

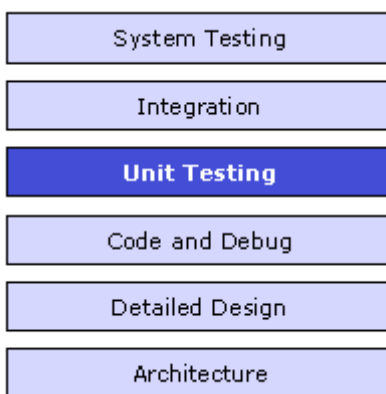
Getting Started With Unit Testing

Abstract

This article walks you through the process required to have a suite of tests run on your project during each build. It uses an open source project as its foundation called NUnit freely available at <http://nunit.sourceforge.net>

Introduction

Unit testing is the type of testing performed by the developers who write the code. It tests methods, properties, classes and assemblies. It is not the functional testing normally performed by an independent quality assurance department. A lot of people talk about unit testing, but by looking at the source code available in many open source projects the practice is rarely done. This article will give you some practical methods to start writing unit tests quickly and easily.

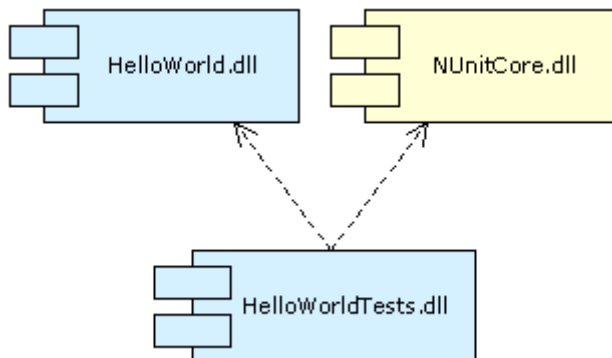


Where unit testing fits into the development cycle.

A Brief Overview of NUnit

In this article I will be using a testing framework called [NUnit](#) to demonstrate unit testing on the .NET platform. The reason for choosing NUnit was that it is a derivative of the popular testing framework used by eXtreme Programming (see links at the end of the article for more information about XP). This framework is available for most languages so that time spent learning the framework can be carried over to other environments. Specifically NUnit is a port of JUnit for .NET by Philip Craig. The current version is 1.10 and has been tested with the .NET SDK Beta 2. The software is available for free and comes with the full source code and documentation.

NUnit works by providing a class framework and test runner application. The class framework allows you to write test cases that inherit from **TestCase**. You package your **TestCase** classes into an assembly that is referred to as a Test Suite. The test runner uses your Test Suite assembly and runs all the tests displaying a message for each test case that fails.



Typical Test Case Dependencies

In the following sections we will cover the details of using NUnit for unit testing.

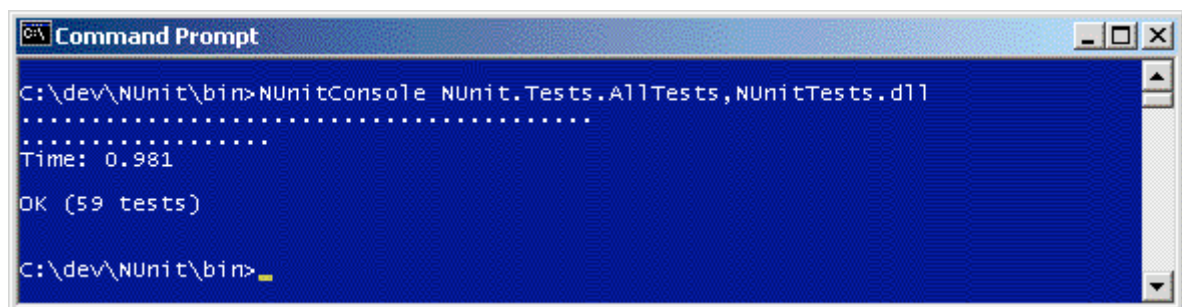
Installing the Software

Before you begin you first need to download NUnit. The package comes with an installer that will copy everything over for you. In the screen shots you will see that I have installed NUnit into `c:\dev\NUnit`. It is useful to install it into a shorter path than the default Program Files folder because you will need to add the bin subfolder to your path if you want to use the console based tools (highly recommended).

