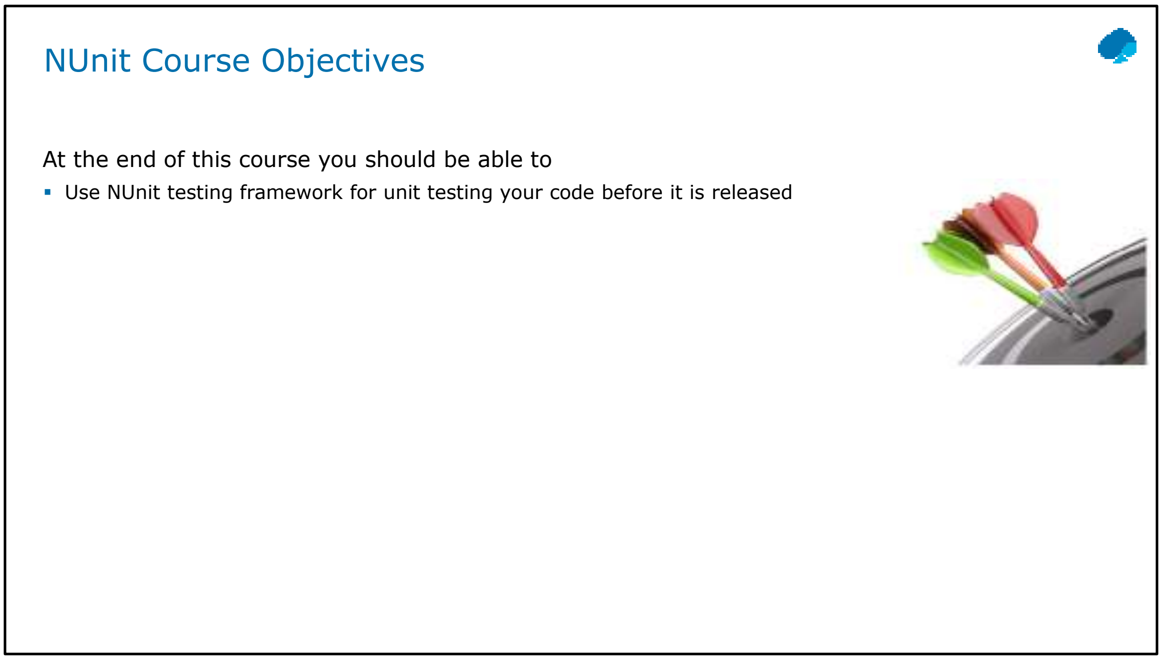
Developer Tools for .NET

Lesson 1

Unit Testing with NUnit & Overview of NCover



NUnit Course Objectives

At the end of this course you should be able to

- Use NUnit testing framework for unit testing your code before it is released

Goals one is going to attain at the end of course is being able to use NUnit testing framework for unit testing your code before it is released.
What is Unit Testing?
Testing of individual software components. Each module is tested alone in an attempt to discover any errors in its code. In computer programming, a unit test is a method of testing the correctness of a particular module of source code.

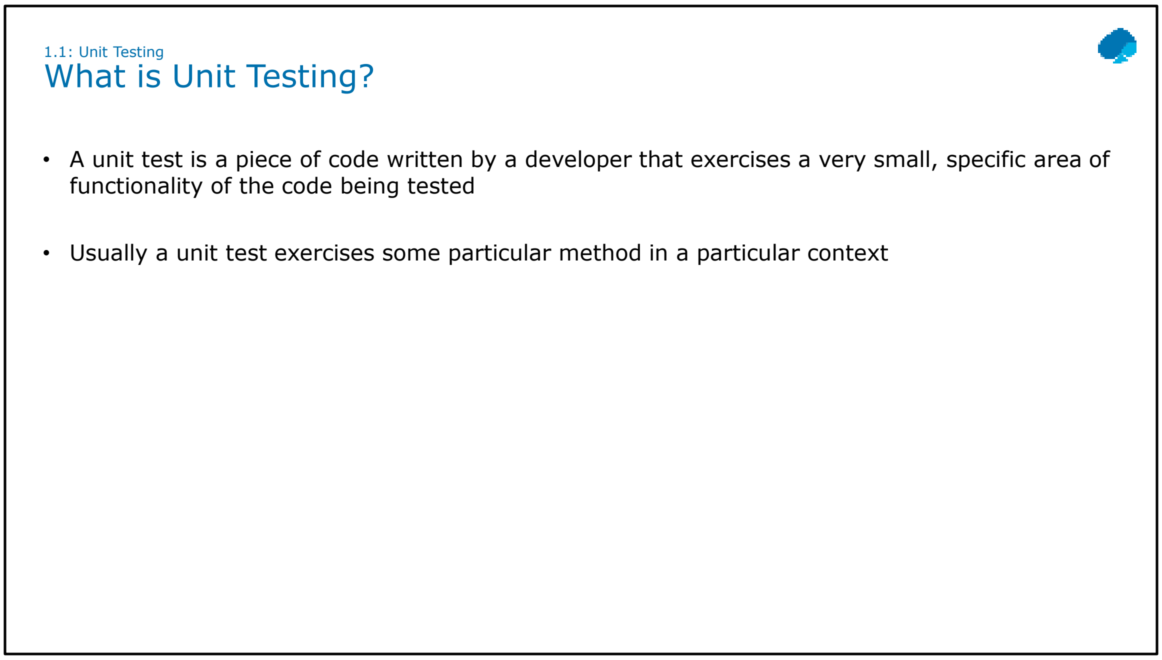
The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect. Each unit is tested separately before integrating them into modules to test the interfaces between modules. Unit testing has proven its value in that a large percentage of defects are identified during its use.

Qualities of Good Unit Test:
These run fast.
These help to localize problems.

Why do we need Unit Testing?
Unit tests find problems early in the development cycle.
Developers will be less afraid to change existing code.
The development process becomes more flexible.
Software development will become more predictable and repeatable

Overview

- Overview
- What is Unit Testing?
- Advantages
- Disadvantages
- Introduction to XUnit Tools
- The NUnit Testing Framework
- Using Nunit
- Microsoft Test Framework
- Code coverage with NCover

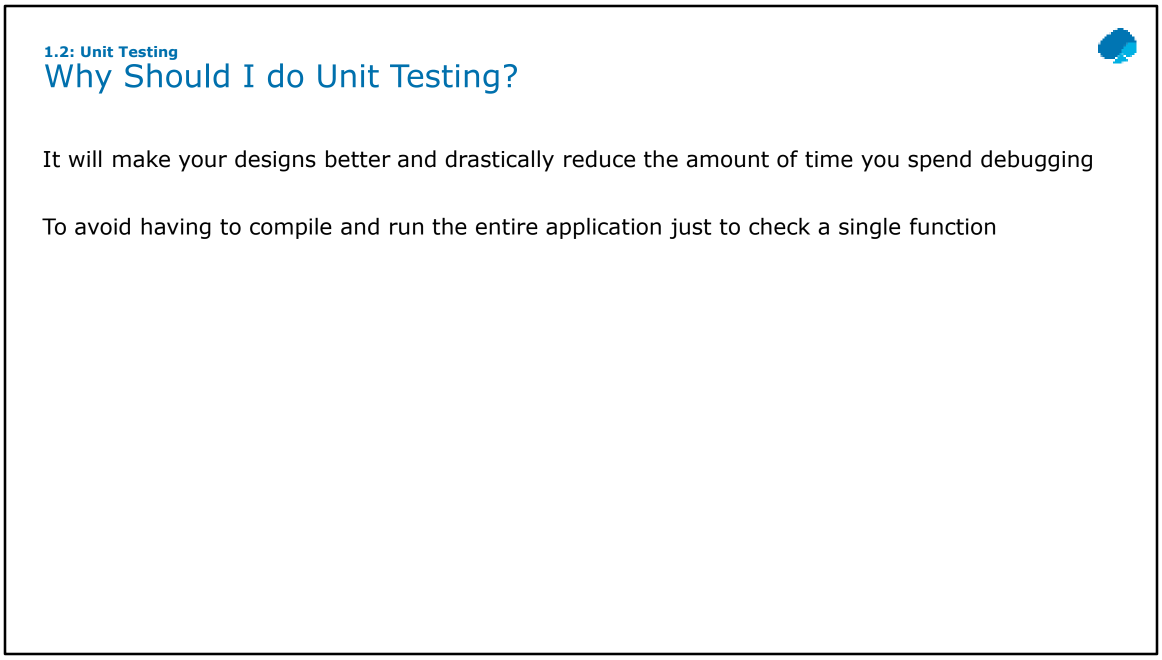


1.1: Unit Testing

What is Unit Testing?

