Chapter 2

NUnit Fundamentals

NUnit Fundamentals

Objectives

After completing this unit you will be able to:

- **Describe the general structure of unit tests.**
- **Outline the features of the NUnit Framework:**
  - Assertions
  - Test Cases
  - Test Fixtures
  - Test Runners
- **Provide for setup and tear down of tests.**
- **Use the NUnit GUI test tool both standalone and integrated with Visual Studio.**

Structure of Unit Tests

- **As we saw in Chapter 1, *unit tests* are tests of specific program components.**
- **Test code is for internal use only and is separate from the production code being tested.**
- **Test code is responsible for doing several things:**

1. Set up resources needed for the test.
2. Call the function or method to be tested.
3. Verify that the called function behaved as expected.
4. Clean up after itself.
   - For simple tests, no special setup or cleanup may be required, but steps 2 and 3 are always performed.

Assertions

- **A central requirement of unit tests is that they must be self-verifying.**
  - It would be very inefficient to require a separate process or human intervention to examine the output of the tests to determine whether or not they passed.
- **We need a mechanism to support self-verification.**
- **This mechanism is called an *assertion*.**
  - An assertion is a statement that some condition is true, and a report will be made if the condition is not true.
- **The notion of assertion is common in many programming languages and frameworks.**
- **The ANSI C runtime library has an *assert()* method that can be used in the C language.**

Assert Example

- **A simple example illustrates the C *assert()* function.**
  - See **Chap02\CMax\Step1**.
  - This example provides unit tests for a **findmax()** function that finds the maximum of three integers.

```
// CMax.c

#include <stdio.h>
#include <assert.h>

int findmax(int x, int y, int z)
{
   int max = x;
   if (y > x)
      max = y;
   if (z > x)
      max = z;
   return max;
}

int main()
{
   assert(findmax(4, 3, 2) == 4);
   printf("Test 1 passed\n");
   assert(findmax(3, 4, 2) == 4);
   printf("Test 2 passed\n");
   assert(findmax(3, 2, 4) == 4);
   printf("Test 3 passed\n");
   assert(findmax(2, 4, 3) == 4);
   printf("Test 4 passed\n");
}
```

Assert Example (Cont'd)

- **The *findmax()* function has a bug. The first three tests pass, but the fourth one fails.**

```
Test 1 passed
Test 2 passed
Test 3 passed
Assertion failed: findmax(2, 4, 3) == 4, file
c:\oic\nunitcs\chap02\cmax\step1\c
max.c, line 24

This application has requested the Runtime to
terminate it in an unusual way.
Please contact the application's support team for
more information.
```

- **Step 2 fixes the bug.**

```
int findmax(int x, int y, int z)
{
   int max = x;
   if (y > max)
      max = y;
   if (z > max)
      max = z;
   return max;
}
```

  - Here is the output:

```
Test 1 passed
Test 2 passed
Test 3 passed
Test 4 passed
```