To verify that everything is working:

- Open a command prompt
- Change the current directory to the `NUnit\bin` folder
- Enter the following command: `NUnitConsole NUnit.Tests.AllTests,NUnitTests.dll`

You should see something like this:



```
Command Prompt
C:\dev\NUnit\bin>NUnitConsole NUnit.Tests.AllTests,NUnitTests.dll
.....
Time: 0.981
OK (59 tests)
C:\dev\NUnit\bin>
```

Successful test run of NUnit using the Console Test Runner.

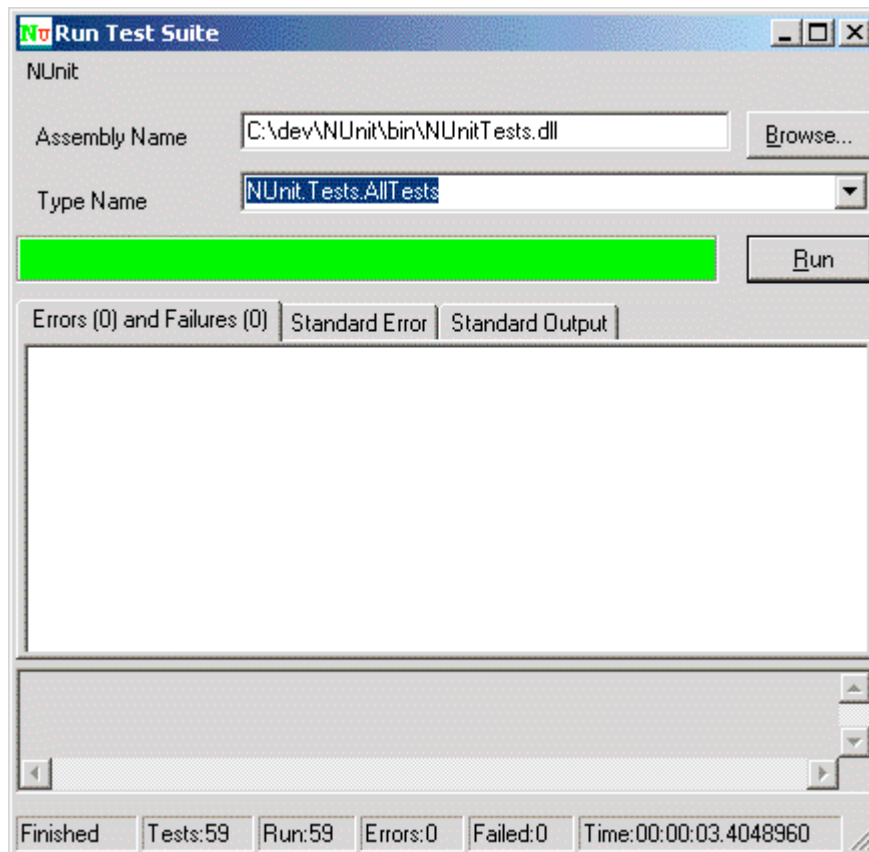
What you have just done is run the tests that come with NUnit on itself.

There is also a GUI front end that you can use but in order to integrate running the tests with your build you will need to be familiar with the console program. The only tricky thing to remember is the Assembly naming convention that .NET uses. The format is `<namespace.classname>,<assembly filename>`

To try out the GUI

- Double click on the `NUnitGUI.exe` program in the `bin` folder.
- Click the **Browse** button and select the `NUnitTests.dll` file.
- Click the **Run** button.

You should see something like this:



Successful unit test of NUnit using the GUI Test Runner.

