

Test Driven Development

Bhanu Korremula
Sr. Programmer Analyst,
Business Systems Division,
Richland County

The only thing constant is change – Heraclitus

Requirements are constantly changing

Software is constantly evolving

To handle constant change go Agile with Test Driven
Development

Topics that Will be Covered

- ▶ Test Driven Development
- ▶ Unit Testing Frameworks
- ▶ Testing Practices
- ▶ Visual Studio 2013 Unit Testing Framework
- ▶ Creating Unit Tests from scratch
- ▶ Test Explorer
- ▶ Data Driven Test
- ▶ Code Coverage
- ▶ Visual Studio 2013 Fakes Testing Framework

What is Test Driven Development?

- ▶ Wikipedia defines Test Driven Development as:
“Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.”
- ▶ Kent Beck is credited with having developed the technique

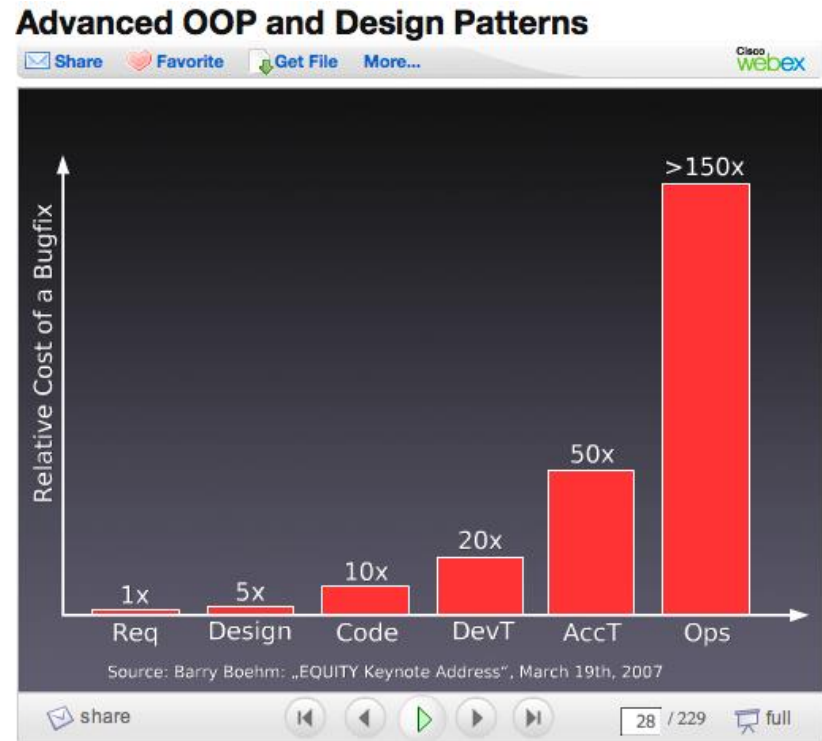
Common Reasons to Ignore Developer Testing

- ▶ Developers may think “I am the one who creates software why should I be spending my time on testing”
- ▶ Managers may think “I don’t want to pay the developers to test its a tester’s job”



Costs of Inadequate Testing

- ▶ Production bugs are expensive to fix
- ▶ The sooner a bug is found and fixed the cost of making and maintaining the software will go down





- ▶ People are creatures of habit some people resist change
- ▶ Don't think of Test Driven Development as whole new way of developing software
- ▶ Think of it as a way to capture various documented and undocumented tests which developers create during software development

Test Driven Development Developer Lingo

- ▶ Code that validates other code
- ▶ Small chunks of “is it working”
- ▶ By small chunks it means write unit tests
- ▶ “Never write a single line of code until you have written a failing automated test” – Kent Beck

Unit Testing Frameworks

Visual Studio Testing Framework
MS Test



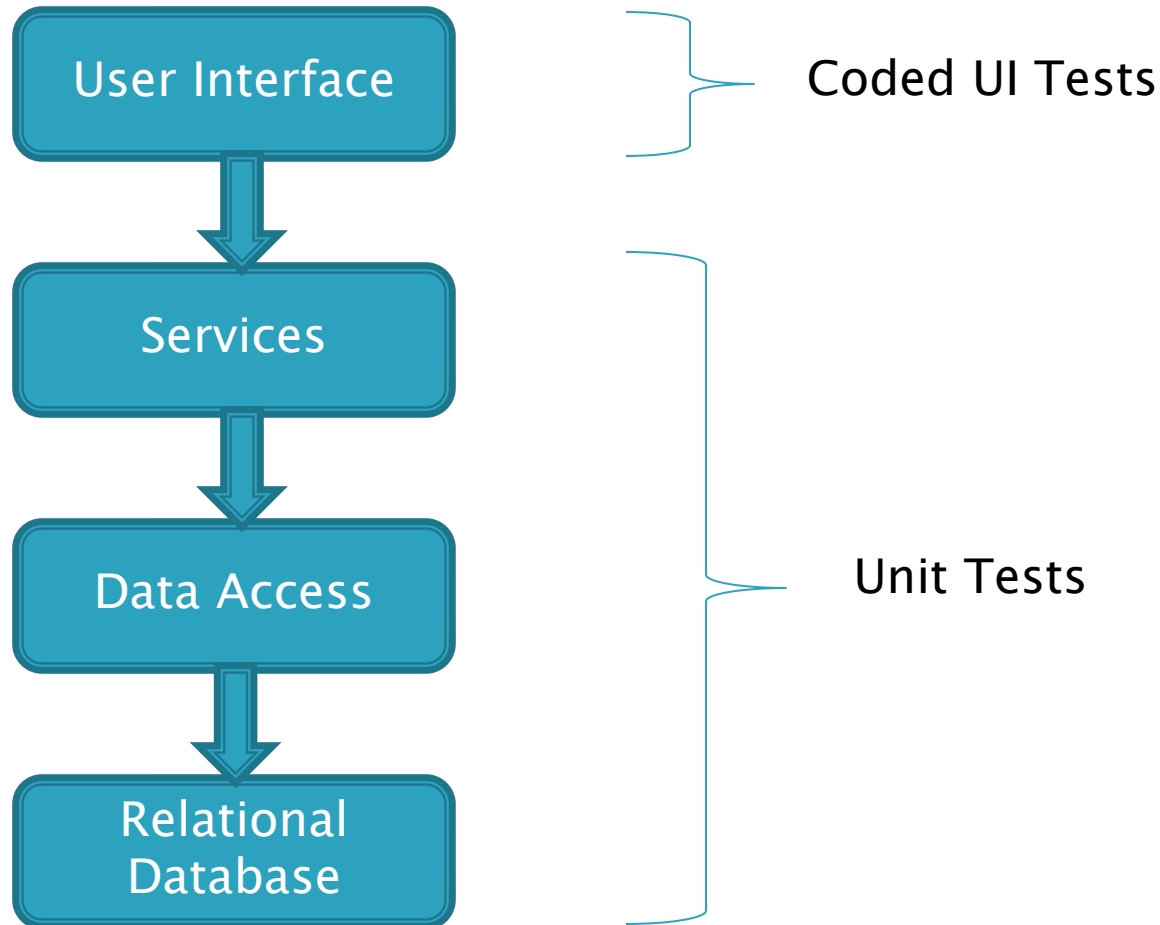
NUnit

XUnit

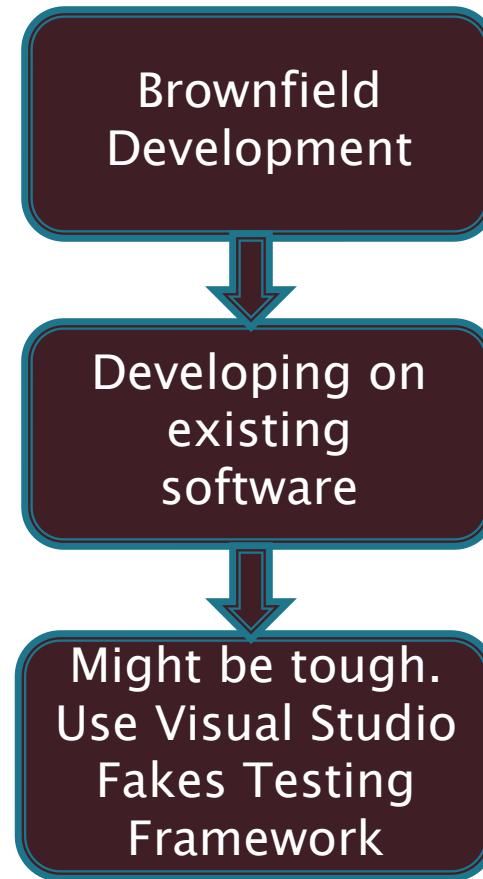
Visual Studio Testing Framework

- ▶ The Visual Studio Unit Testing Framework describes Microsoft's suite of unit testing tools as integrated into some versions of Visual Studio 2005 and later.
- ▶ The Unit Testing Framework is defined in `Microsoft.VisualStudio.TestTools.UnitTesting` in Visual Studio 2013.
- ▶ Unit tests created with the Unit Testing Framework can be executed in Visual Studio or, using `MSTest.exe`, from a command line

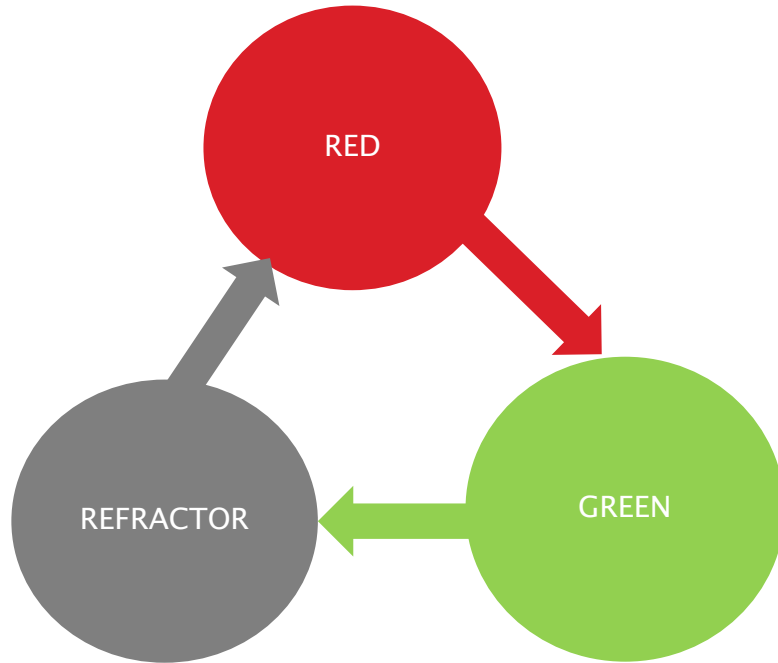
N Tier App and Testing



Testing Types for different software models



Testing Practices



- ▶ Red green refactor cycle
- ▶ Developer should create a failing test and make the bug pass
- ▶ Developers have to test classes, methods or functions at the API level
- ▶ Developers should strive to test code as it is being built

Main Elements of Visual Studio Testing Framework

Test Class

Test classes are declared as such by decorating a class with the `TestClass` attribute. The attribute is used to identify classes that contain test methods.

Test Method

Test methods are declared as such by decorating a unit test method with the `TestMethod` attribute. The attribute is used to identify methods that contain unit test code.

Assertions

An assertion is a piece of code that is run to test a condition or behavior against an expected result.

Initialization and Cleanup methods

Initialization and cleanup methods are used to prepare unit tests before running and cleaning up after unit tests have been executed.

