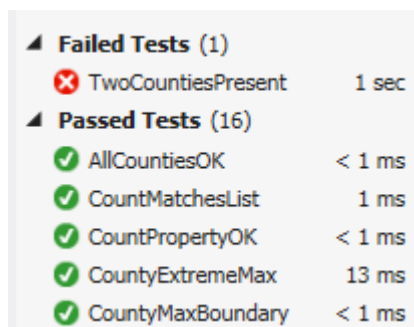
To make this work we need to access the constructor for the class and populate the list from there.

Let's create a new test to test if there are two records already present in the class upon creating an instance.

Create the following test...

```
[TestMethod]
public void TwoCountiesPresent()
{
    //create an instance of the class we want to create
    clsCountyCollection Counties = new clsCountyCollection();
    //test to see that the two values are the same
    Assert.AreEqual(Counties.Count, 2);
}
```

And watch it fail...



▲ Failed Tests (1)	
✖ TwoCountiesPresent	1 sec
▲ Passed Tests (16)	
✔ AllCountiesOK	< 1 ms
✔ CountMatchesList	1 ms
✔ CountPropertyOK	< 1 ms
✔ CountyExtremeMax	13 ms
✔ CountyMaxBoundary	< 1 ms

So let's think about the code for the test and what it does (or more importantly doesn't do!)

The first line creates an instance of the class called Counties

```
clsCountyCollection Counties = new clsCountyCollection();
```

The next test asks the question "are there two records?"

```
Assert.AreEqual(Counties.Count, 2);
```

OK- but the issue at this stage is that the records are not being created in this test. So where do we place the code that creates the two test records?

The answer is that we need to add a constructor to the class `clsCountyCollection` like so...

```
//public constructor for the class
public clsCountyCollection()
{
}
```

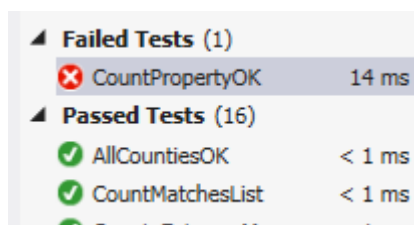
The constructor is a function that runs when the class is created. We place in the constructor code that is used to initialise the class upon instantiation. To create the constructor in C# we use the `public` keyword and then name the function so that it is the same as the name of the class.

The code for adding the two records needs to be written in this function.

The code to add the two test records is as follows...

```
//public constructor for the class
public clsCountyCollection()
{
    //create an instance of the county class to store a county
    clsCounty ACounty = new clsCounty();
    //set the county to Leicestershire
    ACounty.County = "Leicestershire";
    //add the county to the private list of counties
    mAllCounties.Add(ACounty);
    //re initialise the ACounty object to accept a new item
    ACounty = new clsCounty();
    //set the county to Cheshire
    ACounty.County = "Cheshire";
    //add the second county to the private list of counties
    mAllCounties.Add(ACounty);
    //the private list now contains two counties
}
```

If you run the test you will see that you have fixed one problem only to create a new problem...



The test for CountPropertyOK is now failing – why?

```
[TestMethod]
public void CountPropertyOK()
{
    //create an instance of the class we want to create
    clsCountyCollection AllCounties = new clsCountyCollection();
    //create some test data to assign to the property
    Int32 SomeCount = 0;
    //assign the data to the property
    AllCounties.Count = SomeCount;
    //test to see that the two values are the same
    Assert.AreEqual(AllCounties.Count, SomeCount);
}
```

This test is fine so long as there are no items being generated as the object is instantiated.

The problem we have now is that having added two records created in the constructor this test will fail as it assumes there are no records upon creation of the object.

We could fix the test like so...

```
[TestMethod]
public void CountPropertyOK()
{
    //create an instance of the class we want to create
    clsCountyCollection AllCounties = new clsCountyCollection();
    //create some test data to assign to the property
    Int32 SomeCount = 2;
    //assign the data to the property
    AllCounties.Count = SomeCount;
    //test to see that the two values are the same
    Assert.AreEqual(AllCounties.Count, SomeCount);
}
```

Making the test data to 2 rather than 0!

(Once the data is coming from the database we are going to face a new problem.)

Run all tests to make sure everything is working correctly.

We now have a collection class which auto populates with two test records with a working count property.

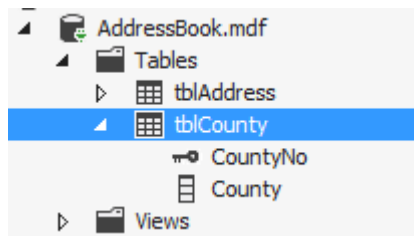
The next step is to make the leap to connecting to the database.

The code we have in our constructor is ok for test purposes however so far as the finished system goes it is a bit rubbish.

```
//public constructor for the class
public clsCountyCollection()
{
    //create an instance of the county class to store a county
    clsCounty ACounty = new clsCounty();
    //set the county to Leicestershire
    ACounty.County = "Leicestershire";
    //add the county to the private list of counties
    mAllCounties.Add(ACounty);
    //re initialise the ACounty object to accept a new item
    ACounty = new clsCounty();
    //set the county to Cheshire
    ACounty.County = "Cheshire";
    //add the second county to the private list of counties
    mAllCounties.Add(ACounty);
    //the private list now contains two counties
}
```

There are far more counties in the UK rather than just two. (Depending on the status of Rutland there is a fairly static amount.)

We shall assume in this example that we have the database already created and it contains a table called tblCounty



The table contains a list of UK counties...

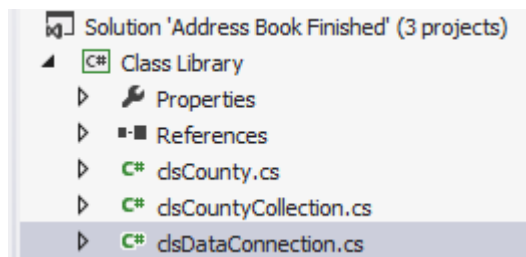
	CountyNo	County
▶	1	Avon
	2	Bedfordshire
	3	Berkshire
	4	Buckinghamshire
	5	Cambridgeshire
	6	Cambridgeshire ...
	7	Cheshire
	8	City of Bristol
	9	City of London
	10	Cleveland
	11	Cornwall
	12	Cumberland
	13	Cumbria
	14	Derbyshire
	15	Devon
	16	Dorset
	17	Durham
	18	East Suffolk
	19	East Sussex
	20	Essex
	21	Gloucestershire
	22	Greater London
	23	Greater Manche...
	24	Hampshire (Cou...
	25	Hereford and W...
	26	Herefordshire
	27	Hertfordshire
	28	Humberside
	29	Huntingdon and ...
	30	Huntingdonshire
	31	Tale of Elv

We will also assume that we have a stored procedure called `sproc_tblCounty_SelectAll` which contains the following SQL...

```
CREATE PROCEDURE sproc_tblCounty_SelectAll
AS
--select all records from tblCounty
select * from tblCounty
```

So what we want to do is re-factor the code for the constructor such that it links to and draws data from this stored procedure.

The first thing we need to do is set up the class clsDataConnection in the class library...



You will need to make sure that the namespace is set up correctly...

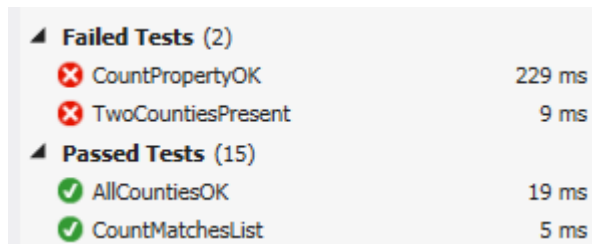
```
//this class uses the adonet sql classes to pr
//it is free for use by anybody so long as you
//Matthew Dean mjdean@dmu.ac.uk De Montfort Uni

namespace Class_Library
{
    public class clsDataConnection
    {
        //connection object used to connect to t
        SqlConnection connectionToDB = new SqlCo
        //data adapter used to transfer data to
        SqlDataAdapter dataAdapter = new SqlData
```

The new code for the constructor of clsCountyCollection is...

```
//public constructor for the class
public clsCountyCollection()
{
    //create an instance of the dataconnection
    clsDataConnection DB = new clsDataConnection();
    //execute the stored procedure to get the list of data
    DB.Execute("sproc_tblCounty_SelectAll");
    //get the count of records
    Int32 RecordCount = DB.Count;
    //set up the index for the loop
    Int32 Index = 0;
    //while there are records to process
    while (Index < RecordCount)
    {
        //create a new instance of the county class
        clsCounty ACounty = new clsCounty();
        //get the county name
        ACounty.County = DB.DataTable.Rows[Index]["County"].ToString();
        //get the primary key
        ACounty.CountyNo = Convert.ToInt32(DB.DataTable.Rows[Index]["CountyNo"]);
        //add the county to the private data member
        mAllCounties.Add(ACounty);
        //increment the index
        Index++;
    }
}
```

If everything has gone to plan you should have two failing tests...



▲ Failed Tests (2)	
✗ CountPropertyOK	229 ms
✗ TwoCountiesPresent	9 ms
▲ Passed Tests (15)	
✓ AllCountiesOK	19 ms
✓ CountMatchesList	5 ms

So now what is the problem?

The problem is that we no longer have two test records hard coded into the constructor we now have 72 records actually stored in the database.

The test for the count property needs to be modified like so...

```
[TestMethod]
public void CountPropertyOK()
{
    //create an instance of the class we want to create
    clsCountyCollection AllCounties = new clsCountyCollection();
    //create some test data to assign to the property
    Int32 SomeCount = 72;
    //assign the data to the property
    AllCounties.Count = SomeCount;
    //test to see that the two values are the same
    Assert.AreEqual(AllCounties.Count, SomeCount);
}
```

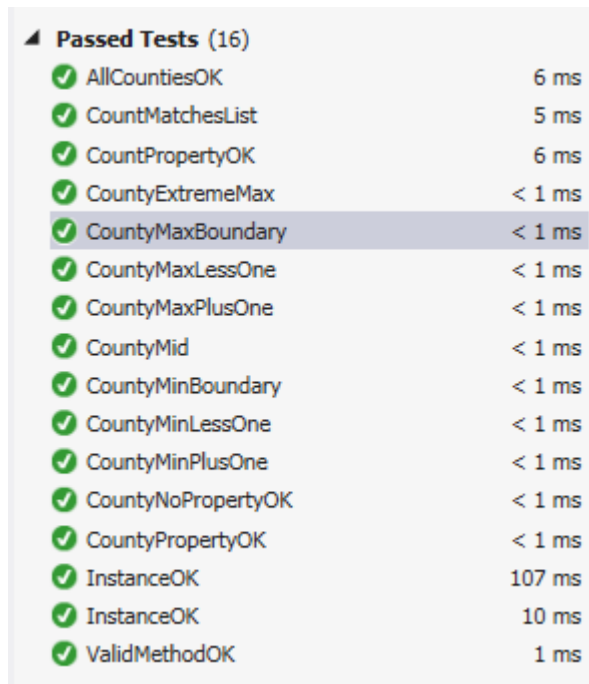
Clearly if Rutland becomes part of Leicestershire or a new county is declared at some point this test will fail (but it will do for now!)

Lastly the following test is actually redundant...

```
[TestMethod]
public void TwoCountiesPresent()
{
    //create an instance of the class we want to create
    clsCountyCollection Counties = new clsCountyCollection();
    //test to see that the two values are the same
    Assert.AreEqual(Counties.Count, 2);
}
```

Rather than deleting it though highlight the code and press Ctrl – K – C to comment the block out in case you do need it later. (Ctrl – K – U reverses the block commenting.)

Run your tests and they all should pass...



A screenshot of a test runner interface showing a list of 16 passed tests. Each test is preceded by a green checkmark icon. The tests are listed in a single column, and their execution times are shown in milliseconds to the right of each test name. The test 'CountyMaxBoundary' is highlighted with a light blue background.

Passed Tests (16)	
✓ AllCountiesOK	6 ms
✓ CountMatchesList	5 ms
✓ CountPropertyOK	6 ms
✓ CountyExtremeMax	< 1 ms
✓ CountyMaxBoundary	< 1 ms
✓ CountyMaxLessOne	< 1 ms
✓ CountyMaxPlusOne	< 1 ms
✓ CountyMid	< 1 ms
✓ CountyMinBoundary	< 1 ms
✓ CountyMinLessOne	< 1 ms
✓ CountyMinPlusOne	< 1 ms
✓ CountyNoPropertyOK	< 1 ms
✓ CountyPropertyOK	< 1 ms
✓ InstanceOK	107 ms
✓ InstanceOK	10 ms
✓ ValidMethodOK	1 ms

Having created the two classes to make the collection class work correctly we shall now add the functionality to the smoke and mirrors prototype.

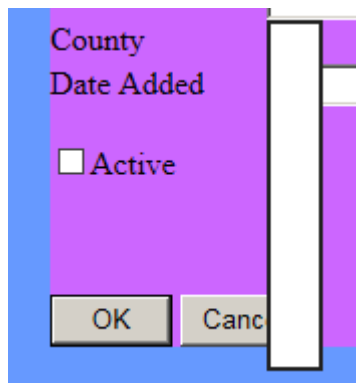
If we run the prototype we see the following main page...

The main page of the prototype features a dark red header bar at the top. The main content area has a light purple background. On the left side, there is a vertical blue bar. In the center of the purple area, there is a large, empty white rectangular box. Below this box, the text "Please Enter a Post Code" is displayed in a dark purple font. Underneath the text is a white input field. Below the input field, there are two buttons: "Apply" and "Display All". At the bottom of the purple area, there are three buttons: "Add", "Edit", and "Delete".

Pressing Add takes us to the following page...

The form page for adding a new record has a dark red header bar at the top. The main content area has a light purple background. On the left side, there is a vertical blue bar. The form consists of several fields: "House No" with a white input field, "Street" with a white input field, "Town" with a white input field, "Post Code" with a white input field, "County" with a dropdown menu showing a downward arrow, and "Date Added" with a white input field. Below these fields is a checkbox labeled "Active". At the bottom of the purple area, there are two buttons: "OK" and "Cancel".

The bit that we want to make work is the drop down list of Counties...



Access the code for the form AnAddress.aspx and create a new function called DisplayCounties ...

```
//function for populating the county drop down list
void DisplayCounties()
{
}
}
```

Inside the function create the following code to populate the drop down list with data...

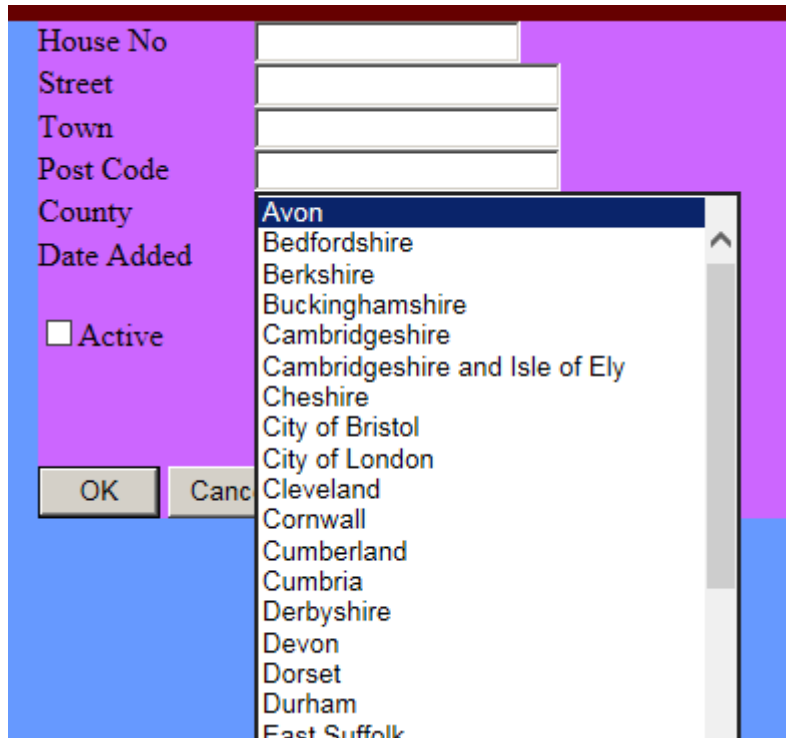
```
//function for populating the county drop down list
void DisplayCounties()
{
    //create an instance of the County Collection
    Class_Library.clsCountyCollection Counties = new Class_Library.clsCountyCollection();
    //set the data source to the list of counties in the collection
    ddlCounty.DataSource = Counties.AllCounties;
    //set the name of the primary key
    ddlCounty.DataValueField = "CountyNo";
    //set the data field to display
    ddlCounty.DataTextField = "County";
    //bind the data to the list
    ddlCounty.DataBind();
}
}
```

Next make a call to the new function in the load event of the form...

```
//event handler for the page load event
protected void Page_Load(object sender, EventArgs e)
{
    if (IsPostBack == false)
    {
        DisplayCounties();
    }
}
}
```

Notice how we use `IsPostBack` to check if this is the first time the page is displayed. If this is not the first time the page is displayed i.e. a post back we don't want to refresh the drop down list.

If you run the form now you should see the drop down list populated with the data from the database.



A screenshot of a web form with a purple background. The form contains several text input fields labeled 'House No', 'Street', 'Town', 'Post Code', and 'County'. Below these is a checkbox labeled 'Active'. At the bottom are 'OK' and 'Cancel' buttons. A dropdown menu is open, showing a list of counties: Avon, Bedfordshire, Berkshire, Buckinghamshire, Cambridgeshire, Cambridgeshire and Isle of Ely, Cheshire, City of Bristol, City of London, Cleveland, Cornwall, Cumberland, Cumbria, Derbyshire, Devon, Dorset, Durham, and East Suffolk. The 'Avon' option is currently selected and highlighted in blue.