Hello World Test

Now that you have now successfully installed and configured the software we will write a very simple class and a test case to test it. The code that we will test will be trivial library that will return the string “Hello, World!” The test that we will write will ensure that only that string returns. The example may seem trivial but it does perform all the basic requirements for a test framework. There is code that needs to be tested, a test case and an automatic way of building and running the tests with a single command.

First we write the test case to test what we expect the code to do. At first this might seem a bit extreme to have to write 50+ lines of code to test Hello World but as you will see the code is very straightforward.

```
// HelloWorldTest.cs
```

Use the same namespace as the class we are testing with a .Test suffix. This isn't a requirement but it does help identify tests quickly.

```
namespace CSharpToday.UnitTesting.Tests {  
    using System;
```

Bring in the NUnit framework and the code we are testing.

```
    using NUnit.Framework;  
    using CSharpToday.UnitTesting;
```

Each test case must inherit from TestCase and have a name ending with Test in order to have NUnit find.

```
public class HelloWorldTest : TestCase {
```

Our entire test suite will consist of this single test case. Refer to the next example for a more complicated test suite.

```
    public static ITest Suite {  
        get { return new TestSuite(typeof(HelloWorldTest)); }  
    }
```

This is the required constructor for all test cases.

```
    public HelloWorldTest(String name) : base(name) {  
    }
```

When you have multiple tests using the same set of objects you can add them to what is called a **Fixture**. Before each test **SetUp** is called. After each test **TearDown** is called.

This may seem like a lot of work but it prevents any side effects and for all but the most complicated tests is fast enough not to notice. If time is starting to be a factor you can create the fixture in the constructor but beware of side effects caused by other test cases.

```
    HelloWorld _hello;  
  
    protected override void SetUp() {  
        _hello = new HelloWorld();  
    }
```

Now we proceed to test every public and protected member in our class. For most test cases you will just use the various **Assert** methods that the **TestCase** class provides. Refer to the **Assertion** class in the NUnit source code for a list of all the different **Assert** variations.

Each test must be named TestXXX, return void, and not take any arguments. I use the convention Test_MemberName_TestDetails.

```
    public void Test_Constructor() {  
        // make sure the object is constructed properly  
        AssertEquals("Hello, World!", _hello.Message);  
    }
```

When possible you should write your tests so that test data can be easily added without copying and pasting. In some cases you may want to store your test data in an external file and read it in the constructor rather than the **SetUp** method to avoid reading the same values for each test.

```
    public void Test_Message() {  
        // any non-null string is allowed in message  
        string[] testdata = new string[] {  
            String.Empty,  
            " ",  
            " Some Padding ",  
            "G'Day!",  
            "Hello, World!"  
        };  
  
        foreach (string value in testdata) {  
            _hello.Message = value;  
            AssertEquals(value, _hello.Message);  
        }  
    }
```

When testing exceptions execute the code that will cause the exception and explicitly catch the exception you are testing. If the exception is not thrown the Fail will cause an error. If a different exception is thrown NUnit will catch it and report it as an error.

```
public void Test_Message_ArgumentNullException() {
    try {
        _hello.Message = null;
        Fail("ArgumentNullException should have been thrown");
    } catch (ArgumentNullException) {
    }
}
}
```

