Lab 4: Unit Tests

Programming in C# for Visual Studio .NET Platform II

Objective

In this exercise, we will introduce MSTest, Microsoft's Unit Testing Framework that is built into Visual Studio.

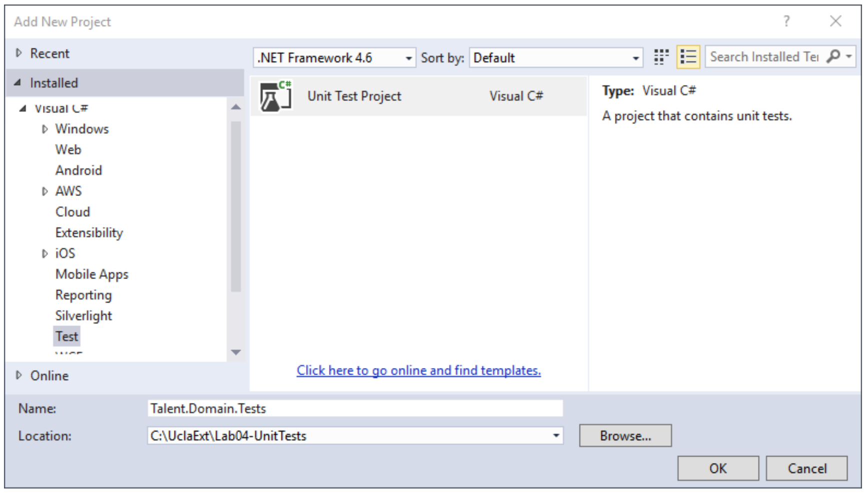
Unit Tests are short pieces of test programs that each test one particular feature of your application – ideally in isolation from the rest of the application.

The point of this exercise is primarily to introduce the concept of testable software and demonstrate the basic idea of unit testing with Visual Studio's built-in testing framework.

Step 1: Download the solution to the Lab 1 from the GitHub repository at https://github.com/entrotech/Lab01-DomainModel. Unzip this to your hard drive and rename the folder to Lab04-UnitTests.

Step 2: Open the solution and remove the Console application project entirely. The application should still compile, but there is no executable to run, so you will not be able to run the solution in the debugger.

Step 3: Create a unit test project. Select the solution node in Solution Explorer, right click and choose Add | New Project from the context menu. There should be a C# template for Unit Test Project. We are going to create a set of tests to exercise the Talent.Domain project, so we will call the Unit Test Project Talent.Domain.Tests.



Type: Visual C#

A project that contains unit tests.

Name: Talent.Domain.Tests

Location: C:\UclaExt\Lab04-UnitTests

Make sure the unit test project gets placed in the Lab04-UnitTests folder with the rest of the solution, and press OK to create the unit test project. You should then see a new project called Talent.Domain.Tests in SE with a source code file called UnitTest1.cs.

If you check the properties of this new project, you will see that it is just a Class Library. The thing that makes it a Unit Test project is that is it set up with a reference to the Microsoft.VisualStudio.QualityTools.UnitTestFramework, and VS creates a starter file UnitTest1.cs to get you started. The UnitTest1.cs file is actually helpful, since it provides a start on creating your first test.

Step 4: Add a Project Reference to Talent.Domain. Select the Talent.Domain.Tests project node in SE and add a project reference to the Talent.Domain project.

Step 5: Create a unit test for Person. Re-name the UnitTest1.cs file to PersonTest, since it will contain all our tests for the Person class. Make sure that the source code file and the class name are both changed to PersonTest. Open the text editor for the PersonTest, and you will see that it is pretty much just a plain old C# class with a method in it. Re-name the method to Person_Instantiate_CreatesObject. I will be following a simple naming convention that is of the form <Class Under Test>_<Action Taken>_<Expected Result>, though there are several other common conventions in common use.