It is worth spending a bit of time to think about what has been achieved here.

We have:

- Taken a class diagram
- Created classes from that diagram
- Implemented methods and properties
- Linked the code to the database
- Linked the resulting code to the presentation layer
- Built a test framework allowing us to be reasonably confident that the code is correct

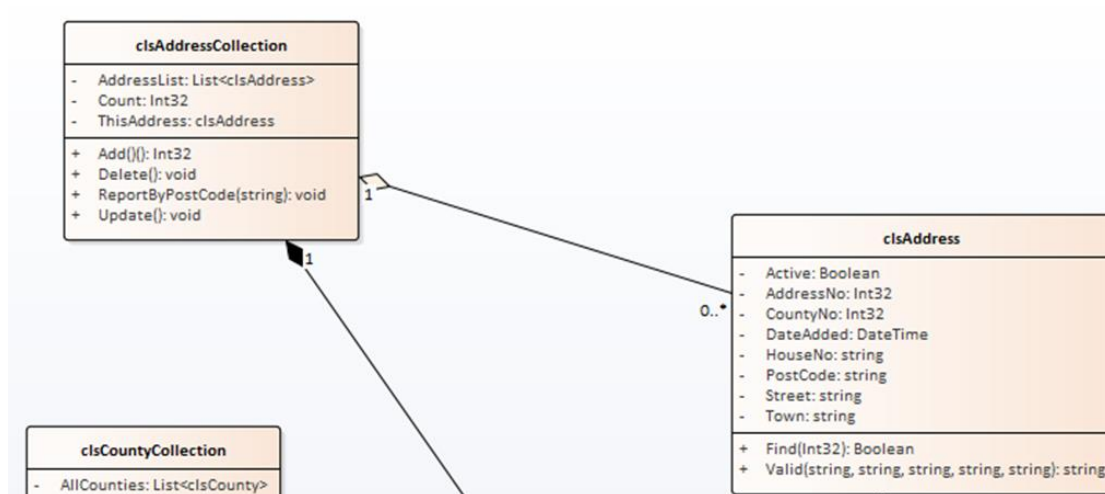
Pathway 3 – Creating the Complex Item Class

So where do we go next?

We will go once more round the loop of selecting classes from the diagram and linking presentation and data layers.

This time we will do something very similar but a little more complicated.

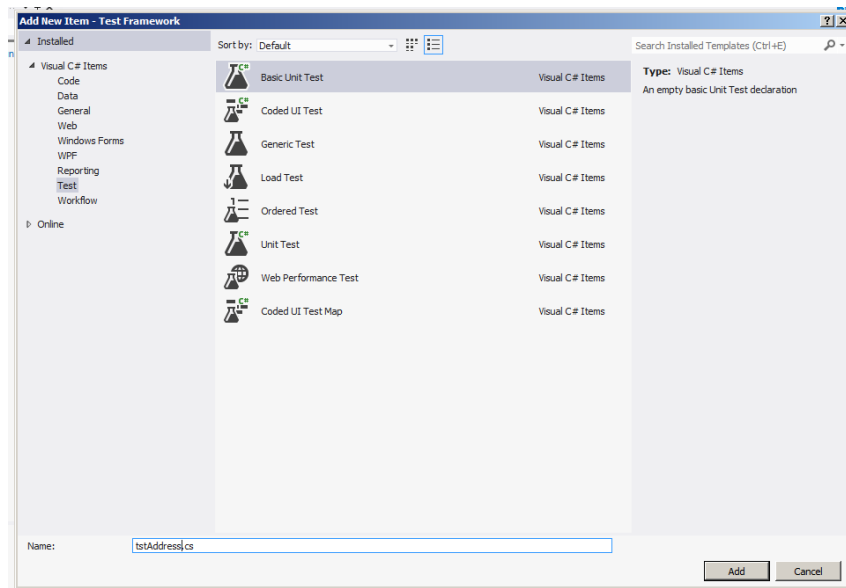
We will implement the class pair clsAddressCollection and clsAddress



As above let's start with something simple. Since **clsAddressCollection** uses **clsAddress** we will start with **clsAddress**.

As above we will start with the most basic test i.e. creating an instance of the required class.

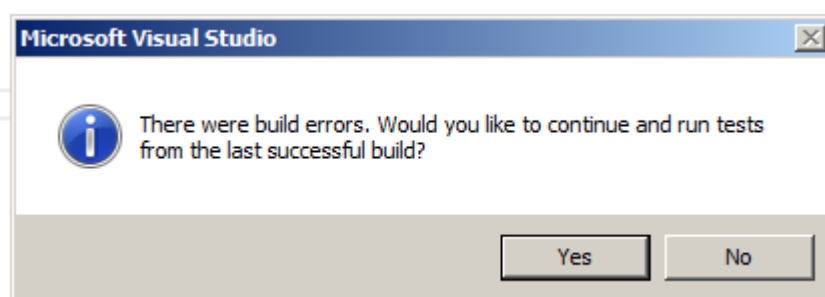
So create the test class...



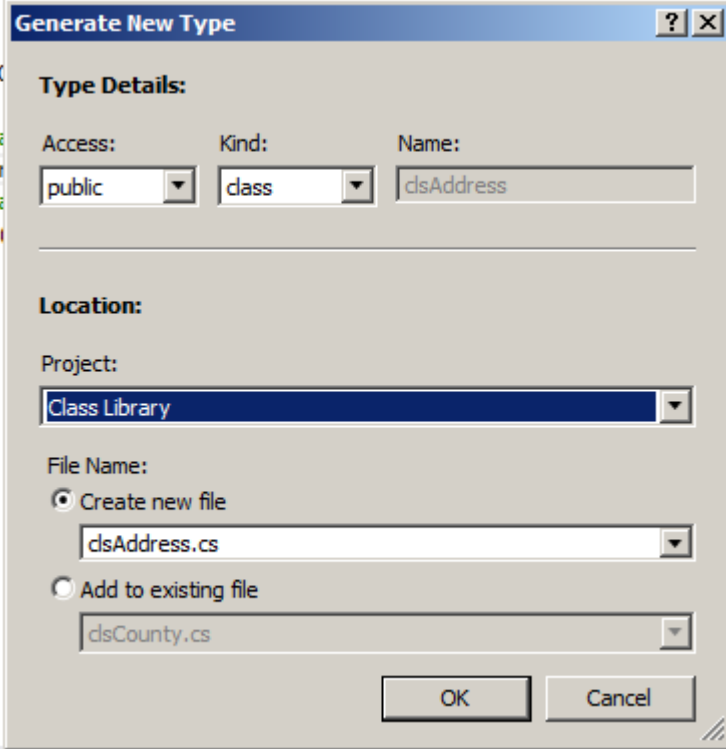
Next create the test for instantiation...

```
[TestMethod]
public void InstanceOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //test to see that it exists
    Assert.IsNotNull(AnAddress);
}
```

Then watch it fail...



Fix the problem...



The 'Generate New Type' dialog box is shown. It has a title bar with a question mark and a close button. The 'Type Details' section contains three fields: 'Access:' with a dropdown set to 'public', 'Kind:' with a dropdown set to 'class', and 'Name:' with a text box containing 'dsAddress'. The 'Location:' section contains a 'Project:' dropdown set to 'Class Library'. Below this, the 'File Name:' section has two radio buttons: 'Create new file' (selected) and 'Add to existing file'. Under 'Create new file' is a text box with 'dsAddress.cs'. Under 'Add to existing file' is a text box with 'dsCounty.cs'. At the bottom are 'OK' and 'Cancel' buttons.

Section	Field	Value
Type Details	Access:	public
	Kind:	class
	Name:	dsAddress
Location	Project:	Class Library
	File Name:	dsAddress.cs (selected)

See the test pass...

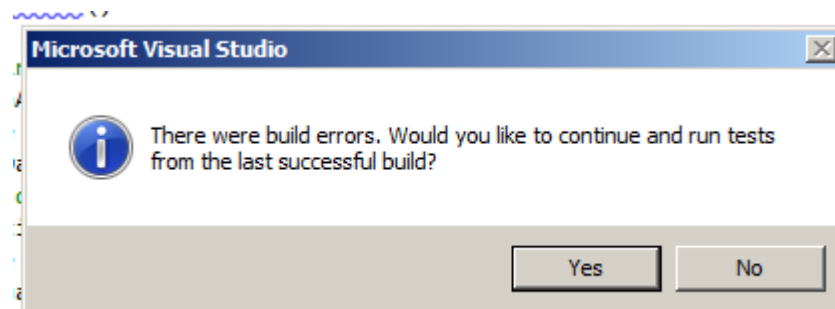
Passed Tests (17)		
✓	AllCountiesOK	9 ms
✓	CountMatchesList	8 ms
✓	CountPropertyOK	53 ms
✓	CountyExtremeMax	< 1 ms
✓	CountyMaxBoundary	< 1 ms
✓	CountyMaxLessOne	< 1 ms
✓	CountyMaxPlusOne	< 1 ms
✓	CountyMid	< 1 ms
✓	CountyMinBoundary	< 1 ms
✓	CountyMinLessOne	< 1 ms
✓	CountyMinPlusOne	< 1 ms
✓	CountyNoPropertyOK	2 ms
✓	CountyPropertyOK	18 ms
✓	InstanceOK	40 ms
✓	InstanceOK	14 sec
✓	InstanceOK	< 1 ms
✓	ValidMethodOK	< 1 ms

Creating the Properties

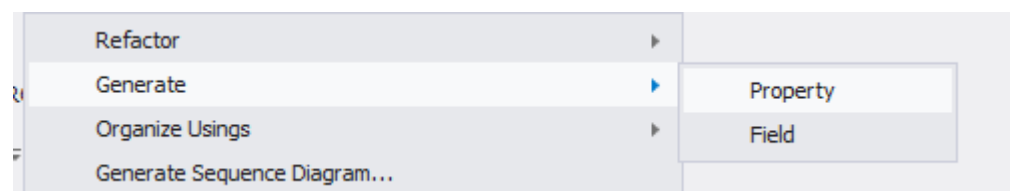
Create a test for the first property...

```
[TestMethod]
public void ActivePropertyOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //create some test data to assign to the property
    Boolean TestData = true;
    //assign the data to the property
    AnAddress.Active = TestData;
    //test to see that the two values are the same
    Assert.AreEqual(AnAddress.Active, TestData);
}
```

Watch it fail...



Fix the problem...



Make sure the test passes...

Passed Tests (18)		
✓	ActivePropertyOK	1 ms
✓	AllCountiesOK	22 ms
✓	CountMatchesList	5 ms

The next property we shall look at is DateAdded as it allows us to look at some useful code.

Take a look at the test code for this date property...

```
[TestMethod]
public void DateAddedPropertyOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //create some test data to assign to the property
    DateTime TestData = DateTime.Now.Date;
    //assign the data to the property
    AnAddress.DateAdded = TestData;
    //test to see that the two values are the same
    Assert.AreEqual(AnAddress.DateAdded, TestData);
}
```

The thing about date tests is that we don't want to be updating the date every time we run the test.

The following line of code gets round this problem.

```
DateTime TestData = DateTime.Now.Date;
```

What it does is assign the variable TestData with today's date taken from DateTime.Now.Date

You will need to create the rest of the tests such that all of the properties are tested and created.

The code for the test class for all properties is as follows...

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

namespace Test_Framework
{
    [TestClass]
    public class tstAddress
    {
        [TestMethod]
        public void InstanceOK()
        {
            //create an instance of the class we want to create
            clsAddress AnAddress = new clsAddress();
            //test to see that it exists
            Assert.IsNotNull(AnAddress);
        }

        [TestMethod]
        public void ActivePropertyOK()
        {
            //create an instance of the class we want to create
            clsAddress AnAddress = new clsAddress();
            //create some test data to assign to the property
            Boolean TestData = true;
        }
    }
}
```

```

        //assign the data to the property
        AnAddress.Active = TestData;
        //test to see that the two values are the same
        Assert.AreEqual(AnAddress.Active, TestData);
    }

    [TestMethod]
    public void DateAddedPropertyOK()
    {
        //create an instance of the class we want to create
        clsAddress AnAddress = new clsAddress();
        //create some test data to assign to the property
        DateTime TestData = DateTime.Now.Date;
        //assign the data to the property
        AnAddress.DateAdded = TestData;
        //test to see that the two values are the same
        Assert.AreEqual(AnAddress.DateAdded, TestData);
    }

    [TestMethod]
    public void AddressNoPropertyOK()
    {
        //create an instance of the class we want to create
        clsAddress AnAddress = new clsAddress();
        //create some test data to assign to the property
        Int32 TestData = 1;
        //assign the data to the property
        AnAddress.AddressNo = TestData;
        //test to see that the two values are the same
        Assert.AreEqual(AnAddress.AddressNo, TestData);
    }

    [TestMethod]
    public void CountyNoPropertyOK()
    {
        //create an instance of the class we want to create
        clsAddress AnAddress = new clsAddress();
        //create some test data to assign to the property
        Int32 TestData = 1;
        //assign the data to the property
        AnAddress.CountyNo = TestData;
        //test to see that the two values are the same
        Assert.AreEqual(AnAddress.CountyNo, TestData);
    }

    [TestMethod]
    public void HouseNoPropertyOK()
    {
        //create an instance of the class we want to create
        clsAddress AnAddress = new clsAddress();
        //create some test data to assign to the property
        string TestData = "21b";
        //assign the data to the property
        AnAddress.HouseNo = TestData;
        //test to see that the two values are the same
        Assert.AreEqual(AnAddress.HouseNo, TestData);
    }

    [TestMethod]
    public void PostCodePropertyOK()
    {

```



```

        //create an instance of the class we want to create
        clsAddress AnAddress = new clsAddress();
        //create some test data to assign to the property
        string TestData = "LE1 4AB";
        //assign the data to the property
        AnAddress.PostCode = TestData;
        //test to see that the two values are the same
        Assert.AreEqual(AnAddress.PostCode, TestData);
    }

    [TestMethod]
    public void StreetPropertyOK()
    {
        //create an instance of the class we want to create
        clsAddress AnAddress = new clsAddress();
        //create some test data to assign to the property
        string TestData = "Some Street";
        //assign the data to the property
        AnAddress.Street = TestData;
        //test to see that the two values are the same
        Assert.AreEqual(AnAddress.Street, TestData);
    }

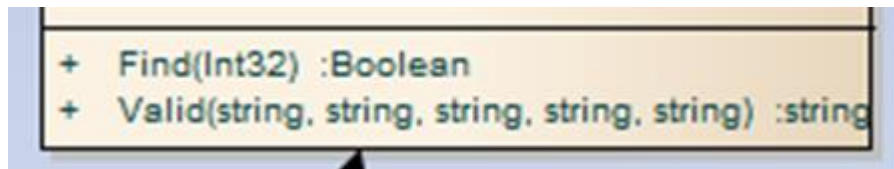
    [TestMethod]
    public void TownPropertyOK()
    {
        //create an instance of the class we want to create
        clsAddress AnAddress = new clsAddress();
        //create some test data to assign to the property
        string TestData = "Leicester";
        //assign the data to the property
        AnAddress.Town = TestData;
        //test to see that the two values are the same
        Assert.AreEqual(AnAddress.Town, TestData);
    }
}
}

```

Creating the Methods

Having created the test framework for the properties of `clsAddress` we need to think about how to approach building the methods for the class.

There are two methods in the class `Find` and `Valid`.



Let's have a go at `Find` first.

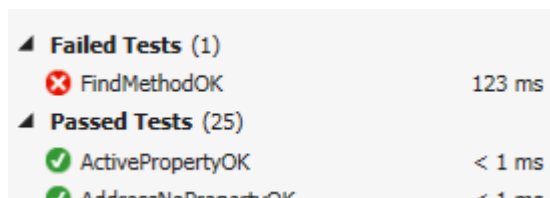
The simple things first, we need to create a test to ensure the existence of the method.

Like so...

```
[TestMethod]
public void FindMethodOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the validation
    Boolean Found = false;
    //create some test data to use with the method
    Int32 AddressNo = 1;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //test to see that the result is correct
    Assert.IsTrue(Found);
}
```

Run the test – watch it fail and then fix it by creating the method.

It should still fail due to the code in the class not being right.



One of the features of TDD is that we are able to put in a quick fix to problems simply to make the test pass.

The following code is an example of such a fix.

```
public bool Find(int AddressNo)
{
    //always return true
    return true;
}
```

Modifying the Find method on clsAddress to always return true forces the test to pass but doesn't exactly fix the problem in the long term.

We need to think a little more about how this method is going to work.

Let's assume that the table in the database contains the following data...

	AddressNo	HouseNo	Street	Town	PostCode	CountyNo	DateAdded	Active
	2	1	Some Street	Leicester	LE1 1BE	35	07/09/2012	True
	3	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
	4	33	High Street	Leicester	LE1 6FG	35	07/08/2012	True
	5	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True

How the method works is that we will send it the value of a record's primary key e.g. 2.

If record 2 is found the method will return a value of true, otherwise false.

As well as returning true or false the method also needs to return the data for all fields of the record being searched for.

For example

```
clsAddress AnAddress = new clsAddress();
AnAddress.Find(2);
string Street = AnAddress.Street;
```

Would set the value of the variable Street to "Some Street".

What we will do first is set up some test data in the table that will be used to check that the method is returning the data we think it should be.

Add a new record to the table like so...

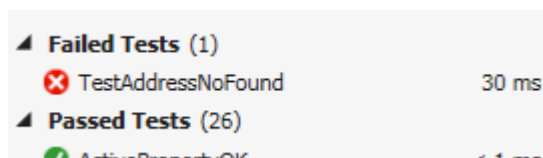
	AddressNo	HouseNo	Street	Town	PostCode	CountyNo	DateAdded	Active
	2	1	Some Street	Leicester	LE1 1BE	35	07/09/2012	True
	3	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
	4	33	High Street	Leicester	LE1 6FG	35	07/08/2012	True
	5	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
▶	21	123	Test Street	Test Town	XXX XXX	1	16/09/2015	True
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

We need to create a series of tests that search for the record and ensure that the correct data is being returned.

The first test is as follows...