Now we write the code to pass the test case.

```
// HelloWorld.cs
namespace CSharpToday.UnitTesting {

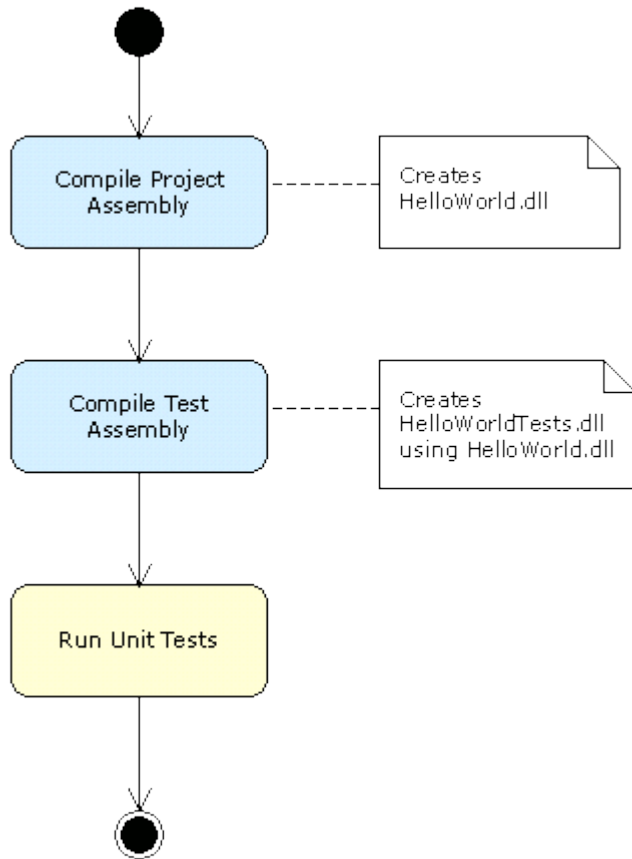
    using System;

    public class HelloWorld {

        string _message = "Hello, World!";

        public string Message {
            get { return _message; }
            set {
                if (value == null) {
                    throw new ArgumentNullException();
                }
                _message = value;
            }
        }
    }
}
```

In order to fully integrate NUnit into the development cycle you want to run your tests after each build. The following diagram shows the build process.



Here is the makefile to build the component, unit test, and run the test. You will have to change the NUNIT macro to point to the bin folder of where NUnit was installed.

```

# makefile for HelloWorld
CSCFLAGS=/nologo /debug
NUNIT=\dev\NUnit\bin

main: build test

clean:
    del HelloWorld.dll
    del HelloWorld.pdb
    del HelloWorld.Tests.dll
    del HelloWorld.Tests.pdb

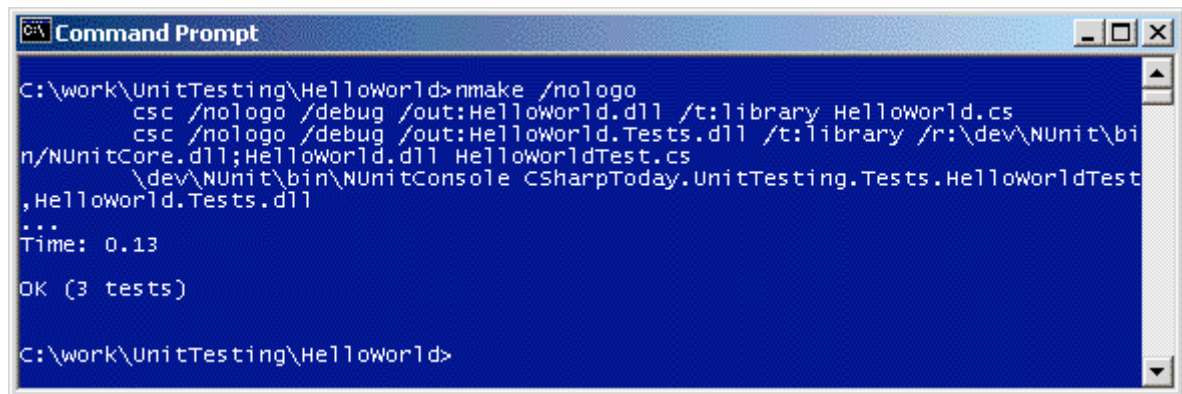
build: HelloWorld.dll HelloWorld.Tests.dll

test: build
    $(NUNIT) \NUnitConsole
    CSharpToday.UnitTesting.Tests.HelloWorldTest,HelloWorld.Tests.dll

HelloWorld.dll: HelloWorld.cs
    csc $(CSCFLAGS) /out:HelloWorld.dll /t:library HelloWorld.cs

HelloWorld.Tests.dll: HelloWorld.dll HelloWorldTest.cs
    csc $(CSCFLAGS) /out:HelloWorld.Tests.dll /t:library
    /r:$(NUNIT)/NUnitCore.dll;HelloWorld.dll HelloWorldTest.cs
  