- A unit test is a piece of code written by a developer that exercises a very small, specific area of functionality of the code being tested
- Usually a unit test exercises some particular method in a particular context

Unit Tests are "programs written to run in batches and test classes. Each typically sends a class a fixed message and verifies it returns the predicted answer." In practical terms this means that you write programs that test the public interfaces of all of the classes in your application. This is not requirements testing or acceptance testing. Rather it is testing to ensure the methods you write are doing what you expect them to do. This can be very challenging to do well. First of all, you have to decide what tools you will use to build your tests. In the past we had large testing engines with complicated scripting languages that were great for dedicated QA teams, but weren't very good for unit testing. What journeyman programmers need is a toolkit that lets them develop tests using the same language and IDE that they are using to develop the application. Most modern Unit Testing frameworks are derived from the framework created by Kent Beck for the first XP project, the Chrysler C3 Project. It was written in Smalltalk and still exists today, although it has gone through many revisions. Later, Kent and Erich Gamma (of Patterns fame) ported it to Java and called it jUnit. Since then, it has been ported to many different languages, including C++, VB, Python, Perl and more.


1.2: Unit Testing

Why Should I do Unit Testing?

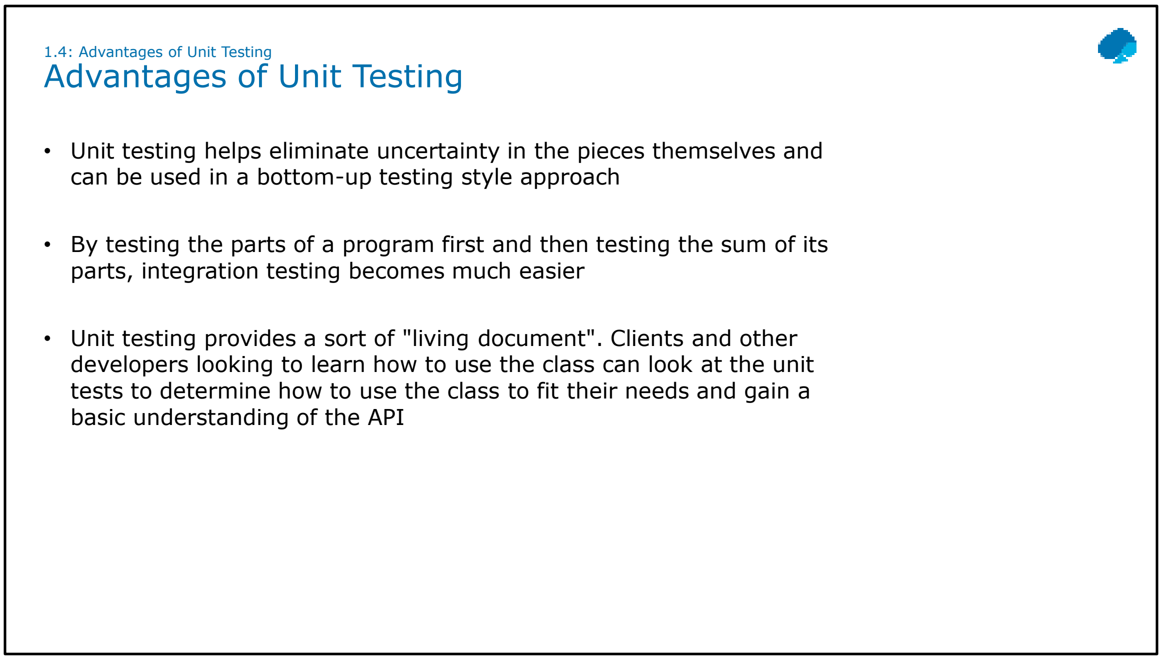
It will make your designs better and drastically reduce the amount of time you spend debugging

To avoid having to compile and run the entire application just to check a single function

This reduces the cycle time between writing code and testing it. The shorter this cycle the better able you are to test specific functionality.

When Should I do Unit Testing?

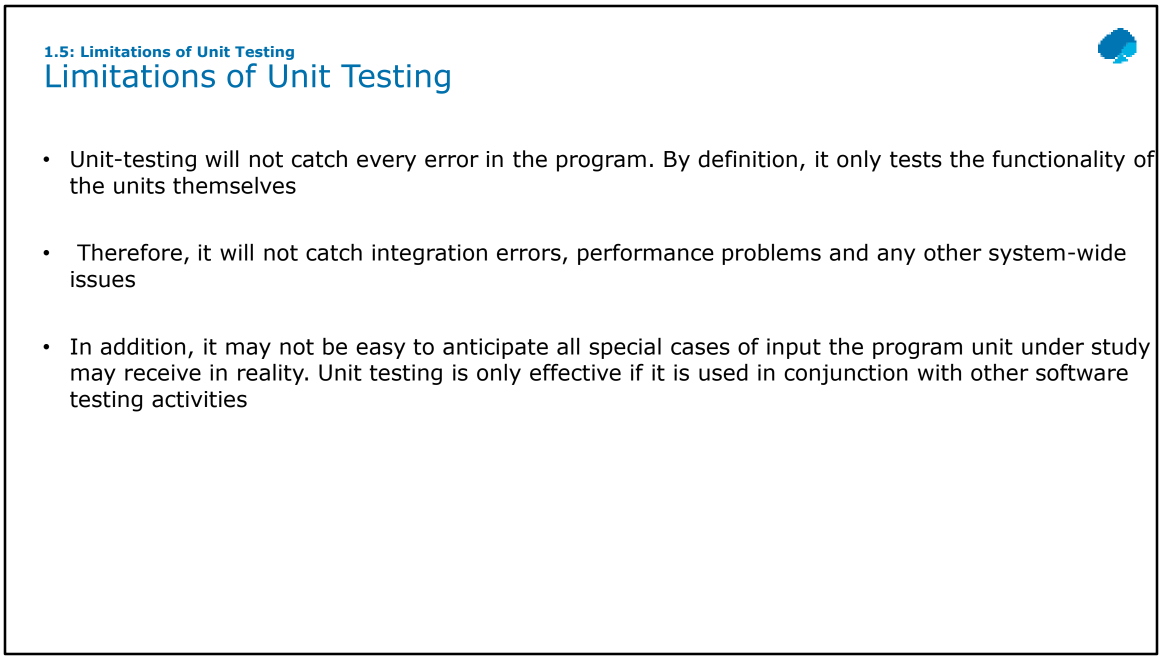
- Before you start to write code!!
- As you write code!!
- Any other time you can!!


1.4: Advantages of Unit Testing

Advantages of Unit Testing

- Unit testing helps eliminate uncertainty in the pieces themselves and can be used in a bottom-up testing style approach
- By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier
- Unit testing provides a sort of "living document". Clients and other developers looking to learn how to use the class can look at the unit tests to determine how to use the class to fit their needs and gain a basic understanding of the API

Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database. This is a mistake, because a unit test should never go outside of its own class boundary. As a result, the software developer abstracts an interface around the database connection, and then implements that interface with their own mock object. This results in loosely coupled code, minimizing dependencies in the system



1.5: Limitations of Unit Testing

Limitations of Unit Testing

- Unit-testing will not catch every error in the program. By definition, it only tests the functionality of the units themselves
- Therefore, it will not catch integration errors, performance problems and any other system-wide issues
- In addition, it may not be easy to anticipate all special cases of input the program unit under study may receive in reality. Unit testing is only effective if it is used in conjunction with other software testing activities

It is unrealistic to test all possible input combinations for any non-trivial piece of software. A unit test can only show the presence of errors; it cannot show the absence of errors.

