Unit Test Cheat Sheet

Here’s a list of elements for unit testing in Visual Studio using MSTest.

Documentation for Visual Studio Testing is here: GitHub - microsoft/vstest-docs: Documentation for the Visual Studio Test Platform.

Necessary Packages

The following packages are necessary for unit testing using Visual Studio’s built in MSTest framework:

- MSTest.TestFramework
- MSTest.TestAdapter
- Microsoft.NET.Test.Sdk
- coverlet.collector

MSTest.TestFramework - is the test library itself. This includes all the classes, attributes, of the testing framework.

MSTest.TestAdapter - is the library that Visual Studio uses to discover and execute tests in your code. Every test suite you use (MSTest, NUnit, etc…) needs a testadapter so Visual Studio can access and execute tests.

Microsoft.NET.Test.Sdk - is the specific sdk for the framework under test.

coverlet.collector - is the test coverage library that is loaded in a generic test project. This can be replaced with another if needed.

Reference Project Test Structure

The library, OGA.Testing.Lib includes a project called, TestTemplate.
This project contains two template classes that provide a common template for all testing projects.

See this article:

https://oga.atlassian.net/wiki/spaces/~311198967/pages/191365133/C+Unit+Test+Template+Classes

See this article if you are testing NET Framework versions: .NET Framework Unit Testing Issues

Test Context Access

As the testing framework runs your tests, it tracks progress and results in a TestContext instance. This object is passed into a test class in a few places, so that your testing logic has access to it.

The TestContext is passed to a test class initializer method (decorated with `[ClassInitialize]`), so the class initializer has any test info needed for setup.

The TestContext is passed to a test class cleanup method (decorated with `[ClassCleanup]`), so the class cleanup logic has any test info needed during cleanup.

However. The TestContext given to the test class initializer is not updated during testing. So, it cannot be directly cached by the test class.

Instead. The MSTest framework will look for the following property signature in each test class, and push the current TestContext into it before each test:

```
[TestClass]
public class YourUnitTests
{
  public TestContext TestContext { get; set; }

  [ClassInitialize]
  public static void TestClassSetup(TestContext context)
  {
    // Copy the given test context instance into our local property...
    // This ensures that all logic in our test class, including class setup, has a consistent reference for test context data.
    TestContext = context;
  }
}
```

> NOTE: The above unit test class contains a property called, TestContext.
> This property is set by reflection, by the testing framework before each test, to ensure the test setup method and test logic has access to any context information it needs.
> If you have tests that need access to testing context data, include the property, shown above, in your test class.

As well. The class setup method, above, also sets the class's local testcontext property with what they are given, to ensure that all testing logic has a consistent reference to TestContext data.

See the bottom of this article for details: https://github.com/microsoft/testfx/issues/255

Generic Test Structure

Below is a comprehensive test class structure, including optional method calls for common setup and teardown logic.

> NOTE: This is for illustration of the available test life-cycle methods. It is NOT how we are currently performing tests with MSTEST.

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

namespace MSTestUnitTests
{
  // A class that contains MSTest unit tests. (Required)
  [TestClass]
  public class YourUnitTests
  {
    /// <summary>
    /// This property is automatically populated by the MSTest framework, each time a test starts.
    /// This is how each test is able to access test context properties, such as the test name, during execution.
    /// Each test can access the current testing context, through this property.
    /// See the bottom of this GH issue article for the explanation: https://github.com/microsoft/testfx/issues/255
    /// Also. See this Confluence article for how to setup a unit test:
    ///     https://oga.atlassian.net/wiki/spaces/~311198967/pages/edit-v2/41418763
    /// </summary>
    public TestContext TestContext { get; set; }

    /// <summary>
    /// This method is called before any test class initializers.
    /// It is meant to setup any assembly-wide config, logging, etc.
    /// </summary>
    [AssemblyInitialize]
    public static void AssemblyInit(TestContext context)
    {
```

```
  // Executes once before the test run. (Optional)
}

/// <summary>
/// This method is called after all testing and test class cleanup methods have completed.
/// It is meant to teardown any assembly-wide config, logging, etc.
/// </summary>
[AssemblyCleanup]
public static void AssemblyCleanup()
{
  // Executes once after the test run. (Optional)
}