```

The following files are available in the source code download. Using a command prompt you can build the project by typing nmake.



```
C:\work\UnitTesting\HelloWorld>nmake /nologo
    csc /nologo /debug /out:HelloWorld.dll /t:library HelloWorld.cs
    csc /nologo /debug /out:HelloWorld.Tests.dll /t:library /r:\dev\NUnit\bin\NUnitCore.dll;HelloWorld.dll HelloWorldTest.cs
    \dev\NUnit\bin\NUnitConsole CSharpToday.UnitTesting.Tests.HelloWorldTest
,HelloWorld.Tests.dll
...
Time: 0.13
OK (3 tests)

C:\work\UnitTesting\HelloWorld>
```

Successful build and run of HelloWorld.cs and HelloWorldTest.cs

The makefile will first build two libraries, HelloWorld.dll which contains our actual code and HelloWorld.Test.dll which contains the code to test HelloWorld.dll. The makefile then runs the NUnitConsole program on our test case. You can see in the output that 3 tests were run and everything was ok.

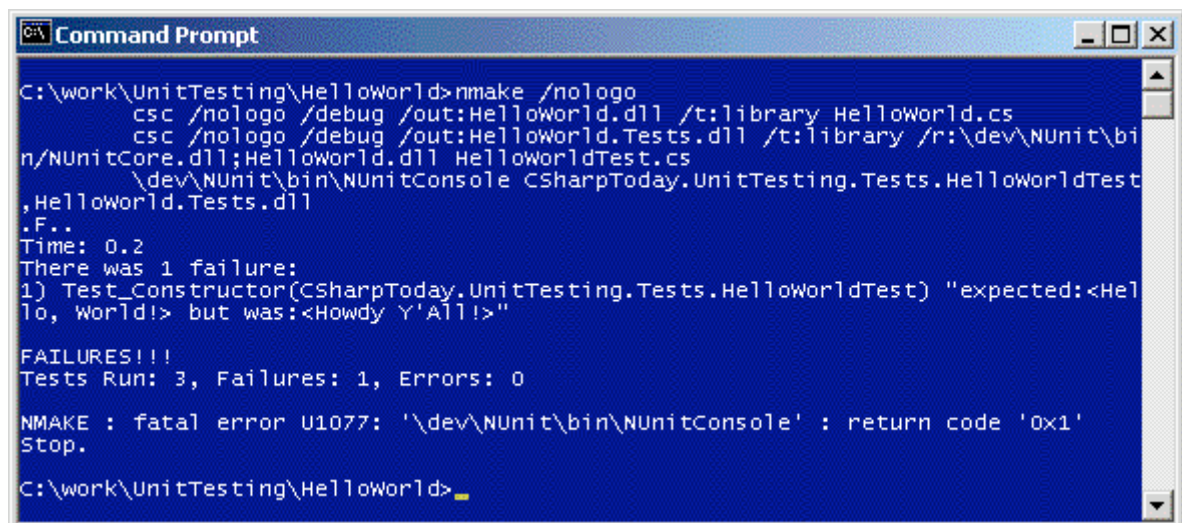
Now lets break the code to see what happens. Edit the HelloWorld.cs file and change the default message to “Howdy Y’All!”

```
namespace CSharpToday.UnitTesting {
    using System;

    public class HelloWorld {
        string _message = "Howdy Y'All!";

        public string Message {
            get { return _message; }
            set {
                if (value == null) {
                    throw new ArgumentNullException();
                }
                _message = value;
            }
        }
    }
}
```