```
[TestMethod]
public void TestAddressNoFound()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the search
    Boolean Found = false;
    //boolean variable to record if data is OK (assume it is)
    Boolean OK = true;
    //create some test data to use with the method
    Int32 AddressNo = 21;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //check the address no
    if (AnAddress.AddressNo != 21)
    {
        OK = false;
    }
    //test to see that the result is correct
    Assert.IsTrue(OK);
}
```

As usual the test should fail...



Let's fix the problem to see if we can resolve the failed test.

Open the code for clsAddress and add a private data member for the property...

```
public class clsAddress
{
    //private data member for the AddressNo property
    private Int32 mAddressNo;
    .. .. ..
```

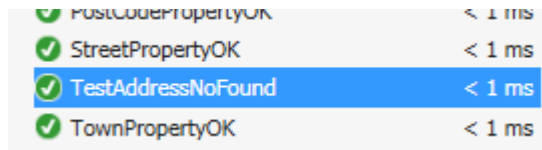
To make this work we will hard code the result into the Find method's function like so...

```
public bool Find(int AddressNo)
{
    //set the private data member to the test data value
    mAddressNo = 21;
    //always return true
    return true;
}
```

Next we need to modify the public AddressNo property so that it returns the value of this private data member...

```
//public property for the address number
public int AddressNo
{
    get
    {
        //return the private data
        return mAddressNo;
    }
    set
    {
        //set the value of the private data member
        mAddressNo = value;
    }
}
```

Let's try the test again...



The good news is that it passes. The bad news is that the fix is still a bit rubbish as it isn't drawing its data from the database.

At the moment we don't care about that!

Let's get the rest of the tests in place like so...

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

namespace Test_Framework
{
    [TestClass]
    public class tstAddress
    {
        [TestMethod]
        public void InstanceOK()
        {
            //create an instance of the class we want to create
            clsAddress AnAddress = new clsAddress();
            //test to see that it exists
            Assert.IsNotNull(AnAddress);
        }

        [TestMethod]
        public void ActivePropertyOK()
        {
            //create an instance of the class we want to create
            clsAddress AnAddress = new clsAddress();
            //create some test data to assign to the property
            Boolean TestData = true;
            //assign the data to the property
            AnAddress.Active = TestData;
            //test to see that the two values are the same
            Assert.AreEqual(AnAddress.Active, TestData);
        }

        [TestMethod]
        public void DateAddedPropertyOK()
        {
            //create an instance of the class we want to create
            clsAddress AnAddress = new clsAddress();
            //create some test data to assign to the property
            DateTime TestData = DateTime.Now.Date;
            //assign the data to the property
            AnAddress.DateAdded = TestData;
            //test to see that the two values are the same
            Assert.AreEqual(AnAddress.DateAdded, TestData);
        }
    }
}
```

```

[TestMethod]
public void AddressNoPropertyOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //create some test data to assign to the property
    Int32 TestData = 1;
    //assign the data to the property
    AnAddress.AddressNo = TestData;
    //test to see that the two values are the same
    Assert.AreEqual(AnAddress.AddressNo, TestData);
}

[TestMethod]
public void CountyNoPropertyOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //create some test data to assign to the property
    Int32 TestData = 1;
    //assign the data to the property
    AnAddress.CountyNo = TestData;
    //test to see that the two values are the same
    Assert.AreEqual(AnAddress.CountyNo, TestData);
}

[TestMethod]
public void HouseNoPropertyOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //create some test data to assign to the property
    string TestData = "21b";
    //assign the data to the property
    AnAddress.HouseNo = TestData;
    //test to see that the two values are the same
    Assert.AreEqual(AnAddress.HouseNo, TestData);
}

[TestMethod]
public void PostCodePropertyOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //create some test data to assign to the property
    string TestData = "LE1 4AB";
    //assign the data to the property
    AnAddress.PostCode = TestData;
    //test to see that the two values are the same
    Assert.AreEqual(AnAddress.PostCode, TestData);
}

```

```

[TestMethod]
public void StreetPropertyOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //create some test data to assign to the property
    string TestData = "Some Street";
    //assign the data to the property
    AnAddress.Street = TestData;
    //test to see that the two values are the same
    Assert.AreEqual(AnAddress.Street, TestData);
}

[TestMethod]
public void TownPropertyOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //create some test data to assign to the property
    string TestData = "Leicester";
    //assign the data to the property
    AnAddress.Town = TestData;
    //test to see that the two values are the same
    Assert.AreEqual(AnAddress.Town, TestData);
}

[TestMethod]
public void FindMethodOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the validation
    Boolean Found = false;
    //create some test data to use with the method
    Int32 AddressNo = 1;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //test to see that the result is correct
    Assert.IsTrue(Found);
}

```



```

[TestMethod]
public void TestAddressNotFound()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the search
    Boolean Found = false;
    //boolean variable to record if data is OK (assume it is)
    Boolean OK = true;
    //create some test data to use with the method
    Int32 AddressNo = 21;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //check the address no
    if (AnAddress.AddressNo != 21)
    {
        OK = false;
    }
    //test to see that the result is correct
    Assert.IsTrue(OK);
}

[TestMethod]
public void TestStreetFound()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the search
    Boolean Found = false;
    //boolean variable to record if data is OK (assume it is)
    Boolean OK = true;
    //create some test data to use with the method
    Int32 AddressNo = 21;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //check the property
    if (AnAddress.Street != "Test Street")
    {
        OK = false;
    }
    //test to see that the result is correct
    Assert.IsTrue(OK);
}

```

```

[TestMethod]
public void TestTownFound()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the search
    Boolean Found = false;
    //boolean variable to record if data is OK (assume it is)
    Boolean OK = true;
    //create some test data to use with the method
    Int32 AddressNo = 21;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //check the property
    if (AnAddress.Town != "Test Town")
    {
        OK = false;
    }
    //test to see that the result is correct
    Assert.IsTrue(OK);
}

[TestMethod]
public void TestPostCodeFound()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the search
    Boolean Found = false;
    //boolean variable to record if data is OK (assume it is)
    Boolean OK = true;
    //create some test data to use with the method
    Int32 AddressNo = 21;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //check the property
    if (AnAddress.PostCode != "XXX XXX")
    {
        OK = false;
    }
    //test to see that the result is correct
    Assert.IsTrue(OK);
}

```

```

[TestMethod]
public void TestCountyNotFound()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the search
    Boolean Found = false;
    //boolean variable to record if data is OK (assume it is)
    Boolean OK = true;
    //create some test data to use with the method
    Int32 AddressNo = 21;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //check the property
    if (AnAddress.CountyNo != 1)
    {
        OK = false;
    }
    //test to see that the result is correct
    Assert.IsTrue(OK);
}

[TestMethod]
public void TestDateAddedFound()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the search
    Boolean Found = false;
    //boolean variable to record if data is OK (assume it is)
    Boolean OK = true;
    //create some test data to use with the method
    Int32 AddressNo = 21;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //check the property
    if (AnAddress.DateAdded != Convert.ToDateTime("16/09/2015"))
    {
        OK = false;
    }
    //test to see that the result is correct
    Assert.IsTrue(OK);
}

```

```

[TestMethod]
public void TestActiveFound()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the search
    Boolean Found = false;
    //boolean variable to record if data is OK (assume it is)
    Boolean OK = true;
    //create some test data to use with the method
    Int32 AddressNo = 21;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //check the property
    if (AnAddress.Active != true)
    {
        OK = false;
    }
    //test to see that the result is correct
    Assert.IsTrue(OK);
}
}
}

```

Here is the code as it stands for the class clsAddress

```

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

namespace Class_Library
{
    public class clsAddress
    {
        //private data member for the AddressNo property
        private Int32 mAddressNo;
        //private data member for HouseNo
        private string mHouseNo;
        //private data member for street
        private string mStreet;
        //private data member for town
        private string mTown;
        //private data member for post code
        private string mPostCode;
        //private data member for county no
        private Int32 mCountyNo;
        //private date added data member
        private DateTime mDateAdded;
        //private data member for active
        private Boolean mActive;

        //public property for active
        public bool Active
        {
            get
            {
                //return the private data
                return mActive;
            }
            set
            {
            }
        }
    }
}

```

```

        {
            //set the private data
            mActive = value;
        }
    }

    //public property for date added
    public DateTime DateAdded
    {
        get
        {
            //return the private data
            return mDateAdded;
        }
        set
        {
            //set the private data
            mDateAdded = value;
        }
    }

    //public property for the address number
    public int AddressNo
    {
        get
        {
            //return the private data
            return mAddressNo;
        }
        set
        {
            //set the value of the private data member
            mAddressNo = value;
        }
    }

    //public property for county no
    public int CountyNo
    {
        get
        {
            //return the private data
            return mCountyNo;
        }
        set
        {
            //set the private data
            mCountyNo = value;
        }
    }

    //public property for house no
    public string HouseNo
    {
        get
        {
            //return private data
            return mHouseNo;
        }
        set
        {
            //set the private data

```

```

        mHouseNo = value;
    }
}

//public property for post code
public string PostCode
{
    get
    {
        //return the private data
        return mPostCode;
    }
    set
    {
        //set the private data
        mPostCode = value;
    }
}

//public data member for street
public string Street
{
    get
    {
        //return the private data
        return mStreet;
    }
    set
    {
        //set the private data
        mStreet = value;
    }
}

//public data member for Town
public string Town
{
    get
    {
        //return the private data
        return mTown;
    }
    set
    {
        //set the private data
        mTown = value;
    }
}

public bool Find(int AddressNo)
{
    //set the private data members to the test data value
    mAddressNo = 21;
    mHouseNo = "123";
    mStreet = "Test Street";
    mTown = "Test Town";
    mPostCode = "XXX XXX";
    mCountyNo = 1;
    mDateAdded = Convert.ToDateTime("16/9/2015");
    mActive = true;
    //always return true
    return true;
}

```

```

    }
}
}

```

At this stage all tests should pass however there is still one important feature that is missing.

That is we are still not actually drawing the data from the database!

The database contains the following data...

	AddressNo	HouseNo	Street	Town	PostCode	CountyNo	DateAdded	Active
	2	1	Some Street	Leicester	LE1 1BE	35	07/09/2012	True
	3	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
	4	33	High Street	Leicester	LE1 6FG	35	07/08/2012	True
	5	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
▶	21	123	Test Street	Test Town	XXX XXX	1	16/09/2015	True
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

And we will make use of the following stored procedure...

```

CREATE PROCEDURE sproc_tblAddress_FilterByAddressNo
--parameter to store the address no we are looking for
    @AddressNo int
AS
--select all fields from the table where the address no matches the parameter data
    select * from tblAddress where AddressNo = @AddressNo
RETURN 0

```

We now need to change the code for the Find method from this...

```
public bool Find(int AddressNo)
{
    //set the private data members to the test data value
    mAddressNo = 21;
    mHouseNo = "123";
    mStreet = "Test Street";
    mTown = "Test Town";
    mPostCode = "XXX XXX";
    mCountyNo = 1;
    mDateAdded = Convert.ToDateTime("16/9/2015");
    mActive = true;
    //always return true
    return true;
}
```

To this...

```
public bool Find(int AddressNo)
{
    //create an instance of the data connection
    clsDataConnection DB = new clsDataConnection();
    //add the parameter for the address no to search for
    DB.AddParameter("@AddressNo", AddressNo);
    //execute the stored procedure
    DB.Execute("sproc_tblAddress_FilterByAddressNo");
    //if one record is found (there should be either one or zero!)
    if (DB.Count == 1)
    {
        //copy the data from the database to the private data members
        mAddressNo = Convert.ToInt32(DB.DataTable.Rows[0]["AddressNo"]);
        mHouseNo = Convert.ToString(DB.DataTable.Rows[0]["HouseNo"]);
        mStreet = Convert.ToString(DB.DataTable.Rows[0]["Street"]);
        mTown = Convert.ToString(DB.DataTable.Rows[0]["Town"]);
        mPostCode = Convert.ToString(DB.DataTable.Rows[0]["PostCode"]);
        mCountyNo = Convert.ToInt32(DB.DataTable.Rows[0]["CountyNo"]);
        mDateAdded = Convert.ToDateTime(DB.DataTable.Rows[0]["DateAdded"]);
        mActive = Convert.ToBoolean(DB.DataTable.Rows[0]["Active"]);
        //return that everything worked OK
        return true;
    }
    //if no record was found
    else
    {
        //return false indicating a problem
        return false;
    }
}
```


You should get the following results when you run your tests...

Failed Tests (1)

FindMethodOK

108 ms

Passed Tests (32)

So what is the problem?

Double click the error to see the problem code...

```
[TestMethod]
public void FindMethodOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the validation
    Boolean Found = false;
    //create some test data to use with the method
    Int32 AddressNo = 1;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //test to see that the result is correct
    Assert.IsTrue(Found);
}
```

In this test we are testing the find method by searching for AddressNo 1.

The problem is that in the test data...

	AddressNo	HouseNo	Street	Town	PostCode	CountyNo	DateAdded	Active
	2	1	Some Street	Leicester	LE1 1BE	35	07/09/2012	True
	3	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
	4	33	High Street	Leicester	LE1 6FG	35	07/08/2012	True
	5	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
▶	21	123	Test Street	Test Town	XXX XXX	1	16/09/2015	True
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

There is no record 1!

Clearly we need to ensure that the test data we are using actually exists.

It would make more sense to change this test such that it tests for record 21 the primary key of the test record like so...

```
[TestMethod]
public void FindMethodOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //boolean variable to store the result of the validation
    Boolean Found = false;
    //create some test data to use with the method
    Int32 AddressNo = 21;
    //invoke the method
    Found = AnAddress.Find(AddressNo);
    //test to see that the result is correct
    Assert.IsTrue(Found);
}
```

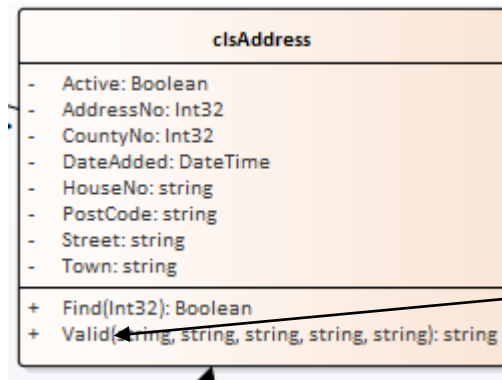
We should now have a decent chunk of functionality that is automatically tested by the test framework.

- clsCounty
 - County property
 - CountyNo property
 - Validation method
- clsCountyCollection
 - A list of counties
 - A count of items in the list
- clsAddress
 - All properties set up
 - A find method which sets all properties should the record be found

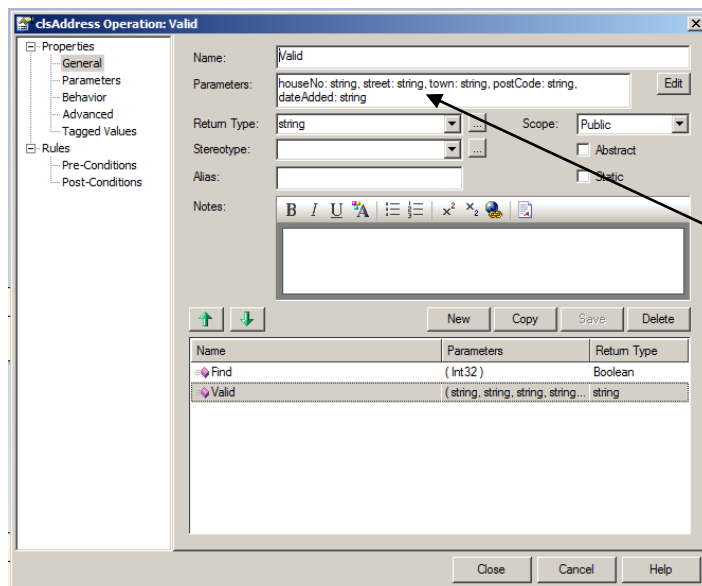
The last feature of clsAddress we shall set up is the method Valid. Once that is completed we shall move onto building clsAddressCollection.

Once the middle layer classes are complete we shall attach them to the presentation layer.

So the next stage is to look at the validation method for clsAddress.



If we examine the validation operation in Enterprise Architect we will see that it consists of several parameters – one for each property...



Note also that it has a return data type of string. The idea is that the function will return a concatenated message of what is wrong with the validated data.

There are other design issues to consider at this stage. As a personal preference I like to have my validation code in all one place. This being the case I need to provide a set of test data to satisfy the needs of the multiple parameters in that test method.

You may however prefer to create multiple validation functions, one for each item of test data. Currently this approach is not within the scope of this document.

There isn't a right answer but if you are taking the latter approach you will need to adapt the examples to suit your design.

(Notice we don't bother testing the primary key value AddressNo as this will be handled by the data layer! Also the Boolean property Active is currently ignored.)

As in all the above cases we need to make sure that the method actually exists.

