Unit Test Your Code 1
Unit Test Basics 5
Generate unit tests for your code with IntelliTest 17
Run unit tests with Test Explorer 25
Creating and Running Unit Tests for Windows Store Apps 36
Writing Unit Tests 42
Creating and Running Unit Tests for Managed Code 43
Quick Start Test Driven Development with Test Explorer 53
Isolating Code Under Test with Microsoft Fakes 62
How To Create a Data-Driven Unit Test 67
Use UI Automation To Test Your Code 72
Walkthrough Creating, Editing and Maintaining a Coded UI Test 89
Test Windows Phone 8.1 Apps with Coded UI Tests 99
Creating a Data-Driven Coded UI Test 119
Get started with developer testing tools 132
User acceptance testing 141
Track test status 145
Run manual tests 154

Unit Test Your Code

Visual Studio 2015

Unit tests give developers and testers a quick way to look for logic errors in the methods of classes in Visual C#, Visual Basic, and Visual C++ projects.

The unit test tools include:

1. **Test Explorer.** Test Explorer lets you run unit tests and view their results. Test Explorer can use any unit test framework, including a third-party framework, that has an adapter for the Explorer.

2. **Microsoft unit test framework for managed code.** The Microsoft unit test framework for managed code is installed with Visual Studio and provides a framework for testing .NET code.

3. **Microsoft unit test framework for C++.** The Microsoft unit test framework for C++ is installed with Visual Studio and provides a framework for testing native code.

4. **Code coverage tools.** You can determine the amount of product code that your unit tests exercise from one command in Test Explorer.

5. **Microsoft Fakes isolation framework.** The Microsoft Fakes isolation framework can create substitute classes and methods for production and system code that create dependencies in the code under test. By implementing the fake delegates for a function, you control the behavior and output of the dependency object.

You can also use IntelliTest to explore your .NET code to generate test data and a suite of unit tests. For every statement in the code, a test input is generated that will execute that statement. A case analysis is performed for every conditional branch in the code.

Key tasks

Use the following topics to help with understanding and creating unit tests:

Tasks	Associated Topics
Quick starts and walkthroughs: Use the following topics to learn unit testing in Visual Studio from code examples.	• Walkthrough: Creating and Running Unit Tests for Managed Code • Quick Start: Test Driven Development with Test Explorer • Unit testing existing C++ applications with Test Explorer • Unit testing native code with Test Explorer

Unit testing with Test Explorer: Learn how Test Explorer can help create more productive and efficient unit tests.	• Unit Test Basics • Create a unit test project • Run unit tests with Test Explorer • Install third-party unit test frameworks • Upgrading Unit Tests from Visual Studio 2010
Unit testing managed code:	• Writing Unit Tests for the .NET Framework with the Microsoft Unit Test Framework for Managed Code
Unit testing C++ code	• Writing Unit tests for C/C++ with the Microsoft Unit Testing Framework for C++
Isolating unit tests	• Isolating Code Under Test with Microsoft Fakes
Use code coverage to identify what proportion of your project's code is being tested using unit tests: Learn about the code coverage feature of Visual Studio Application Lifecycle Management testing tools.	• Using Code Coverage to Determine How Much Code is being Tested
Perform stress and performance analysis by using load tests for your unit tests: You can create a load test and add your unit tests to it to help isolate performance and stress issues in your application. **Note** Creating and using load tests requires Visual Studio Enterprise.	• e2985d15-60a7-4177-93b4-f986c2936337 • 03cc073e-9bdf-4530-ae46-504a51884594 • 3d6128d2-82b0-42fc-bda2-23a8aa03be07
Set and enforce quality gates: You can create quality gates to enforce that tests are run before code is checked in to help ensure the quality of the code.	• Set and Enforce Quality Gates
Extend the unit test type: You can add functionality to your tests that might not be in the Unit Test Framework. For example, you can add a test property that specifies if a test should run as a normal user or not. Or you can extend the framework to add row attributes to a method and use the data in that row inside the test.	For sample code of how to extend the unit test framework, see the following Microsoft Web site.

Set testing options: For example, you can specify where test results are stored.	Configure unit tests by using a .runsettings file

Related tasks

Reviewing Test Results in Microsoft Test Manager

Describes test results and ways to work with them, including how to view, save, and delete them.

Running System Tests Using Microsoft Visual Studio

Provides links to information about using Visual Studio as opposed to using Microsoft Test Manager to run automated tests.

Reference

Microsoft.VisualStudio.TestTools.UnitTesting
Describes the UnitTesting namespace, which provides attributes, exceptions, asserts, and other classes that support unit testing.

Microsoft.VisualStudio.TestTools.UnitTesting.Web
Describes the UnitTesting.Web namespace, which extends the UnitTesting namespace by providing support for ASP.NET and Web service unit tests.

External resources

Videos

Channel 9: Unit testing your Windows Store apps built using XAML

Forums

Visual Studio Unit Testing

Guidance

Testing for Continuous Delivery with Visual Studio 2012 – Chapter 2: Unit Testing: Testing the Inside

Reference

Content Index for Unit Tests

See Also

Improve Code Quality
Testing the application

Unit Test Basics

Visual Studio 2015

Updated: January 7, 2016

Check that your code is working as expected by creating and running unit tests. It's called unit testing because you break down the functionality of your program into discrete testable behaviors that you can test as individual *units*. Visual Studio Test Explorer provides a flexible and efficient way to run your unit tests and view their results in Visual Studio. Visual Studio installs the Microsoft unit testing frameworks for managed and native code. Use a *unit testing framework* to create unit tests, run them, and report the results of these tests. Rerun unit tests when you make changes to test that your code is still working correctly. When you use Visual Studio Enterprise, you can run tests automatically after every build.

Unit testing has the greatest effect on the quality of your code when it's an integral part of your software development workflow. As soon as you write a function or other block of application code, create unit tests that verify the behavior of the code in response to standard, boundary, and incorrect cases of input data, and that check any explicit or implicit assumptions made by the code. With *test driven development*, you create the unit tests before you write the code, so you use the unit tests as both design documentation and functional specifications.

You can quickly generate test projects and test methods from your code, or manually create the tests as you need them. When you use IntelliTest to explore your .NET code, you can generate test data and a suite of unit tests. For every statement in the code, a test input is generated that will execute that statement. Find out how to generate unit tests for your code.

Test Explorer can also run third-party and open source unit test frameworks that have implemented Test Explorer add-on interfaces. You can add many of these frameworks through the Visual Studio Extension Manager and the Visual Studio gallery. See Install third-party unit test frameworks

- Quick starts
- The MyBank Solution example
- Create unit test projects and test methods
- Write your tests
- Run tests in Test Explorer
- Run and view tests

Unit testing overview

Quick starts

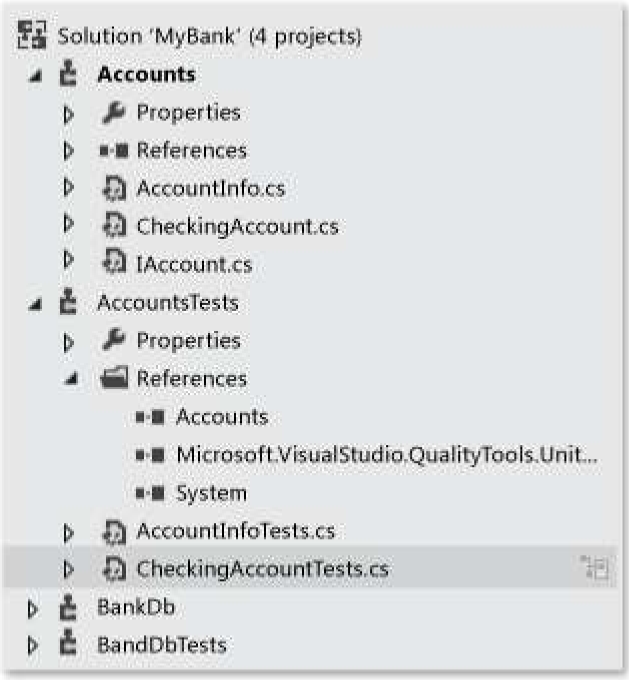
For an introduction to unit testing that takes you directly into coding, see one of these topics:

- Walkthrough: Creating and Running Unit Tests for Managed Code

- Quick Start: Test Driven Development with Test Explorer
- Unit testing native code with Test Explorer

The MyBank Solution example

In this topic, we use the development of a fictional application called `MyBank` as an example. You don't need the actual code to follow the explanations in this topic. Test methods are written in C# and presented by using the Microsoft Unit Testing Framework for Managed Code, However, the concepts are easily transferred to other languages and frameworks.



Our first attempt at a design for the `MyBank` application includes an accounts component that represents an individual account and its transactions with the bank, and a database component that represents the functionality to aggregate and manage the individual accounts.

We create a `MyBank` solution that contains two projects:

- `Accounts`
- `BankDb`

Our first attempt at designing the `Accounts` project contain a class to hold basic information about an account, an interface that specifies the common functionality of any type of account, like depositing and withdrawing assets from the account, and a class derived from the interface that represents a checking account. We begin the Accounts projects by creating the following source files:

- `AccountInfo.cs` defines the basic information for an account.
- `IAccount.cs` defines a standard `IAccount` interface for an account, including methods to deposit and withdraw

assets from an account and to retrieve the account balance.

- `CheckingAccount.cs` contains the `CheckingAccount` class that implements the `IAccounts` interface for a checking account.

We know from experience that one thing a withdrawal from a checking account must do is to make sure that the amount withdrawn is less than the account balance. So we override the `IAccount.Withdaw` method in `CheckingAccount` with a method that checks for this condition. The method might look like this:

C#

```
public void Withdraw(double amount)
{
    if(m_balance >= amount)
    {
        m_balance -= amount;
    }
    else
    {
        throw new ArgumentException(amount, "Withdrawal exceeds balance!")
    }
}
```

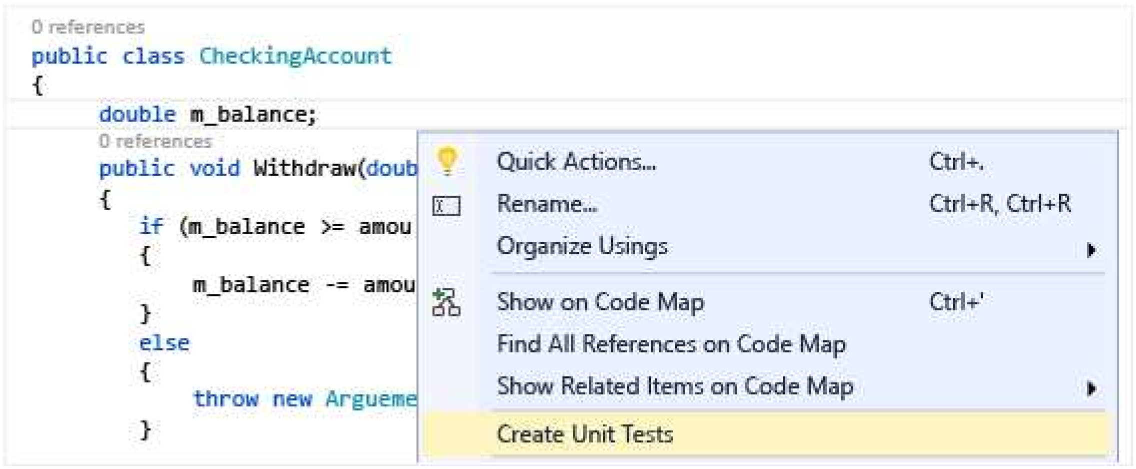
Now that we have some code, it's time for testing.

Create unit test projects and test methods

It is often quicker to generate the unit test project and unit test stubs from your code. Or you can choose to create the unit test project and tests manually depending on your requirements.

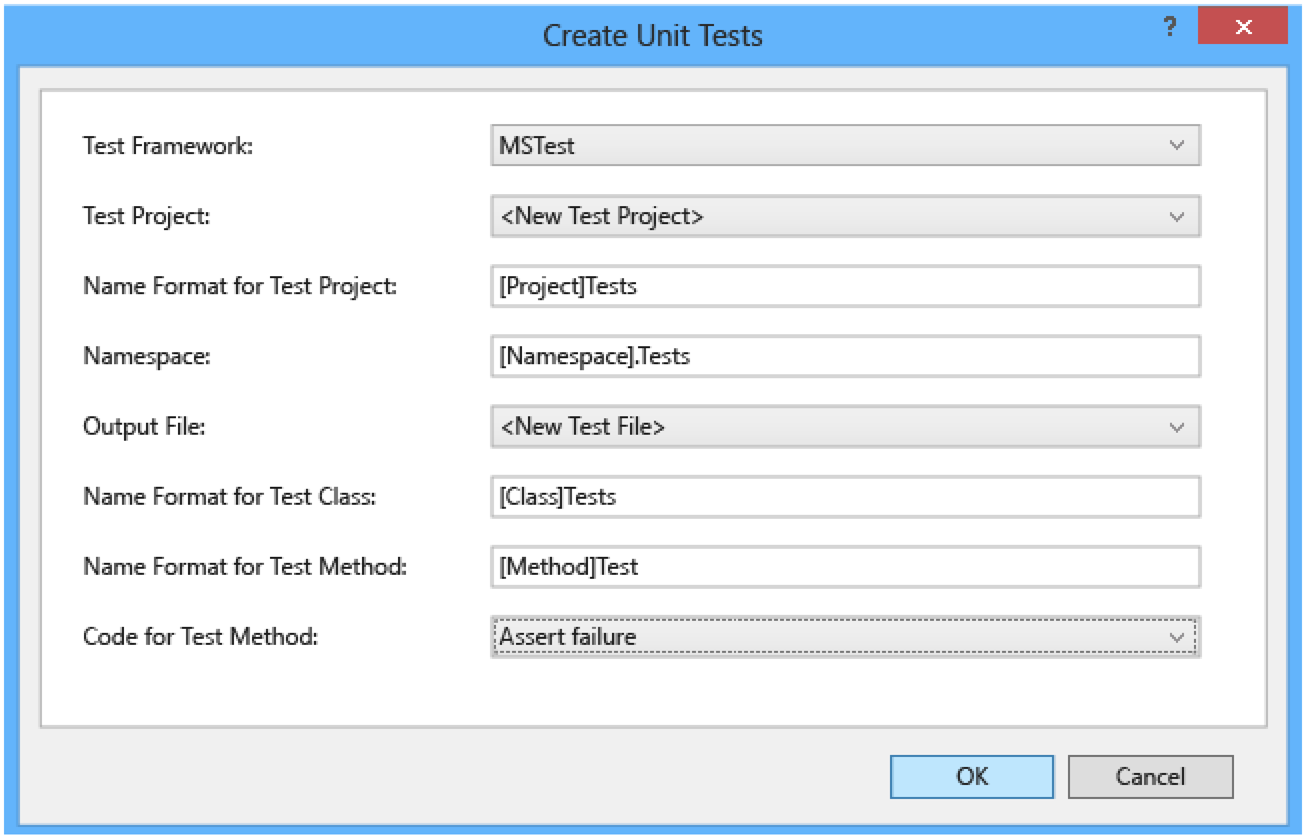
Generate unit test project and unit test stubs

1. From the code editor window, right-click and choose **Create Unit Tests** from the context menu.



```
0 references
public class CheckingAccount
{
    double m_balance;
    0 references
    public void Withdraw(doub
    {
        if (m_balance >= amou
        {
            m_balance -= amou
        }
        else
        {
            throw new Argueme
        }
```

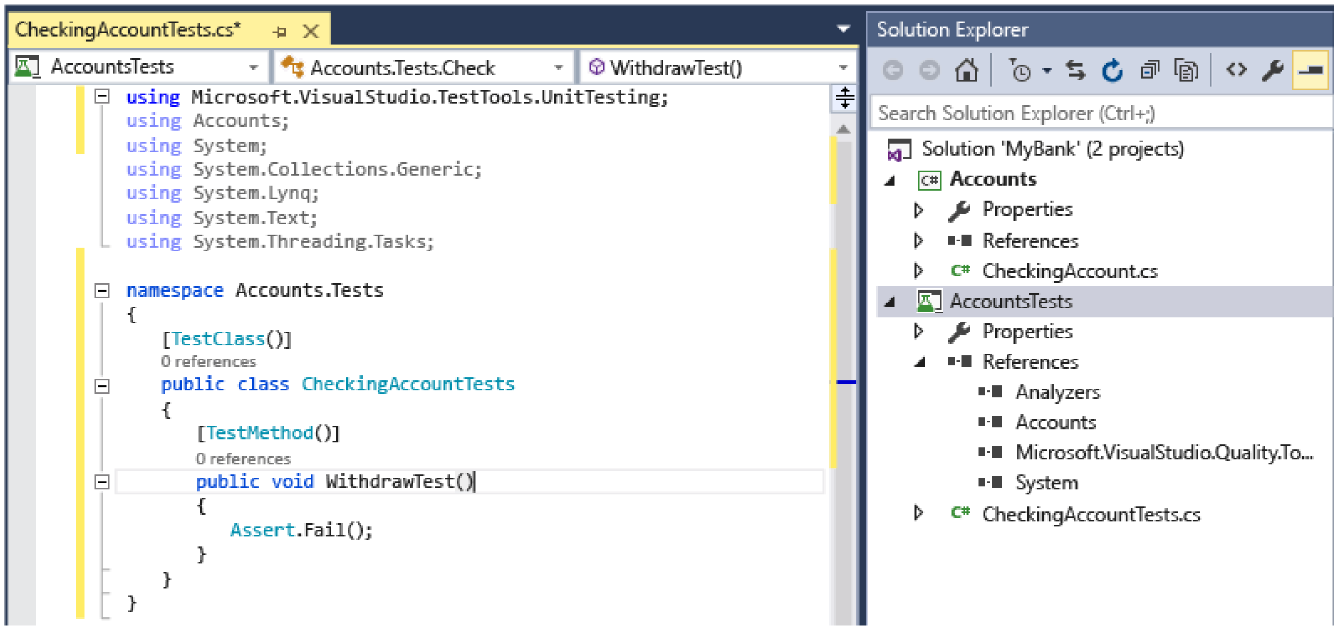
2. Click OK to accept the defaults to create your unit tests, or change the values used to create and name the unit test project and the unit tests. You can select the code that is added by default to the unit test methods.



Create Unit Tests

Test Framework:	MSTest
Test Project:	<New Test Project>
Name Format for Test Project:	[Project]Tests
Namespace:	[Namespace].Tests
Output File:	<New Test File>
Name Format for Test Class:	[Class]Tests
Name Format for Test Method:	[Method]Test
Code for Test Method:	Assert failure

3. The unit test stubs are created in a new unit test project for all the methods in the class.



CheckingAccountTests.cs*

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Accounts;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Accounts.Tests
{
    [TestClass()]
    public class CheckingAccountTests
    {
        [TestMethod()]
        public void WithdrawTest()
        {
            Assert.Fail();
        }
    }
}
```

4. Now jump ahead to learn how to add code to the unit test methods to make your unit test meaningful, and any extra unit tests that you might want to add to thoroughly test your code.

Create your unit test project and unit tests manually

A unit test project usually mirrors the structure of a single code project. In the MyBank example, you add two unit test projects named `AccountsTests` and `BankDbTests` to the `MyBanks` solution. The test project names are arbitrary, but adopting a standard naming convention is a good idea.

To add a unit test project to a solution:

1. On the **File** menu, choose **New** and then choose **Project** (Keyboard Ctrl + Shift + N).

2. On the New Project dialog box, expand the **Installed** node, choose the language that you want to use for your test project, and then choose **Test**.

3. To use one of the Microsoft unit test frameworks, choose **Unit Test Project** from the list of project templates. Otherwise, choose the project template of the unit test framework that you want to use. To test the `Accounts` project of our example, you would name the project `AccountsTests`.

   Warning

   Not all third-party and open source unit test frameworks provide a Visual Studio project template. Consult the framework document for information about creating a project.

4. In your unit test project, add a reference to the code project under test, in our example to the Accounts project.

   To create the reference to the code project:

   a. Select the project in Solution Explorer.

   b. On the **Project** menu, choose **Add Reference**.

   c. On the Reference Manager dialog box, open the **Solution** node and choose **Projects**. Select the code project name and close the dialog box.

Each unit test project contains classes that mirror the names of the classes in the code project. In our example, the `AccountsTests` project would contain the following classes:

- `AccountInfoTests` class contains the unit test methods for the `AccountInfo` class in the `BankAccount` project

- `CheckingAccountTests` class contains the unit test methods for `CheckingAccount` class.

Write your tests

The unit test framework that you use and Visual Studio IntelliSense will guide you through writing the code for your unit tests for a code project. To run in Test Explorer, most frameworks require that you add specific attributes to identify unit test methods. The frameworks also provide a way—usually through assert statements or method attributes—to indicate whether the test method has passed or failed. Other attributes identify optional setup methods that are at class initialization and before each test method and teardown methods that are run after each test method and before the class is destroyed.

The AAA (Arrange, Act, Assert) pattern is a common way of writing unit tests for a method under test.

- The **Arrange** section of a unit test method initializes objects and sets the value of the data that is passed to the method under test.
- The **Act** section invokes the method under test with the arranged parameters.
- The **Assert** section verifies that the action of the method under test behaves as expected.

To test the `CheckingAccount.Withdraw` method of our example, we can write two tests: one that verifies the standard behavior of the method, and one that verifies that a withdrawal of more than the balance will fail. In the `CheckingAccountTests` class, we add the following methods:

C#

```
[TestMethod]
public void Withdraw_ValidAmount_ChangesBalance()
{
    // arrange
    double currentBalance = 10.0;
    double withdrawal = 1.0;
    double expected = 9.0;
    var account = new CheckingAccount("JohnDoe", currentBalance);
    // act
    account.Withdraw(withdrawal);
    double actual = account.Balance;
    // assert
    Assert.AreEqual(expected, actual);
}

[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void Withdraw_AmountMoreThanBalance_Throws()
{
    // arrange
    var account = new CheckingAccount("John Doe", 10.0);
    // act
    account.Withdraw(20.0);
    // assert is handled by the ExpectedException
}
```

Note that `Withdraw_ValidAmount_ChangesBalance` uses an explicit `Assert` statement to determine whether the test method passes or fails, while `Withdraw_AmountMoreThanBalance_Throws` uses the `ExpectedException` attribute to determine the success of the test method. Under the covers, a unit test framework wraps test methods in try/catch statements. In most cases, if an exception is caught, the test method fails and the exception is ignored. The `ExpectedException` attribute causes the test method to pass if the specified exception is thrown.

For more information about the Microsoft Unit Testing Frameworks, see one of the following topics:

- Writing Unit Tests for the .NET Framework with the Microsoft Unit Test Framework for Managed Code
- Writing Unit tests for C/C++ with the Microsoft Unit Testing Framework for C++

Set timeouts for unit tests

To set a timeout on an individual test method:

VB

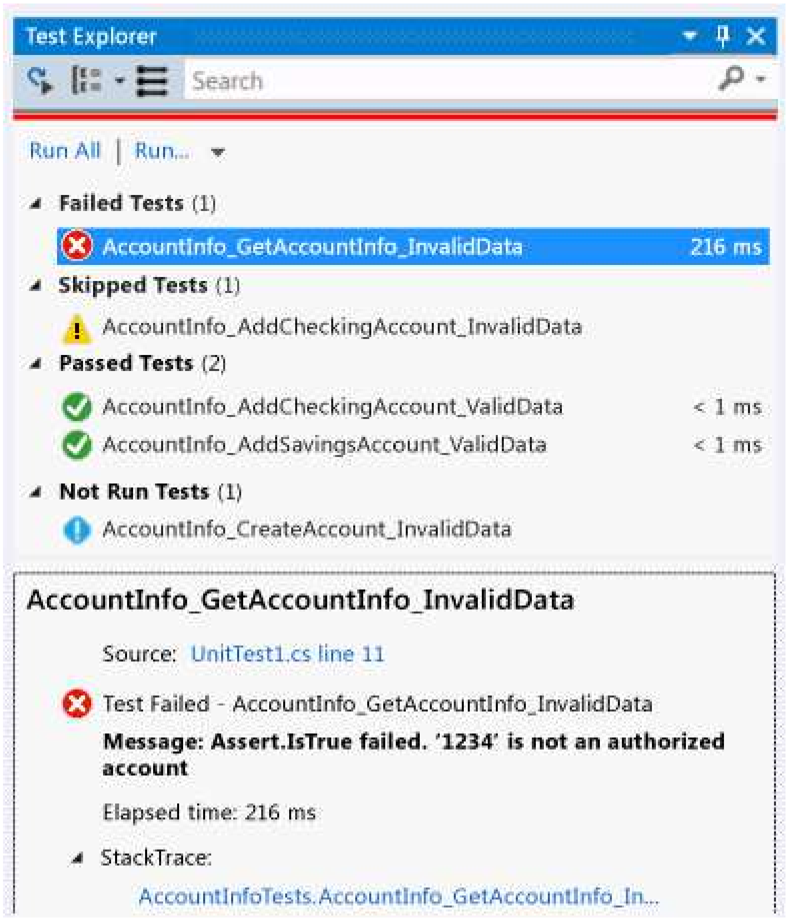
To set the timeout to the maximum allowed:

C#

```
[TestMethod]
[Timeout(TestTimeout.Infinite)]  // Milliseconds
public void My_Test ()
{ ...
}
```

Run tests in Test Explorer

When you build the test project, the tests appear in Test Explorer. If Test Explorer is not visible, choose **Test** on the Visual Studio menu, choose **Windows**, and then choose **Test Explorer**.



Failed Tests (1)
AccountInfo_GetAccountInfo_InvalidData 216 ms
Skipped Tests (1)
AccountInfo_AddCheckingAccount_InvalidData
Passed Tests (2)
AccountInfo_AddCheckingAccount_ValidData < 1 ms
AccountInfo_AddSavingsAccount_ValidData < 1 ms
Not Run Tests (1)
AccountInfo_CreateAccount_InvalidData

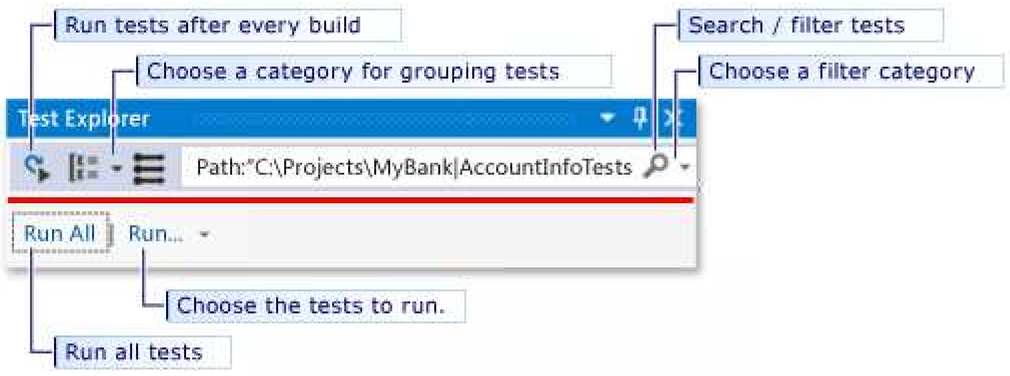
AccountInfo_GetAccountInfo_InvalidData
Source: UnitTest1.cs line 11
Test Failed - AccountInfo_GetAccountInfo_InvalidData
Message: Assert.IsTrue failed. '1234' is not an authorized account
Elapsed time: 216 ms
StackTrace:
AccountInfoTests.AccountInfo_GetAccountInfo_In...

As you run, write, and rerun your tests, the default view of Test Explorer displays the results in groups of **Failed Tests**, **Passed Tests**, **Skipped Tests** and **Not Run Tests**. You can choose a group heading to open the view that displays all them tests in that group.

You can also filter the tests in any view by matching text in the search box at the global level or by selecting one of the pre-defined filters. You can run any selection of the tests at any time. The results of a test run are immediately apparent in the pass/fail bar at the top of the explorer window. Details of a test method result are displayed when you select the test.

Run and view tests

The Test Explorer toolbar helps you discover, organize, and run the tests that you are interested in.



You can choose **Run All** to run all your tests, or choose **Run** to choose a subset of tests to run. After you run a set of tests, a summary of the test run appears at the bottom of the Test Explorer window. Select a test to view the details of that test in the bottom pane. Choose **Open Test** from the context menu (Keyboard: F12) to display the source code for the selected test.

If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution with the ☰ toggle button on the toolbar. This can noticeably reduce the time taken to run all the tests.

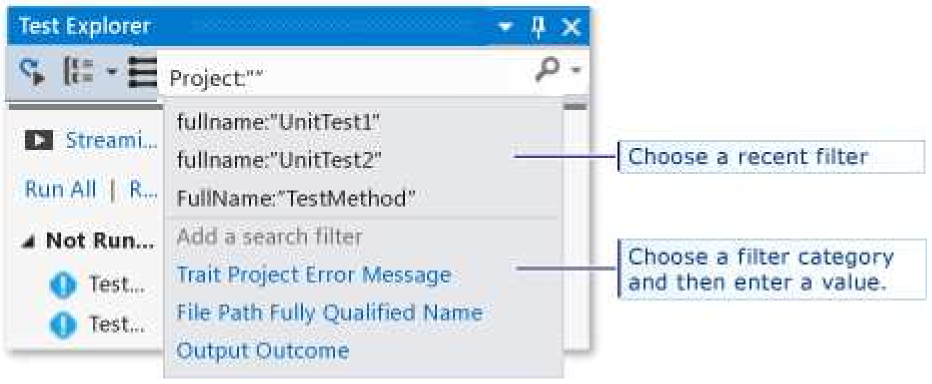
Run tests after every build

Warning
Running unit tests after every build is supported only in Visual Studio Enterprise.

	To run your unit tests after each local build, choose **Test** on the standard menu, choose **Run Tests After Build** on the Test Explorer toolbar.

Filter and group the test list

When you have a large number of tests, you can Type in Test Explorer search box to filter the list by the specified string. You can restrict your filter event more by choosing from the filter list.



	To group your tests by category, choose the **Group By** button.

For more information, see Run unit tests with Test Explorer

Q&A

Q: How do I debug unit tests?

A: Use Test Explorer to start a debugging session for your tests. Stepping through your code with the Visual Studio debugger seamlessly takes you back and forth between the unit tests and the project under test. To start debugging:

1. In the Visual Studio editor, set a breakpoint in one or more test methods that you want to debug.

   Note

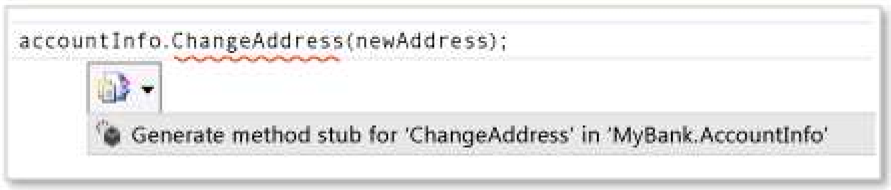
   Because test methods can run in any order, set breakpoints in all the test methods that you want to debug.

2. In Test Explorer, select the test methods and then choose **Debug Selected Tests** from the shortcut menu.

Learn more details about debugging unit tests.

Q: If I'm using TDD, how do I generate code from my tests?

A: Use IntelliSense to generate classes and methods in your project code. Write a statement in a test method that calls the class or method that you want to generate, then open the IntelliSense menu under the call. If the call is to a constructor of the new class, choose **Generate new type** from the menu and follow the wizard to insert the class in your code project. If the call is to a method, choose **Generate new method** from the IntelliSense menu.



```
accountInfo.ChangeAddress(newAddress);
```

Generate method stub for 'ChangeAddress' in 'MyBank.AccountInfo'

Q: Can I create unit tests that take multiple sets of data as input to run the test?

A: Yes. *Data-driven test methods* let you test a range of values with a single unit test method. Use a `DataSource` attribute for the test method that specifies the data source and table that contains the variable values that you want to test. In the method body, you assign the row values to variables using the `TestContext.DataRow[`*ColumnName*`]` indexer.

Note

These procedures apply only to test methods that you write by using the Microsoft unit test framework for managed code. If you're using a different framework, consult the framework documentation for equivalent functionality.

For example, assume we add an unnecessary method to the `CheckingAccount` class that is named `AddIntegerHelper`. `AddIntegerHelper` adds two integers.

To create a data-driven test for the `AddIntegerHelper` method, we first create an Access database named `AccountsTest.accdb` and a table named `AddIntegerHelperData`. The `AddIntegerHelperData` table defines columns to specify the first and second operands of the addition and a column to specify the expected result. We fill a number of rows with appropriate values.

C#

```
    [DataSource(
        @"Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\Projects\MyBank\TestData
    \AccountsTest.accdb",
        "AddIntegerHelperData"
    )]
    [TestMethod()]
    public void AddIntegerHelper_DataDrivenValues_AllShouldPass()
    {
        var target = new CheckingAccount();
        int x = Convert.ToInt32(TestContext.DataRow["FirstNumber"]);
        int y = Convert.ToInt32(TestContext.DataRow["SecondNumber"]);
        int expected = Convert.ToInt32(TestContext.DataRow["Sum"]);
        int actual = target.AddIntegerHelper(x, y);
        Assert.AreEqual(expected, actual);
    }
```

The attributed method runs once for each row in the table. Test Explorer reports a test failure for the method if any of the iterations fail. The test results detail pane for the method shows you the pass/fail status method for each row of data.

Learn more about data-driven unit tests.

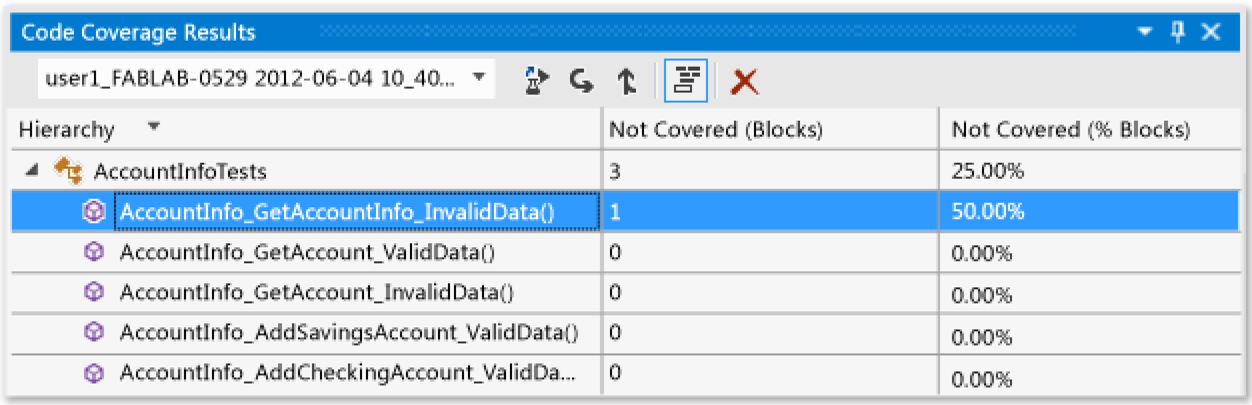
Q: Can I view how much of my code is tested by my unit tests?

A: Yes. You can determine the amount of your code that is actually being tested by your unit tests by using the Visual Studio code coverage tool. Native and managed languages and all unit test frameworks that can be run by the Unit Test Framework are supported.

You can run code coverage on selected tests or on all tests in a solution. The Code Coverage Results window displays the percentage of the blocks of product code that were exercised by line, function, class, namespace and module.

To run code coverage for test methods in a solution, choose **Tests** on the Visual Studio menu and then choose **Analyze code coverage**.

Coverage results appear in the Code Coverage Results window.



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)
AccountInfoTests	3	25.00%
AccountInfo_GetAccountInfo_InvalidData()	1	50.00%
AccountInfo_GetAccount_ValidData()	0	0.00%
AccountInfo_GetAccount_InvalidData()	0	0.00%
AccountInfo_AddSavingsAccount_ValidData()	0	0.00%
AccountInfo_AddCheckingAccount_ValidDa...	0	0.00%

Learn more about code coverage .

Q: How can I test methods in my code that have external dependencies?

A: Yes. If you have Visual Studio Enterprise, Microsoft Fakes can be used with test methods that you write by using unit test frameworks for managed code.

Microsoft Fakes uses two approaches to create substitute classes for external dependencies.

1. *Stubs* generate substitute classes derived from the parent interface of the target dependency class. Stub methods can be substituted for public virtual methods of the target class.

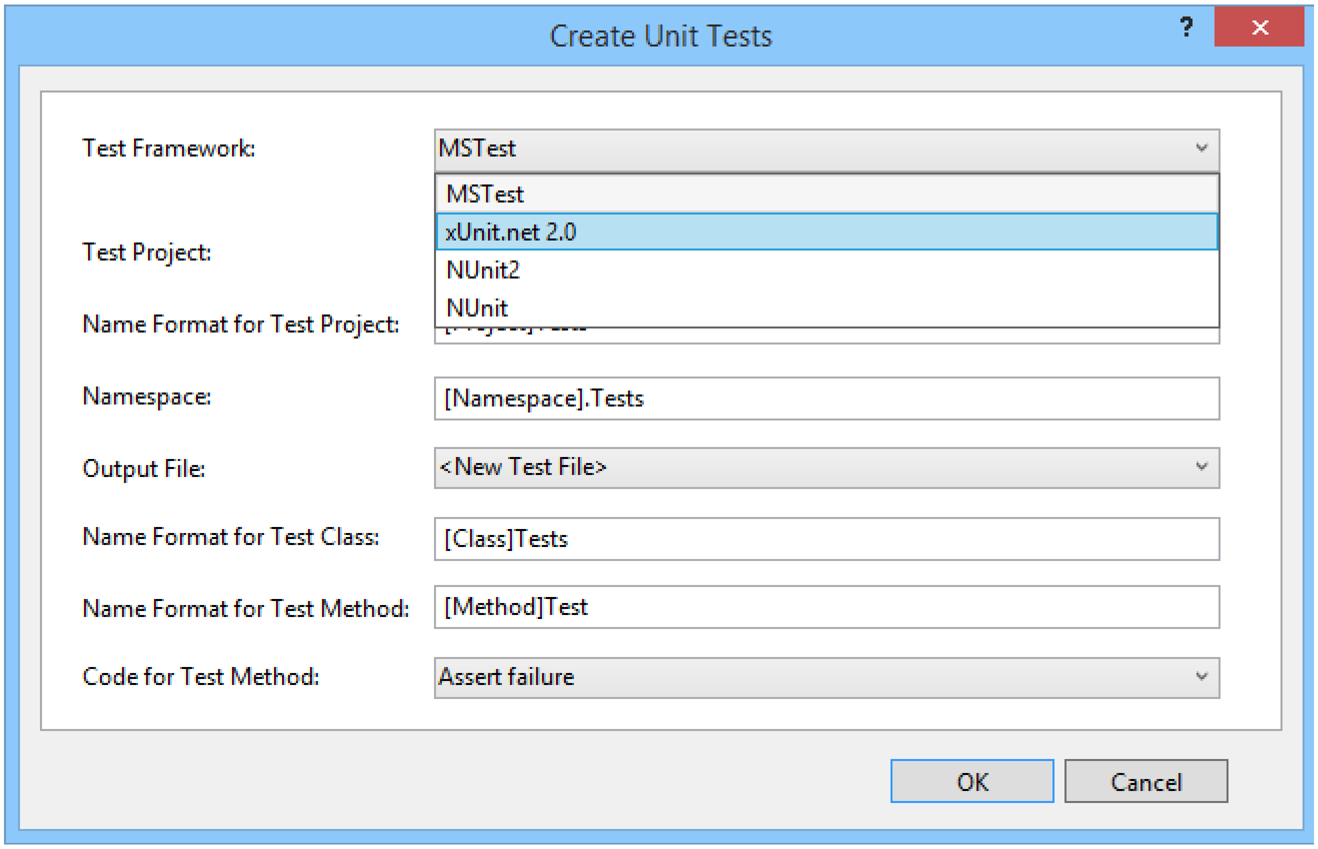
2. *Shims* use runtime instrumentation to divert calls to a target method to a substitute shim method for non-virtual methods.

In both approaches, you use the generated delegates of calls to the dependency method to specify the behavior that you want in the test method.

Learn more about isolating unit test methods with Microsoft Fakes.

Q: Can I use other unit test frameworks to create unit tests?

A: Yes, follow these steps to find and install other frameworks. After you restart Visual Studio, reopen your solution to create your unit tests, and then select your installed frameworks here:


Create Unit Tests

Test Framework: MSTest
MSTest
xUnit.net 2.0
NUnit2
NUnit

Test Project:

Name Format for Test Project:

Namespace: [Namespace].Tests

Output File: <New Test File>

Name Format for Test Class: [Class]Tests

Name Format for Test Method: [Method]Test

Code for Test Method: Assert failure

Your unit test stubs will be created using the selected framework.

Generate unit tests for your code with IntelliTest

Visual Studio 2015

Updated: October 5, 2015

IntelliTest explores your .NET code to generate test data and a suite of unit tests. For every statement in the code, a test input is generated that will execute that statement. A case analysis is performed for every conditional branch in the code. For example, if statements, assertions, and all operations that can throw exceptions are analyzed. This analysis is used to generate test data for a parameterized unit test for each of your methods, creating unit tests with high code coverage.

When you run IntelliTest, you can easily see which tests are failing and add any necessary code to fix them. You can select which of the generated tests to save into a test project to provide a regression suite. As you change your code, rerun IntelliTest to keep the generated tests in sync with your code changes.

IntelliTest is available for C# only and does not support x64 configuration.

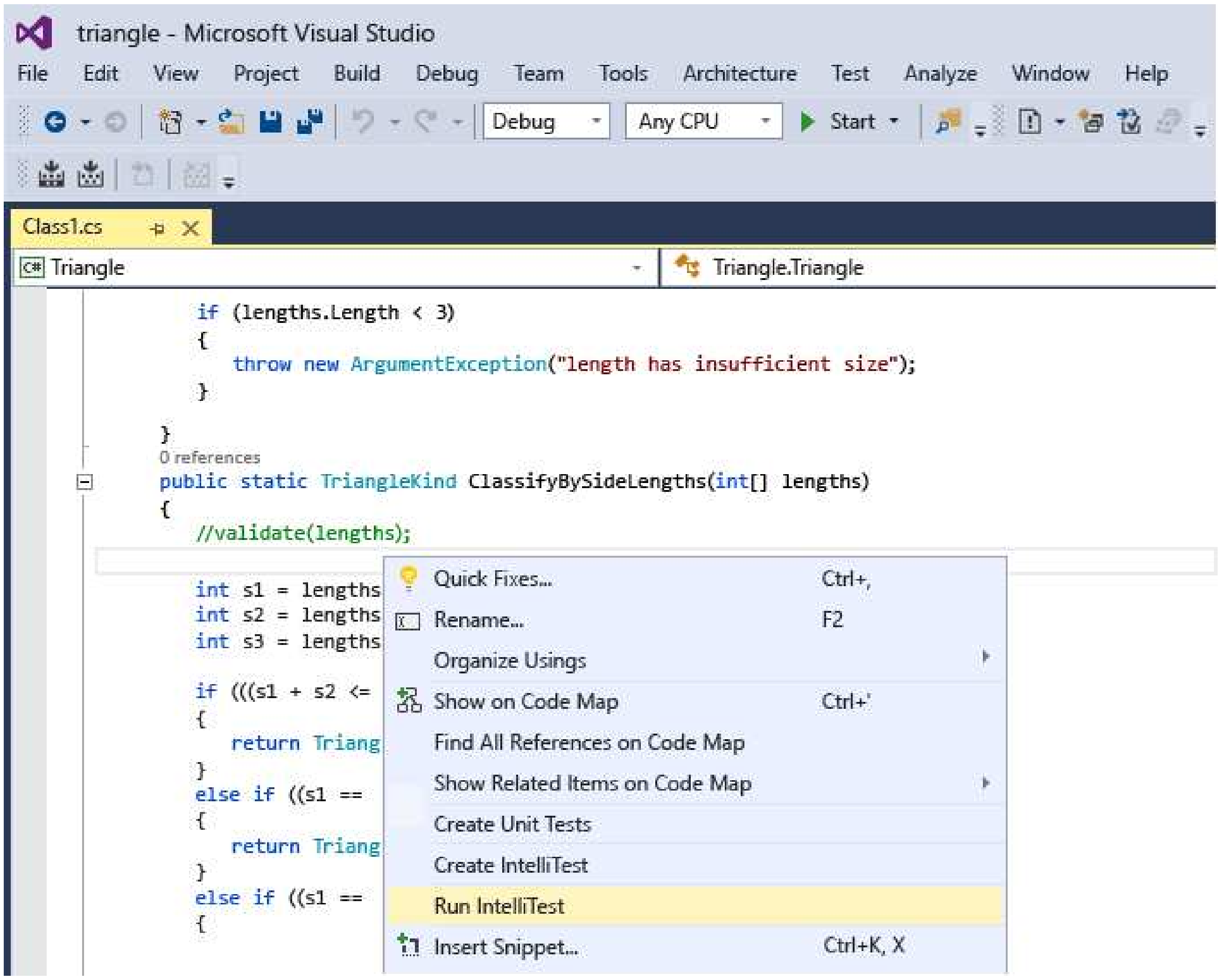
Get started with IntelliTest

You'll need Visual Studio Enterprise.

Explore: Use IntelliTest to explore your code and generate unit tests

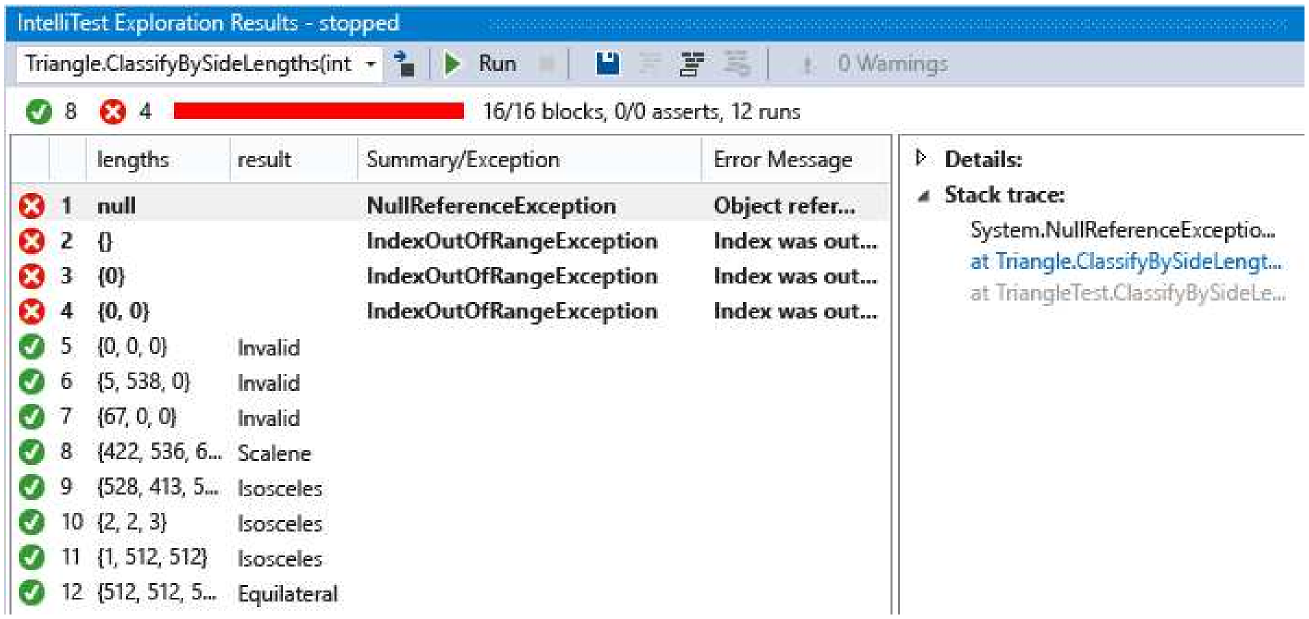
To generate unit tests, your types must be public. Otherwise, create unit tests first before you generate them.

1. Open your solution in Visual Studio. Then open the class file that has methods you want to test.
2. Right-click in a method in your code and choose **Run IntelliTest** to generate unit tests for the code in your method.


triangle - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Debug
Any CPU
Start
Class1.cs
Triangle
Triangle.Triangle
0 references
Quick Fixes... Ctrl+,
Rename... F2
Organize Usings
Show on Code Map Ctrl+'
Find All References on Code Map
Show Related Items on Code Map
Create Unit Tests
Create IntelliTest
Run IntelliTest
Insert Snippet... Ctrl+K, X

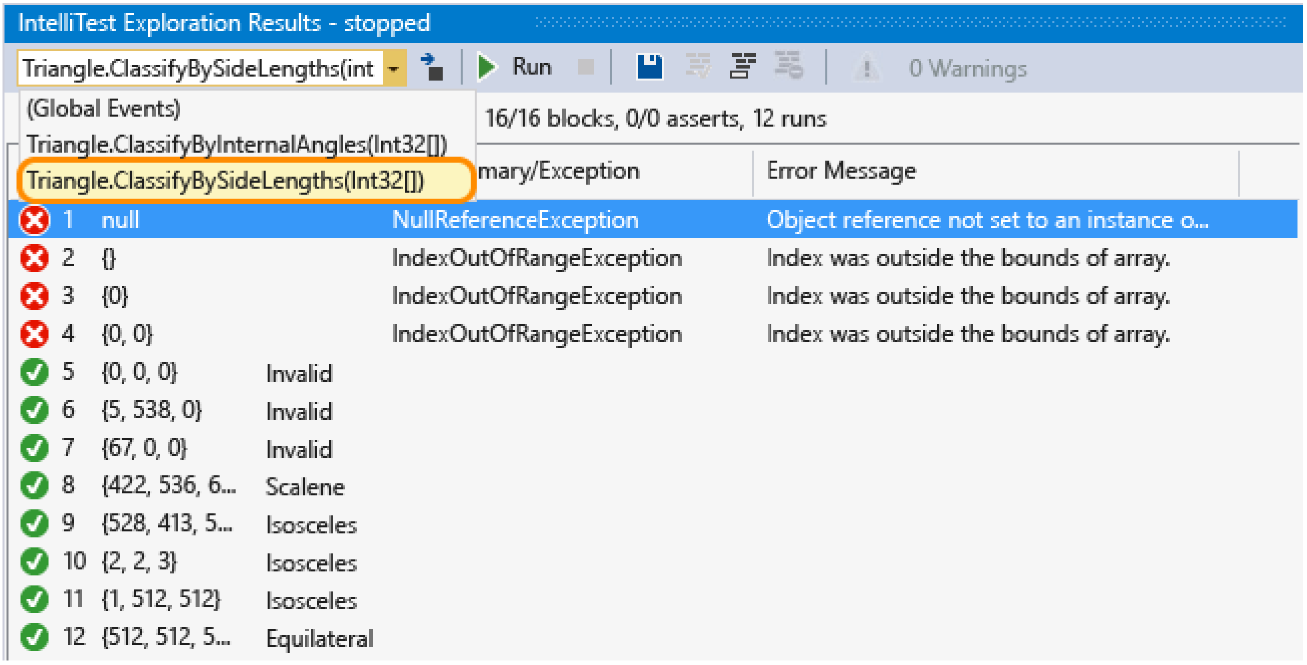
```
        if (lengths.Length < 3)
        {
            throw new ArgumentException("length has insufficient size");
        }