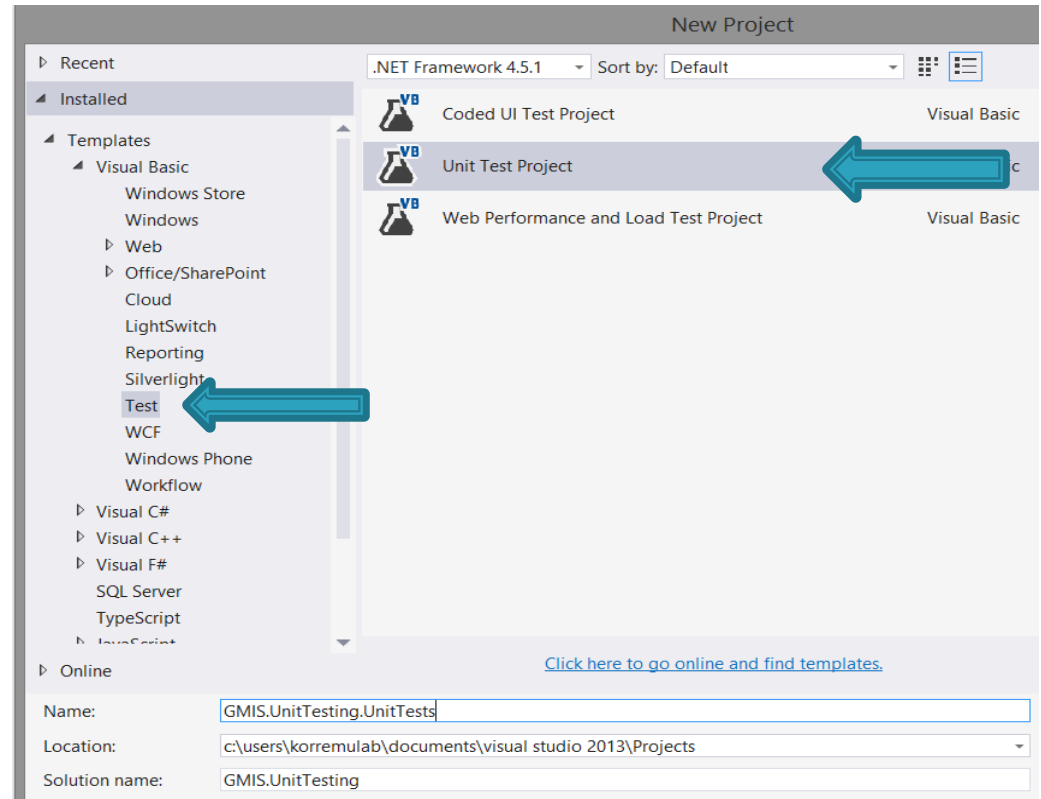
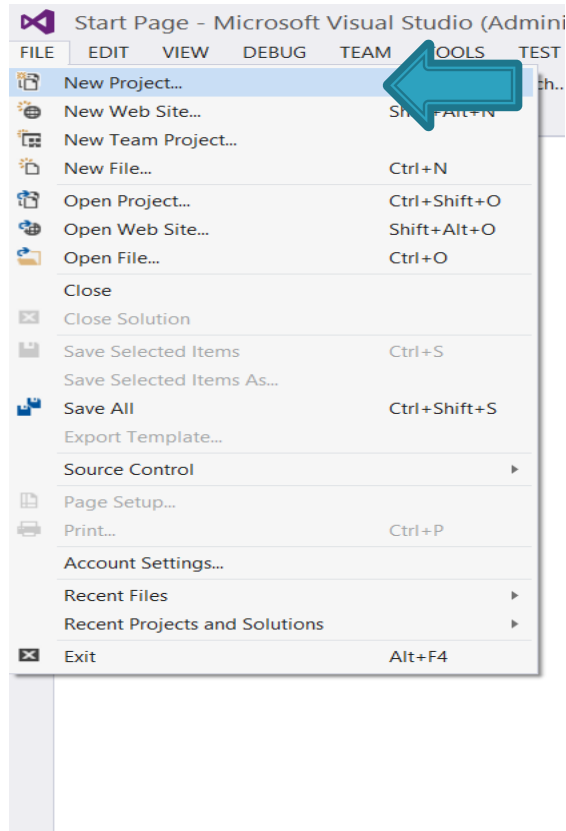
Asserts

- ▶ Verifies conditions in unit tests using true/false propositions
- ▶ Normally used to compare expected and actual value

├ Methods

| | Name | Description |
|----|---|--|
| ➤S | <code>AreEqual(Object, Object)</code> | Verifies that two specified objects are equal. The assertion fails if the objects are not equal. |
| ➤S | <code>AreEqual(Double, Double, Double)</code> | Verifies that two specified doubles are equal, or within the specified accuracy of each other. The assertion fails if they are not within the specified accuracy of each other. |
| ➤S | <code>AreEqual(Object, Object, String)</code> | Verifies that two specified objects are equal. The assertion fails if the objects are not equal. Displays a message if the assertion fails. |
| ➤S | <code>AreEqual(Single, Single, Single)</code> | Verifies that two specified singles are equal, or within the specified accuracy of each other. The assertion fails if they are not within the specified accuracy of each other. |
| ➤S | <code>AreEqual(String, String, Boolean)</code> | Verifies that two specified strings are equal, ignoring case or not as specified. The assertion fails if they are not equal. |
| ➤S | <code>AreEqual(Double, Double, Double, String)</code> | Verifies that two specified doubles are equal, or within the specified accuracy of each other. The assertion fails if they are not within the specified accuracy of each other. Displays a message if the assertion fails. |
| ➤S | <code>AreEqual(Object, Object, String, Object())</code> | Verifies that two specified objects are equal. The assertion fails if the objects are not equal. Displays a message if the assertion fails, and applies the specified formatting to it. |
| ➤S | <code>AreEqual(Single, Single, Single, String)</code> | Verifies that two specified singles are equal, or within the specified accuracy of each other. The assertion fails if they are not within the specified accuracy of each other. Displays a message if the assertion fails. |
| ➤S | <code>AreEqual(String, String, Boolean, CultureInfo)</code> | Verifies that two specified strings are equal, ignoring case or not as specified, and using the culture info specified. The assertion fails if they are not equal. |
| ➤S | <code>AreEqual(String, String, Boolean, String)</code> | Verifies that two specified strings are equal, ignoring case or not as specified. The assertion fails if they are not equal. Displays a message if the assertion fails. |
| ➤S | <code>AreEqual(Double, Double, Double, String, Object())</code> | Verifies that two specified doubles are equal, or within the specified accuracy of each other. The assertion fails if they are not within the specified accuracy of each other. Displays a message if the assertion fails, and applies the specified formatting to it. |
| ➤S | <code>AreEqual(Single, Single, Single, String, Object())</code> | Verifies that two specified singles are equal, or within the specified accuracy of each other. The assertion fails if they are not within the specified accuracy of each other. Displays a message if the assertion fails, and applies the specified formatting to it. |
| ➤S | <code>AreEqual(String, String, Boolean, CultureInfo, String)</code> | Verifies that two specified strings are equal, ignoring case or not as specified, and using the culture info specified. The assertion fails if they are not equal. Displays a message if the assertion fails. |
| ➤S | <code>AreEqual(String, String, Boolean, String, Object())</code> | Verifies that two specified strings are equal, ignoring case or not as specified. The assertion fails if they are not equal. Displays a message if the assertion fails, and applies the specified formatting to it. |

Create a Unit Test Project



Go to New Project → Visual Basic → Test → Unit Test Project to create one

What Does Default Unit Test Template Contain?

```
Imports System.Text
Imports Microsoft.VisualStudio.TestTools.UnitTesting

0 references
<TestClass()> Public Class UnitTest1
    0 references
    <TestMethod()> Public Sub TestMethod1()
    End Sub
End Class
```

- ▶ The unit test project by default comes with test class and test method/function
- ▶ The project has two main attributes first one is `<TestClass()>` and second is `<TestMethod()>`
- ▶ `<TestClass()>` informs the .NET that class will have test functions
- ▶ `<TestMethod()>` informs .NET to execute the unit test function

Code The Test First

```
Imports System.Text
```

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
```

0 references

```
<TestClass()> Public Class CalculatorTestFixture
```

0 references

```
<TestMethod()> Public Sub AddTest()
```

```
    Dim calc As New Calculator
```

```
    'Dim calc As New Calculator
```

```
    Dim expected As Integer = 5
```

```
    Dim actual As Integer = 0
```

```
    actual = calc.Add(2, 3)
```

```
    Assert.AreEqual(expected, actual, "Did not get the right value from the add method")
```

```
End Sub
```

```
End Class
```

Build the Test Project

```
1 Imports System.Text
2 Imports Microsoft.VisualStudio.TestTools.UnitTesting
3
4 <TestClass()> Public Class CalculatorTestFixture
5
6     0 references
7     <TestMethod()> Public Sub AddTest()
8         Dim calc As New Calculator
9         'Dim calc As New Calculator
10        Dim expected As Integer = 5
11        Dim actual As Integer = 0
12        actual = calc.Add(2, 3)
13        Assert.AreEqual(expected, actual, "Did not get the right value from the add method")
14    End Sub
15 End Class
```

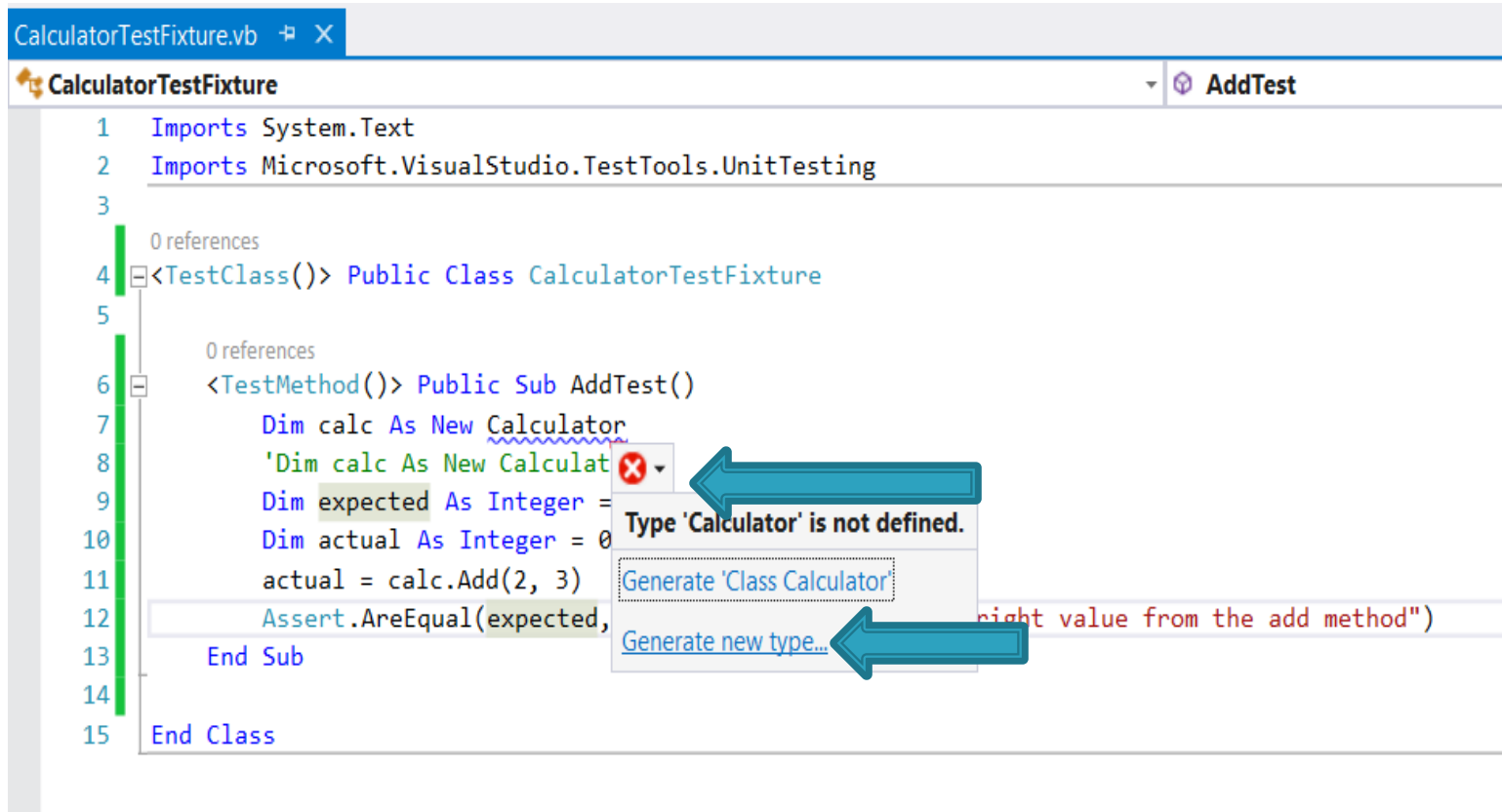
Error List

1 Error | 0 Warnings | 0 Messages

| Description |
|-------------------------------------|
| 1 Type 'Calculator' is not defined. |

After the build we will get a compilation of the errors

Generate The Class From Unit Test



The screenshot shows a Visual Studio code editor window for a file named `CalculatorTestFixture.vb`. The code is as follows:

```
1 Imports System.Text
2 Imports Microsoft.VisualStudio.TestTools.UnitTesting
3
4 <TestClass()> Public Class CalculatorTestFixture
5
6     0 references
7     <TestMethod()> Public Sub AddTest()
8         Dim calc As New Calculator
9         'Dim calc As New Calculat
10        Dim expected As Integer =
11        Dim actual As Integer = 0
12        actual = calc.Add(2, 3)
13        Assert.AreEqual(expected,
14                          right value from the add method")
15    End Sub
16 End Class
```

A red squiggly line under the word `Calculator` on line 8 indicates a compilation error. A context menu is open over this error, displaying the message "Type 'Calculator' is not defined." and two options: "Generate 'Class Calculator'" and "Generate new type...". Two blue arrows point to these options from the right side of the image.

Click on generate new type

Select Appropriate Project for Class Generation

```
1 Imports System.Text
2 Imports Microsoft.VisualStudio.TestTools.UnitTesting
3
4 <TestClass()> Public Class CalculatorTestFixture
5
6     0 references
7     <TestMethod()> Public Sub AddTest()
8         Dim calc As New Calculator
9         'Dim calc As New Calculator
10        Dim expected As Integer = 5
11        Dim actual As Integer = 0
12        actual = calc.Add(2, 3)
13        Assert.AreEqual(expected, actual,
14        End Sub
15 End Class
```

Generate New Type

Type Details:

Access: Public Kind: Class Name: Calculator

Location:

Project: GMIS.Business

File Name:

Create new file
Calculator.vb

Add to existing file
<Current File>

OK Cancel

Build Again

The screenshot shows a Visual Studio code editor with the following code:

```
1 Imports System.Text
2 Imports Microsoft.VisualStudio.TestTools.UnitTesting
3 Imports GMIS.Business
4
5 <TestClass(>> Public Class CalculatorTestFixture
6
7     0 references
8     <TestMethod(>> Public Sub AddTest()
9         Dim calc As New Calculator
10        'Dim calc As New Calculator
11        Dim expected As Integer = 5
12        Dim actual As Integer = 0
13        actual = calc.Add(2, 3)
14        Assert.AreEqual(expected, actual, "Did not get the right value from the add method")
15    End Sub
16 End Class
```

The Error List pane at the bottom shows the following error:

| Description |
|--|
| 1 Error 0 Warnings 0 Messages |
| 1 'Add' is not a member of 'GMIS.Business.Calculator'. |

Two blue arrows are present: one pointing to the `calc.Add(2, 3)` line in the code, and another pointing to the error message in the Error List pane.


Build breaks at the function which has not implemented yet

Generate The Function Code From Unit Test

```
CalculatorTestFixture.vb  [X]
(General)  (Declarations)
1 Imports System.Text
2 Imports Microsoft.VisualStudio.TestTools.UnitTesting
3 Imports GMIS.Business
4
5 <TestClass()> Public Class CalculatorTestFixture
6
7     <TestMethod()> Public Sub AddTest()
8         Dim calc As New Calculator
9         'Dim calc As New Calculator
10        Dim expected As Integer = 5
11        Dim actual As Integer = 0
12        actual = calc.Add(2, 3)
13        Assert.AreEqual(5, actual, "Did not get the right value from the add method")
14    End Sub
15
16 End Class
```

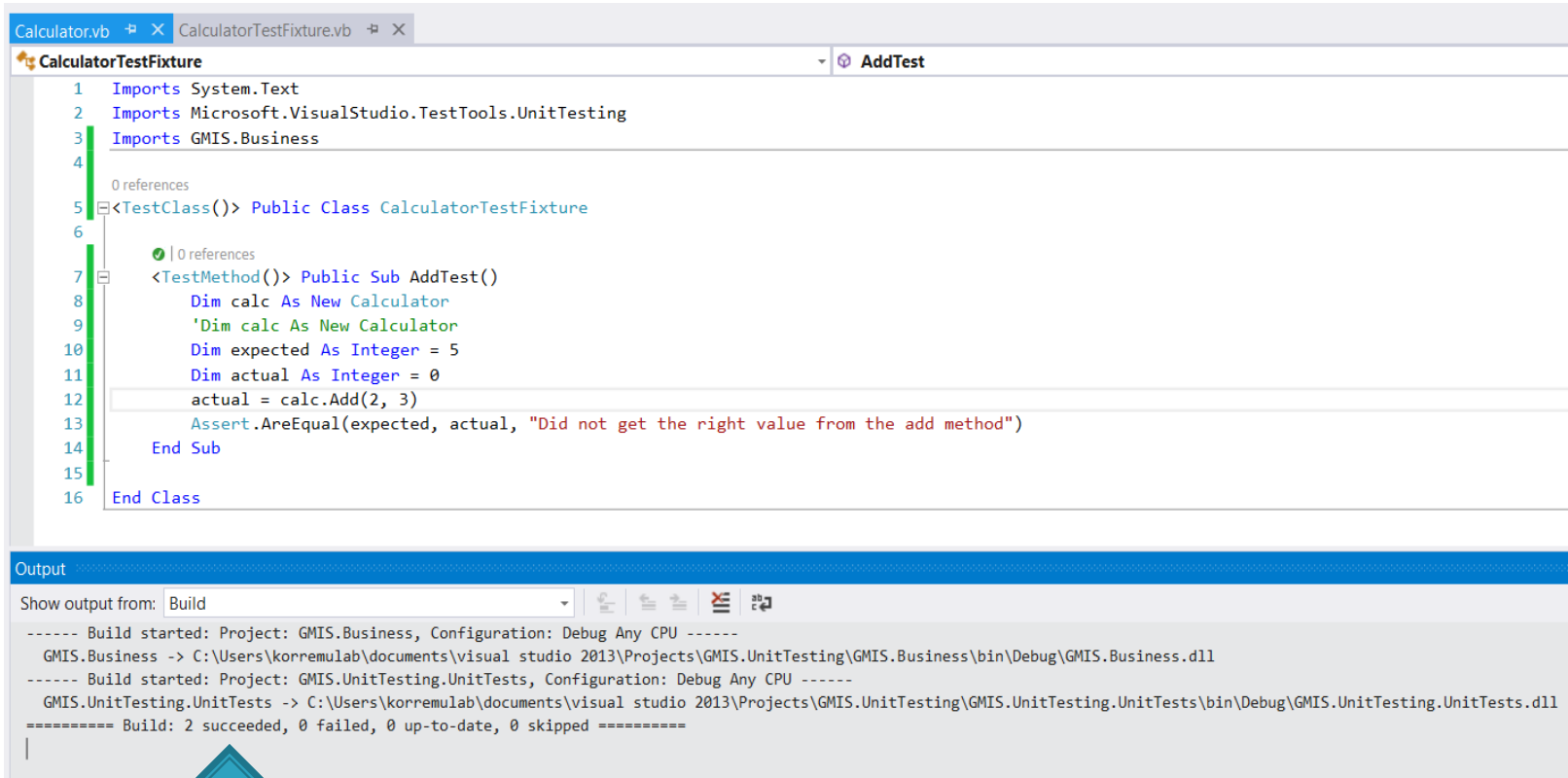
'Add' is not a member of 'GMIS.Business.Calculator'.

- Generate method stub for 'Add' in 'GMIS.Business.Calculator'
- Generate property stub for 'Add' in 'GMIS.Business.Calculator'



Generate function code and build

Rebuild the Solution



The screenshot shows the Visual Studio IDE with two tabs: Calculator.vb and CalculatorTestFixture.vb. The CalculatorTestFixture.vb tab is active, showing the following code:

```
1 Imports System.Text
2 Imports Microsoft.VisualStudio.TestTools.UnitTesting
3 Imports GMIS.Business
4
5 <TestClass()> Public Class CalculatorTestFixture
6
7     <TestMethod()> Public Sub AddTest()
8         Dim calc As New Calculator
9         'Dim calc As New Calculator
10        Dim expected As Integer = 5
11        Dim actual As Integer = 0
12        actual = calc.Add(2, 3)
13        Assert.AreEqual(expected, actual, "Did not get the right value from the add method")
14    End Sub
15
16 End Class
```

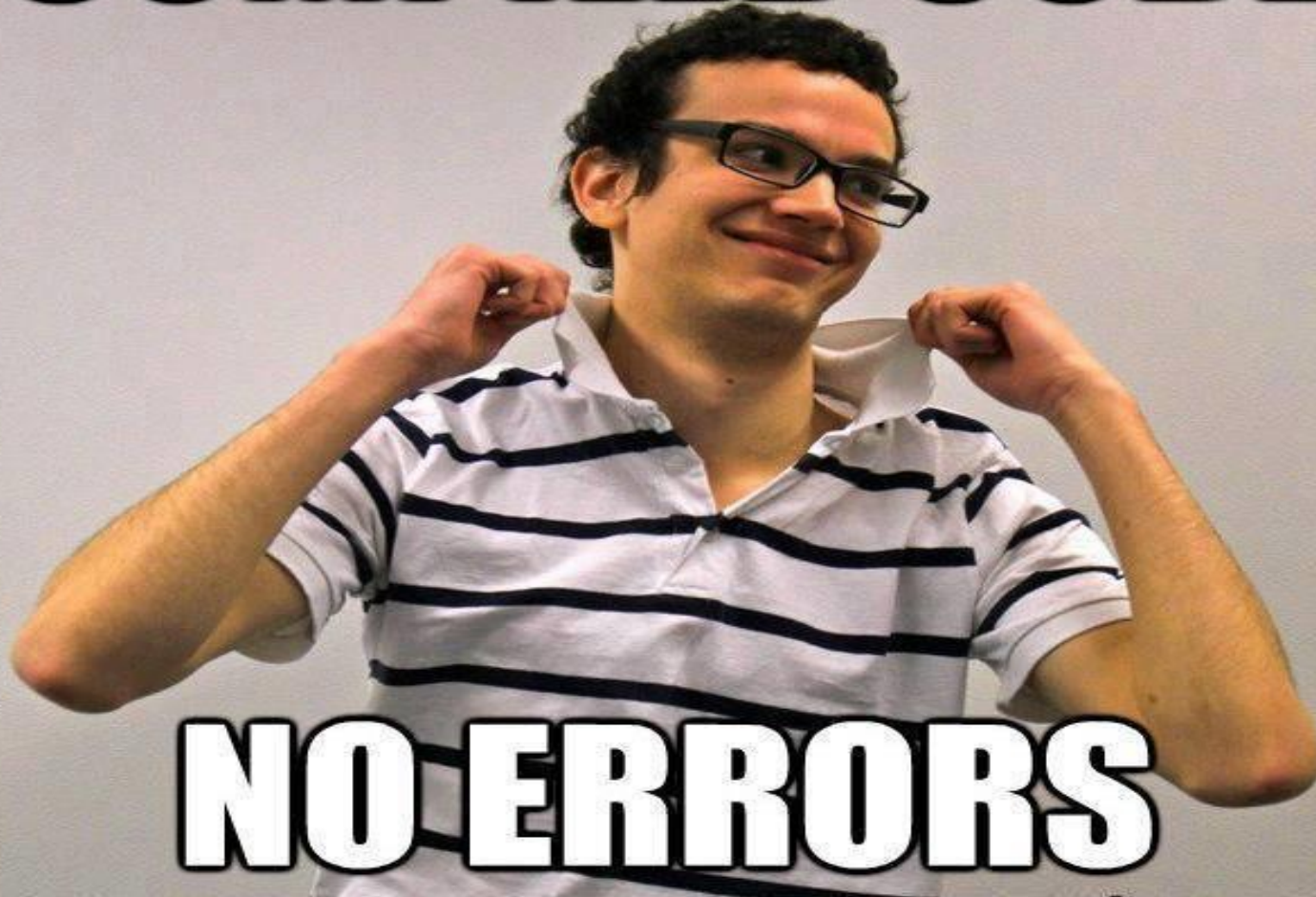
Below the code editor is the Output window, which shows the following build output:

```
Show output from: Build
----- Build started: Project: GMIS.Business, Configuration: Debug Any CPU -----
GMIS.Business -> C:\Users\korremulab\documents\visual studio 2013\Projects\GMIS.UnitTesting\GMIS.Business\bin\Debug\GMIS.Business.dll
----- Build started: Project: GMIS.UnitTesting.UnitTests, Configuration: Debug Any CPU -----
GMIS.UnitTesting.UnitTests -> C:\Users\korremulab\documents\visual studio 2013\Projects\GMIS.UnitTesting\GMIS.UnitTesting.UnitTests\bin\Debug\GMIS.UnitTesting.UnitTests.dll
===== Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```



Builds successfully

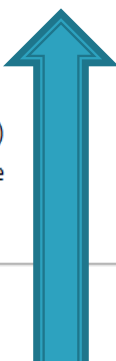
COMPILED CODE



NO ERRORS


Setup of the Test Class

```
<TestClass(>  
0 references  
Public Class CalculatorTestFixture  
  
    Private testContextInstance As TestContext  
  
    '''<summary>  
    '''Gets or sets the test context which provides  
    '''information about and functionality for the current test run.  
    '''</summary>  
0 references  
    Public Property TestContext() As TestContext  
        Get  
            Return testContextInstance  
        End Get  
        Set(ByVal value As TestContext)  
            testContextInstance = value  
        End Set  
    End Property
```



TextContext() used to come as default in test class

```
Private m_SystemUnderTest As Calculator  
  
    '''<summary>  
    '''Gets or sets the test context which provides  
    '''information about and functionality for the current test run.  
    '''</summary>  
4 references | 2/4 passing  
    Public ReadOnly Property SystemUnderTest() As Calculator  
        Get  
            If m_SystemUnderTest Is Nothing Then  
                m_SystemUnderTest = New Calculator()  
            End If  
            Return m_SystemUnderTest  
        End Get  
  
    End Property
```



Replace TextContext() with the common class declaration to reduce code

Setup of the Test Class

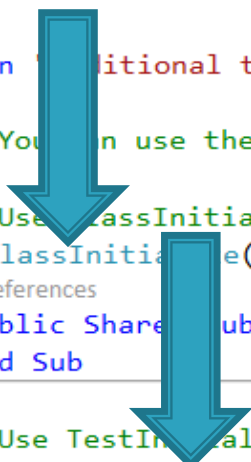
```
#Region "Additional test attributes"  
'  
' You can use the following additional attributes as you write your tests:  
'  
' Use <ClassInitialize> to run code before running the first test in the class  
<ClassInitialize(>  
0 references  
Public Shared Sub MyClassInitialize(ByVal testContext As TestContext)  
End Sub  


---

  
' Use <TestInitialize> to run code before running each test  
<TestInitialize(>  
0 references  
Public Sub MyTestInitialize()  
End Sub  


---