The following test does this...

```
[TestMethod]
0 references
public void ValidMethodOK()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}
```

The first big point to note is the question of where are the variables for the Valid method being declared?

```
Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
```

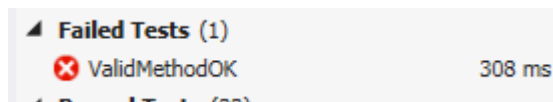
The answer is that we need to define the “good” test data once at the top of the test class giving it class level scope like so...

```
1  using System;
2  using Microsoft.VisualStudio.TestTools.UnitTesting;
3  using Class_Library;
4
5  namespace Test_Framework
6  {
7      [TestClass]
8      0 references
9      public class tstAddress
10     {
11         //good test data
12         //create some test data to pass to the method
13         string HouseNo = "12b";
14         string Street = "some street";
15         string Town = "Leicester";
16         string PostCode = "LE1 1AS";
17         string DateAdded = DateTime.Now.Date.ToString();
18     }
```

There are two benefits to doing this; firstly having the good test data with class level scope we make it available to all of the tests we are about to create, secondly if at a later date we decide to add a new property to the class then we add the new test data here once. If we don't make the test data declared once and we have two hundred tests, we are going to have to duplicate the same data two hundred times – too much like hard work for my tastes!

Also note how we are creating variables, one for each parameter. At this stage each variable should contain valid data.

Run the test – watch it fail – fix the test by creating the method. It should still fail!



Why?

Take a look at the auto generated code for the Valid method...

```
public string Valid(string houseNo, string street, string town, string postCode, string dateAdded)
{
    throw new NotImplementedException();
}
```

To force this test to pass we need to change the function like so...

```
public string Valid(string houseNo, string street, string town, string postCode, string dateAdded)
{
    return "";
}
```

The test should now pass.

The next stage is to create tests for each parameter such that we have a degree of confidence in the code for validation function.

The procedure we shall follow is...

- Decide on a single parameter to test
- Decide on what aspect we are testing – mid, min, max etc.
- Apply the specific test to the parameter whilst providing all of the other parameters with valid data

The first parameter we shall create the unit testing for is HouseNo.

A good place to start is with the test plan format used previously.

Description of Item to Be Tested:

--

Required Field **Y / N**

Test Type	Test Data	Expected Result	Actual Result
Extreme Min			
Min -1			
Min (Boundary)			
Min +1			
Max -1			
Max (Boundary)			
Max +1			
Mid			
Extreme Max			
Invalid data type			
Other tests			

Additional Notes / Instructions:

--

Using this with the data definition for the table we should be able to construct a series of tests for this parameter.

```
CREATE TABLE [dbo].[tblAddress] (  
    [AddressNo] INT IDENTITY (1, 1) NOT NULL,  
    [HouseNo] VARCHAR (6) NOT NULL,  
    [Street] VARCHAR (50) NOT NULL,  
    [Town] VARCHAR (50) NOT NULL,  
    [PostCode] VARCHAR (9) NOT NULL,  
    [CountyNo] INT NOT NULL,  
    [DateAdded] DATE NOT NULL,  
    [Active] BIT NULL,  
    PRIMARY KEY CLUSTERED ([AddressNo] ASC)  
);
```

HouseNo is defined in the database as having a maximum field length of 6 characters and may not be null/blank.

This gives us the following test plan...

Description of Item to Be Tested:

HouseNo field which stores the house number of the property. It may contain letters e.g. 15b.

Required Field **Y**

Test Type	Test Data	Expected Result	Actual Result
Extreme Min	NA		
Min -1	0 characters	Fail	
Min (Boundary)	1 character	Pass	
Min +1	2 characters	Pass	
Max -1	5 characters	Pass	
Max (Boundary)	6 characters	Pass	
Max +1	7 characters	Fail	
Mid	3 characters	Pass	
Extreme Max	500 characters	Fail	
Invalid data type			
Other tests			

Additional Notes / Instructions:

--

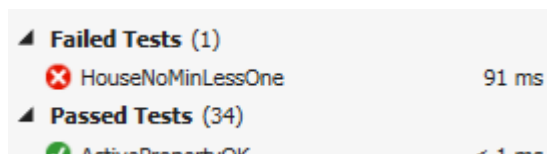
We are now in a position to create the first test for this parameter...

```
[TestMethod]
References
public void HouseNoMinLessOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create some test data to pass to the method
    string HouseNo = ""; //this should trigger an error
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}
```

Notice how we are now testing that there IS an error by Assert.IsFalse!

Remember that in testing the HouseNo we need to make sure that all of the other parameters have test data that passes. We know this is the case as we have declared the “good” test data at the top of the class. All we need to do in this and subsequent tests is override the “good” data for a single property with the test data we want to check.

The test should fail...



Let's create some code to trap the error correctly...

```
public string Valid(string houseNo, string street, string town, string postCode, string dateAdded)
{
    //create a string variable to store the error
    String Error = "";
    //if the HouseNo is blank
    if (houseNo.Length == 0)
    {
        //record the error
        Error = Error + "The house no may not be blank : ";
    }
    //return any error messages
    return Error;
}
```

Things to note!

1. Notice that the parameter names have automatically had their case set! If you test `HouseNo` rather than `houseNo` you will be testing your public property not the parameter!
2. Note the concatenation for the error message. The string has been formatted with an extra space at the end! This means that when the next message is concatenated it is a bit easier to read.

The test should now pass.

The following tests should all pass with the validation code as it currently stands.

```
public void HouseNoMin()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create some test data to pass to the method
    string HouseNo = "a"; //this should be ok
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void HouseNoMinPlusOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create some test data to pass to the method
    string HouseNo = "aa"; //this should be ok
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void HouseNoMaxLessOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create some test data to pass to the method
    string HouseNo = "aaaaa"; //this should be ok
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}
```

```

[TestMethod]
public void HouseNoMax()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create some test data to pass to the method
    string HouseNo = "aaaaaa"; //this should be ok
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void HouseNoMid()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create some test data to pass to the method
    string HouseNo = "aaa"; //this should be ok
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

```

The last couple of tests are worth paying a little attention to.

Firstly the max + 1 test should fail as the validation code doesn't address it...

```
[TestMethod]
0 references
public void HouseNoMaxPlusOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create some test data to pass to the method
    string HouseNo = "aaaaaaa"; //this should fail
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}
```

▲ Failed Tests (1)

✖ HouseNoMaxPlusOne 2 sec

▲ Passed Tests (40)

Modifying the validation function like so will deal with the problem...

```
public string Valid(string houseNo, string street, string town, string postCode, string dateAdded)
{
    //create a string variable to store the error
    String Error = "";
    //if the HouseNo is blank
    if (houseNo.Length == 0)
    {
        //record the error
        Error = Error + "The house no may not be blank : ";
    }
    //if the house no is greater than 6 characters
    if (houseNo.Length > 6)
    {
        //record the error
        Error = Error + "The house no must be less than 6 characters : ";
    }
    //return any error messages
    return Error;
}
```

The last test to pay a little attention to is the extreme max test...

```
[TestMethod]
References
public void HouseNoExtremeMax()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create some test data to pass to the method
    string HouseNo = "";
    HouseNo = HouseNo.PadRight(500, 'a'); //this should fail
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}
```

If you get excited about typing 500 characters into a computer then please don't let me stop you. However it makes far more sense to make use of the PadRight method.

One parameter down - four more to go!

We shall concentrate on the DateAdded parameter next, there are a few useful coding tricks here.

Below is the test plan for the DateAdded parameter.

Description of Item to Be Tested:

DateAdded. This value contains the date on which the record is added to the database. The value may not be less than or greater than today's date. It must be set to a value and may not be blank.

Required Field Y / N

Test Type	Test Data	Expected Result	Actual Result
Extreme Min	Today's date less 100 years		
Min -1	Yesterday's date		
Min (Boundary)	Today's date		
Min +1	Tomorrow's date		
Max -1	NA (Same as min less 1)		
Max (Boundary)	NA (Same as min)		
Max +1	NA (Same as min + 1)		
Mid	NA (The date must be today)		
Extreme Max	Today's date plus 100 years		
Invalid data type	Any non date data		
Other tests	NA		

Additional Notes / Instructions:

--

The issue to consider when creating test data for dates is that the test data will change depending on what day we enter the test data.

We need to design the test methods in such a way that this date sensitivity is taken into account.

Here is the first test - for extreme min...

```
[TestMethod]
0 references
public void DateAddedExtremeMin()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create a variable to store the test date data
    DateTime TestDate;
    //set the date totodays date
    TestDate = DateTime.Now.Date;
    //change the date to whatever the date is less 100 years
    TestDate = TestDate.AddYears(-100);
    //convert the date variable to a string variable
    string DateAdded = TestDate.ToString();
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}
```

Notice how we are using a variable of data type DateTime to create the test data...

First we declare the variable...

```
//create a variable to store the test date data
DateTime TestDate;
```

Next we set it to today's date....

```
//set the date totodays date
TestDate = DateTime.Now.Date;
```

Next we set it to today's date less 100 years using the AddYears method...

```
//change the date to whatever the date is less 100 years
TestDate = TestDate.AddYears(-100);
```


Lastly we copy the test data to our string variable to be passed to the validation function...

```
//convert the date variable to a string variable
string DateAdded = TestDate.ToString();
```

The big advantage of using the DateTime data type is that it gives us a whole load of methods for date and time manipulation that the string data type doesn't give us.

If we run the test it should fail...



Which means we need to add some code to the validation method like so...

```
public string Valid(string houseNo, string street, string town, string postCode, string dateAdded)
{
    //create a string variable to store the error
    String Error = "";
    //create a temporary variable to store date values
    DateTime DateTemp;
    //if the HouseNo is blank
    if (houseNo.Length == 0)
    {
        //record the error
        Error = Error + "The house no may not be blank : ";
    }
    //if the house no is greater than 6 characters
    if (houseNo.Length > 6)
    {
        //record the error
        Error = Error + "The house no must be less than 6 characters : ";
    }
    //copy the dateAdded value to the DateTemp variable
    DateTemp = Convert.ToDateTime(dateAdded);
    if (DateTemp < DateTime.Now.Date)
    {
        //record the error
        Error = Error + "The date cannot be in the past : ";
    }
    //return any error messages
    return Error;
}
```

Run the test and it should pass.

Notice in the validation function how we use the variable TempDate.

The reason we have this variable is so that we may take advantage of the DateTime data type's date/time methods.

e.g....

```
//check to see if the date is less than today's date
if (DateTemp < DateTime.Now.Date)
{
```

Here are the all of the boundary tests for DateAdded...

```
[TestMethod]
public void DateAddedExtremeMin()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create a variable to store the test date data
    DateTime TestDate;
    //set the date totodays date
    TestDate = DateTime.Now.Date;
    //change the date to whatever the date is less 100 years
    TestDate = TestDate.AddYears(-100);
    //convert the date variable to a string variable
    string DateAdded = TestDate.ToString();
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}
```

```

[TestMethod]
public void DateAddedMinLessOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create a variable to store the test date data
    DateTime TestDate;
    //set the date to todays date
    TestDate = DateTime.Now.Date;
    //change the date to whatever the date is less 1 day
    TestDate = TestDate.AddDays(-1);
    //convert the date variable to a string variable
    string DateAdded = TestDate.ToString();
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}

[TestMethod]
public void DateAddedMin()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create a variable to store the test date data
    DateTime TestDate;
    //set the date to todays date
    TestDate = DateTime.Now.Date;
    //convert the date variable to a string variable
    string DateAdded = TestDate.ToString();
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

```

```

[TestMethod]
public void DateAddedMinPlusOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create a variable to store the test date data
    DateTime TestDate;
    //set the date to todays date
    TestDate = DateTime.Now.Date;
    //change the date to whatever the date is plus 1 day
    TestDate = TestDate.AddDays(1);
    //convert the date variable to a string variable
    string DateAdded = TestDate.ToString();
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}

[TestMethod]
public void DateAddedExtremeMax()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create a variable to store the test date data
    DateTime TestDate;
    //set the date to todays date
    TestDate = DateTime.Now.Date;
    //change the date to whatever the date is plus 100 years
    TestDate = TestDate.AddYears(100);
    //convert the date variable to a string variable
    string DateAdded = TestDate.ToString();
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}

```

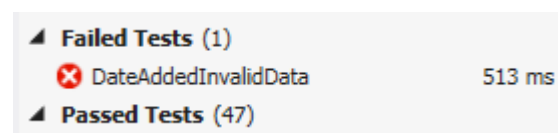
Here is the validation code to deal with these tests...

```
public string Valid(string houseNo, string street, string town, string postCode, string dateAdded)
{
    //create a string variable to store the error
    String Error = "";
    //create a temporary variable to store date values
    DateTime DateTemp;
    //if the HouseNo is blank
    if (houseNo.Length == 0)
    {
        //record the error
        Error = Error + "The house no may not be blank : ";
    }
    //if the house no is greater than 6 characters
    if (houseNo.Length > 6)
    {
        //record the error
        Error = Error + "The house no must be less than 6 characters : ";
    }
    //copy the dateAdded value to the DateTemp variable
    DateTemp = Convert.ToDateTime(dateAdded);
    if (DateTemp < DateTime.Now.Date)
    {
        //record the error
        Error = Error + "The date cannot be in the past : ";
    }
    //check to see if the date is greater than today's date
    if (DateTemp > DateTime.Now.Date)
    {
        //record the error
        Error = Error + "The date cannot be in the future : ";
    }
    //return any error messages
    return Error;
}
```

The final test we will look at is the test for invalid data...

```
[TestMethod]
0 references
public void DateAddedInvalidData()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //set the DateAdded to a non date value
    string DateAdded = "this is not a date!";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}
```

As usual the test should fail...



To solve this problem we will use the debugger.

Press F9 to generate a break point like so...

```
[TestMethod]
❌ | 0 references
public void DateAddedInvalidData()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //create some test data to pass to the method
    string HouseNo = "32a";
    string Street = "some street";
    string Town = "Leicester";
    string PostCode = "LE1 1AS";
    //set the DateAdded to a non date value
    string DateAdded = "this is not a date!";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}
```

Now right click on the test and select debug tests...

```







[TestMethod]
0 references
public void DateAddedInvalidData()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //set the DateAdded to a non date value
    string DateAdded = "this is not a date!";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, St
    //test to see that the result is co
    Assert.AreNotEqual(Error, "");
}

```

```

[TestMethod]
0 references
public void PostCodeMinLessOne()
{
    //create an instance of the class w
    clsAddress AnAddress = new clsAddre
    //string variable to store any erro
    String Error = "";
    //create some test data to pass to
    string HouseNo = "32a";
    string Street = "some street";
    string Town = "Leicester";
}

```

	Quick Actions and Refactorings...	Ctrl+.
	Rename...	Ctrl+R, Ctrl+R
	Remove and Sort Usings	Ctrl+R, Ctrl+G
	Peek Definition	Alt+F12
	Go To Definition	F12
	Go To Implementation	Ctrl+F12
	Find All References	Shift+F12
	View Call Hierarchy	Ctrl+K, Ctrl+T
	Run Tests	Ctrl+R, T
	Debug Tests	Ctrl+R, Ctrl+T
	IntelliTest	
	Live Unit Testing	

Visual Studio should then stop at the break point with yellow highlighting...

```
[TestMethod]
0 references
public void DateAddedInvalidData()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //set the DateAdded to a non date value
    string DateAdded = "this is not a date!";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}
```

Press F11 to step into the code for the validation method.

```
public string Valid(string houseNo, string street, string
{
    ≤1ms elapsed
    //create a string variable to store the error
    String Error = "";
    //create a temporary variable to store date values
```

Now press F10 to step line by line through the function.

When the following line is highlighted, stop and inspect the data value about to be assigned...

```
    //record the error
    Error = Error + "The house no must be less than 6 c
}
//copy the dateAdded value to the DateTemp variable
DateTemp = Convert.ToDateTime(dateAdded); ≤2ms elapsed
if (DateTemp < DateTime.Now.Date)
{
    //record the error
```

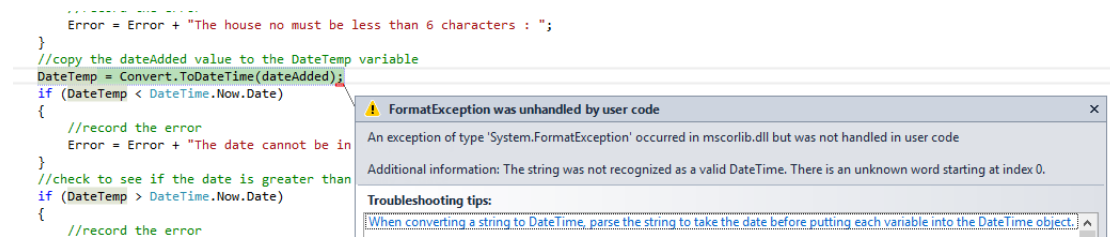
The string variable DateAdded contains the text data “this is not a date!”.

```
    }
    //copy the dateAdded value to the DateTemp variable
    DateTemp = Convert.ToDateTime(dateAdded); ≤2ms elapsed
    if (DateTemp < DateTime.Now.Date)
    {
        //record the error
        Error = Error + "The date cannot be in the past : ";
```

The assignment operation is about to convert that data to a date value and assign it to the variable DateTemp.

Here is where the problem lies.

Press F10 one more time and watch the program crash...



Stop the program running and let's add the code to fix the problem.

The issue is that we need to test the data stored in DateAdded prior to trying to assign it to the variable DateTemp.

If the data is a valid date then go ahead with the assignment and proceed to the remaining date validation.

If the data is not a valid date then stop working with the data as if it is a date and flag an error.

The following code will do this for us...

```
try  
{  
    //copy the dateAdded value to the DateTemp variable  
    DateTemp = Convert.ToDateTime(dateAdded);  
    if (DateTemp < DateTime.Now.Date)  
    {  
        //record the error  
        Error = Error + "The date cannot be in the past : ";  
    }  
    //check to see if the date is greater than today's date  
    if (DateTemp > DateTime.Now.Date)  
    {  
        //record the error  
        Error = Error + "The date cannot be in the future : ";  
    }  
}  
catch  
{  
    //record the error  
    Error = Error + "The date was not a valid date : ";  
}  
-----
```

Two parameters down three to go!