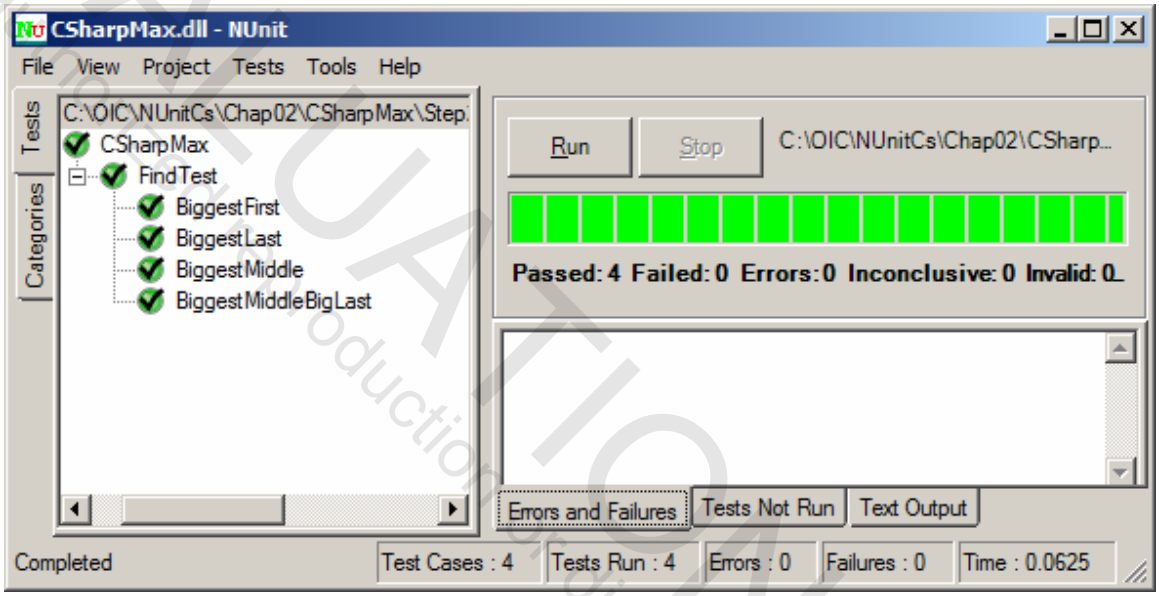
NUnit Framework

- **Although completely hand-written unit tests using only primitive library features are feasible, it is not efficient.**
- **The NUnit Framework that we introduced in Chapter 1 provides many features to simplify writing and running unit tests:**
  - An **Assert** class with a variety of methods for testing assertion conditions.
  - A custom attribute **[Test]** to designate a method as a test case.
  - A custom attribute **[TestFixture]** to designate a class that encapsulates a group of test methods sharing a common set of run-time resources.
  - A test runner that automates the running of all the tests.

Lab 2A

NUnit Tests of Maximum Method

In this lab, you will develop a C# version of the function to find the maximum of three integers. You will then review use of NUnit by developing and running test cases.



Detailed instructions are contained in the Lab 2A write-up at the end of the chapter.

Suggested time: 20 minutes

NUnit Assert Class

- **The NUnit Framework provides an *Assert* class with a number of static methods.**
  - These helper methods can help you determine whether the test passed or failed.
  - They record failures (when an assertion is false) and errors (when an unexpected exception occurs).
- **Failures and errors are reported through NUnit.**
  - You won't see a system-generated exception message as in a failure of the **assert()** method of the C runtime library.
  - In the **nunit.exe** tool you will see red with an X displayed, plus explanatory text.
- **When a failure or error occurs, the current test method is aborted, and execution continues with the next method in the test fixture.**

Assert.AreEqual()

- **There are many overload versions of the *AreEqual()* method.**

```
Assert.AreEqual(expected, actual)
Assert.AreEqual(expected, actual, message)
```

  - Typically **expected** is a hard coded value representing the value you expect to see.
  - **actual** is the value actually produced by the code that you are testing.
  - The optional parameter **message** is a string which will be displayed by NUnit upon failure.

- **Any object may be tested for equality.**
  - The **Equals()** method of the **Object** class will be used for comparison.
  - There are special overloads for the built-in data types of **int**, **uint**, **decimal**, **float**, and **double**.
  - For **float** and **decimal** there is an overload available that takes a parameter **delta** that may be used as a tolerance, specifying how close to equals the result should be.

```
Assert.AreEqual(expected, actual, delta)
```

- **In Visual Studio you can use Intellisense to see all the possible overloaded methods.**

More Assert Methods

- **There are many other Assert methods, which can be viewed via Intellisense.**
  - Many come in pairs with a **Not** variant, and there is always an optional string **message** parameter.

