# xUnit.net

## Unit Testing for C# and ASP.Net Applications

Michael Setteducati

Dr. Raunak

CS483 - Software Testing

11 December 2017

xUnit.net is a Unit Testing Framework for testing C# code and ASP.Net applications. It also supports F#, VB.Net and other ASP.Net languages. xUnit is free and open source, and works with the most popular .Net IDEs including Visual Studio, Resharper, and Ryder. Additionally, it has a command line interface for developers who prefer to run their unit tests from the command line. It also supports testing Xamarin applications, which are cross-platform (iOS, Android, Windows) applications built using the ASP.Net framework.[1] Overall, xUnit seems to be one of the most complete unit testing platforms for the .Net framework based on its plethora of features and ease of use, and is generally accepted by the C# community as adequate to test code that will eventually run in production environments.

Regarding installation and use, xUnit was quite easy to download and use. I used Visual Studio Community Edition to write my code and tests, which is the most common IDE used with C# and ASP.Net. Most C# solutions, especially those built in Visual Studio, use Nuget Package Manager to ensure that dependencies are met, especially in large, complex solutions. Note that a C# "solution" is basically the equivalent of a project file in other programming languages. In order to use xUnit with the solution you are working with, one must add the xunit package for the core xUnit functionality. Additionally, I used the Visual Studio runner to run my tests inside Visual Studio, so I also had to add the xunit.runner.visualstudio package. Once these dependencies were installed, I simply had to import xunit to my test classes and was then able to use the various attributes and functionality that xUnit provides. When you want to run these tests, one simply opens the Unit Test pad, and selects Run All. The Test Results pad then opens

[1] https://xunit.github.io/#documentation

up which gives the output of these tests. Overall, installation and running of xUnit was a very user friendly process, and it seamlessly integrated with Visual Studio Community Edition I have installed on my mac. (See Appendix A for screenshots of installing xunit and running xunit tests.)

Regarding documentation, xUnit's open source documentation is complete and easy to follow. It gives examples of how to set up xUnit, and how to run it via the console or in Visual Studio. It also gives examples of Facts and Theories, which will be covered next. There is also a good amount of literature available online regarding xUnit, as it has become increasingly popular in the C# community. Many developers are starting to migrate their unit tests xUnit over the older nUnit because of its focus on extensibility and flexibility that writing theories allow for.[2] xUnit allows for developers to create custom classes that return a test set of data to run its unit tests against, which allows for less redundant code. Tests can be written once and then passed a variety of inputs to run the test functions against, as opposed to re-writing the same code with different input in multiple assertions. This is where the true power of xUnit tests come into play, and is discussed in further detail below.

xUnit primarily uses two distinct attributes when writing test functions: [Fact] and [Theory]. Facts indicate that whatever is in this function is undoubtedly true, and is quite similar to the @Test markup in jUnit. Facts are just basic test methods as we are familiar with in other programming languages, especially jUnit.

[2] https://raygun.com/blog/unit-testing-frameworks-c/

Theories are similar to facts, but with a subtle difference: they are only necessarily true for certain inputs of data. This data is passed through following attributes such as [InlineData], [MemberData], or [ClassData]. InlineData is used for primitive types such as integers or strings, ClassData is used for passing a class type with certain objects instantiated, and MemberData is used for passing a function with certain objects instantiated. InlineData is the simplest to use, for example you would create a test function with a certain number of parameters, and then pass these parameters through various [InlineData] attributes; the test function is then run for each [InlineData] attribute. All in all, Facts and Theories are the primary ways of writing unit tests using xUnit. (See Appendix B for an example of the same test function written using Fact and Theory).

Additionally, xUnit has built in “setup” and “teardown” functionality for running its unit tests. This is done by making the test class inherit from the IDisposable interface. As criteria for this interface, it then must implement a function called “public void Dispose(),” which acts as the “teardown” function. The class should also implement a default constructor which acts as the “setup” function. This implementation allows for the test class to easily implement a DatabaseFixture, which is imperative in testing ASP.Net applications because ASP.Net applications generally need a database for the backend to interface with. This allows developers to create a small, simple database only used for testing, where all of the data is known. The test functions can then use this instantiated DatabaseFixture to run its test code against, which is better practice than using a Production or Development environment database to test against.