Here are the test plans for the remaining three parameters with their test methods...

Description of Item to Be Tested:

PostCode: Required field max length 9 characters. (We will ignore the complexities of post code formatting for the moment!)

Required Field **Y**

Test Type	Test Data	Expected Result	Actual Result
Extreme Min	NA		
Min -1	Blank	Error	
Min (Boundary)	1 character	OK	
Min +1	2 characters	OK	
Max -1	8 characters	OK	
Max (Boundary)	9 characters	OK	
Max +1	10 characters	Error	
Mid	4 characters	OK	
Extreme Max	500 characters	Error	
Invalid data type	We shall ignore this for the moment but post codes do have a format we may need to look at.		
Other tests			

Additional Notes / Instructions:

--

```

[TestMethod]
public void PostCodeMinLessOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should fail
    string PostCode = "";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}

[TestMethod]
public void PostCodeMin()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string PostCode = "a";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void PostCodeMinPlusOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string PostCode = "aa";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void PostCodeMaxLessOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string PostCode = "aaaaaaaa";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]

```

```

public void PostCodeMax()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string PostCode = "aaaaaaaaa";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void PostCodeMaxPlusOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should fail
    string PostCode = "aaaaaaaaa";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}

[TestMethod]
public void PostCodeMid()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string PostCode = "aaaa";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

```

Description of Item to Be Tested:

Street : May not be blank max length 50 characters

Required Field **Y**

Test Type	Test Data	Expected Result	Actual Result
Extreme Min	NA		
Min -1	Blank	Error	
Min (Boundary)	1 character	OK	
Min +1	2 characters	OK	
Max -1	49 characters	OK	
Max (Boundary)	50 characters	OK	
Max +1	51 characters	Error	
Mid	25 characters	OK	
Extreme Max	500 characters	Error	
Invalid data type	NA		
Other tests	NA		

Additional Notes / Instructions:

--

```

[TestMethod]
public void StreetMinLessOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should fail
    string Street = "";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}

[TestMethod]
public void StreetMin()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string Street = "a";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void StreetMinPlusOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string Street = "aa";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void StreetMaxLessOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string Street = "";
    Street = Street.PadRight(49, 'a');
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

```

```

[TestMethod]
public void StreetMax()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string Street = "";
    Street = Street.PadRight(50, 'a');
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void StreetMaxPlusOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should fail
    string Street = "";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}

[TestMethod]
public void StreetMid()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string Street = "";
    Street = Street.PadRight(25, 'a');
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

```

Description of Item to Be Tested:

Town : required field max length 50 characters

Required Field **Y**

Test Type	Test Data	Expected Result	Actual Result
Extreme Min	NA		
Min -1	Blank	Error	
Min (Boundary)	1 character	OK	
Min +1	2 characters	OK	
Max -1	49 characters	OK	
Max (Boundary)	50 characters	OK	
Max +1	51 characters	Error	
Mid	25 characters	OK	
Extreme Max	500 characters	Error	
Invalid data type	NA		
Other tests	NA		

Additional Notes / Instructions:

--


```

[TestMethod]
public void TownMinLessOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should fail
    string Town = "";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}

[TestMethod]
public void TownMin()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string Town = "a";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void TownMinPlusOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string Town = "aa";
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void TownMaxLessOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string Town = "";
    Town = Town.PadRight(49, 'a');
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

```

```

[TestMethod]
public void TownMax()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string Town = "";
    Town = Town.PadRight(50, 'a');
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

[TestMethod]
public void TownMaxPlusOne()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should fail
    string Town = "";
    Town = Town.PadRight(51, 'a');
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreNotEqual(Error, "");
}

[TestMethod]
public void TownMid()
{
    //create an instance of the class we want to create
    clsAddress AnAddress = new clsAddress();
    //string variable to store any error message
    String Error = "";
    //this should pass
    string Town = "";
    Town = Town.PadRight(25, 'a');
    //invoke the method
    Error = AnAddress.Valid(HouseNo, Street, Town, PostCode, DateAdded);
    //test to see that the result is correct
    Assert.AreEqual(Error, "");
}

```

Here is the code for the validation method...

```
public string Valid(string houseNo, string street, string town, string
postCode, string dateAdded)
{
    //create a string variable to store the error
    String Error = "";
    //create a temporary variable to store date values
    DateTime DateTemp;
    //if the HouseNo is blank
    if (houseNo.Length == 0)
    {
        //record the error
        Error = Error + "The house no may not be blank : ";
    }
    //if the house no is greater than 6 characters
    if (houseNo.Length > 6)
    {
        //record the error
        Error = Error + "The house no must be less than 6 characters : ";
    }
    try
    {
        //copy the dateAdded value to the DateTemp variable
        DateTemp = Convert.ToDateTime(dateAdded);
        if (DateTemp < DateTime.Now.Date)
        {
            //record the error
            Error = Error + "The date cannot be in the past : ";
        }
        //check to see if the date is greater than today's date
        if (DateTemp > DateTime.Now.Date)
        {
            //record the error
            Error = Error + "The date cannot be in the future : ";
        }
    }
    catch
    {
        //record the error
        Error = Error + "The date was not a valid date : ";
    }
    //is the post code blank
    if (postCode.Length == 0)
    {
        //record the error
        Error = Error + "The post code may not be blank : ";
    }
    //if the post code is too long
    if (postCode.Length > 9)
    {
        //record the error
        Error = Error + "The post code must be less than 9 characters : ";
    }
    //is the street blank
    if (street.Length == 0)
    {
        //record the error
        Error = Error + "The street may not be blank : ";
    }
    //if the street is too long
```

```

    if (street.Length > 50)
    {
        //record the error
        Error = Error + "The street must be less than 50 characters : ";
    }
    //is the town blank
    if (town.Length == 0)
    {
        //record the error
        Error = Error + "The town may not be blank : ";
    }
    //if the town is too long
    if (town.Length > 50)
    {
        //record the error
        Error = Error + "The town must be less than 50 characters : ";
    }
    //return any error messages
    return Error;
}

```

We have now completed the following:

clsCounty

- County property

- CountyNo property

- Validation method

clsCountyCollection

- A list of counties

- A count of items in the list

clsAddress

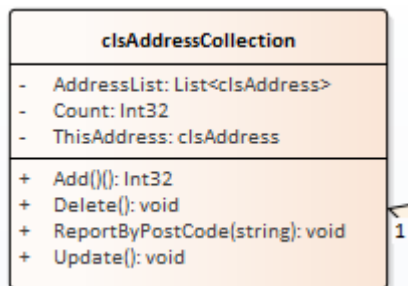
- All properties set up

- A find method which sets all properties should the record be found

- A validation method

Pathway 4 – Creating the Complex Collection Class

The final class to complete is the class `clsAddressCollection`.



As always start with the simple stuff first.

Create the test class `tstAddressCollection`...

```
tstAddressCollection.cs  [X]
Test_Framework.tstAddressCollection

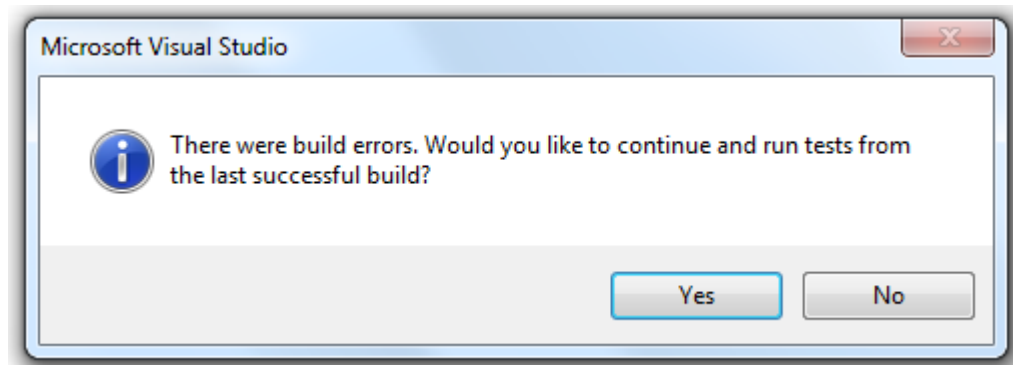
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Test_Framework
{
    [TestClass]
    public class tstAddressCollection
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

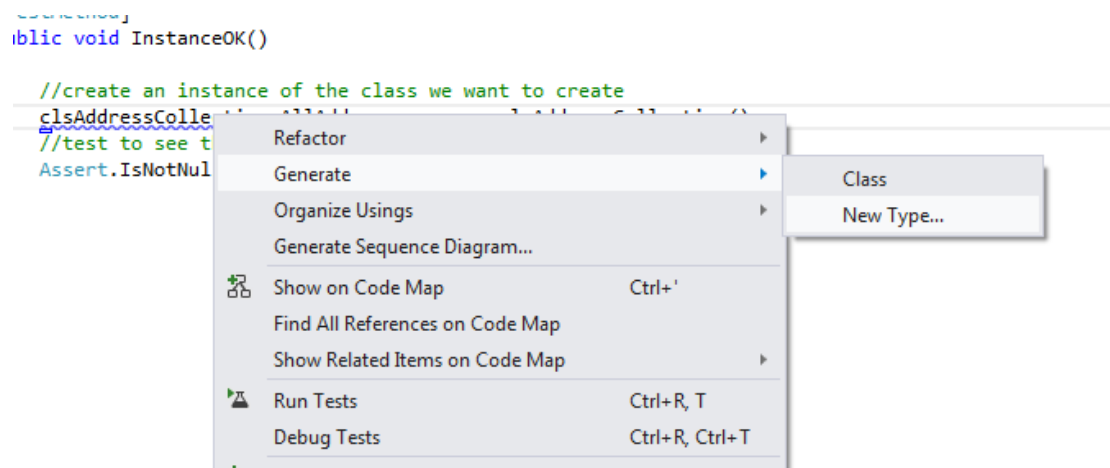
Next create a test to see if we can create an instance of the class...

```
[TestMethod]
public void InstanceOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //test to see that it exists
    Assert.IsNotNull(AllAddresses);
}
```

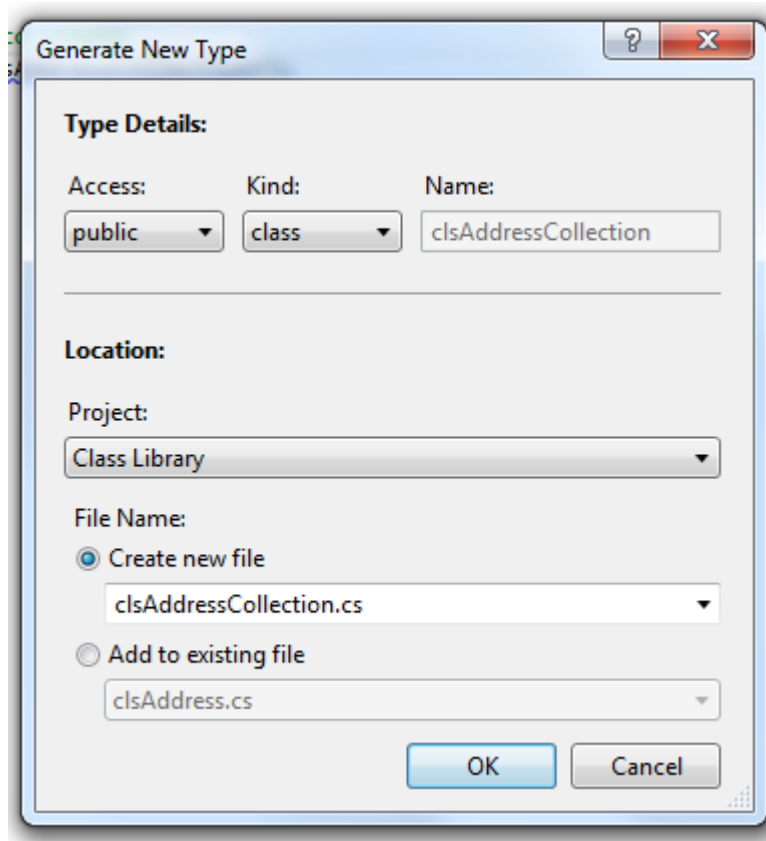
Watch the test fail...



Now fix the test...



Making sure the new class file is created in the class library...



Then see the test pass.

Creating the Properties

Next we create the properties for the class.

It doesn't matter at this stage what order we create them in so we will use the order on the class diagram:

- AddressList
- Count
- ThisAddress

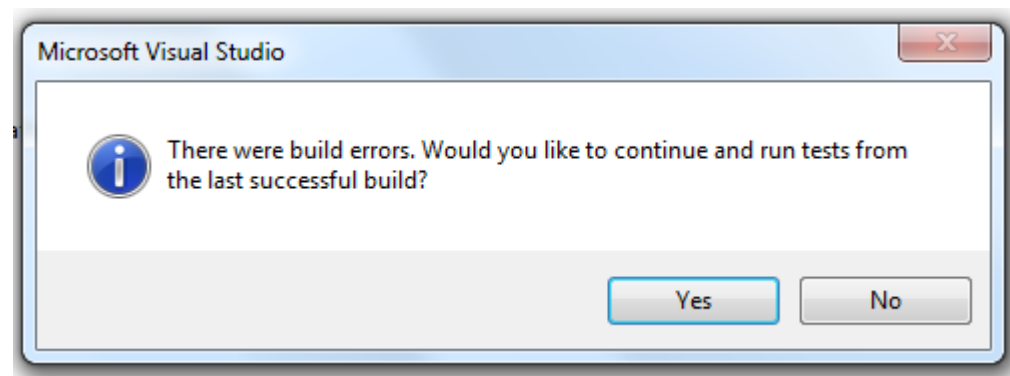
We need to first import into the class file the .NET library for handling collections...

```
using System;  
using Microsoft.VisualStudio.TestTools.UnitTesting;  
using Class_Library;  
using System.Collections.Generic;
```

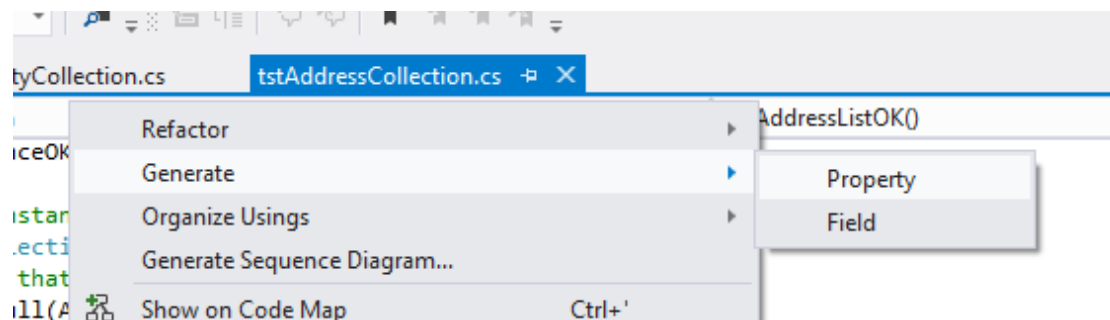

Now we may create the test for the AddressList property...

```
[TestMethod]
public void AddressListOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create some test data to assign to the property
    //in this case the data needs to be a list of objects
    List<clsAddress> TestList = new List<clsAddress>();
    //add an item to the list
    //create the item of test data
    clsAddress TestItem = new clsAddress();
    //set its properties
    TestItem.Active = true;
    TestItem.AddressNo = 1;
    TestItem.CountyNo = 1;
    TestItem.DateAdded = DateTime.Now.Date;
    TestItem.HouseNo = "123a";
    TestItem.PostCode = "LE1 1WE";
    TestItem.Street = "some street";
    TestItem.Town = "some town";
    //add the item to the test list
    TestList.Add(TestItem);
    //assign the data to the property
    AllAddresses.AddressList = TestList;
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.AddressList, TestList);
}
```

As usual the test should fail...



Fix the test to see it pass...



Here are the remaining tests for the other two properties....

```
[TestMethod]
public void CountPropertyOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create some test data to assign to the property
    Int32 SomeCount = 0;
    //assign the data to the property
    AllAddresses.Count = SomeCount;
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.Count, SomeCount);
}

[TestMethod]
public void ThisAddressPropertyOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create some test data to assign to the property
    clsAddress TestAddress = new clsAddress();
    //set the properties of the test object
    TestAddress.Active = true;
    TestAddress.AddressNo = 1;
    TestAddress.CountyNo = 1;
    TestAddress.DateAdded = DateTime.Now.Date;
    TestAddress.HouseNo = "123a";
    TestAddress.PostCode = "LE1 1WE";
    TestAddress.Street = "some street";
    TestAddress.Town = "some town";
    //assign the data to the property
    AllAddresses.ThisAddress = TestAddress;
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.ThisAddress, TestAddress);
}
```

(I will let you sort out the code that generates the two properties!)

In a similar fashion to the code for clsCountyCollection we will set about creating the code that drives the Count property and the AddressList.

The AddressList will provide us with an indexed list of Address from the database table.

The Count property will always tell us how many addresses there are in this list.

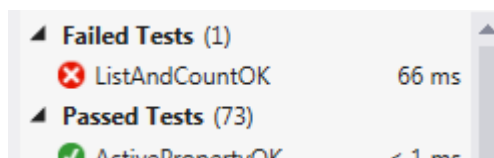
We have already created the tests to make sure the two properties exist so let's spend a little time writing the code to create the functionality.

The first test we shall write is one to test if records can be added to the lists and the Count property correctly reports how many items are in the list.

Here is the test...