```

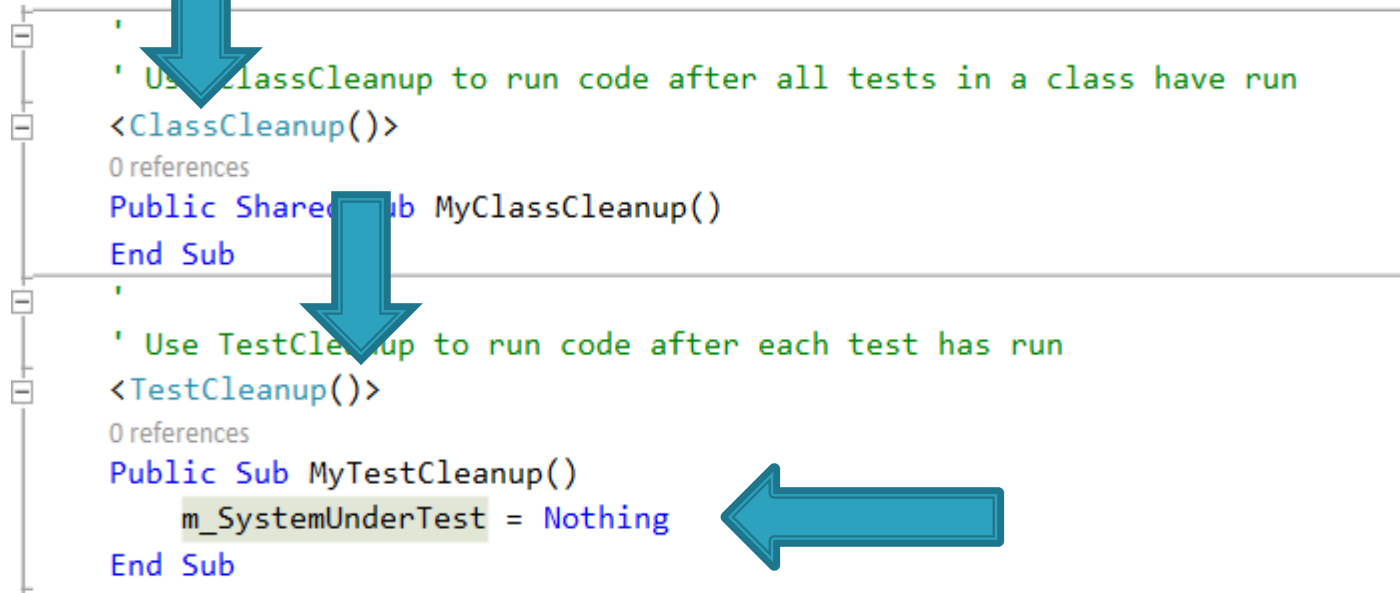


- ▶ Functions marked with the `<ClassInitialize(>` attribute will run before the test class
- ▶ Functions marked with `<TestInitialize(>` attribute will run before each test

Cleanup/Teardown of the Test Class

```
' Use <ClassCleanup> to run code after all tests in a class have run
<ClassCleanup()>
0 references
Public Shared Sub MyClassCleanup()
End Sub

' Use <TestCleanup> to run code after each test has run
<TestCleanup()>
0 references
Public Sub MyTestCleanup()
    m_SystemUnderTest = Nothing
End Sub
```

The diagram shows two code snippets. The first snippet is for a class cleanup method, marked with <ClassCleanup()>. A blue arrow points from the title 'Cleanup/Teardown of the Test Class' down to this snippet. The second snippet is for a test cleanup method, marked with <TestCleanup()>. A blue arrow points from the title down to this snippet. A third blue arrow points from the right towards the assignment statement 'm_SystemUnderTest = Nothing' in the second snippet.

- ▶ Functions marked with the <ClassCleanup()> attribute will run after the test class
- ▶ Functions marked with <TestCleanup()> attribute will run after each test

Features of Test Explorer

Run unit tests

Debug tests

Identify long running tests

Group tests into categories

Code coverage

The screenshot shows the Test Explorer window with the following features highlighted by callouts:

- Choose Home to see a summary of results**: Points to the Home icon in the toolbar.
- Choose Run All to build and run tests**: Points to the Run All button in the toolbar.
- Bar indicates test progress and results**: Points to the red progress bar at the top of the window.
- Show a full list**: Points to the Show All link under Failed Tests.
- Select a result to show more detail**: Points to the SignatureTest result in the list.
- Links open the code**: Points to the source link (UnitTest1.cs line 14) in the error message.

The Test Explorer window displays the following content:

TEST EXPLORER

Home ▾

Run All | Run... ▾

Failed Tests (1 of 1)

Show All

SignatureTest 426 ms

Passed Tests (3 of 3)

Show All

- QuickNonZero < 1 ms
- SqrtValueRange < 1 ms
- RootTestNegativeInput < 1 ms

SignatureTest

Test Failed - SignatureTest

Message: Test method Fabrikam.Math.UnitTest1.UnitTest1.SignatureTest threw exception: System.ArgumentOutOfRangeException: Specified argument was out of the range of valide values.

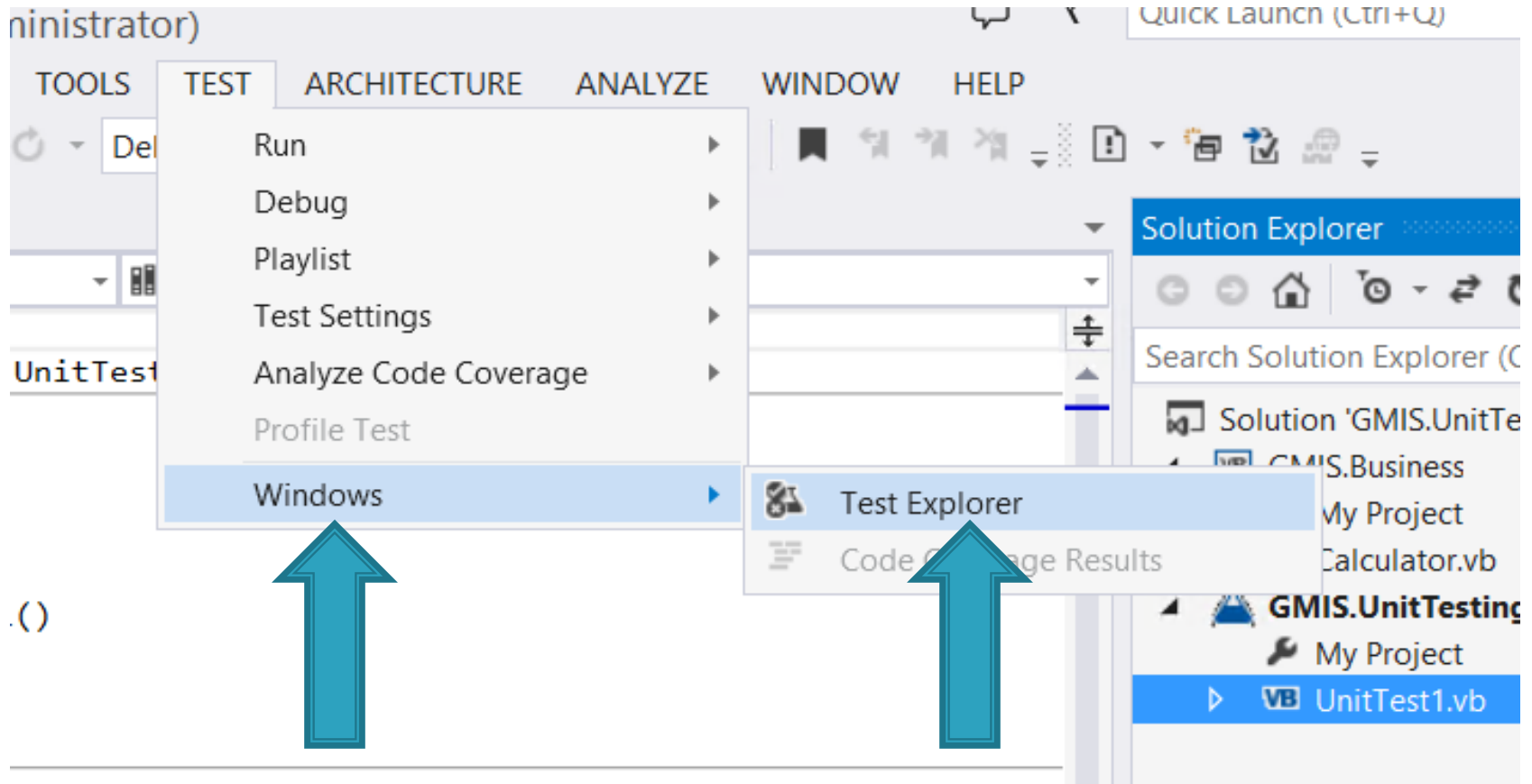
Elapsed time: 426 ms

Source: [UnitTest1.cs line 14](#)

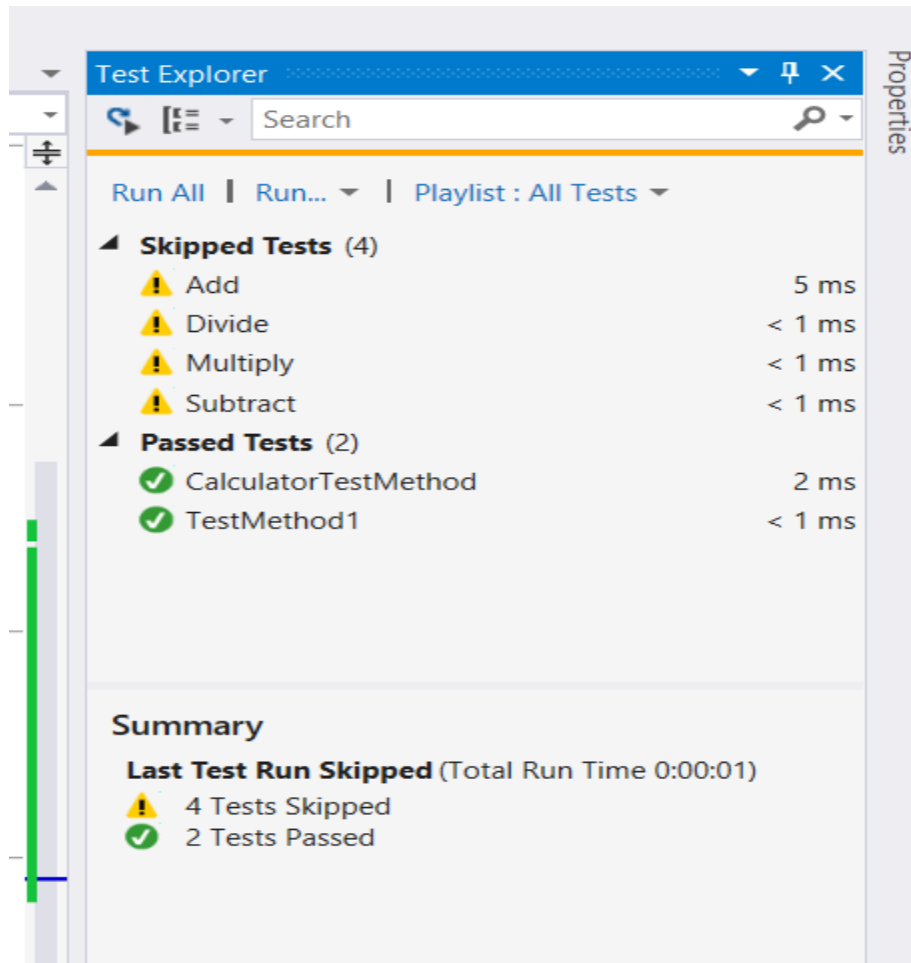
Stack Trace:

- LocalMath.SquareRoot(Double x)
- UnitTest1.SignatureTest()

How to Get to Test Explorer?