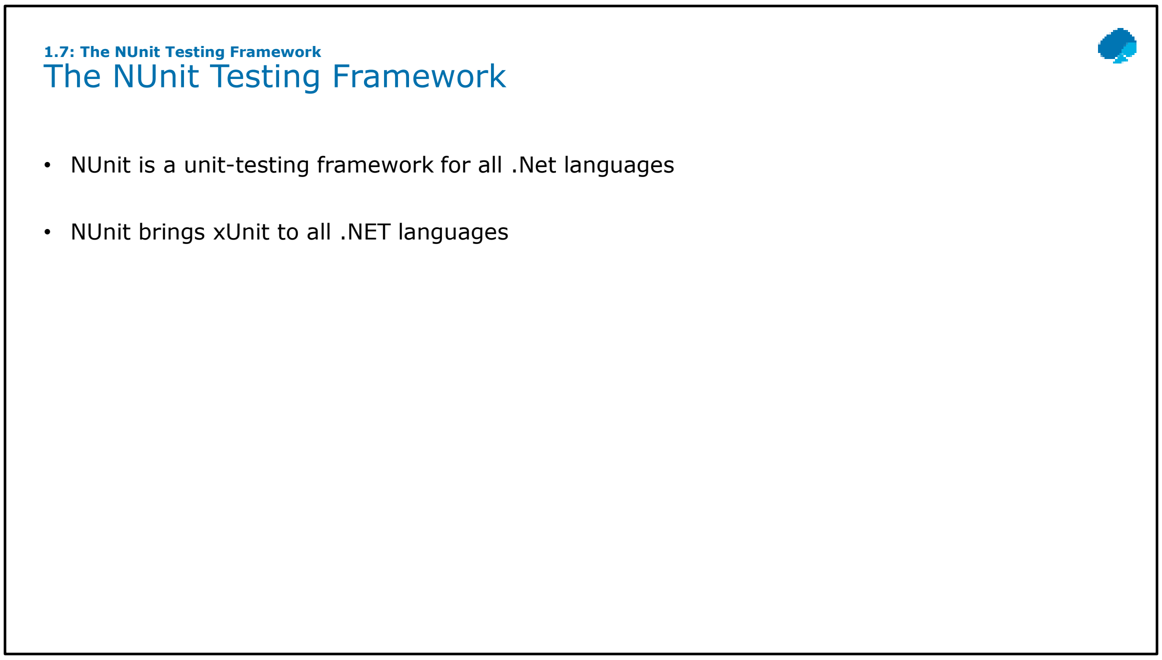
XUnits Tools

- In order to run tests in the code, people typically use XUnit tools
- JUnit for Java, NUnit for .NET (nunit.org)

- These tools often come in two flavors: command-line and GUI

- XUnit tools allow you to run all your tests and see which passed and which failed

- Best of all, these tools are free

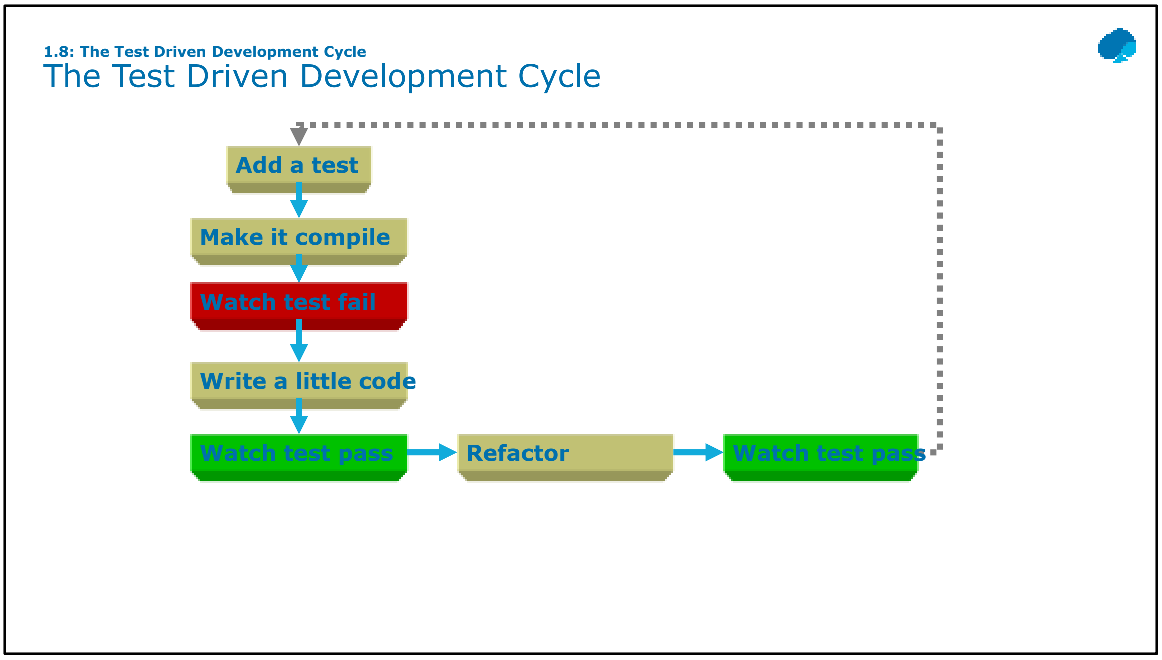


1.7: The NUnit Testing Framework

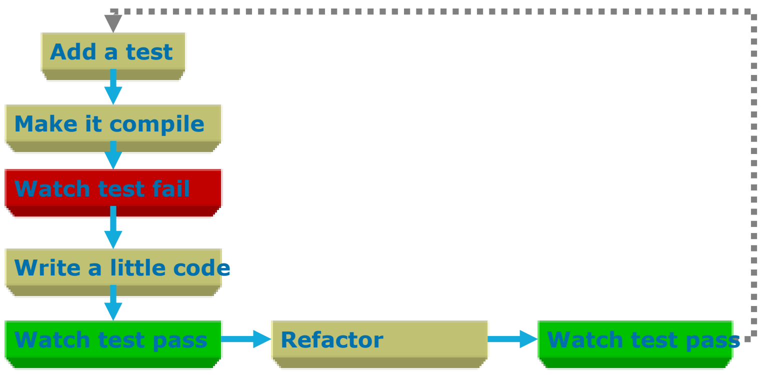
The NUnit Testing Framework

- NUnit is a unit-testing framework for all .Net languages
- NUnit brings xUnit to all .NET languages

NUnit is an application designed to facilitate unit testing. It consists of both a command line and Window's interface, allowing it to be used both interactively and in automated test batches or integrated with the build process.

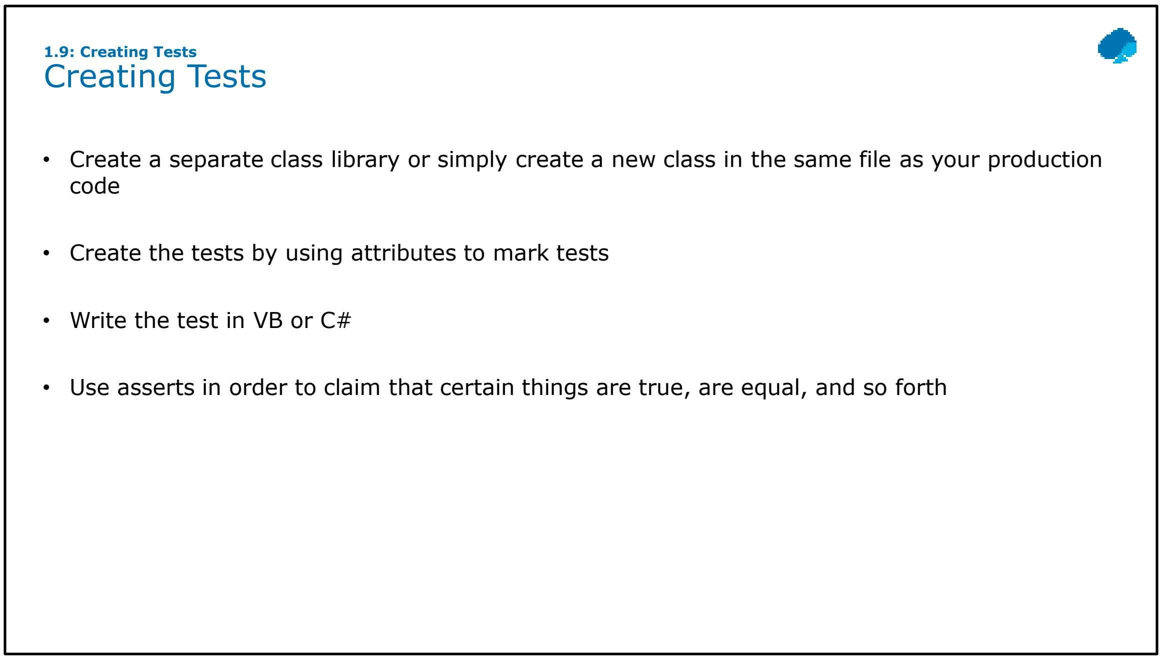

1.8: The Test Driven Development Cycle

The Test Driven Development Cycle



Instead of writing functional code first and then your testing code as an afterthought, if you write it at all, you instead write your test code before your functional code. Furthermore, you do so in very small steps – one test and a small bit of corresponding functional code at a time. A programmer taking a TDD approach refuses to write a new function until there is first a test that fails because that function isn't present.