Now when we build the project the code compiles but the test case now fails.



```
C:\work\UnitTesting\HelloWorld>nmake /nologo
    csc /nologo /debug /out:HelloWorld.dll /t:library HelloWorld.cs
    csc /nologo /debug /out:HelloWorld.Tests.dll /t:library /r:\dev\NUnit\bin\NUnitCore.dll;HelloWorld.dll HelloWorldTest.cs
    \dev\NUnit\bin\NUnitConsole CSharpToday.UnitTesting.Tests.HelloWorldTest
,HelloWorld.Tests.dll
.F..
Time: 0.2
There was 1 failure:
1) Test_Constructor(CSharpToday.UnitTesting.Tests.HelloWorldTest) "expected:<Hello, World!> but was:<Howdy Y'All!>"

FAILURES!!!
Tests Run: 3, Failures: 1, Errors: 0

NMAKE : fatal error U1077: '\dev\NUnit\bin\NUnitConsole' : return code '0x1'
Stop.

C:\work\UnitTesting\HelloWorld>
```

Successful build but failed test case

In most cases you will be able to go to the failed code and correct the problem as in this case. In some cases you will find that the test case needs changing. When this happens make sure the test case is wrong before changing it. It could very well be that the test case is showing a subtle bug.

A Real World Example

The Hello World example will give you a basic introduction to using NUnit but a more complicated example is needed to show you a variety of different testing patterns. The WordCounter example in the .NET SDK will be used as the code to test against. Included in the support materials for this article is a folder called **WordCount**. This folder contains the program source code with minimal changes to WordCounter, a complete unit test called WordCounter.Tests and a makefile that performs a one step build and test.

The code in WordCounter.Tests is commented and you are encouraged to look through it to see ways of testing a variety of classes. The most important file to look at is the **AllTests.cs** file as this explains how to create a test suite with more than one test. The other test cases show examples of testing methods that require files as input, checking for thrown exceptions, and a good pattern for making test data easy to add without resorting to massive copy and pasting.

You now have the foundation you need to get started using NUnit for unit testing. The rest of this article will provide you with some insight into unit testing and some help in writing more complicated tests.

Test First Programming

You might have noticed in the Hello World Test example that I wrote the test case before I wrote the code. This is called Test First Programming. Test first programming is a method of forcing you understand exactly what the code should and should not do before you write it. By writing tests first you are forced to have a very clear understanding of exactly what is needed by the code to pass the test.

The basic process is to:

- Start with a working code base that is passing 100% of all tests.
- Write the tests that will test the new functionality that you want to add. If you try to build now you will get compile and link errors. This is correct.
- Add the new functionality into the main code.
- Build the code and run the test. If all goes well your build will compile and all the tests will pass. When this happens, and it often does, you will get a feeling that something is not right. This is because you would normally start testing the code but since you have already done that you can start on the next change.

Test first programming works best when you wish to add functionality to an existing code base. If you are starting a project from scratch or building a prototype test first programming can really slow you down by writing a lot of test cases that get thrown away or drastically changed. If you do have a clear idea of what the design of the code will be then write the tests, otherwise wait until you know more. Once the design solidifies go back and add all the tests before you add too much functionality otherwise you will never catch up. This is real problem because writing a good unit test does take time.

Best Practices for Writing Tests

Here are some guidelines that I use that you might want to follow as you develop your unit tests. Of course any unit testing is better than none but if you are going to spend the time to write the tests you might as well write them in a manner that will make them easily extendable and as error free as possible.