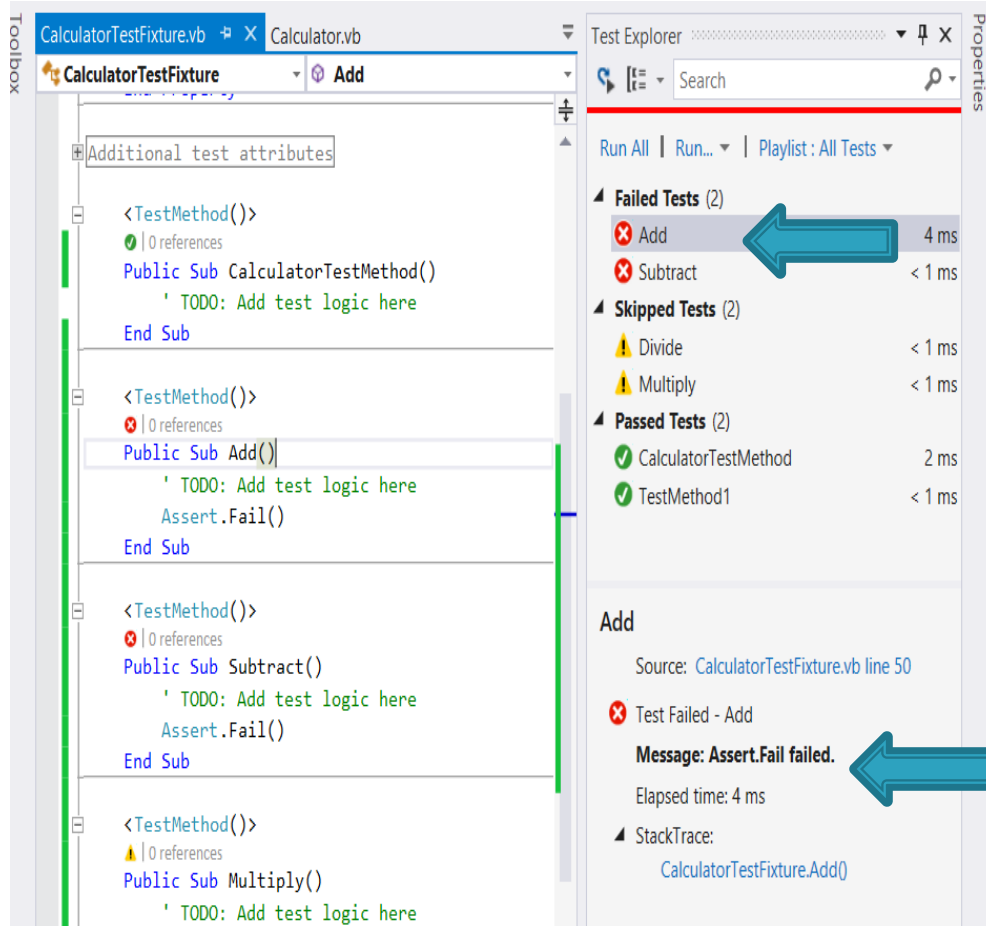


Test Explorer Features



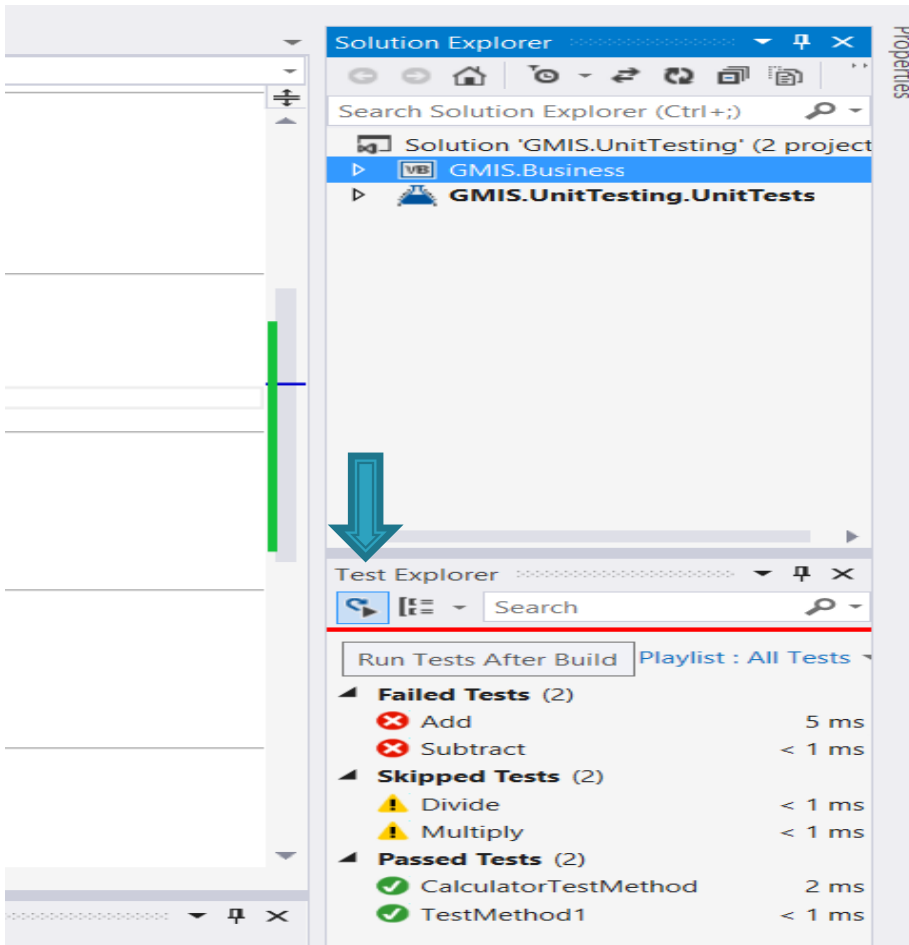
- ▶ As new tests are written they are automatically added to the test explorer
- ▶ All tests can be run with a single click

Test Explorer Identifying Failed Tests



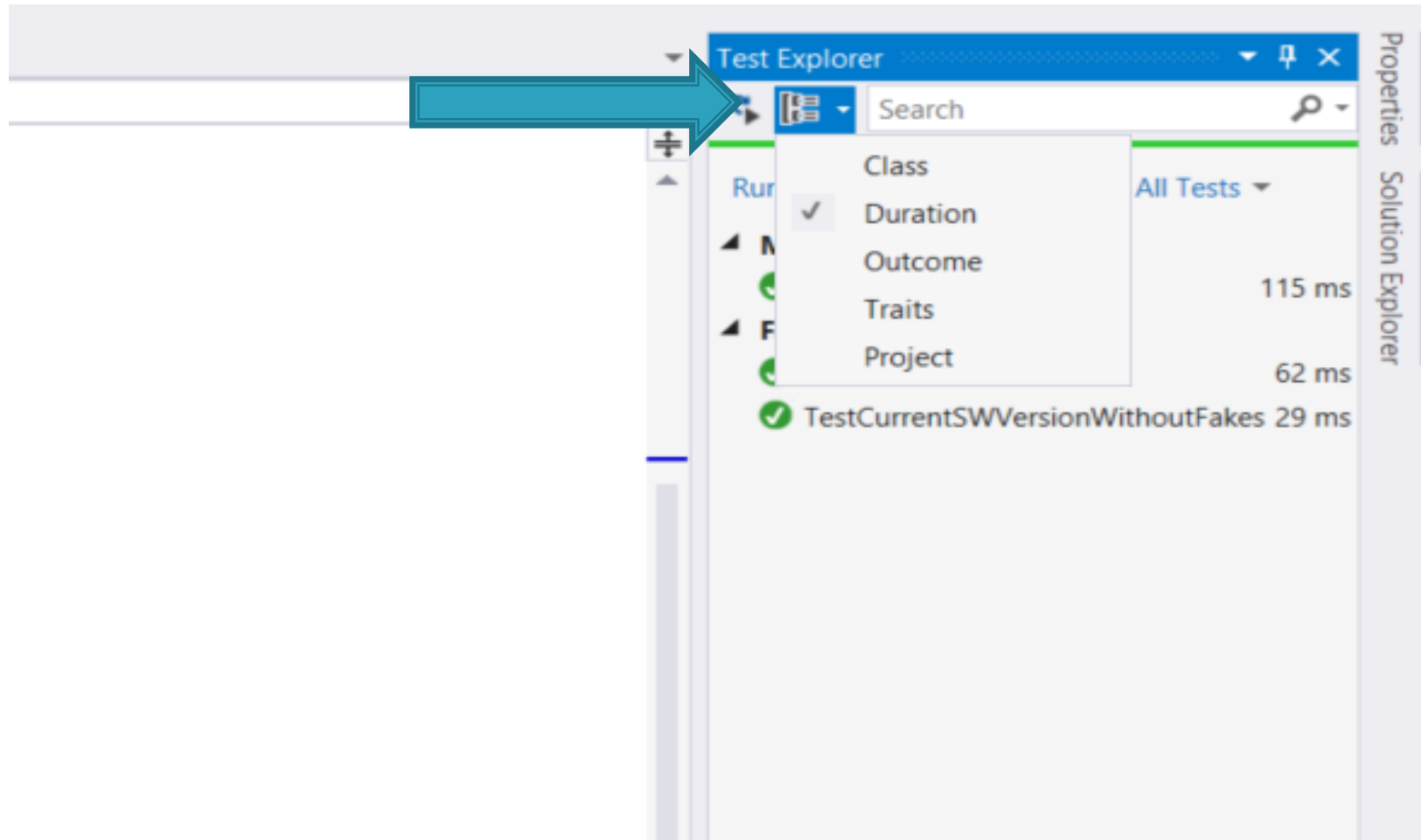
- ▶ Failing, passing and inconclusive tests can be easily identified at one glance
- ▶ With a single click one can go to the code of the failed test case and also look at the stack trace
- ▶ Developer can also run just one failed test

Running Tests After the Build

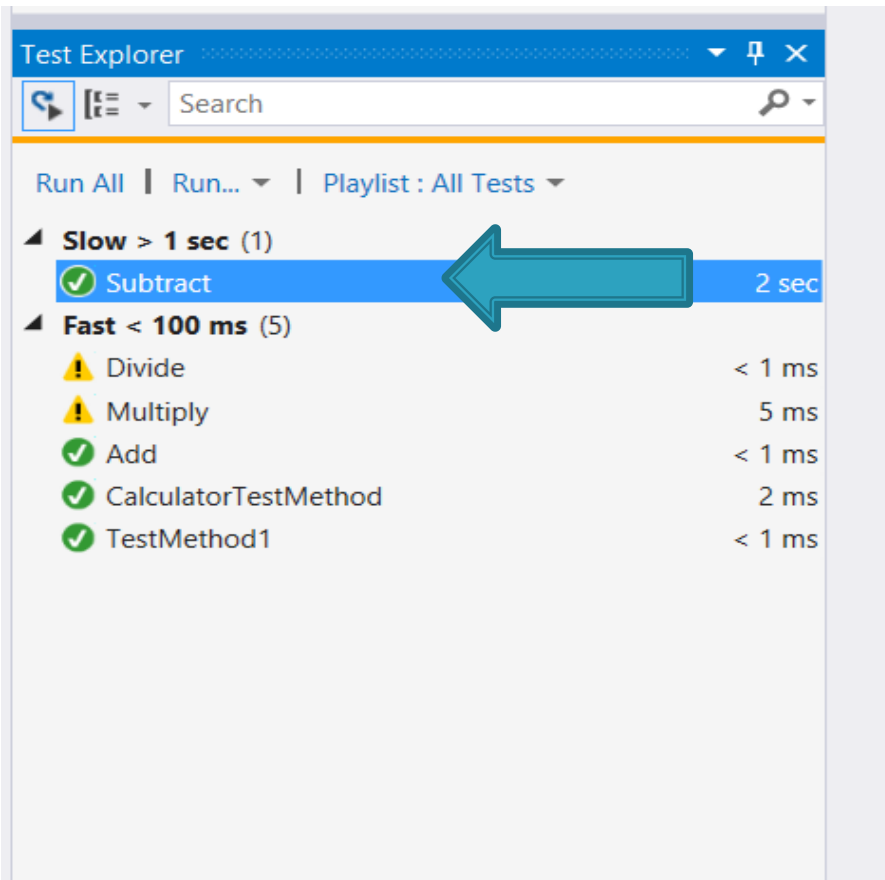


- ▶ Tests can also be set to run immediately after the build by clicking on the button shown in the picture.