[ClassInitialize]
public static void TestFixtureSetup(TestContext context)
{
  // Executes once for the test class. (Optional)
}

/// Called before the first test is run.
[ClassCleanup]
public static void TestFixtureTearDown()
{
  // Runs once after all tests in this class are executed. (Optional)
  // Not guaranteed that it executes instantly after all tests from the class.
}

// Called before each test.
[TestInitialize]
public void Setup()
{
  // Runs before each test. (Optional)
}

/// Called after each test.
[TestCleanup]
public void TearDown()
{
  // Runs after each test. (Optional)
```

```
    }

    // Mark that this is a unit test method. (Required)
    [TestMethod]
    public void YouTestMethod()
    {
      // Your test code goes here.
    }

    [Ignore]
    [TestMethod]
    public void YouTestMethod()
    {
      // Your test code goes here.
    }
  }
}
```

Necessary Elements

Here are the necessary elements for a test:

- Tests must be separated into test methods that verify or challenge one aspect of functionality.
- Each test method needs to be a public instance signature with no arguments, like: `public void TestSomething() {...}`
- The test method must be decorated with the `TestMethod` attribute
- Test methods must be contained in a class, decorated with the `TestClass` attribute

Optional Elements

To reduce the extra fluff and ceremony bulk in test methods, several optional calls can be created to perform setup and teardown at the test level, class level, and assembly/project scope.

Below is a list of these setup and teardown method calls allowed by the framework.

Test Setup and Teardown

These methods bracket each test method, and are called before and after each test in a test class.

These are public void methods with no arguments, and decorated as follows:

```
[TestInitialize]
public void Setup()
{
  // Runs before each test. (Optional)
}

[TestCleanup]
public void TearDown()
{
  // Runs after each test. (Optional)
}
```

Class Setup and Teardown

These methods bracket all test methods in a single test class, and are called before any test is started in a test class, and after all tests in a test class are complete.

These are static public void methods, with the following signatures and attributes:

```
[ClassInitialize]
static public void TestFixtureSetup(TestContext context)
{
  // Executes once for the test class. (Optional)
}

[ClassCleanup]
static public void TestFixtureTearDown()
{
  // Runs once after all tests in this class are executed. (Optional)
  // Not guaranteed that it executes instantly after all tests from the class.
}
```

Assembly Setup and Teardown

If you have any project-wide setup and teardown logic that must run before all test methods and class test setup, or must run after all test methods have executed and all class teardown is complete, create an independent class, like the example below, and put this assembly-wide setup and teardown logic in it:

> NOTE: The assembly-wide test setup and teardown methods must be: in a class marked `TestClass`, and have the same signature as the below methods. Missing any of these

> requirements, will cause the Test framework to not execute your assembly-wide setup/teardown methods.

> NOTE: Also. Reflection is used to locate the assembly-wide setup/teardown methods, which has not always been good at drilling through base classes. So, be sure to explicitly include the methods in your project, and do not rely on inheritance. Use the explicit methods in your project to, in turn, call any assembly-wide setup that you leverage in base classes they inherit from.

```
[AssemblyInitialize]
public static void AssemblyInit(TestContext context)
{
  // Executes once before the test run. (Optional)
}

[AssemblyCleanup]
public static void AssemblyCleanup()
{
  // Executes once after the test run. (Optional)
}
```

Disabled Tests

Marking a Test Method with the `[Ignore]` attribute, will track it in the test catalog, but it will not execute.