Why TDD?
A significant advantage of TDD is that it enables you to take small steps when writing software.

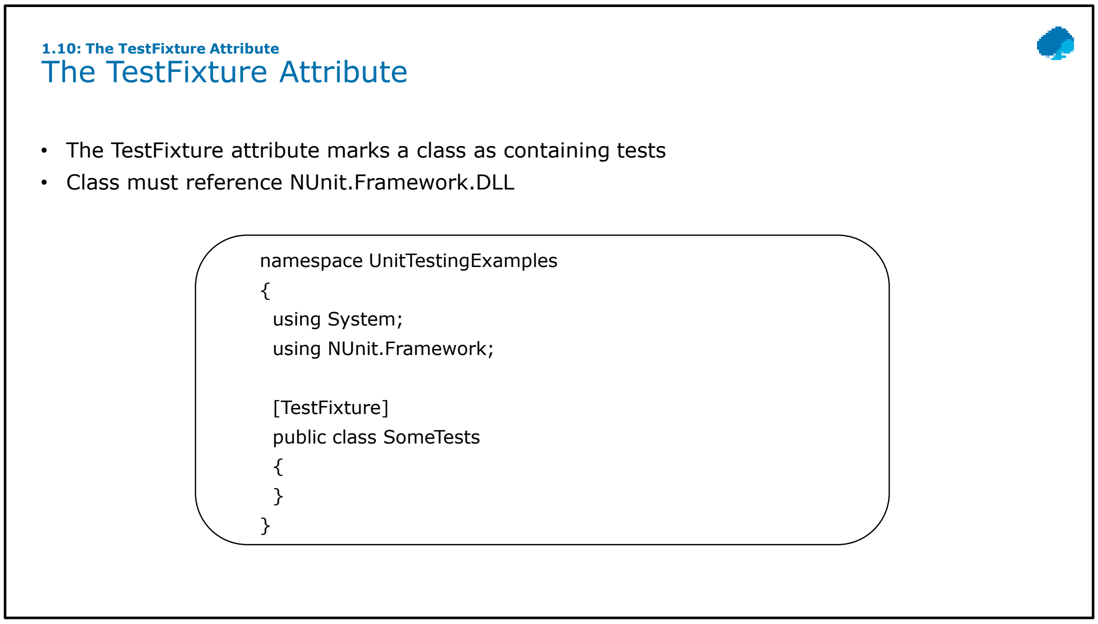


1.9: Creating Tests

Creating Tests

- Create a separate class library or simply create a new class in the same file as your production code
- Create the tests by using attributes to mark tests
- Write the test in VB or C#
- Use asserts in order to claim that certain things are true, are equal, and so forth

Assertions are central to unit testing in any of the xUnit frameworks, and NUnit is no exception. NUnit provides a rich set of assertions as static methods of the Assert class.
If an assertion fails, the method call does not return and an error is reported. If a test contains multiple assertions, any that follow the one that failed will not be executed. For this reason, it's usually best to try for one assertion per test.



1.10: The TestFixture Attribute

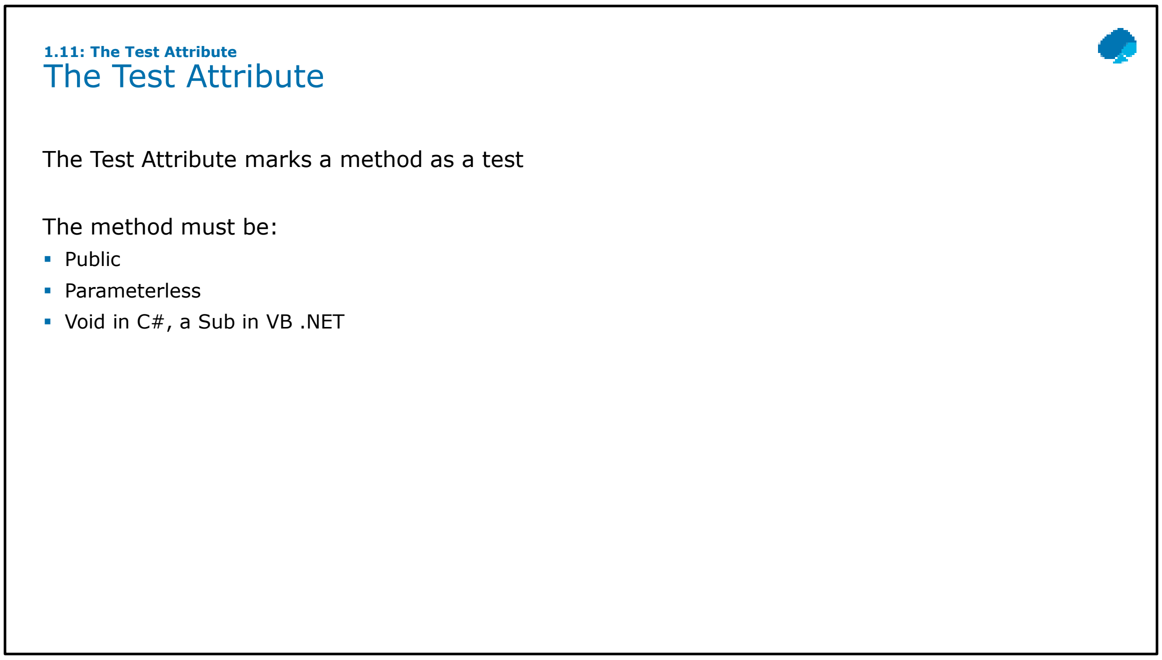
The TestFixture Attribute

- The TestFixture attribute marks a class as containing tests
- Class must reference NUnit.Framework.DLL

```
namespace UnitTestingExamples
{
  using System;
  using NUnit.Framework;

  [TestFixture]
  public class SomeTests
  {
  }
}
```

The TestFixture attribute designates that a class is a test fixture. Classes thus designated contain setup, teardown, and unit tests.



The Test Attribute

The Test Attribute marks a method as a test

The method must be:

- Public
- Parameterless
- Void in C#, a Sub in VB .NET

The Test attribute indicates that a method in the test fixture is a unit test. The unit test engine invokes all the methods indicated with this attribute once per test fixture, invoking the set up method prior to the test method and the tear down method after the test method, if they have been defined.

The test method signature must be specific: public void xxx(), where "xxx" is a descriptive name of the test. In other words, a public method taking no parameters and returning no parameters.

Upon return from the method being tested, the unit test typically performs an assertion to ensure that the method worked correctly.

1.11: The Test Attribute

The Test Attribute (contd..)