Overall, this setup and teardown functionality is very similar to the unit testing functionality we are accustomed to in jUnit, and generally has no major differences.[3]

Next, I needed to procure some C# code to test with xUnit. Unfortunately, I had no C# projects that I was able to use because all of the C# programming I have done is property of the company I work for, so in order to run xUnit on some C# code, I created a new C# solution with two basic C# files. The first file is called RecursiveMath.cs, which implements five functions that recursively add, subtract, multiply, divide, and exponentiate. The second file is called SortingAlgorithms.cs, which implements Selection Sort, Insertion Sort, and Merge Sort. I then wrote Unit Tests for these algorithms using xUnit in TestRecursiveMath.cs and TestSortingAlgorithms.cs. I picked these types of functions to test because they are relatively trivial functions to write, and the resulting unit tests are also straightforward. I wanted to focus on the functionality of xUnit as a tool itself, rather than writing complicated unit tests for a more complicated program, and losing sight of some of its functionality in these potentially complex unit tests.

In writing the test cases for these two classes, I simultaneously evaluated the user experience that xUnit provides. I was looking for it to be intuitive to use, as well as familiar to use. Since this is not the first unit testing framework I have worked with, I had a certain expectation regarding unit testing frameworks going into the evaluation of this tool. Overall, I would say that xUnit was completely satisfactory regarding my unit testing expectations, as it did everything I needed it to

[3] http://xunit.github.io/docs/comparisons.html

do, and even more. Using Theories to concisely write Unit Tests is a very neat feature, and I could see how this could be even more helpful in large applications where unit tests become quite repetitive.

Additionally, I found xUnit extremely easy to learn, and I believe anyone with unit testing experience would agree with this. The documentation is complete, and also provides a comparison of equivalent statements in other C# unit testing frameworks for those familiar with another framework. After quickly reading through this documentation, I knew exactly how I would be able to write unit tests in xUnit given my experience with jUnit. Finally, the tests were very easy to run in Visual Studio, which I also consider a major positive as I much prefer a GUI Unit Testing experience over a Command Line interface.

Next, xUnit employs a white-box testing approach, since it is only used for writing unit tests. Unit Tests are generally written using a white-box approach, so developers can see what the code is doing and effectively test the code to ensure that it is doing just that. The fundamental approach that xUnit employs is running unit tests written for objects, or even static functions. xUnit is ideal for testing any kind of software, as long as a test class is written for each class in the software. This ensures that the class operates as it is intended. Of course, integration tests should also be written for complex software systems to ensure that the classes work together as they are intended, however, this is beyond the scope of xUnit's functionality.

The other two most popular unit testing frameworks for C# and ASP.Net applications are MSTest (Visual Studio's built in unit testing framework) and nUnit. First, MSTest is a basic unit testing framework which allows for writing multiple Test Methods, as well as "setup" and "cleanup" methods. It seems very similar to jUnit, as it is able to basically just do the bare minimum in terms of unit testing.

Next, nUnit is capable of doing everything MSTest supports, however nUnit also allows for running test functions with certain parameters by passing these parameters through a TestCase attribute, similar to the way that parameters are passed using InlineData attributes in xUnit. This is convenient for writing more concise test cases, however, nUnit only supports passing primitive data types in this way. On the contrary, xUnit also supports dynamically building objects and passing them using ClassData or MemberData attributes, as previously discussed.

Overall, in comparing these three unit testing tools, all three of these frameworks are complete, community accepted frameworks that are capable of writing robust unit tests, and many developers simply used what they are accustomed to. xUnit is the newest of the three frameworks, and it supports dynamic testing and allows for the greatest degree of flexibility, which is why it has gained a great amount of traction in recent years. nUnit is the most "proven" framework, as it has been around for a good amount of time and has a lot of literature written on it. MSTest is arguably the most convenient framework to use because it is installed in Visual Studio out of the box, however it does not have as full of a featureset as the other two

frameworks have. All-in-all, any of the three choices would allow for unit tests to check the "correctness" of a program or software system.[4]

After analyzing xUnit in its entirety, considering the installation process, writing unit tests, running these tests and analyzing the output, I can confidently say that xUnit has left an overwhelmingly positive impression on me. I found it intuitive and effective to use in writing simple unit tests, and quite extensible in writing more complicated unit tests that may use a database or predefined objects. I can confidently recommend xUnit in both personal and commercial ASP.Net applications, and it seems as though the C# community would agree with this statement. Furthermore, after exploring this test tool I will be recommending to my supervisor at my internship that we migrate our unit tests to xUnit, rather than the built in MSTest framework that we had been using.