```
[Ignore]
[TestMethod]
public void YouTestMethod()
{
  // Your test code goes here.
}
```

Execution Flow

This is the order of setup/teardown and test method execution:

```
AssemblyInitialize - executes before all tests in a project.
    ClassInitialize - executes before all tests in a class.
        TestInitialize - executes before each test
```

```
        TestMethod_One
    TestCleanup - executes after each test
    ... other tests...
ClassCleanup - executes after all tests in a class have completed.
... other test classes...
AssemblyCleanup - executes after all tests have completed.
```

Test Assertions

In the Microsoft test framework, these are the assertions that are possible:

```
Assert.Fail("Some failure result message."); // Used as a statement of failure wherever it occurs.
Assert.AreEqual(28, _actualFuel); // Tests whether the specified values are equal.
Assert.AreNotEqual(28, _actualFuel); // Tests whether the specified values are unequal. Same as AreEqual for numeric values.
Assert.AreSame(_expectedRocket, _actualRocket); // Tests whether the specified objects both refer to the same object
Assert.AreNotSame(_expectedRocket, _actualRocket); // Tests whether the specified objects refer to different objects
Assert.IsTrue(_isThereEnoughFuel); // Tests whether the specified condition is true
Assert.IsFalse(_isThereEnoughFuel); // Tests whether the specified condition is false
Assert.IsNull(_actualRocket); // Tests whether the specified object is null
Assert.IsNotNull(_actualRocket); // Tests whether the specified object is non-null
Assert.IsInstanceOfType(_actualRocket, typeof(Falcon9Rocket)); // Tests whether the specified object is an instance of the expected type
Assert.IsNotInstanceOfType(_actualRocket, typeof(Falcon9Rocket)); // Tests whether the specified object is not an instance of type
StringAssert.Contains(_expectedBellatrixTitle, "Bellatrix"); // Tests whether the specified string contains the specified substring
StringAssert.StartsWith(_expectedBellatrixTitle, "Bellatrix"); // Tests whether the specified string begins with the specified substring
StringAssert.Matches("(281)388-0388", @"(?d{3})?-? *d{3}-? *-?d{4}"); // Tests whether the specified string matches a regular expression
StringAssert.DoesNotMatch("281)388-0388", @"(?d{3})?-? *d{3}-? *-?d{4}"); // Tests whether the specified string does not match a regular expression
CollectionAssert.AreEqual(_expectedRockets, _actualRockets); // Tests whether the specified collections have the same elements in the same order and quantity.
CollectionAssert.AreNotEqual(_expectedRockets, _actualRockets); // Tests whether the specified collections does not have the same elements or the elements are in a different order and quantity.
```

```
CollectionAssert.AreEquivalent(_expectedRockets, _actualRockets); // Tests whether two collections contain the same elements.
CollectionAssert.AreNotEquivalent(_expectedRockets, _actualRockets); // Tests whether two collections contain different elements.
CollectionAssert.AllItemsAreInstancesOfType(_expectedRockets, _actualRockets); // Tests whether all elements in the specified collection are instances of the expected type
CollectionAssert.AllItemsAreNotNull(_expectedRockets); // Tests whether all items in the specified collection are non-null
CollectionAssert.AllItemsAreUnique(_expectedRockets); // Tests whether all items in the specified collection are unique
CollectionAssert.Contains(_actualRockets, falcon9); // Tests whether the specified collection contains the specified element
CollectionAssert.DoesNotContain(_actualRockets, falcon9); // Tests whether the specified collection does not contain the specified element
CollectionAssert.IsSubsetOf(_expectedRockets, _actualRockets); // Tests whether one collection is a subset of another collection
CollectionAssert.IsNotSubsetOf(_expectedRockets, _actualRockets); // Tests whether one collection is not a subset of another collection
Assert.ThrowsException<ArgumentNullException>(() => new Regex(null)); // Tests whether the code specified by delegate throws exact given exception of type T
```

Test Parallelization

Normally, tests are executed sequentially, one class at a time, and one method at at time. However, you can allow for multiple tests to be performed in parallel, by defining it at the assembly level, with the following attribute:

```
[assembly: Parallelize(Workers = 0, Scope = ExecutionScope.MethodLevel)]
```

Setting the ExecutionScope to MethodLevel allows the test runner to execute all tests in parallel.

Setting the ExecutionScope to ClassLevel allows the test runner to execute test classes in parallel, but test methods within a class are serialized.

Omitting this optional attribute, defaults to no all sequential test execution.

Revision #2
Created 11 May 2025 06:51:44 by glwhite
Updated 11 May 2025 07:01:42 by glwhite