Filtering By Various Test Categories

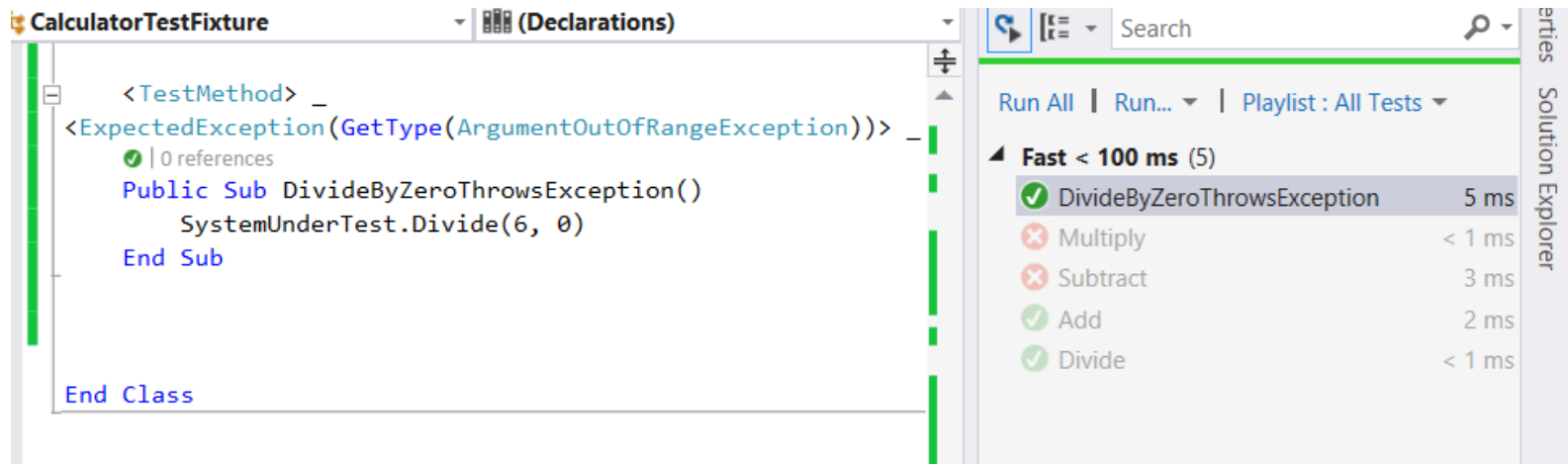


Identify Long Running Tests



- ▶ Some tests take more time
- ▶ Test explorer will help to identify those tests

Add Tests for Exceptional Cases




- ▶ Add a test which specially tests for an exception
- ▶ Added `<ExpectedException(GetType(ArgumentOutOfRangeException))>` attribute to the function and test expects an exception from code

Data Driven Tests

- ▶ Unit test method can be setup to retrieve the value from the data source
- ▶ The method is run successively for each row in the data source
- ▶ This makes it easy to test a variety of input by using a single method

```
[DataSource(@"Provider=Microsoft.SqlServerCe.Client.4.0; Data Source=C:\Data\MathsData.sdf;", "Numbers")]  
[TestMethod()]  
public void AddIntegers_FromDataSourceTest()  
{  
    var target = new Maths();  
  
    // Access the data  
    int x = Convert.ToInt32(TestContext.DataRow["FirstNumber"]);  
    int y = Convert.ToInt32(TestContext.DataRow["SecondNumber"]);  
    int expected = Convert.ToInt32(TestContext.DataRow["Sum"]);  
    int actual = target.IntegerMethod(x, y);  
    Assert.AreEqual(expected, actual,  
        "x:<{0}> y:<{1}>",  
        new object[] {x, y});  
}
```



Code Coverage

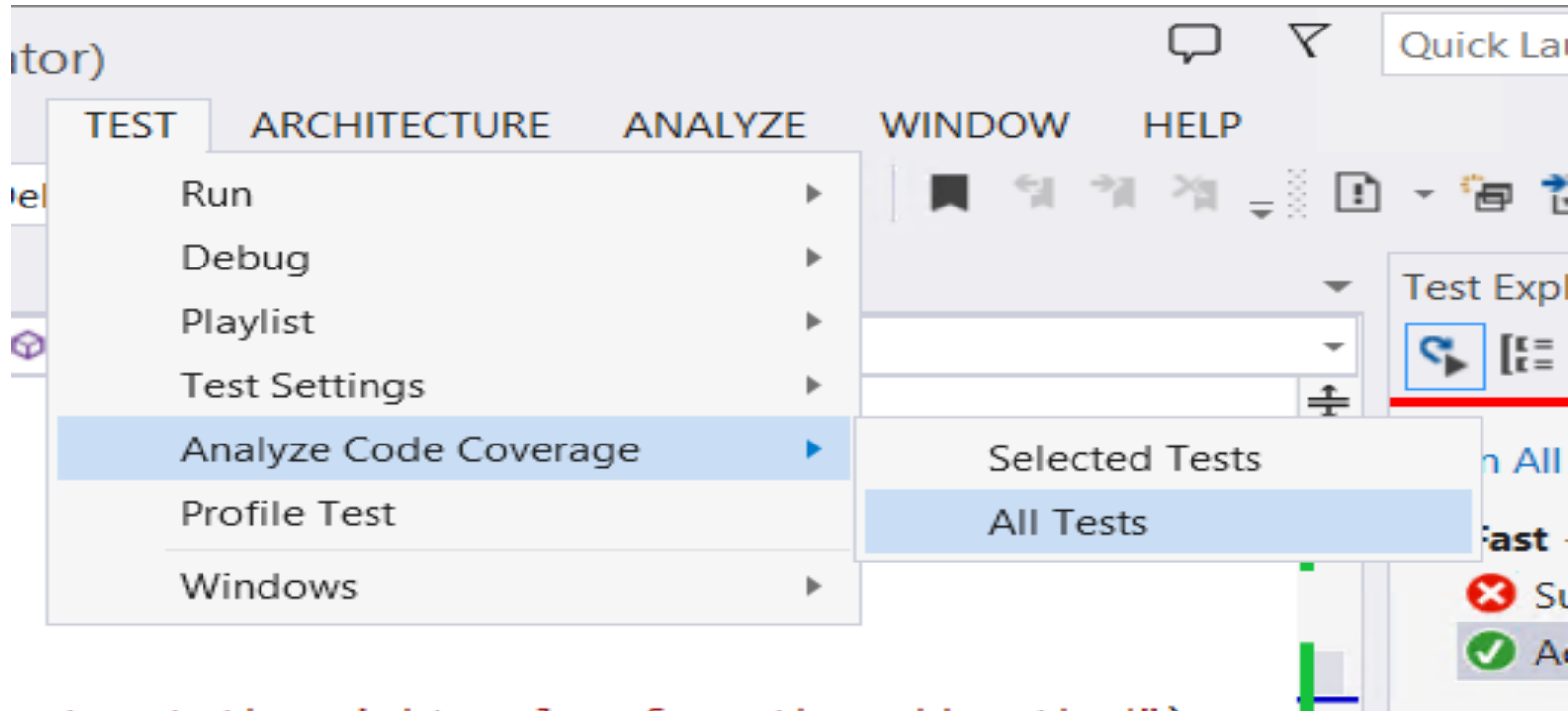
Will determine what percentage of project's code is covered by unit tests

Tests should cover a large portion of the code

Good to have more than 80 % of the code covered by the tests

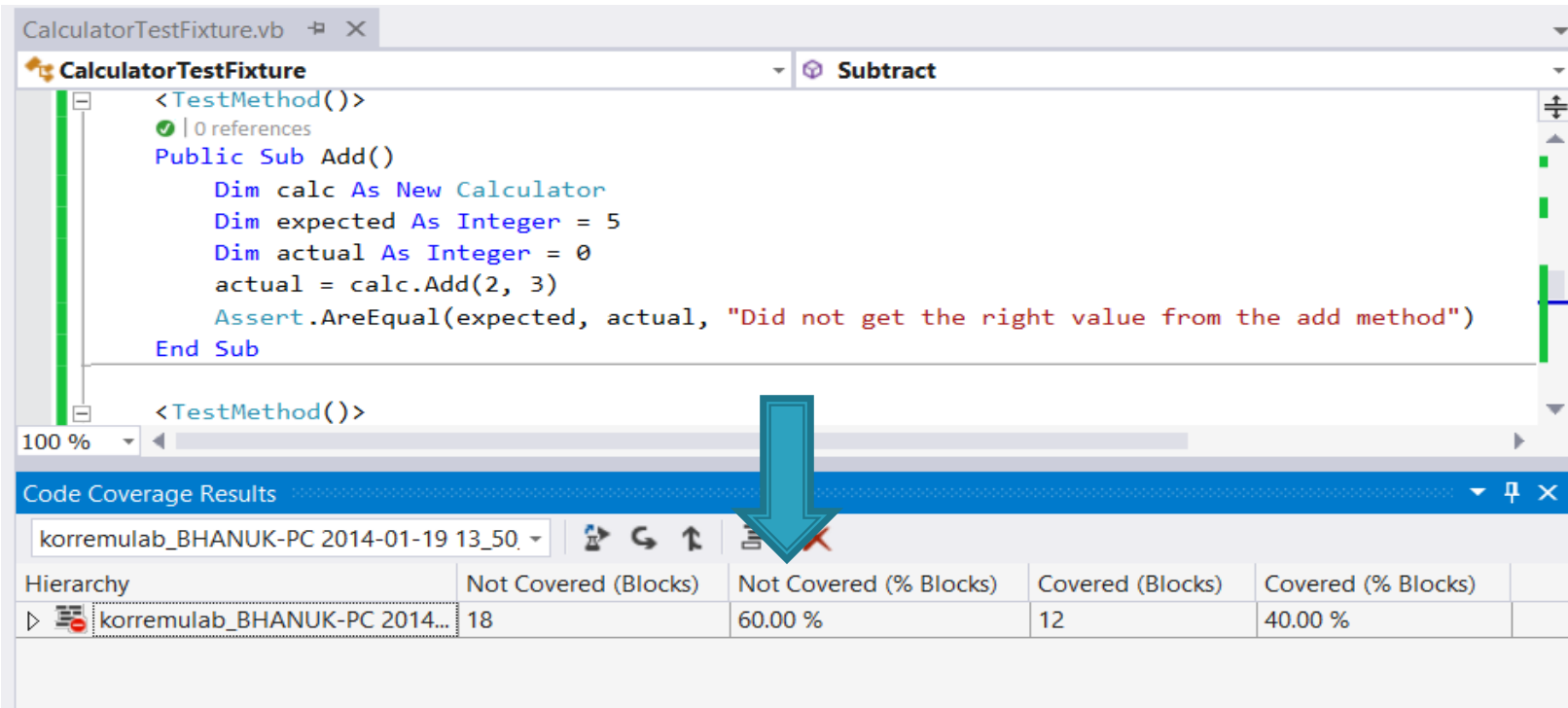
Remember just because you have 100 % code coverage you may not have working code

How to Run Code Coverage?



Go to Test → Analyze Code Coverage → All Tests

Code Coverage Results



The screenshot shows a Visual Studio window with a code editor and a Code Coverage Results window. The code editor displays the following code:

```
<TestMethod(>  
  | 0 references  
  Public Sub Add()  
    Dim calc As New Calculator  
    Dim expected As Integer = 5  
    Dim actual As Integer = 0  
    actual = calc.Add(2, 3)  
    Assert.AreEqual(expected, actual, "Did not get the right value from the add method")  
  End Sub  
</TestMethod(>
```

The Code Coverage Results window shows the following data:

| Hierarchy | Not Covered (Blocks) | Not Covered (% Blocks) | Covered (Blocks) | Covered (% Blocks) |
|---------------------------------------|----------------------|------------------------|------------------|--------------------|
| korremulab_BHANUK-PC 2014-01-19 13_50 | 18 | 60.00 % | 12 | 40.00 % |

A blue arrow points from the Code Coverage Results window to the code editor, indicating the coverage status of the code.

Only 40 % of code is covered in the above example. The developer needs to write more tests.

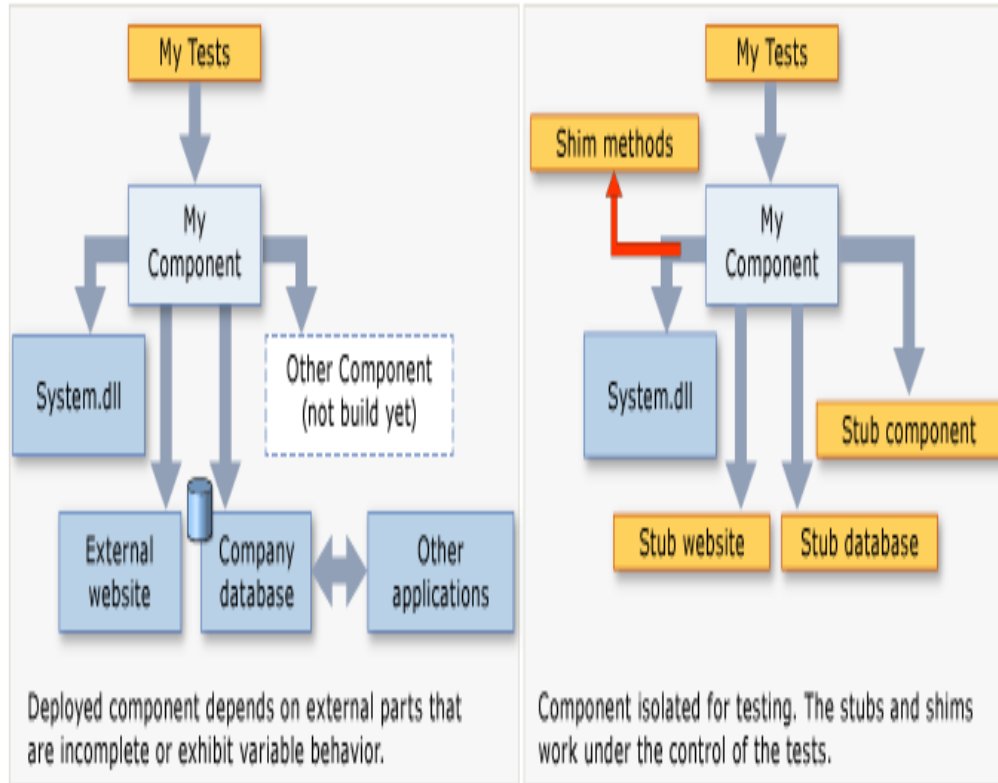
Code Coverage Drilldown Results

| Code Coverage Results | | | | | |
|---------------------------------------|----------------------|------------------------|------------------|--------------------|--|
| korremulab_BHANUK-PC 2014-01-22 21_09 | | | | | |
| Hierarchy | Not Covered (Blocks) | Not Covered (% Blocks) | Covered (Blocks) | Covered (% Blocks) | |
| ▲ korremulab_BHANUK-PC 2014... | 18 | 41.86 % | 25 | 58.14 % | |
| ▲ gmis.business.dll | 11 | 64.71 % | 6 | 35.29 % | |
| ▲ GMIS.Business | 6 | 50.00 % | 6 | 50.00 % | |
| ▲ Calculator | 6 | 50.00 % | 6 | 50.00 % | |
| Add(Integer, Int... | 0 | 0.00 % | 2 | 100.00 % | |
| Divide(Integer, l... | 6 | 100.00 % | 0 | 0.00 % | |
| Multiply(Integer,... | 0 | 0.00 % | 2 | 100.00 % | |
| Subtract(Integer... | 0 | 0.00 % | 2 | 100.00 % | |
| ▷ GMIS.Business.My | 5 | 100.00 % | 0 | 0.00 % | |
| ▷ gmis.unittesting.unittests.dll | 7 | 26.92 % | 19 | 73.08 % | |