In terms of usage scenarios, xUnit is fully capable of testing any sort of program written in C#. This might be a simple program, like the one I ran my tests on, or a more complicated web or mobile (Xamarin) application; as long as the code is well organized into distinct classes, xUnit will be able to test it effectively. Since it is a unit testing framework, unit tests are most effective when a class performs object specific actions, meaning it is most effective in object-oriented design instances.

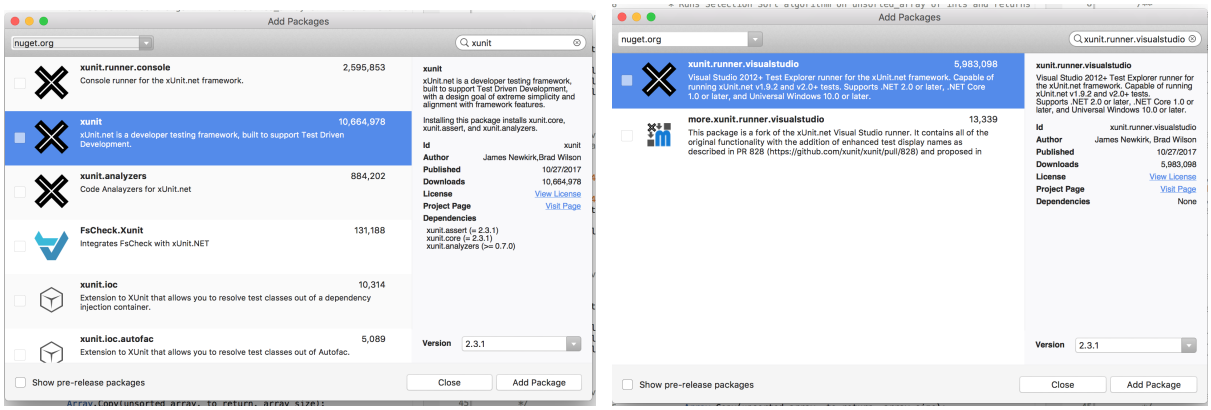
[4] https://raygun.com/blog/unit-testing-frameworks-c/

The only instance I can see potential trouble using xUnit is in ASP.Net Web Forms applications, where a front-end .aspx file has a corresponding .aspx.cs file. These files typically take care of operations such as OnPageLoad() and OnClick() (for ASP controls), which means that they are not really distinct objects; they are ultimately just the "code behind" a certain front-end webpage. This might be more difficult to test because it might not be possible to test for certain inputs and outputs, because these functions are typically void functions that just perform an action on the page whenever ASP invokes this function. However, well designed applications should not really need extensive testing on these aspx.cs files because the business logic and class logic should be defined in exterior classes, and these are the classes that require unit testing, which xUnit would be able to test effectively. As long as these exterior classes work as intended as verified by unit tests, and the web page functions as intended (for example OnPageLoad() does what is intended by manually checking), there is no real reason to unit test these "code behind" classes.

In the end, xUnit is a well designed and thoroughly documented unit testing framework for the ASP.Net platform. xUnit is simple to install as a dependency and run its tests right in Visual Studio, as well as alternate ASP.Net IDEs such as Resharper or Ryder. xUnit allows for basic, expected unit testing functionality via the [Fact] attribute, and anyone familiar with general unit testing would be able to write xUnit tests using only this attribute. Additionally, it allows for flexible, dynamic tests using the [Theory] attribute, which is what sets xUnit apart from its ASP.Net unit testing counterparts such as MSTest and nUnit. The ASP community has taken an increased likening to xUnit because of its modern take on unit tests, e.g. supporting dynamic unit tests, as opposed to the more "old fashioned" approach consisting of numerous redundant

assertions. xUnit is capable of test simple C# classes, as well as complex ASP.Net applications that will eventually be deployed into a Production environment. Overall, after writing unit tests in xUnit, as well as doing research on the other ASP.Net unit testing frameworks, I can confidently say that xUnit is my unit testing tool of choice for C# and ASP.Net applications.