AreNotEqual	**expected** does not equal **actual**
IsNull IsNotNull	Given object is **null** (or is not **null**)
AreSame AreNotSame	**expected** and **actual** refer (or do not refer) to the same object
IsTrue	Given Boolean condition is true
IsFalse	Given Boolean condition is false
Fail	Fail the test immediately
Greater	**expected** is greater than **actual** (objects implement IComparable)
Less	**expected** is less than **actual**
IsEmpty IsNotEmpty	String or collection is (or is not) empty
Contains	**expected** object is contained in the **actual** collection
IsInstanceOfType IsNotInstanceOfType	**actual** object is (or is not) of **expected** type

Test Case

- **The fundamental unit of testing with NUnit is a *test case*.**
  - A test case is a programmer test, which is a low-level test intended to verify behavior at the method or class level.
  - A test case is **self-validating**, having a built-in mechanism to report success or failure.
  - A test case can be automatically discovered by a test runner.
  - A test case can be automatically executed by a test runner.
  - A test case executes independently of other test cases; one test case should not produce any side effects that could change the results from other test cases.

Test Methods

- **In NUnit, test cases are specified by *test methods*, which are methods of a *test class*.**
- **A test method is marked by the *[Test]* attribute.**

```
[Test]
public void BiggestFirst()
{
   Assert.AreEqual(Find.Max(4, 3, 2), 4);
}
```

- **A test method must have the following features:**
  - It is declared as **public**.
  - It is an instance method (not static).
  - It returns **void**.
  - It takes no parameters.

Test Fixture

- **In NUnit, test methods are encapsulated in a test class, called a *test fixture*.**
- **The test methods in a test fixture share a common set of resources.**
- **The test class is marked with the *[TestFixture]* attribute.**

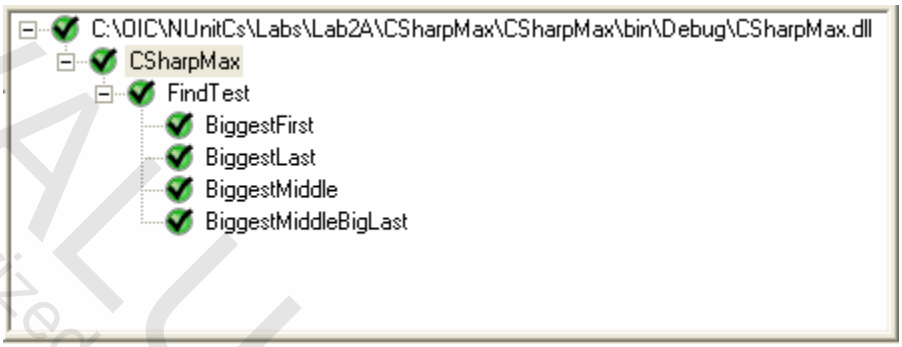
```
[TestFixture]
public class FindTest
{
   [Test]
   public void BiggestFirst()
   {
      Assert.AreEqual(Find.Max(4, 3, 2), 4);
   }
   [Test]
   public void BiggestMiddle()
   {
      Assert.AreEqual(Find.Max(3, 4, 2), 4);
   }
   [Test]
   public void BiggestLast()
   {
      Assert.AreEqual(Find.Max(3, 2, 4), 4);
   }
   [Test]
   public void BiggestMiddleBigLast()
   {
      Assert.AreEqual(Find.Max(2, 4, 3), 4);
   }
}
```

Test Runner

- **An essential component of an effective unit testing system is a facility to automatically run the tests.**
- **A *test runner* is a program that automatically discovers test cases, runs them, and reports on the results.**
- **An example of a test runner is the *nunit.exe* program.**
- **A test runner uses .NET reflection to dynamically discover and execute test methods.**
- **The nunit.exe test runner shows test methods grouped under test classes in a tree view control.**

Test Case Hierarchy

- **The test cases are arranged in a hierarchy, represented in nunit.exe by the tree view.**



  - Top node represents the assembly containing the test code.
  - Next come nodes for each namespace in the assembly.
  - Next come nodes for each text fixture in the namespace.
  - The leaf nodes represent test cases.

