Nunit V2.0 - 11 July, 2002 (Beta 1)

This is the second major release of the xUnit based unit testing tool for Microsoft .NET. It is written entirely in C# and has been completely redesigned to take advantage of many .NET language features, for example custom attributes and other reflection related capabilities.

Installation

By default the installation program places the all files in the directory:

```
c:\Program Files\Nunit V2.0
```

In the installation directory there are four sub-directories; bin, doc, samples, and src.

Note: The current version of the installation may require that the project be placed in the default directory. This is something that is being worked on and should be corrected in the future.

Start Menu

The installation program places a number of items in the Start menu. There is a shortcut to the Forms interface executable. (There is also a shortcut placed directly on the desktop). In addition to the executable file the menu items under Samples bring up a Visual Studio.NET solution file for the particular sample. The source shortcut brings up the Visual Studio.NET solution file for the entire project.

Installation Verification

The way that we verify that the installation has worked successfully is to install the program and run the tests that were used to build the program. These tests can be found in the assembly, `nunit.tests.dll`. There should be 131 tests and they should all pass. In addition, you can also run the test contained the `nunit-console.exe` executable. This is an acceptance test to verify that the XML result that is produced by the console program passes the XML schema validation.

Changes in V2.0

- Attribute based mechanism for identifying test fixtures and test methods. In previous versions of NUnit and JUnit for that matter, inheritance was used as the mechanism for identifying which classes were test fixtures and a naming convention was used to identify test methods in these classes. The Custom Attributes available in .NET provide a much more straightforward and less ambiguous mechanism for identification. See the attributes section below for details.
- Automatic creation of test suites. In previous versions of NUnit it was required to manually construct suites of tests using a Suite property. This has been replaced with dynamic creation of suites based on namespaces. Given an assembly the program will search through the assembly looking for test fixtures. Once it finds one it creates a suite based for each namespace and the specific test fixture. For example, `Nunit.Tests.SuccessTests` will build a containing Suite called “Nunit” which will contain another suite called “Tests” which in turn will create another suite called “SuccessTests” which finally will hold the individual test methods. This corrects an error prone task where things may or may not have been included in a suite.
- Additional samples, Managed C++ and Visual J#. The existing samples (C#, VB.NET, Money) have been upgraded to the new version. Money-port has been included to demonstrate the minimal amount of effort required to upgrade.
- A class `TestFixture` in `Nunit.Framework.dll` has been provided for backwards compatibility. See the “Money-port” project as a sample of what is required to upgrade to the new version. The absolute minimum that is required is to change the namespace from `NUnit.Frameowrk` to `Nunit.Framework` and change the inheritance from `TestCase` to `TestFixture`.
- New namespace: `Nunit` instead of `NUnit`. This conforms more with the standard way of expressing namespaces in .NET.
- The exception that is thrown when an Assertion fails is now called `AssertionException`. In previous versions this was called `AssertionFailedError`.

Forms Interface

- Tree based display in forms interface with test status indication and the ability to run individual tests or suites from the tree.
- Dynamic reloading of an assembly using AppDomains and shadow copying. This also applies if you add or change tests. The assembly will be reloaded and the display will be updated automatically.
- Window sizes are flexible due to the introduction of splitters on major sub-areas of the screen.
- File->Recent Assemblies menu item. The program keeps track of the 5 most recently used assemblies. If no command line arguments are used to start the forms executable the most recent assembly is loaded by default.
- The “Run” button is the default button on the form, which allows hitting return to start the tests.

Console Interface

- The console interface command line parameters have been modified to be more explicit and conform to similar programs available in .NET. See the Command line parameters section in this document for details.
- XML Output. The console program produces and XML output suitable for inclusion into other existing systems.
- Customizing of console output is provided through the use of XSLT. See Summary.xslt for the translation that is provided for the release. Using your own transform file is possible via a command line argument.
- Backwards compatibility with running suites. The Forms interface currently has no mechanism to run suites defined with the suite property. Using the fixture command line argument you must specify a class that is either a TestFixture or contains a Suite attribute.

Attributes

TestFixture

This is the attribute that marks a class that contains test methods. This attribute is contained in the `Nunit.Framework` namespace. In previous versions of NUnit the programmer was required to inherit from a class called `TestCase`. This is more flexible because it allows a TestFixture attribute to be put on any class.

Note: There are a few restrictions. The class must have a default constructor. The class must also be a publicly exported type or the program that dynamically builds suites will not see it.

TestFixture Example

```
namespace Nunit.Tests
{
   using System;
   using Nunit.Framework;

   [TestFixture]
   public class SuccessTest
   {
      // ...
   }
}
```

Test

The Test attribute marks a specific method inside a class that has already been marked as a TestFixture, as a test method. For backwards compatibility with previous versions of Nunit a test method will also be found if the first 4 letters are “test” regardless of case.

Note: The signature for a test method is defined as follows; `public void MethodName()` There must be no parameters. If the programmer marks a test method that does not have the correct signature it will not be run and it will appear in the Test Not Run area in the UI that ran the program.

