Introduction to Unit Testing with Visual Studio

Every developer needs to test their code, or have it tested by someone. Many developers are not great at testing their own code. The main reason is we tend to test only the “happy path” through the functionality that we wrote. We often avoid testing the boundaries of our code such as invalid inputs, exceptions that might occur, etc. One way to become a better tester is to start writing unit tests. While it takes more time up-front to write unit tests, it saves a ton of time when you must regression test changes to existing features.

Starting with Visual Studio 2008, Microsoft added a unit testing framework right into Visual Studio. There are also several third-part testing frameworks you may use. In this blog post, you are going to learn the basics of using the unit test framework in Visual Studio.

Our Method to Test

For this blog post, you are going to build a method that checks to see if a file exists. You are then going to build unit tests to check each type of input you can pass to this method.

To start, create a new Class Library project in Visual Studio using C#. Set the name of this class library project to **MyClasses**. Rename the Class1.cs file created by Visual Studio to **FileProcess**. Add a method in this class called FileExists as shown in the following code snippet.

```
public bool FileExists(string fileName) {
  if (string.IsNullOrEmpty(fileName)) {
    throw new ArgumentNullException("fileName");
  }

  return File.Exists(fileName);
}
```

This is a very simple method, yet it requires at least three unit test methods to ensure this method works with all possible inputs. The three possible values you can pass to the **fileName** parameter are:

- A file name that exists
- A file name that does not exist
- A null or empty string

Create a Test Project

Right mouse click on your MyClasses solution and choose **Add** | **New Project**. From the list of templates, click on the **Visual C#** | **Test** | **Unit Test Project**. Set the **Name** to **MyClassesTest**. Click the **OK** button. Rename the **UnitTest1.cs** file to **FileProcessTest.cs**. You are going to test the method in your MyClasses class library, so you need to add a reference to that project. Right mouse click on the References folder in the MyClassesTest project and select MyClasses. Add a using statement at the top of the FileProcessTest.cs file.

```
using MyClasses;
```

Create Stubs for all Tests

As you have identified three different tests, it is a good idea to go ahead and write all three methods right away so you don't forget what you want to test. The code shown below shows how you structure your test class.

```
[TestClass]
public class FileProcessTest
{
  [TestMethod]
  public void FileNameDoesExist() {
    Assert.Inconclusive();
  }

  [TestMethod]
  public void FileNameDoesNotExist() {
    Assert.Inconclusive();
  }

  [TestMethod]
  public void
     FileNameNullOrEmpty_ThrowsArgumentNullException() {
    Assert.Inconclusive();
  }
}
```

The first thing you notice about this class is the presence of the [TestClass] attribute before the class definition. This attribute informs the unit test framework that this class is one that can be included in the testing process. Next, you notice that each method is prefixed with a [TestMethod] attribute. Again, this is to inform the unit test framework that this is a test method that needs to run. Within each method, call the Assert.Inconclusive() method. This informs the testing framework that you have not written any code for this test method. This is not a success, or a failure, of the test. This shows up as a skipped test in the Test Explorer window. The advantage of using this method is this gives you a checklist of the tests that you still need to write.

After adding this code, right mouse click in your code window and choose **Run Tests** from the context-sensitive menu that appears. After the code runs, a Test Explorer window appears with the results of the test(s) as shown in Figure 1.

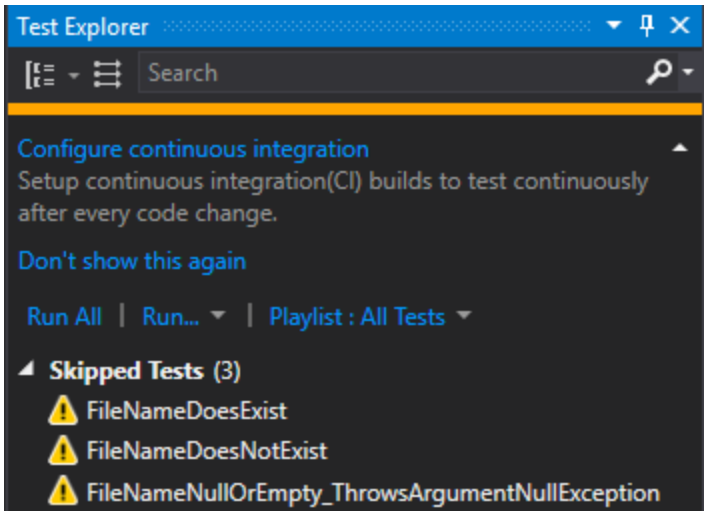

Figure 1: Skipped tests are the result of the Inconclusive method call

Write the Tests

It is now time to start writing the code in the unit tests to create each of the three possible inputs identified for this method. The first one is to test that a file exists. Modify the FileNameDoesExist method shown below. Feel free to change the drive letter, path and file name to a file that exists on your computer.

```
[TestMethod]
public void FileNameDoesExist() {
  FileProcess fp = new FileProcess();
  bool fromCall;

  fromCall = fp.FileExists(@"C:\Windows\Regedit.exe");

  Assert.IsTrue(fromCall);
}
```

After adding this code, right mouse click in your code window and choose **Run Tests** from the context-sensitive menu that appears. After the code runs, a Test Explorer window appears with the results of the test. If the file exists, the window should display something that looks like Figure 2.