The following code illustrates the use of this attribute

```
[TestFixture]
  public class SomeTests
  {
    [Test]
    public void TestOne()
    {
      // Do something...
    }
  }
```

The Asset Class

The Assert class contains a number of static (shared) methods

These methods are used to return a True or False from the test

Methods include:

- AreEqual
- AreSame
- IsTrue
- IsFalse
- IsNull
- IsNotNull
- Fail
- Ignore

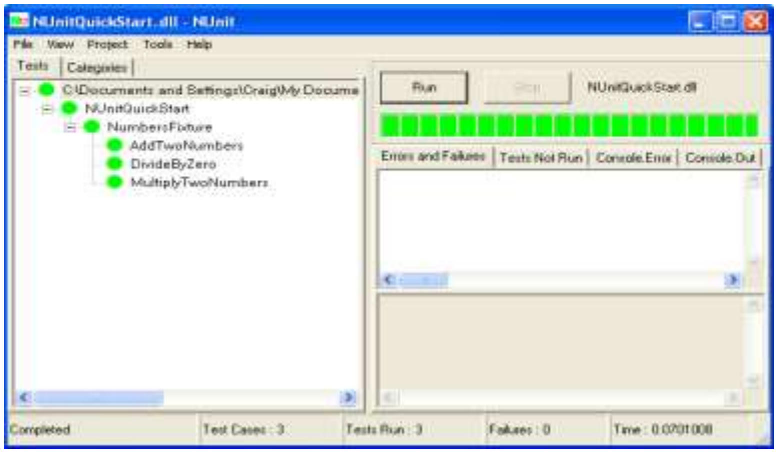
Example: A Simple Test

```
[TestFixture]
public class MyTests
{
  [Test]
  public void AddTest()
  {
    int sum=myMath.Add(2,3);
    Assert.AreEqual(5,sum);
  }
}
```

Running Tests

- NUnit comes with two different Test Runner applications: a Windows GUI app and a console XML app
- To use the GUI app, just run the application and tell it where your test assembly resides
- The test assembly is the class library (or executable) that contains the Test Fixtures
- The app will then show you a graphical view of each class and test that is in that assembly
- To run the entire suite of tests, simple click the Run button

NUnit GUI Screen



C:\Documents and Settings\Craig\My Docume
NUnitQuickStart
NumbersFixture
AddTwoNumbers
DivideByZero
MultiplyTwoNumbers

NUnitQuickStart.dll

Completed Test Cases : 3 Tests Run : 3 Failures : 0 Time : 0.0701008

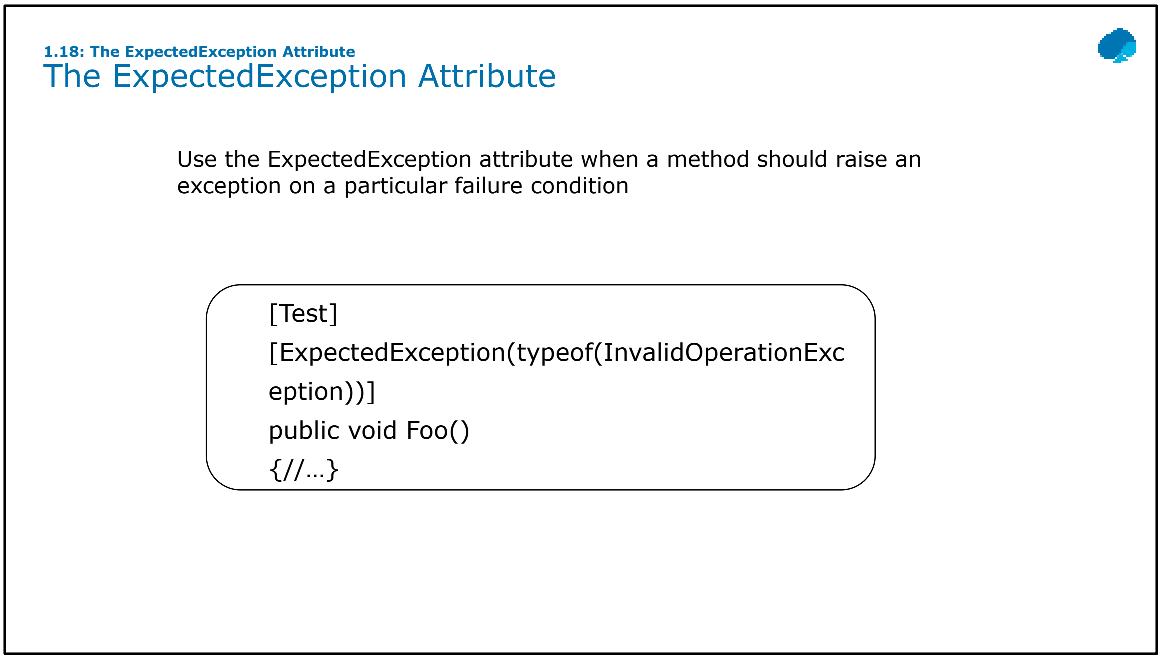
Red/Green/Refactor

- Red/Green/Refactor is the TDD mantra, and it describes the three states you go through when writing code using TDD methods
- Red
  Write a failing test
- Green
  Write the code to satisfy the test
- Refactor
  Improve the code without changing its functionality

1.17: Demo

Demo on Using NUnit Framework

Using NUnit Framework

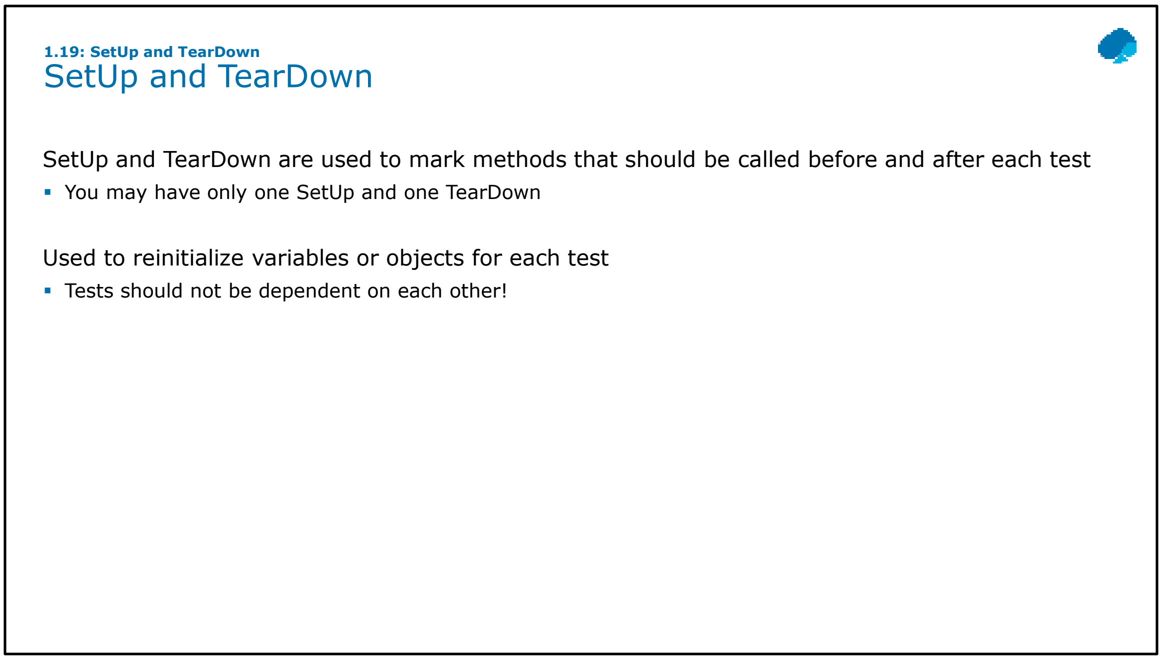


The ExpectedException Attribute

Use the ExpectedException attribute when a method should raise an exception on a particular failure condition

```
[Test]
[ExpectedException(typeof(InvalidOperationExc
eption))]
public void Foo()
{//…}
```

The ExpectedException attribute is an optional attribute that can be added to a unit test method (designated using the Test attribute).
As unit testing should in part verify that the method under test throws the appropriate exceptions, this attribute causes the unit test engine to catch the exception and pass the test if the correct exception is thrown.