```
[TestMethod]
public void ListAndCountOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create some test data to assign to the property
    //in this case the data needs to be a list of objects
    List<clsAddress> TestList = new List<clsAddress>();
    //add an item to the list
    //create the item of test data
    clsAddress TestItem = new clsAddress();
    //set its properties
    TestItem.Active = true;
    TestItem.AddressNo = 1;
    TestItem.CountyNo = 1;
    TestItem.DateAdded = DateTime.Now.Date;
    TestItem.HouseNo = "123a";
    TestItem.PostCode = "LE1 1WE";
    TestItem.Street = "some street";
    TestItem.Town = "some town";
    //add the item to the test list
    TestList.Add(TestItem);
    //assign the data to the property
    AllAddresses.AddressList = TestList;
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.Count, TestList.Count);
}
```

This should of course fail...



We now need to modify clsAddressCollection in very similar ways to how we modified clsCountyCollection.

We need to create a private data member for the list...

```
public class clsAddressCollection
{
    //private data member for the list
    List<clsAddress> mAddressList = new List<clsAddress>();
    // . . . . .
}
```

Next we need to expose the private data via the public property...

```
//public property for the address list
public List<clsAddress> AddressList
{
    get
    {
        //return the private data
        return mAddressList;
    }
    set
    {
        //set the private data
        mAddressList = value;
    }
}
```

Once that is set up we may return the value of the Count property of the list as the value of our own Count property...

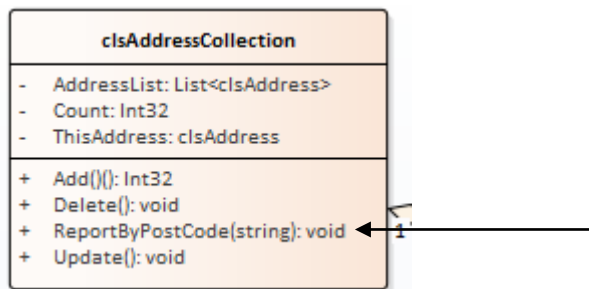
```
//public property for the address list
public List<clsAddress> AddressList
{
    get
    {
        //return the private data
        return mAddressList;
    }
    set
    {
        //set the private data
        mAddressList = value;
    }
}

//public property for count
public int Count
{
    get
    {
        //return the count of the list
        return mAddressList.Count;
    }
    set
    {
        //we shall worry about this later
    }
}
```

The test should now pass however the code is still rubbish!

The way we want this object to behave is that when we create an instance of the class the list is populated with all of the addresses in the database.

When we invoke the ReportByPostCode method...



The AddressList should only contain the addresses that match the post code supplied as a filter.

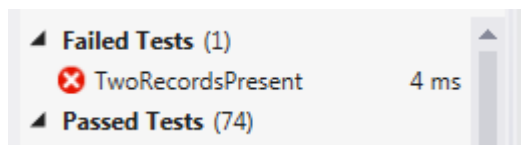
Let's create a test that gets us going in the right direction but will almost certainly be scrapped later.

Take a look at the following test...

```
[TestMethod]
public void TwoRecordsPresent()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.Count, 2);
}
```

This test assumes that there are always two records in the AllAddresses object upon instantiation.

Clearly this test will fail (as always).



OK - so let's add some code to the class's constructor.

Like so...

```
//constructor for the class
public clsAddressCollection()
{
    //create the items of test data
    clsAddress TestItem = new clsAddress();
    //set its properties
    TestItem.Active = true;
    TestItem.AddressNo = 1;
    TestItem.CountyNo = 1;
    TestItem.DateAdded = DateTime.Now.Date;
    TestItem.HouseNo = "123a";
    TestItem.PostCode = "LE1 1WE";
    TestItem.Street = "some street";
    TestItem.Town = "some town";
    //add the item to the test list
    mAddressList.Add(TestItem);
    //re initialise the object for some new data
    TestItem = new clsAddress();
    //set its properties
    TestItem.Active = true;
    TestItem.AddressNo = 2;
    TestItem.CountyNo = 2;
    TestItem.DateAdded = DateTime.Now.Date;
    TestItem.HouseNo = "99";
    TestItem.PostCode = "LE1 7YY";
    TestItem.Street = "another street";
    TestItem.Town = "another town";
    //add the item to the test list
    mAddressList.Add(TestItem);
}
```

This adds two test records to the private list such that the test for two records now passes.

The bad news is that another test now fails...



So what's the problem?

This test...

```
[TestMethod]
public void CountPropertyOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create some test data to assign to the property
    Int32 SomeCount = 0;
    //assign the data to the property
    AllAddresses.Count = SomeCount;
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.Count, SomeCount);
}
```

This test assumes that there will always be zero records in the list. This was true until we started adding code to the constructor just now.

We will apply a quick fix to this just to make the test pass like so...

```
[TestMethod]
public void CountPropertyOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create some test data to assign to the property
    Int32 SomeCount = 2;
    //assign the data to the property
    AllAddresses.Count = SomeCount;
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.Count, SomeCount);
}
```

Honestly the code is still rubbish!

What we really want to be doing here is reading the data from the database rather than hard coding records like this.

Let's assume we have the following data...

AddressNo	HouseNo	Street	Town	PostCode	CountyNo	DateAdded	Active
2	1	Some Street	Leicester	LE1 1BE	35	07/09/2012	True
3	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
4	33	High Street	Leicester	LE1 6FG	35	07/08/2012	True
5	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
21	123	Test Street	Test Town	XXX XXX	1	16/09/2015	True
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

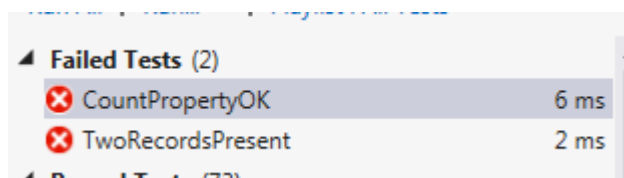
Along with the following stored procedure...

```
CREATE PROCEDURE sproc_tblAddress_SelectAll
AS
--select all the records from the table
    select * from tblAddress
RETURN 0
```

The following code should do the trick...

```
//constructor for the class
public clsAddressCollection()
{
    //var for the index
    Int32 Index = 0;
    //var to store the record count
    Int32 RecordCount = 0;
    //object for data connection
    clsDataConnection DB = new clsDataConnection();
    //execute the stored procedure
    DB.Execute("sproc_tblAddress_SelectAll");
    //get the count of records
    RecordCount = DB.Count;
    //while there are records to process
    while (Index < RecordCount)
    {
        //create a blank address
        clsAddress AnAddress = new clsAddress();
        //read in the fields from the current record
        AnAddress.Active = Convert.ToBoolean(DB.DataTable.Rows[Index]["Active"]);
        AnAddress.AddressNo = Convert.ToInt32(DB.DataTable.Rows[Index]["AddressNo"]);
        AnAddress.CountyNo = Convert.ToInt32(DB.DataTable.Rows[Index]["CountyNo"]);
        AnAddress.DateAdded = Convert.ToDateTime(DB.DataTable.Rows[Index]["DateAdded"]);
        AnAddress.HouseNo = Convert.ToString(DB.DataTable.Rows[Index]["HouseNo"]);
        AnAddress.PostCode = Convert.ToString(DB.DataTable.Rows[Index]["PostCode"]);
        AnAddress.Street = Convert.ToString(DB.DataTable.Rows[Index]["Street"]);
        AnAddress.Town = Convert.ToString(DB.DataTable.Rows[Index]["Town"]);
        //add the record to the private data member
        mAddressList.Add(AnAddress);
        //point at the next record
        Index++;
    }
}
```

If you run your tests now you should see the following results...



The first problem is that we no longer have a guaranteed two records...

```
[TestMethod]
public void CountPropertyOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create some test data to assign to the property
    Int32 SomeCount = 2;
    //assign the data to the property
    AllAddresses.Count = SomeCount;
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.Count, SomeCount);
}
```

To be honest we could get rid of this test now as it is tested more thoroughly by this test...

```
[TestMethod]
public void ListAndCountOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create some test data to assign to the property
    //in this case the data needs to be a list of objects
    List<clsAddress> TestList = new List<clsAddress>();
    //add an item to the list
    //create the item of test data
    clsAddress TestItem = new clsAddress();
    //set its properties
    TestItem.Active = true;
    TestItem.AddressNo = 1;
    TestItem.CountyNo = 1;
    TestItem.DateAdded = DateTime.Now.Date;
    TestItem.HouseNo = "123a";
    TestItem.PostCode = "LE1 1WE";
    TestItem.Street = "some street";
    TestItem.Town = "some town";
    //add the item to the test list
    TestList.Add(TestItem);
    //assign the data to the property
    AllAddresses.AddressList = TestList;
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.Count, TestList.Count);
}
```

The following test...

```
[TestMethod]
public void TwoRecordsPresent()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.Count, 2);
}
```

Is also redundant and may be removed.

Unlike the testing in pathway 2 for clsCountyCollection which should always have 72 records we cannot predict how many records will be present in the collection clsAddressCollection.

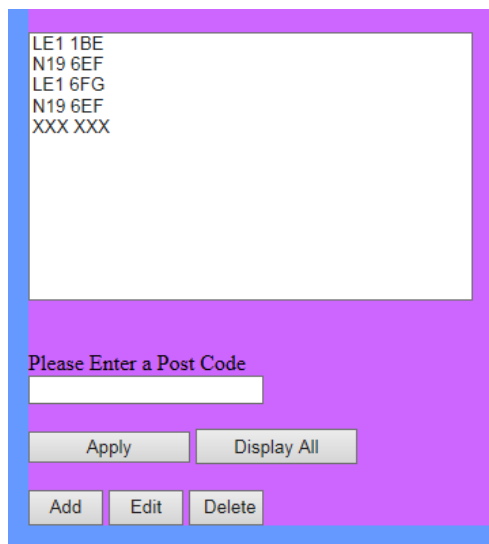
Getting rid of the two tests above should mean that all of the tests pass and we now have a usable and tested collection class which is ready to be linked to the presentation layer.

In the smoke and mirrors prototype we need to add the following code to the form Default.aspx

```
public partial class _Default : System.Web.UI.Page
{
    //this function handles the load event for the page
    protected void Page_Load(object sender, EventArgs e)
    {
        //if this is the first time the page is displayed
        if (IsPostBack == false)
        {
            //update the list box
            DisplayAddresses();
        }
    }

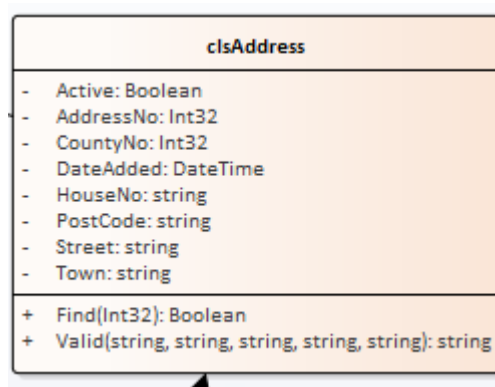
    void DisplayAddresses()
    {
        //create an instance of the County Collection
        Class_Library.clsAddressCollection Addresses = new Class_Library.clsAddressCollection();
        //set the data source to the list of counties in the collection
        lstAddresses.DataSource = Addresses.AddressList;
        //set the name of the primary key
        lstAddresses.DataValueField = "AddressNo";
        //set the data field to display
        lstAddresses.DataTextField = "PostCode";
        //bind the data to the list
        lstAddresses.DataBind();
    }
}
```

This code should update the list box with a list of post codes.



At this point it is worth thinking about how the property `ThisAddress` is going to work and how it relates to the methods `Add`, `Update` and `Delete`.

`ThisAddress` is an instance of `clsAddress` which we have already seen looks like this...



The `find` method allows us to search for a specific address based on its primary key value.

Assuming we have the following data...

AddressNo	HouseNo	Street	Town	PostCode	CountyNo	DateAdded	Active
2	1	Some Street	Leicester	LE1 1BE	35	07/09/2012	True
3	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
4	33	High Street	Leicester	LE1 6FG	35	07/08/2012	True
5	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
21	123	Test Street	Test Town	XXX XXX	1	16/09/2015	True
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

The following code...

```
clsAddress SomeAddress = new clsAddress();  
SomeAddress.Find(3);
```

Would locate the second record in the list setting the values of the properties of SomeAddress to match the data stored in the record.

We can use the find method of clsAddress via any instances of clsAddressCollection.

Take a look at the following code...

```
clsAddressCollection AllAddresses = new clsAddressCollection();  
AllAddresses.ThisAddress.Find(3);
```

This code would achieve exactly the same outcome as the previous code example.

Having the ThisAddress property allows us to point to any address we want to do something to, by searching on its primary key.

Once we are pointing at a specific record we may use Add, Update and Delete to change the data pointed to by ThisAddress.

So...

```
clsAddressCollection AllAddresses = new clsAddressCollection();  
AllAddresses.ThisAddress.Find(3);  
AllAddresses.Delete();
```

Would delete the address with the primary key value of 3.

Or...

```
clsAddressCollection AllAddresses = new clsAddressCollection();  
AllAddresses.ThisAddress.Find(3);  
AllAddresses.ThisAddress.HouseNo = "22";  
AllAddresses.Update();
```

Would set the value of HouseNo for record 3 to "22".

Or...

```
clsAddressCollection AllAddresses = new clsAddressCollection();  
AllAddresses.ThisAddress.Active = true;  
AllAddresses.ThisAddress.CountyNo = 1;  
AllAddresses.ThisAddress.DateAdded = DateTime.Now.Date;  
AllAddresses.ThisAddress.HouseNo = "22";  
AllAddresses.ThisAddress.PostCode = "LE1 1WE";  
AllAddresses.ThisAddress.Street = "Some Street";  
AllAddresses.ThisAddress.Town = "Some Town";  
AllAddresses.Add();
```

Would add a new record based on the data we assigned to ThisAddress.

Creating the Methods

The good news is that having created all of the tests for clsAddress we have a working find method already!

This means that we are in the position to press-on to testing Add, Update and Delete.

Creating the Add Method

Here is the test method for Add...

```
[TestMethod]
public void AddMethodOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create the item of test data
    clsAddress TestItem = new clsAddress();
    //var to store the primary key
    Int32 PrimaryKey = 0;
    //set its properties
    TestItem.Active = true;
    TestItem.AddressNo = 1;
    TestItem.CountyNo = 1;
    TestItem.DateAdded = DateTime.Now.Date;
    TestItem.HouseNo = "123a";
    TestItem.PostCode = "LE1 1WE";
    TestItem.Street = "some street";
    TestItem.Town = "some town";
    //set ThisAddress to the test data
    AllAddresses.ThisAddress = TestItem;
    //add the record
    PrimaryKey = AllAddresses.Add();
    //set the primary key of the test data
    TestItem.AddressNo = PrimaryKey;
    //find the record
    AllAddresses.ThisAddress.Find(PrimaryKey);
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.ThisAddress, TestItem);
}
```

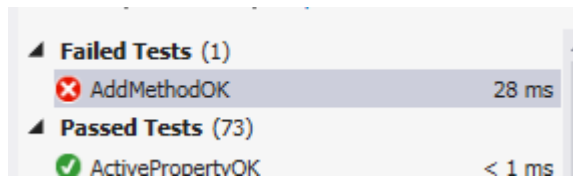
We have created rather more code here than usual. As we create more of the system and our confidence increases we have the option to make bigger steps.

Notice what we are trying to do here:

1. Create the collection
2. Create some test data
3. Use the test data to set the ThisAddress property
4. Add the record to the database retrieving the primary key of the new record
5. Find the record to check that it exists
6. Compare the data found with the original test data (they should be the same)!

Run the test and watch it fail.

Create the Add method and the test will still fail!



Why? It is because we don't have any suitable code in the class to implement the method.

To fix this we will need to create a new private data member, modify the ThisAddress property and finally add some code to the Add method.

First let's create the private data member at the top of the class...

```
public class clsAddressCollection
{
    //private data member for the list
    List<clsAddress> mAddressList = new List<clsAddress>();
    //private data member thisAddress
    clsAddress mThisAddress = new clsAddress();
}
```

Next we expose the private data member by modifying the public property like so...

```
//public property for ThisAddress
public clsAddress ThisAddress
{
    get
    {
        //return the private data
        return mThisAddress;
    }
    set
    {
        //set the private data
        mThisAddress = value;
    }
}
```

Lastly we put a fix in the Add method to force the test to pass...

```
public int Add()
{
    //adds a new record to the database based on the values of mThisAddress
    //set the primary key value of the new record
    mThisAddress.AddressNo = 123;
    //return the primary key of the new record
    return mThisAddress.AddressNo;
}
```

The good news is that it forces the test to pass. The bad news is that the code is pretty rubbish!

It would make more sense now to refine the code to make it actually work.

Assuming we have the following table...

AddressNo	HouseNo	Street	Town	PostCode	CountyNo	DateAdded	Active
2	1	Some Street	Leicester	LE1 1BE	35	07/09/2012	True
3	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
4	33	High Street	Leicester	LE1 6FG	35	07/08/2012	True
5	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
21	123	Test Street	Test Town	XXX XXX	1	16/09/2015	True
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Along with the following stored procedure...

```
CREATE PROCEDURE sproc_tblAddress_Insert
--create parameters for the stored procedure
    @HouseNo varchar (6),
    @Street varchar (50),
    @Town varchar (50),
    @PostCode varchar (9),
    @CountyNo int,
    @DateAdded date,
    @Active bit
AS
--insert the new record
INSERT INTO tblAddress (HouseNo, Street, Town, PostCode, CountyNo, DateAdded, Active)
values (@HouseNo, @Street, @Town, @PostCode, @CountyNo, @DateAdded, @Active)

--return the primary key value of the new record
return @@Identity
```


The amended function for Add should do the trick...