    }
    0 references
    public static TriangleKind ClassifyBySideLengths(int[] lengths)
    {
        //validate(lengths);

        int s1 = lengths
        int s2 = lengths
        int s3 = lengths

        if (((s1 + s2 <=
        {
            return Triang
        }
        else if ((s1 ==
        {
            return Triang
        }
        else if ((s1 ==
        {
```

IntelliTest runs your code many times with different inputs. Each run is represented in the table showing the input test data and the resulting output or exception.


IntelliTest Exploration Results - stopped
Triangle.ClassifyBySideLengths(int
Run
0 Warnings
8
4
16/16 blocks, 0/0 asserts, 12 runs
Details:
Stack trace:
System.NullReferenceExceptio...
at Triangle.ClassifyBySideLengt...
at TriangleTest.ClassifyBySideLe...

		lengths	result	Summary/Exception	Error Message
✗	1	null		NullReferenceException	Object refer...
✗	2	{}		IndexOutOfRangeException	Index was out...
✗	3	{0}		IndexOutOfRangeException	Index was out...
✗	4	{0, 0}		IndexOutOfRangeException	Index was out...
✓	5	{0, 0, 0}	Invalid		
✓	6	{5, 538, 0}	Invalid		
✓	7	{67, 0, 0}	Invalid		
✓	8	{422, 536, 6...	Scalene		
✓	9	{528, 413, 5...	Isosceles		
✓	10	{2, 2, 3}	Isosceles		
✓	11	{1, 512, 512}	Isosceles		
✓	12	{512, 512, 5...	Equilateral		

To generate unit tests for all the public methods in a class, simply right-click in the class rather than a specific method. Then choose **Run IntelliTest**. Use the drop-down list in the Exploration Results window to display the unit tests and the input data for each method in the class.


IntelliTest Exploration Results - stopped
Triangle.ClassifyBySideLengths(int
Run
0 Warnings
(Global Events)
Triangle.ClassifyByInternalAngles(Int32[])
Triangle.ClassifyBySideLengths(Int32[])

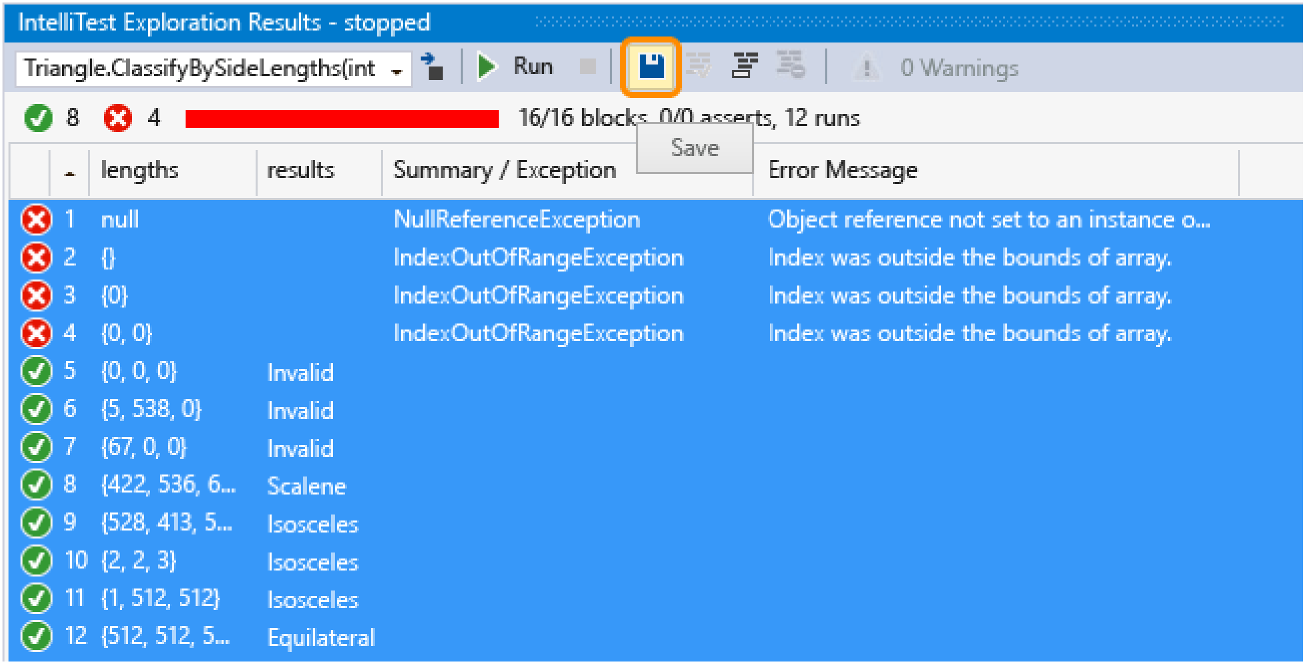
16/16 blocks, 0/0 asserts, 12 runs

			mary/Exception	Error Message
✗ 1	null		NullReferenceException	Object reference not set to an instance o...
✗ 2	{}		IndexOutOfRangeException	Index was outside the bounds of array.
✗ 3	{0}		IndexOutOfRangeException	Index was outside the bounds of array.
✗ 4	{0, 0}		IndexOutOfRangeException	Index was outside the bounds of array.
✓ 5	{0, 0, 0}	Invalid		
✓ 6	{5, 538, 0}	Invalid		
✓ 7	{67, 0, 0}	Invalid		
✓ 8	{422, 536, 6...	Scalene		
✓ 9	{528, 413, 5...	Isosceles		
✓ 10	{2, 2, 3}	Isosceles		
✓ 11	{1, 512, 512}	Isosceles		
✓ 12	{512, 512, 5...	Equilateral		

For tests that pass, check that the reported results in the result column match your expectations for your code. For tests that fail, fix your code as appropriate. Then rerun IntelliTest to validate the fixes.

Persist: Save the unit tests as a regression suite

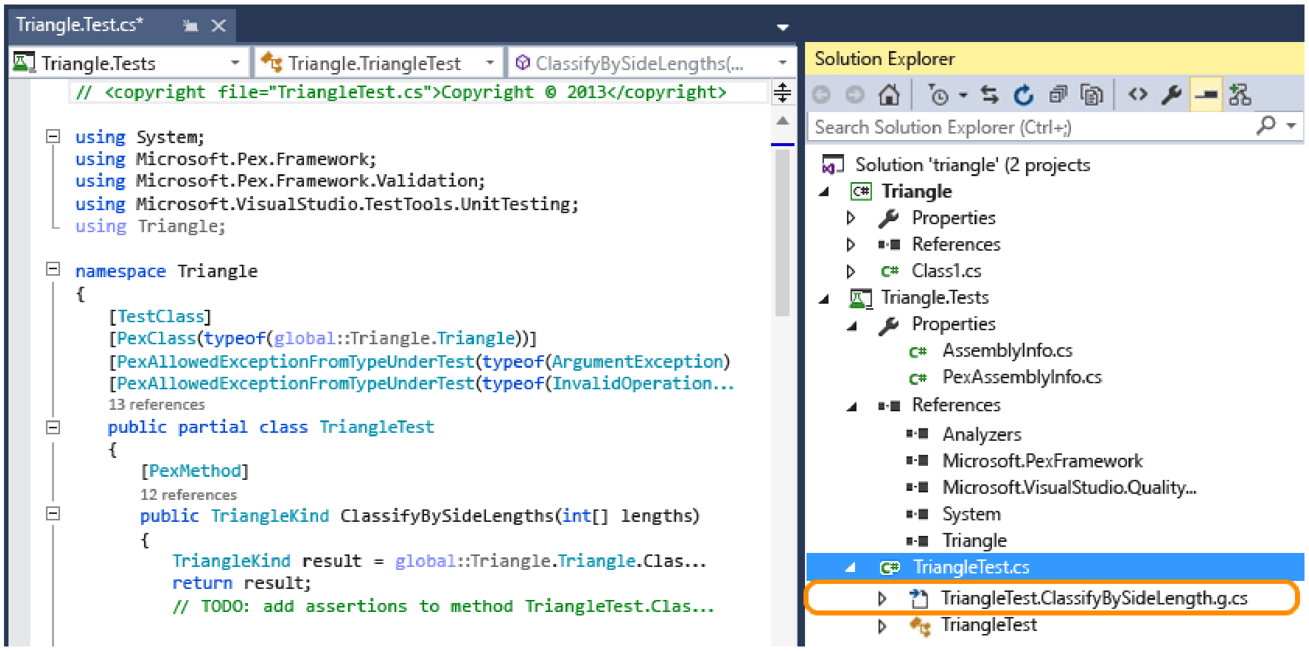
1. Select the data rows that you want to save with the parameterized unit test into a test project.


IntelliTest Exploration Results - stopped
Triangle.ClassifyBySideLengths(int
Run
0 Warnings
✓ 8 ✗ 4
Save

16/16 blocks, 0/0 asserts, 12 runs

	lengths	results	Summary / Exception	Error Message
✗ 1	null		NullReferenceException	Object reference not set to an instance o...
✗ 2	{}		IndexOutOfRangeException	Index was outside the bounds of array.
✗ 3	{0}		IndexOutOfRangeException	Index was outside the bounds of array.
✗ 4	{0, 0}		IndexOutOfRangeException	Index was outside the bounds of array.
✓ 5	{0, 0, 0}	Invalid		
✓ 6	{5, 538, 0}	Invalid		
✓ 7	{67, 0, 0}	Invalid		
✓ 8	{422, 536, 6...	Scalene		
✓ 9	{528, 413, 5...	Isosceles		
✓ 10	{2, 2, 3}	Isosceles		
✓ 11	{1, 512, 512}	Isosceles		
✓ 12	{512, 512, 5...	Equilateral		

You can view the test project and the parameterized unit test that has been created - the individual unit tests, corresponding to each of the rows, are saved in the .g.cs file in the test project, and a parameterized unit test is saved in its corresponding .cs file. You can run the unit tests and view the results from Test Explorer just as you would for any unit tests that you created manually.



```
// <copyright file="TriangleTest.cs">Copyright © 2013</copyright>

using System;
using Microsoft.Pex.Framework;
using Microsoft.Pex.Framework.Validation;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Triangle;

namespace Triangle
{
    [TestClass]
    [PexClass(typeof(global::Triangle.Triangle))]
    [PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException)
    [PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperation...
    13 references
    public partial class TriangleTest
    {
        [PexMethod]
        12 references
        public TriangleKind ClassifyBySideLengths(int[] lengths)
        {
            TriangleKind result = global::Triangle.Triangle.Clas...
            return result;
            // TODO: add assertions to method TriangleTest.Clas...
```

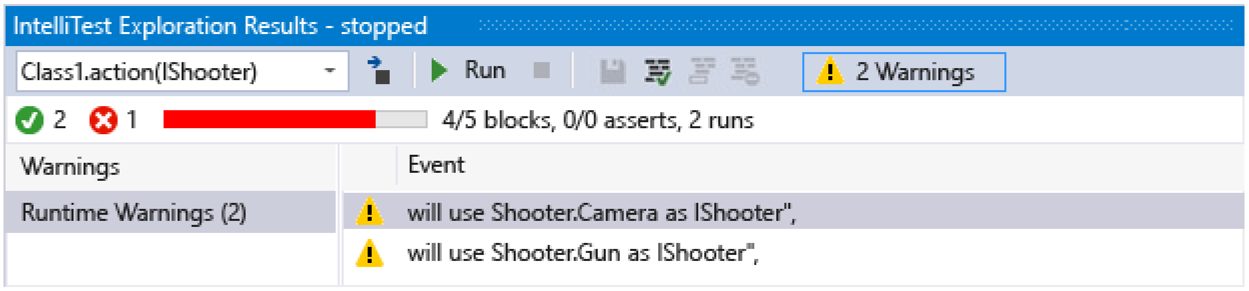
Any necessary references are also added to the test project.

If the method code changes, rerun IntelliTest to keep the unit tests in sync with the changes.

Assist: Use IntelliTest to focus code exploration

1. If you have more complex code, IntelliTest assists you with focusing exploration of your code. For example, if you have a method that has an interface as a parameter, and there is more than one class that implements that interface, IntelliTest discovers those classes and reports a warning.

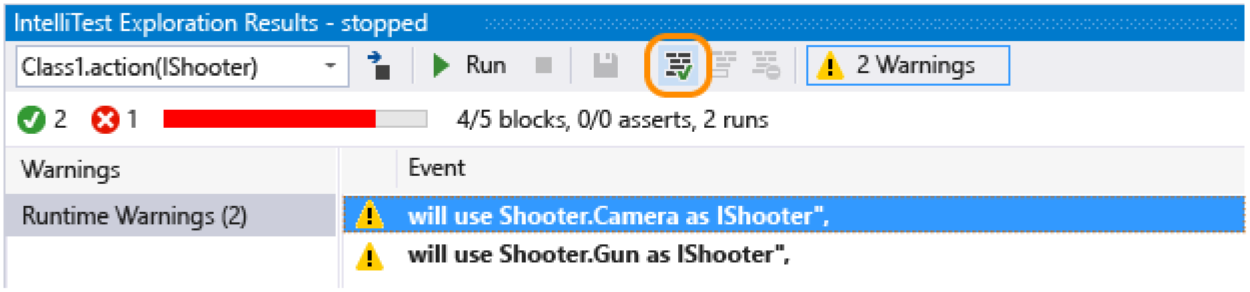
   View the warnings to decide what you want to do.



4/5 blocks, 0/0 asserts, 2 runs

Warnings	Event
Runtime Warnings (2)	will use Shooter.Camera as IShooter",
	will use Shooter.Gun as IShooter",

2. After you investigate the code and understand what you want to test, you can fix the warning to choose which classes to use to test the interface.



IntelliTest Exploration Results - stopped

Class1.action(IShooter) | Run | 2 Warnings

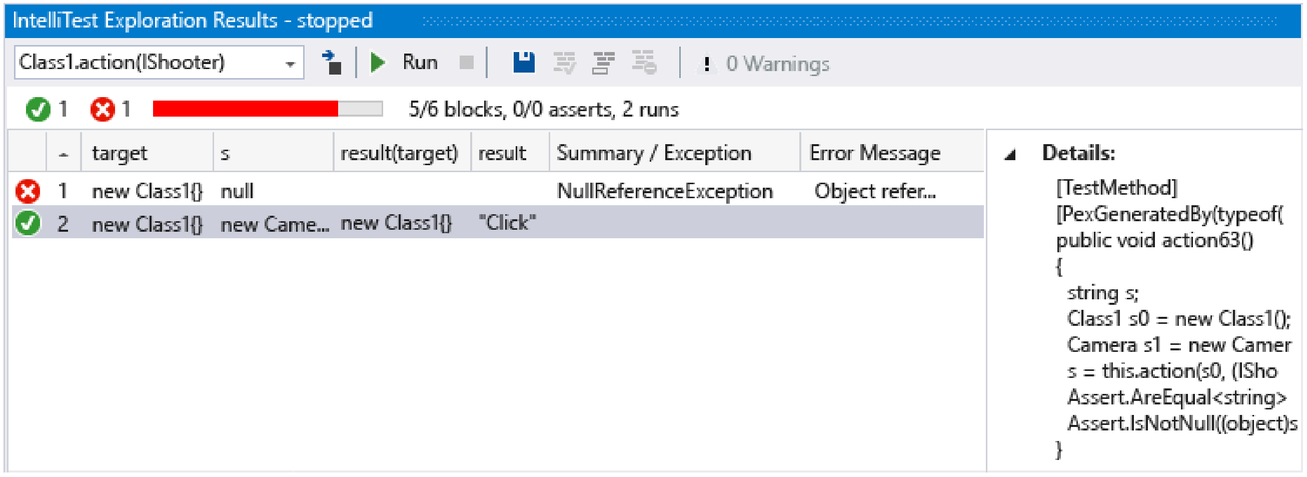
✓ 2 ✗ 1 4/5 blocks, 0/0 asserts, 2 runs

Warnings	Event
Runtime Warnings (2)	will use Shooter.Camera as IShooter",
	will use Shooter.Gun as IShooter",

This choice is added into the PexAssemblyInfo.cs file.

```
[assembly: PexUseType(typeof(Camera))]
```

3. Now you can rerun IntelliTest to generate a parameterized unit test and test data just using the class that you fixed.



IntelliTest Exploration Results - stopped

Class1.action(IShooter) | Run | 0 Warnings

✓ 1 ✗ 1 5/6 blocks, 0/0 asserts, 2 runs

		target	s	result(target)	result	Summary / Exception	Error Message
✗	1	new Class1{}	null			NullReferenceException	Object refer...
✓	2	new Class1{}	new Came...	new Class1{}	"Click"		

Details:

```
[TestMethod]
[PexGeneratedBy(typeof(
public void action63()
{
  string s;
  Class1 s0 = new Class1();
  Camera s1 = new Camer
  s = this.action(s0, (ISho
  Assert.AreEqual<string>
  Assert.IsNotNull((object)s
}
```

Specify: Use IntelliTest to validate correctness properties that you specify in code

Specify the general relationship between inputs and outputs that you want the generated unit tests to validate. This specification is encapsulated in a method that looks like a test method but is universally quantified. This is the parameterized unit test method, and any assertions you make must hold for all possible input values that IntelliTest can generate.

Q & A

Q: Can you use IntelliTest for unmanaged code?

A: No, IntelliTest only works with managed code.

Q: When does a generated test pass or fail?

A: It passes like any other unit test if no exceptions occur. It fails if any assertion fails, or if the code under test throws an unhandled exception.

If you have a test that can pass if certain exceptions are thrown, you can set one of the following attributes based on your requirements at the test method, test class or assembly level:

- **PexAllowedExceptionAttribute**
- **PexAllowedExceptionFromTypeAttribute**
- **PexAllowedExceptionFromTypeUnderTestAttribute**
- **PexAllowedExceptionFromAssemblyAttribute**

Q: Can I add assumptions to the parameterized unit test?

A: Yes, use assumptions to specify which test data is not required for the unit test for a specific method. Use the PexAssume class to add assumptions. For example, you can add an assumption that the lengths variable is not null like this.

```
PexAssume.IsNotNull(lengths);
```

If you add an assumption and rerun IntelliTest, the test data that is no longer relevant will be removed.

Q: Can I add assertions to the parameterized unit test?

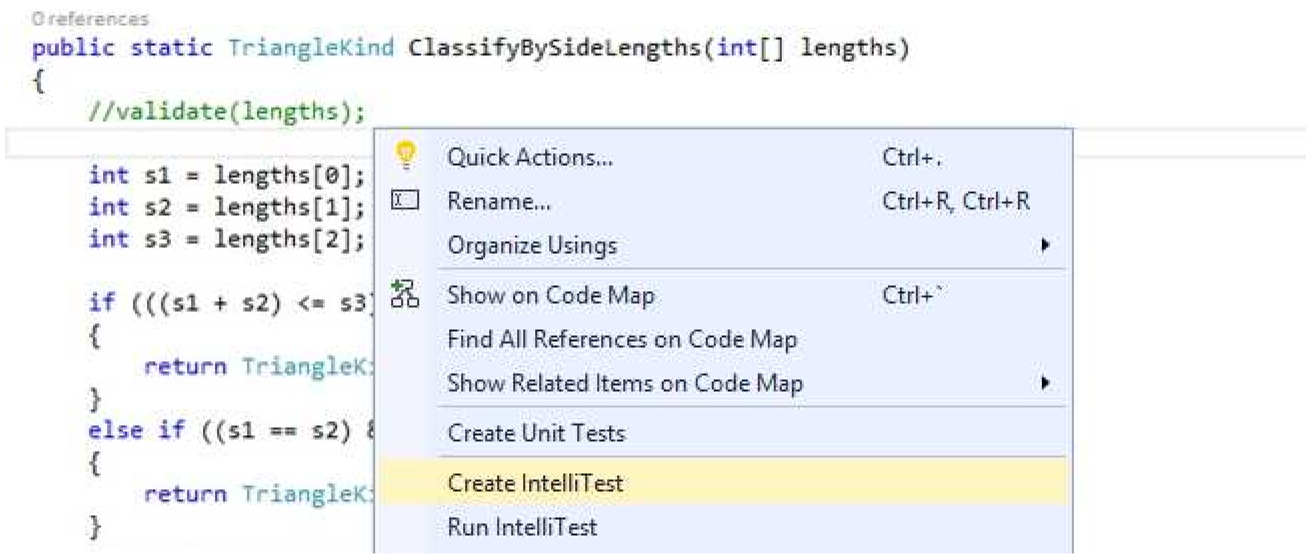
A: Yes, IntelliTest will check that what you are asserting in your statement is in fact correct when it runs the unit tests. Use the PexAssert class or the assertion API that comes with the test framework to add assertions. For example, you can add an assertion that two variables are equal.

```
PexAssert.AreEqual(a, b);
```

If you add an assertion and rerun IntelliTest, it will check that your assertion is valid and the test fails if it is not.

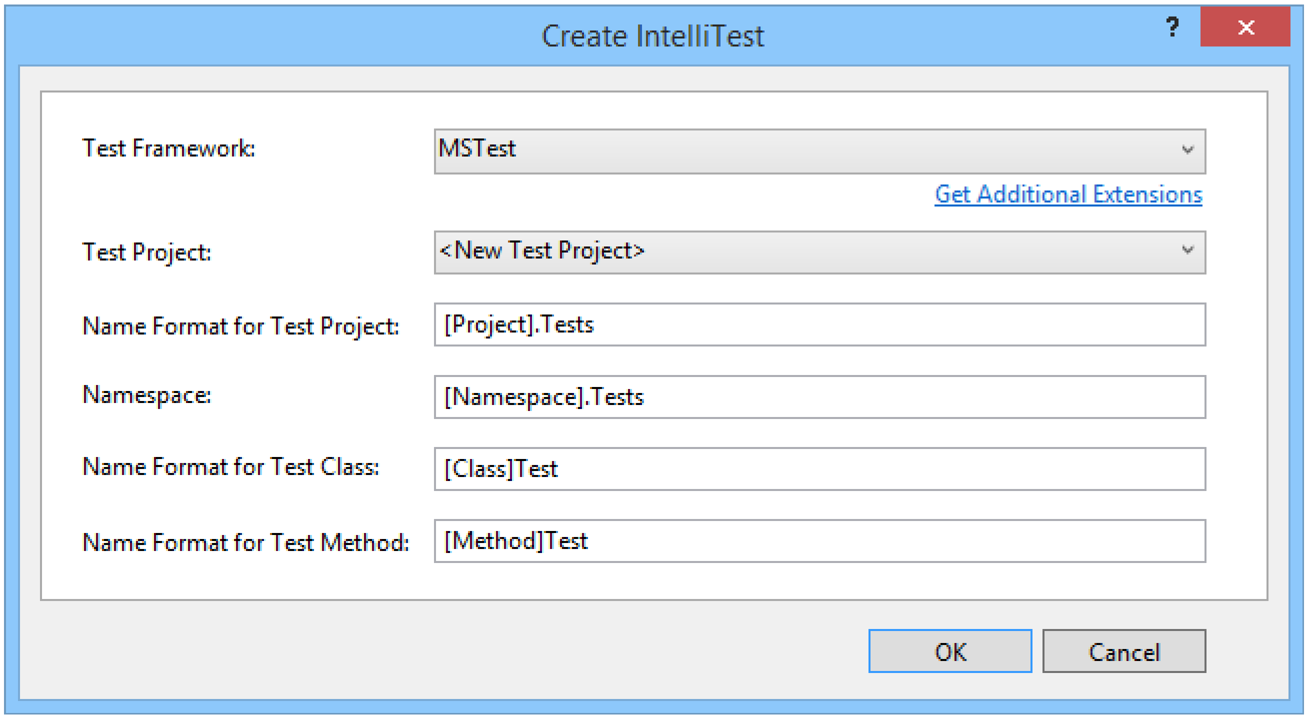
Q: Can I generate parameterized unit tests without running IntelliTest first?

A: Yes, right-click in the class or method, then choose **Create IntelliTest**.



```
0 references
public static TriangleKind ClassifyBySideLengths(int[] lengths)
{
    //validate(lengths);

    int s1 = lengths[0];
    int s2 = lengths[1];
    int s3 = lengths[2];

    if (((s1 + s2) <= s3
    {
        return TriangleK
    }
    else if ((s1 == s2) &
    {
        return TriangleK
    }
```

Accept the default format to generate your tests, or change how your project and tests are named. You can create a new test project or save your tests to an existing project.

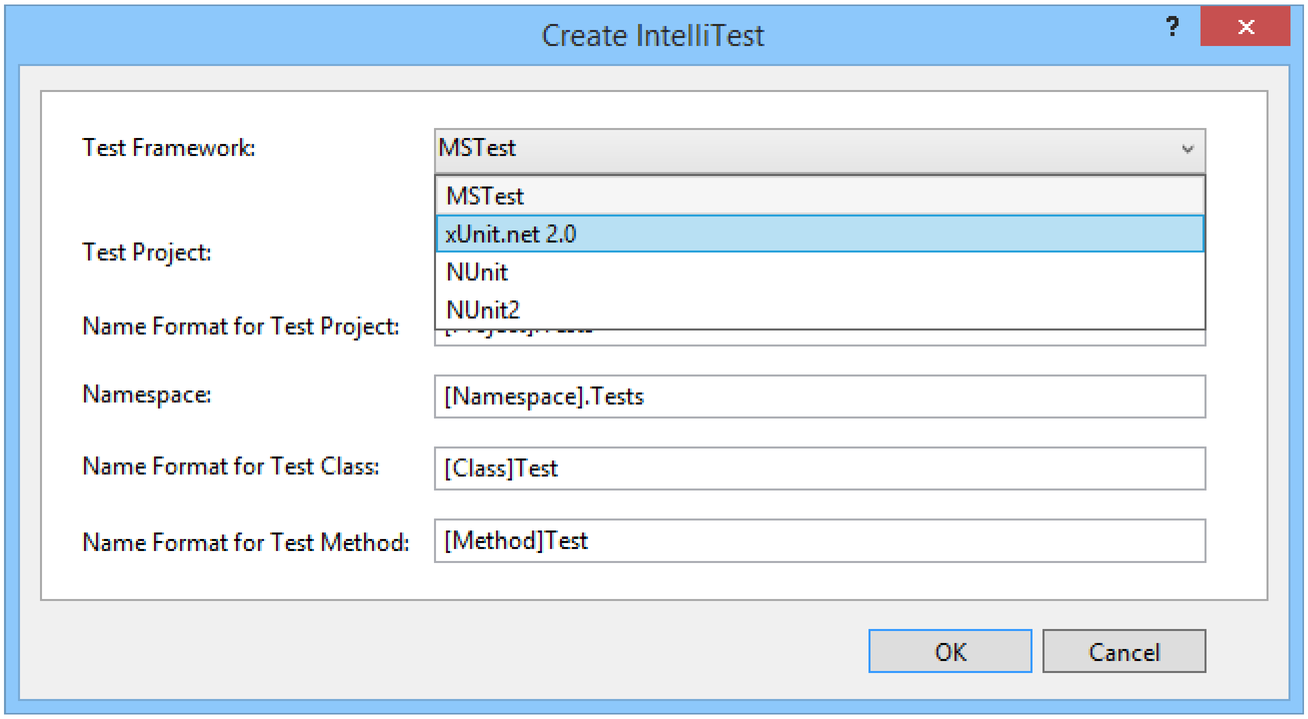


Create IntelliTest

Test Framework:	MSTest
Test Project:	<New Test Project>
Name Format for Test Project:	[Project].Tests
Namespace:	[Namespace].Tests
Name Format for Test Class:	[Class]Test
Name Format for Test Method:	[Method]Test

Q: Can I use other unit test frameworks with IntelliTest?

A: Yes, follow these steps to find and install other frameworks. After you restart Visual Studio and reopen your solution, right-click in the class or method, then choose **Create IntelliTest**. Select your installed framework here:



Test Framework: MSTest

MSTest
xUnit.net 2.0
NUnit
NUnit2

Test Project:

Name Format for Test Project:

Namespace: [Namespace].Tests

Name Format for Test Class: [Class]Test

Name Format for Test Method: [Method]Test

Then run IntelliTest to generate individual unit tests in their corresponding .g.cs files.

Q: Can I learn more about how the tests are generated?

A: Yes, to get a high-level overview, read this blog post.

Run unit tests with Test Explorer

Visual Studio 2015

Use Test Explorer to run unit tests from Visual Studio or third-party unit test projects, group tests into categories, filter the test list, and create, save, and run playlists of tests. You can also debug tests and analyze test performance and code coverage.

Contents

Unit test frameworks and test projects

Run tests in Test Explorer

View test results

Group and filter the test list

Create custom playlists

Debug and analyze unit tests

External resources

Unit test frameworks and test projects

Visual Studio includes the Microsoft unit testing frameworks for both managed and native code. However, Test Explorer can also run any unit test framework that has implemented a Test Explorer adapter. For more information about installing third-party unit test frameworks, see Install third-party unit test frameworks

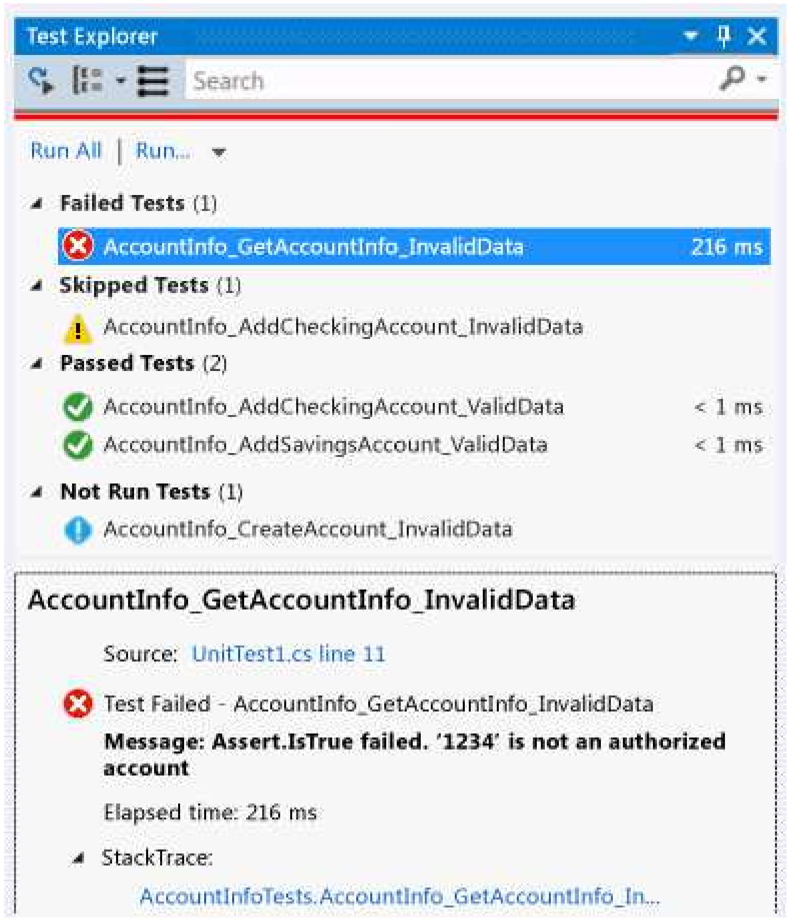
Test Explorer can run tests from multiple test projects in a solution and from test classes that are part of the production code projects. Test projects can use different unit test frameworks. When the code under test is written for the .NET Framework, the test project can be written in any language that also targets the .NET Framework, regardless of the language of the target code. Native C/C++ code projects must be tested by using a C++ unit test framework.

Contents

Run tests in Test Explorer

Run tests|Run tests after every build

When you build the test project, the tests appear in Test Explorer. If Test Explorer is not visible, choose **Test** on the Visual Studio menu, choose **Windows**, and then choose **Test Explorer**.

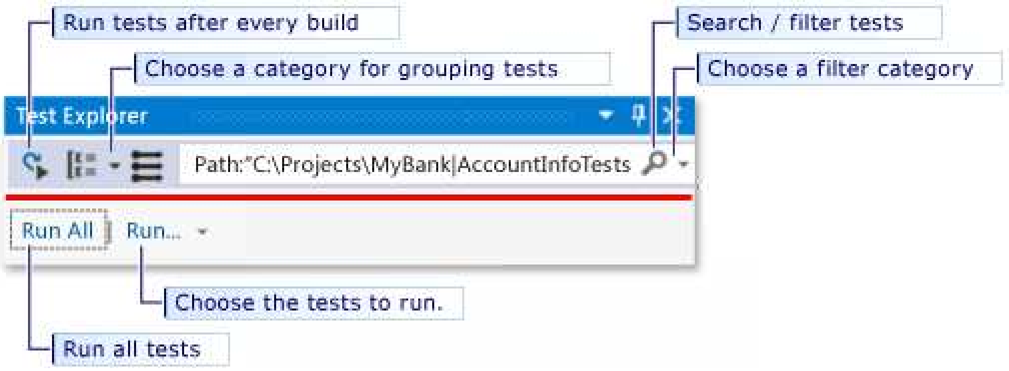


Failed Tests (1)
AccountInfo_GetAccountInfo_InvalidData 216 ms
Skipped Tests (1)
AccountInfo_AddCheckingAccount_InvalidData
Passed Tests (2)
AccountInfo_AddCheckingAccount_ValidData < 1 ms
AccountInfo_AddSavingsAccount_ValidData < 1 ms
Not Run Tests (1)
AccountInfo_CreateAccount_InvalidData

AccountInfo_GetAccountInfo_InvalidData
Source: UnitTest1.cs line 11
Test Failed - AccountInfo_GetAccountInfo_InvalidData
Message: Assert.IsTrue failed. '1234' is not an authorized account
Elapsed time: 216 ms
StackTrace:
AccountInfoTests.AccountInfo_GetAccountInfo_In...

As you run, write, and rerun your tests, Test Explorer displays the results in default groups of **Failed Tests**, **Passed Tests**, **Skipped Tests** and **Not Run Tests**. You can change the way Test Explorer groups your tests.

You can perform much of the work of finding, organizing and running tests from the Test Explorer toolbar.



Contents

Run tests

You can run all the tests in the solution, all the tests in a group, or a set of tests that you select. Do one of the following:

- To run all the tests in a solution, choose **Run All**.
- To run all the tests in a default group, choose **Run...** and then choose the group on the menu.
- Select the individual tests that you want to run, open the context menu for a selected test and then choose **Run**

Selected Tests.

- If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution with the toggle button on the toolbar. This can noticeably reduce the time taken to run all the tests.

The pass/fail bar at the top of the Test Explorer window is animated as the tests run. At the conclusion of the test run, the pass/fail bar turns green if all tests passed or turns red if any test failed.

Contents

Run tests after every build

Warning

Running unit tests after every build is supported in Visual Studio Enterprise.

To run your unit tests after each local build, choose **Test** on the standard menu, and then choose **Run Tests After Build** on the Test Explorer toolbar.

Contents

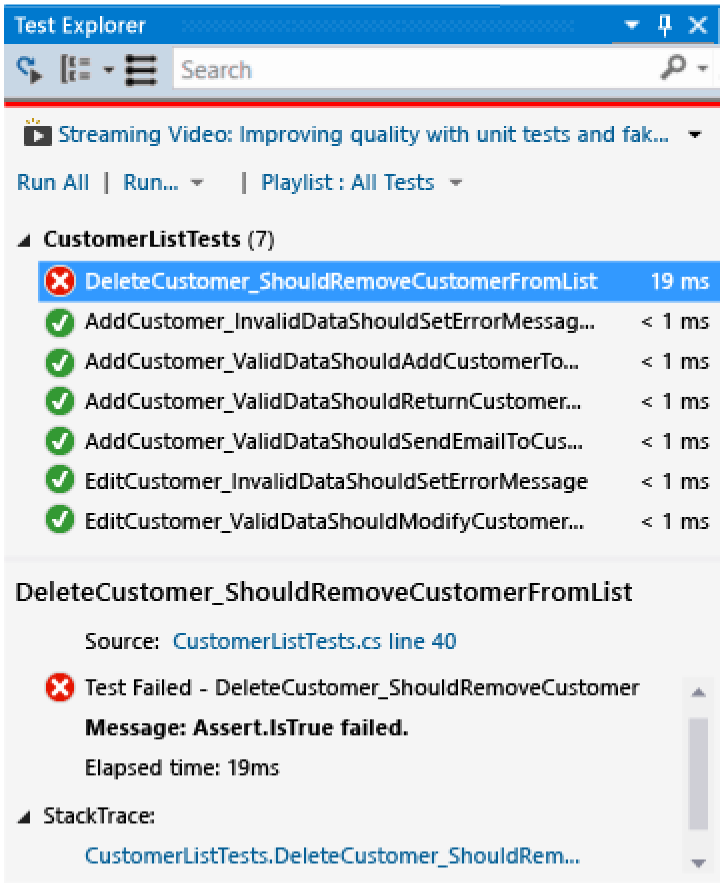
View test results

View test details|View the source code of a test method

As you run, write, and rerun your tests, Test Explorer displays the results in groups of **Failed Tests**, **Passed Tests**, **Skipped Tests** and **Not Run Tests**. The details pane at the bottom of Test Explorer displays a summary of the test run.

View test details

To view the details of an individual test, select the test.


Test Explorer
Search
Streaming Video: Improving quality with unit tests and fak...
Run All | Run... | Playlist : All Tests

CustomerListTests (7)

DeleteCustomer_ShouldRemoveCustomerFromList 19 ms
AddCustomer_InvalidDataShouldSetErrorMessag... < 1 ms
AddCustomer_ValidDataShouldAddCustomerTo... < 1 ms
AddCustomer_ValidDataShouldReturnCustomer... < 1 ms
AddCustomer_ValidDataShouldSendEmailToCus... < 1 ms
EditCustomer_InvalidDataShouldSetErrorMessage < 1 ms
EditCustomer_ValidDataShouldModifyCustomer... < 1 ms

DeleteCustomer_ShouldRemoveCustomerFromList

Source: CustomerListTests.cs line 40

Test Failed - DeleteCustomer_ShouldRemoveCustomer

Message: Assert.IsTrue failed.

Elapsed time: 19ms

StackTrace:

CustomerListTests.DeleteCustomer_ShouldRem...

The test details pane displays the following information:

- The source file name and the line number of the test method.
- The status of the test.
- The elapsed time that the test method took to run.

If the test fails, the details pane also displays:

- The message returned by the unit test framework for the test.
- The stack trace at the time the test failed.

Contents

View the source code of a test method

To display the source code for a test method in the Visual Studio editor, select the test and then choose **Open Test** on the context menu (Keyboard: F12).

Contents

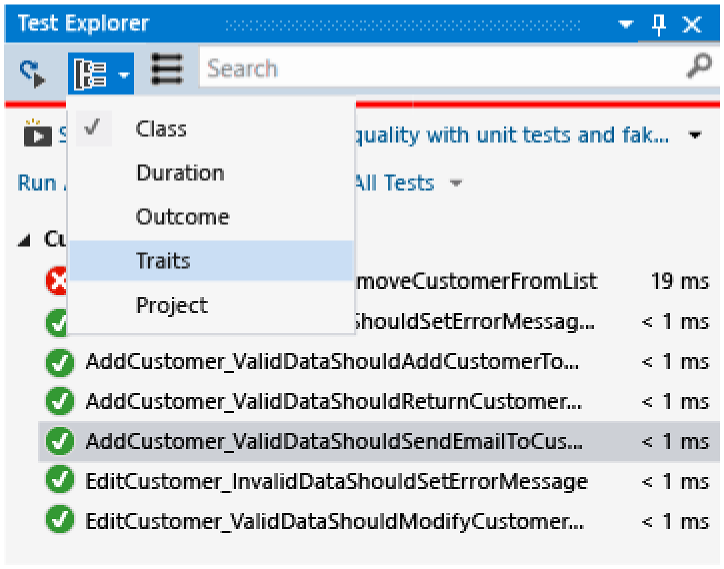
Group and filter the test list

Grouping the test list|Group by traits|Search and filter the test list

Test Explorer lets you group your tests into predefined categories. Most unit test frameworks that run in Test Explorer let you define your own categories and category/value pairs to group your tests. You can also filter the list of tests by matching strings against test properties.

Grouping the test list

To change the way that tests are organized, choose the down arrow next to the **Group By** button and select a new grouping criteria.


Test Explorer
Search
Class
Duration
Outcome
Traits
Project
quality with unit tests and fak...
Run
All Tests
moveCustomerFromList 19 ms
ShouldSetErrorMessag... < 1 ms
AddCustomer_ValidDataShouldAddCustomerTo... < 1 ms
AddCustomer_ValidDataShouldReturnCustomer... < 1 ms
AddCustomer_ValidDataShouldSendEmailToCus... < 1 ms
EditCustomer_InvalidDataShouldSetErrorMessage < 1 ms
EditCustomer_ValidDataShouldModifyCustomer... < 1 ms

Test Explorer groups

Group	Description
Duration	Groups test by execution time: **Fast**, **Medium**, and **Slow**.
Outcome	Groups tests by execution results: **Failed Tests**, **Skipped Tests**, **Passed Tests**.
Traits	Groups test by category/value pairs that you define. The syntax to specify trait categories and values is defined by the unit test framework.
Project	Groups test by the name of the projects.

Contents

Group by traits

A trait is usually a category name/value pair, but it can also be a single category. Traits can be assigned to methods that are identified as a test method by the unit test framework. A unit test framework can define trait categories. You can add values to the trait categories to define your own category name/value pairs. The syntax to specify trait categories and values is defined by the unit test framework.

Traits in the Microsoft Unit Testing Framework for Managed Code

In the Microsoft unit test framework for managed apps, you define a trait name/ value pair in a TestPropertyAttribute attribute. The test framework also contains these predefined traits:

Trait	Description
OwnerAttribute	The Owner category is defined by the unit test framework and requires you to provide a string value of the owner.
PriorityAttribute	The Priority category is defined by the unit test framework and requires you to provide an integer value of the priority.
TestCategoryAttribute	The TestCategory attribute enables you to provide a category without a value. A category defined by the TestCategory attribute can also be the category of a TestProperty attribute.
TestPropertyAttribute	The TestProperty attribute enables you to define trait category/value pair.

Traits in the Microsoft Unit Testing Framework for C++

To define a trait, use the `TEST_METHOD_ATTRIBUTE` macro. For example, to define a trait named `TEST_MY_TRAIT`:

C++