Ignoring Tests

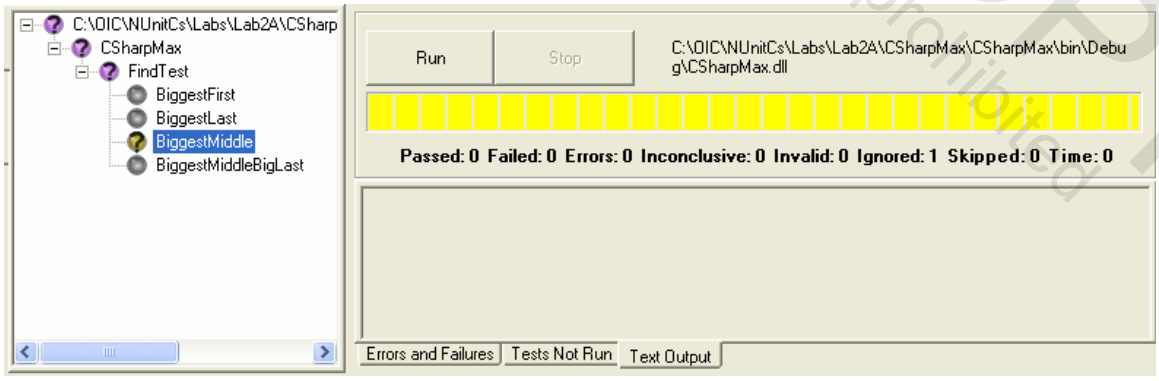
- **A test case can be made non-runnable by marking it with the *[Ignore]* attribute.**

```
[Ignore]
[TestMethod]
public void BiggestMiddle()
{
   Assert.AreEqual(4, Find.Max(3, 4, 2));
}
```

- **This feature may be useful during development to temporarily disable running certain tests which will be known to fail.**
  - The system being tested may have a known bug or have not yet implemented certain functionality.
- **It may also be useful to temporarily disable certain long-running tests that are known to succeed.**

Test Case Selection

- **If you click the Run button in nunit.exe, you will run all the runnable test cases in the assembly.**
  - As discussed on the previous page, a test case can be made non-runnable by marking it with the **[Ignore]** attribute.
- **You can run only the test cases under a particular node by right-clicking on the node and selecting Run from the context menu.**
- **You can run an individual test case in the same way, or simply by double-clicking on the test case leaf node.**
- **The tree view will show all the nodes that are not selected in gray, and the other nodes according to the result:**
  - Green with a check mark if the test case passes
  - Red with an X if the test case fails
  - Yellow with a question mark if the selected test case is not run for any reason



Demo: Using nunit.exe

- **Let's demonstrate a number of features of the nunit.exe test runner using an extension of the *CSharpMax* program you did in the lab.**
  - See **Demos\CSharpMax**, which is backed up in the directory **Chap02\CSharpMax\Step3**.

1. Examine the code in **Find.cs**. In the **Find** class we've added a method **Div()**.

```
public static int Div(int x, int y)
{
   return x / y;
}
```

2. In the **FindTest** class we've marked the second test method with the **[Ignore]** attribute.

```
[TestFixture]
public class FindTest
{
   [Test]
   public void BiggestFirst()
   {
      Assert.AreEqual(Find.Max(4, 3, 2), 4);
   }
   [Test]
   [Ignore]
   public void BiggestMiddle()
   {
      Assert.AreEqual(Find.Max(3, 4, 2), 4);
   }
   ...
```