Test Example

```
namespace Nunit.Tests
{
   using System;
   using Nunit.Framework;

   [TestFixture]
   public class SuccessTest
   {
      [Test]
      public void Success()
      {}

      public void TestSuccess()
      {
         // will also be run -- backwards compatibility
      }
   }
```

SetUp and TearDown

These two attributes are used inside a TestFixture to provide a common set of functions that are performed prior (SetUp) and after (TearDown) a test method is called. A TestFixture can have only one SetUp method and only one TearDown method. If more than one of each type is defined the TestFixture will not be run. It will compile however.

SetUp/TearDown Example

This example is from SetupTest.cs in Nunit.Tests.dll. It verifies that the SetUp method (Init) is called and the TearDown method (Destroy) is called during the execution of the test fixture.

```
[TestFixture]
public class SetUpTest
{
   internal class SetUpAndTearDownFixture
   {
      internal bool wasSetUpCalled;
      internal bool wasTearDownCalled;

      [SetUp]
      public void Init()
      {
         wasSetUpCalled = true;
      }
```



```
        [TearDown]
        public void Destroy()
        {
            wasTearDownCalled = true;
        }

        [Test]
        public void Success(){}
    }

    [Test]
    public void MakeSureSetUpAndTearDownAreCalled()
    {
        SetUpAndTearDownFixture testFixture = new
                SetUpAndTearDownFixture();

        TestSuite suite = new TestSuite("SetUpAndTearDownSuite");
        suite.Add(testFixture);
        suite.Run(NullListener.NULL);
        Assertion.Assert(testFixture.wasSetUpCalled);
        Assertion.Assert(testFixture.wasTearDownCalled);
    }

}
```

SetUp/TearDown and Inheritance

These two attributes are inherited from base classes. Therefore, if a base class has defined a SetUp method that method will be called prior to execution of test methods in the derived class. If you wish to add more SetUp/TearDown functionality in a derived class you need to mark the method with the appropriate attribute and then call the base classes method.

SetUp/TearDown Inheritance Example

The following code is part of the SetupTest.cs file that was used in the previous example. It simply demonstrates that SetUp/TearDown are called when inherited.

```
internal class InheritSetUpAndTearDown : SetUpAndTearDownFixture
{
    [Test]
    public void AnotherTest(){}
}

[Test]
public void CheckInheritedSetUpAndTearDownAreCalled()
{
    InheritSetUpAndTearDown testFixture = new
          InheritSetUpAndTearDown();

    TestSuite suite = new TestSuite("SetUpAndTearDownSuite");
    suite.Add(testFixture);
    suite.Run(NullListener.NULL);
```

```
        Assertion.Assert(testFixture.wasSetUpCalled);
        Assertion.Assert(testFixture.wasTearDownCalled);

    }
```

ExpectedException

This is the way to specify that the execution of a test will throw an exception. This attribute takes a parameter which is a Type. The runner will execute the test and if it throws an the specific exception then the test passes. If it throws a different exception the test will fail. This is true even if the thrown exception inherits from the expected exception.

ExpectedException Example

```
namespace Nunit.Tests
{
    using System;
    using Nunit.Framework;
    using Nunit.Core;

    [TestFixture]
    public class ExpectExceptionTest
    {
        [Test]
        [ExpectedException(typeof(Exception))]
        public void TestSingle()
        {
            throw new Exception("single exception");
        }
    }
}
```


Suite

The Suite Attribute is used to define subsets of suites based on user preference. The belief by the developers of this version is that the need for this will diminish because of the dynamic creation mechanism provided by the framework. However it is provided for backwards compatibility.

Note: There is no way to run user-defined suites in the forms interface.

Suite Attribute syntax

The following is `Nunit.Tests.AllTests` from the `nunit.tests.dll`.

```
namespace Nunit.Tests
{
   using System;
   using Nunit.Framework;
   using Nunit.Core;

   public class AllTests
   {
      [Suite]
      public static TestSuite Suite
      {
         get
         {
            TestSuite suite = new TestSuite("All Tests");
            suite.Add(new OneTestCase());
            suite.Add(new Assemblies.AssemblyTests());
            suite.Add(new AssertionTest());
            return suite;
         }
      }
   }
}
```

Ignore

The ignore attribute is an attribute to not run a test or test fixture for a period of time. The person marks either a Test or a TestFixture with the Ignore Attribute. The running program sees the attribute and does not run the test or tests. The progress bar will turn yellow if a test is not run and the test will be mentioned in the reports that it was not run.

This feature should be used to temporally not run a test or fixture. This is a better mechanism than commenting out the test or renaming methods since the tests will be compiled with the rest of the code and there is an indication at run time that a test is not being run. This insures that tests will not be forgotten.

Test Fixture Syntax

```
[TestFicxture]
[Ignore("Reason")]
public class MyTests
{
   // all tests will not be run
}
```

Test Syntax

```
[TestFicxture]
public class MyTests
{
   [Test] public void MyTest()
   {
      // test will be run
   }

   [Test]
   [Ignore("Reason")]
   public void MyIgnoredTests()
   {
      // test will not be run
   }
}
```

Upgrading from V1.11 to V2.0

Upgrading from V1.11 to V2.0 requires a minimal amount of work. Since the framework still looks for test methods by name in addition to the attributes no test method will need to be modified to upgrade to the new version. The only changes are related to the new namespace. Change `NUnit.Framework` to `Nunit.Framework`. The other change is to use our newly created adapter "`Nunit.Framework.TestFixture`" which is specifically for backward compatibility. In all previous releases the way a class was marked as a fixture was they inherited from TestCase. To upgrade, change `TestCase` to `TestFixture` and you are finished.