```
public int Add()
{
    //adds a new record to the database based on the values of thisAddress
    //connect to the database
    clsDataConnection DB = new clsDataConnection();
    //set the parameters for the stored procedure
    DB.AddParameter("@HouseNo", mThisAddress.HouseNo);
    DB.AddParameter("@Street", mThisAddress.Street);
    DB.AddParameter("@Town", mThisAddress.Town);
    DB.AddParameter("@PostCode", mThisAddress.PostCode);
    DB.AddParameter("@CountyNo", mThisAddress.CountyNo);
    DB.AddParameter("@DateAdded", mThisAddress.DateAdded);
    DB.AddParameter("@Active", mThisAddress.Active);
    //execute the query returning the primary key value
    return DB.Execute("sproc_tblAddress_Insert");
}
```

Lastly by examining the database we are able to see that a new record has been added...

	AddressNo	HouseNo	Street	Town	PostCode	CountyNo	DateAdded	Active
▶	2	1	Some Street	Leicester	LE1 1BE	35	07/09/2012	True
	3	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
	4	33	High Street	Leicester	LE1 6FG	35	07/08/2012	True
	5	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
	21	123	Test Street	Test Town	XXX XXX	1	16/09/2015	True
	22	123a	some street	some town	LE1 1WE	1	22/09/2015	True
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Having created the testing and the code for the Add method let's add this to the presentation layer.

Open the form Default.aspx...

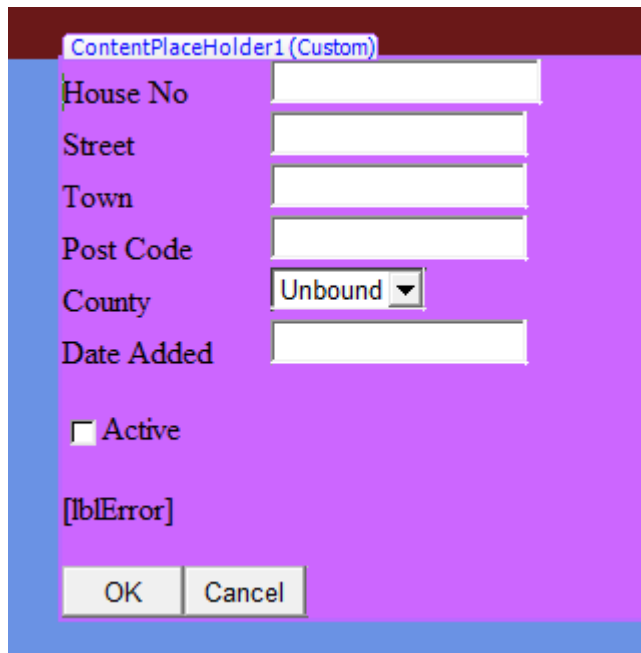
The screenshot shows a web form with a purple background. At the top, there is a label 'ContentPlaceHolder1 (Custom)'. Below it is a large white rectangular area labeled 'Unbound'. Underneath this area is a label '[lblError]'. Below the error label is the text 'Please Enter a Post Code' followed by a text input field. Below the input field are two buttons: 'Apply' and 'Display All'. At the bottom of the form are three buttons: 'Add', 'Edit', and 'Delete'.

And access the event handler for the click event...

```
//event handler for the add button
protected void btnAdd_Click(object sender, EventArgs e)
{
    //store -1 into the session object to indicate this is a new record
    Session["AddressNo"] = -1;
    //redirect to the data entry page
    Response.Redirect("AnAddress.aspx");
}
```

Notice how a value of -1 is placed in the session object on the server.

Once this value is set we then re direct to the page AnAddress.aspx...



What we want to happen is that the user enters data into the fields on this form, presses OK and the data is added to the table via the middle layer classes.

We need to create a function which...

- Validates the data
- Captures the data
- Adds it to the database

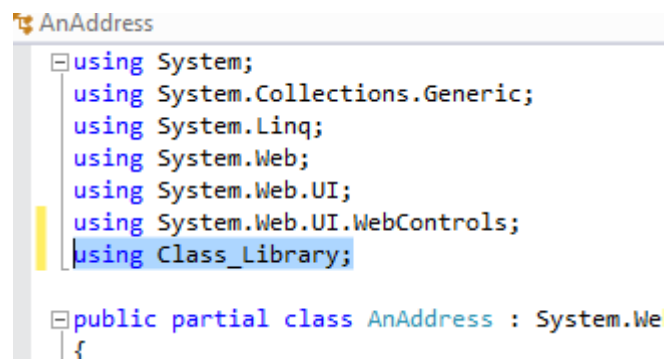
Create the following function inside AnAddress.aspx...

```
//function for adding new records
1 reference
void Add()
{
    //create an instance of the address book
    Class_Library.clsAddressCollection AddressBook = new Class_Library.clsAddressCollection();
    //validate the data on the web form
    String Error = AddressBook.ThisAddress.Valid(txtHouseNo.Text, txtStreet.Text, txtTown.Text, txtPostCode.Text, txtDateAdded.Text);
    //if the data is OK then add it to the object
    if (Error == "")
    {
        //get the data entered by the user
        AddressBook.ThisAddress.HouseNo = txtHouseNo.Text;
        AddressBook.ThisAddress.Street = txtStreet.Text;
        AddressBook.ThisAddress.Town = txtTown.Text;
        AddressBook.ThisAddress.PostCode = txtPostCode.Text;
        AddressBook.ThisAddress.DateAdded = Convert.ToDateTime(txtDateAdded.Text);
        AddressBook.ThisAddress.Active = chkActive.Checked;
        AddressBook.ThisAddress.CountyNo = Convert.ToInt32(ddlCounty.SelectedValue);
        //add the record
        AddressBook.Add();
    }
    else
    {
        //report an error
        lblError.Text = "There were problems with the data entered " + Error;
    }
}
```

Notice the line of code...

```
//create an instance of the address book
Class_Library.clsAddressCollection AddressBook = new Class_Library.clsAddressCollection();
//validate the data on the web form
```

If you want to make this a bit more concise add a “using” to the top of the code like so...



```
AnAddress
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using Class_Library;

public partial class AnAddress : System.Web
```

In which case the function changes like so...

```
//function for adding new records
1 reference
void Add()
{
    //create an instance of the address book
    clsAddressCollection AddressBook = new clsAddressCollection();
    //validate the data on the web form
    String Error = AddressBook.ThisAddress.Valid(txtHouseNo.Text, txtStreet.Text, txtTown.Text, txtPostCode.Text, txtDateAdded.Text);
    //if the data is OK then add it to the object
    if (Error == "")
    {
        //get the data entered by the user
        AddressBook.ThisAddress.HouseNo = txtHouseNo.Text;
        AddressBook.ThisAddress.Street = txtStreet.Text;
        AddressBook.ThisAddress.Town = txtTown.Text;
        AddressBook.ThisAddress.PostCode = txtPostCode.Text;
        AddressBook.ThisAddress.DateAdded = Convert.ToDateTime(txtDateAdded.Text);
        AddressBook.ThisAddress.Active = chkActive.Checked;
        AddressBook.ThisAddress.CountyNo = Convert.ToInt32(ddlCounty.SelectedValue);
        //add the record
        AddressBook.Add();
        //all done so redirect back to the main page
        Response.Redirect("Default.aspx");
    }
    else
    {
        //report an error
        lblError.Text = "There were problems with the data entered " + Error;
    }
}

//function for updating records
```

You will need to add a call to the click event handler for the OK button...

```
//event handler for the ok button
protected void btnOK_Click(object sender, EventArgs e)
{
    //add the new record
    Add();
    //all done so redirect back to the main page
    Response.Redirect("Default.aspx");
}
```

Again it is worth noting how quickly we were able to add the functionality to the presentation layer once we had the middle layer classes completed.

Also remember that having created the test framework we can be fairly sure that the functionality does what we think it should.

Creating the Delete Method

The next method we shall test is the Delete method...

```
[TestMethod]
public void DeleteMethodOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create the item of test data
    clsAddress TestItem = new clsAddress();
    //var to store the primary key
    Int32 PrimaryKey = 0;
    //set its properties
    TestItem.Active = true;
    TestItem.AddressNo = 1;
    TestItem.CountyNo = 1;
    TestItem.DateAdded = DateTime.Now.Date;
    TestItem.HouseNo = "123a";
    TestItem.PostCode = "LE1 1WE";
    TestItem.Street = "some street";
    TestItem.Town = "some town";
    //set ThisAddress to the test data
    AllAddresses.ThisAddress = TestItem;
    //add the record
    PrimaryKey = AllAddresses.Add();
    //set the primary key of the test data
    TestItem.AddressNo = PrimaryKey;
    //find the record
    AllAddresses.ThisAddress.Find(PrimaryKey);
    //delete the record
    AllAddresses.Delete();
    //now find the record
    Boolean Found = AllAddresses.ThisAddress.Find(PrimaryKey);
    //test to see that the record was not found
    Assert.IsFalse(Found);
}
```

It will fail due to no Delete method. Create the method and watch it fail for lack of suitable code.



Assuming we have a stored procedure like so...

```

CREATE PROCEDURE [dbo].sproc_tblAddress_Delete
--this stored procedure is situated in the data layer (App_Data folder)

--this stored procedure is used to delete a single record in the table tblAddress
--it accepts a single parameter @AddressNo and returns no value

    --declare the parameter
    @AddressNo int

AS
    --delete the record in tblAddress specified by the value of @AddressNo
    delete from tblAddress where AddressNo = @AddressNo;

```

Here is the finished delete method...

```

public void Delete()
{
    //deletes the record pointed to by thisAddress
    //connect to the database
    clsDataConnection DB = new clsDataConnection();
    //set the parameters for the stored procedure
    DB.AddParameter("@AddressNo", mThisAddress.AddressNo);
    //execute the stored procedure
    DB.Execute("sproc_tblAddress_Delete");
}

```

As before we will link the middle layer to the presentation layer.

From the form Default.aspx...

ContentPlaceHolder1(Custom)

Unbound

[lblError]

Please Enter a Post Code

Apply Display All

Add Edit Delete

Access the click event handler for Delete...

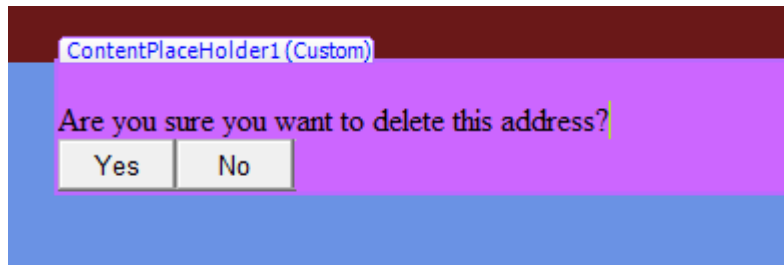
```
//event handler for the delete button
protected void btnDelete_Click(object sender, EventArgs e)
{
    //var to store the primary key value of the record to be deleted
    Int32 AddressNo;
    //if a record has been selected from the list
    if (lstAddresses.SelectedIndex != -1)
    {
        //get the primary key value of the record to delete
        AddressNo = Convert.ToInt32(lstAddresses.SelectedValue);
        //store the data in the session object
        Session["AddressNo"] = AddressNo;
        //redirect to the delete page
        Response.Redirect("Delete.aspx");
    }
    else //if no record has been selected
    {
        //display an error
        lblError.Text = "Please select a record to delete from the list";
    }
}
```

Notice how the code checks to see that an entry has been selected from the list.

Once this is OK we obtain the primary key value from the selected item of the list.

Next we pass the value of the primary key to the session object.

Lastly we redirect to the web form Delete.aspx...



In order to obtain the primary key value of the record to delete we need to access the value stored in the session object.

We will do this via the load event like so...

```
//event handler for the load event
protected void Page_Load(object sender, EventArgs e)
{
    //get the number of the address to be deleted from the session object
    AddressNo = Convert.ToInt32(Session["AddressNo"]);
}
```

Notice how the variable AddressNo is declared at the top of the code giving it page level scope...

```
public partial class Delete : System.Web.UI.Page
{
    //var to store the primary key value of the record to be deleted
    Int32 AddressNo;

    //event handler for the load event
```

Here is the function...

```
void DeleteAddress()
{
    //function to delete the selected record

    //create a new instance of the address book
    clsAddressCollection AddressBook = new clsAddressCollection();
    //find the record to delete
    AddressBook.ThisAddress.Find(AddressNo);
    //delete the record
    AddressBook.Delete();
}
```


Here is the amended event handler...

```
//event handler for the yes button
protected void btnYes_Click(object sender, EventArgs e)
{
    //delete the record
    DeleteAddress();
    //redirect back to the main page
    Response.Redirect("Default.aspx");
}
```

Creating the Update Method

Lastly we shall create the Update method.

Here is the full test...

```

[TestMethod]
public void UpdateMethodOK()
{
    //create an instance of the class we want to create
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create the item of test data
    clsAddress TestItem = new clsAddress();
    //var to store the primary key
    Int32 PrimaryKey = 0;
    //set its properties
    TestItem.Active = true;
    TestItem.CountyNo = 1;
    TestItem.DateAdded = DateTime.Now.Date;
    TestItem.HouseNo = "123a";
    TestItem.PostCode = "LE1 1WE";
    TestItem.Street = "some street";
    TestItem.Town = "some town";
    //set ThisAddress to the test data
    AllAddresses.ThisAddress = TestItem;
    //add the record
    PrimaryKey = AllAddresses.Add();
    //set the primary key of the test data
    TestItem.AddressNo = PrimaryKey;
    //modify the test data
    TestItem.Active = false;
    TestItem.CountyNo = 3;
    TestItem.DateAdded = DateTime.Now.Date;
    TestItem.HouseNo = "123b";
    TestItem.PostCode = "LE2 2WE";
    TestItem.Street = "another street";
    TestItem.Town = "another town";
    //set the record based on the new test data
    AllAddresses.ThisAddress = TestItem;
    //update the record
    AllAddresses.Update();
    //find the record
    AllAddresses.ThisAddress.Find(PrimaryKey);
    //test to see ThisAddress matches the test data
    Assert.AreEqual(AllAddresses.ThisAddress, TestItem);
}

```

As usual it will fail due to lack of a stub for the method and a lack of code.

Here is the stored procedure we are using...

```
CREATE PROCEDURE sproc_tblAddress_Update
--create the parameters for the stored procedure
    @AddressNo int,
    @HouseNo varchar (6),
    @Street varchar (50),
    @Town varchar (50),
    @PostCode varchar (9),
    @CountyNo int,
    @DateAdded date,
    @Active bit

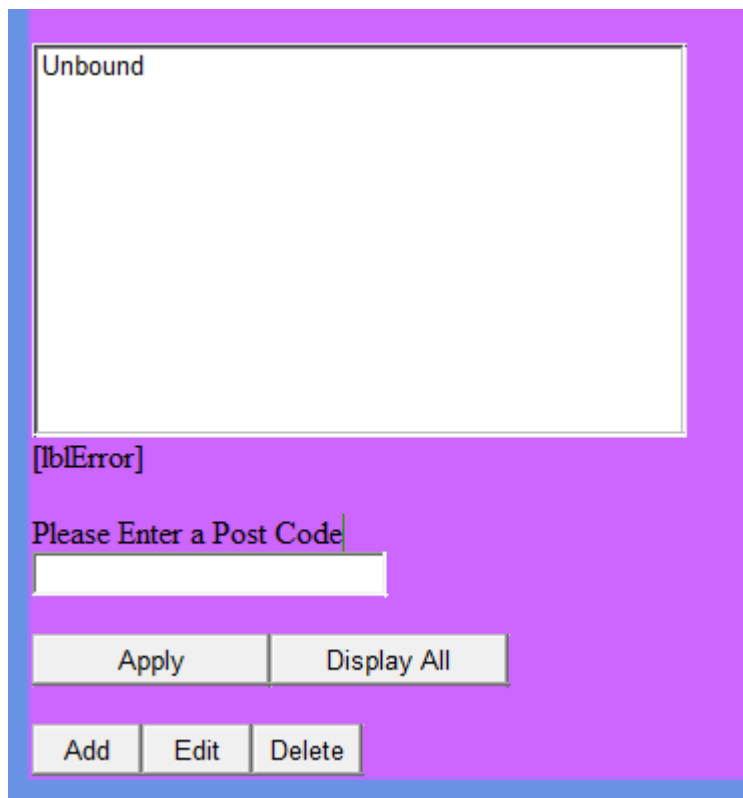
AS
--update the record as specified by @AddressNo value
update tblAddress
set HouseNo=@HouseNo,
    Street=@Street,
    Town=@Town,
    PostCode=@PostCode,
    CountyNo=@CountyNo,
    DateAdded=@DateAdded,
    Active=@Active
where AddressNo=@AddressNo
```

Here is the finished Update method...

```
public void Update()
{
    //update an existing record based on the values of thisAddress
    //connect to the database
    clsDataConnection DB = new clsDataConnection();
    //set the parameters for the stored procedure
    DB.AddParameter("@AddressNo", mThisAddress.AddressNo);
    DB.AddParameter("@HouseNo", mThisAddress.HouseNo);
    DB.AddParameter("@Street", mThisAddress.Street);
    DB.AddParameter("@Town", mThisAddress.Town);
    DB.AddParameter("@PostCode", mThisAddress.PostCode);
    DB.AddParameter("@CountyNo", mThisAddress.CountyNo);
    DB.AddParameter("@DateAdded", mThisAddress.DateAdded);
    DB.AddParameter("@Active", mThisAddress.Active);
    //execute the stored procedure
    DB.Execute("sproc_tblAddress_Update");
}
```

As before with Add and Delete we are now in a position to bolt the middle layer to the presentation layer.

In the form Default.aspx...



Access the click event handler for Edit...