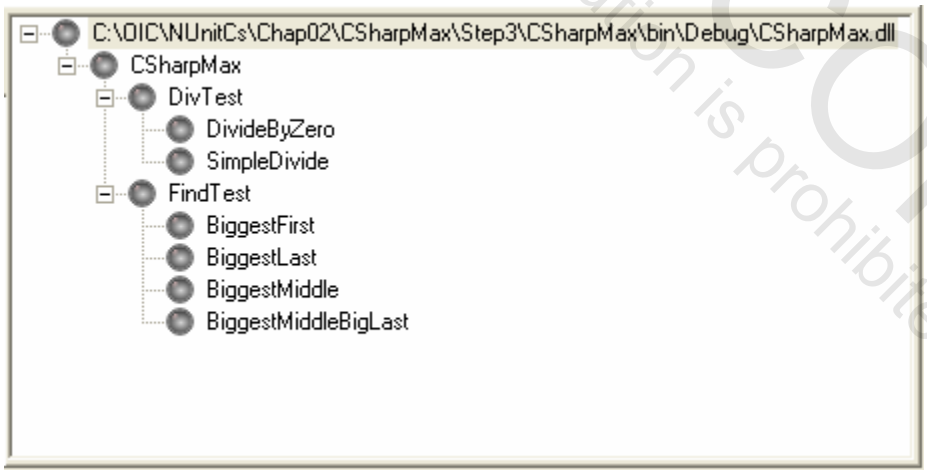
Demo: Using nunit.exe (Cont'd)

3. We've provided a second test class **DivTest** to test the **Div()** method.

```
[TestFixture]
public class DivTest
{
   [Test]
   public void SimpleDivide()
   {
      Assert.AreEqual(Find.Div(10, 2), 5);
   }
   [Test]
   public void DivideByZero()
   {
      // This does not raise an exception!
      Assert.AreEqual(Find.Div(10, 0), 5);
   }
}
```

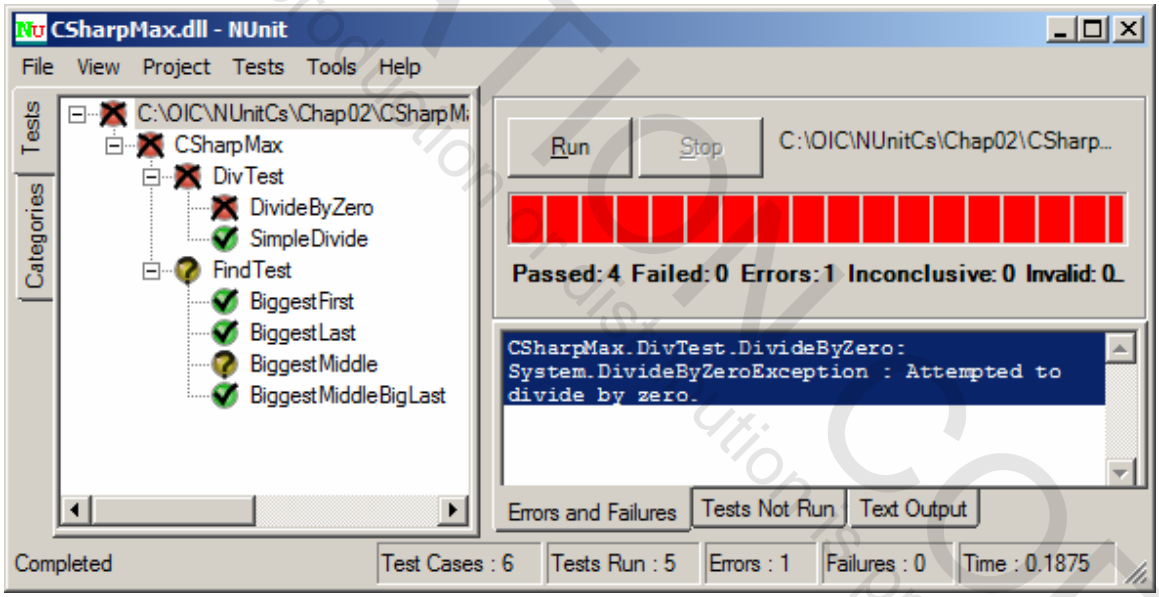
4. Build the class library.