SetUp and TearDown

SetUp and TearDown are used to mark methods that should be called before and after each test

- You may have only one SetUp and one TearDown

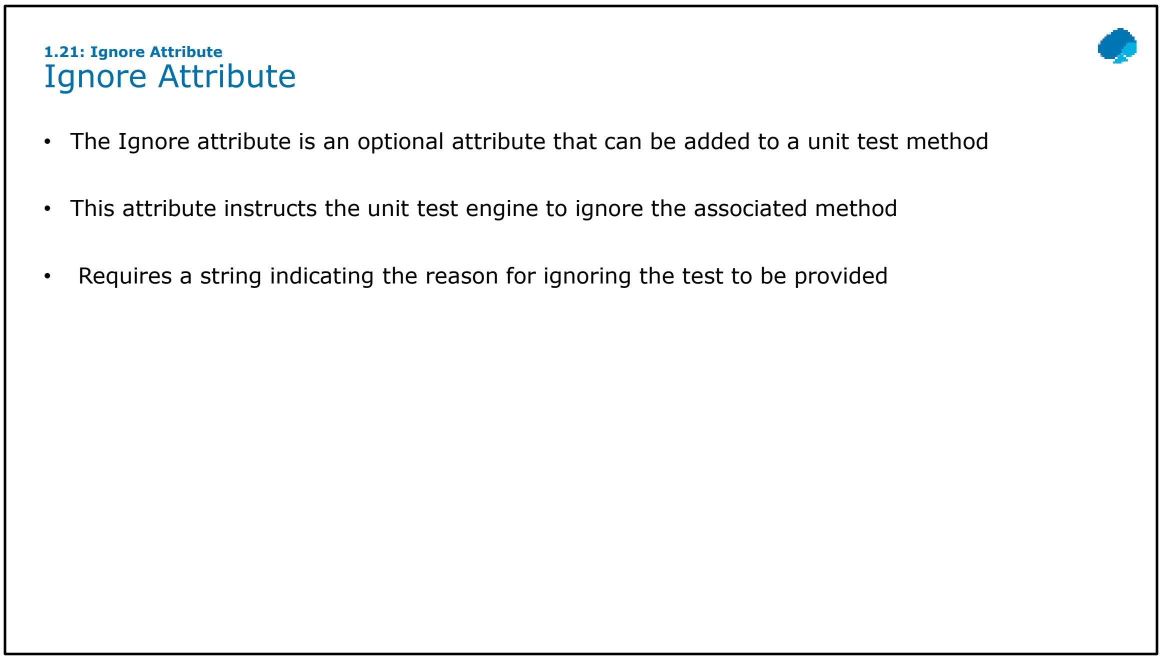
Used to reinitialize variables or objects for each test

- Tests should not be dependent on each other!

The SetUp attribute is associated with a specific method inside the test fixture class. It instructs the unit test engine that this method should be called prior to invoking each unit test. A test fixture can only have one SetUp method.
The TearDown attribute is associated with a specific method inside the test fixture class. It instructs the unit test engine that this method should be called after invoking each unit test. A test fixture can only have one TearDown method.

TestFixtureSetUp and TestFixtureTearDown

- The TestFixtureSetUp and TestFixtureTearDown attributes mark methods that run before and after each test fixture is run
  - You may have only one TestFixtureSetUp and one TestFixtureTearDown
- Usually used to open and close database connections, files, or logging tools



1.21: Ignore Attribute

Ignore Attribute

- The Ignore attribute is an optional attribute that can be added to a unit test method
- This attribute instructs the unit test engine to ignore the associated method
- Requires a string indicating the reason for ignoring the test to be provided

You probably won't use this attribute very often, but when you need it, you'll be glad it's there. If you need to indicate that a test should not be run, use the Ignore attribute.

If you feel the need to temporarily comment out a test, use this instead. It lets you keep the test in your arsenal and it will continually remind you in the test runner output.

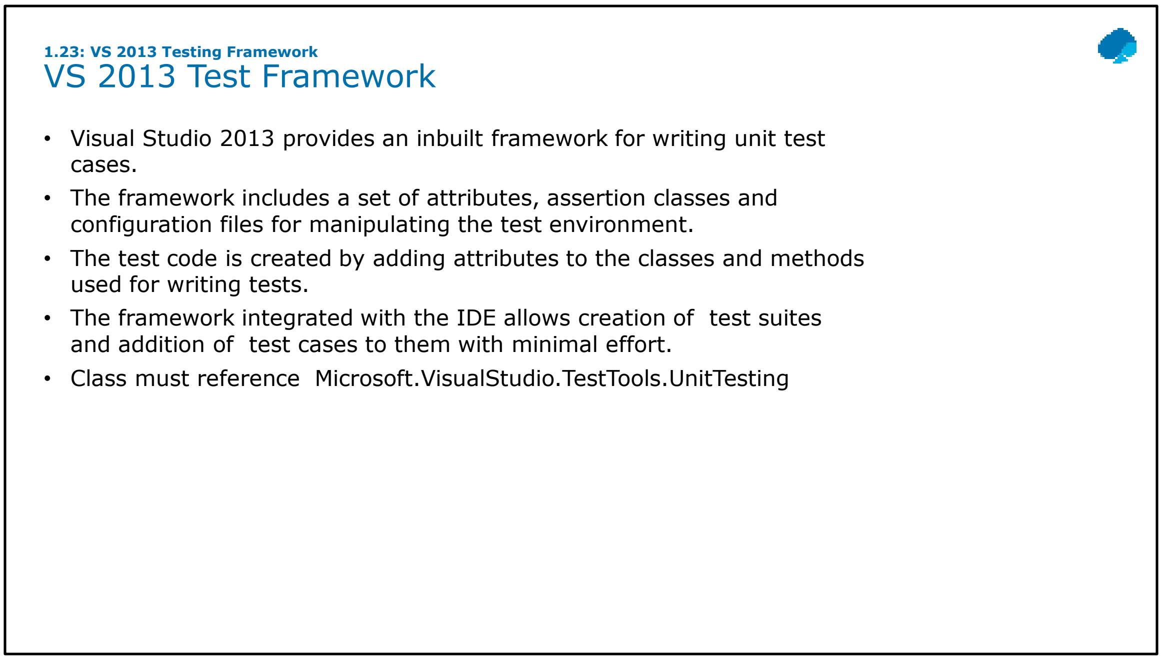
Ignore Attribute (contd..)

```
[TestFixture]
  public class SomeTests
  {
    [Test]
    [Ignore("We're skipping this one for now.")]
    public void TestOne()
    {
      // Do something...
    }
  }
```

Demo

Extending the Simple Example of TDD and NUnit

- Adding setup and teardown routines
- Handling expected exceptions



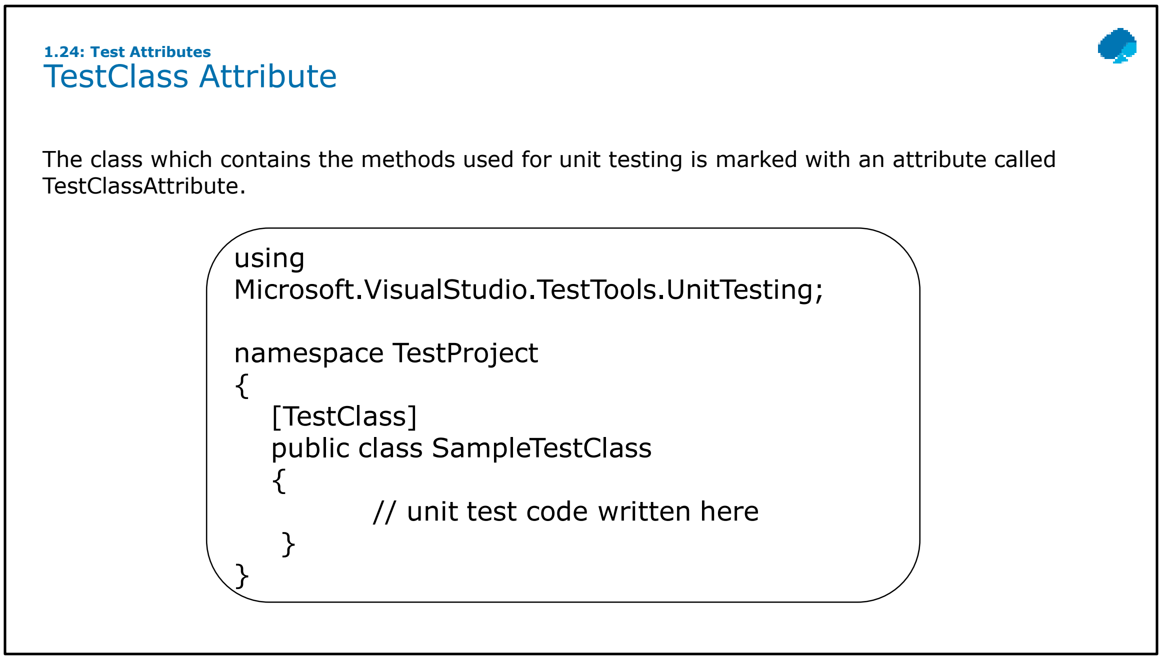
VS 2013 Test Framework

- Visual Studio 2013 provides an inbuilt framework for writing unit test cases.
- The framework includes a set of attributes, assertion classes and configuration files for manipulating the test environment.
- The test code is created by adding attributes to the classes and methods used for writing tests.
- The framework integrated with the IDE allows creation of test suites and addition of test cases to them with minimal effort.
- Class must reference Microsoft.VisualStudio.TestTools.UnitTesting