Do use a separate test class for each class that you test. Name the class with the same name with a Test suffix. The test class should be in a different file so that you can compile a separate standalone library for all your test cases.

```
// Customer.cs
public class Customer {
    // implementation
}
```

```
// CustomerTest.cs
public class CustomerTest : TestCase {
    // implementation
}
```

Do use a separate method for each method you are testing when possible. In many cases you will need more than one test method for each method you are testing. In some cases this will not be practical as you can see in the RealWorld example. I use the convention Test_MemberName_TestDetail. For very simple test cases drop TestDetail and use the Test_MemberName format. For example when testing an Email property of a Customer class I might have the following methods in the test case.

```
public class CustomerTest : TestCase {
    public void Test_Email_ValidData {
        // test case implementation
    }

    public void Test_Email_InvalidData {
        // test case implementation
    }

    public void Test_Email_ArgumentNullException {
        // test case implementation
    }
}
```

Do test with valid data and invalid data.

Do test all assertions. Refer to the `HelloWorldTest` class for an example of how to do this.

Do concentrate your first tests on the functionality that can be easily tested. Properties and methods that perform calculations are ideal. Simple get, set properties are not normally tested unless they perform error checking. This helps get you a substantial test framework up and running with a minimal time investment. Just keep in mind that the simple things are not likely to be the cause of the problems.

Do not test screen output. The amount of work that is required to test this is usually not worth it. A simple off by one pixel can cause all your test cases to fail. Consider using a test that simply calls the function to ensure that nothing crashes.

Do get positive feedback into your test cycle. Adopt a practice that before a bug fix or new feature can be added you must first add a test case to test the new code. The test case should pass once the bug fix or feature is added.

Do use arrays of data when testing a wide range of input values. In the following example a private Input struct is created to hold the input argument and the expected value. A loop is used to iterate over each input. In the future if you find a certain input cause an error you can now quickly add it to the test case with a single line.

```
public class BooleanOptionTest : TestCase {
    private struct Input {

        public string argument;
        public bool expectedValue;

        public Input(string argument, bool expectedValue) {
            this.argument = argument;
            this.expectedValue = expectedValue;
        }
    }

    public void Test_Parse_ValidData() {
        ArrayList inputs = new ArrayList();
        inputs.Add(new Input("/option", true));
        inputs.Add(new Input("/option", true));
    }
}
```

```

        inputs.Add(new Input("/option+", true));
        inputs.Add(new Input("/option-", false));
        inputs.Add(new Input(" /option ", true));
        inputs.Add(new Input(" /option+ ", true));
        inputs.Add(new Input("\t/option- \t ", false));

        foreach (Input input in inputs) {
            BooleanOption option = new BooleanOption("option", false);
            option.Parse(input.argument);
            AssertEquals(input.expectedValue, option.ToBoolean());
        }
    }
}

```

Conclusion

Unit testing is a powerful tool in developing solid code that will distinguish you among professional developers. By writing unit tests you can feel safe when making wide scale changes to the internal implementation because everything will be tested. You will be able to quickly ship bug fixes without having to worry if your fix breaks some other part of the project. Using the free open source test framework NUnit your investment in time learning will be well made. NUnit is based on JUnit, the unit testing de facto standard in the Java world and is available for virtually all programming languages.

Resources

Link to code that supports the article.

NUnit: <http://nunit.sourceforge.net/>

Contains the latest version of NUnit. In the Source Forge section of the site you can access any previous release, a discussion board and the latest news.

JUnit: <http://www.junit.org/>

The Java version of the software that NUnit was ported from. JUnit is heavily used in the Java world and this site contains some of the best essays on using unit testing. Due to the similar nature of JUnit and NUnit and Java to .NET much of the content is very relevant to .NET programmers.

Extreme Programming: A gentle introduction: <http://www.extremeprogramming.org/>

A good introduction to XP (eXtreme Programming) that explains how unit testing fits into the whole XP development process.

XPractices: <http://www.xprogramming.com/Practices/xpractices.htm>

A list of practices that were followed in an early project using XP. There are some really good ideas to consider in this section. Part of a larger XP Portal site.

McConnell, Steve. *Code Complete*. Microsoft Press 1993.

Chapter 25 is specifically about unit testing but the entire book is highly recommended.