```
#define TEST_MY_TRAIT(traitValue) TEST_METHOD_ATTRIBUTE(L"MyTrait", traitValue)
```

To use the defined trait in your unit tests:

```
BEGIN_TEST_METHOD_ATTRIBUTE(Method1)
    TEST_OWNER(L"OwnerName")
    TEST_PRIORITY(1)
    TEST_MY_TRAIT(L"thisTraitValue")
END_TEST_METHOD_ATTRIBUTE()

TEST_METHOD(Method1)
{
    Logger::WriteMessage("In Method1");
    Assert::AreEqual(0, 0);
}
```

C++ trait attribute macros

Macro	Description
`TEST_METHOD_ATTRIBUTE(attributeName, attributeValue)`	Use the TEST_METHOD_ATTRIBUTE macro to define a trait.
`TEST_OWNER(ownerAlias)`	Use the predefined Owner trait to specify an owner of the test method.
`TEST_PRIORITY(priority)`	Use the predefined Priority trait to assign relative priorities to your test methods.

Contents

Search and filter the test list

You can use Test Explorer filters to limit the test methods in your projects that you view and run.

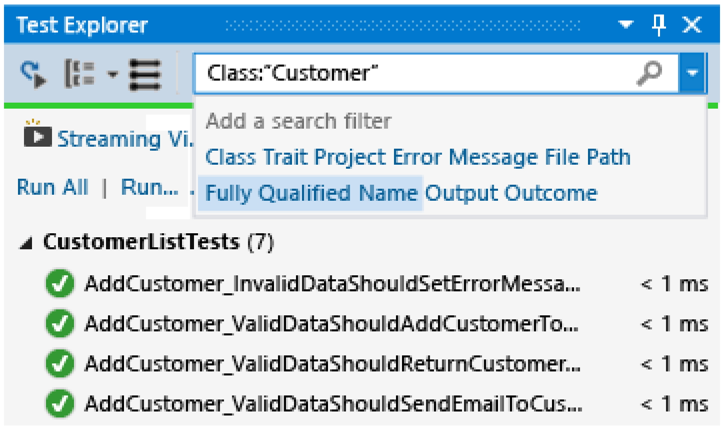
When you type a string in in the Test Explorer search box and choose ENTER, the test list is filtered to display only those tests whose fully qualified names contain the string.

To filter by a different criteria:

1. Open the drop-down list to the right of the search box.

2. Choose a new criteria.

3. Enter the filter value between the quotation marks.


Test Explorer
Class:"Customer"
Add a search filter
Class Trait Project Error Message File Path
Fully Qualified Name Output Outcome
Streaming Vi...
Run All | Run...
CustomerListTests (7)
AddCustomer_InvalidDataShouldSetErrorMessa... < 1 ms
AddCustomer_ValidDataShouldAddCustomerTo... < 1 ms
AddCustomer_ValidDataShouldReturnCustomer... < 1 ms
AddCustomer_ValidDataShouldSendEmailToCus... < 1 ms

Note
Searches are case insensitive and match the specified string to any part of the criteria value.

Qualifier	Description

Trait	Searches both trait category and value for matches. The syntax to specify trait categories and values are defined by the unit test framework.
Project	Searches the test project names for matches.
Error Message	Searches the user-defined error messages returned by failed asserts for matches.
File Path	Searches the fully qualified file name of test source files for matches.
Fully Qualified Name	Searches the fully qualified file name of test namespaces, classes, and methods for matches.
Output	Searches the user-defined error messages that are written to standard output (stdout) or standard error (stderr). The syntax to specify output messages are defined by the unit test framework.
Outcome	Searches the Test Explorer category names for matches: **Failed Tests**, **Skipped Tests**, **Passed Tests**.

To exclude a subset of the results of a filter, use the following syntax:

```
FilterName:"Criteria" -FilterName:"SubsetCriteria"
```

For example,

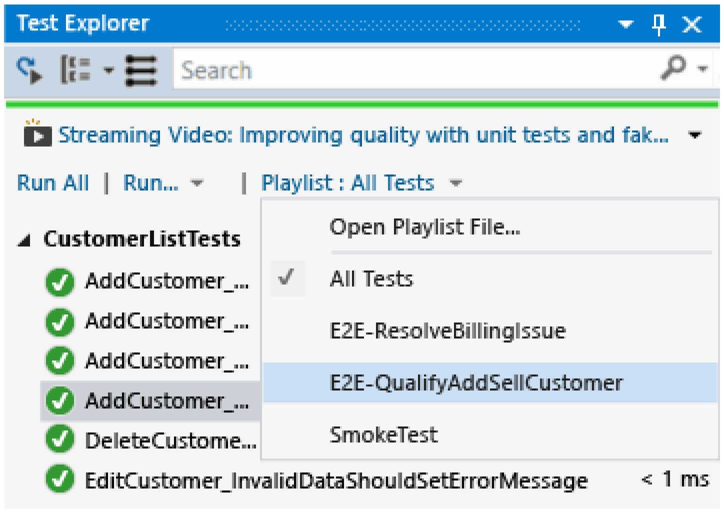
```
FullName:"MyClass" - FullName:"PerfTest"
```

returns all tests that include "MyClass" in their name except those tests that also include "PerfTest" in their name.

Contents

Create custom playlists

You can create and save a list of tests that you want to run or view as a group. When you select a playlist, the tests in the list are displayed Test Explorer. You can add a test to more than one playlist, and all tests in your project are available when you choose the default **All Tests** playlist.


Test Explorer
Search
Streaming Video: Improving quality with unit tests and fak...
Run All | Run... | Playlist : All Tests
CustomerListTests
AddCustomer_...
AddCustomer_...
AddCustomer_...
AddCustomer_...
DeleteCustome...
EditCustomer_InvalidDataShouldSetErrorMessage < 1 ms
Open Playlist File...
All Tests
E2E-ResolveBillingIssue
E2E-QualifyAddSellCustomer
SmokeTest

To create a playlist, choose one or more tests in Test Explorer. On the context menu, choose **Add to Playlist**, **NewPlaylist**. Save the file with the name and location that you specify in the **Create New Playlist** dialog box.

To add tests to a playlist, choose one or more tests in Test Explorer. On the context menu, choose **Add to Playlist**, and then choose the playlist that you want to add the tests to.

To open a playlist, choose Test, Playlist from the Visual Studio menu, and either choose from the list of recently used playlists, or choose Open Playlist to specify the name and location of the playlist.

If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution with the ☰ toggle button on the toolbar. This can noticeably reduce the time taken to run all the tests.

Contents

Debug and analyze unit tests

Debug unit tests|Diagnose test method performance issues|Analyze unit test code coverage

Debug unit tests

You can use Test Explorer to start a debugging session for your tests. Stepping through your code with the Visual Studio debugger seamlessly takes you back and forth between the unit tests and the project under test. To start debugging:

1. In the Visual Studio editor, set a breakpoint in one or more test methods that you want to debug.

> **Note**
>
> Because test methods can run in any order, set breakpoints in all the test methods that you want to debug.

2. In Test Explorer, select the test methods and then choose **Debug Selected Tests** on the context menu.

For more information, about the debugger, see Debugging in Visual Studio.

Contents

Diagnose test method performance issues

To diagnose why a test method is taking too much time, select the method in Test Explorer and then choose Profile on the context menu. See Using Profiling Tools.

Analyze unit test code coverage

Note

Unit test code coverage is available only in Visual Studio Enterprise.

You can determine the amount of your product code that is actually being tested by your unit tests by using the Visual Studio code coverage tool. You can run code coverage on selected tests or on all tests in a solution.

To run code coverage for test methods in a solution:

1. Choose **Tests** on the Visual Studio menu and then choose **Analyze code coverage**.
2. Choose one of the following commands from the sub-menu:
    - **Selected tests** runs the test methods that you have selected in Test Explorer.
    - **All tests** runs all the test methods in the solution.

The Code Coverage Results window displays the percentage of the blocks of product code that were exercised by line, function, class, namespace and module.

For more information, see Using Code Coverage to Determine How Much Code is being Tested.

Contents

External resources

Guidance

Testing for Continuous Delivery with Visual Studio 2012 – Chapter 2: Unit Testing: Testing the Inside

See Also

Unit Test Your Code
Run a unit test as a 64-bit process

Walkthrough: Creating and Running Unit Tests for Windows Store Apps

Visual Studio 2015

Visual Studio includes support for unit testing managed Windows 8.x Store apps and includes unit test library templates for Visual C#, Visual Basic and Visual C++.

Tip
For more information about developing Windows 8.x Store apps, see Getting started with Windows Store apps.

Visual Studio provides the following unit testing functionality:

- Create unit test projects
- Edit the Manifest for the Unit Test Project
- Code the Unit Test
- Run Unit Tests

The following procedures describe the steps to create, run and debug unit tests for managed Windows 8 Windows 8.x Store app.

Prerequisites

Visual Studio

Create unit test projects

To create a unit test project for a Windows Store app

1. From the **File** menu, choose **New Project**.

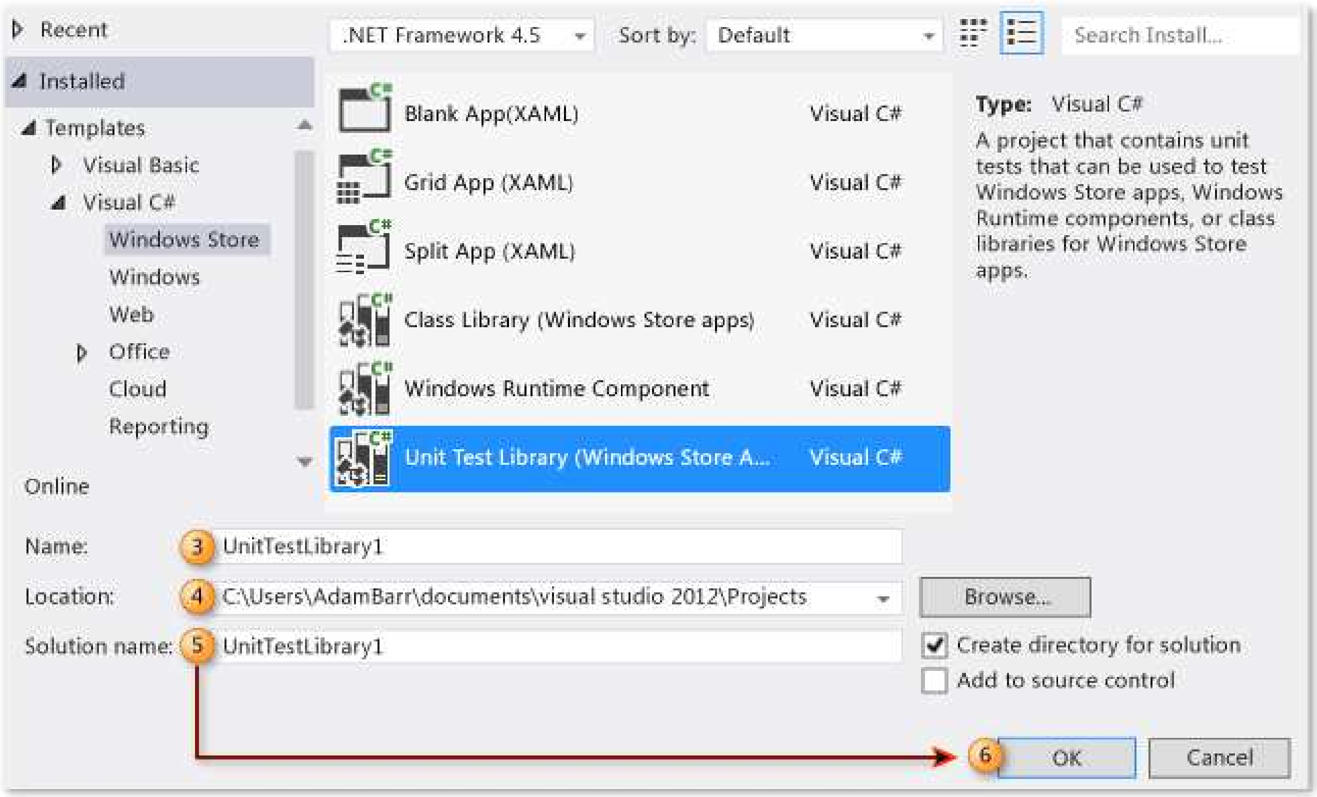
   The New Project dialog displays.

2. Under Templates, choose the programming language you want to create unit test in and then choose the associated Windows 8.x Store unit test library. For example, choose **Visual C#** , then choose **Windows Store**, and then choose **Unit Test Library (Windows Store apps)**.

Note

Visual Studio includes unit test library templates for Visual C#, Visual Basic and Visual C++.

3. (Optional) In the **Name** textbox, enter the name you want to use for the Windows 8.x Storeunit test project.
4. (Optional) Modify the path where you want to create the project by entering it in the **Location** textbox, or choosing the **Browse** button.
5. (Optional) In the **Solution** name textbox, enter that name you want to use for your solution.
6. Leave the **Create directory for solution** option selected and choose the **OK** button.


Recent
Installed
Templates
Visual Basic
Visual C#
Windows Store
Windows
Web
Office
Cloud
Reporting
Online
.NET Framework 4.5
Sort by: Default
Search Install...
Blank App(XAML) Visual C#
Grid App (XAML) Visual C#
Split App (XAML) Visual C#
Class Library (Windows Store apps) Visual C#
Windows Runtime Component Visual C#
Unit Test Library (Windows Store A... Visual C#
Browse...
Add to source control
6
OK
Cancel

Type: Visual C#

A project that contains unit tests that can be used to test Windows Store apps, Windows Runtime components, or class libraries for Windows Store apps.

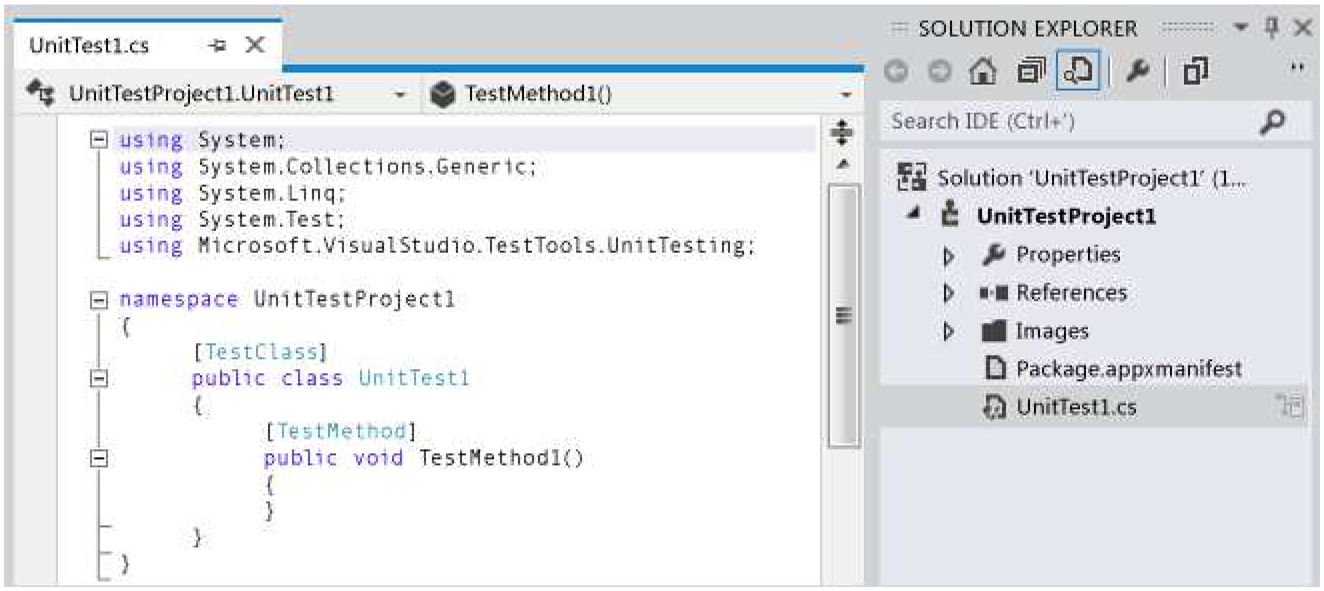
Name: 3 UnitTestLibrary1

Location: 4 C:\Users\AdamBarr\documents\visual studio 2012\Projects

Solution name: 5 UnitTestLibrary1

[✓] Create directory for solution

Solution Explorer is populated with your new Windows 8.x Storeunit test project and the code editor displays the default unit test titled UnitTest1.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Test;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Edit the Manifest for the Unit Test Project

It may be necessary to edit the manifest for the unit test project to provide required capabilities to run the app.

To edit the unit test project's Windows Store application manifest file

1. In Solution Explorer, in the new Windows 8.x Store unit test project, right-click the Package.appxmanifest file and choose **Open**.

   The Manifest Designer displays for editing.

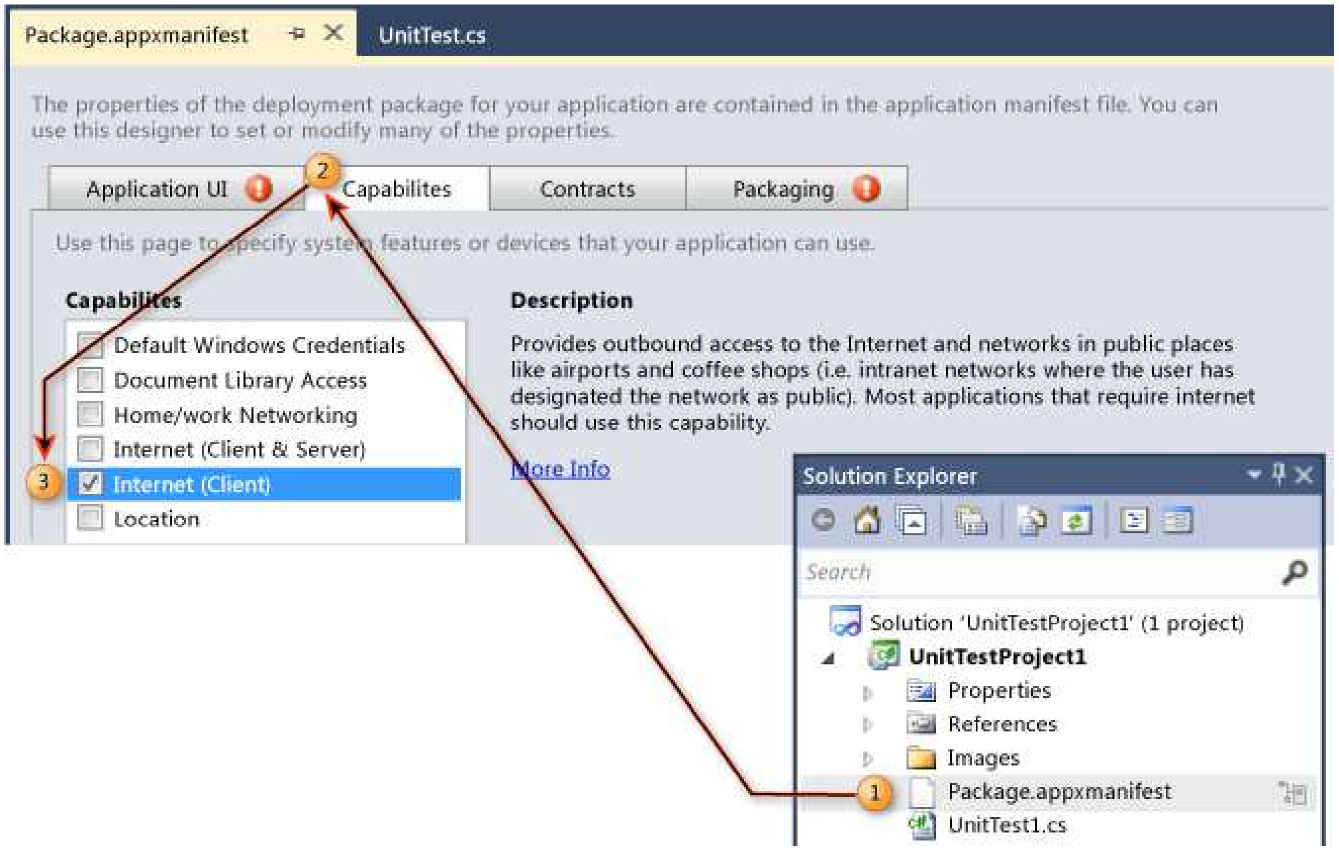
2. In the Manifest Designer, choose the **Capabilities** tab.

3. In the list under **Capabilities**, select the capabilities that you need your unit test and the code that it testing to have. For example, select the **Internet** checkbox if the unit test needs and the code it is testing need to have the capability to access the internet.

Note

The capabilities you select should only include capabilities that are necessary for the Windows 8.x Store unit test to function correctly. The capabilities should never have to include capabilities that are not part of the Windows 8.x Store app being tested and generally should be a subset of the capabilities specified for the Windows 8.x Storeapp under test.

For more information about the Manifest Designer, see Configure a Windows 8.1 app package by using the manifest designer.


Package.appxmanifest
UnitTest.cs
The properties of the deployment package for your application are contained in the application manifest file. You can use this designer to set or modify many of the properties.
Application UI
Capabilites
Contracts
Packaging
Use this page to specify system features or devices that your application can use.
Capabilities
Default Windows Credentials
Document Library Access
Home/work Networking
Internet (Client & Server)
Internet (Client)
Location
Description
Provides outbound access to the Internet and networks in public places like airports and coffee shops (i.e. intranet networks where the user has designated the network as public). Most applications that require internet should use this capability.
More Info
Solution Explorer
Search
Solution 'UnitTestProject1' (1 project)
UnitTestProject1
Properties
References
Images
Package.appxmanifest
UnitTest1.cs
1
2
3

Code the Unit Test

To code the unit test for a Windows Store app

1. In the Code Editor, edit the unit test and add the asserts and logic required for your test.

   For more information, see in Using the Assert Classes in the MSDN library.

Run Unit Tests

To build the solution and run the unit test using Test Explorer

1. On the **Test** menu, choose **Windows**, and then choose **Test Explorer**.

   Test Explorer displays without your test being listed.

2. From the **Build** menu, choose **Build Solution**.

   Your unit test is now listed.

Note

You must build the solution to update the list of unit tests in Test Explorer.

Warning

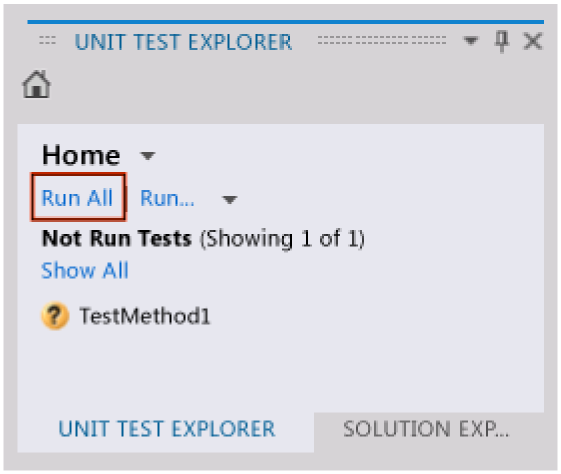
Visual Studio known issue: You must open Test Explorer prior to building the test project.

3. In Test Explorer, choose the unit test you created.

Tip

Test Explorer provides a link to the source code next to **Source:**.

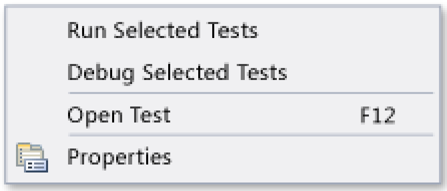
4. Choose **Run All**.



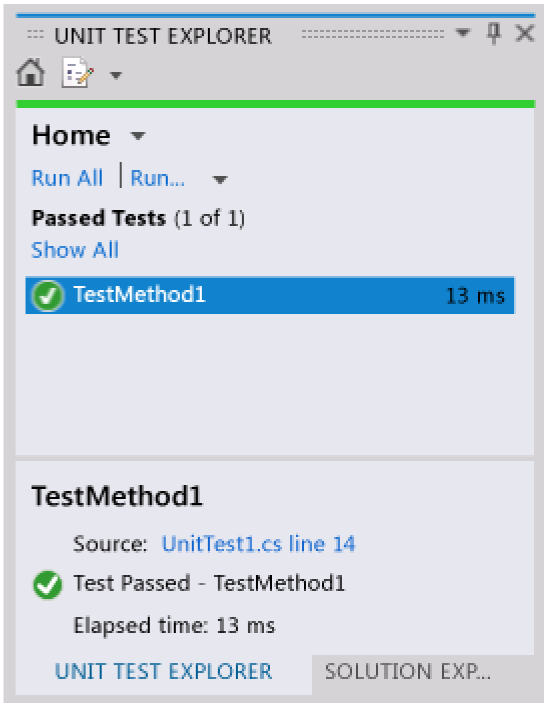
Tip

You can select one or more unit tests listed in Explorer and then right-click and choose **Run Selected Tests**.

Additionally, you can choose to **Debug Selected Tests**, **Open Test**, and use the **Properties** option.



The unit test runs. Upon completion, Test Explorer displays the test status, elapsed time and provides a link to the source.



Passed Tests (1 of 1)

TestMethod1 13 ms

TestMethod1

Source: UnitTest1.cs line 14

Test Passed - TestMethod1

Elapsed time: 13 ms

External Resources

Videos

Channel 9: Unit testing your Windows Store apps built using XAML

Forums

Visual Studio Unit Testing

MSDN Library

MSDN Library – Creating and Running Unit Tests for Existing Code (Visual Studio 2010)

See Also

Testing Store apps with Visual Studio
Build and test a Windows Store app using Team Foundation Build

Writing Unit Tests for the .NET Framework with the Microsoft Unit Test Framework for Managed Code

Visual Studio 2015

In this section

Walkthrough: Creating and Running Unit Tests for Managed Code

Quick Start: Test Driven Development with Test Explorer

Using Microsoft.VisualStudio.TestTools.UnitTesting Members in Unit Tests

Using the Assert Classes

How To: Create a Data-Driven Unit Test

Unit tests for Generic Methods

How to: Configure Unit Tests to Target An Earlier Version of the .NET Framework

Sample Project for Creating Unit Tests

Walkthrough: Creating and Running Unit Tests for Managed Code

Visual Studio 2015

This walkthrough will step you through creating, running, and customizing a series of unit tests using the Microsoft unit test framework for managed code and the Visual Studio Test Explorer. You start with a C# project that is under development, create tests that exercise its code, run the tests, and examine the results. Then you can change your project code and re-run the tests.

This topic contains the following sections:

Prepare the walkthrough

Create a unit test project

Create the test class

- Test class requirements

Create the first test method

- Test method requirements

Build and run the test

Fix your code and rerun your tests

Use unit tests to improve your code

Note

This walkthrough uses the Microsoft unit test framework for managed code. Test Explorer also can run tests from third party unit test frameworks that have adapters for Test Explorer. For more information, see Install third-party unit test frameworks

Note

For information about how to run tests from a command line, see Walkthrough: using the command-line test utility.

Prerequisites

- The Bank project. See Sample Project for Creating Unit Tests.

Prepare the walkthrough

1. Open Visual Studio.
2. On the **File** menu, point to **New** and then click **Project**.

   The **New Project** dialog box appears.
3. Under **Installed Templates**, click **Visual C#**.
4. In the list of application types, click **Class Library**.
5. In the **Name** box, type **Bank** and then click **OK**.

   Note

   If the name "Bank" is already used, choose another name for the project.

   The new Bank project is created and displayed in Solution Explorer with the Class1.cs file open in the Code Editor.

   Note

   If the Class1.cs file is not open in the Code Editor, double-click the file Class1.cs in Solution Explorer to open it.
6. Copy the source code from the Sample Project for Creating Unit Tests.
7. Replace the original contents of Class1.cs with the code from the Sample Project for Creating Unit Tests.
8. Save the file as BankAccount.cs
9. On the **Build** menu, click **Build Solution**.

You now have a project named Bank. It contains source code to test and tools to test it with. The namespace for Bank, **BankAccountNS**, contains the public class **BankAccount**, whose methods you will test in the following procedures.

In this quick start, we focus on the `Debit` method.The Debit method is called when money is withdrawn an account and contains the following code:

C#

```
// method under test
public void Debit(double amount)
{
    if(amount > m_balance)
```

```
        {
            throw new ArgumentOutOfRangeException("amount");
        }
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException("amount");
        }
        m_balance += amount;
    }
```

Create a unit test project

Prerequisite: Follow the steps in the procedure, Prepare the walkthrough.

To create a unit test project

1. On the **File** menu, choose **Add**, and then choose **New Project ...**.
2. In the New Project dialog box, expand **Installed**, expand **Visual C#**, and then choose **Test**.
3. From the list of templates, select **Unit Test Project**.
4. In the **Name** box, enter BankTest, and then choose **OK**.

   The **BankTests** project is added to the the **Bank** solution.
5. In the **BankTests** project, add a reference to the **Bank** solution.

   In Solution Explorer, select **References** in the **BankTests** project and then choose **Add Reference...** from the context menu.
6. In the Reference Manager dialog box, expand **Solution** and then check the **Bank** item.

Create the test class

We need a test class for verifying the `BankAccount` class. We can use the UnitTest1.cs that was generated by the project template, but we should give the file and class more descriptive names. We can do that in one step by renaming the file in Solution Explorer.

Renaming a class file

In Solution Explorer, select the UnitTest1.cs file in the BankTests project. From the context menu, choose **Rename**, and then rename the file to BankAccountTests.cs. Choose **Yes** on the dialog that asks if you want to rename all references in the project to the code element 'UnitTest1'. This step changes the name of the class to `BankAccountTest`.

The BankAccountTests.cs file now contains the following code:

C#

```
// unit test code
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BankTests
{
    [TestClass]
    public class BankAccountTests
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Add a using statement to the project under test

We can also add a using statement to the class to let us to call into the project under test without using fully qualified names. At the top of the class file, add:

C#

```
using BankAccountNS;
```

Test class requirements

The minimum requirements for a test class are the following:

- The `[TestClass]` attribute is required in the Microsoft unit testing framework for managed code for any class that contains unit test methods that you want to run in Test Explorer.
- Each test method that you want Test Explorer to run must have the `[TestMethod]`attribute.

You can have other classes in a unit test project that do not have the `[TestClass]` attribute, and you can have other methods in test classes that do not have the `[TestMethod]` attribute. You can use these other classes and methods in your test methods.

Create the first test method

In this procedure, we will write unit test methods to verify the behavior of the `Debit` method of the `BankAccount` class. The method is listed above.

By analyzing the method under test, we determine that there are at least three behaviors that need to be checked:

1. The method throws an ArgumentOutOfRangeException if the debit amount is greater than the balance.

2. It also throws `ArgumentOutOfRangeException` if the debit amount is less than zero.

3. If the checks in 1.) and 2.) are satisfied, the method subtracts the amount from the account balance.

In our first test, we verify that a valid amount (one that is less than the account balance and that is greater than zero) withdraws the correct amount from the account.

To create a test method

1. Add a using `BankAccountNS;` statement to the BankAccountTests.cs file.

2. Add the following method to that `BankAccountTests` class:

C#

```
// unit test code
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
    // arrange
    double beginningBalance = 11.99;
    double debitAmount = 4.55;
    double expected = 7.44;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // act
    account.Debit(debitAmount);

    // assert
    double actual = account.Balance;
    Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");
}
```

The method is rather simple. We set up a new `BankAccount` object with a beginning balance and then withdraw a valid amount. We use the Microsoft unit test framework for managed code AreEqual method to verify that the ending balance is what we expect.

Test method requirements

A test method must meet the following requirements:

- The method must be decorated with the `[TestMethod]` attribute.
- The method must return `void`.
- The method cannot have parameters.

Build and run the test

To build and run the test

1. On the **Build** menu, choose **Build Solution**.

   If there are no errors, the UnitTestExplorer window appears with **Debit_WithValidAmount_UpdatesBalance** listed in the **Not Run Tests** group. If Test Explorer does not appear after a successful build, choose **Test** on the menu, then choose **Windows**, and then choose **Test Explorer**.

2. Choose **Run All** to run the test. As the test is running the status bar at the top of the window is animated. At the end of the test run, the bar turns green if all the test methods pass, or red if any of the tests fail.

3. In this case, the test does fail. The test method is moved to the **Failed Tests**. group. Select the method in Test Explorer to view the details at the bottom of the window.

Fix your code and rerun your tests

Analyze the test results

The test result contains a message that describes the failure. For the `AreEquals` method, message displays you what was expected (the (**Expected<*XXX*>**parameter) and what was actually received (the **Actual<*YYY*>** parameter). We were expecting the balance to decline from the beginning balance, but instead it has increased by the amount of the withdrawal.

A reexamination of the Debit code shows that the unit test has succeeded in finding a bug. The amount of the withdrawal is added to the account balance when it should be subtracted.

Correct the bug

To correct the error, simply replace the line

C#

```
m_balance += amount;
```

with

C#

```
m_balance -= amount;
```

Rerun the test

In Test Explorer, choose **Run All** to rerun the test. The red/green bar turns green, and the test is moved to the **Passed Tests** group.

Use unit tests to improve your code

This section describes how an iterative process of analysis, unit test development, and refactoring can help you make your production code more robust and effective.

Analyze the issues

After creating a test method to confirm that a valid amount is correctly deducted in the `Debit` method, we can turn to remaining cases in our original analysis:

1. The method throws an `ArgumentOutOfRangeException` if the debit amount is greater than the balance.
2. It also throws `ArgumentOutOfRangeException` if the debit amount is less than zero.

Create the test methods

A first attempt at creating a test method to address these issues seems promising:

C#

```
//unit test method
[TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange()
{
    // arrange
    double beginningBalance = 11.99;
    double debitAmount = -100.00;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // act
    account.Debit(debitAmount);

    // assert is handled by ExpectedException
}
```

We use the ExpectedExceptionAttribute attribute to assert that the right exception has been thrown. The attribute causes the test to fail unless an `ArgumentOutOfRangeException` is thrown. Running the test with both positive and negative `debitAmount` values and then temporarily modifying the method under test to throw a generic ApplicationException when the amount is less than zero demonstrates that test behaves correctly. To test the case when the amount withdrawn is greater than the balance, all we need to do is:

1. Create a new test method named `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange`.
2. Copy the method body from `Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange` to the new method.
3. Set the `debitAmount` to a number greater than the balance.

Run the tests

Running the two methods with different values for `debitAmount` demonstrates that the tests adequately handle our remaining cases. Running all three tests confirm that all cases in our original analysis are correctly covered.

Continue the analysis

However, the last two test methods are also somewhat troubling. We cannot be certain which condition in the code under test throws when either test runs. Some way of differentiating the two conditions would be helpful. As we think about the problem more, it becomes apparent that knowing which condition was violated would increase our confidence in the tests. This information would also very likely be helpful to the production mechanism that handles the exception when it is thrown by the method under test. Generating more information when the method throws would assist all concerned, but the `ExpectedException` attribute cannot supply this information..

Looking at the method under test again, we see both conditional statements use an `ArgumentOutOfRangeException` constructor that takes name of the argument as a parameter:

C#

```csharp
throw new ArgumentOutOfRangeException("amount");
```

From a search of the MSDN Library, we discover that a constructor exists that reports far richer information. ArgumentOutOfRangeException`(String, Object, String)` includes the name of the argument, the argument value, and a user-defined message. We can refactor the method under test to use this constructor. Even better, we can use publicly available type members to specify the errors.

Refactor the code under test

We first define two constants for the error messages at class scope:

C#

```csharp
// class under test
public const string DebitAmountExceedsBalanceMessage = "Debit amount exceeds balance";
public const string DebitAmountLessThanZeroMessage = "Debit amount less than zero";
```

We then modify the two conditional statements in the `Debit` method:

C#

```csharp
// method under test
// ...
    if (amount > m_balance)
    {
        throw new ArgumentOutOfRangeException("amount", amount,
DebitAmountExceedsBalanceMessage);
    }

    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException("amount", amount,
DebitAmountLessThanZeroMessage);
```

```
    }
// ...
```

Refactor the test methods

In our test method, we first remove the `ExpectedException` attribute. In its place, we catch the thrown exception and verify that it was thrown in the correct condition statement. However, we must now decide between two options to verify our remaining conditions. For example in the `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange` method, we can take one of the following actions:

- Assert that the `ActualValue` property of the exception (the second parameter of the `ArgumentOutOfRangeException` constructor) is greater than the beginning balance. This option requires that we test the `ActualValue` property of the exception against the `beginningBalance` variable of the test method, and also requires then verify that the `ActualValue` is greater than zero.
- Assert that the message (the third parameter of the constructor) includes the `DebitAmountExceedsBalanceMessage` defined in the `BankAccount` class.

The StringAssert.Contains method in the Microsoft unit test framework enables us to verify the second option without the calculations that are required of the first option.

A second attempt at revising `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange` might look like:

C#

```
[TestMethod]
public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
{
    // arrange
    double beginningBalance = 11.99;
    double debitAmount = 20.0;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // act
    try
    {
        account.Debit(debitAmount);
    }
    catch (ArgumentOutOfRangeException e)
    {
        // assert
        StringAssert.Contains(e.Message, BankAccount. DebitAmountExceedsBalanceMessage);
    }
}
```

Retest, rewrite, and reanalyze

When we retest the test methods with different values, we encounter the following facts:

1. If we catch the correct error by using an assert where `debitAmount` that is greater than the balance, the `Contains` assert passes, the exception is ignored, and so the test method passes. This is the behavior we want.

2. If we use a `debitAmount` that is less than 0, the assert fails because the wrong error message is returned. The assert also fails if we introduce a temporary `ArgumentOutOfRange` exception at another point in the method under test code path. This too is good.

3. If the `debitAmount` value is valid (i.e., less than the balance but greater than zero, no exception is caught, so the assert is never caught. The test method passes. This is not good, because we want the test method to fail if no exception is thrown.

The third fact is a bug in our test method. To attempt to resolve the issue, we add a Fail assert at the end of the test method to handle the case where no exception is thrown.

But retesting shows that the test now fails if the correct exception is caught. The catch statement resets the exception and the method continues to execute, failing at the new assert. To resolve the new problem, we add a `return` statement after the `StringAssert`. Retesting confirms that we have fixed our problems. Our final version of the `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange` looks like the following:

C#

```
[TestMethod]
public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
{
    // arrange
    double beginningBalance = 11.99;
    double debitAmount = 20.0;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // act
    try
    {
        account.Debit(debitAmount);
    }
    catch (ArgumentOutOfRangeException e)
    {
        // assert
        StringAssert.Contains(e.Message, BankAccount. DebitAmountExceedsBalanceMessage);
        return;
    }
    Assert.Fail("No exception was thrown.");
}
```

In this final section, the work that we did improving our test code led to more robust and informative test methods. But more importantly, the extra analysis also led to better code in our project under test.