At this point, the PersonTest.cs file should look like this:

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

namespace Talent.Domain.Tests
{
    [TestClass]
    public class PersonTest
    {
        [TestMethod]
        public void Person_Instantiate_CreatesObject()
        {

        }
    }
}
```

The [TestClass] attribute above the class declaration and the [TestMethod] attribute above the method declaration allow the MSTest framework to locate classes and bits of code that the framework should run.

A unit test is usually written in three parts:

- "Arrange" is some code that performs any setup required to prepare for the test.
- "Act" is code that attempts to perform the steps to be tested, and
- "Assert" is some code that analyzes the post-test conditions to see if they meet expectations.

Our first test will just test that the default constructor for a Person creates a Person object. A good practice when writing unit tests is to first write a test that fails to make sure that the test will fail if the desired outcome is not achieved, so a failing test for this test would occur if the

Person object was not created. We can do this by just omitting the call to the constructor we will want to test like this:

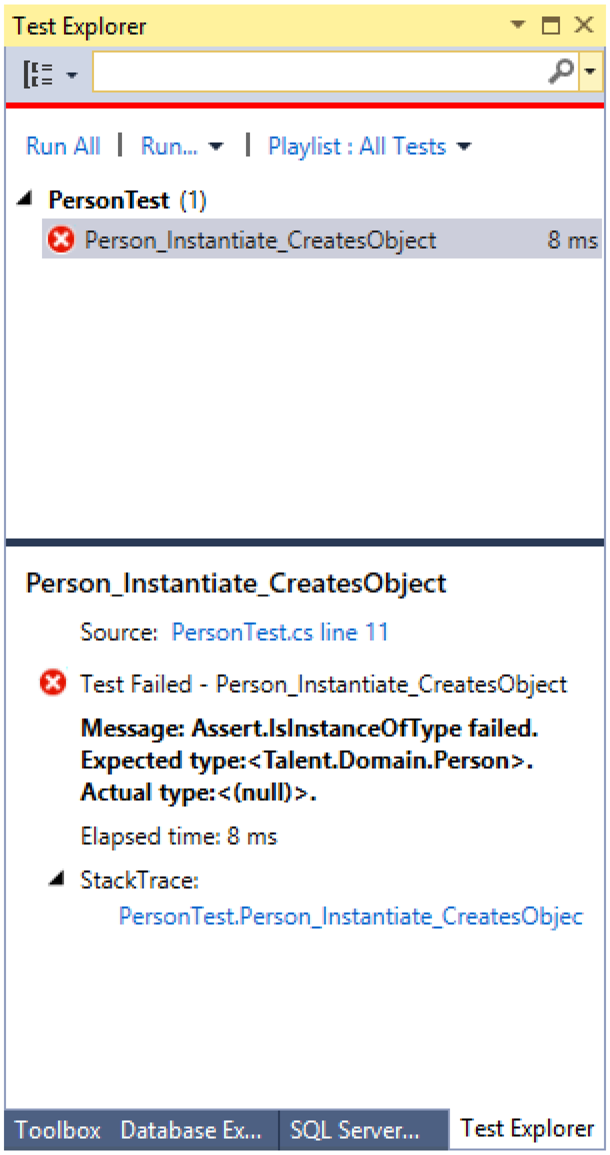
```
[TestMethod]
public void Person_Instantiate_CreatesObject()
{
    // Arrange
    Person p = null;