5. Start **nunit.exe** and do File | Open to load **CSharpMax.dll**. Notice the hierarchy of test cases.



Demo: Using nunit.exe (Cont'd)

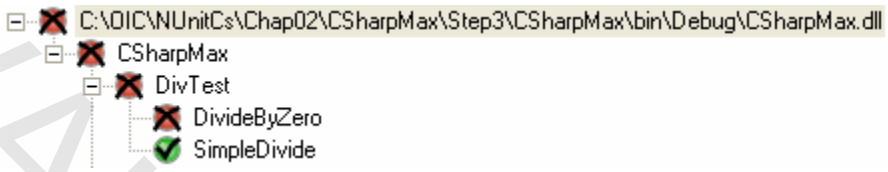
6. Click the Run button to run all the runnable test cases in the assembly.
   - In the **DivTest** class, one test case fails and one succeeds. Note that an exception causes a method to fail as does an assertion failure; the result is reported in red, rather than causing the test runner to abort (as happened earlier in the C language example we looked at earlier).
   - In the **FindTest** class, three tests cases run (all of which succeed), and one test case is not run (the one marked by **[Ignore]**).



7. Experiment with running selected test cases by right-clicking on a node and selecting Run from the context menu. You can also simply double-click a leaf node to run a single test case.

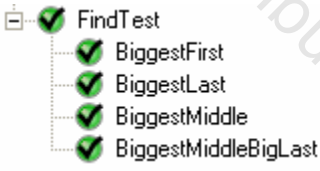
Coloring Parent Nodes

- **A parent node is colored red with an X if any test case in the subtree fails.**



- **A parent node is colored yellow with a question mark if any test case in the subtree does not run.**

- **A parent tree is colored green with a check mark if all the test cases in the subtree succeed.**
  - In our example, remove the **[Ignore]** attribute from the **BiggestMiddle()** method and run again.



Test Setup and Tear Down

- **The NUnit Framework provides custom attributes that you can use to set up and tear down tests.**
- **You can ensure that all tests are initialized in the same manner by means of the *[Setup]* attribute.**
  - Place this attribute before a method, which will then be called prior to the execution of each test in the test fixture.
- **You can ensure that all tests are cleaned up in the same manner by means of the *[TearDown]* attribute.**
  - Place this attribute before a method, which will then be called immediately after the execution of each test in the test fixture.

```
[TestFixture]
public class QueueFixture
{
   private MyQueue que;
   private int SizeQueue;

   [SetUp]
   public void SetupQueue()
   {
      que = new MyQueue(SizeQueue);
      Console.WriteLine(
         "Setup up queue of size {0}", SizeQueue);
   }
   [TearDown]
   public void TearDownQueue()
   {
      Console.WriteLine("Tear down queue");
   }
   ...
```