The Microsoft.VisualStudio.TestTools.UnitTesting namespace exposes a host of predefined attributes and classes. For example, the Assert class contains methods which can be used to verify conditions in unit testing.
An Assertion is a Boolean expression that describes what must be true when some action has been executed.
If an assertion method fails, the method call does not return, and an error is reported. Hence, if a test contains multiple assertions, any assertions following the one that has failed will not be executed. It’s usually best to try for one assertion per test.
A test case is the smallest unit required to test a functionality. We need to identify initial conditions, test actions and expected output for each test case.
A test suite is a group of related test cases.

Creating Tests Using Visual Studio 2013

- Create a separate class library or simply create a new class in the same project which contains the code to be tested.
- Create the tests by using predefined attributes.
- Write the test case code .
- Use appropriate Asserts.



1.24: Test Attributes

TestClass Attribute

The class which contains the methods used for unit testing is marked with an attribute called TestClassAttribute.

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

namespace TestProject
{
    [TestClass]
    public class SampleTestClass
    {
                // unit test code written here
     }
}
```

Add a reference to Microsoft.VisualStudio.TestTools.UnitTesting namespace to use the above mentioned attribute.

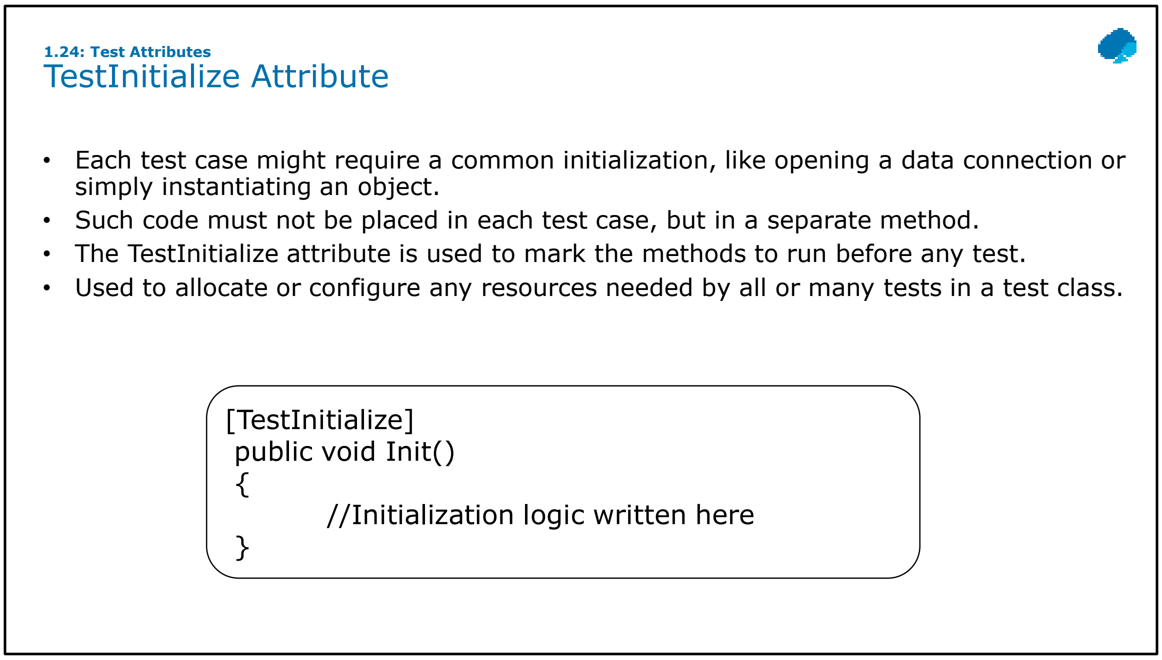
The class which contains the above attribute is called a Test Class , and will contain the unit test code.

Test Classes must be public, with a public parameter less constructor.

TestMethod Attribute

- The TestMethodAttribute is used to mark a method as a test method.
- A method marked with this attribute must be public, void and without parameters.

```
[TestClass]
public class SampleTestClass
{
    [TestMethod]
    public void TestOne()
    {
        //Unit Test logic written here
    }
}
```



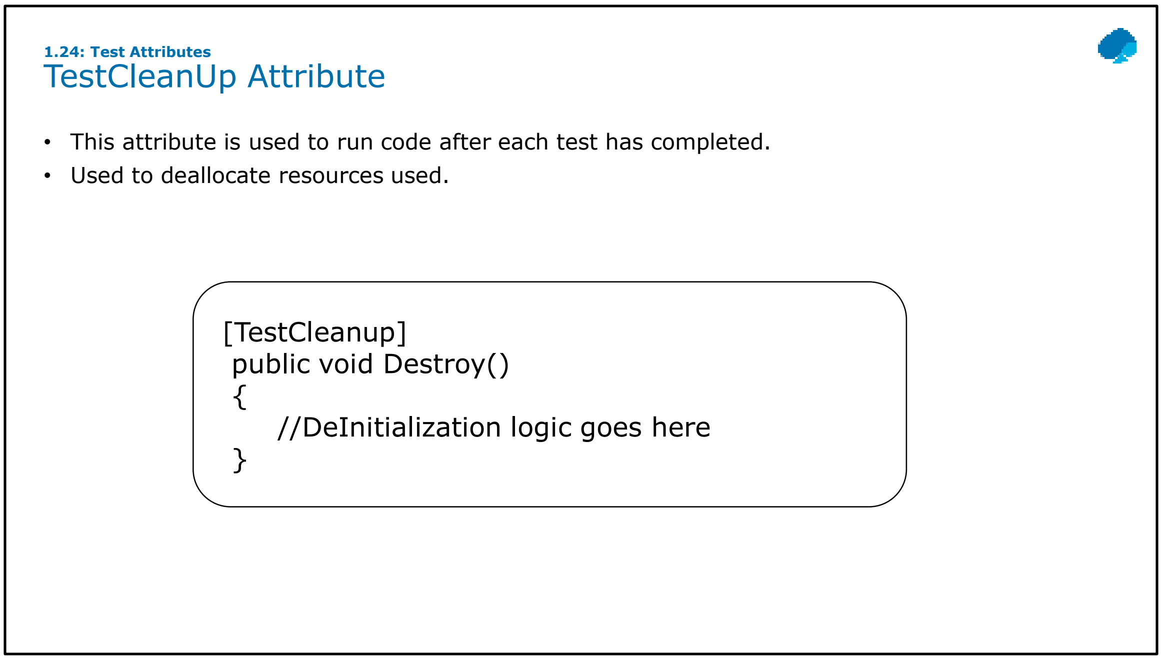
1.24: Test Attributes

TestInitialize Attribute

- Each test case might require a common initialization, like opening a data connection or simply instantiating an object.
- Such code must not be placed in each test case, but in a separate method.
- The TestInitialize attribute is used to mark the methods to run before any test.
- Used to allocate or configure any resources needed by all or many tests in a test class.