    //Act

    // Assert
    Assert.IsInstanceOfType(p, typeof(Person));
}
```

The Assert phase usually consists of one or more calls to static methods on the Assert class from the MSTest framework that assert expected conditions – in this case that p is assigned to an instance of a Person. MSTest will call a Unit Test "passed" if all the assertions are satisfied, or call the test "failed" if one or more assertions are not satisfied.

Step 6: Run the Unit Test. You will notice that our solution does not have any projects that are executable as an application. When you run Unit Tests, Visual Studio acts as the executable application. MSTest will search the application for classes marked with the [TestClass] attribute to find Unit Test Classes. From the TEST menu on the VS toolbar, choose Windows |Test Explorer to open the Test Explorer window. If your solution has been compiled, TE will show a list of all the unit test methods that it finds in your solution. The outline icon on the toolbar allows you to group the unit tests by various criteria. The Run All link allows you to run all the test in your solution, and then there are a few other options to run various combinations of the tests. You should only have one test shown, so you can just press the Run All link to execute the test. After running the unit test, here is what my TE window looked like:


Test Explorer
Run All | Run... | Playlist : All Tests
PersonTest (1)
Person_Instantiate_CreatesObject 8 ms
Toolbox
Database Ex...
SQL Server...
Test Explorer

Person_Instantiate_CreatesObject

Source: PersonTest.cs line 11

Test Failed - Person_Instantiate_CreatesObject

Message: Assert.IsInstanceOfType failed. Expected type:<Talent.Domain.Person>. Actual type:<(null)>.

Elapsed time: 8 ms

StackTrace:
PersonTest.Person_Instantiate_CreatesObjec

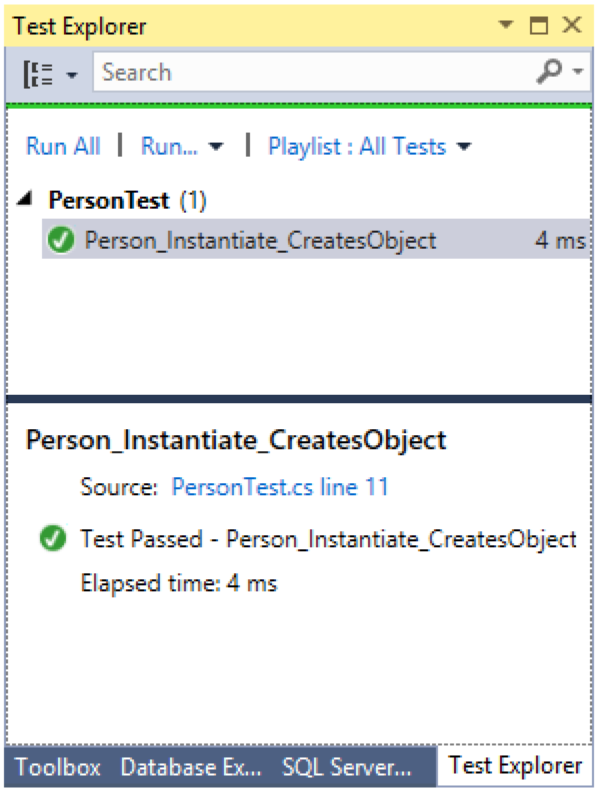
The X indicates that my unit test failed. When you select a test in the upper pane, the lower pane will give you more details. You can also debug unit tests by setting a breakpoint in the unit test source code, then selecting a test in the TE, right-clicking on it and choosing “Debug Selected Tests” from the context menu. You can then step through the code of the unit test (and any code under test that is called by your unit test code, just as you would when debugging any other source code.

Step 7: Making the first test pass. We expected the unit test to fail, since we did not actually call the constructor. So let’s just add a call to the constructor:

```
[TestMethod]
public void Person_Instantiate_CreatesObject()
{
    // Arrange
    Person p = null;
    //Act
    p = new Person();

    // Assert
    Assert.IsInstanceOfType(p, typeof(Person));
}
```

Now you can press Run All in TE and verify that the test passes:



```
PersonTest (1)
  Person_Instantiate_CreatesObject    4 ms

Person_Instantiate_CreatesObject
    Source: PersonTest.cs line 11
    Test Passed - Person_Instantiate_CreatesObject
    Elapsed time: 4 ms
```

Notice that the test ran for me in 4ms.

So that is how we can create and run a basic unit test. So let's try a few more.

Step 8: A test of the instantiated properties. To create more tests of the same class, we can just add additional methods to the PersonTest class, making sure to use the [TestMethod] attribute on each, so MSTest can identify the method as a unit test. Let's create a test that checks that the constructor initialized the public properties the way we want. First add a new method called Person_Instantiate_SetsPublicProperties(). First, set it up so the test will fail. This just makes sure that we are getting our assertions right:

```
[TestMethod]
public void Person_Instantiate_SetsDefaultProperties()
{
    // Arrange
    Person p = null;

    // Act
    p = new Person();

    // Assert
    Assert.IsTrue(p.Salutation == "Something");
}
```

You can run all the tests and see that our first test passed again and this test failed as we expected. Now we can change our test to Assert that Salutation is set to null:

```
[TestMethod]
public void Person_Instantiate_SetsPulicProperties()
{
    // Arrange
    Person p = null;

    // Act
    p = new Person();