Suite property

The existing `Suite` property will not be found by the new program. These must be changed to the "Suite" attribute for the test runners to find them. Another alternative is that these suites are no longer needed due to the automatic capability that is built in to the new version.

AssertionFailedError

If you have written code expecting the exception, `AssertionFailedError` this must be changed to `AssertionException`.

Command Line Parameters

Forms Interface

The forms interface has two command line options. If the program is started without any command line parameters it automatically loads the most recently loaded assembly. It does not automatically run it just loads the assembly. The forms interface also keeps track of the 5 most recently loaded assemblies. To access these see the File->Recent Assemblies menu item.

The other option is to specify the assembly on the command line. The following will start the forms interface with the assembly nunit.tests.dll

```
nunit-gui.exe /assembly:nunit.tests.dll
```

Console Interface

The console interface has a few additional options compared to the forms interface. The console program must always specify a command line parameter. The console interface always creates an XML representation of the test results. This file by default is called TestResult.xml and is placed in the working directory.

Note: By default the nunit-console program is not added to your path. You must do this manually if this is the desired behavior.

Specifying an Assembly

The console program must always have an assembly specified. To run the tests contained in the nunit.tests.dll use the following command.

```
nunit-console /assembly:nunit.tests.dll
```

Specifying an Assembly and a Fixture

When specifying a a fixture you must give the full name of the test fixture along with the containing assembly. For example, to run the Nunit.Tests.AssertionTests in the nunit.tests.dll assembly use the following command.

```
nunit-console /fixture:Nunit.Tests.AssertionTests
/assembly:nunit.tests.dll
```

Specifying the XML file name

As stated above the console program always creates an XML representation of the test results. To change the name to console-test.xml use the following command line option,

```
nunit-console /assembly:nunit.tests.dll /xml:console-test.xml
```

Specifying the Transform file

The console interface uses XSLT to transform the test results from the XML file to what is printed to the screen when the program executes. The console interface has a default transformation that is part of the executable. To specify your own transformation named myTransform.xslt use the following command line option.

```
nunit-console /assembly:nunit.tests.dll
/transform:myTransform.xslt
```

Note: For additional information see the XML schema for the test results. This file is in the same directory as the executable and is called `Results.xsd`. The default transform `Summary.xslt` is located in the nunit-console source directory.

Samples

- C# - This sample demonstrates three failing unit tests written in C#.
- VB.NET - This sample demonstrates what should be three failing unit tests, but VB.NET does not throw an exception when you divide by zero even though the documentation says it should.
- Managed C++ - This is the same example as the others with 3 failing unit tests. This is correct when compiled in Debug mode. In Release mode this behaves like VB.NET in that the divide by zero test succeeds.
- Visual J# - This has the same tests as the other samples, except written in Visual J#. If you have not installed this when you go to open the Visual Studio solution file it will inform you that the file solution file cannot be loaded.
- Money - This is C# version of the money example which is found in most xUnit implementations. Thanks Kent.
- Money-port - This is an example of the minimum amount of work that is needed to upgrade from previous versions of NUnit to this version. The minimum amount of work involves having your test class inherit from `TestFixture` instead of `TestCase`. You also must change the namespace to be `Nunit.Framework` instead of `NUnit.Framework` (case is important).

To-do List

- Console program with no parameters runs the most recent assembly.
- NAnt Integration
- Running user defined suites in the forms interface. They currently can only be run from the console.
- `nunit.org` website for tips, tools and discussion.
- Visual Studio Integration
  - Open a Visual Studio.net solution file and build a suite of all TestFixtures in the solution file.
  - Templates for TestFixture file types and Test assemblies.
  - Navigation between output of the forms window to the specific file in Visual Studio.
- Possibly using a config file for a test assembly to specify paths to search for dependant assemblies.

Code contribution philosophy

First and foremost we have done our best to insure that this program was developed test-first. As such we will accept no code changes without associated tests. As for bug fixes we plan to follow the procedure that we write a failing unit test first and then fix it. In this fashion the number of tests that we have will grow over time. Lastly, if a change breaks an existing test it is the responsibility of the person making the change to first understand the ramification of the change and either fix the test

or alter the modification that caused the problem. That said, if you are interested, so are we, please help make Nunit the best xUnit tool in any language.

Copyright

Developers

- James Newkirk (newkirk_james@hotmail.com)
- Michael C. Two (mailto:2@thoughtworks.com)
- Alexei A. Vorontsov (mailto:aavoront@thoughtworks.com)
- Phillip Craig (mailto:philip@pobox.com)

Thanks and Acknowledgements

- Doug de la Torre
- Ethan Smith
- Joe Hildebrand
- Kent Beck and Erich Gamma for starting the whole thing.