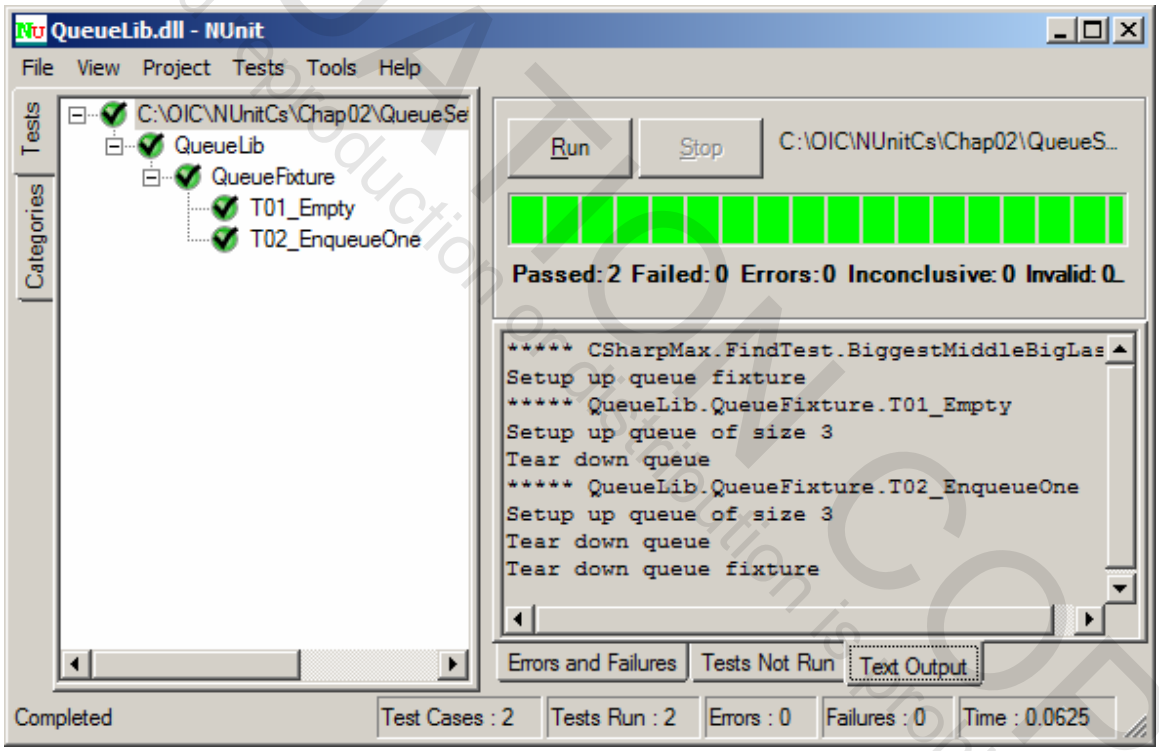
Test Fixture Setup and Tear Down

- **The NUnit Framework also provides custom attributes that you can use to set up and tear down test fixtures.**
  - While test setup and teardown are done on a per-method basis, test fixture setup and teardown are performed on a per-class basis.
  - A method marked with the **[TestFixtureSetup]** attribute is called once for the entire test fixture, before any of the test cases are executed.
  - A method marked with the **[TestFixtureTearDown]** attribute is called once for the entire test fixture, after all of the test cases are executed.

```
[TestFixture]
public class QueueFixture
{
   ...
   [TestFixtureSetUp]
   public void SetupQueueFixture()
   {
      SizeQueue = 3;
      Console.WriteLine("Setup up queue fixture");
   }
   [TestFixtureTearDown]
   public void TearDownQueueFixture()
   {
      Console.WriteLine("Tear down queue fixture");
   }
   ...
```

Test Setup Example

- **A version of the tests for our queue class illustrates test setup and teardown.**
  - See **Chap02\QueueSetup**.
  - The relevant code for this example was shown on the previous pages.
  - Here is the result of running these tests under NUnit. You can examine the console output from the Text Output tab.



Using NUnit with Visual Studio

- **Instead of running nunit.exe as a standalone program, you may find it convenient to start NUnit from within Visual Studio.**
  - There are commercial add-ons to NUnit that you can use.
  - However, you can gain much of the convenience without any add-ons by making a simple setting to your Visual Studio class library project.
- **We will illustrate by configuring the QueueSetup example to run the tests from within Visual Studio.**
  - Do your work in **Demos\QueueSetup**, which is backed up in **Chap02\QueueSetup**.
  - The final project is in **Chap02\QueueSetupVs**.
- **Also it is possible to tightly integrate unit test frameworks such as NUnit into Visual Studio by making use of a *test adapter*[1].**
  - However, the Visual Studio test runner is not as fully featured as the NUnit test runner.

[1] At the time of updating this course an NUnit test adapter for Visual Studio 2015 was not available.