    // Assert
    Assert.IsTrue(p.Salutation == null);
}
```

If we run our test again, we see that this test passes. This verifies that the Salutation property is set to an empty string – recall that we wrote the Person class to initialize each string property, rather than leaving them null.

Unit Tests have a tendency to flush out incomplete or ambiguous specifications, and you can really take the view that Unit Tests **are** your detailed requirements, expressed in a fashion that you can always verify very quickly whether your application meets all these detailed requirements by simply executing all the unit tests – how cool is that?

In principle, you should have a completely separate unit test for each tiny detail of your classes behavior, but you should exercise common sense on this front. Rather than creating separate unit tests to validate that each property is initialized as I expect, I think its fine to expand the current test by adding assertions for each property to this single unit test, so it checks that all Person properties are initialized as expected:

```
[TestMethod]
public void Person_Instantiate_SetsDefaultProperties()
{
    // Arrange
    Person p = null;

    // Act
    p = new Person();

    // Assert
    Assert.IsTrue(p.Salutation == null);
    Assert.IsTrue(p.FirstName == null);
    Assert.IsTrue(p.MiddleName == null);
    Assert.IsTrue(p.LastName == null);
    Assert.IsTrue(p.Suffix == null);
    Assert.IsTrue(p.DateOfBirth == null);
    Assert.IsTrue(p.Height == null);
    Assert.IsTrue(p.HairColorId == 0);
    Assert.IsTrue(p.EyeColorId == 0);
    Assert.IsNotNull(p.Credits);
    Assert.IsTrue(p.Credits.Count == 0);
    Assert.IsNotNull(p.PersonAttachments);
    Assert.IsTrue(p.PersonAttachments.Count == 0);
    Assert.IsTrue(p.FirstLastName == String.Empty);
    Assert.IsTrue(p.LastFirstName == String.Empty);
    Assert.IsTrue(p.FullName == String.Empty);
    Assert.IsTrue(p.Age == null);
    Assert.IsTrue(p.ToString() == "");
}
```

Look through these assertions and see if that is what you would have expected each property to have as a starting value.

A common mantra in Unit Testing is “Red-Green-Refactor”, which refers to this cycle of writing a unit test that fails because a feature is not implemented or included in the test, then fixing the code under test or the unit test itself to pass, then moving on to the next requirement to implement. Some folks take this practice to the extreme with a practice called **Test Driven Development.** The idea is that they start by writing a unit test that describes a feature before they write a single line of code to implement the feature itself. Then they write the absolute minimum amount of code that gets the unit test to pass before moving on to write another unit test describing the next requirement of the unit. Then they add the absolute minimum amount of code or refactor existing code in the class under test that makes both tests pass. They continue doing this until they can think of no more requirements to add in the form of unit tests, and what they are left with is a unit that meets all its requirements with the minimum amount of code, and a set of requirements in the form of executable unit tests that prove compliance with the requirements.

We’re not going to go that far, but we do want to add a few more tests for the Person class. Part of the “Art” of Unit Testing is deciding what we really need to have unit tests for. For example, we could test that each of the property setters and getters works by setting properties to some value, then retrieving them to make sure we get the same value back. We probably don’t need to do this for properties where we haven’t added any logic to the getters and setter, but recall that we have added a bit of logic in each or the setter for our string properties to set the property to null if a consumer of the class attempts to set the property value to null.

Step 9: Testing the computed name properties. The LastFirstName, FirstLastName and FullName properties do have some logic in their computations, so we should have unit tests for them.

```
[TestMethod]
public void Person_AllNameParts_ComputedNamesAreCorrect()
{
    // Arrange
    Person p = null;

    p = new Person()
    {
        Salutation = "Mr.",
        FirstName = "George",
        MiddleName = "A",
        LastName = "Jetson",
        Suffix = "Sr."
    };

    // Assert
    Assert.IsTrue(p.FirstLastName == "George Jetson");
    Assert.IsTrue(p.LastFirstName == "Jetson, George");
    Assert.IsTrue(p.FullName == "Mr. George A Jetson, Sr.");
}