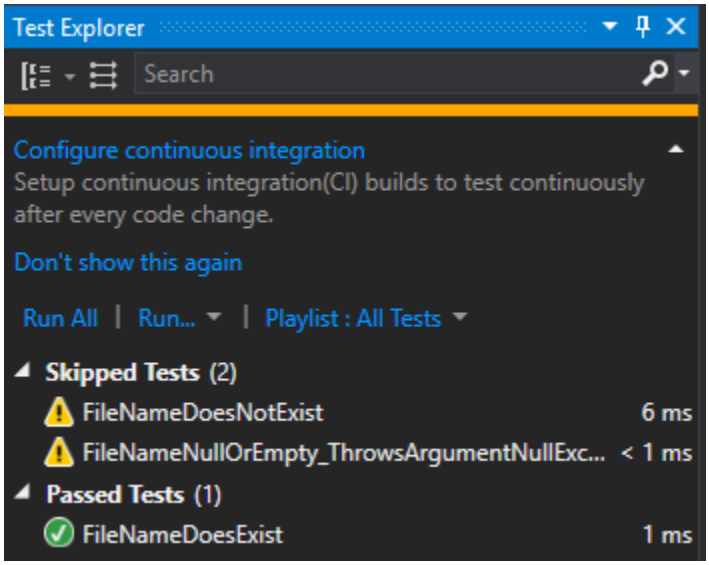

Figure 2: Test results appear in the Test Explorer window

The next method to write is to test for a file that does not exist. Modify the **FileNameDoesNotExist** method in your FileProcessTest class to test this condition. Write the code shown in the following code snippet.

```
[TestMethod]
public void FileNameDoesNotExist() {
  FileProcess fp = new FileProcess();
  bool fromCall;

  fromCall = fp.FileExists(@"C:\NotExists.bad");

  Assert.IsFalse(fromCall);
}
```

Once again, run these tests and you should now see two passed tests in your Test Explorer window.

Handling Exceptions

You should always test for any exceptions being thrown from your methods. There are two ways to handle a thrown exception; add a catch block in your test method or add an [ExpectedException] attribute. In the FileExists method an ArgumentNullException is thrown if a null or blank value is passed to the method. Let's take a look at the two ways to handle this thrown exception.

Add a test to the FileProcessTest class named **FileNameNullOrEmpty_CausesArgumentNullException_UsingAttribute**. Add an [ExpectedException] attribute after the [TestMethod] attribute on this method as shown in the following code snippet.

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void FileNameNullOrEmpty_
            CausesArgumentNullException_UsingAttribute () {
  FileProcess fp = new FileProcess();

  fp.FileExists("");
}
```

Run this test and you should see that this test passes.

The second way to handle this thrown exception is to wrap up the call to the FileExists method within a try…catch block in your test method. The catch block should check to see if the exception is a ArgumentNullException. If it is, then the test received the correct return value. If no exception was thrown, or any other kind of exception is returned from FileExists, call the Assert.Fail() method to let the unit test framework that this test failed. Modify the **FileNameNullOrEmpty_ThrowsArgumentNullException** method as shown in the following code snippet.

```
[TestMethod]
public void FileNameNullOrEmpty_ThrowsArgumentNullException()
{
  FileProcess fp = new FileProcess();

  try {
    fp.FileExists("");
  }
  catch (ArgumentNullException) {
    // Test was a success
    return;
  }

  // Fail the test
  Assert.Fail("Call to FileExists() did NOT throw
               an ArgumentNullException.");
}
```

Run the unit tests one more time, and you should now see four passed tests in the Test Explorer window.

Summary

Visual Studio makes it easy to get started creating unit tests. Take advantage of the unit test framework built-in to Visual Studio. It is important to think of as many ways as possible to break your code. Then, write all the unit tests to test that your code does not break. Use the ExpectedException attribute to help you with exceptions that are thrown from your methods.

Sample Code

You can download the code for this sample at www.pdsa.com/downloads. Choose the category “PDSA Blogs”, then locate the sample **Introduction to Unit Testing**.