Quick Start: Test Driven Development with Test Explorer

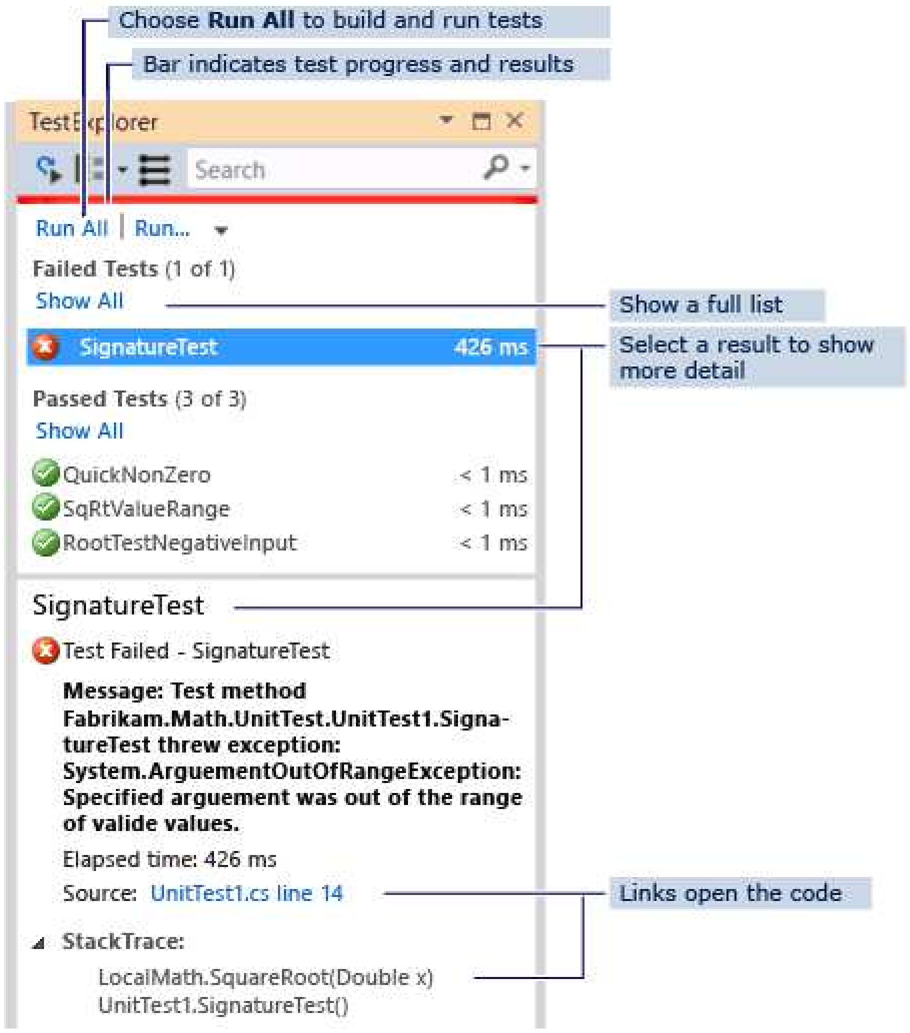
Visual Studio 2015

We recommend that you create unit tests to help keep your code working correctly through many incremental steps of development. There are several frameworks that you can use to write unit tests, including some developed by third parties. Some test frameworks are specialized to testing in different languages or platforms. Test Explorer provides a single interface for unit tests in any of these frameworks. Adapters are available for the most commonly-used frameworks, and you can write your own adapters for other frameworks.

Test Explorer supersedes the unit test windows found in earlier editions of Visual Studio. Its benefits include:

- Run .NET, unmanaged, database and other kinds of tests using a single interface.
- Use the unit test framework of your choice, such as NUnit or MSTest frameworks.
- See in one window all the information that you need.

Using Test Explorer



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

Elapsed time: 426 ms

Source: UnitTest1.cs line 14

StackTrace:

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

To Run Unit Tests by using Test Explorer

1. Create unit tests that use the test frameworks of your choice.

   For example, to create a test that uses the MSTest Framework:

   a. Create a test project.

      In the **New Project** dialog box, expand **Visual Basic**, **Visual C#**, or **Visual C++**, and then choose **Test**.

      Select **Unit Test Project**.

   b. Write each unit test as a method. Prefix each test method with the `[TestMethod]` attribute.

2. If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution with the ☰ toggle button on the toolbar. This can noticeably reduce the time taken to run all the tests.

3. On the menu bar, choose **Test**, **Run Unit Tests**, **All Tests**.

   The solution builds and the tests run.

   Test Explorer opens and displays a summary of the results.

To see a full list of tests: Choose **Show All** in any category.

To see the details of a test result: Select the test in Test Explorer to view details such as exception messages in the details pane.

To navigate to the code of a test: Double-click the test in Test Explorer, or choose **Open Test** on the shortcut menu.

To debug a test: Open the shortcut menu for one or more tests, and then choose **Debug Selected Tests**.

Important

The results that are displayed are for the most recent run. The colored results bar shows only the results for the tests that ran. For example, if you run several tests and some of them fail, and then run only the successful tests, then the results bar will show all green.

Note

If no test appears, make sure that you have installed an adapter to connect Test Explorer to the test framework that you are using. For more information, see Using Different Test Frameworks with Test Explorer.

Walkthrough: Using Unit Tests to Develop a Method

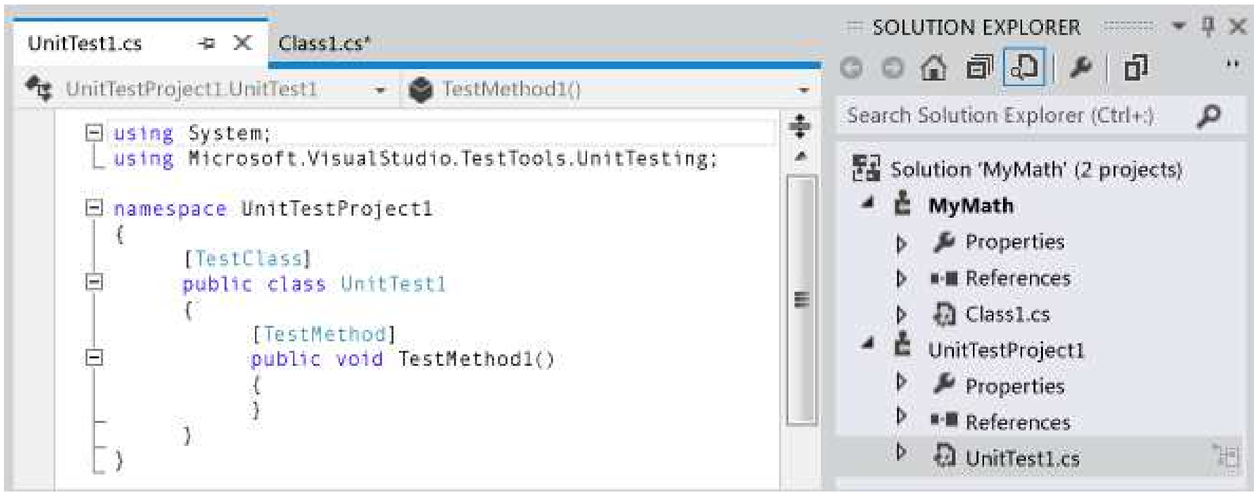
This walkthrough demonstrates how to develop a tested method in C# using the Microsoft Unit Test framework. You can easily adapt it for other languages, and to use other test frameworks such as NUnit. For more information, see Using Different Test Frameworks.

Creating the Test and Method

1. Create a Visual C# Class Library project. This project will contain the code that we want to deliver. In this example, it is named **MyMath**.

2. Create a Test project.

    - In the **New Project** dialog, choose **Visual C#**, **Test** and then choose **Unit Test Project**.



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

namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

3. Write a basic test method. Verify the result obtained for a specific input:

C#

```
[TestMethod]
public void BasicRooterTest()
{
  // Create an instance to test:
  Rooter rooter = new Rooter();
  // Define a test input and output value:
  double expectedResult = 2.0;
  double input = expectedResult * expectedResult;
  // Run the method under test:
  double actualResult = rooter.SquareRoot(input);
  // Verify the result:
  Assert.AreEqual(expectedResult, actualResult,
      delta: expectedResult / 100);
}
```

4. Generate the method from the test.

   a. Place the cursor on `Rooter`, and then on the shortcut menu choose **Generate**, **New Type**.

   b. In the **Generate New Type** dialog box, set **Project** to the class library project. In this example, it is **MyMath**.

   c. Place the cursor on `SquareRoot`, and then on the shortcut menu choose **Generate**, **Method Stub**.

5. Run the unit test.

   a. On the **Test** menu, choose **Run Unit Tests**, **All Tests**.

      The solution builds and runs.

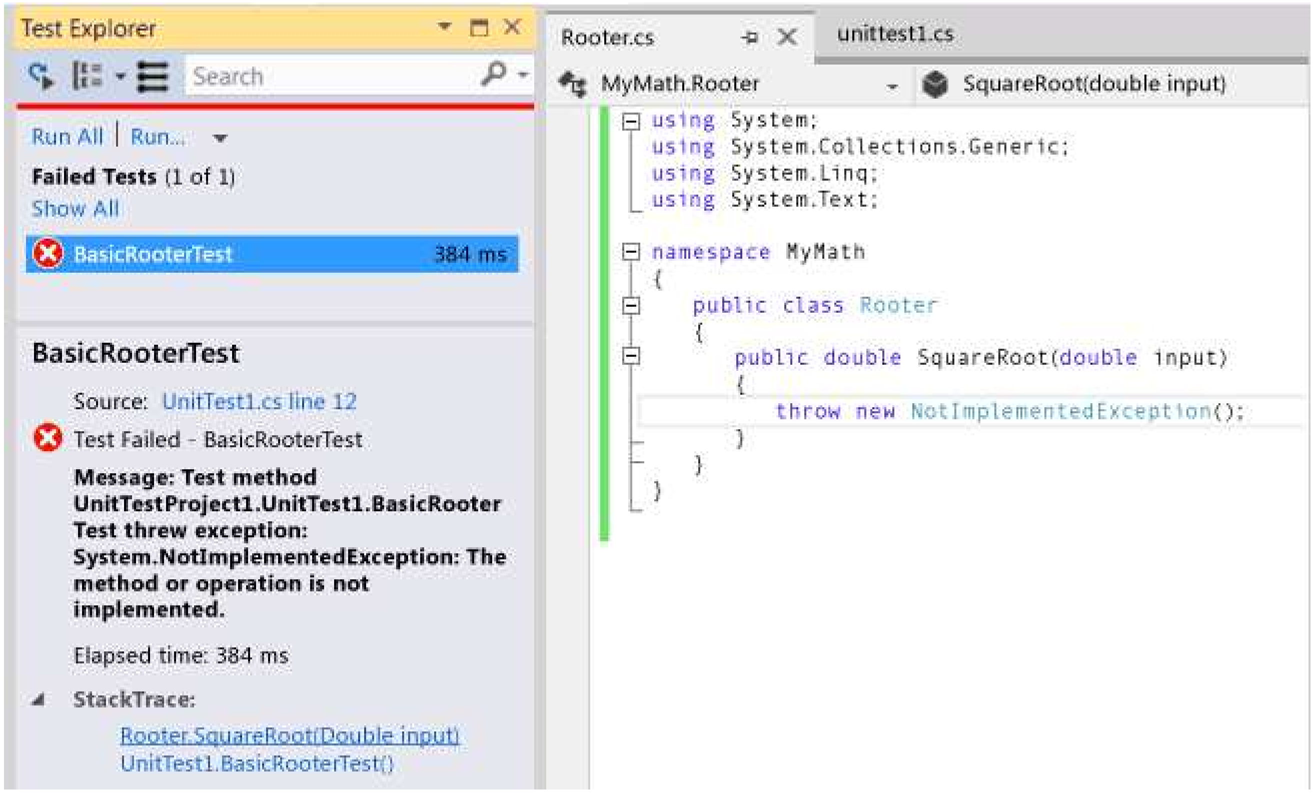
      Test Explorer opens and displays the results.

      The test appears under **Failed Tests**.

6. Select the name of the test.

   The details of the test appear in the lower part of Test Explorer.

7. Select the items under **Stack Trace** to see where the test failed.



BasicRooterTest

Source: UnitTest1.cs line 12

Test Failed - BasicRooterTest

Message: Test method UnitTestProject1.UnitTest1.BasicRooterTest threw exception: System.NotImplementedException: The method or operation is not implemented.

Elapsed time: 384 ms

StackTrace:

Rooter.SquareRoot(Double input)
UnitTest1.BasicRooterTest()

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MyMath
{
    public class Rooter
    {
        public double SquareRoot(double input)
        {
            throw new NotImplementedException();
        }
    }
}
```

At this point, you have created a test and a stub that you will modify so that the test passes.

After every change, make all the tests pass

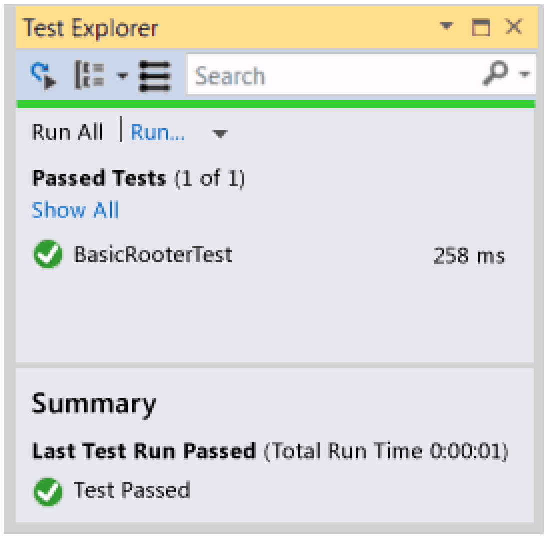
1. In **MyMath\Rooter.cs**, improve the code of `SquareRoot`:

```C#
public double SquareRoot(double input)
 {
   return input / 2;
 }
```

2. In Test Explorer, choose **Run All**.

   The code builds and the test runs.

   The test passes.



BasicRooterTest 258 ms

Summary

Last Test Run Passed (Total Run Time 0:00:01)

Test Passed

Add tests to extend the range of inputs

1. To improve your confidence that your code works in all cases, add tests that try a broader range of input values.

Tip

Avoid altering existing tests that pass. Instead, add new tests. Change existing tests only when the user requirements change. This policy helps ensure that you don’t lose existing functionality as you work to extend the code.

In your test class, add the following test, which tries a range of input values:

C#

```
[TestMethod]
public void RooterValueRange()
{
  // Create an instance to test:
  Rooter rooter = new Rooter();
  // Try a range of values:
  for (double expectedResult = 1e-8;
      expectedResult < 1e+8;
      expectedResult = expectedResult * 3.2)
  {
    RooterOneValue(rooter, expectedResult);
  }
}

private void RooterOneValue(Rooter rooter, double expectedResult)
{
  double input = expectedResult * expectedResult;
  double actualResult = rooter.SquareRoot(input);
  Assert.AreEqual(expectedResult, actualResult,
      delta: expectedResult / 1000);
}
```

2. In Test Explorer, choose **Run All**.

   The new test fails, although the first test still passes.

   To find the point of failure, select the failing test and then in the lower part of Test Explorer, select the top item of the **Stack Trace**.

3. Inspect the method under test to see what might be wrong. In the **MyMath.Rooter** class, rewrite the code:

```
public double SquareRoot(double input)
{
  double result = input;
  double previousResult = -input;
  while (Math.Abs(previousResult - result) > result / 1000)
  {
    previousResult = result;
    result = result - (result * result - input) / (2 * result);
  }
  return result;
}
```

4. In Test Explorer, choose **Run All**.

   Both tests now pass.

Add tests for exceptional cases

1. Add a test for negative inputs:

C#

```
[TestMethod]
 public void RooterTestNegativeInputx()
 {
     Rooter rooter = new Rooter();
     try
     {
         rooter.SquareRoot(-10);
     }
     catch (ArgumentOutOfRangeException e)
     {
         return;
     }
     Assert.Fail();
 }
```

2. In Test Explorer, choose **Run All**.

   The method under test loops, and must be canceled manually.

3. Choose **Cancel**.

   The test stops after 10 seconds.

4. Fix the method code:

C#

```

public double SquareRoot(double input)
{
  if (input <= 0.0)
  {
    throw new ArgumentOutOfRangeException();
  }
...
```

5. In Test Explorer, choose **Run All**.

   All the tests pass.

Refactor without changing tests

1. Simplify the code, but do not change the tests.

Tip

A *refactoring* is a change that is intended to make the code perform better or to make the code easier to understand. It is not intended to alter the behavior of the code, and therefore the tests are not changed.

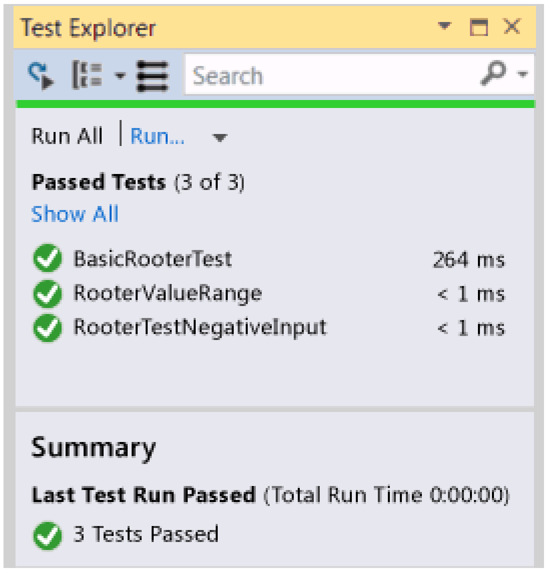
We recommend that you perform refactoring steps separately from steps that extend functionality. Keeping the tests unchanged gives you confidence that you have not accidentally introduced bugs while refactoring.

C#

```
public class Rooter
{
  public double SquareRoot(double input)
  {
    if (input <= 0.0)
    {
      throw new ArgumentOutOfRangeException();
    }
    double result = input;
    double previousResult = -input;
    while (Math.Abs(previousResult - result) > result / 1000)
    {
      previousResult = result;
      result = (result + input / result) / 2;
      //was: result = result - (result * result - input) / (2*result);
    }
    return result;
  }
}
```

2. Choose **Run All**.

All the tests still pass.



Passed Tests (3 of 3)

BasicRooterTest	264 ms
RooterValueRange	< 1 ms
RooterTestNegativeInput	< 1 ms

Summary

Last Test Run Passed (Total Run Time 0:00:00)

3 Tests Passed

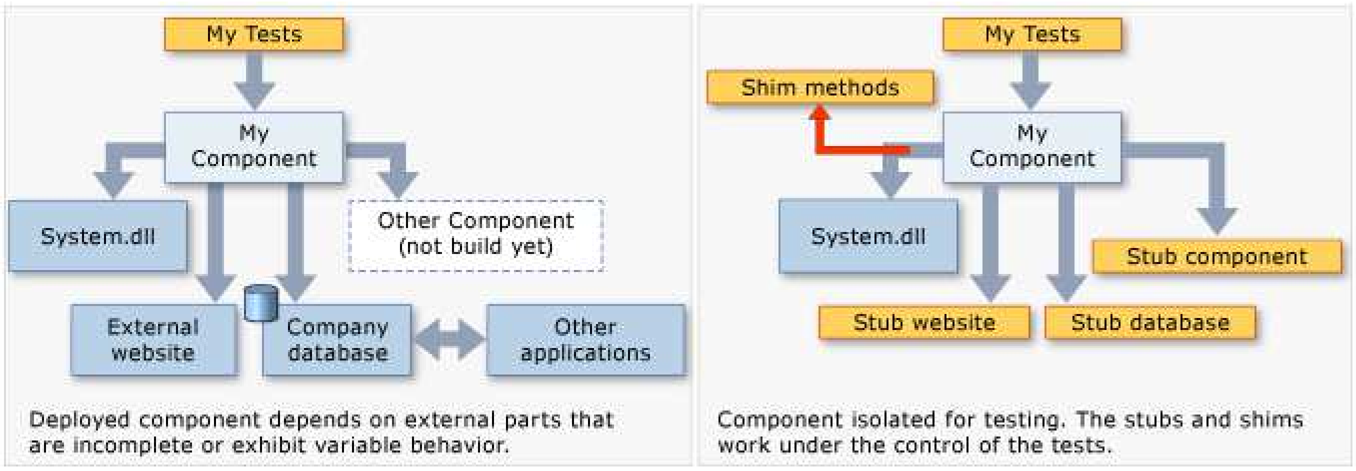
Isolating Code Under Test with Microsoft Fakes

Visual Studio 2015

Microsoft 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.

Fakes come in two flavors:

- 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. (By "component" we mean a class or group of classes that are designed and updated together and typically contained in an assembly.)
- 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.


My Tests
My Component
System.dll
Other Component (not build yet)
External website
Company database
Other applications
Deployed component depends on external parts that are incomplete or exhibit variable behavior.
My Tests
Shim methods
My Component
System.dll
Stub component
Stub website
Stub database
Component isolated for testing. The stubs and shims work under the control of the tests.

Requirements

- Visual Studio Enterprise

Choosing between stub and shim types

Typically, you would consider a Visual Studio project to be a component, because you develop and update those classes at the same time. You would consider using stubs and shims for calls that the project makes to other projects in your solution, or to other assemblies that the project references.

As a general guide, use stubs for calls within your Visual Studio solution, and shims for calls to other referenced assemblies. This is because within your own solution it is good practice to decouple the components by defining interfaces in the way that stubbing requires. But external assemblies such as System.dll typically are not provided with separate interface definitions, so you must use shims instead.

Other considerations are:

Performance. Shims run slower because they rewrite your code at run time. Stubs do not have this performance overhead and are as fast as virtual methods can go.

Static methods, sealed types. You can only use stubs to implement interfaces. Therefore, stub types cannot be used for static methods, non-virtual methods, sealed virtual methods, methods in sealed types, and so on.

Internal types. Both stubs and shims can be used with internal types that are made accessible by using the assembly attribute InternalsVisibleToAttribute.

Private methods. Shims can replace calls to private methods if all the types on the method signature are visible. Stubs can only replace visible methods.

Interfaces and abstract methods. Stubs provide implementations of interfaces and abstract methods that can be used in testing. Shims can't instrument interfaces and abstract methods, because they don't have method bodies.

In general, we recommend that you use stub types to isolate from dependencies within your codebase. You can do this by hiding the components behind interfaces. Shim types can be used to isolate from third-party components that do not provide a testable API.

Getting started with stubs

For a more detailed description, see Using stubs to isolate parts of your application from each other for unit testing.

1. **Inject interfaces**

   To use stubs, you have to write the code you want to test in such a way that it does not explicitly mention classes in another component of your application. By "component" we mean a class or classes that are developed and updated together, and typically contained in one Visual Studio project. Variables and parameters should be declared by using interfaces and instances of other components should be passed in or created by using a factory. For example, if StockFeed is a class in another component of the application, then this would be considered bad:

   ```
   return (new StockFeed()).GetSharePrice("COOO"); // Bad
   ```

   Instead, define an interface that can be implemented by the other component, and which can also be implemented by a stub for test purposes:

   VB

   ```
   Public Function GetContosoPrice(feed As IStockFeed) As Integer
    Return feed.GetSharePrice("COOO")
   End Function
   ```

2. **Add Fakes Assembly**

a. In Solution Explorer, expand the test project's reference list. If you are working in Visual Basic, you must choose **Show All Files** in order to see the reference list.

b. Select the reference to the assembly in which the interface (for example IStockFeed) is defined. On the shortcut menu of this reference, choose **Add Fakes Assembly**.

c. Rebuild the solution.

3. In your tests, construct instances of the stub and provide code for its methods:

VB

```
<TestClass()> _
Class TestStockAnalyzer

    <TestMethod()> _
    Public Sub TestContosoStockPrice()
        ' Arrange:
        ' Create the fake stockFeed:
        Dim stockFeed As New StockAnalysis.Fakes.StubIStockFeed
        With stockFeed
            .GetSharePriceString = Function(company)
                                       Return 1234
                                   End Function
        End With
        ' In the completed application, stockFeed would be a real one:
        Dim componentUnderTest As New StockAnalyzer(stockFeed)
        ' Act:
        Dim actualValue As Integer = componentUnderTest.GetContosoPrice
        ' Assert:
        Assert.AreEqual(1234, actualValue)
    End Sub
End Class
```

The special piece of magic here is the class `StubIStockFeed`. For every interface in the referenced assembly, the Microsoft Fakes mechanism generates a stub class. The name of the stub class is the derived from the name of the interface, with "`Fakes.Stub`" as a prefix, and the parameter type names appended.

Stubs are also generated for the getters and setters of properties, for events, and for generic methods. For more information, see Using stubs to isolate parts of your application from each other for unit testing.

Getting started with shims

(For a more detailed description, see Using shims to isolate your application from other assemblies for unit testing.)

Suppose your component contains calls to `DateTime.Now`:

C#

```
// Code under test:
        public int GetTheCurrentYear()
        {
           return DateTime.Now.Year;
        }
```

During testing, you would like to shim the `Now` property, because the real version inconveniently returns a different value at every call.

To use shims, you don't have to modify the application code or write it a particular way.

1. **Add Fakes Assembly**

   In Solution Explorer, open your unit test project's references and select the reference to the assembly that contains the method you want to fake. In this example, the `DateTime` class is in **System.dll**. To see the references in a Visual Basic project, choose **Show All Files**.

   Choose **Add Fakes Assembly**.

2. **Insert a shim in a ShimsContext**

   VB

```
<TestClass()> _
Public Class TestClass1
    <TestMethod()> _
    Public Sub TestCurrentYear()
        Using s = Microsoft.QualityTools.Testing.Fakes.ShimsContext.Create()
            Dim fixedYear As Integer = 2000
            ' Arrange:
            ' Detour DateTime.Now to return a fixed date:
            System.Fakes.ShimDateTime.NowGet = _
                Function() As DateTime
                    Return New DateTime(fixedYear, 1, 1)
                End Function

            ' Instantiate the component under test:
            Dim componentUnderTest = New MyComponent()
            ' Act:
            Dim year As Integer = componentUnderTest.GetTheCurrentYear
            ' Assert:
            ' This will always be true if the component is working:
            Assert.AreEqual(fixedYear, year)
        End Using
    End Sub
End Class
```

   Shim class names are made up by prefixing `Fakes.Shim` to the original type name. Parameter names are appended to the method name. (You don't have to add any assembly reference to System.Fakes.)

The previous example uses a shim for a static method. To use a shim for an instance method, write `AllInstances` between the type name and the method name:

```
System.IO.Fakes.ShimFile.AllInstances.ReadToEnd = ...
```

(There is no 'System.IO.Fakes' assembly to reference. The namespace is generated by the shim creation process. But you can use 'using' or 'Import' in the usual way.)

You can also create shims for specific instances, for constructors, and for properties. For more information, see Using shims to isolate your application from other assemblies for unit testing.

In this section

Using stubs to isolate parts of your application from each other for unit testing

Using shims to isolate your application from other assemblies for unit testing

Code generation, compilation, and naming conventions in Microsoft Fakes

How To: Create a Data-Driven Unit Test

Visual Studio 2015

Using the Microsoft unit test framework for managed code, you can set up a unit test method to retrieve values used in the test method from a data source. The method is run successively for each row in the data source, which makes it easy to test a variety of input by using a single method.

This topic contains the following sections:

- The method under test
- Creating a data source
- Adding a TestContext to the test class
- Writing the test method
    - Specifying the DataSourceAttribute
    - Using TestContext.DataRow to access the data
- Running the test and viewing results

Creating a data-driven unit test involves the following steps:

1. Create a data source that contains the values that you use in the test method. The data source can be any type that is registered on the machine that runs the test.
2. Add a private TestContext field and a public `TestContext` property to the test class.
3. Create a unit test method and add a DataSourceAttribute attribute to it.
4. Use the DataRow indexer property to retrieve the values that you use in a test.

The method under test

As an example, let's assume that we have created:

1. A solution called `MyBank` that accepts and processes transactions for different types of accounts.
2. A project in `MyBank` called `BankDb` that manages the transactions for accounts.
3. A class called `Maths` in the `DbBank` project that performs the mathematical functions to ensure that any transaction is advantageous to the bank.

4. A unit test project called `BankDbTests` to test the behavior of the `BankDb` component.

5. A unit test class called `MathsTests` to verify the behavior of the `Maths` class.

We will test a method in `Maths` that adds two integers using a loop:

```
public int AddIntegers(int first, int second)
{
    int sum = first;
    for( int i = 0; i < second; i++)
    {
        sum += 1;
    }
    return sum;
}
```

Creating a data source

To test the `AddIntegers` method, we create a data source that specifies a range of values for the parameters and the sum that you expect to be returned. In our example, we create a Sql Compact database named `MathsData` and a table named `AddIntegersData` that contains the following column names and values

FirstNumber	SecondNumber	Sum
0	1	1
1	1	2
2	-3	-1

Adding a TestContext to the test class

The unit test framework creates a `TestContext` object to store the data source information for a data-driven test. The framework then sets this object as the value of the `TestContext` property that we create.

```
private TestContext testContextInstance;
public TestContext TestContext
{
    get { return testContextInstance; }
    set { testContextInstance = value; }
}
```

In your test method, you access the data through the `DataRow` indexer property of the `TestContext`.

Writing the test method

The test method for `AddIntegers` is fairly simple. For each row in the data source, we call `AddIntegers` with the **FirstNumber** and **SecondNumber** column values as parameters, and we verify the return value against **Sum** column value:

```
[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});

}
```

Note that the `Assert` method includes a message that displays the `x` and `y` values of a failed iteration. By default, the asserted values, `expected` and `actual`, are already included in the details of a failed test.

Specifying the DataSourceAttribute

The `DataSource` attribute specifies the connection string for the data source and the name of the table that you use in the test method. The exact information in the connection string differs, depending on what kind of data source you are using. In this example, we used a SqlServerCe database.

```
[DataSource(@"Provider=Microsoft.SqlServerCe.Client.4.0;Data Source=C:\Data
\MathsData.sdf", "AddIntegersData")]
```

The DataSource attribute has three constructors.

```
[DataSource(dataSourceSettingName)]
```

A constructor with one parameter uses connection information that is stored in the app.config file for the solution. The *dataSourceSettingsName* is the name of the Xml element in the config file that specifies the connection information.

Using an app.config file allows you to change the location of the data source without making changes to the unit test itself. For information about how to create and use an app.config file, see Walkthrough: Using a Configuration File to Define a Data Source

```
[DataSource(connectionString, tableName)]
```

The `DataSource` constructor with two parameters specifies the connection string for the data source and the name of the table that contains the data for the test method.

The connection strings depend on the type of the type of data source, but it should contain a Provider element that specifies the invariant name of the data provider.

```
[DataSource(
    dataProvider,
    connectionString,
    tableName,
    dataAccessMethod
    )]
```

Using TestContext.DataRow to access the data

To access the data in the `AddIntegersData` table, use the `TestContext.DataRow` indexer. `DataRow` is a DataRow object, so we retrieve column values by index or column names. Because the values are returned as objects, we need to convert them to the appropriate type:

```
int x = Convert.ToInt32(TestContext.DataRow["FirstNumber"]);
```

Running the test and viewing results

When you have finished writing a test method, build the test project. The test method appears in the Test Explorer window in the **Not Run Tests** group. As you run, write, and rerun your tests, Test Explorer displays the results in groups of **Failed Tests**, **Passed Tests**, and **Not Run Tests**. You can choose **Run All** to run all your tests, or choose **Run...** to choose a subset of tests to run.

The test results bar at the top of the Explorer is animated as your test runs. At the end of the test run, the bar will be green if all of the tests have passed or red if any of the tests have failed. A summary of the test run appears in the details pane at the bottom of the Test Explorer window. Select a test to view the details of that test in the bottom pane.

If you ran the `AddIntegers_FromDataSourceTest` method in our example, the results bar turns red and the test method is moved to the **Failed Tests** A data-driven test fails if any of the iterated methods from the data source fails. When you choose a failed data-driven test in the Test Explorer window, the details pane displays the results of each iteration that is identified by the data row index. In our example, it appears that the `AddIntegers` algorithm does not handle negative values correctly.

When the method under test is corrected and the test rerun, the results bar turns green and the test method is moved to the **Passed Test** group.

See Also

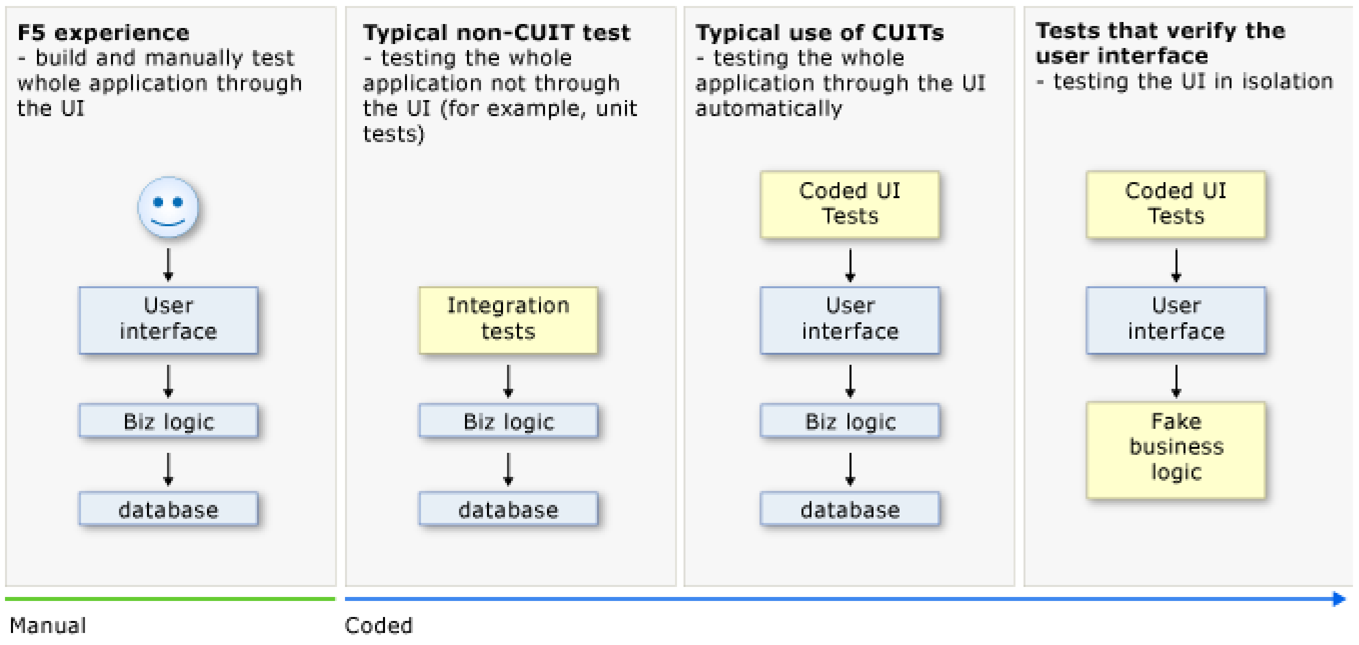
Microsoft.VisualStudio.TestTools.UnitTesting.DataSourceAttribute
Microsoft.VisualStudio.TestTools.UnitTesting.TestContext
TestContext.DataRow
Microsoft.VisualStudio.TestTools.UnitTesting.Assert
How to: Create and Run a Unit Test
Unit Test Your Code
Run unit tests with Test Explorer
Writing Unit Tests for the .NET Framework with the Microsoft Unit Test Framework for Managed Code

Use UI Automation To Test Your Code

Visual Studio 2015

Automated tests that drive your application through its user interface (UI) are known as *coded UI tests* (CUITs). These tests include functional testing of the UI controls. They let you verify that the whole application, including its user interface, is functioning correctly. Coded UI Tests are particularly useful where there is validation or other logic in the user interface, for example in a web page. They are also frequently used to automate an existing manual test.

As shown in the following illustration, a typical development experience might be one where, initially, you simply build your application (F5) and click through the UI controls to verify that things are working correctly. You then might decide to create a coded test so that you don't need to continue to test the application manually. Depending on the particular functionality being tested in your application, you can write code for either a functional test, or for an integration test that might or might not include testing at the UI level. If you simply want to directly access some business logic, you might code a unit test. However, under certain circumstances, it can be beneficial to include testing of the various UI controls in your application. A coded UI test can automate the initial (F5) scenario, verifying that code churn does not impact the functionality of your application.


F5 experience
- build and manually test whole application through the UI
User interface
Biz logic
database
Typical non-CUIT test
- testing the whole application not through the UI (for example, unit tests)
Integration tests
Biz logic
database
Typical use of CUITs
- testing the whole application through the UI automatically
Coded UI Tests
User interface
Biz logic
database
Tests that verify the user interface
- testing the UI in isolation
Coded UI Tests
User interface
Fake business logic
Manual
Coded

Creating a coded UI test is easy. You simply perform the test manually while the CUIT Test Builder runs in the background. You can also specify what values should appear in specific fields. The CUIT Test Builder records your actions and generates code from them. After the test is created, you can edit it in a specialized editor that lets you modify the sequence of actions.

Alternatively, if you have a test case that was recorded in Microsoft Test Manager, you can generate code from that. For more information, see Record and play back manual tests.

The specialized CUIT Test Builder and editor make it easy to create and edit coded UI tests even if your main skills are concentrated in testing rather than coding. But if you are a developer and you want to extend the test in a more advanced way, the code is structured so that it is straightforward to copy and adapt. For example, you might record a test to buy something at a website, and then edit the generated code to add a loop that buys many items.

Requirements

- Visual Studio Enterprise

For more information about which platforms and configurations are supported by coded UI tests, see Supported Configurations and Platforms for Coded UI Tests and Action Recordings.

In this topic

- Creating Coded UI Tests
  - Main procedure
  - Starting and stopping the application
  - Validating the properties of UI Controls
- Customizing your coded UI test
  - The Generated Code
  - Coding UI control actions and properties
  - Debugging
- What's Next

Creating Coded UI Tests

1. **Create a Coded UI Test project.**

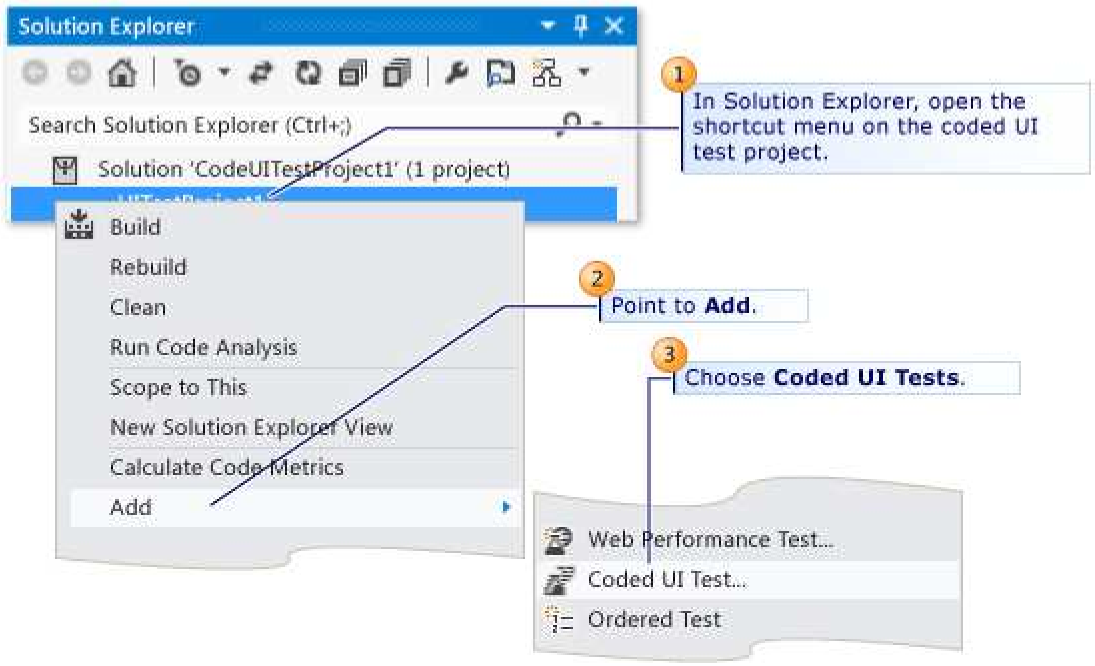
   Coded UI tests must be contained in a coded UI test project. If you don't already have a coded UI test project, create one. In **Solution Explorer**, on the shortcut menu of the solution, choose **Add**, **New Project** and then select either **Visual Basic** or **Visual C#**. Next, choose **Test**, **Coded UI Test**.

   - *I don't see the **Coded UI Test** project templates.*

     You might be using a version of Visual Studio that does not support coded UI tests. To create coded UI tests, you must use Visual Studio Enterprise.

2. **Add a coded UI test file.**

   If you just created a Coded UI project, the first CUIT file is added automatically. To add another test file, open the shortcut menu on the coded UI test project, point to **Add**, and then choose **Coded UI Test**.

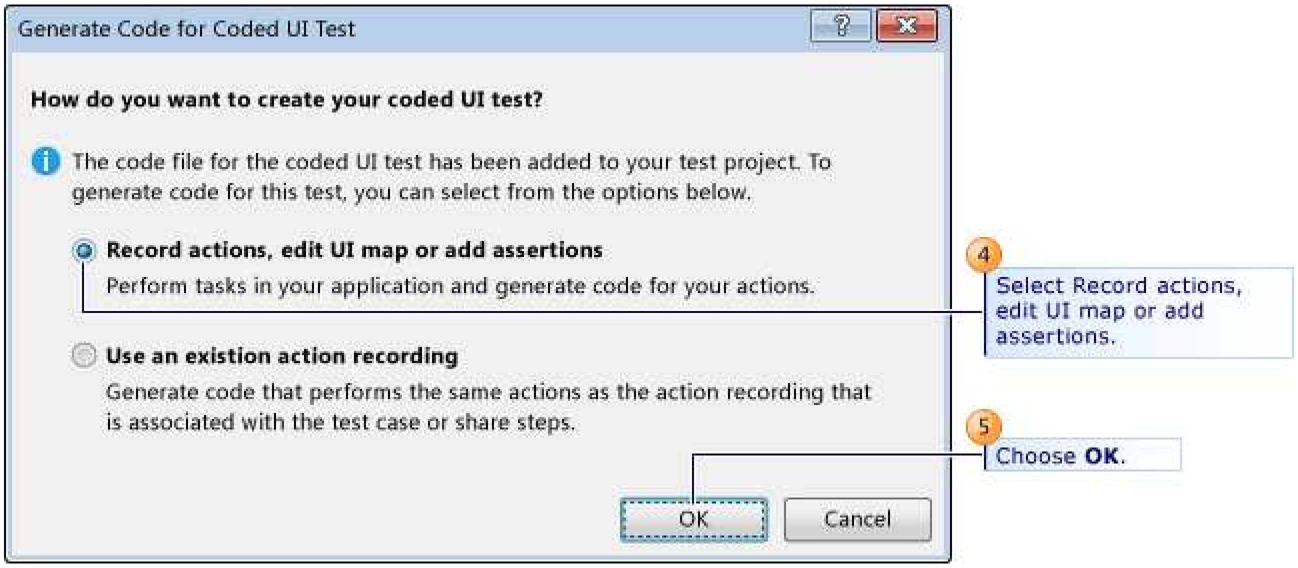


1 In Solution Explorer, open the shortcut menu on the coded UI test project.

2 Point to **Add**.

3 Choose **Coded UI Tests**.

In the **Generate Code for Coded UI Test** dialog box, choose **Record actions, edit UI map or add assertions**.


Generate Code for Coded UI Test

How do you want to create your coded UI test?

The code file for the coded UI test has been added to your test project. To generate code for this test, you can select from the options below.

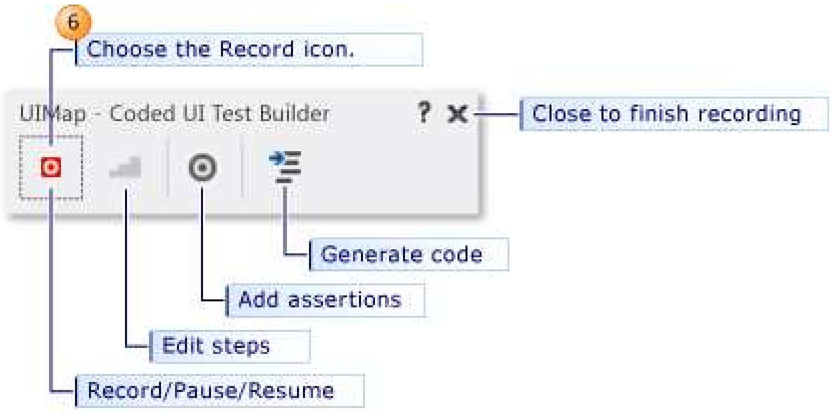
[✓ **Record actions, edit UI map or add assertions**]
Perform tasks in your application and generate code for your actions.

Use an existion action recording
Generate code that performs the same actions as the action recording that is associated with the test case or share steps.

4 Select Record actions, edit UI map or add assertions.

5 Choose **OK**.

The Coded UI Test Builder appears and Visual Studio is minimized.


6 Choose the Record icon.
UIMap - Coded UI Test Builder
Close to finish recording
Generate code
Add assertions
Edit steps
Record/Pause/Resume

3. **Record a sequence of actions**.

To start recording, choose the **Record** icon. Perform the actions that you want to test in your application, including starting the application if that is required.

For example, if you are testing a web application, you might start a browser, navigate to the web site, and log in to the application.

To pause recording, for example if you have to deal with incoming mail, choose **Pause**.

Warning

All actions performed on the desktop will be recorded. Pause the recording if you are performing actions that may lead to sensitive data being included in the recording.

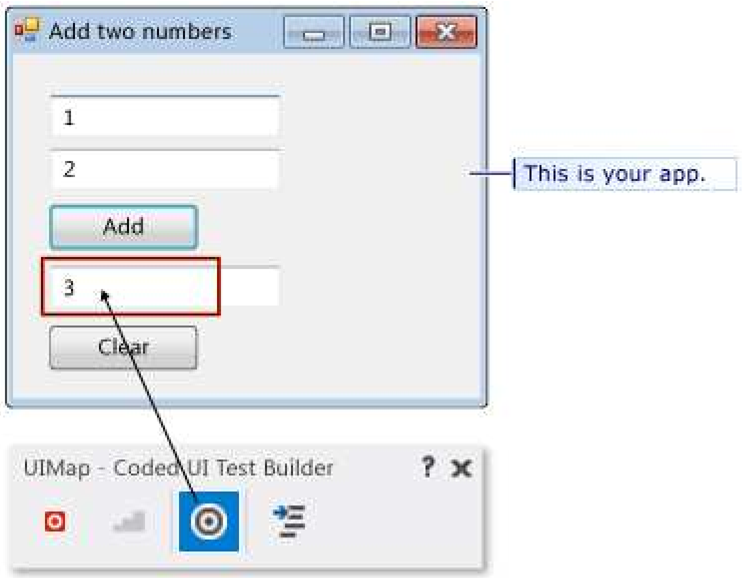
To delete actions that you recorded by mistake, choose **Edit Actions**.

To generate code that will replicate your actions, choose the **Generate Code** icon and type a name and description for your coded UI test method.

4. **Verify the values in UI fields such as text boxes**.

   Choose **Add Assertions** in the Coded UI Test Builder, and then choose a UI control in your running application. In the list of properties that appears, select a property, for example, **Text** in a text box. On the shortcut menu, choose **Add Assertion**. In the dialog box, select the comparison operator, the comparison value, and the error message.

   Close the assertion window and choose **Generate Code**.



Tip

Alternate between recording actions and verifying values. Generate code at the end of each sequence of actions or verifications. If you want, you will be able to insert new actions and verifications later.

For more details, see Validating Properties of Controls.

5. **View the generated test code**.

   To view the generated code, close the UI Test Builder window. In the code, you can see the names that you gave to each step. The code is in the CUIT file that you created:

   C#

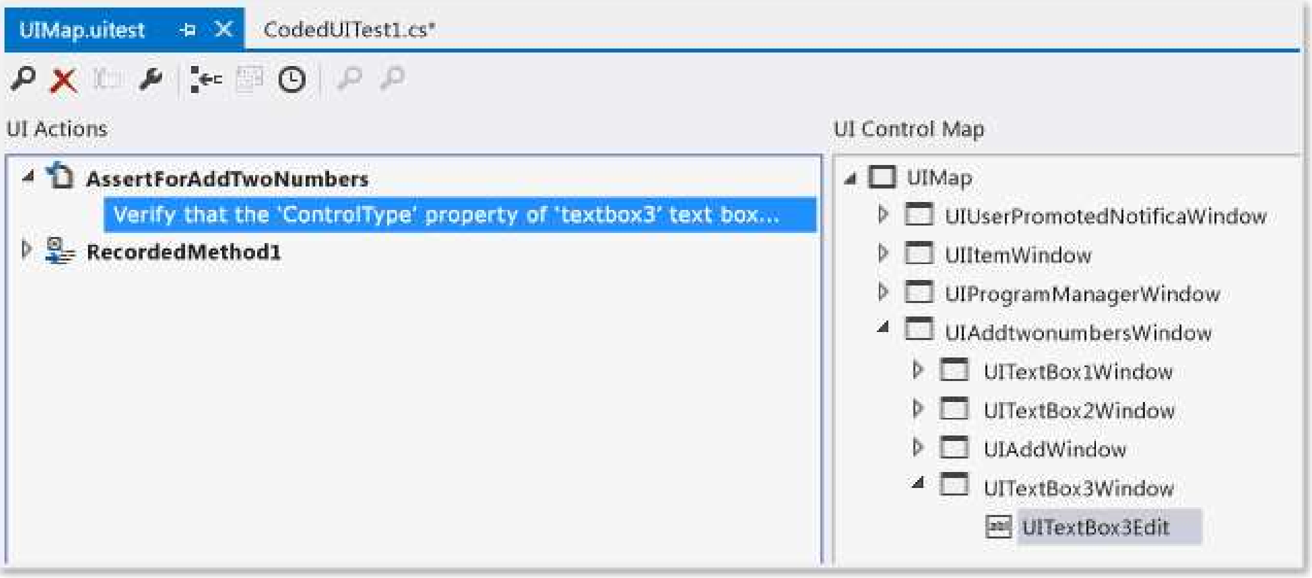
```
[CodedUITest]
public class CodedUITest1
{ ...
  [TestMethod]
  public void CodedUITestMethod1()
  {
      this.UIMap.AddTwoNumbers();
      this.UIMap.VerifyResultValue();
      // To generate more code for this test, select
      // "Generate Code" from the shortcut menu.
  }
}
```

6. **Add more actions and assertions**.

   Place the cursor at the appropriate point in the test method and then, on the shortcut menu, choose **Generate Code for Coded UI Test**. New code will be inserted at that point.

7. **Edit the detail of the test actions and the assertions**.

   Open UIMap.uitest. This file opens in the Coded UI Test Editor, where you can edit any sequence of actions that you recorded as well as edit your assertions.



AssertForAddTwoNumbers
Verify that the 'ControlType' property of 'textbox3' text box...
RecordedMethod1

UIMap
UIUserPromotedNotificaWindow
UIItemWindow
UIProgramManagerWindow
UIAddtwonumbersWindow
UITextBox1Window
UITextBox2Window
UIAddWindow
UITextBox3Window
UITextBox3Edit

For more information, see Editing Coded UI Tests Using the Coded UI Test Editor.

8. **Run the test**.

Use Test Explorer, or open the shortcut menu in the test method, and then choose **Run Tests**. For more information about how to run tests, see Run unit tests with Test Explorer and *Additional options for running coded UI tests* in the What's next? section at the end of this topic.

The remaining sections in this topic provide more detail about the steps in this procedure.

For a more detailed example, see Walkthrough: Creating, Editing and Maintaining a Coded UI Test. In the walkthrough, you will create a simple Windows Presentation Foundation (WPF) application to demonstrate how to create, edit, and maintain a coded UI test. The walkthrough provides solutions for correcting tests that have been broken by various timing issues and control refactoring.

Starting and stopping the application under test

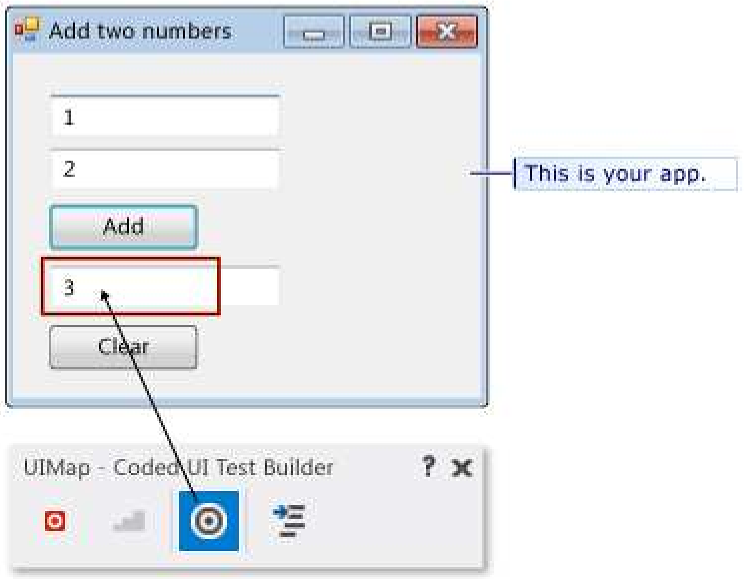
I don't want to start and stop my application, browser, or database separately for each test. How do I avoid that?

- If you do not want to record the actions to start your application under test, you must start your application before you choose the **Record** icon.
- At the end of a test, the process in which the test runs is terminated. If you started your application in the test, the application usually closes. If you do not want the test to close your application when it exits, you must add a .runsettings file to your solution and use the `KeepExecutorAliveAfterLegacyRun` option. For more information, see Configure unit tests by using a .runsettings file.
- You can add a test initialize method, identified by a [TestInitialize] attribute, which runs code at the start of each test method. For example, you could start the application from the TestInitialize method.
- You can add a test cleanup method, identified by a [TestCleanup] attribute, that runs code at the end of each test method. For example, the method to close the application could be called from the TestCleanup method.

Validating the properties of UI controls

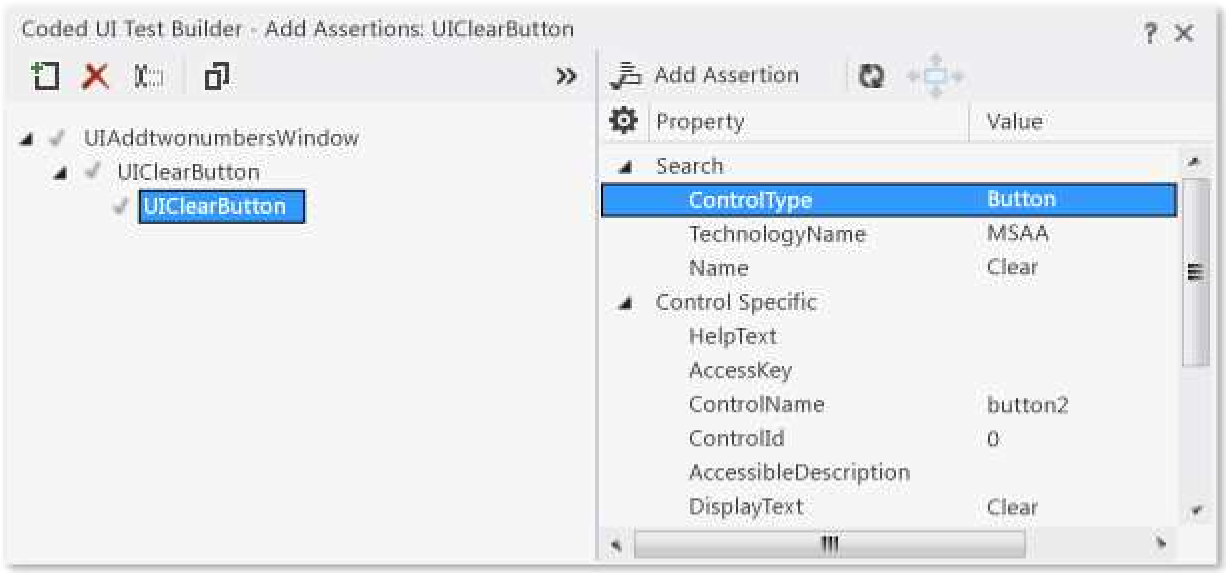
You can use the **Coded UI Test Builder** to add a user interface (UI) control to the T:Microsoft.VisualStudio.TestTools.UITest.Common.UIMap.UIMap for your test, or to generate code for a validation method that uses an assertion for a UI control.

To generate assertions for your UI controls, choose the **Add Assertions** tool in the Coded UI Test Builder and drag it to the control on the application under test that you want to verify is correct. When the box outlines your control, release the mouse. The control class code is immediately created in the `UIMap.Designer.cs` file.



The properties for this control are now listed in the **Add Assertions** dialog box.

Another way of navigating to a particular control is to choose the arrow **(<<)** to expand the view for the **UI Control Map**. To find a parent, sibling, or child control, you can click anywhere on the map and use the arrow keys to move around the tree.

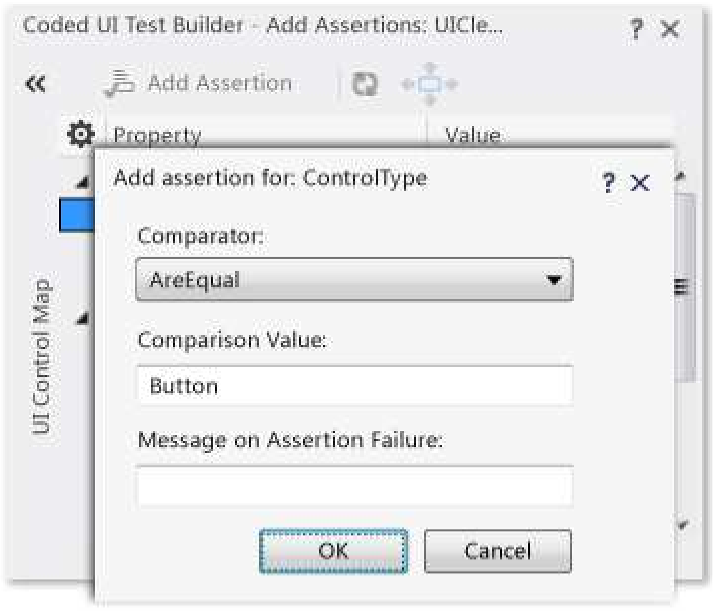


Property	Value
Search	
ControlType	Button
TechnologyName	MSAA
Name	Clear
Control Specific	
HelpText	
AccessKey	
ControlName	button2
ControlId	0
AccessibleDescription	
DisplayText	Clear

- *I don't see any properties when I select a control in my application, or I don't see the control in the UI Control Map.*

  In the application code, the control that you want to verify must have a unique ID, such as an HTML ID attribute, or a WPF UId. You might need to update the application code to add these IDs.

Next, open the shortcut menu on the property for the UI control that you want to verify, and then point to **Add Assertion**. In the **Add Assertion** dialog box, select the **Comparator** for your assertion, for example AreEqual, and type the value for your assertion in **Comparison Value**.


Coded UI Test Builder - Add Assertions: UICle...
Add Assertion
Property
Value
UI Control Map
Add assertion for: ControlType
Comparator:
AreEqual
Comparison Value:
Button
Message on Assertion Failure:
OK
Cancel

When you have added all your assertions for your test, choose **OK**.

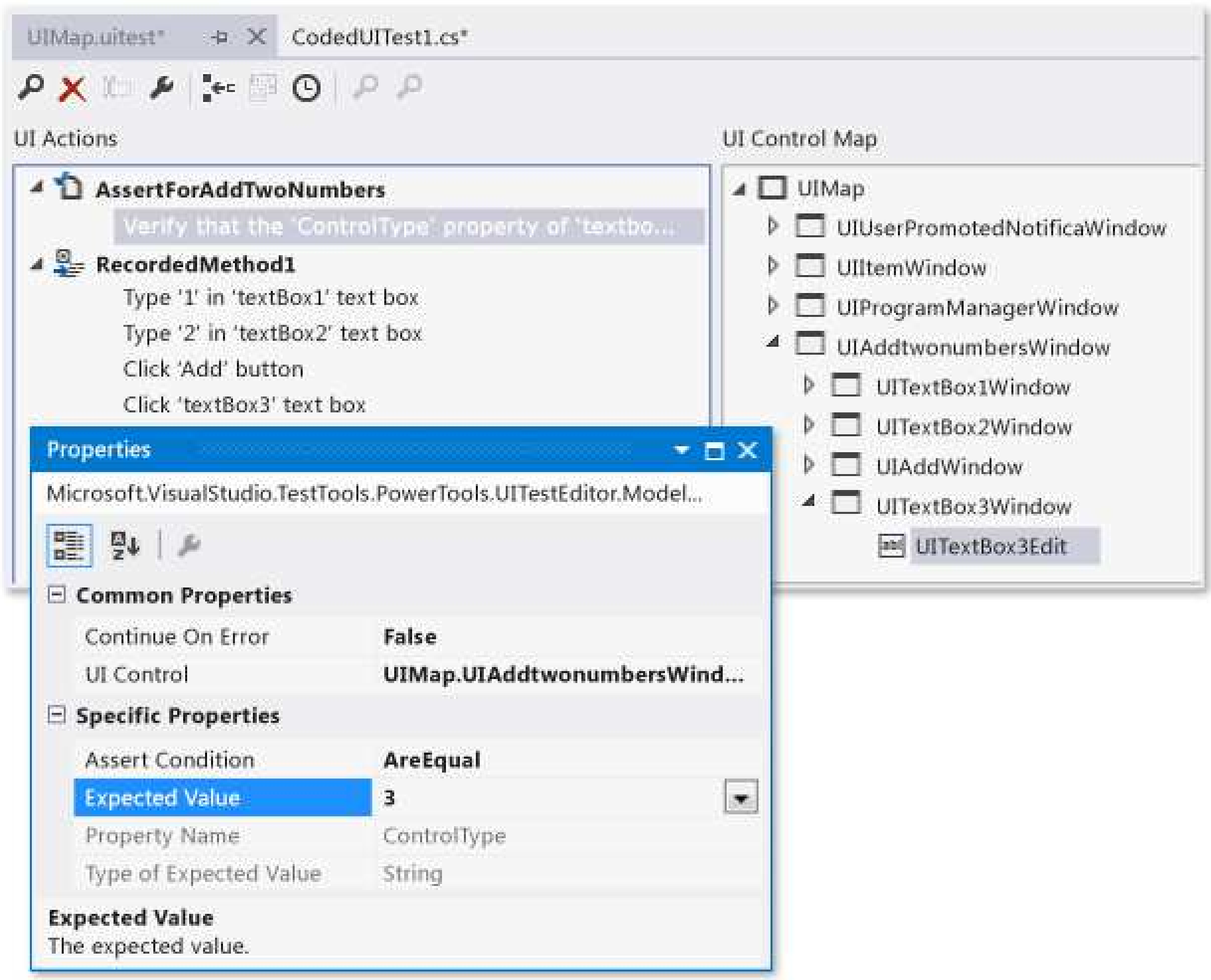
To generate the code for your assertions and add the control to the UI map, choose the **Generate Code** icon. Type a name for your coded UI test method and a description for the method, which will be added as comments for the method. Choose **Add and Generate**. Next, choose the **Close** icon to close the **Coded UI Test Builder**. This generates code similar to the following code. For example, if the name you entered is `AssertForAddTwoNumbers`, the code will look like this example:

- Adds a call to the assert method AssertForAddTwoNumbers to the test method in your coded UI test file:

```
[TestMethod]
public void CodedUITestMethod1()
{
    this.UIMap.AddTwoNumbers();
    this.UIMap.AssertForAddTwoNumbers();
}
```

  You can edit this file to change the order of the steps and assertions, or to create new test methods. To add more code, place the cursor on the test method and on the shortcut menu choose **Generate Code for Coded UI Test**.

- Adds a method called `AssertForAddTwoNumbers` to your UI map (UIMap.uitest). This file opens in the Coded UI Test Editor, where you can edit the assertions.



For more information, see Editing Coded UI Tests Using the Coded UI Test Editor.

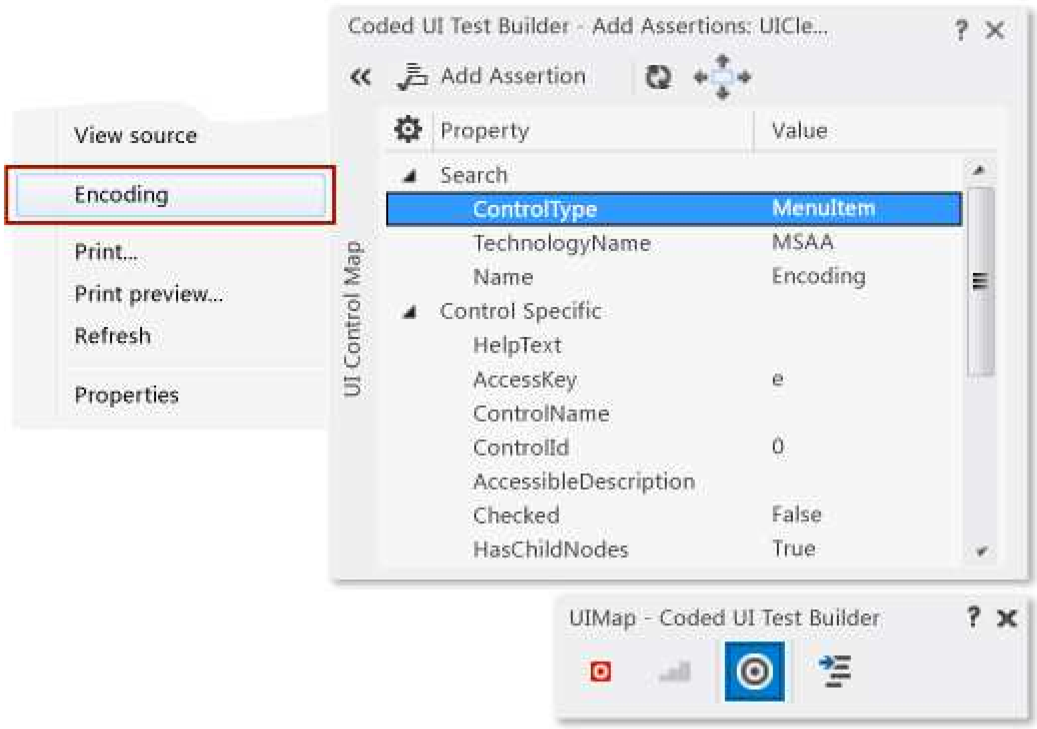
You can also view the generated code of the assertion method in UIMap.Designer.cs. However, you should not edit this file. If you want to make an adapted version of the code, copy the methods to another file such as UIMap.cs, rename the methods, and edit them there.

```
public void AssertForAddTwoNumbers()
{
    ...
}
```

The control I want to select loses focus and disappears when I try to select the Add Assertions tool from the Coded UI Test Builder. How do I select the control?

Selecting a hidden control using the keyboard

Sometimes, when adding controls and validating their properties, you might have to use the keyboard. For example, when you try to record a coded UI test that uses a context menu control, the list of menu items in the control will lose focus and disappear when you try to select the Add Assertions tool from the Coded UI Test Builder. This is demonstrated in the following illustration, where the context menu in Internet Explorer will lose focus and disappear if you try to select it with the Add Assertions tool.



Property	Value
Search	
ControlType	MenuItem
TechnologyName	MSAA
Name	Encoding
Control Specific	
HelpText	
AccessKey	e
ControlName	
ControlId	0
AccessibleDescription	
Checked	False
HasChildNodes	True

To use the keyboard to select a UI control, hover over the control with the mouse. Then hold down the **Ctrl** key and the **I** key at the same time. Release the keys. The control is recorded by the Coded UT Test Builder.

Warning

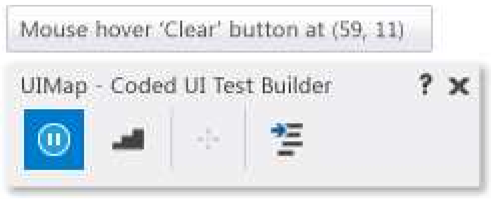
If you use Microsoft Lync, you must close Lync before you start the Coded UI Test Builder. Microsoft Lync interferes with the **Ctrl+I** keyboard shortcut.

I can't record a mouse hover on a control. Is there a way around this?

Manually recording mouse hovers

Under some circumstances, a particular control that's being used in a coded UI test might require you to use the keyboard to manually record mouse hover events. For example, when you test a Windows Form or a Windows Presentation Foundation (WPF) application, there might be custom code. Or, there might be special behavior defined for hovering over a control, such as a tree node expanding when a user hovers over it. To test circumstances like these, you have to manually notify the Coded UI Test Builder that you are hovering over the control by pressing predefined keyboard keys.

When you perform your coded UI test, hover over the control. Then press and hold Ctrl, while you press and hold the Shift and R keys on your keyboard. Release the keys. A mouse hover event is recorded by the Coded UT Test Builder.


Mouse hover 'Clear' button at (59, 11)
UIMap - Coded UI Test Builder

After you generate the test method, code similar to the following example will be added to the UIMap.Desinger.cs file:

C#

```
// Mouse hover '1' label at (87, 9)
Mouse.Hover(uIItem1Text, new Point(87, 9));
```

The key assignment for capturing mouse hover events is being used elsewhere in my environment. Can I change the default key assignment?

Configuring mouse hover keyboard assignments

If necessary, the default keyboard assignment of Ctrl+Shift+R that is used to apply mouse hovering events in your coded UI tests can be configured to use different keys.

Warning

You should not have to change the keyboard assignments for mouse hover events under ordinary circumstances. Use caution when reassigning the keyboard assignment. Your choice might already be in use elsewhere within Visual Studio or the application being tested.

To change the keyboard assignments, you must modify the following configuration file:

<drive letter:>\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\CodedUITestBuilder.exe.config

In the configuration file, change the values for the `HoverKeyModifier` and `HoverKey` keys to modify the keyboard assignments:

```
<!-- Begin : Background Recorder Settings -->
<!-- HoverKey to use. -->
<add key="HoverKeyModifier" value="Control, Shift"/>
<add key="HoverKey" value="R"/>
```

I'm having issues with recording mouse hovers on a website. Is there a fix for this, too?

Setting implicit mouse hovers for the web browser

In many websites, when you hover over a particular control, it expands to show additional details. Generally, these look like menus in desktop applications. Because this is a common pattern, coded UI tests enable implicit hovers for Web browsing. For example, if you record hovers in Internet Explorer, an event is fired. These events can lead to redundant hovers getting recorded. Because of this, implicit hovers are recorded with `ContinueOnError` set to `true` in the UI test configuration file. This allows playback to continue if a hover event fails.

To enable the recording of implicit hovers in a Web browser, open the configuration file:

<drive letter:>\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\CodedUITestBuilder.exe.config

Verify that the configuration file has the key `RecordImplicitiHovers` set to a to a value of `true` as shown in the following sample:

```
<!--Use this to enable/disable recording of implicit hovers.-->
<add key="RecordImplicitHover" value="true"/>
```

Customizing your coded UI test

After you've created your coded UI test, you can edit it by using any of the following tools in Visual Studio:

- **Coded UI Test Builder:** Use the Coded UI Test Builder to add additional controls and validation to your tests. See the section Adding controls and validating their properties in this topic.
- **Coded UI Test Editor:** The Coded UI Test Editor lets you easily modify your coded UI tests. Using the Coded UI Test Editor, you can locate, view, and edit your test methods. You can also edit UI actions and their associated controls in the UI control map. For more information, see Editing Coded UI Tests Using the Coded UI Test Editor.
- **Code Editor:**
  - Manually add code for the controls in your test as described in the Coding UI control actions and properties section in this topic.
  - After you create a coded UI test, you can modify it to be data-driven. For more information, see Creating a Data-Driven Coded UI Test.
  - In a coded UI test playback, you can instruct the test to wait for certain events to occur, such as a window to appear, the progress bar to disappear, and so on. To do this, add the appropriate UITestControl.WaitForControlXXX() method. For a complete list of the available methods, see Making Coded UI Tests Wait For Specific Events During Playback. For an example of a coded UI test that waits for a control to be enabled using the WaitForControlEnabled method, see Walkthrough: Creating, Editing and Maintaining a Coded UI Test.
  - Coded UI tests include support for some of the HTML5 controls that are included in Internet Explorer 9 and Internet Explorer 10. For more information, see Using HTML5 Controls in Coded UI Tests.
  - **Coded UI test coding guidance:**

- Anatomy of a Coded UI Test
- Best Practices for Coded UI Tests
- Testing a Large Application with Multiple UI Maps
- Supported Configurations and Platforms for Coded UI Tests and Action Recordings

The Generated Code

When you choose **Generate Code**, several pieces of code are created:

- **A line in the test method.**

C#

```
[CodedUITest]
public class CodedUITest1
{ ...
  [TestMethod]
  public void CodedUITestMethod1()
  {
      this.UIMap.AddTwoNumbers();
      // To generate more code for this test, select
      // "Generate Code" from the shortcut menu.      }
}
```

You can right-click in this method to add more recorded actions and verifications. You can also edit it manually to extend or modify the code. For example, you could enclose some of the code in a loop.

You can also add new test methods and add code to them in the same way. Each test method must have the `[TestMethod]` attribute.

- **A method in UIMap.uitest**

This method includes the detail of the actions you recorded or the value that you verified. You can edit this code by opening UIMap.uitest. It opens in a specialized editor in which you can delete or refactor the recorded actions.

You can also view the generated method in UIMap.Designer.cs. This method performs the actions that you recorded when you run the test.

C#

```
// File: UIMap.Designer.cs
public partial class UIMap
{
  /// <summary>
  /// Add two numbers
  /// </summary>
  public void AddTwoNumbers()
```

```
    { ...   }
  }
```

Warning

You should not edit this file, because it will be regenerated when you create more tests.

You can make adapted versions of these methods by copying them to UIMap.cs. For example, you could make a parameterized version that you could call from a test method:

C#

```
  // File: UIMap.cs
  public partial class UIMap // Same partial class
  {
    /// <summary>
    /// Add two numbers – parameterized version
    /// </summary>
    public void AddTwoNumbers(int firstNumber, int secondNumber)
    { ...   // Code modified to use parameters.
    }
  }
```

- **Declarations in UIMap.uitest**

  These declarations represent the UI controls of the application that are used by your test. They are used by the generated code to operate the controls and access their properties.

  You can also use them if you write your own code. For example, you can have your test method choose a hyperlink in a Web application, type a value in a text box, or branch off and take different testing actions based on a value in a field.

  You can add multiple coded UI tests and multiple UI map objects and files to facilitate testing a large application. For more information, see Testing a Large Application with Multiple UI Maps.

For more information about the generated code, see Anatomy of a Coded UI Test.

Coding UI control actions and properties

When you work with UI test controls in coded UI tests they are separated into two parts: actions and properties.

- The first part consists of actions that you can perform on UI test controls. For example, coded UI tests can simulate mouse clicks on a UI test control, or simulate keys typed on the keyboard to affect a UI test control.

- The second part consists of enabling you to get and set properties on a UI test control. For example, coded UI tests can get the count of items in a **ListBox**, or set a **CheckBox** to the selected state.

Accessing Actions of UI Test Control

To perform actions on UI test controls, such as mouse clicks or keyboard actions, use the methods in the Mouse and Keyboard classes:

- To perform a mouse-oriented action, such as a mouse click, on a UI test control, use Click.

  ```
  Mouse.Click(buttonCancel);
  ```

- To perform a keyboard-oriented action, such as typing into an edit control, use SendKeys.

  ```
  Keyboard.SendKeys(textBoxDestination, @"C:\Temp\Output.txt");
  ```

Accessing Properties of UI Test Control

To get and set UI control specific property values, you can directly get or set the values the properties of a control, or you can use the UITestControl.GetProperty and UITestControl.SetProperty methods with the name of the specific property that you want you get or set.

GetProperty returns an object, which can then be cast to the appropriate Type. SetProperty accepts an object for the value of the property.

To get or set properties from UI test controls directly

- With controls that derive from T:Microsoft.VisualStudio.TestTools.UITesting.UITestControl, such as T:Microsoft.VisualStudio.TestTools.UITesting.HtmlControls.HtmlList or T:Microsoft.VisualStudio.TestTools.UITesting.WinControls.WinComboBox, you can get or set their property values directly, as follows:

  ```
  int i = myHtmlList.ItemCount;
  myWinCheckBox.Checked = true;
  ```

To get properties from UI test controls

- To get a property value from a control, use GetProperty.

- To specify the property of the control to get, use the appropriate string from the **PropertyNames** class in each control as the parameter to GetProperty.

- GetProperty returns the appropriate data type, but this return value is cast as an Object. The return Object must then be cast as the appropriate type.

  Example:

  ```
  int i = (int)GetProperty(myHtmlList.PropertyNames.ItemCount);
  ```

To set properties for UI test controls

- To set a property in a control, use SetProperty.

- To specify the property of the control to set, use the appropriate string from the **PropertyNames** class as the first parameter to SetProperty, with the property value as the second parameter.

Example:

```
SetProperty(myWinCheckBox.PropertyNames.Checked, true);
```

Debugging

You can analyze Coded UI tests using coded UI test logs. Coded UI test logs filter and record important information about your coded UI test runs. The format of the logs lets you debug issues quickly. For more information, see Analyzing Coded UI Tests Using Coded UI Test Logs.

What's next?

Additional options for running coded UI tests: You can run coded UI tests directly from Visual Studio, as described earlier in this topic. Additionally, you can run automated UI tests from Microsoft Test Manager, or from Team Foundation Build. When coded UI tests are automated, they have to interact with the desktop when you run them, unlike other automated tests.

- How to: Run Tests from Microsoft Visual Studio
- Running Automated Tests in Microsoft Test Manager
- How to: Configure and Run Scheduled Tests After Building Your Application
- Run tests in your build process
- Running automated tests from the command line
- How to: Set Up Your Test Agent to Run Tests that Interact with the Desktop
- [retired] Using Coded UI Tests in Load Tests

Adding support for custom controls: The coded UI testing framework does not support every possible UI and might not support the UI you want to test. For example, you cannot immediately create a coded UI test of the UI for Microsoft Excel. However, you can create an extension to the coded UI testing framework that will support a custom control.

- Enable Coded UI Testing of Your Controls
- Extending Coded UI Tests and Action Recordings to Support Microsoft Excel

Coded UI Tests are often used to automate manual tests. For additional guidance, see Testing for Continuous Delivery with Visual Studio 2012 – Chapter 5: Automating System Tests. For more information about manual tests, see [retired] Creating Manual Test Cases Using Microsoft Test Manager. For more information about automated system tests, see Creating Automated Tests Using Microsoft Test Manager.

External Resources

Guidance

Testing for Continuous Delivery with Visual Studio 2012 – Chapter 2: Unit Testing: Testing the Inside

Testing for Continuous Delivery with Visual Studio 2012 – Chapter 5: Automating System Tests

FAQ

Coded UI Tests FAQ - 1

Coded UI Tests FAQ -2

Forum

Visual Studio UI Automation Testing (includes CodedUI)

See Also

T:Microsoft.VisualStudio.TestTools.UITest.Common.UIMap.UIMap
Assert
Improve Code Quality
Walkthrough: Creating, Editing and Maintaining a Coded UI Test
Anatomy of a Coded UI Test
Best Practices for Coded UI Tests
Testing a Large Application with Multiple UI Maps
Editing Coded UI Tests Using the Coded UI Test Editor
Supported Configurations and Platforms for Coded UI Tests and Action Recordings
Upgrading Coded UI Tests from Visual Studio 2010
Generating a Coded UI Test from an Existing Action Recording

Walkthrough: Creating, Editing and Maintaining a Coded UI Test

Visual Studio 2015

In this walkthrough, you will create a simple Windows Presentation Foundation (WPF) application to demonstrate how to create, edit, and maintain a coded UI test. The walkthrough provides solutions for correcting tests that have been broken by various timing issues and control refactoring.

Prerequisites

For this walkthrough you will need:

- Visual Studio Enterprise

Create a Simple WPF Application

1. On the **FILE** menu, point to **New**, and then select **Project**.

   The **New Project** dialog box appears.

2. In the **Installed** pane, expand **Visual C#**, and then select **Windows Desktop**.

3. Above the middle pane, verify that the target framework drop-down list is set to **.NET Framework 4.5**.

4. In the middle pane, select the **WPF Application** template.

5. In the **Name** text box, type **SimpleWPFApp**.

6. Choose a folder where you will save the project. In the **Location** text box, type the name of the folder.

7. Choose **OK**.

   The WPF Designer for Visual Studio opens and displays MainWindow of the project.

8. If the toolbox is not currently open, open it. Choose the **VIEW** menu, and then choose **Toolbox**.

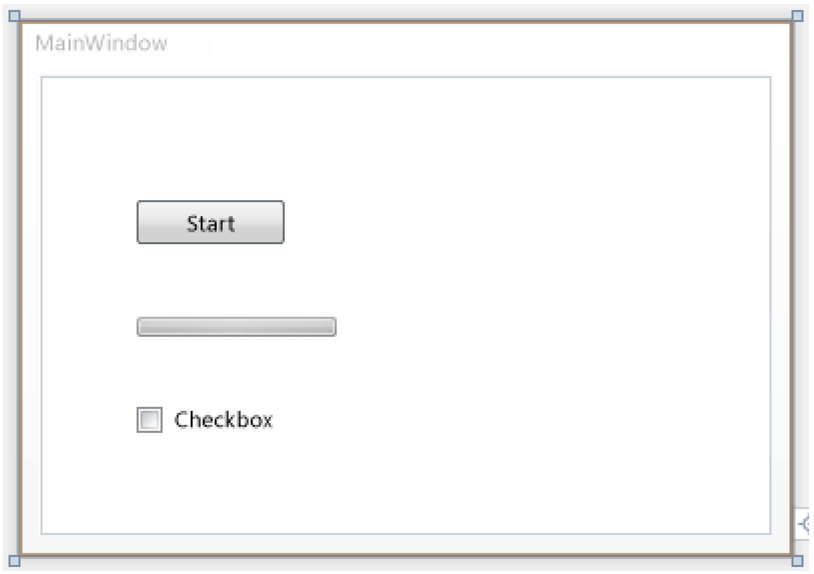
9. Under the **All WPF Controls** section, drag a **Button**, **CheckBox** and **ProgressBar** control onto the MainWindow in the design surface.

10. Select the Button control. In the Properties window, change the value for the **Name** property from <No Name> to button1. Then change the value for the **Content** property from Button to Start.

11. Select the ProgressBar control. In the Properties window, change the value for the value for the **Name** property from <No Name> to progressBar1. Then change the value for the **Maximum** property from **100** to **10000**.

12. Select the Checkbox control. In the Properties window, change the value for the **Name** property from <No Name> to

checkBox1 and clear the **IsEnabled** property.



13. Double-click the button control to add a click event handler.

    The MainWindow.xmal.cs is displayed in the Code Editor with the cursor in the new button1_Click method.

14. At the top of the MainWindow class, add a delegate. The delegate will be used for the progress bar. To add the delegate, add the following code:

C#

```
public partial class MainWindow : Window
{
        private delegate void ProgressBarDelegate(System.Windows.DependencyProperty
dp, Object value);

    public MainWindow()
    {

        InitializeComponent();
    }
```

15. In the button1_Click method, add the following code:

C#

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    double progress = 0;

    ProgressBarDelegate updatePbDelegate =
        new ProgressBarDelegate(progressBar1.SetValue);

    do
```

```
        {
            progress ++;

            Dispatcher.Invoke(updatePbDelegate,
                System.Windows.Threading.DispatcherPriority.Background,
                new object[] { ProgressBar.ValueProperty, progress });
            progressBar1.Value = progress;
        }
        while (progressBar1.Value != progressBar1.Maximum);

        checkBox1.IsEnabled = true;
    }
```

16. Save the file.

Verify the WPF Application Runs Correctly

1. On the **DEBUG** menu, select **Start Debugging** or press **F5**.
2. Notice that the check box control is disabled. Choose **Start**.

   In a few seconds, the progress bar should be 100% complete.
3. You can now select the check box control.
4. Close SimpleWPFApp.

Create and Run a Coded UI Test for SimpleWPFApp

1. Locate the SimpleWPFApp application that you created earlier. By default, the application will be located at C:\Users\<username>\Documents\Visual Studio <version>\Projects\SimpleWPFApp\SimpleWPFApp\bin\Debug\SimpleWPFApp.exe
2. Create a desktop shortcut to the SimpleWPFApp application. Right-click SimpleWPFApp.exe and choose **Copy**. On your desktop, right-click and choose **Paste shortcut**.

Tip

A shortcut to the application makes it easier to add or modify Coded UI tests for your application because it lets you start the application quickly.

3. In Solution Explorer, right-click the solution, choose **Add** and then select **New Project**.

   The **Add New Project** dialog box appears.
4. In the **Installed** pane, expand **Visual C#**, and then select **Test**.

5. In the middle pane, select the **Coded UI Test Project** template.

6. Choose **OK**.

   In Solution Explorer, the new coded UI test project named **CodedUITestProject1** is added to your solution.

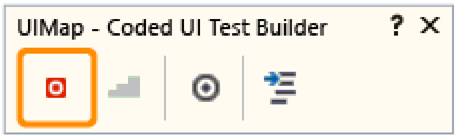
   The **Generate Code for Coded UI Test** dialog box appears.

7. Select the **Record actions, edit UI map or add assertions** option and choose **OK**.

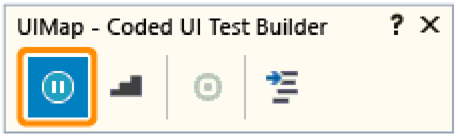
   The UIMap – Coded UI Test Builder appears, and the Visual Studio window is minimized.

   For more information about the options in the dialog box, see Creating Coded UI Tests.

8. Choose **Start Recording** on the UIMap – Coded UI Test Builder.

   You can pause the recording if needed, for example if you have to deal with incoming mail.

   Warning

   All actions performed on the desktop will be recorded. Pause the recording if you are performing actions that may lead to sensitive data being included in the recording.

9. Launch the SimpleWPFApp using the desktop shortcut.

   As before, notice that the check box control is disabled.

10. On the SimpleWPFApp, choose **Start**.

    In a few seconds, the progress bar should be 100% complete.

11. Check the check box control which is now enabled.

12. Close the SimpleWPFApp application.

13. On the UIMap - Coded UI Test Builder, choose **Generate Code**.

14. In the Method Name type **SimpleAppTest** and choose **Add and Generate**. In a few seconds, the Coded UI test appears and is added to the Solution.

15. Close the UIMap – Coded UI Test Builder.

    The CodedUITest1.cs file appears in the Code Editor.

16. Save your project.

Run the Coded UI Test

1. From the **TEST** menu, choose **Windows** and then choose **Test Explorer**.

2. From the **BUILD** menu, choose **Build Solution**.

3. In the CodedUITest1.cs file, locate the **CodedUITestMethod** method, right-click and select **Run Tests**, or run the test from Test Explorer.

   While the coded UI test runs, the SimpleWPFApp is visible. It conducts the steps that you did in the previous procedure. However, when the test tries to select the check box for the check box control, the Test Results window shows that the test failed. This is because the test tries to select the check box but is not aware that the check box control is disabled until the progress bar is 100% complete. You can correct this and similar issues by using the various `UITestControl.WaitForControlXXX()` methods that are available for coded UI testing. The next procedure will demonstrate using the `WaitForControlEnabled()` method to correct the issue that caused this test to fail. For more information, see Making Coded UI Tests Wait For Specific Events During Playback.

Edit and Rerun the Coded UI Test

1. In the Test Explorer window, select the failed test and in the **StackTrace** section, choose the first link to **UIMap.SimpleAppTest()**.

2. The UIMap.Designer.cs file opens with the point of error highlighted in the code:

   C#

```
// Select 'CheckBox' check box
uICheckBoxCheckBox.Checked = this.SimpleAppTestParams.UICheckBoxCheckBoxChecked;
```

3. To correct this problem, you can make the coded UI test wait for the CheckBox control to be enabled before continuing on to this line using the `WaitForControlEnabled()` method.

   Warning

   Do not modify the UIMap.Designer.cs file. Any code changes you make in the UIMapDesigner.cs file will be overwritten every time you generate code using the UIMap - Coded UI Test Builder. If you have to modify a recorded method, you must copy it to UIMap.cs file and rename it. The UIMap.cs file can be used to override methods and properties in the UIMapDesigner.cs file. You must remove the reference to the original method in the Coded UITest.cs file and replace it with the renamed method name.

4. In Solution Explorer, locate **UIMap.uitest** in your coded UI test project.

5. Open the shortcut menu for **UIMap.uitest** and choose **Open**.

   The coded UI test is displayed in the Coded UI Test Editor. You can now view and edit the coded UI test.

6. In the **UI Action** pane, select the test method (SimpleAppTest) that you want to move to the UIMap.cs or UIMap.vb file to facilitate custom code functionality which won't be overwritten when the test code is recompiled.

7. Choose the **Move Code** button on the Coded UI Test Editor toolbar.

8. A Microsoft Visual Studio dialog box is displayed. It warns you that the method will be moved from the UIMap.uitest file to the UIMap.cs file and that you will no longer be able to edit the method using the Coded UI Test Editor. Choose **Yes**.

   The test method is removed from the UIMap.uitest file and no longer is displayed in the UI Actions pane. To edit the moved test file, open the UIMap.cs file from Solution Explorer.

9. On the Visual Studio toolbar, choose **Save**.

   The updates to the test method are saved in the UIMap.Designer file.

   Caution

   Once you have moved the method, you can no longer edit it using the Coded UI Test Editor. You must add your custom code and maintain it using the Code Editor.