[TestMethod]
public void Person_NoFirstName_ComputedNamesAreCorrect()
{
    // Arrange
    Person p = null;

    p = new Person()
    {
        Salutation = "Mr.",
        MiddleName = "A",
        LastName = "Jetson",
        Suffix = "Sr."
    };

    // Assert
    Assert.IsTrue(p.FirstLastName == "Jetson");
    Assert.IsTrue(p.LastFirstName == "Jetson");
    Assert.IsTrue(p.FullName == "Mr. A Jetson, Sr.");
}

[TestMethod]
public void Person_NoLastName_ComputedNamesAreCorrect()
{
    // Arrange
    Person p = null;

    p = new Person()
    {
        Salutation = "Mr.",
        FirstName = "George",
        MiddleName = "A",
        Suffix = "Sr."
    };

    // Assert
    Assert.IsTrue(p.FirstLastName == "George");
    Assert.IsTrue(p.LastFirstName == "George");
    Assert.IsTrue(p.FullName == "Mr. George A, Sr.");
}

[TestMethod]
public void Person_NoFirstOrLastName_ComputedNamesAreCorrect()
{
    // Arrange
    Person p = null;

    p = new Person()
    {
        Salutation = "Mr.",
        MiddleName = "A",
        Suffix = "Sr."
    };

    // Assert
    Assert.IsTrue(p.FirstLastName == "");
    Assert.IsTrue(p.LastFirstName == "");
    Assert.IsTrue(p.FullName == "Mr. A, Sr.");
}

[TestMethod]
public void Person_OnlyFirstName_ComputedNamesAreCorrect()
{
    // Arrange
    Person p = null;

    p = new Person()
    {
        FirstName = "George"
    };

    // Assert
    Assert.IsTrue(p.FirstLastName == "George");
    Assert.IsTrue(p.LastFirstName == "George");
    Assert.IsTrue(p.FullName == "George");
}

[TestMethod]
public void Person_OnlyLastName_ComputedNamesAreCorrect()
{
    // Arrange
    Person p = null;

    p = new Person()
    {
        FirstName = "Jetson"
    };

    // Assert
    Assert.IsTrue(p.FirstLastName == "Jetson");
    Assert.IsTrue(p.LastFirstName == "Jetson");
    Assert.IsTrue(p.FullName == "Jetson");
}
```

Add these unit tests to your project and make sure they run successfully.

Step 10: Testing the Age Property. The Age property is the hardest to test effectively because the result depends not only on the DateOfBirth, but also on the Date the test is run:

```
public int? Age
{
    get
    {
        if (DateOfBirth.HasValue == false) return null;
        var today = DateTime.Today;
        int years = today.Year - DateOfBirth.Value.Year;
        if (
            (DateOfBirth.Value.Date.Month > today.Month)
            || (DateOfBirth.Value.Date.Month == today.Month
                && DateOfBirth.Value.Date.Day > today.Day))
        {
            years--;
        }
        return years >= 0 ? years : 0;
    }
}
```

The problem is that this code uses the DateTime.Today static property, and this is "hard-wired" to report the actual current date. We probably do not need to test that DateTime.Today does indeed return the correct current date, but we would ideally like to make sure that our calculation is correct for various combinations of current date and DateOfBirth.

To really do this thoroughly, we would need to rewrite the Age calculation to get the current date using an interface or abstract class, perhaps an interface called ICurrentDate. The Person class would then use this interface to obtain the current date. When run in a production environment, the Person instance would we wired to an object that implements the ICurrentDate interface such that it returns the actual date. For unit testing purposes, the Person instance under test would be wired up to an object that implements the ICurrentDate but returns whatever date we want to use as the current date for unit testing purposes.

I decided to do something a bit simpler for my first unit test of the Age property by choosing a DateOfBirth exactly ten years before the current date, to see if the calculation returns 10:

```
[TestMethod]
public void Person_AgeExactly10_Returns10()
{
    // Arrange
    Person p = null;

    p = new Person()
    {
        DateOfBirth = DateTime.Today.AddYears(-10)
    };

    // Assert
    Assert.IsTrue(p.Age == 10);
}
```

I made sure the test failed if I commented out the DateOfBirth assignment first, but then uncommented it and the test ran successfully.

Then I figured it would be good to test what happens if the DateOfBirth is one day less than ten years before the current date, just to make sure I get 9:

```
[TestMethod]
public void Person_AgeLessThan10_Returns9()
{
    // Arrange
    Person p = null;

    p = new Person()
    {
        DateOfBirth = DateTime.Today.AddYears(-10).AddDays(1)
    };

    // Assert
    Assert.IsTrue(p.Age == 9);
}
```

The first time I used this sample project and unit test, this test failed. I wish I could say that I had intentionally set this up to make the unit test interesting, but the fact is that it was an unintentional error on my part in the Age calculation.

Finally, I wrote a quick unit test to confirm that the Age will not be negative if DateOfBirth is after the Current Date:

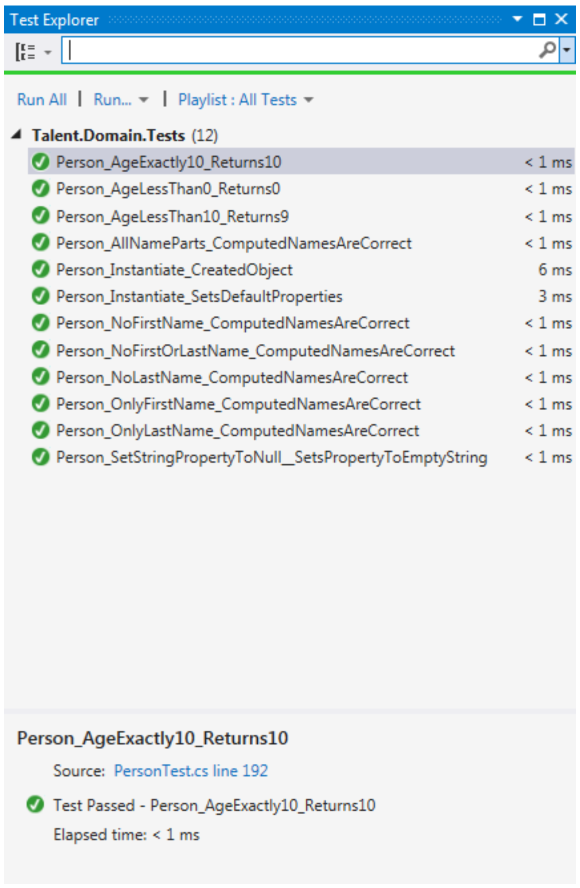
```
[TestMethod]
public void Person_AgeLessThan0_Returns0()
{
    // Arrange
    Person p = null;

    p = new Person()
    {
        DateOfBirth = DateTime.Today.AddYears(45)
    };

    // Assert
    Assert.IsTrue(p.Age == 0);
}
```

You could argue that we shouldn't allow entering a DateOfBirth in the future. If we want to enforce that rule, we could add some validation to our application to prevent entry of an invalid DateOfBirth.

Now I have a nice little suite of Unit Tests that I can run simply by pressing the Run All link to make sure that any changes I might have made do not break any of the requirements of my Person class:



```
Talent.Domain.Tests (12)
  Person_AgeExactly10_Returns10                          < 1 ms
  Person_AgeLessThan0_Returns0                           < 1 ms
  Person_AgeLessThan10_Returns9                          < 1 ms
  Person_AllNameParts_ComputedNamesAreCorrect            < 1 ms
  Person_Instantiate_CreatedObject                         6 ms
  Person_Instantiate_SetsDefaultProperties                 3 ms
  Person_NoFirstName_ComputedNamesAreCorrect             < 1 ms
  Person_NoFirstOrLastName_ComputedNamesAreCorrect       < 1 ms
  Person_NoLastName_ComputedNamesAreCorrect              < 1 ms
  Person_OnlyFirstName_ComputedNamesAreCorrect           < 1 ms
  Person_OnlyLastName_ComputedNamesAreCorrect            < 1 ms
  Person_SetStringPropertyToNull_SetsPropertyToEmptyString < 1 ms