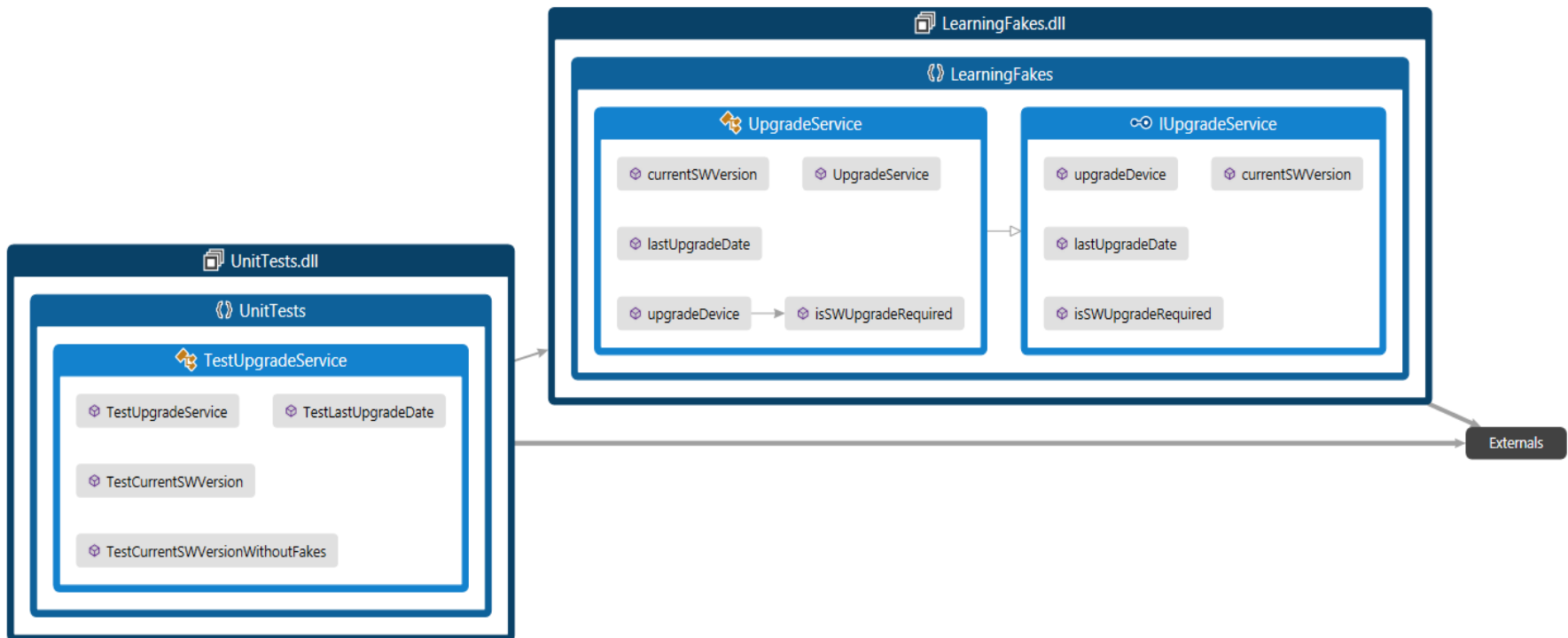
You can drill down and see which function is not covered by Unit Tests

Visual Studio Fakes Testing Framework



- ▶ Fakes help you isolate the code you are testing by replacing other parts of the application with *stubs* or *shims*. These are small pieces of code that are under the control of your tests. By isolating your code for testing, you know that if the test fails, the cause is there and not somewhere else. Stubs and shims also let you test your code even if other parts of your application are not working yet.

Sample Project Dependency Diagram



<http://www.codeproject.com/Articles/582812/Unit-testing-with-Fakes-with-Visual-studio-Premium>

Test Class With No Fakes

```
↳  
    [TestMethod]  
    | 0 references  
    public void TestCurrentSWVersionWithoutFakes()  
    {  
        int expected = 5;  
        UpgradeService us = new UpgradeService();  
        int actual = us.currentSWVersion(1);  
        Assert.AreEqual(expected, actual, "Same Versions found");  
    }
```

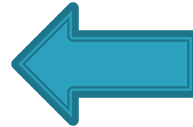
Here we are creating the instance of the class and calling its method

Stubs

- ▶ A stub replaces a class with a small substitute that implements the same interface. To use stubs, you have to design your application so that each component depends only on interfaces, and not on other components.
- ▶ Creates delegate based functions to be used in the test class
- ▶ The functions have to be consumed using the anonymous and lambda functions

Test Class With Stubs

```
[TestMethod]
| 0 references
public void TestCurrentSWVersion()
{
    int expected = 10;
    IUpgradeService us = new LearningFakes.Fakes.StubIUpgradeService()
    {
        CurrentSWVersionInt32 = (DeviceID) => { return 10;},
        IsSWUpgradeRequiredInt32 = (DeviceID) => { return true; }
    };
    int actual = us.currentSWVersion(1);
    Assert.AreEqual(expected, actual, "Same Versions found");
}
```



Interfaces are being used
Visual Studio Fakes Testing Framework creates fake
implementations of methods


<http://www.codeproject.com/Articles/582812/Unit-testing-with-Fakes-with-Visual-studio-Premium>

Shims

- ▶ A shim modifies the compiled code of your application at run time so that instead of making a specified method call, it runs the shim code that your test provides. Shims can be used to replace calls to assemblies that you cannot modify, such .NET assemblies.
- ▶ This is used where classes are tightly coupled and there are no interfaces

Test Class With Shim

```
[TestMethod]
| 0 references
public void TestLastUpgradeDate()
{
    using (ShimsContext.Create())
    {
        System.Fakes.ShimDateTime.NowGet =
            () =>
            { return new DateTime(2010, 11, 5); };
        var fakeTime = DateTime.Now; // It is always DateTime(2010, 11, 5);
        Console.WriteLine(fakeTime.ToShortDateString());
    }
    var correctTime = DateTime.Now;
    Console.WriteLine(correctTime.ToShortDateString());
}
```



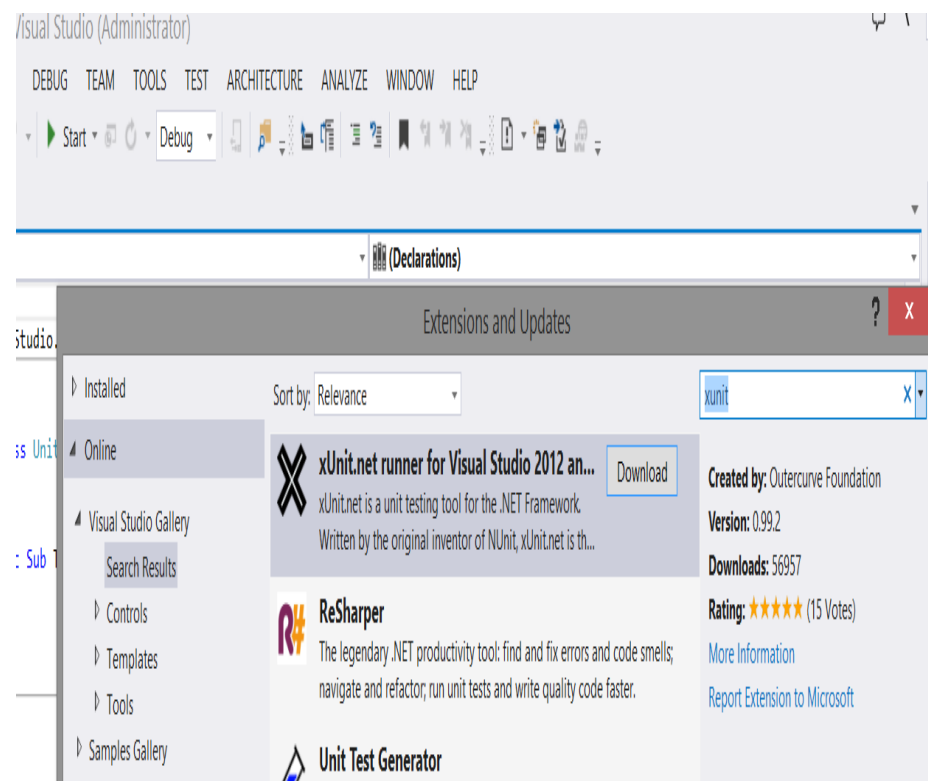
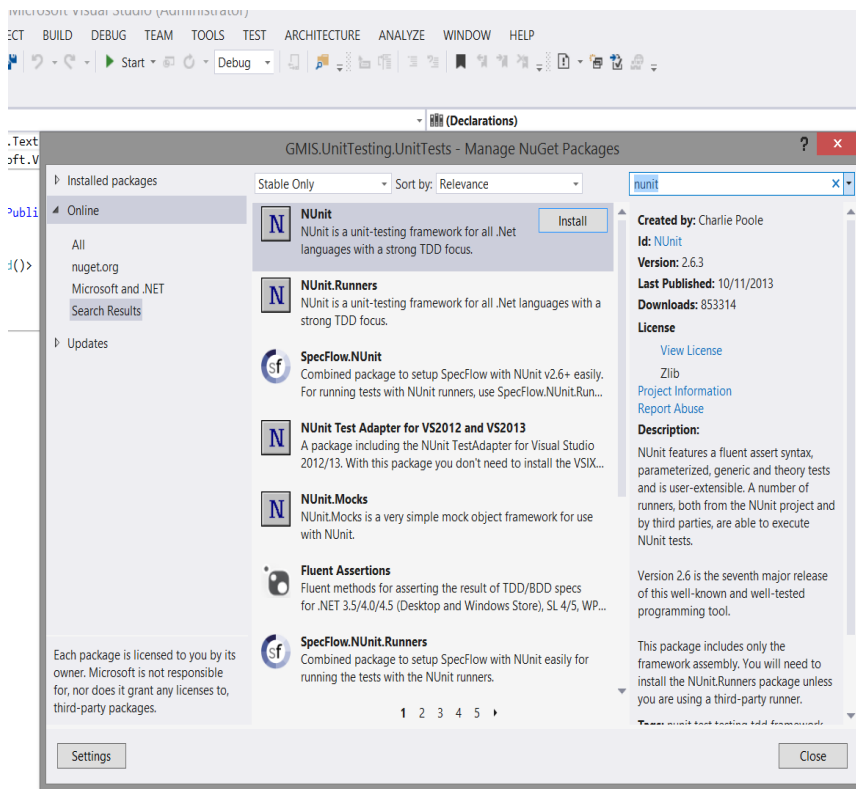
Used when code has no interfaces

Visual Studio Fakes Testing Framework creates fake implementations of .NET methods

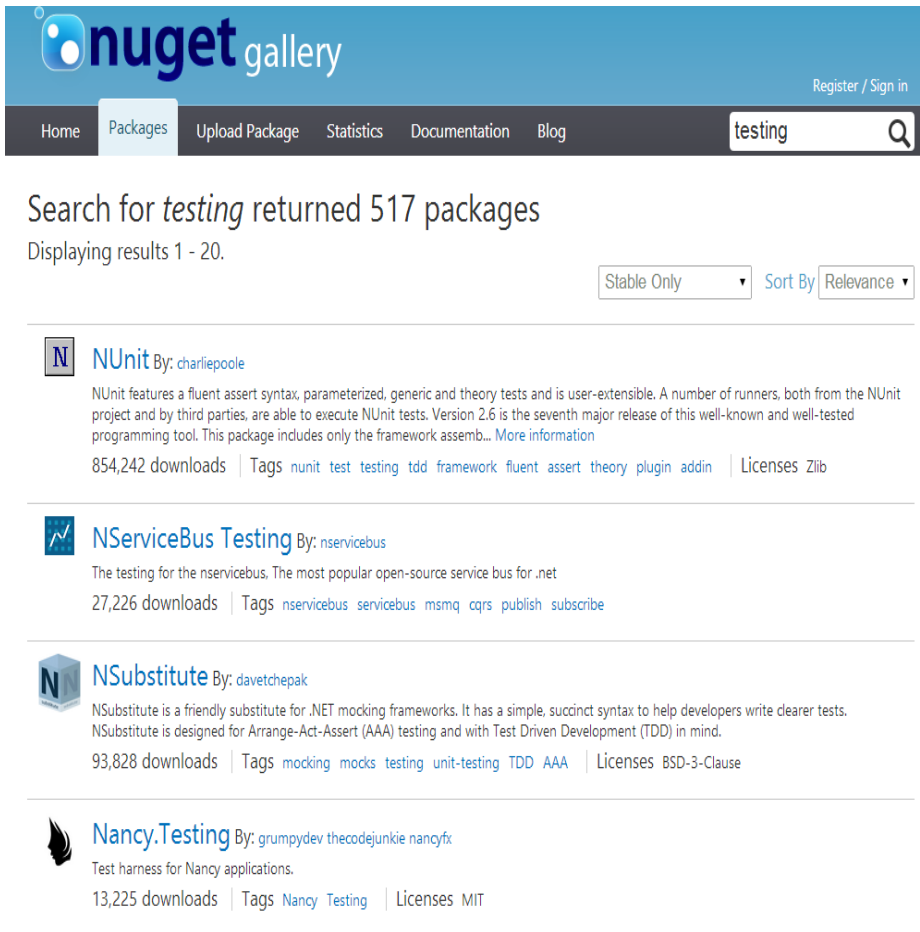
You get the current DateTime outside the ShimsContext Scope

Use NuGet to Get Other Testing Frameworks

Search NuGet and install other testing frameworks like NUnit or XUnit



What is NuGet?