10. Rename the method from `SimpleAppTest()` to `ModifiedSimpleAppTest()`

11. Add the following using statement to the file:

    C#

```
using Microsoft.VisualStudio.TestTools.UITesting.WpfControls;
```

12. Add the following `WaitForControlEnabled()` method before the offending line of code identified previously:

    C#

```
uICheckBoxCheckBox.WaitForControlEnabled();

// Select 'CheckBox' check box
uICheckBoxCheckBox.Checked = this.SimpleAppTestParams.UICheckBoxCheckBoxChecked;
```

13. In the CodedUITest1.cs file, locate the **CodedUITestMethod** method and either comment out or rename the reference to the original SimpleAppTest() method and then replace it with the new ModifiedSimpleAppTest():

    C#

```
[TestMethod]
        public void CodedUITestMethod1()
        {
            // To generate code for this test, select "Generate Code for Coded UI
Test" from the shortcut menu and select one of the menu items.
            // For more information on generated code, see http://go.microsoft.com
/fwlink/?LinkId=179463
            //this.UIMap.SimpleAppTest();
            this.UIMap.ModifiedSimpleAppTest();
        }
```

14. On the **BUILD** menu, choose **Build Solution**.

15. Right-click the **CodedUITestMethod** method and select **Run Tests**.

16. This time the coded UI test successfully completes all the steps in the test and **Passed** is displayed in the Test Explorer window.

Refactor a Control in the SimpleWPFApp

1. In the MainWindow.xaml file, in the Designer, select the button control.

2. At the top of the Properties window, change the **Name** property value from button1 to buttonA.

3. On the **BUILD** menu, choose **Build Solution**.

4. In Test Explorer, run **CodedUITestMethod1**.

   The test fails because the coded UI test cannot locate the button control that was originally mapped in the UIMap as button1. Refactoring can impact coded UI tests in this manner.

5. In the Test Explorer window, in the **StackTrace** section, choose the first link next to **UIMpa.ModifiedSimpleAppTest()**.

   The UIMap.cs file opens. The point of error is highlighted in the code:

   C#

```
// Click 'Start' button
Mouse.Click(uIStartButton, new Point(27, 10));
```

   Notice that the line of code earlier in this procedure is using `UiStartButton`, which is the UIMap name before it was refactored.

   To correct the issue, you can add the refactored control to the UIMap by using the Coded UI Test Builder. You can update the test's code to use the code, as demonstrated in the next procedure.

Map Refactored Control and Edit and Rerun the Coded UI Test

1. In the CodedUITest1.cs file, in the **CodedUITestMethod1()** method, right-click, select **Generate Code for Coded UI**

Test and then choose **Use Coded UI Test Builder**.

The UIMap – Coded UI Test Builder appears.

2. Using the desktop shortcut you created earlier, run the SimpleWPFApp application that you created earlier.

3. On the UIMap – Coded UI Test Builder, drag the crosshair tool to the **Start** button on the SimpleWPFApp.

   The **Start** button is enclosed in a blue box and the Coded UI Test Builder takes a few seconds to process the data for the selected control and displays the controls properties. Notice that the **AutomationUId** is named **buttonA**.

4. In the properties for the control, choose the arrow at the upper-left corner to expand the UI Control Map. Notice that **UIStartButton1** is selected.

5. In the toolbar, choose the **Add control to UI Control Map**.

   The status at the bottom of the window verifies the action by displaying **Selected control has been added to the UI control map**.

6. On the UIMap – Coded UI Test Builder, choose **Generate Code**.

   The Coded UI Test Builder – Generate Code appears with a note indicating that no new method is required and that code will only be generated for the changes to the UI control map.

7. Choose **Generate**.

8. Close SimpleWPFApp.exe.

9. Close UIMap – Coded UI Test Builder.

   The UIMap – Coded UI Test Builder takes a few seconds to process the UI control map changes.

10. In Solution Explorer, open the UIMap.Designer.cs file.

11. In the UIMap.Designer.cs file, locate the UIStartButton1 property. Notice the `SearchProperties` is set to `"buttonA"`:

C#

```
public WpfButton UIStartButton1
        {
            get
            {
                if ((this.mUIStartButton1 == null))
                {
                    this.mUIStartButton1 = new WpfButton(this);
                    #region Search Criteria

this.mUIStartButton1.SearchProperties[WpfButton.PropertyNames.AutomationId] =
"buttonA";
                    this.mUIStartButton1.WindowTitles.Add("MainWindow");
                    #endregion
                }
                return this.mUIStartButton1;
            }
        }
```

Now you can modify the coded UI test to use the newly mapped control. As pointed out in the previous procedure if you want to override any methods or properties in the coded UI test, you must do so in the UIMap.cs file.

12. In the UIMap.cs file, add a constructor and specify the `SearchProperties` property of the `UIStartButton` property to use the `AutomationID` property with a value of `"buttonA":`

C#

```
public UIMap()
        {

this.UIMainWindowWindow.UIStartButton.SearchProperties[WpfButton.PropertyNames.Automat
= "buttonA";
        }
```

13. On the **BUILD** menu, choose **Build Solution**.

14. In Test Explorer, run CodedUITestMethod1.

This time, the coded UI test successfully completes all the steps in the test. In the Test Results Window, you will see a status of **Passed**.

External Resources

Videos

Coded UI Tests-DeepDive-Episode1-GettingStarted

Coded UI Tests-DeepDive-Episode2-MaintainenceAndDebugging

Coded UI Tests-DeepDive-Episode3-HandCoding

Hands on lab

MSDN Virtual Lab: Introduction to Creating Coded UI Tests with Visual Studio 2010

FAQ

Coded UI Tests FAQ - 1

Coded UI Tests FAQ -2

Forum

Visual Studio UI Automation Testing (includes CodedUI)

See Also

Use UI Automation To Test Your Code
Getting Started with the WPF Designer
Supported Configurations and Platforms for Coded UI Tests and Action Recordings
Editing Coded UI Tests Using the Coded UI Test Editor

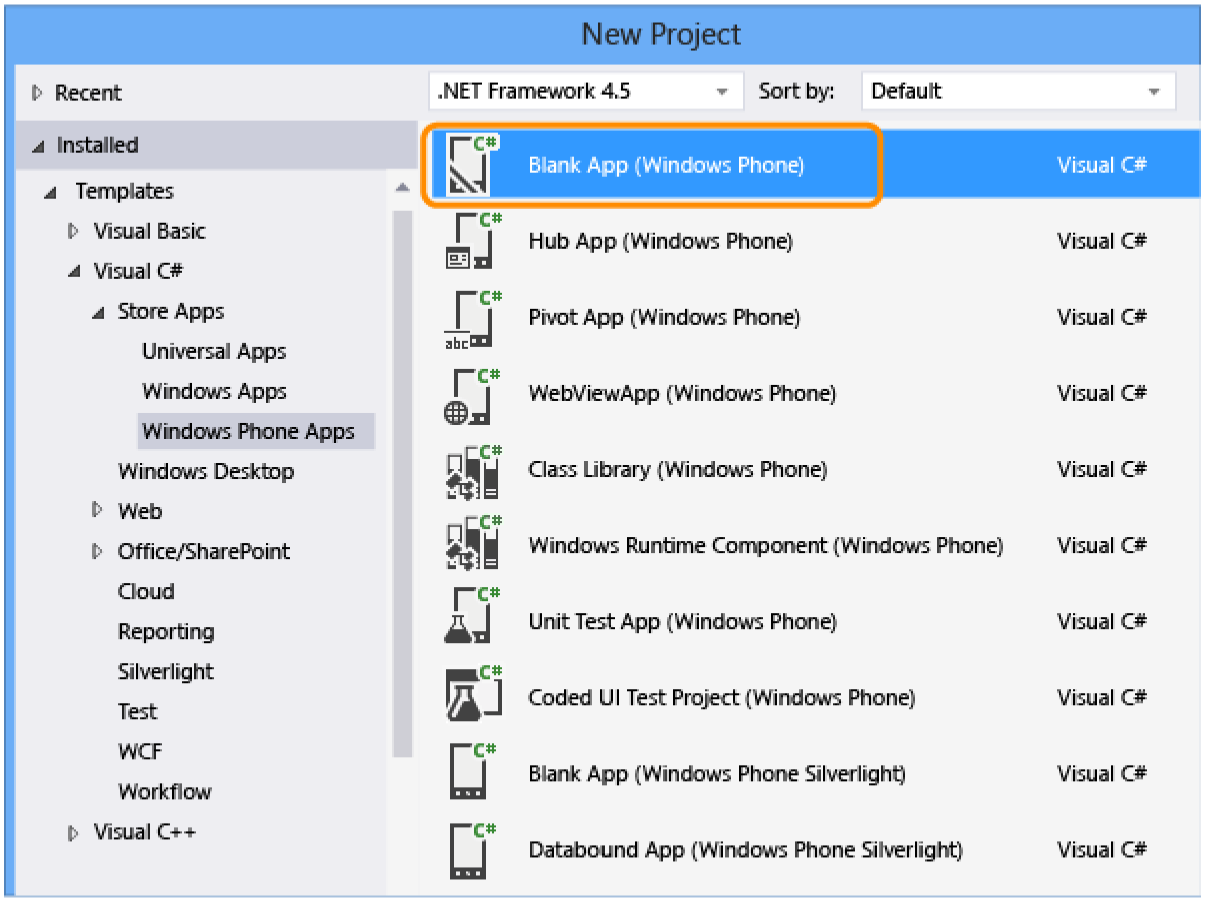
Test Windows Phone 8.1 Apps with Coded UI Tests

Visual Studio 2015

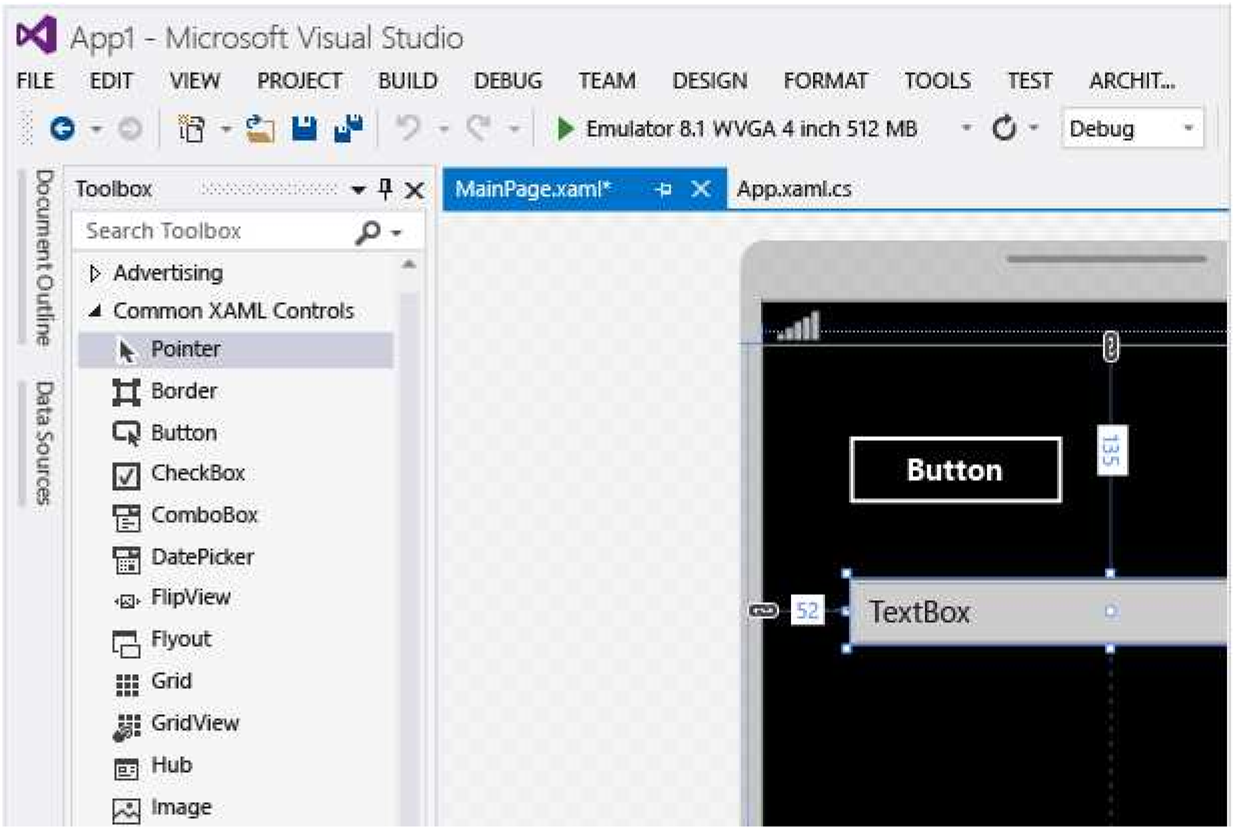
Use coded UI tests to test your Windows Phone apps.

Create a simple Windows Phone app

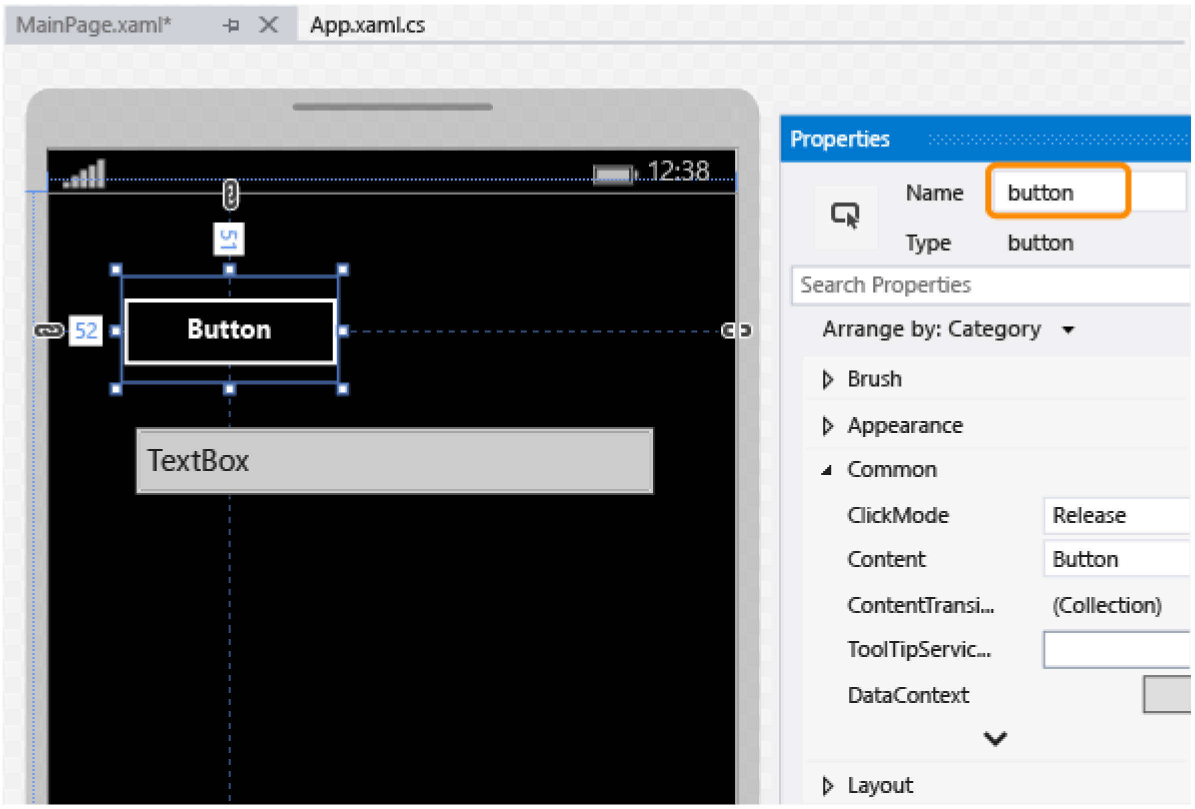
1. Create a new project for a blank Windows Phone app using either Visual C# or Visual Basic template.


New Project
Recent
.NET Framework 4.5
Sort by:
Default
Installed
Templates
Visual Basic
Visual C#
Store Apps
Universal Apps
Windows Apps
Windows Phone Apps
Windows Desktop
Web
Office/SharePoint
Cloud
Reporting
Silverlight
Test
WCF
Workflow
Visual C++
Blank App (Windows Phone) Visual C#
Hub App (Windows Phone) Visual C#
Pivot App (Windows Phone) Visual C#
WebViewApp (Windows Phone) Visual C#
Class Library (Windows Phone) Visual C#
Windows Runtime Component (Windows Phone) Visual C#
Unit Test App (Windows Phone) Visual C#
Coded UI Test Project (Windows Phone) Visual C#
Blank App (Windows Phone Silverlight) Visual C#
Databound App (Windows Phone Silverlight) Visual C#

2. In Solution Explorer, open MainPage.xaml. From the Toolbox, drag a button control and a textbox control to the design surface.

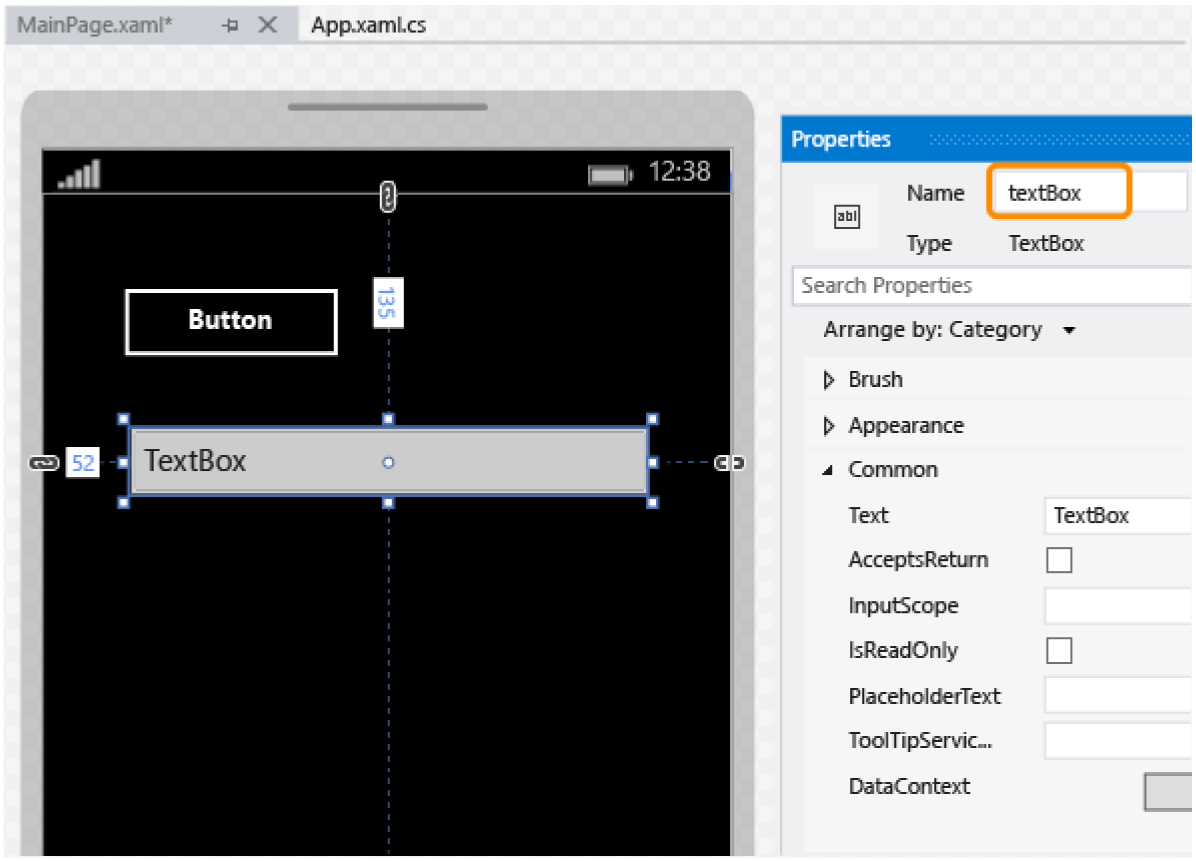


3. In the Properties window, name the button control.



Name: button
Type: button
ClickMode: Release
Content: Button

4. Name the textbox control.



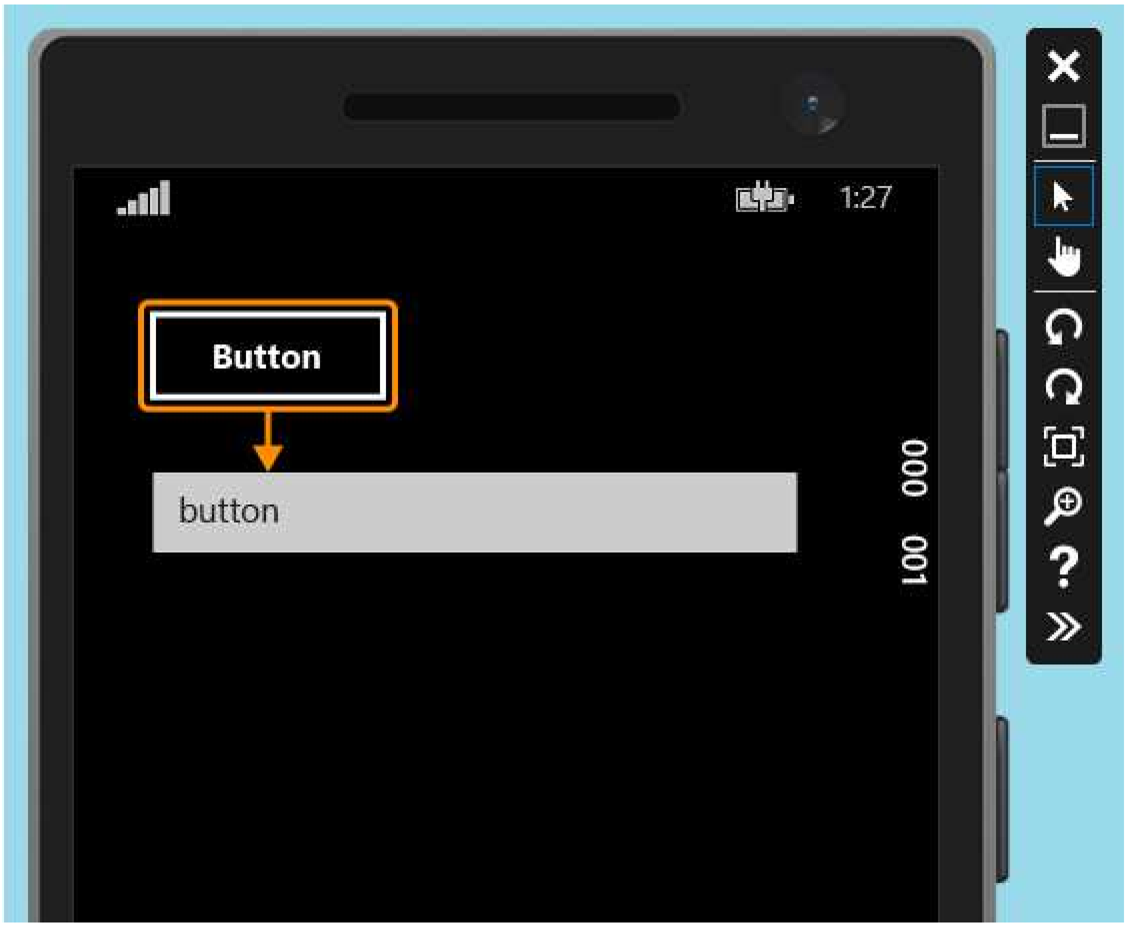
Name textBox
Type TextBox
Search Properties
Arrange by: Category
Brush
Appearance
Common
Text TextBox
AcceptsReturn
InputScope
IsReadOnly
PlaceholderText
ToolTipServic...
DataContext

5. On designer surface, double-click the button control and add the following code:

VB

```
Public NotInheritable Class MainPage
    Inherits Page

    Private Sub button_Click(sender As Object, e As RoutedEventArgs) Handles
Button.Click
        Me.textBox.Text = Me.button.Name
    End Sub
End Class
```

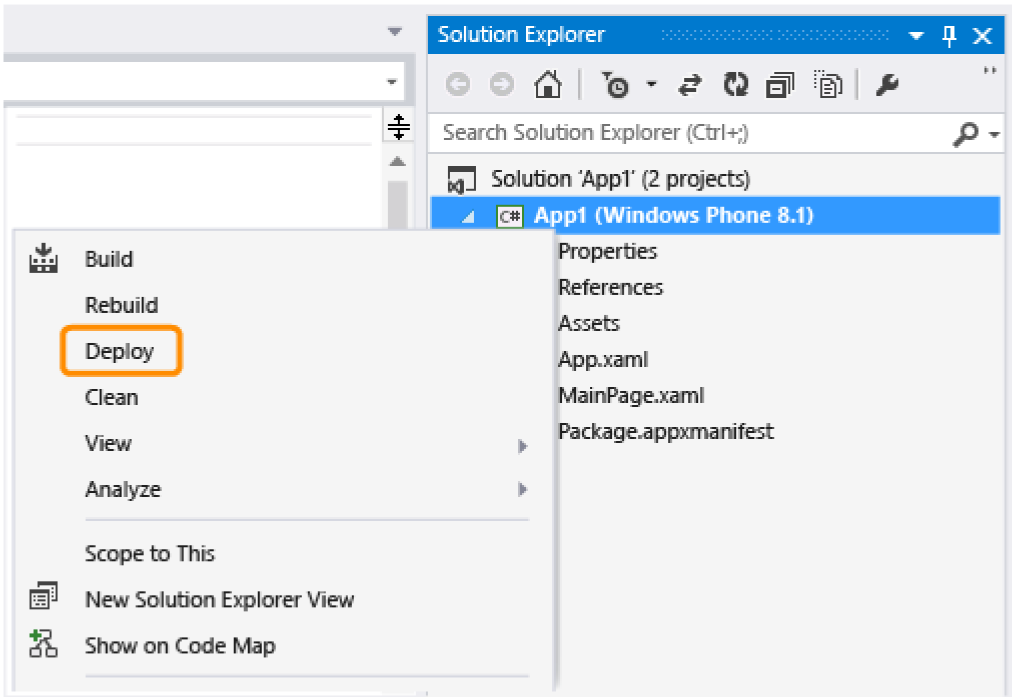
6. Press F5 to run your Windows Phone app in the emulator and verify that it's working.



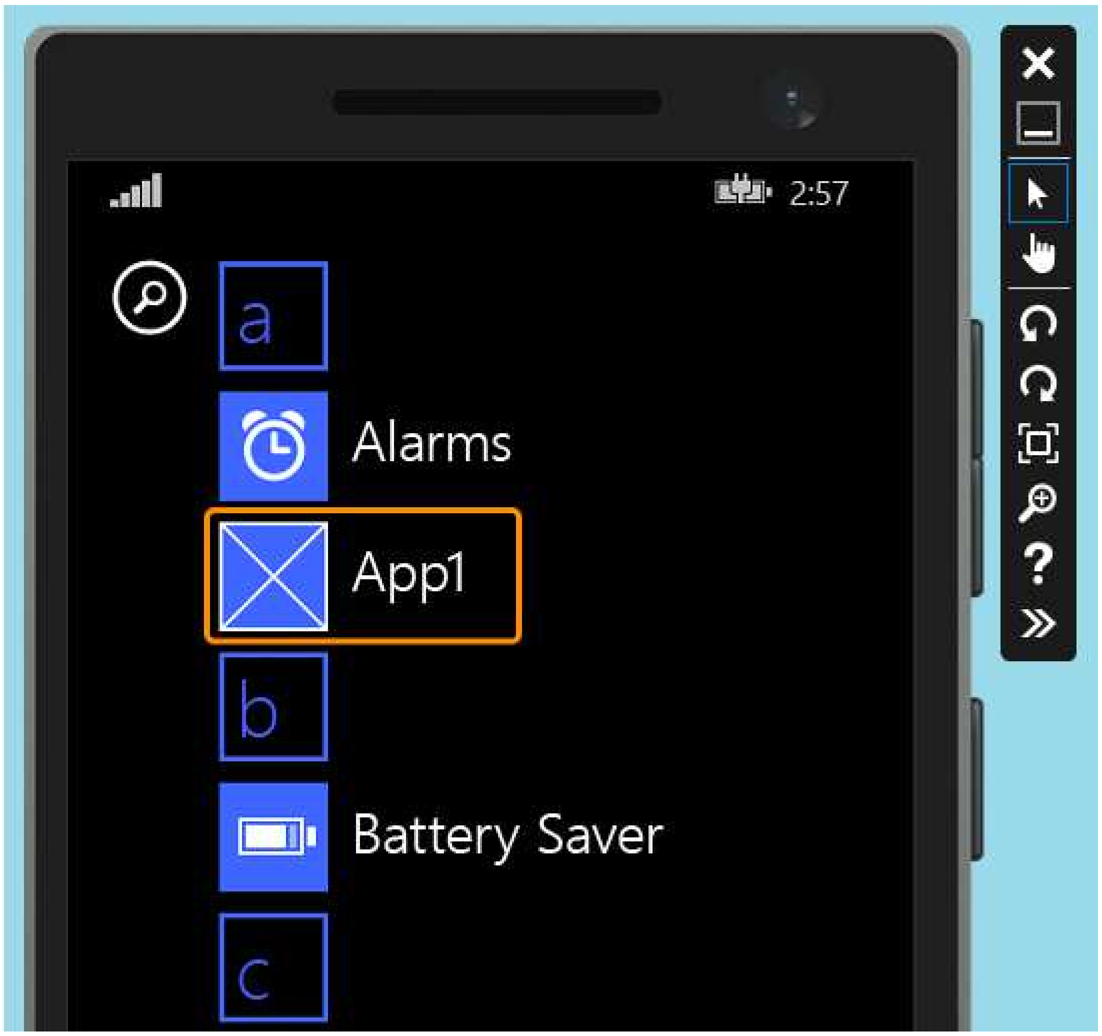
7. Exit the emulator.

Deploy the Windows Phone app

1. Before a coded UI test can map an app's controls, you have to deploy the app.



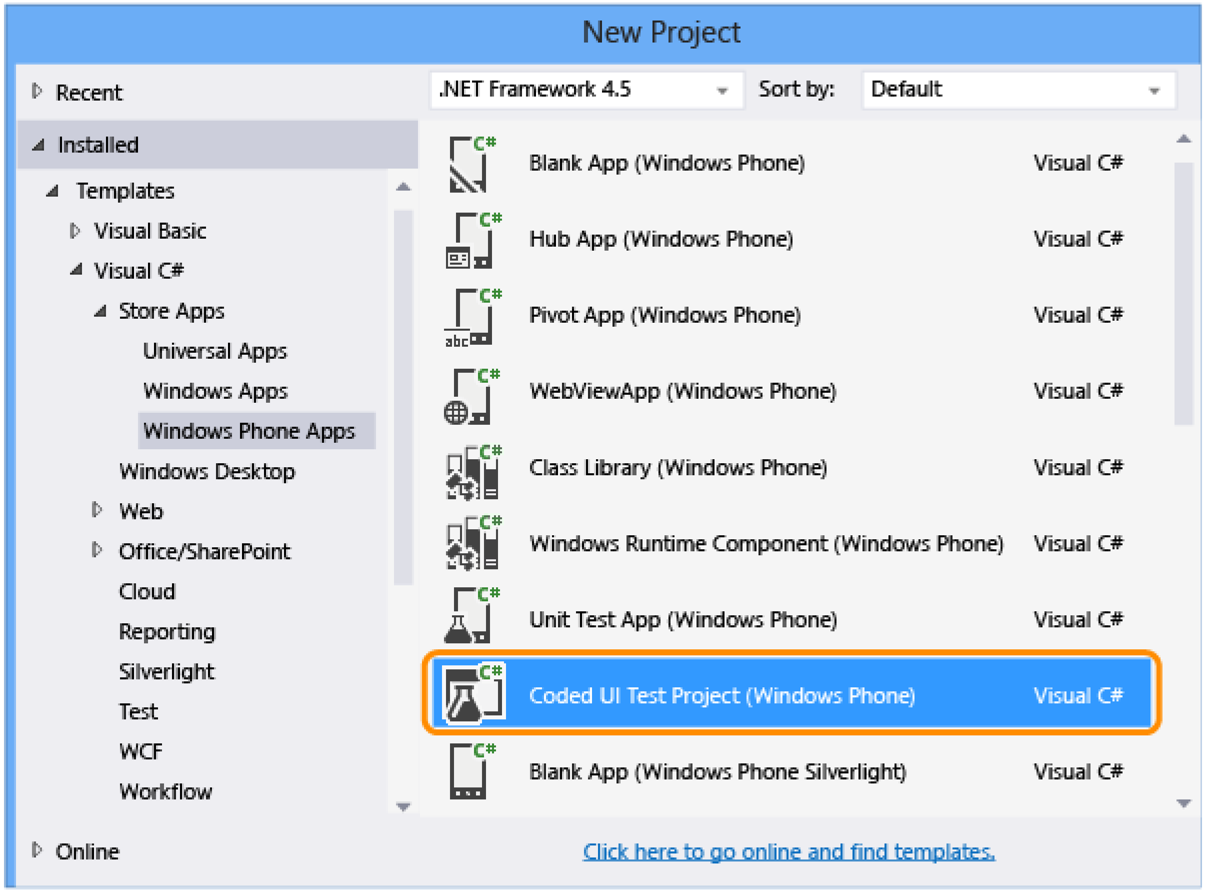
The emulator starts. The app is now available for testing.



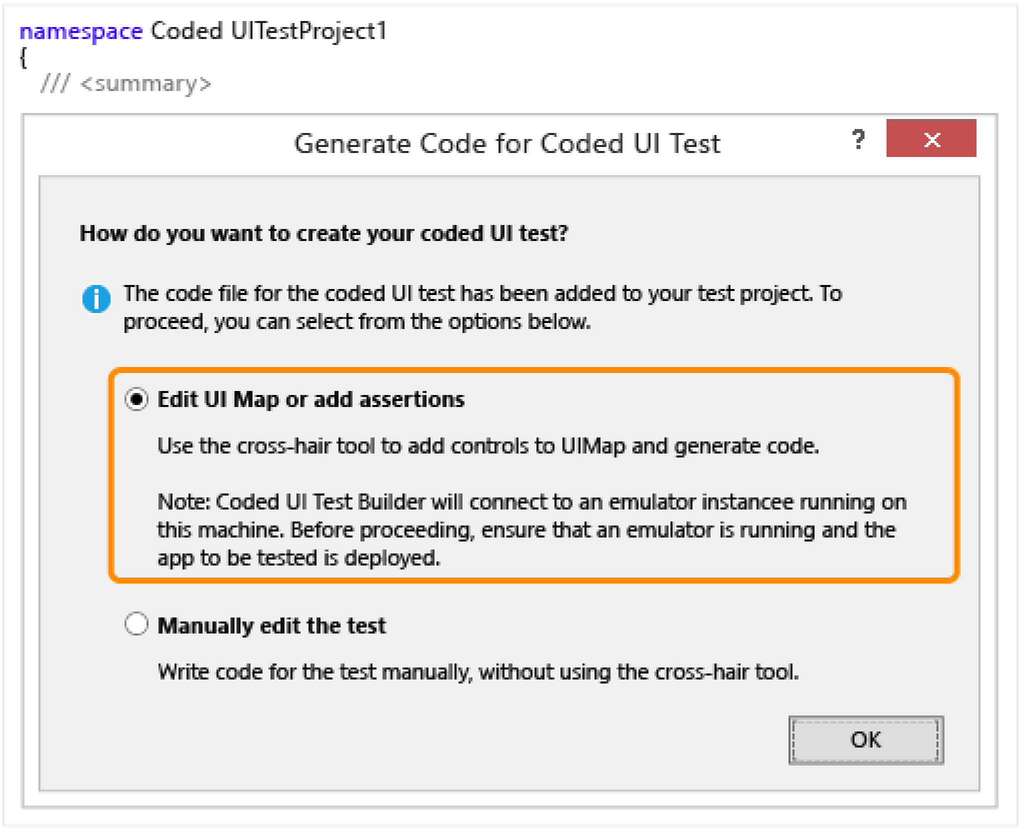
Keep the emulator running while you create your coded UI test.

Create a coded UI test for the Windows Phone app

1. Add a new coded UI test project to the solution with the Windows Phone app.


New Project
Recent
.NET Framework 4.5
Sort by:
Default
Installed
Templates
Visual Basic
Visual C#
Store Apps
Universal Apps
Windows Apps
Windows Phone Apps
Windows Desktop
Web
Office/SharePoint
Cloud
Reporting
Silverlight
Test
WCF
Workflow
Online
Blank App (Windows Phone) Visual C#
Hub App (Windows Phone) Visual C#
Pivot App (Windows Phone) Visual C#
WebViewApp (Windows Phone) Visual C#
Class Library (Windows Phone) Visual C#
Windows Runtime Component (Windows Phone) Visual C#
Unit Test App (Windows Phone) Visual C#
Coded UI Test Project (Windows Phone) Visual C#
Blank App (Windows Phone Silverlight) Visual C#
Click here to go online and find templates.

2. Choose to edit the UI map using the cross-hair tool.



```
namespace Coded UITestProject1
{
  /// <summary>
```

Generate Code for Coded UI Test

How do you want to create your coded UI test?

The code file for the coded UI test has been added to your test project. To proceed, you can select from the options below.

[✓ **Edit UI Map or add assertions**]

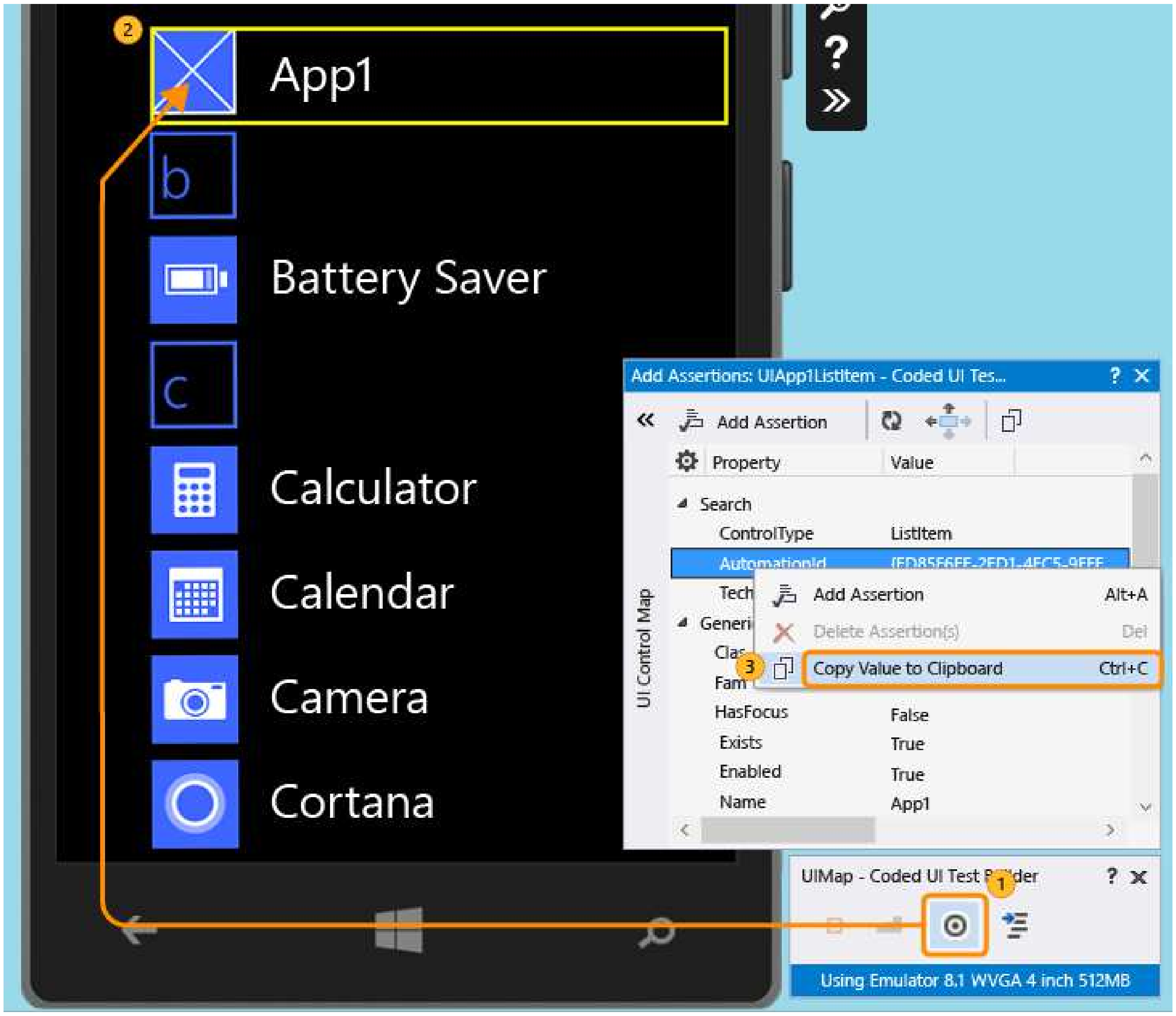
Use the cross-hair tool to add controls to UIMap and generate code.

Note: Coded UI Test Builder will connect to an emulator instancee running on this machine. Before proceeding, ensure that an emulator is running and the app to be tested is deployed.

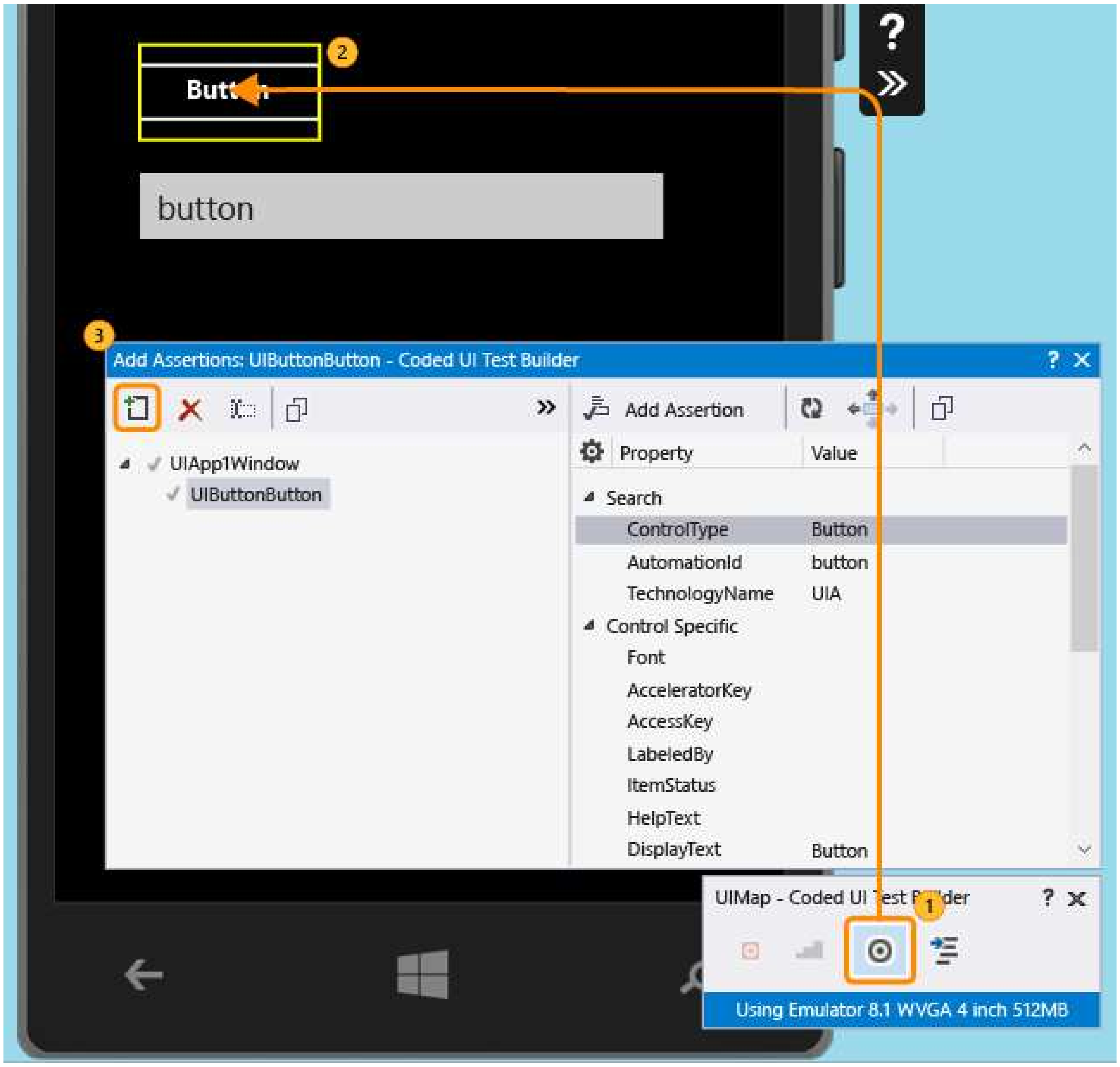
Manually edit the test

Write code for the test manually, without using the cross-hair tool.