Person_AgeExactly10_Returns10
   Source: PersonTest.cs line 192
   Test Passed - Person_AgeExactly10_Returns10
   Elapsed time: < 1 ms
```

If you look at the number of lines of code in the PersonTest.cs file, you will see that we now have about as many lines of code in our unit tests as we have in the Person class itself. This may seem crazy, but think about how sure we are that the Person class behaves exactly the way we intended. If we do find a bug that slips through our unit tests, then we can just add another unit test for that bug, and we will catch it if it ever pops up again.

These all run very quickly, and with virtually no effort on our part, so there is no excuse not to run them after any change to your application.

You might be surprised how often you find that a unit test breaks in an area that you didn't think was related to the changes you made!

I currently work on a large project that has been under development for over five years where dozens of different programmers have worked on it at various times. I wish I could say that we could unit test all the parts of the project in a few minutes, but the fact of the matter is that it was not written from the start with unit tests, and we probably spend over half of our time tracking down bugs that would never have slipped through unit tests if we had them.

Step 11: Write some unit tests to verify that the default sorting works as intended. Here is a basic one that just tries one case where all the last names are different and verifies that the results are sorted by last name:

```
[TestMethod]
public void Person_Sort_LastNamesInOrder()
{
    List<Person> people = new List<Person>
    {
        new Person {LastName = "Smithwick" },
        new Person {LastName = "Smith" },
        new Person {LastName = "Rojas" }
    };

    // Act
    people.Sort();

    // Assert
    Assert.IsTrue(people[0].LastName == "Rojas");
    Assert.IsTrue(people[1].LastName == "Smith");
    Assert.IsTrue(people[2].LastName == "Smithwick");
}
```

See if you can add a few more tests that

1. Verify that when two people have the same last name, the people with the same last name are then sorted by first name.
2. Verify that if a person has a null LastName, then they appear before people that have a LastName.
3. If you have a person with a null LastName and a person with a LastName of "", which appears first and why? Is there really a difference to the user between a person with a null LastName and one with a LastName of ""? What kind of changes could we make to the Person class or Sort method to treat them as equivalent?
4. Verify that when two people have the same first and last name, then they are sorted by birthdate (or in increasing order by age.
5. Verify that when two people have the same first and last name, but only one has a DateOfBirth, the person with no DateOfBirth appears first.

Step 12: Try your hand at writing a set of Unit Tests for the Show domain object. Create a separate unit test class called ShowTest. You probably will only need a few of them – mainly because the Show class doesn't have any calculations.

Summary

Unless you have worked on non-trivial programming projects, unit tests may seem like more work than is really necessary – and this can be true for tiny demonstration applications that you write yourself for your own amusement, and for many of the classroom or conference presentations you might see.

However, when you work as a developer on real projects, especially larger projects with a team of developers, you will find that good unit test can save you a lot of pain in the long run. To recap:

- Unit tests allow you to verify that every requirement is satisfied if you write a good unit test for each unit requirement, consequently,
- The set of unit tests comprises a complete set of requirements that are guaranteed to be up to date.
- Unit tested components are much more likely to be bug-free, resulting in fewer and more easily detected problems when you start integrating the units into bigger subsystems.
- A good set of unit tests is a main pillar of regression testing – that is, testing as much of the application as possible after significant enhancements and before releases of the product, especially as the size and complexity of the application grows over time.
- Unit test do not need a User Interface. Instead, a test runner application runs the test for us in an automated fashion and analyze the results for correctness. This is a huge leap forward over what we were doing by writing one-off console applications, running them by hand, and staring at the program output, trying to determine if the results are what we expected.

So far, our domain objects only “live” as long as our application runs, since they are just C# objects in memory and disappear when our application shuts down. In the next lesson, we give our objects a much longer lifetime, by saving them to a data store, and allowing them to be retrieved later by our application. We’ll start by learning about entities and relational databases, and then make just a few adjustments to how we design our C# domain model classes, and work on passing objects to and from databases from our C# code.

Further Reading

We’ve just scratched the surface of Unit Testing. There is lots of material online, but a book I really like is:

The Art of Unit Testing with Examples in .NET. Roy Osherove, Manning, 2009.

Professional Test-Driven Develoment with C#. James Bender and Jeff McWherter, Wrox, 2011

Finished Solution

You can download my finished solution from the GitHub repo at https://github.com/entrotech/Lab04-UnitTests.