## Appendix A - Installation and Running Tests




*Figures 1 & 2: Installing xunit and xunit.runner.visualstudio using nuget package manager. Simply search nuget for xunit and xunit.runner.visualstudio and select "Add Package."*

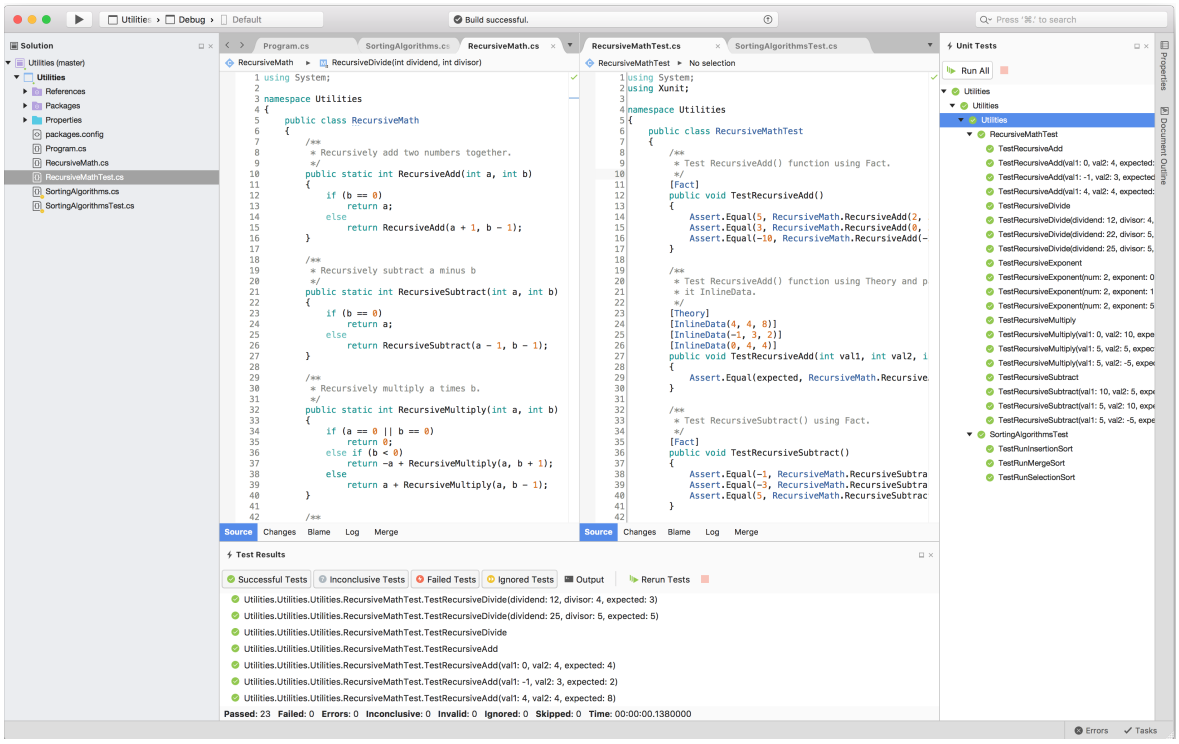



*Figure 3: Running Unit Tests and Viewing Test Results. Simply select "Run All" in the Unit Tester pad, and view the results in the Test Results pad.*

**Appendix B - Fact and Theory Examples**

```
8          /**
9           * Test RecursiveAdd() function using Fact.
10          */
11         [Fact]
12         public void TestRecursiveAdd()
13         {
14             Assert.Equal(5, RecursiveMath.RecursiveAdd(2, 3));
15             Assert.Equal(3, RecursiveMath.RecursiveAdd(0, 3));
16             Assert.Equal(-10, RecursiveMath.RecursiveAdd(-20, 10));
17         }
```

*Figure 1: Example of using Fact to write a test function.*

```
19         /**
20          * Test RecursiveAdd() function using Theory and passing
21          * it InlineData.
22          */
23         [Theory]
24         [InlineData(4, 4, 8)]
25         [InlineData(-1, 3, 2)]
26         [InlineData(0, 4, 4)]
27         public void TestRecursiveAdd(int val1, int val2, int expected)
28         {
29             Assert.Equal(expected, RecursiveMath.RecursiveAdd(val1, val2));
30         }
```

*Figure 2: Example of using Theory to write the same test function as in Figure 1, but more concisely.*