```
//event handler for the edit button
protected void btnEdit_Click(object sender, EventArgs e)
{
    //var to store the primary key value of the record to be edited
    Int32 AddressNo;
    //if a record has been selected from the list
    if (lstAddresses.SelectedIndex != -1)
    {
        //get the primary key value of the record to edit
        AddressNo = Convert.ToInt32(lstAddresses.SelectedValue);
        //store the data in the session object
        Session["AddressNo"] = AddressNo;
        //redirect to the edit page
        Response.Redirect("AnAddress.aspx");
    }
    else//if no record has been selected
    {
        //display an error
        lblError.Text = "Please select a record to delete from the list";
    }
}
```

Note as with the Delete event handler we first check to see if the list has been selected.

Assuming it has we then place the primary key value into the session object.

Then we redirect to the page AnAddress.aspx.

Compare the event handler operations for Add and Update.

For Add we place a -1 in the session object like so...

```
//store -1 into the session object to indicate this is a new record
Session["AddressNo"] = -1;
//redirect to the data entry page
Response.Redirect("AnAddress.aspx");
```

For Update we place the value of the primary key into the session object...

```
//store the data in the session object
Session["AddressNo"] = AddressNo;
//redirect to the edit page
Response.Redirect("AnAddress.aspx");
```

When we arrive in the page AnAddress.aspx we may use this to identify if the web form needs to add a new record or update an existing record.

The following code in the load event obtains the primary key value...

```
//variable to store the primary key with page level scope
Int32 AddressNo;

//event handler for the page load event
protected void Page_Load(object sender, EventArgs e)
{
    //get the number of the address to be processed
    AddressNo = Convert.ToInt32(Session["AddressNo"]);
    if (IsPostBack == false)
    {
        //populate the list of counties
        DisplayCounties();
    }
}
```

Notice how the AddressNo variable is declared with page level scope.

Here is the Update function...

```
//function for updateing records
1 reference
void Update()
{
    //create an instance of the address book
    Class_Library.clsAddressCollection AddressBook = new Class_Library.clsAddressCollection();
    //validate the data on the web form
    String Error = AddressBook.ThisAddress.Valid(txtHouseNo.Text, txtStreet.Text, txtTown.Text, txtPostCode.Text, txtDateAdded.Text);
    //if the data is OK then add it to the object
    if (Error == "")
    {
        //find the record to update
        AddressBook.ThisAddress.Find(AddressNo);
        //get the data entered by the user
        AddressBook.ThisAddress.HouseNo = txtHouseNo.Text;
        AddressBook.ThisAddress.Street = txtStreet.Text;
        AddressBook.ThisAddress.Town = txtTown.Text;
        AddressBook.ThisAddress.PostCode = txtPostCode.Text;
        AddressBook.ThisAddress.DateAdded = Convert.ToDateTime(txtDateAdded.Text);
        AddressBook.ThisAddress.Active = chkActive.Checked;
        AddressBook.ThisAddress.CountyNo = Convert.ToInt32(ddlCounty.SelectedValue);
        //update the record
        AddressBook.Update();
        //all done so redirect back to the main page
        Response.Redirect("Default.aspx");
    }
    else
    {
        //report an error
        lblError.Text = "There were problems with the data entered " + Error;
    }
}
```

Here is the event handler for the OK button...

```
//event handler for the ok button
0 references
protected void btnOK_Click(object sender, EventArgs e)
{
    if (AddressNo == -1)
    {
        //add the new record
        Add();
    }
    else
    {
        //update the record
        Update();
    }
}
```

We are almost there!

We still have one last issue to address.

When the form AnAddress.aspx is displayed for editing a record it isn't much good if it doesn't display the existing data...

House No

Street

Town

Post Code

County Avon

Date Added

☐ Active

OK Cancel

The following function solves this problem...

```
void DisplayAddress()
{
    //create an instance of the address book
    clsAddressCollection AddressBook = new clsAddressCollection();
    //find the record to update
    AddressBook.ThisAddress.Find(AddressNo);
    //display the data for this record
    txtHouseNo.Text = AddressBook.ThisAddress.HouseNo;
    txtStreet.Text = AddressBook.ThisAddress.Street;
    txtTown.Text = AddressBook.ThisAddress.Town;
    txtPostCode.Text = AddressBook.ThisAddress.PostCode;
    txtDateAdded.Text = AddressBook.ThisAddress.DateAdded.ToString();
    chkActive.Checked = AddressBook.ThisAddress.Active;
    ddlCounty.SelectedValue = AddressBook.ThisAddress.CountyNo.ToString();
}
```

Also we only want to display the existing data when the form is first loaded, not once the user has entered some new data.

We will need to modify the load event like so...

```
//event handler for the page load event
protected void Page_Load(object sender, EventArgs e)
{
    //get the number of the address to be processed
    AddressNo = Convert.ToInt32(Session["AddressNo"]);
    if (IsPostBack == false)
    {
        //populate the list of counties
        DisplayCounties();
        //if this is not a new record
        if (AddressNo != -1)
        {
            //display the current data for the record
            DisplayAddress();
        }
    }
}
```

You should now have a fully implemented set of classes with an associated test framework.

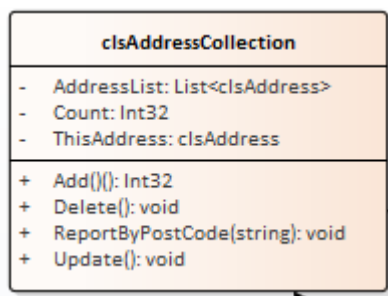
Are we done yet?

Not really there is still plenty of work to complete.

Creating the Post Code Filter

Filtering/reporting data is a really important part of any system. If we have 10,000 records we do not want to make the user trawl through them manually. There needs to be mechanisms to limiting the data and seeing only the records of interest.

The last method we will create is the method ReportByPostCode...



As with all previous examples it's business as usual.

We need to start with a test to ensure that the method exists in the class like so...

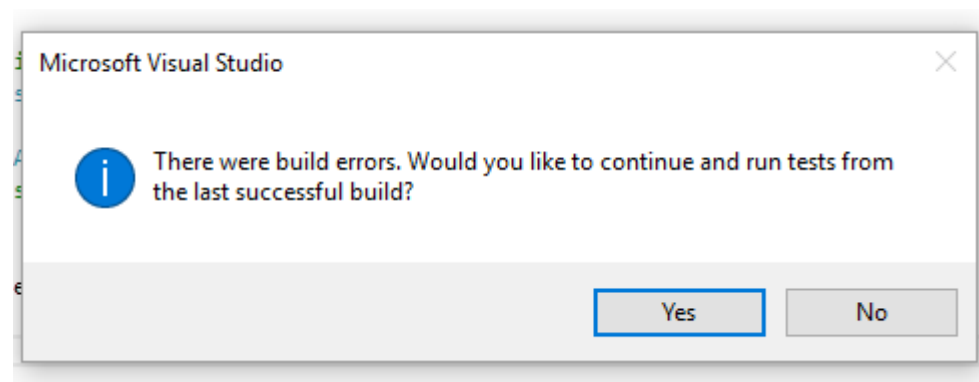

```

[TestMethod]
References
public void ReportByPostCodeMethodOK()
{
    //create an instance of the class containing unfiltered results
    clsAddressCollection AllAddresses = new clsAddressCollection();
    //create an instance of the filtered data
    clsAddressCollection FilteredAddresses = new clsAddressCollection();
    //apply a blank string (should return all records);
    FilteredAddresses.ReportByPostCode("");
    //test to see that the two values are the same
    Assert.AreEqual(AllAddresses.Count, FilteredAddresses.Count);
}

```

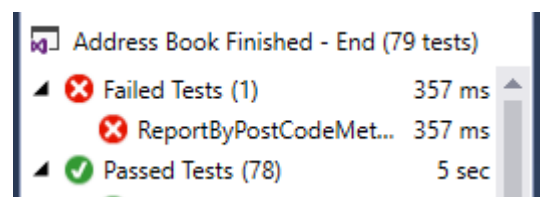
In this test we are applying a blank string to the filtered data. This filter should produce all results. By using a second instance of `clsAddressCollection` we may compare the two. A filter of blank string should produce the same count of records as the unfiltered results.

Run the tests and watch it fail.



Now fix the test by generating the method stub.

It should still fail...



Look at the function for the method and note the code you need to get rid of...

```

1 reference | 0/1 passing
public void ReportByPostCode(string PostCode)
{
    throw new NotImplementedException();
}

```

Lastly tidy up the parameter so that it makes more sense.

```

1 reference | 0/1 passing
public void ReportByPostCode(string PostCode)
{
    //filters the records based on a full or partial post code
}

```

The test should now pass.

Having applied a test that looks for all records we shall apply a test that should produce no records.

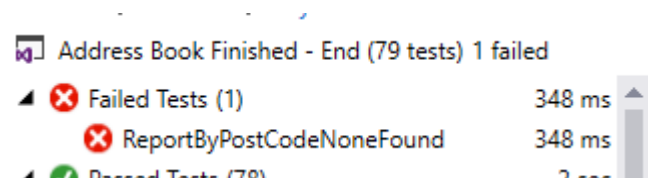
```

[TestMethod]
0 references
public void ReportByPostCodeNoneFound()
{
    //create an instance of the filtered data
    clsAddressCollection FilteredAddresses = new clsAddressCollection();
    //apply a post code that doesn't exist
    FilteredAddresses.ReportByPostCode("xxx xxx");
    //test to see that there are no records
    Assert.AreEqual(0, FilteredAddresses.Count);
}

```

(For this to work we need to make sure that the test data doesn't contain a post code "xxx xxx")

Run the test and watch it fail...



We now need to think about how to make the function work properly.

At this stage in the game we may as well cut to the chase.

Assume we have a stored procedure called `sproc_tblAddress_FilterByPostCode...`

```

CREATE PROCEDURE [dbo].sproc_tblAddress_FilterByPostCode
--this stored procedure uses the like function to find post codes that match the value
--stored in the parameter @PostCode
--the stored procedure doesn't return a value

    --declare the parameter as varchar(8)
    @PostCode varchar(9)

AS
--select all fields from any records that have a post code like this post code
select * from tblAddress where PostCode like @PostCode+'%';

```

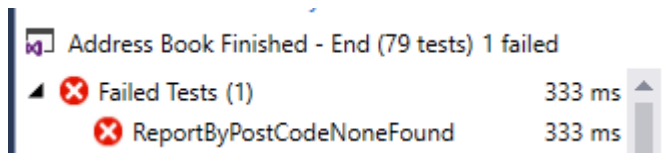
The following code should make the filter work...

```

2 references | 1/2 passing
public void ReportByPostCode(string PostCode)
{
    //filters the records based on a full or partial post code
    //connect to the database
    clsDataConnection DB = new clsDataConnection();
    //send the PostCode parameter to the database
    DB.AddParameter("@PostCode", PostCode);
    //execute the stored procedure
    DB.Execute("sproc_tblAddress_FilterByPostCode");
}

```

Run the tests...



Watch the test fail again!

What is the problem now?

If we look at the code in the constructor that generates the collection we see the following...

```

//constructor for the class
public clsAddressCollection()
{
    //var for the index
    Int32 Index = 0;
    //var to store the record count
    Int32 RecordCount = 0;
    //object for data connection
    clsDataConnection DB = new clsDataConnection();
    //execute the stored procedure
    DB.Execute("sproc_tblAddress_SelectAll");
    //get the count of records
    RecordCount = DB.Count;
    //while there are records to process
    while (Index < RecordCount)
    {
        //create a blank address
        clsAddress AnAddress = new clsAddress();
        //read in the fields from the current record
        AnAddress.Active = Convert.ToBoolean(DB.DataTable.Rows[Index]["Active"]);
        AnAddress.AddressNo = Convert.ToInt32(DB.DataTable.Rows[Index]["AddressNo"]);
        AnAddress.CountyNo = Convert.ToInt32(DB.DataTable.Rows[Index]["CountyNo"]);
        AnAddress.DateAdded = Convert.ToDateTime(DB.DataTable.Rows[Index]["DateAdded"]);
        AnAddress.HouseNo = Convert.ToString(DB.DataTable.Rows[Index]["HouseNo"]);
        AnAddress.PostCode = Convert.ToString(DB.DataTable.Rows[Index]["PostCode"]);
        AnAddress.Street = Convert.ToString(DB.DataTable.Rows[Index]["Street"]);
        AnAddress.Town = Convert.ToString(DB.DataTable.Rows[Index]["Town"]);
        //add the record to the private data member
        mAddressList.Add(AnAddress);
        //point at the next record
        Index++;
    }
}

```

Here we are populating the private array list mAddressList with the data from the data table in the DB object.

When we apply our post code filter...

```

2 references | 1/2 passing
public void ReportByPostCode(string PostCode)
{
    //filters the records based on a full or partial post code
    //connect to the database
    clsDataConnection DB = new clsDataConnection();
    //send the PostCode parameter to the database
    DB.AddParameter("@PostCode", PostCode);
    //execute the stored procedure
    DB.Execute("sproc_tblAddress_FilterByPostCode");
}

```

The data in this DB object is not being sent to the private array list.

The neatest way to fix this is as follows.

Firstly create a new function called PopulateArray.

```

void PopulateArray(clsDataConnection DB)
{
    //populates the array list based on the data table in the parameter DB
}

```

Notice how this function accepts a parameter called DB of type clsDataCollection.

Now add the code to the function like so...

```

void PopulateArray(clsDataConnection DB)
{
    //populates the array list based on the data table in the parameter DB
    //var for the index
    Int32 Index = 0;
    //var to store the record count
    Int32 RecordCount;
    //get the count of records
    RecordCount = DB.Count;
    //clear the private array list
    mAddressList = new List<clsAddress>();
    //while there are records to process
    while (Index < RecordCount)
    {
        //create a blank address
        clsAddress AnAddress = new clsAddress();
        //read in the fields from the current record
        AnAddress.Active = Convert.ToBoolean(DB.DataTable.Rows[Index]["Active"]);
        AnAddress.AddressNo = Convert.ToInt32(DB.DataTable.Rows[Index]["AddressNo"]);
        AnAddress.CountyNo = Convert.ToInt32(DB.DataTable.Rows[Index]["CountyNo"]);
        AnAddress.DateAdded = Convert.ToDateTime(DB.DataTable.Rows[Index]["DateAdded"]);
        AnAddress.HouseNo = Convert.ToString(DB.DataTable.Rows[Index]["HouseNo"]);
        AnAddress.PostCode = Convert.ToString(DB.DataTable.Rows[Index]["PostCode"]);
        AnAddress.Street = Convert.ToString(DB.DataTable.Rows[Index]["Street"]);
        AnAddress.Town = Convert.ToString(DB.DataTable.Rows[Index]["Town"]);
        //add the record to the private data member
        mAddressList.Add(AnAddress);
        //point at the next record
        Index++;
    }
}

```

We now have a function that is dedicated to the task of copying whatever data is in a Data Connection to the private array list.

We need to modify the constructor like so...

```

//constructor for the class
public clsAddressCollection()
{
    //object for data connection
    clsDataConnection DB = new clsDataConnection();
    //execute the stored procedure
    DB.Execute("sproc_tblAddress_SelectAll");
    //populate the array list with the data table
    PopulateArray(DB);
}

```

And now modify the filter like so...

2 references | 1/2 passing

```
public void ReportByPostCode(string PostCode)
{
    //filters the records based on a full or partial post code
    //connect to the database
    clsDataConnection DB = new clsDataConnection();
    //send the PostCode parameter to the database
    DB.AddParameter("@PostCode", PostCode);
    //execute the stored procedure
    DB.Execute("sproc_tblAddress_FilterByPostCode");
    //populate the array list with the data table
    PopulateArray(DB);
}
```

Since both function make use of the same code to populate the array list via the PopulateArray function the data should now be updated correctly.

The test should now pass!

We are now able to establish that a blank filter is producing the correct number of records. We also know that an invalid filter produces zero records. However this doesn't tell us if they are the correct records.

The final test to perform would be to create a couple of test records in the table that have the same post code.

	AddressNo	HouseNo	Street	Town	PostCode	CountyNo	DateAdded	Active
	2	1	Some Street	Leicester	LE1 1BE	35	07/09/2012	True
	3	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
	4	33	High Street	Leicester	LE1 6FG	35	07/08/2012	True
	5	22	The Road	Nottingham	N19 6EF	48	07/08/2012	True
	34	123a	some street	some town	LE1 1WE	1	20/05/2016	True
	36	123b	another street	another town	yyy yyy	3	20/05/2016	False
	37	123a	some street	some town	yyy yyy	1	20/05/2016	True
	39	123b	another street	another town	LE2 2WE	3	20/05/2016	False
▶	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

If we filter on this data we should obtain these records and these two records only.

The following test makes a start on this...

```

[TestMethod]
0 references
public void ReportByPostCodeTestDataFound()
{
    //create an instance of the filtered data
    clsAddressCollection FilteredAddresses = new clsAddressCollection();
    //var to store outcome
    Boolean OK = true;
    //apply a post code that doesn't exist
    FilteredAddresses.ReportByPostCode("yyy yyy");
    //check that the correct number of records are found
    if (FilteredAddresses.Count == 2)
    {
        //check that the first record is ID 36
        if (FilteredAddresses.AddressList[0].AddressNo != 36)
        {
            OK = false;
        }
        //check that the first record is ID 37
        if (FilteredAddresses.AddressList[1].AddressNo != 37)
        {
            OK = false;
        }
    }
    else
    {
        OK = false;
    }
    //test to see that there are no records
    Assert.IsTrue(OK);
}

```

This test only looks at the primary key values and doesn't check the data of the other fields.

Having got this far there is still a lot to do. The filtering needs attaching to the presentation layer and you will also need to re-factor your code to make it as efficient as possible.