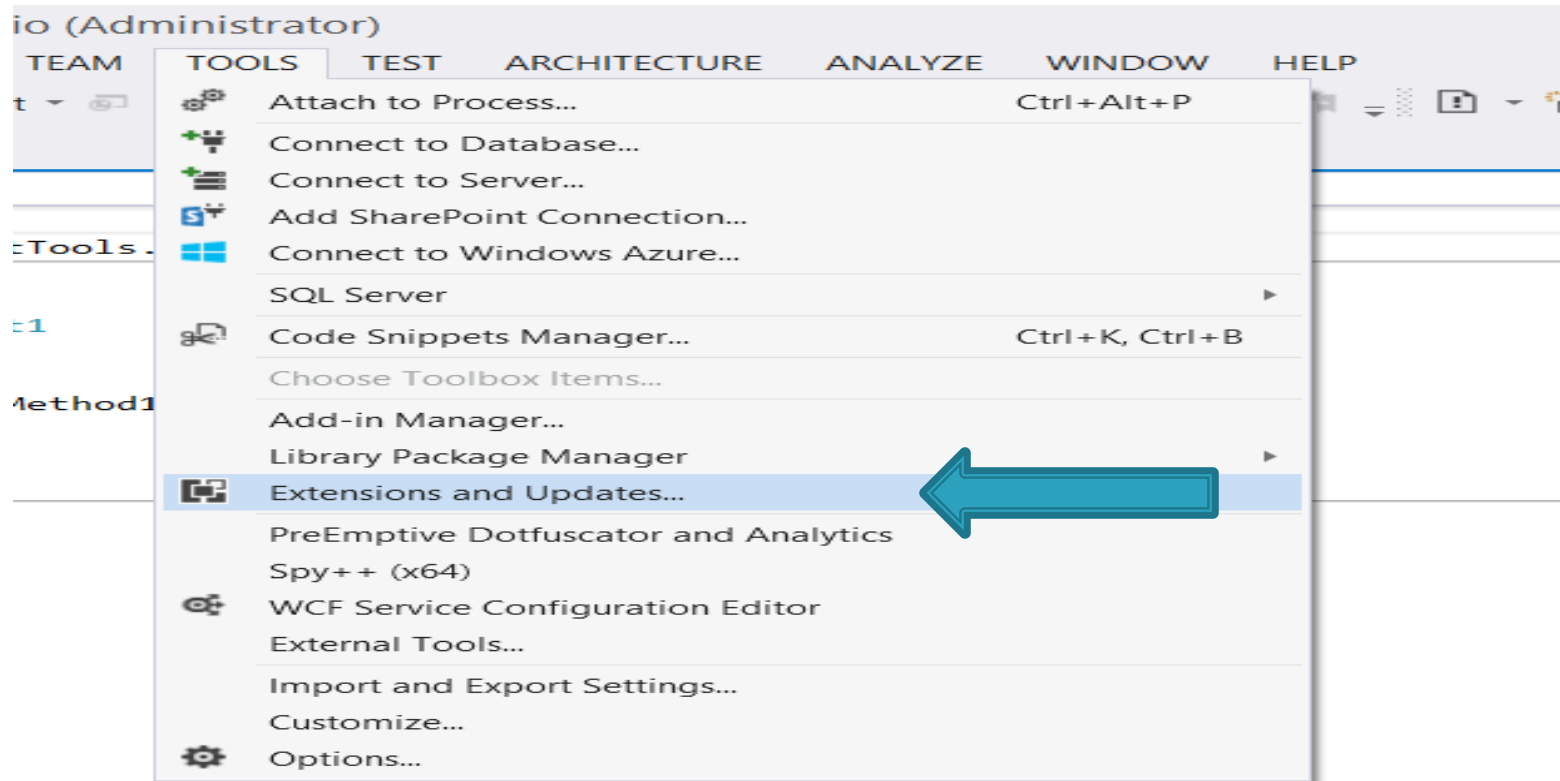


The screenshot shows the NuGet Gallery website interface. At the top, there is a navigation bar with the NuGet logo and the word 'gallery'. Below the navigation bar, there are tabs for 'Home', 'Packages', 'Upload Package', 'Statistics', 'Documentation', and 'Blog'. A search bar contains the text 'testing'. Below the search bar, it says 'Search for *testing* returned 517 packages' and 'Displaying results 1 - 20.'. There are two dropdown menus: 'Stable Only' and 'Sort By Relevance'. Below these are four package listings:

- NUnit** By: charliepoole
NUnit features a fluent assert syntax, parameterized, generic and theory tests and is user-extensible. A number of runners, both from the NUnit project and by third parties, are able to execute NUnit tests. Version 2.6 is the seventh major release of this well-known and well-tested programming tool. This package includes only the framework assem... [More information](#)
854,242 downloads | Tags [nunit](#) [test](#) [testing](#) [tdd](#) [framework](#) [fluent](#) [assert](#) [theory](#) [plugin](#) [addin](#) | Licenses [Zlib](#)
- NServiceBus Testing** By: nservicebus
The testing for the nservicebus. The most popular open-source service bus for .net
27,226 downloads | Tags [nservicebus](#) [servicebus](#) [msmq](#) [cqrs](#) [publish](#) [subscribe](#)
- NSubstitute** By: davetheapak
NSubstitute is a friendly substitute for .NET mocking frameworks. It has a simple, succinct syntax to help developers write clearer tests. NSubstitute is designed for Arrange-Act-Assert (AAA) testing and with Test Driven Development (TDD) in mind.
93,828 downloads | Tags [mocking](#) [mocks](#) [testing](#) [unit-testing](#) [TDD](#) [AAA](#) | Licenses [BSD-3-Clause](#)
- Nancy.Testing** By: grumpydev thecodejunkie nancyfx
Test harness for Nancy applications.
13,225 downloads | Tags [Nancy](#) [Testing](#) | Licenses [MIT](#)

- ▶ NuGet is the package manager for the Microsoft development platform including .NET. The NuGet client tools provide the ability to produce and consume packages. The NuGet Gallery is the central package repository used by all package authors and consumers.

How to Get to NuGet?



Go to Tools → Extensions and Updates

Disadvantages of TDD

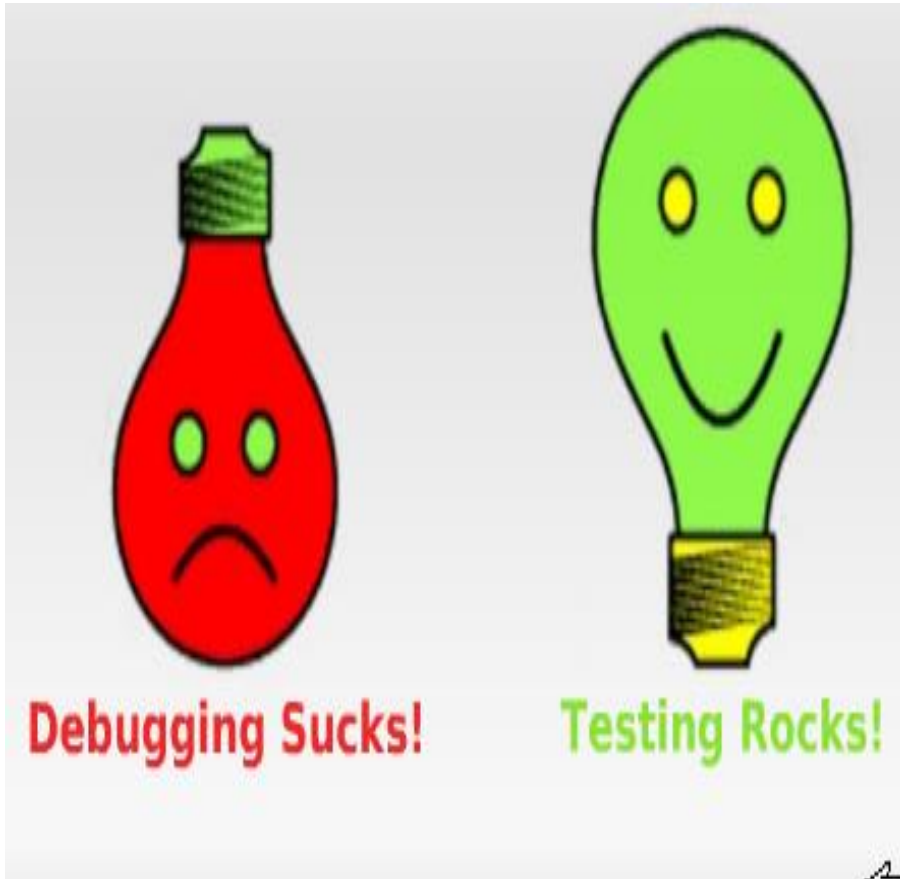
- ▶ Learning curve
- ▶ Might add complexity to the code
- ▶ Design changes
- ▶ Continuous tweaking

Advantages of the Test Driven Development



- ▶ Higher quality code and fewer defects
- ▶ Well documented code
- ▶ Automated regression testing

Takeaways



- ▶ This picture is posted above the toilets at Google
- ▶ Its better to test and fix bugs in the code before it goes live
- ▶ Developers should strive to be proactive rather than reactive, remember the main goal of the IT shop is to ship working software. Good practices that will enhance productivity will need to be encouraged.

TestDrivenDevelopment,
RedGreenRefractorCycle,
NuGet, LongRunningTests,
Fakes, Debugging,
VisualStudio2013, NUnit,
Shims, TestSetup,
TestExplorer,
XUnit Stubs,
UnitTests, CodeCoverage,
CodeTestFirst,





Contact Info

- ▶ Twitter: <https://twitter.com/bkraider>
- ▶ Facebook: <https://www.facebook.com/bkraider>
- ▶ Linked In: <http://www.linkedin.com/pub/bhanu-korremula/9/38/966>
- ▶ Blog: <http://bhanukorremula.wordpress.com/>
- ▶ Work Email: korremulab@rcgov.us
- ▶ Personal Email: bkraider@gmail.com

References

- ▶ http://en.wikipedia.org/wiki/Test_Driven_Development
- ▶ <http://msdn.microsoft.com/en-us/library/hh212233.aspx>
- ▶ <http://www.jrothman.com/2000/10/what-does-it-cost-you-to-fix-a-defect-and-why-should-you-care/>
- ▶ <http://superwebdeveloper.com/2009/11/25/the-incredible-rate-of-diminishing-returns-of-fixing-software-bugs/>
- ▶ http://en.wikipedia.org/wiki/Unit_testing
- ▶ Pluralsight Test First Development, Visual Studio ALM
- ▶ <http://www.amazon.co.uk/Engineering-Economics-Prentice-Hall-computing-technology/dp/0138221227>
- ▶ <http://sqa.stackexchange.com/questions/2899/industry-average-bug-fix-cost>
- ▶ <http://msdn.microsoft.com/en-us/library/hh270865.aspx>
- ▶ <http://www.microsoftvirtualacademy.com/training-courses/software-testing-with-visual-studio-2012-exam-70-497-jump-start#?fbid=HdC7BLz8mVC>
- ▶ <http://channel9.msdn.com/Series/Visual-Studio-2012-Premium-and-Ultimate-Overview/Visual-Studio-Ultimate-2012-Improving-quality-with-unit-tests-and-fakes>
- ▶ <http://academictips.org/blogs/the-story-of-a-woodcutter/>
- ▶ <http://www.flickr.com/photos/dullhunk/3712840085/>
- ▶ <http://blog.felho.hu/wp-content/debuggingsuckststingrocks.png>
- ▶ <http://www.nuget.org/>
- ▶ <http://www.nunit.org/>
- ▶ <http://xunit.codeplex.com/>

References

- ▶ <http://msdn.microsoft.com/en-us/library/dd537628.aspx>
- ▶ <http://stackoverflow.com/questions/64333/disadvantages-of-test-driven-development>
- ▶ <http://msdn.microsoft.com/en-us/library/ms182527.aspx>
- ▶ <http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.assert.aspx>