```
[TestInitialize]
 public void Init()
 {
           //Initialization logic written here
 }
```

Any method marked with this attribute will run once for every iteration in the test. If you need to do initialization operations once, that apply to the entire test, use the ClassInitializeAttribute.



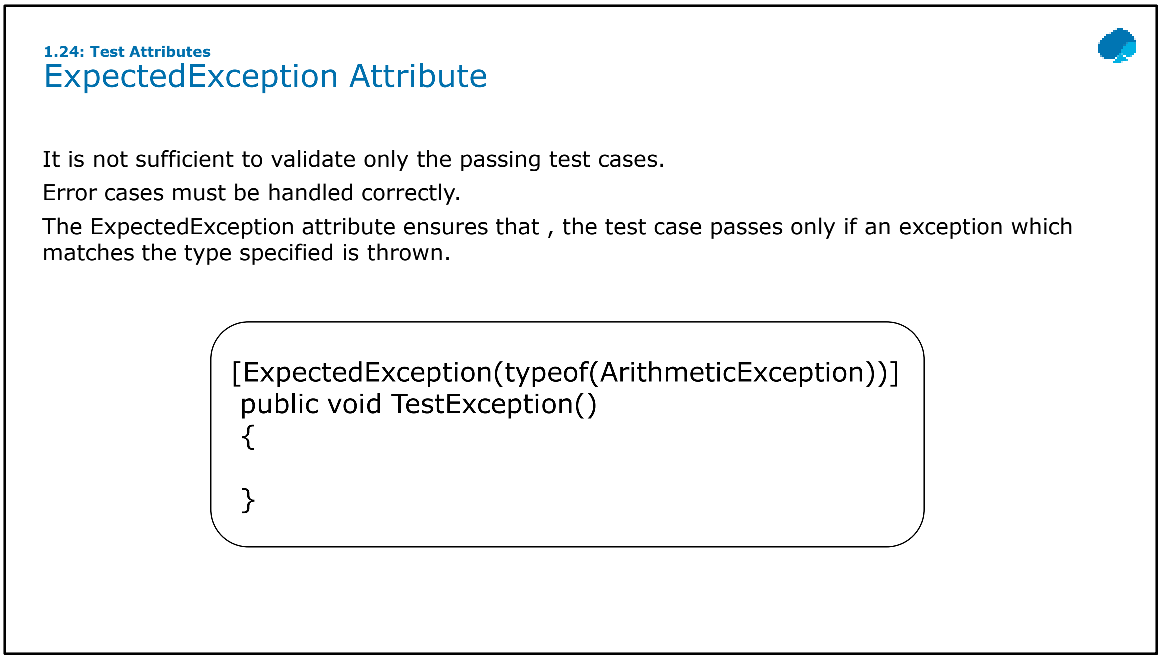
1.24: Test Attributes

TestCleanUp Attribute

- This attribute is used to run code after each test has completed.
- Used to deallocate resources used.

```
[TestCleanup]
 public void Destroy()
 {
      //DeInitialization logic goes here
 }
```

Use TestCleanup to run code after each test has run. If you need to perform clean up operations once, that apply to the entire test, use the ClassCleanupAttribute.



1.24: Test Attributes

ExpectedException Attribute

It is not sufficient to validate only the passing test cases.

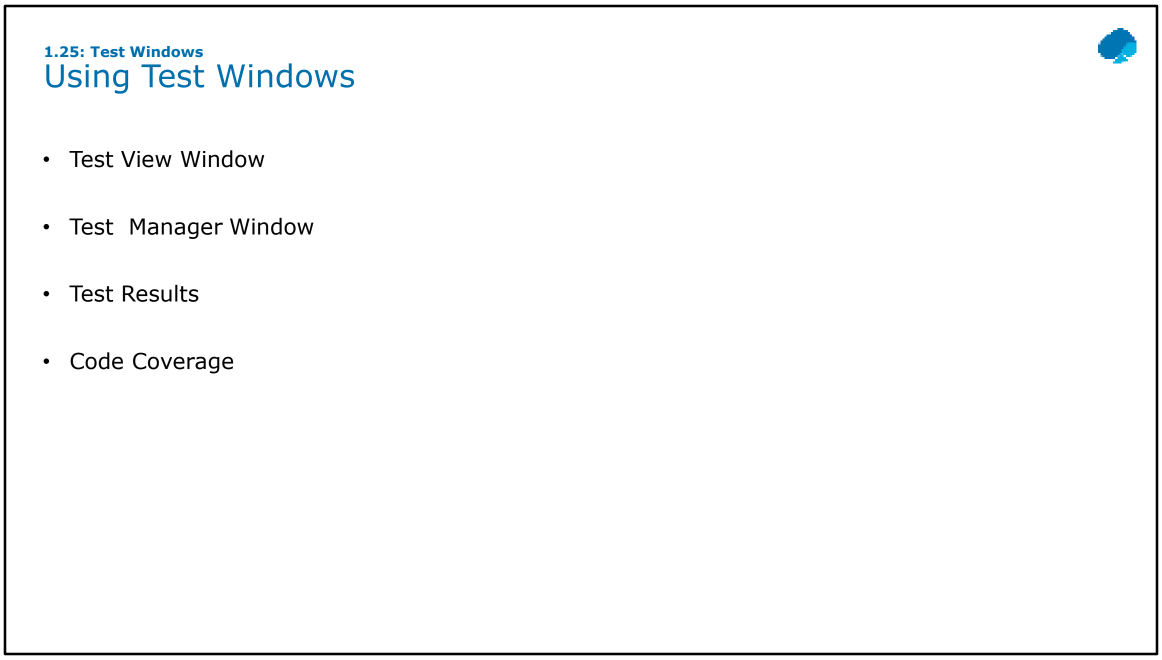
Error cases must be handled correctly.

The ExpectedException attribute ensures that , the test case passes only if an exception which matches the type specified is thrown.

```
[ExpectedException(typeof(ArithmeticException))]
 public void TestException()
 {

 }
```

You can also overload the attribute constructor with a string, to specify the error message which must be reported.



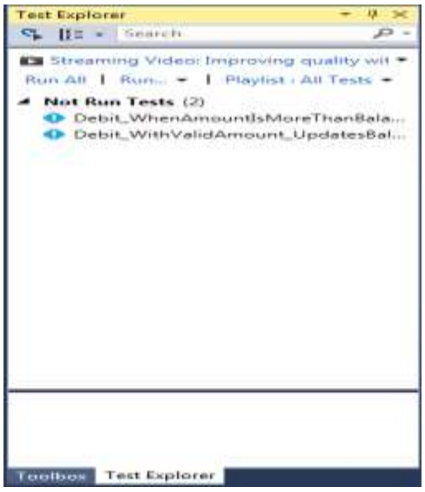
1.25: Test Windows

Using Test Windows

- Test View Window
- Test Manager Window
- Test Results
- Code Coverage

a. The Test View Window: this window shows a list of all loaded tests. You can use this window for running , grouping or filtering tests. The window can be opened through the Test -> Windows Menu Item (at the top of the IDE).
b. The Test Manager Window: Similar to Test View, but can organize tests. Used to show the list of tests, the list of loaded tests, and the list of unorganized tests.
c. The Test Results Window: used to view test results. You can also re run a test, pause/ stop a test. The window exposes functionality to publish or import test reports using TFS (if connected).
d. The Code Coverage Window: shows the percentage of code covered by the tests. Uses coloring to indicate the code covered, partially covered or not covered by the test case.

Using Test Windows


Test Explorer
Search
Streaming Video: Improving quality wit
Run All | Run... | Playlist : All Tests
Toolbox
Test Explorer

Not Run Tests (2)
Debit_WhenAmountIsMoreThanBala...
Debit_WithValidAmount_UpdatesBal...

Demo

Creating a test project in VS 2013

1.23: Programming Sequence

The Typical Programming Sequence

- Write a test
- Run the test. It fails to compile because the code you're trying to test doesn't even exist yet!
- Write a bare-bones stub to make the test compile
- Run the test. It should fail. (If it doesn't, then the test wasn't very good.)
- Implement the code to make the test pass
- Run the test. It should pass. (If it doesn't, back up one step and try again.)
- Start over with a new test!

Code Coverage with NCover

- Code coverage analysis provides valuable insight about the code
- Unit tests work best when 100 percent code overage of the methods in the project is achieved
- NCover works by monitoring the applications execution using the CLR Profiler
- NCover provides a powerful combination with NUnit

Demo on Using NCover

Using NCover

Summary

- NUnit provides a powerful framework for unit testing
- Developers assisted in implementing repeatable, best-practice processes

People matter, results count.

About Capgemini

With more than 190,000 people, Capgemini is present in over 40 countries and celebrates its 50th Anniversary year in 2017. A global leader in consulting, technology and outsourcing services, the Group reported 2016 global revenues of EUR 12.5 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.

Learn more about us at

www.capgemini.com