3. Use the cross-hair tool to select the app, then copy the value for the app's **AutomationId** property, which will be used later to start the app in the test.

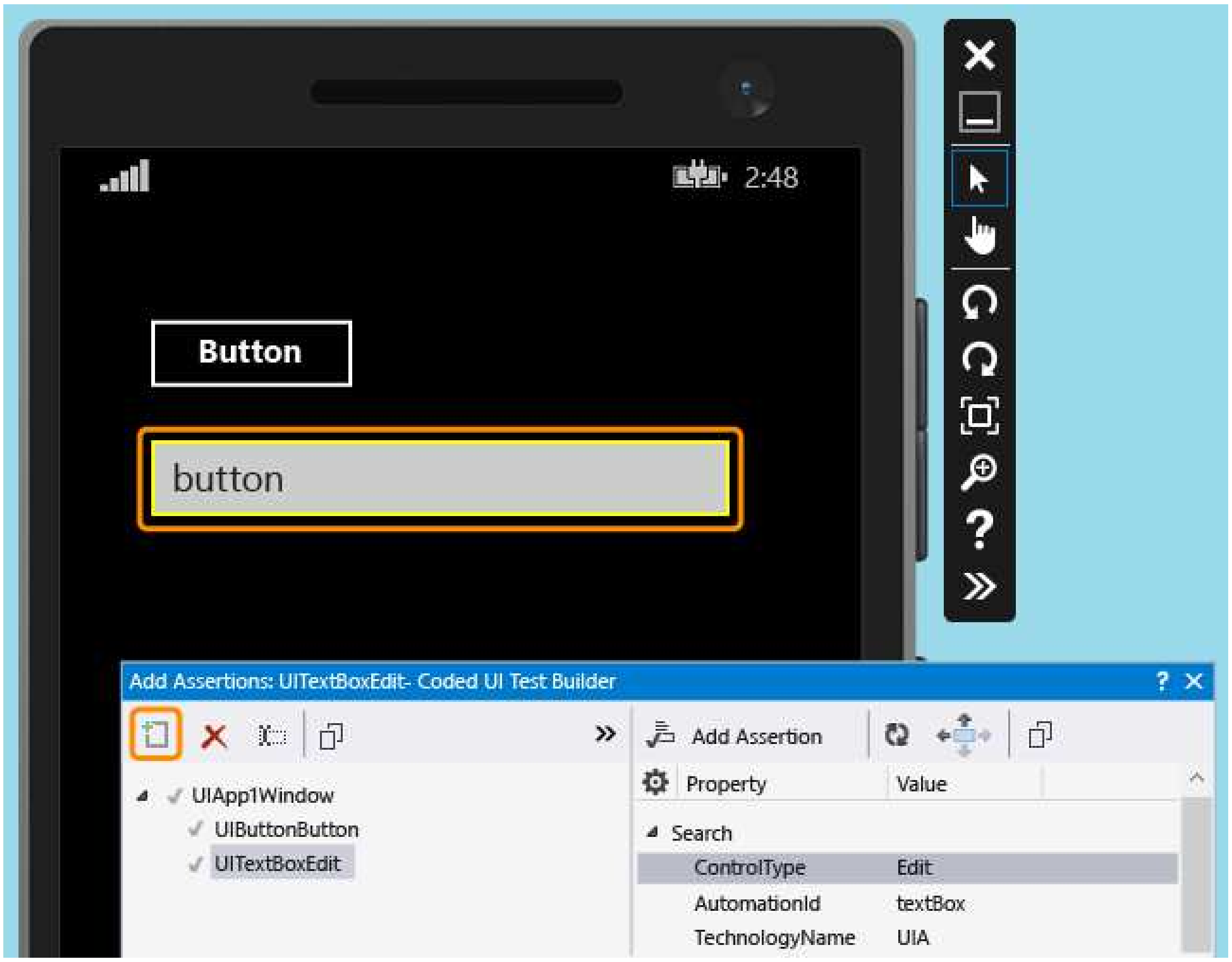


4. In the emulator, start the app and use the cross-hair tool to select the button control. Then add the button control to the UI control map.


Button
button
2
3
1
Add Assertions: UIButtonButton - Coded UI Test Builder
UIApp1Window
UIButtonButton
Add Assertion
UIMap - Coded UI Test Builder
Using Emulator 8.1 WVGA 4 inch 512MB

Property	Value
Search	
ControlType	Button
AutomationId	button
TechnologyName	UIA
Control Specific	
Font	
AcceleratorKey	
AccessKey	
LabeledBy	
ItemStatus	
HelpText	
DisplayText	Button

5. To add the textbox control to the UI control map, repeat the previous step.



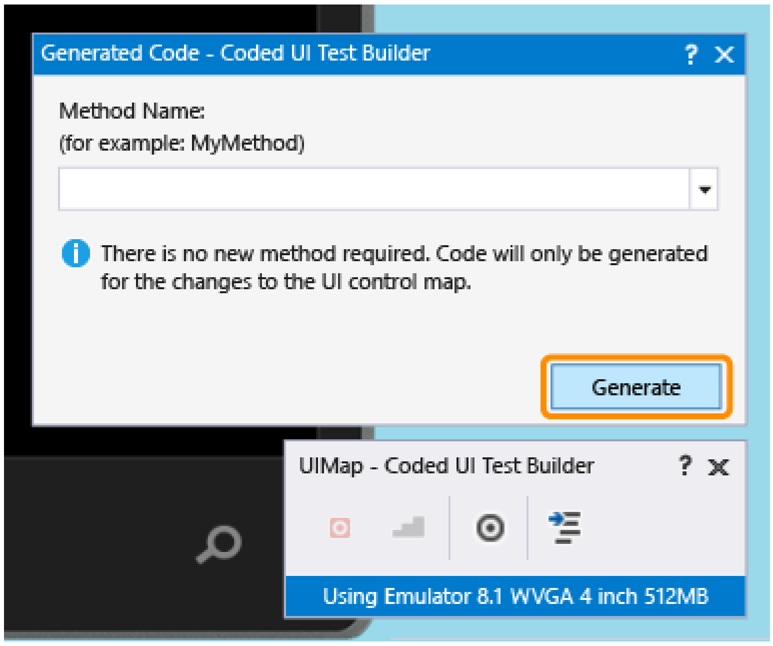
Add Assertions: UITextBoxEdit- Coded UI Test Builder

Add Assertion

- UIApp1Window
  - UIButtonButton
  - UITextBoxEdit

Property	Value
Search	
ControlType	Edit
AutomationId	textBox
TechnologyName	UIA

6. Generate code to create code for changes to the UI control map.



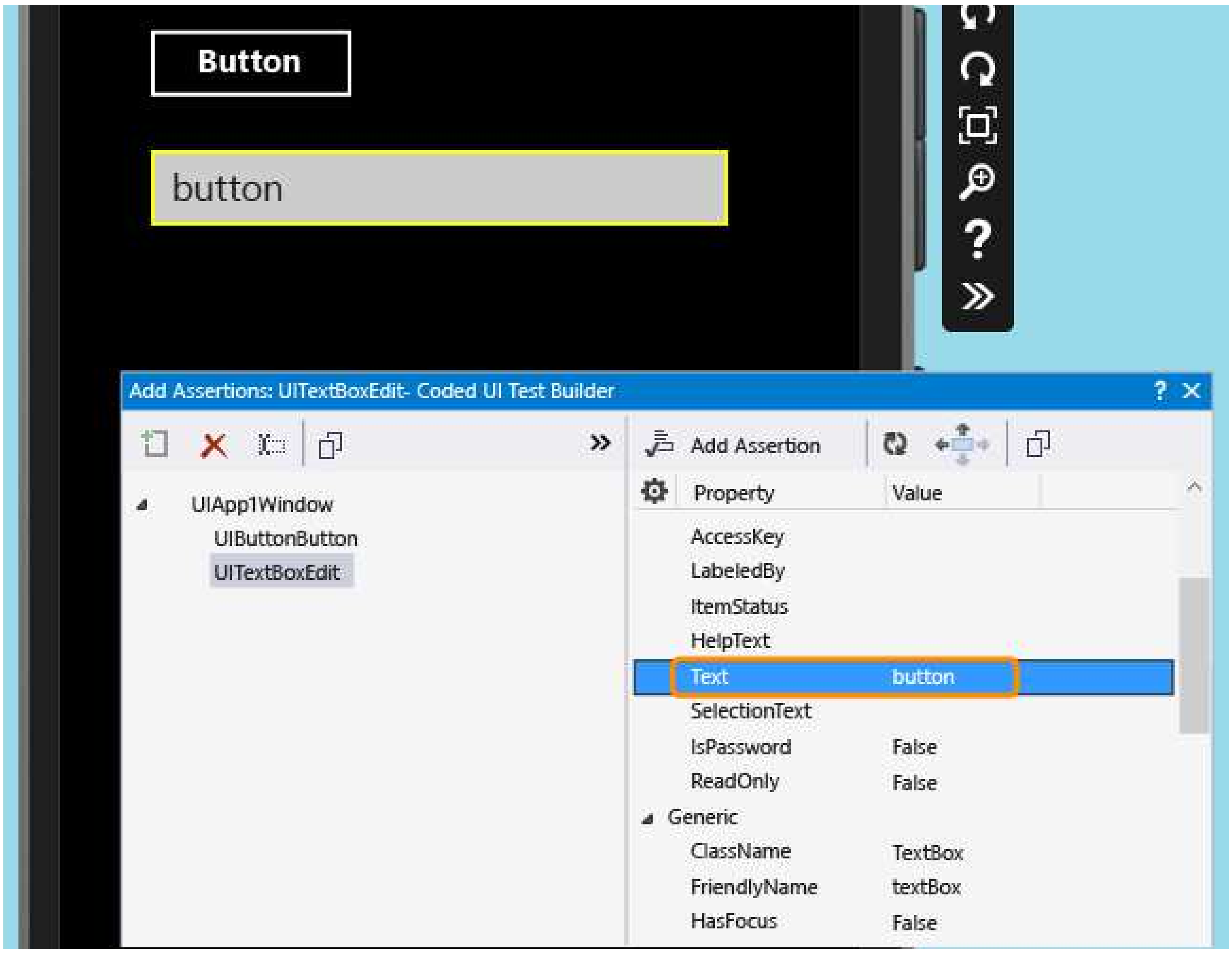
Generated Code - Coded UI Test Builder

Method Name:
(for example: MyMethod)

There is no new method required. Code will only be generated for the changes to the UI control map.

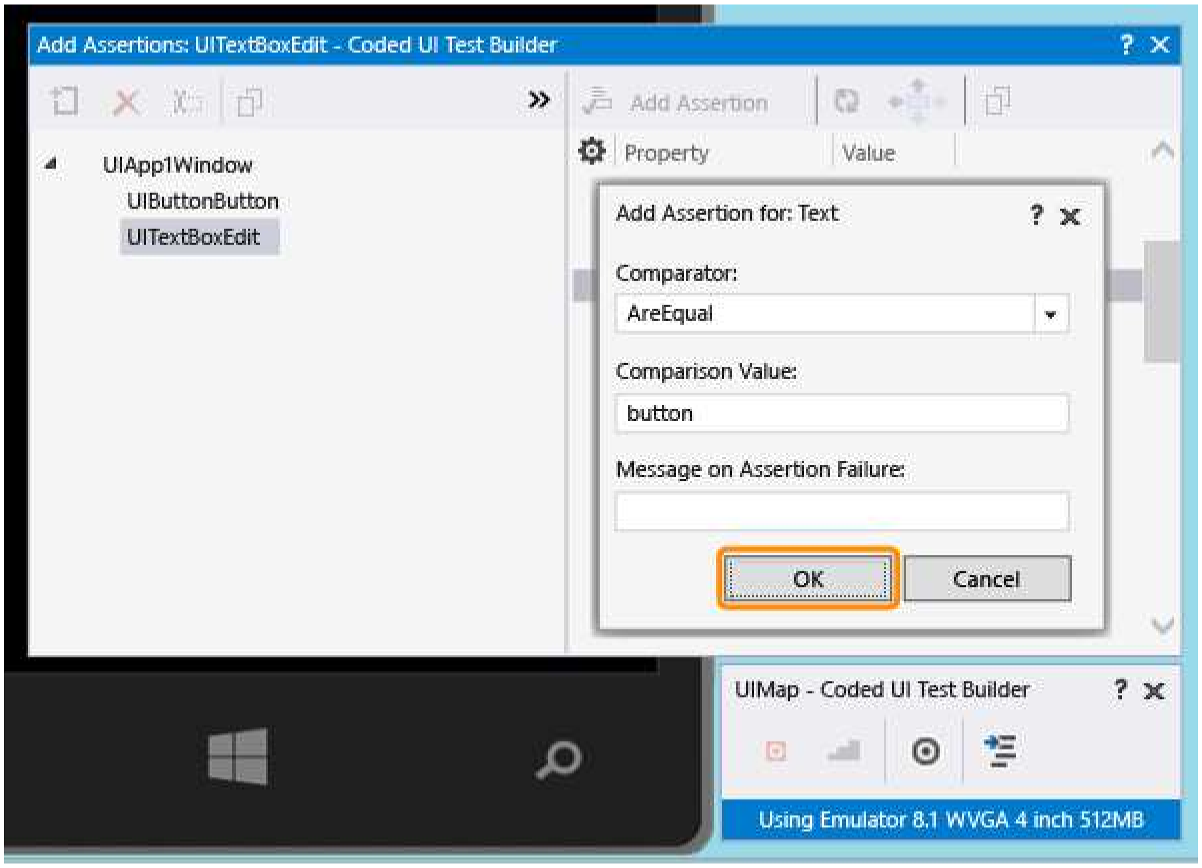
Generate
UIMap - Coded UI Test Builder
Using Emulator 8.1 WVGA 4 inch 512MB

7. Use the cross-hair tool to select the textbox control, and then select the **Text** property.



Property	Value
AccessKey	
LabeledBy	
ItemStatus	
HelpText	
Text	button
SelectionText	
IsPassword	False
ReadOnly	False
Generic	
ClassName	TextBox
FriendlyName	textBox
HasFocus	False

8. Add an assertion. It will be used in the test to verify that the value is correct.


Add Assertions: UITextBoxEdit - Coded UI Test Builder
Add Assertion
Property
Value
UIApp1Window
UIButtonButton
UITextBoxEdit
UIMap - Coded UI Test Builder
Using Emulator 8.1 WVGA 4 inch 512MB

Add Assertion for: Text

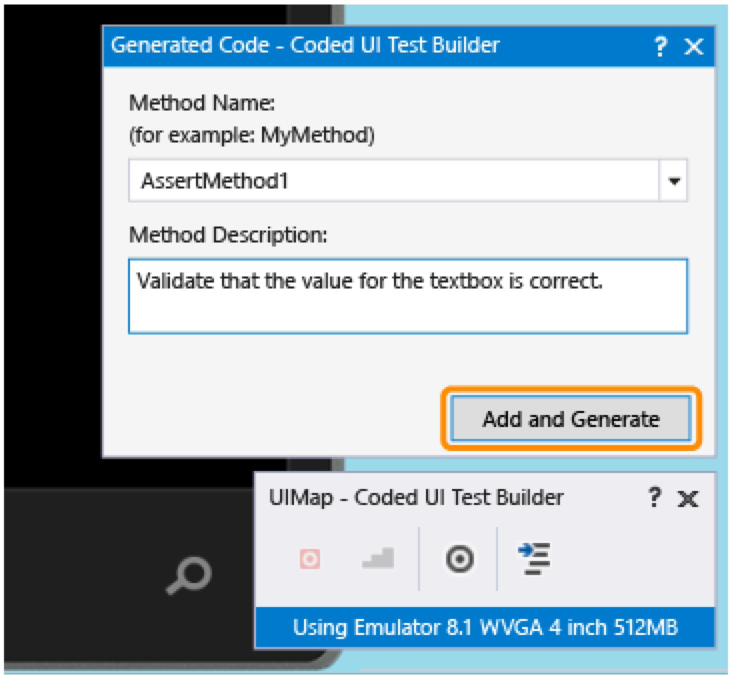
Comparator: AreEqual

Comparison Value: button

Message on Assertion Failure:

OK Cancel

9. Add and generate code for the assert method.


Generated Code - Coded UI Test Builder
UIMap - Coded UI Test Builder
Using Emulator 8.1 WVGA 4 inch 512MB

Method Name:
(for example: MyMethod)

AssertMethod1

Method Description:

Validate that the value for the textbox is correct.

Add and Generate

10. **Visual C#**

    In Solution Explorer, open the UIMap.Designer.cs file to view the code you just added for the assert method and the controls.

    Visual Basic

    In Solution Explorer, open the CodedUITest1.vb file. In the CodedUITestMethod1() test method code, right-click the call to the assertion method that was automatically added `Me.UIMap.AssertMethod1()` and choose **Go To Definition**. This will open the UIMap.Designer.vb file in the code editor so you can view the code you added for the assert method and the controls.

Warning

Do not modify the UIMap.designer.cs or UIMap.Designer.vb file directly. If you do this, the changes to the file will be overwritten each time the test is generated.

Assert method

VB

```
Public Sub AssertMethod1()
    Dim uITextBoxEdit As XamlEdit = Me.UIApp1Window.UITextBoxEdit

    'Verify that the 'Text' property of 'textBox' text box equals 'button'
    Assert.AreEqual(Me.AssertMethod1ExpectedValues.UITextBoxEditText,
uITextBoxEdit.Text)
End Sub
```

Controls

VB

```
#Region "Properties"
Public ReadOnly Property UIButtonButton() As XamlButton
    Get
        If (Me.mUIButtonButton Is Nothing) Then
            Me.mUIButtonButton = New XamlButton(Me)

Me.mUIButtonButton.SearchProperties(XamlButton.PropertyNames.AutomationId) =
"button"
        End If
        Return Me.mUIButtonButton
    End Get
End Property

Public ReadOnly Property UITextBoxEdit() As XamlEdit
    Get
        If (Me.mUITextBoxEdit Is Nothing) Then
            Me.mUITextBoxEdit = New XamlEdit(Me)

Me.mUITextBoxEdit.SearchProperties(XamlEdit.PropertyNames.AutomationId) = "textBox"
        End If
        Return Me.mUITextBoxEdit
    End Get
End Property
#End Region
```

```
#Region "Fields"
Private mUIButtonButton As XamlButton

Private mUITextBoxEdit As XamlEdit
#End Region
```

11. In Solution Explorer, open the CodedUITest1.cs or CodedUITest1.vb file. You can now add code to the CodedUTTestMethod1 method for the actions needed to run the test. Use the controls that were added to the UIMap to add code:

    a. Launch the Windows Phone app using the automation ID property you copied to the clipboard previously:

    VB

    ```
    XamlWindow.Launch("ed85f6ff-2fd1-4ec5-9eef-696026c3fa7b_cyrqexqw8cc7c!App");
    ```

    b. Add a gesture to tap the button control:

    VB

    ```
    Gesture.Tap(Me.UIMap.UIApp1Window.UIButtonButton)
    ```

    c. Verify that the call to the assert method that was automatically generated comes after launching the app and tap gesture on the button:

    VB

    ```
    Me.UIMap.AssertMethod1()
    ```

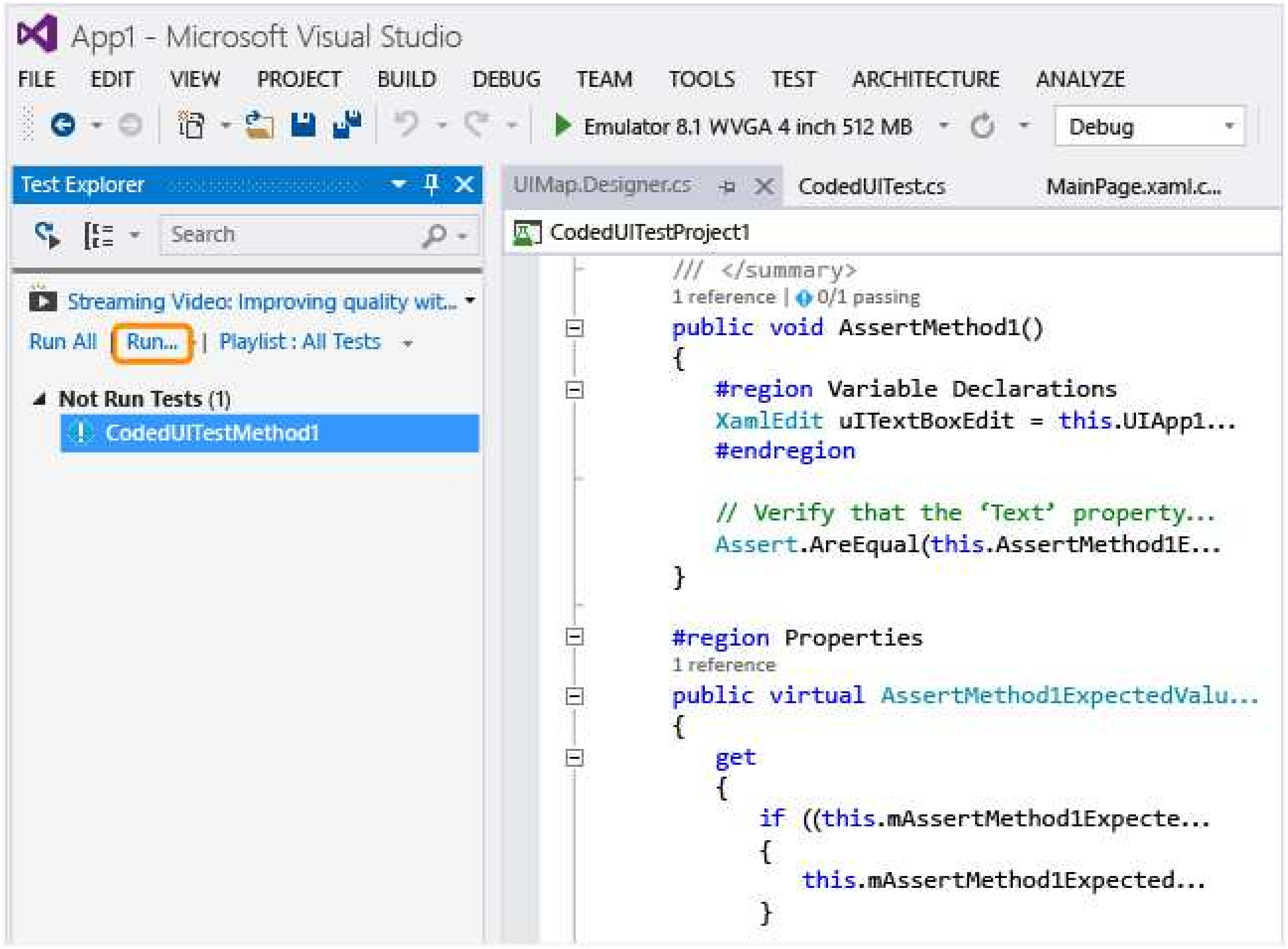
After the code is added, the CodedUITestMethod1 test method should appear as follows:

VB

```
<CodedUITest>
Public Class CodedUITest1

    <TestMethod()>
    Public Sub CodedUITestMethod1()
        '
        ' To generate code for this test, select "Generate Code for Coded UI Test"
from the shortcut menu and select one of the menu items.
        '
        ' Launch the app.
        XamlWindow.Launch("ed85f6ff-2fd1-4ec5-9eef-696026c3fa7b_cyrqexqw8cc7c!App")

        '// Tap the button.
        Gesture.Tap(Me.UIMap.UIApp1Window.UIButtonButton)

        Me.UIMap.AssertMethod1()
    End Sub
```

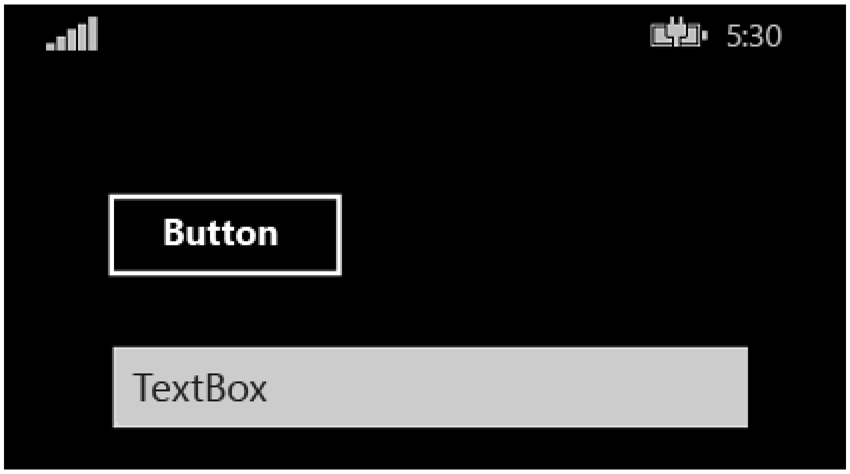
Run the coded UI test

1. Build your test and then run the test using the test explorer.

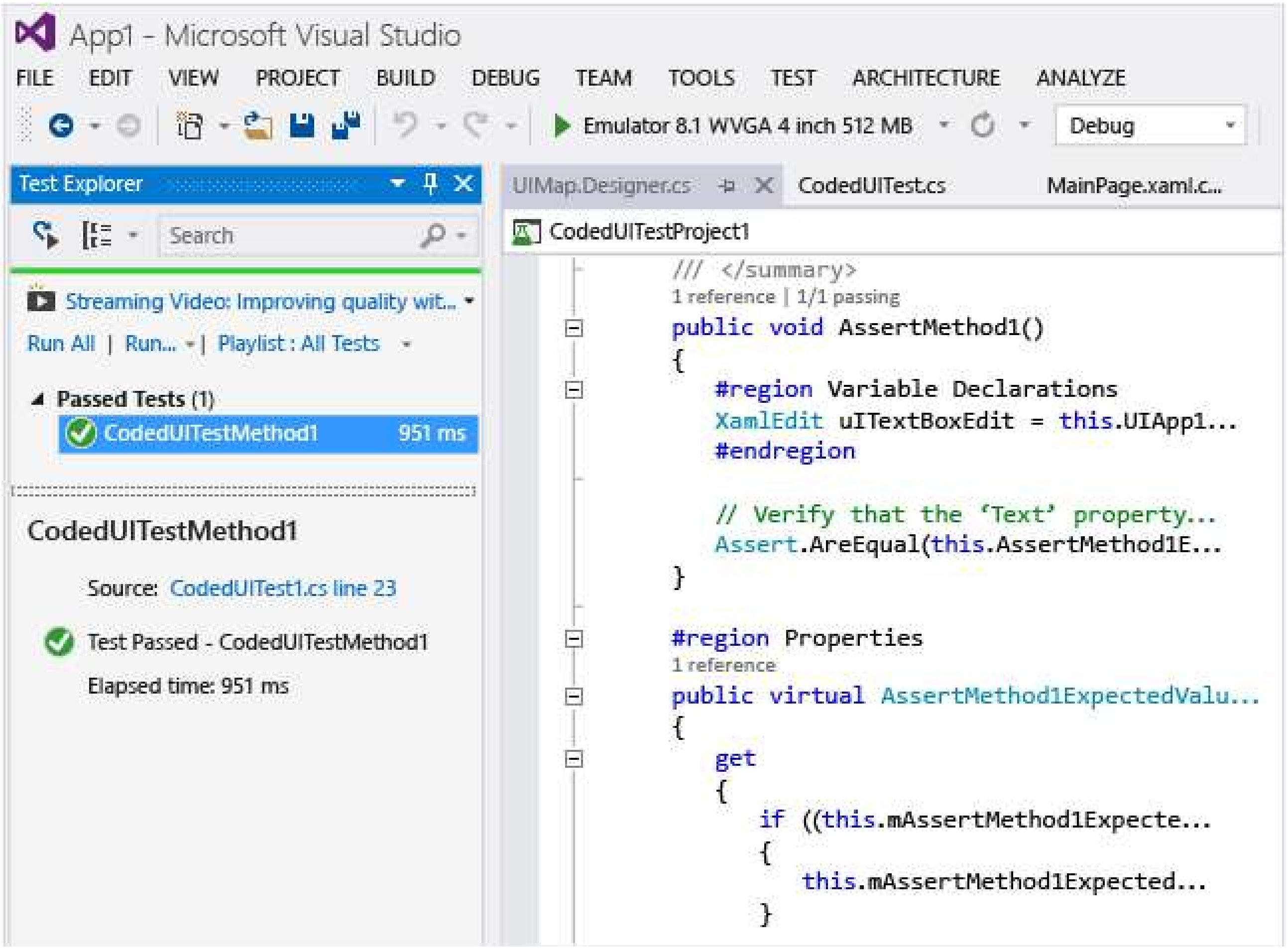

App1 - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE
Emulator 8.1 WVGA 4 inch 512 MB
Debug
Test Explorer
Search
Streaming Video: Improving quality wit...
Run All | Run... | Playlist : All Tests
Not Run Tests (1)
CodedUITestMethod1
UIMap.Designer.cs
CodedUITest.cs
MainPage.xaml.c...
CodedUITestProject1

```
        /// </summary>
        1 reference | 0/1 passing
        public void AssertMethod1()
        {
            #region Variable Declarations
            XamlEdit uITextBoxEdit = this.UIApp1...
            #endregion

            // Verify that the 'Text' property...
            Assert.AreEqual(this.AssertMethod1E...
        }

        #region Properties
        1 reference
        public virtual AssertMethod1ExpectedValu...
        {
            get
            {
                if ((this.mAssertMethod1Expecte...
                {
                    this.mAssertMethod1Expected...
                }
```

The Windows Phone app launches, the action to tap the button is completed, and the textbox's Text property is populated and validated using the assert method.



After the test is finished, the test explorer confirms that the test passed.


App1 - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE
Emulator 8.1 WVGA 4 inch 512 MB
Debug
Test Explorer
Search
UIMap.Designer.cs
CodedUITest.cs
MainPage.xaml.c...
CodedUITestProject1
Streaming Video: Improving quality wit...
Run All | Run... | Playlist : All Tests
Passed Tests (1)
CodedUITestMethod1 951 ms

CodedUITestMethod1

Source: CodedUITest1.cs line 23

Test Passed - CodedUITestMethod1

Elapsed time: 951 ms

```
        /// </summary>
        1 reference | 1/1 passing
        public void AssertMethod1()
        {
            #region Variable Declarations
            XamlEdit uITextBoxEdit = this.UIApp1...
            #endregion

            // Verify that the 'Text' property...
            Assert.AreEqual(this.AssertMethod1E...
        }

        #region Properties
        1 reference
        public virtual AssertMethod1ExpectedValu...
        {
            get
            {
                if ((this.mAssertMethod1Expecte...
                {
                    this.mAssertMethod1Expected...
                }
```

Use Data-driven coded UI tests on Windows Phone apps

To test different conditions, a coded UI test can be run multiple times with different sets of data.

Data-driven Coded UI tests for Windows Phone are defined using the DataRow attribute on a test method. In the following example, x and y use the values of 1 and 2 for the first iteration and -1 and -2 for the second iteration of the test.

```
[DataRow(1, 2, DisplayName = "Add positive numbers")]
[DataRow(-1, -2, DisplayName = "Add negative numbers")]
[TestMethod]
public void DataDrivingDemo_MyTestMethod(int x, int y)
```

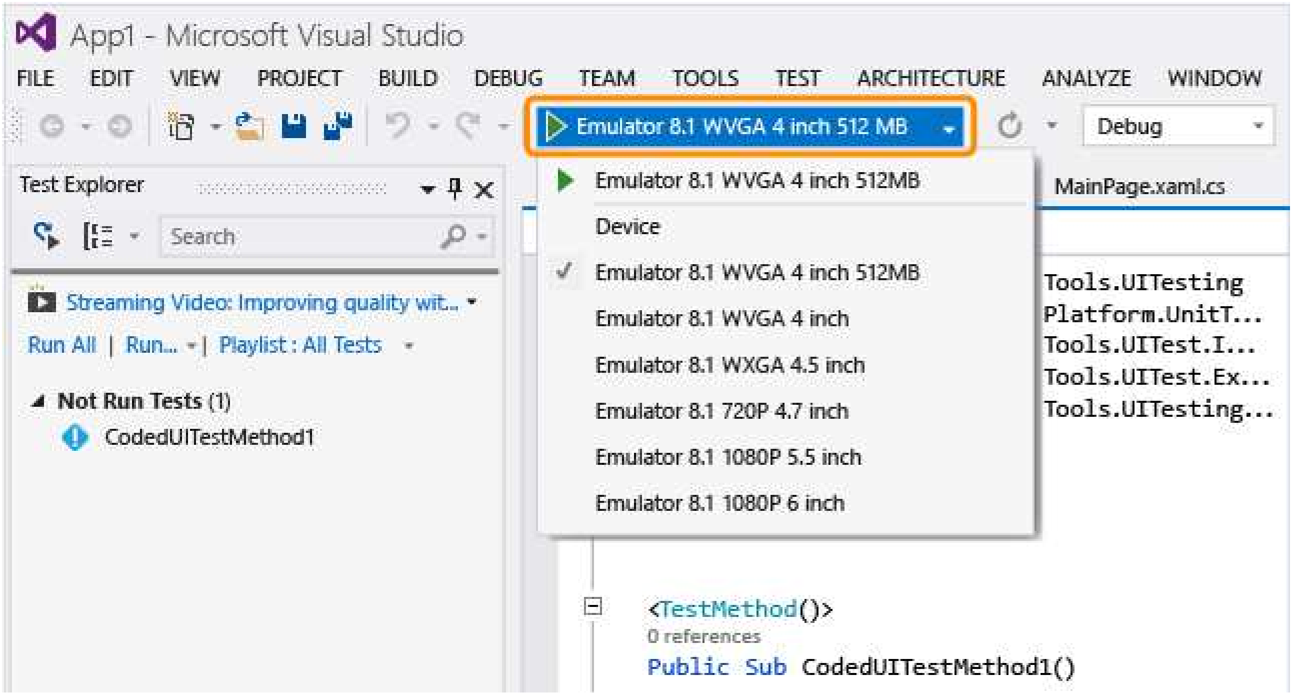
Q & A

Q: Do I have to deploy the Windows Phone app in the emulator in order to map UI controls?

A: Yes, the coded UI test builder requires that an emulator be running and the app be deployed to it. Otherwise, it will throw an error message saying that no running emulator could be found.

Q: Can tests be executed on the emulator only, or can I also use a physical device?

A: Either option is supported. The target for test execution is selected by changing the emulator type or selecting device in the device toolbar. If Device is selected, a Phone Blue device needs to be connected to one of the machine's USB ports.


App1 - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW
Emulator 8.1 WVGA 4 inch 512 MB
Debug
Test Explorer
Search
Streaming Video: Improving quality wit...
Run All | Run... | Playlist : All Tests
Not Run Tests (1)
CodedUITestMethod1
Emulator 8.1 WVGA 4 inch 512MB
Device
Emulator 8.1 WVGA 4 inch 512MB
Emulator 8.1 WVGA 4 inch
Emulator 8.1 WXGA 4.5 inch
Emulator 8.1 720P 4.7 inch
Emulator 8.1 1080P 5.5 inch
Emulator 8.1 1080P 6 inch
MainPage.xaml.cs

```
Tools.UITesting
Platform.UnitT...
Tools.UITest.I...
Tools.UITest.Ex...
Tools.UITesting...

<TestMethod()>
0 references
Public Sub CodedUITestMethod1()
```

Q: Why don't I see the option to record my coded UI test in the Generate Code for a Coded UI Test dialog?

A: The option to record is not supported for Windows Phone apps.

Q: Can I create a coded UI test for my Windows Phone apps based on WinJS, Silverlight or HTML5?

A: No, only XAML based apps are supported.

Q: Can I create coded UI tests for my Windows Phone apps on a system that is not running Windows 8.1 or Windows 10?

A: No, the Coded UI Test Project templates are only available on Windows 8.1 and Windows 10. To create automation for Universal Windows Platform (UWP) apps, you'll need Windows 10.

Q: How do I create coded UI tests for Universal Windows Platform (UWP) apps?

A: Depending on the platform where you're testing your UWP app, create coded UI test project in one of these ways:

- A UWP app running on local machine will run as a Store app. To test this, you must use the **Coded UI Test**

Project (Windows) template. To find this template when you create a new project, go to the **Windows**, **Universal** node. Or go to the **Windows**, **Windows 8**, **Windows** node.

- A UWP app running on mobile device or emulator will run as a Phone app. To test this, you must use the **Coded UI Test Project (Windows Phone)** template. To find this template when you create a new project, go to the **Windows**, **Universal** node. Or go to the **Windows**, **Windows 8**, **Windows Phone** node.

After you create the project, authoring a test stays the same as before.

Q: Can I select controls that are outside the emulator?

A: No, the builder will not detect them.

Q: Can I use the coded UI test builder to map controls using a physical phone device?

A: No, The builder can only map UI elements if your app has been deployed to the emulator.

Q: Why can't I modify the code in the UIMap.Designer file?

A: Any code changes you make in the UIMapDesigner.cs file will be overwritten every time you generate code using the UIMap - Coded UI Test Builder. If you have to modify a recorded method, you must copy it to UIMap.cs file and rename it. The UIMap.cs file can be used to override methods and properties in the UIMapDesigner.cs file. You must remove the reference to the original method in the Coded UITest.cs file and replace it with the renamed method name.

Q: Can I run a coded UI test on my Windows Phone app from the command-line?

A: Yes, you use a runsettings file to specify the target device for test execution. For example:

vstest.console.exe "pathToYourCodedUITestDll" /settings:devicetarget.runsettings

Sample runsettings file:

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
<MSPhoneTest>
<!--to specify test execution on device, use a TargetDevice option as follows-->
<TargetDevice>Device</TargetDevice>
<!--to specify an emulator instead, use a TargetDevice option like below-->
<!--<TargetDevice>Emulator 8.1 WVGA 4 inch 512MB</TargetDevice>-->
</MSPhoneTest>
</RunSettings>
```

Q: What are the differences between coded UI tests for XAML-based Windows Store apps and Windows Phone apps?

A: These are some of the key differences:

Feature	Windows Store apps	Windows Phone apps
Target for running tests	Local or remote computer. Remote computers can be specified when you use an automated test case to run tests. See Automate a test case in Microsoft Test Manager.	Emulator or device. See, Q: Can tests be executed on the emulator only, or can I also use a physical device? in this topic.
Execute from the command-line	Settings file not required to specify target.	Runsettings file required to specify target.
Specialized classes for Shell Controls	T:Microsoft.VisualStudio.TestTools.UITesting.DirectUIControls.DirectUIControl	UITestControl
WebView control in a XAML app	Supported if you use Html* specialized classes to interact with HTML elements. See Microsoft.VisualStudio.TestTools.UITesting.HtmlControls.	Not supported.
Execute automated tests from MTM	Supported.	Not supported.
Data-driven tests	See Data-driven tests for information about using external data-sources and using DataSource attribute on a test method.	Data is specified inline, using DataRow attribute on a test method. See Use Data-driven coded UI tests on Windows Phone apps in this topic.

For information about coded UI tests for Windows Store apps, see Test Windows Store 8.1 Apps with Coded UI Tests.

External resources

Microsoft Visual Studio Application Lifecycle Management blog: Using Coded UI to test XAML-based Windows Phone apps

See Also

Use UI Automation To Test Your Code

Creating a Data-Driven Coded UI Test

Visual Studio 2015

To test different conditions, you can run your tests multiple times with different parameter values. Data-driven coded UI tests are a convenient way to do this. You define parameter values in a data source, and each row in the data source is an iteration of the coded UI test. The overall result of the test will be based on the outcome for all the iterations. For example, if one test iteration fails, the overall test result is failure.

Requirements

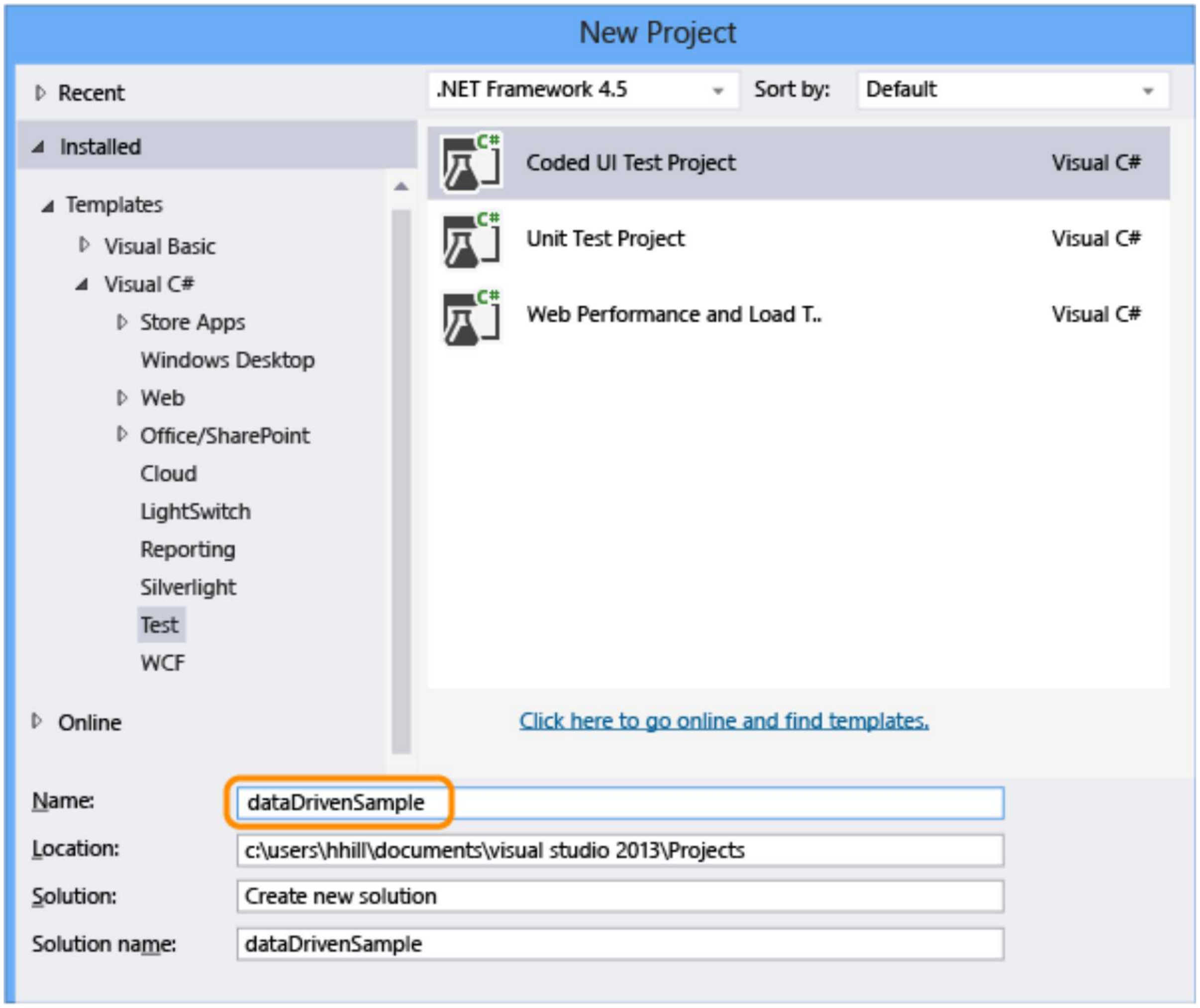
- Visual Studio Enterprise

Create a data-driven coded UI test

This sample creates a coded UI test that runs on the Windows Calculator application. It adds two numbers together and uses an assertion to validate that the sum is correct. Next, the assertion and the parameter values for the two numbers are coded to become data-driven and stored in a comma-separated value (.csv) file.

Step 1 - Create a coded UI test

1. Create a project.



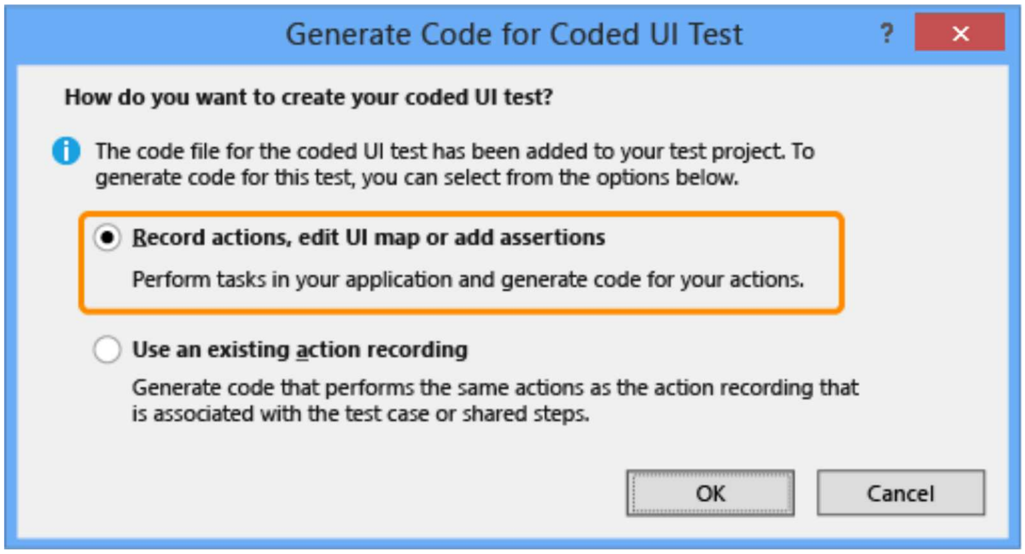
Name: dataDrivenSample

Location: c:\users\hhill\documents\visual studio 2013\Projects

Solution: Create new solution

Solution name: dataDrivenSample

2. Choose to record the actions.



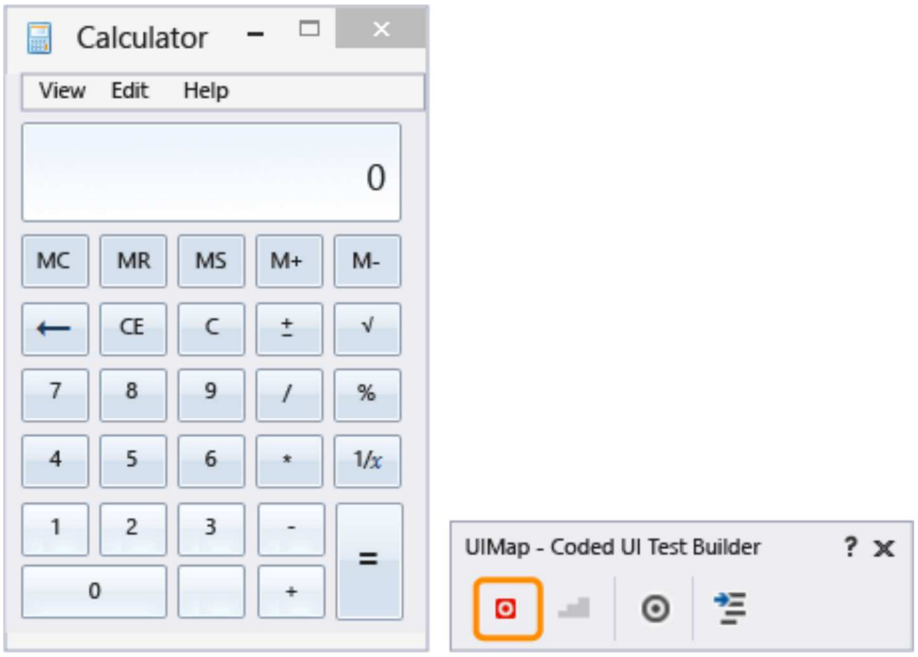
How do you want to create your coded UI test?

The code file for the coded UI test has been added to your test project. To generate code for this test, you can select from the options below.

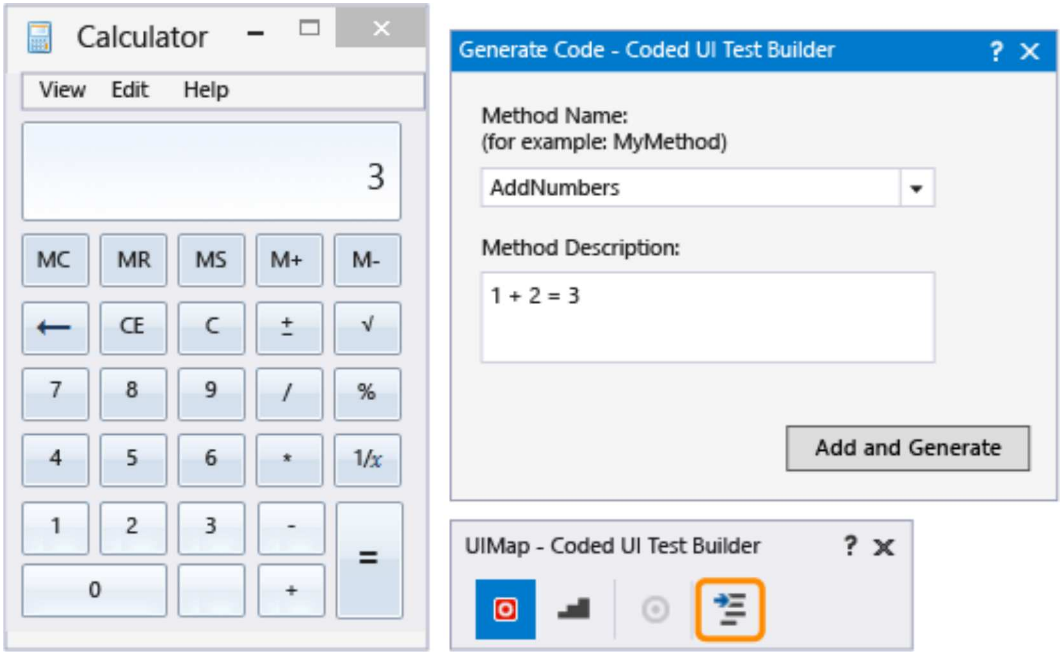
[✓ **Record actions, edit UI map or add assertions**]
Perform tasks in your application and generate code for your actions.

Use an existing action recording
Generate code that performs the same actions as the action recording that is associated with the test case or shared steps.

3. Open the calculator app and start recording the test.



4. Add 1 plus 2, pause the recorder, and generate the test method. Later we'll replace the values of this user input with values from a data file.



Method Name:
(for example: MyMethod)
AddNumbers

Method Description:
1 + 2 = 3

Close the test builder. The method is added to the test:

C#

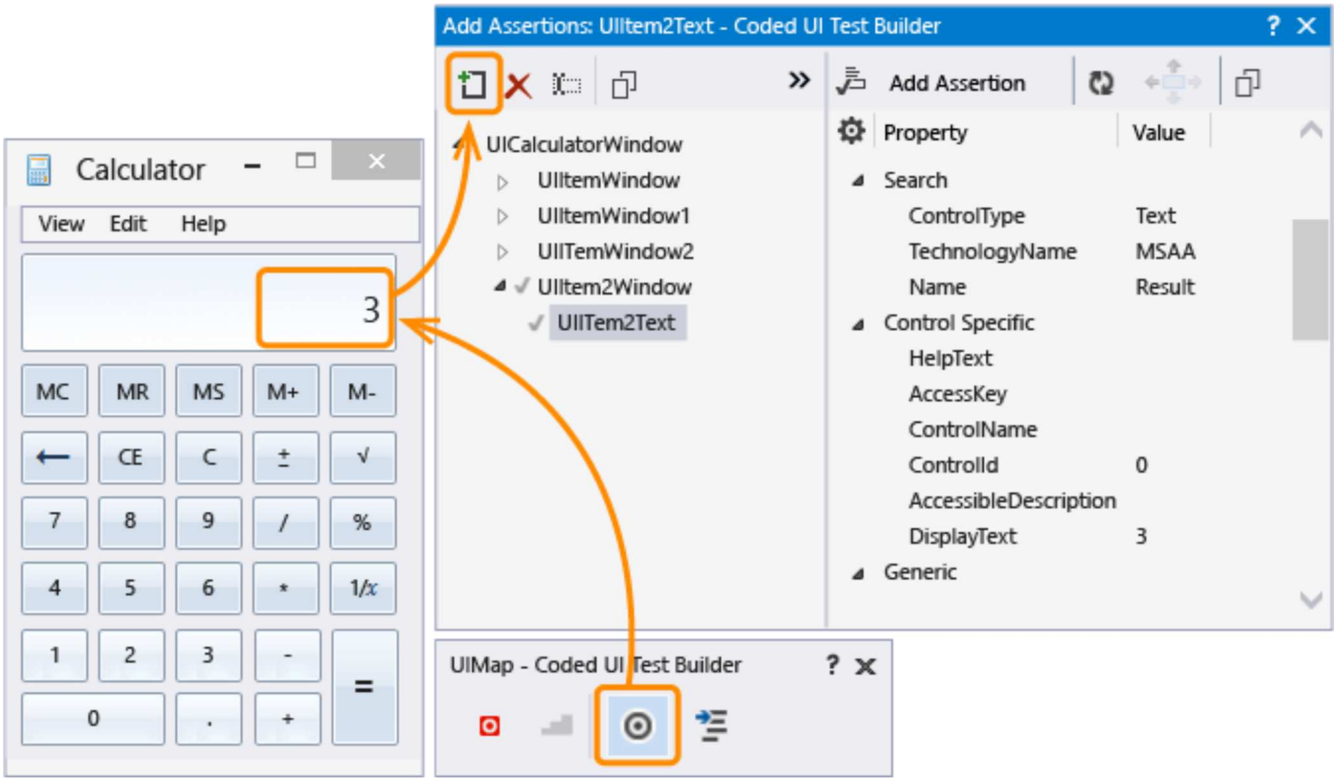
```
[TestMethod]
public void CodedUITestMethod1()
{
    // To generate code for this test, select "Generate Code for Coded UI Test"
from the shortcut menu and select one of the menu items.
    this.UIMap.AddNumbers();

}
```

5. Use the `AddNumbers()` method to verify that the test runs. Place the cursor in the test method shown above, open the context menu, and choose **Run Tests**. (Keyboard shortcut: Ctrl + R, T).

   The test result that shows if the test passed or failed is displayed in the Test Explorer window. To open the Test Explorer window, from the **TEST** menu, choose **Windows** and then choose **Test Explorer**.

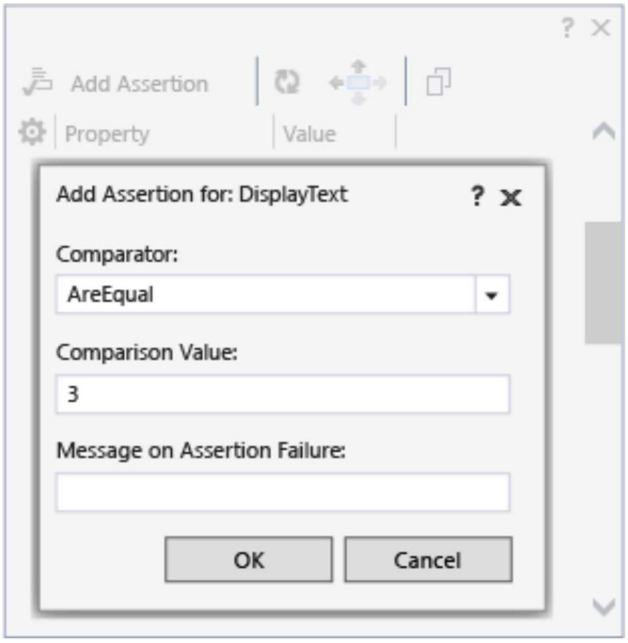
6. Because a data source can also be used for assertion parameter values—which are used by the test to verify expected values—let's add an assertion to validate that the sum of the two numbers is correct. Place the cursor in the test method shown above, open the context menu and choose **Generate Code for Coded UI Test**, and then **Use Coded UI Test Builder**.

   Map the text control in the calculator that displays the sum.



Property	Value
Search	
ControlType	Text
TechnologyName	MSAA
Name	Result
Control Specific	
HelpText	
AccessKey	
ControlName	
ControlId	0
AccessibleDescription	
DisplayText	3
Generic	

7. Add an assertion that validates that the value of the sum is correct. Choose the **DisplayText** property that has the value of **3** and then choose **Add Assertion**. Use the **AreEqual** comparator and verify that the comparison value is **3**.



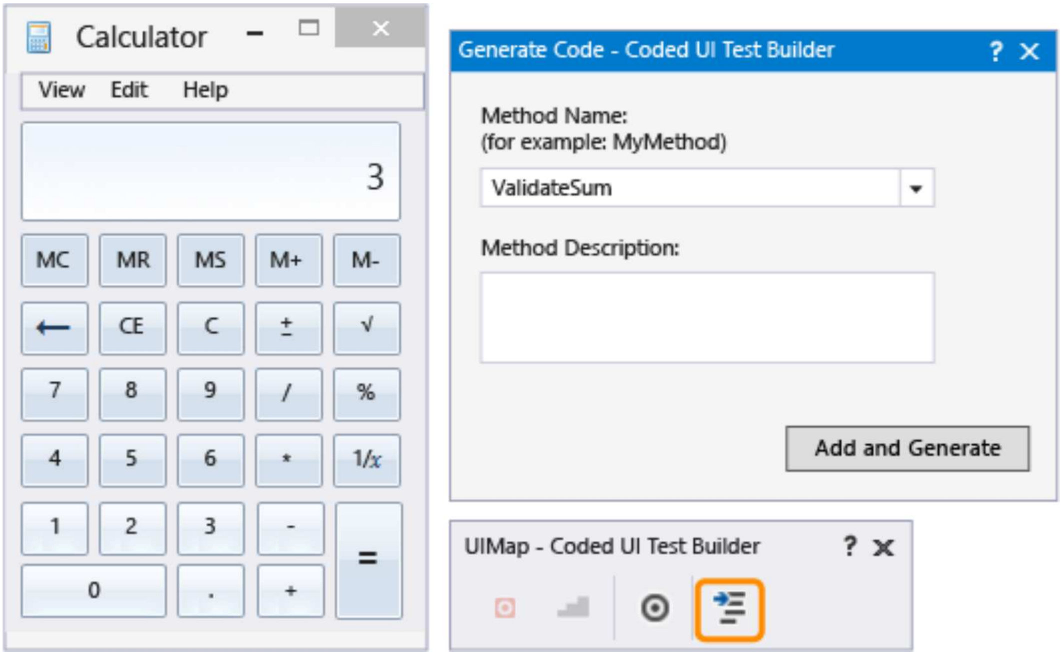
Add Assertion for: DisplayText

Comparator: AreEqual

Comparison Value: 3

Message on Assertion Failure:

8. After configuring the assertion, generate code from the builder again. This creates a new method for the validation.



Method Name:
(for example: MyMethod)

ValidateSum

Method Description:

Because the `ValidateSum` method validates the results of the `AddNumbers` method, move it to the bottom of the code block.

C#

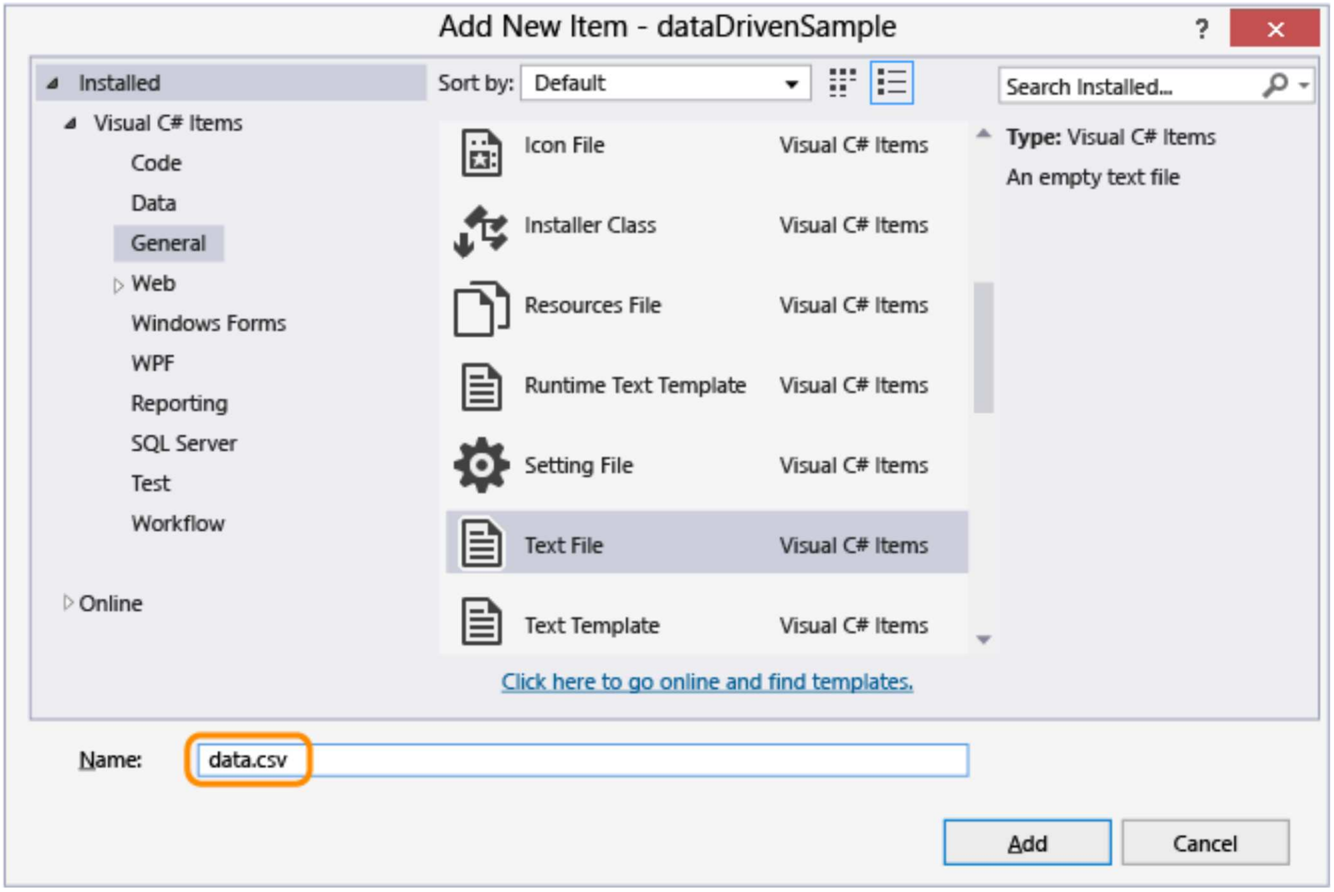
```
public void CodedUITestMethod1()
{

    // To generate code for this test, select "Generate Code for Coded UI Test"
from the shortcut menu and select one of the menu items.
    this.UIMap.AddNumbers();
    this.UIMap.ValidateSum();

}
```

9. Verify that the test runs by using the `ValidateSum()` method. Place the cursor in the test method shown above, open the context menu, and choose **Run Tests**. (Keyboard shortcut:Ctrl + R, T).

   At this point, all the parameter values are defined in their methods as constants. Next, let's create a data set to make our test data-driven.

Step 2 - Create a data set

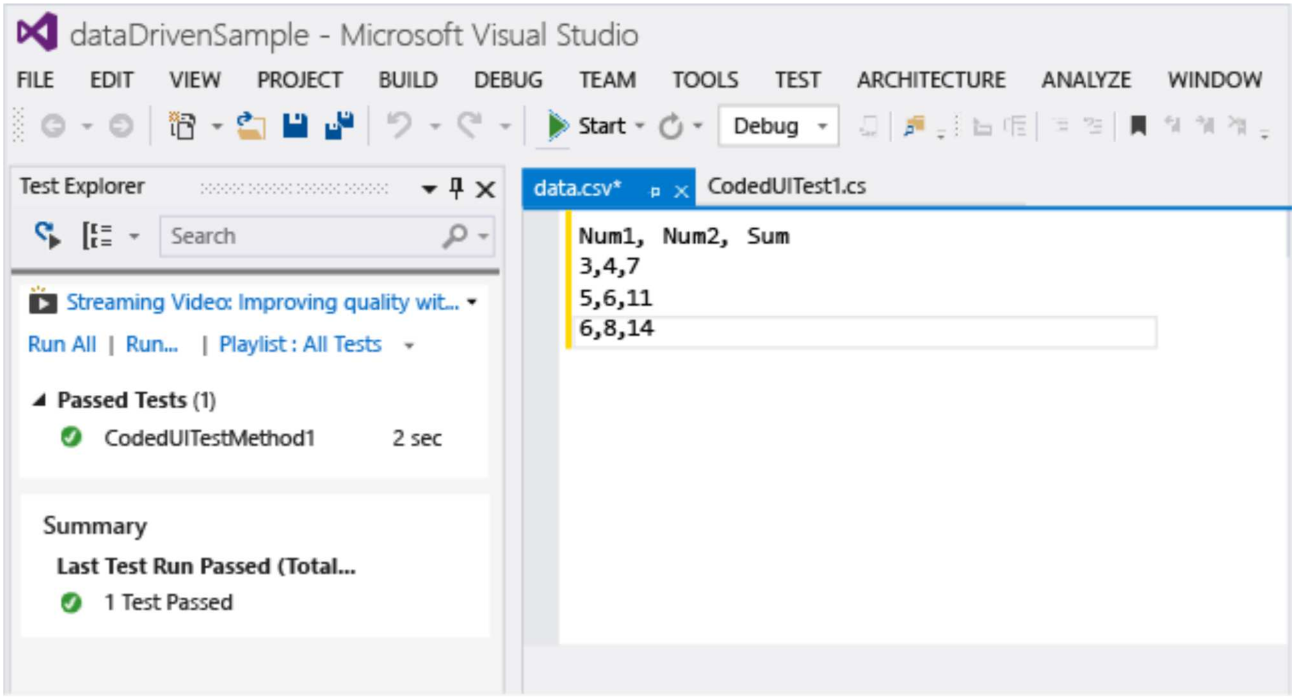
1. Add a text file to the dataDrivenSample project named **data.csv**.


Add New Item - dataDrivenSample
Installed
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
SQL Server
Test
Workflow
Online
Sort by: Default
Search Installed...
Icon File Visual C# Items
Installer Class Visual C# Items
Resources File Visual C# Items
Runtime Text Template Visual C# Items
Setting File Visual C# Items
Text File Visual C# Items
Text Template Visual C# Items
Type: Visual C# Items
An empty text file
Click here to go online and find templates.
Name: data.csv
Add
Cancel

2. Populate the .csv file with the following data:

Num1	Num2	Sum
3	4	7
5	6	11
6	8	14

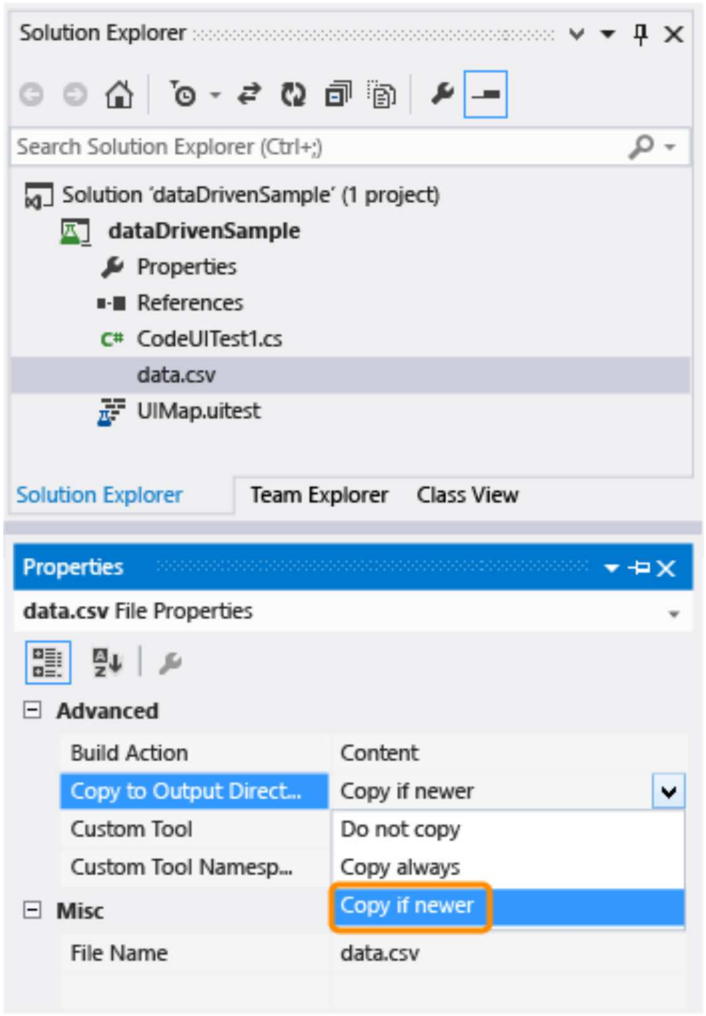
   After adding the data, the file should appear as the following:


dataDrivenSample - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW
Start
Debug
Test Explorer
Search
Streaming Video: Improving quality wit...
Run All | Run... | Playlist : All Tests
Passed Tests (1)
CodedUITestMethod1 2 sec
Summary
Last Test Run Passed (Total...
1 Test Passed
data.csv*
CodedUITest1.cs

```
Num1, Num2, Sum
3,4,7
5,6,11
6,8,14
```

3. It is important to save the .csv file using the correct encoding. On the **FILE** menu, choose **Advanced Save Options** and choose **Unicode (UTF-8 without signature) – Codepage 65001** as the encoding.

4. The .csv file, must be copied to the output directory, or the test can't run. Use the Properties window to copy it.



Build Action	Content
Copy to Output Direct...	Copy if newer
Custom Tool	
Custom Tool Namesp...	
File Name	data.csv

Now that we have the data set created, let's bind the data to the test.

Step 3 – Add data source binding

1. To bind the data source, add a `DataSource` attribute within the existing `[TestMethod]` attribute that is immediately above the test method.

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
"|DataDirectory|\\data.csv", "data#csv", DataAccessMethod.Sequential),
DeploymentItem("data.csv"), TestMethod]
public void CodedUITestMethod1()
{

    // To generate code for this test, select "Generate Code for Coded UI Test"
from the shortcut menu and select one of the menu items.
    this.UIMap.AddNumbers();
    this.UIMap.ValidateSum();

}
```

   The data source is now available for you to use in this test method.

   Tip

   See data source attribute samples in the Q & A section for samples of using other data source types such as XML, SQL Express and Excel.

2. Run the test.

   Notice that the test runs through three iterations. This is because the data source that was bound contains three rows of data. However, you will also notice that the test is still using the constant parameter values and is adding 1 + 2 with a sum of 3 each time.

   Next, we'll configure the test to use the values in the data source file.

Step 4 – Use the data in the coded UI test

1. Add `using Microsoft.VisualStudio.TestTools.UITesting.WinControls` to the top of the CodedUITest.cs file:

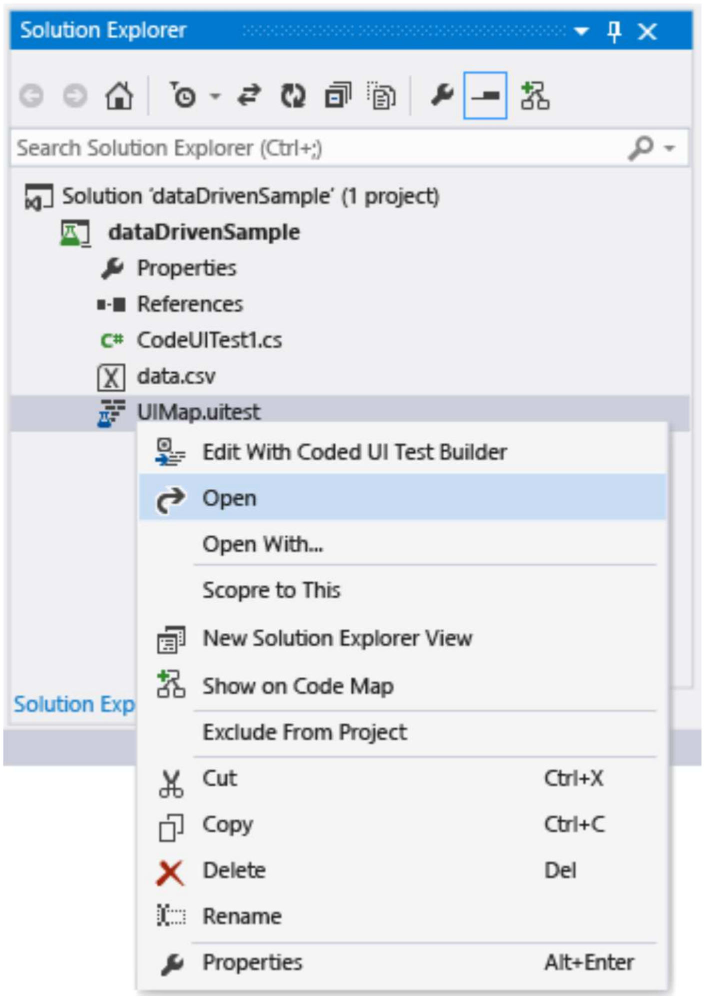
```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using System.Windows.Input;
using System.Windows.Forms;
using System.Drawing;
using Microsoft.VisualStudio.TestTools.UITesting;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.VisualStudio.TestTools.UITest.Extension;
using Keyboard = Microsoft.VisualStudio.TestTools.UITesting.Keyboard;
using Microsoft.VisualStudio.TestTools.UITesting.WinControls;
```

2. Add `TestContext.DataRow[]` in the `CodedUITestMethod1()` method which will apply values from the data source. The data source values override the constants assigned to UIMap controls by using the controls `SearchProperties`:

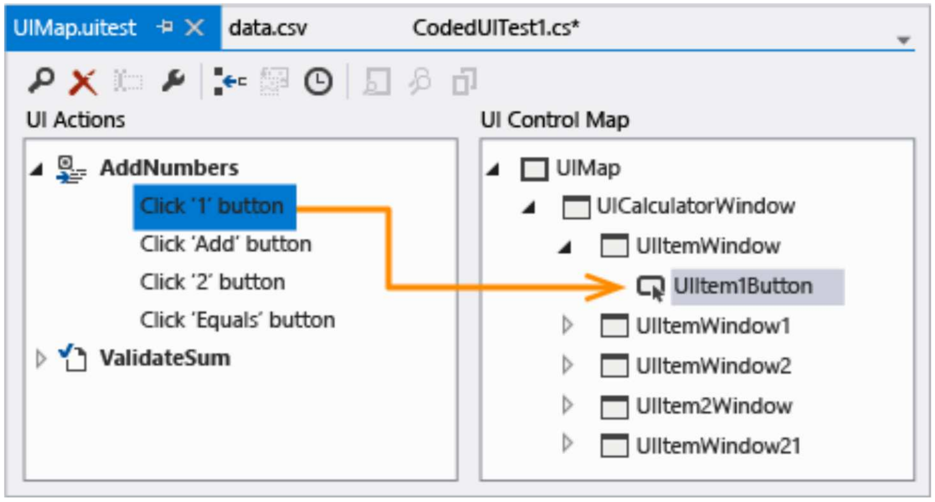
```
public void CodedUITestMethod1()
{

    // To generate code for this test, select "Generate Code for Coded UI Test"
from the shortcut menu and select one of the menu items.

this.UIMap.UICalculatorWindow.UIItemWindow.UIItem1Button.SearchProperties[WinButton.
= TestContext.DataRow["Num1"].ToString();

this.UIMap.UICalculatorWindow.UIItemWindow21.UIItem2Button.SearchProperties[WinButto
= TestContext.DataRow["Num2"].ToString();
    this.UIMap.AddNumbers();
    this.UIMap.ValidateSumExpectedValues.UIItem2TextDisplayText =
TestContext.DataRow["Sum"].ToString();
    this.UIMap.ValidateSum();

}
```

To figure out which search properties to code the data to, use the Coded UI Test Editor.

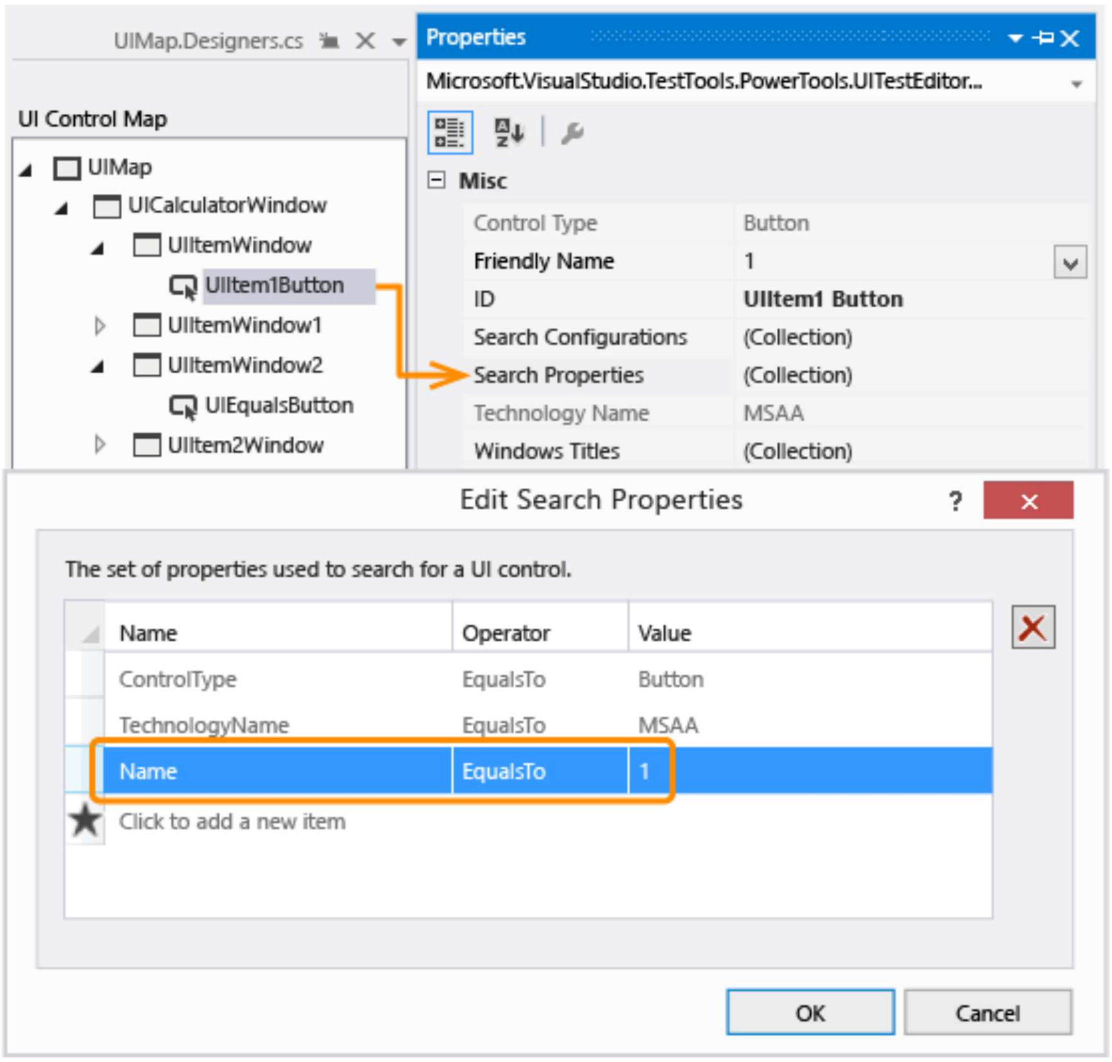
- Open the UIMap.uitest file.



- Choose the UI action and observe the corresponding UI control mapping. Notice how the mapping corresponds to the code, for example, `this.UIMap.UICalculatorWindow.UIItemWindow.UIItem1Button`.



- In the Properties Window, open **Search Properties**. The search properties **Name** value is what is being manipulated in the code using the data source. For example, the `SearchProperties` is being assigned the values in the first column of each data row: `UIItem1Button.SearchProperties[WinButton.PropertyNames.Name] = TestContext.DataRow["Num1"].ToString();`. For the three iterations, this test will change the **Name** value for the search property to 3, then 5, and finally 6.



Property	Value
Control Type	Button
Friendly Name	1
ID	UIItem1 Button
Search Configurations	(Collection)
Search Properties	(Collection)
Technology Name	MSAA
Windows Titles	(Collection)

The set of properties used to search for a UI control.

Name	Operator	Value
ControlType	EqualsTo	Button
TechnologyName	EqualsTo	MSAA
Name	EqualsTo	1
Click to add a new item		

3. Save the solution.

Step 5 – Run the data-driven test

1. Verify that the test is now data-driven by running the test again.

   You should see the test run through the three iterations using the values in the .csv file. The validation should work as well and the test should display as passed in the Test Explorer.

Guidance

For additional information, see Testing for Continuous Delivery with Visual Studio 2012 – Chapter 2: Unit Testing: Testing the Inside and Testing for Continuous Delivery with Visual Studio 2012 – Chapter 5: Automating System Tests

Q & A

What are the data source attributes for other data source types, such as SQL Express or XML?

You can use the sample data source strings in the table below by copying them to your code and making the necessary customizations.

Date Source Type	**Data Source Attribute**
CSV	`[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV", "|DataDirectory|\\data.csv", "data#csv", DataAccessMethod.Sequential), DeploymentItem("data.csv"), TestMethod]`
Excel	`DataSource("System.Data.Odbc", "Dsn=ExcelFiles;Driver={Microsoft Excel Driver (*.xls)};dbq=|DataDirectory|\\Data.xls;defaultdir=.; driverid=790;maxbuffersize=2048;pagetimeout=5;readonly=true", "Sheet1$", DataAccessMethod.Sequential), DeploymentItem("Sheet1.xls"), TestMethod]`
Test case in Team Foundation Server	`[DataSource("Microsoft.VisualStudio.TestTools.DataSource.TestCase", "http://vlm13261329:8080/tfs/DefaultCollection;Agile", "30", DataAccessMethod.Sequential), TestMethod]`
XML	`[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML", "|DataDirectory|\\data.xml", "Iterations", DataAccessMethod.Sequential), DeploymentItem("data.xml"), TestMethod]`
SQL Express	`[DataSource("System.Data.SqlClient", "Data Source=.\\sqlexpress; Initial Catalog=tempdb;Integrated Security=True", "Data", DataAccessMethod.Sequential), TestMethod]`

Q: Can I use data-driven tests on my Windows Phone app?

A: Yes. Data-driven Coded UI tests for Windows Phone are defined using the DataRow attribute on a test method. In the following example, x and y use the values of 1 and 2 for the first iteration and -1 and -2 for the second iteration of the test.

```
[DataRow(1, 2, DisplayName = "Add positive numbers")]
[DataRow(-1, -2, DisplayName = "Add negative numbers")]
[TestMethod]
public void DataDrivingDemo_MyTestMethod(int x, int y)
```

For more information, see Use Data-driven coded UI tests on Windows Phone apps.

Q: Why can’t I modify the code in the UIMap.Designer file?

A: Any code changes you make in the UIMapDesigner.cs file will be overwritten every time you generate code using the UIMap - Coded UI Test Builder. In this sample, and in most cases, the code changes needed to enable a test to use a data source can be made to the test's source code file (that is, CodedUITest1.cs).

If you have to modify a recorded method, you must copy it to UIMap.cs file and rename it. The UIMap.cs file can be used to override methods and properties in the UIMapDesigner.cs file. You must remove the reference to the original method in the Coded UITest.cs file and replace it with the renamed method name.

See Also

T:Microsoft.VisualStudio.TestTools.UITest.Common.UIMap.UIMap
Assert
Use UI Automation To Test Your Code
Creating Coded UI Tests
Best Practices for Coded UI Tests
Supported Configurations and Platforms for Coded UI Tests and Action Recordings

Table of contents

Testing scenarios
▸ Manual and exploratory testing
▸ Performance testing
▸ Continuous testing
▾ Developer testing tools
  Overview
  Get started
  Create Unit Tests command
  Generate tests with IntelliTest
  Run tests with Test Explorer
  Determine code coverage
  System test with Visual Studio
  Create system tests with VS
  Run system tests with VS
  Submit bugs in VS
  Submit bugs in TFS
  ▸ IntelliTest reference manual
▸ Test lab management
API reference for test tools

Get started with developer testing tools

Last Updated: 8/4/2016

IN THIS ARTICLE +

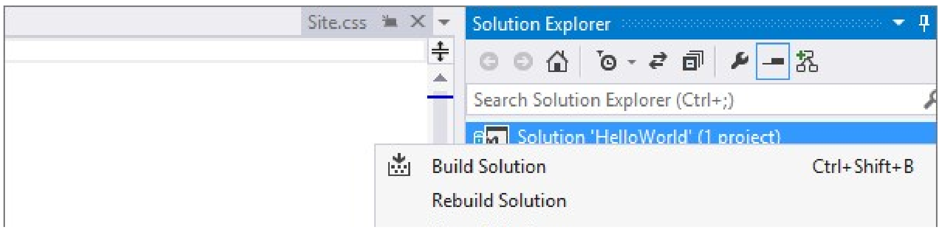
Visual Studio 2015 | Previous version

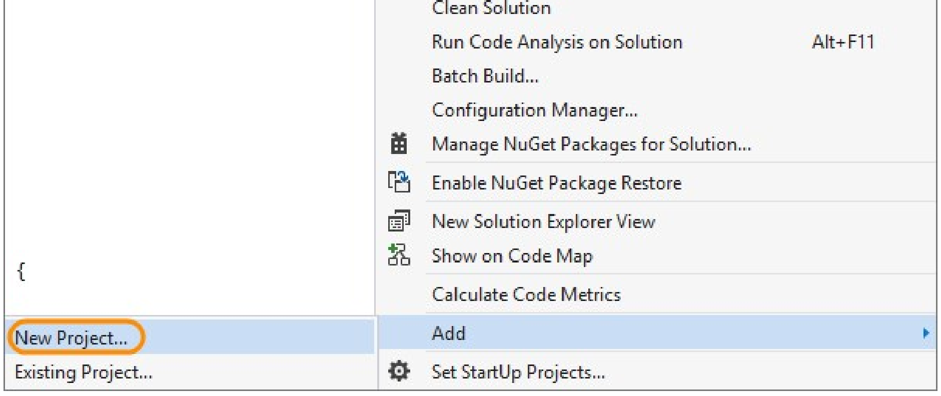
Use Visual Studio to define and run your unit tests to maintain code health, ensure code coverage, and to find errors and faults before your customers do.

Create unit tests

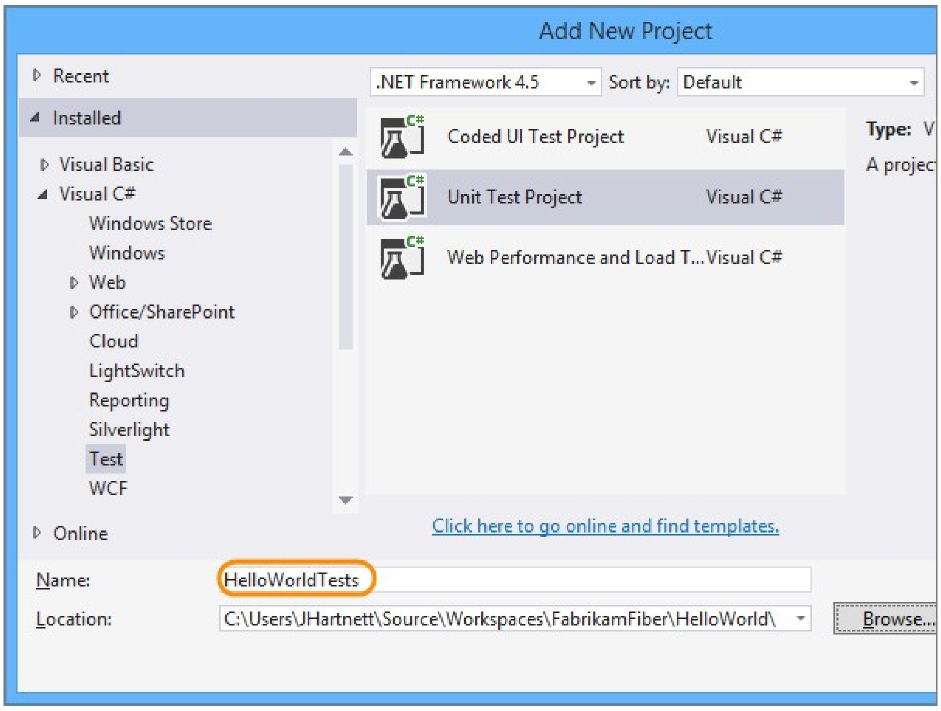
Create unit tests and run them frequently to make sure your code is working properly.

1. Create a unit test project.





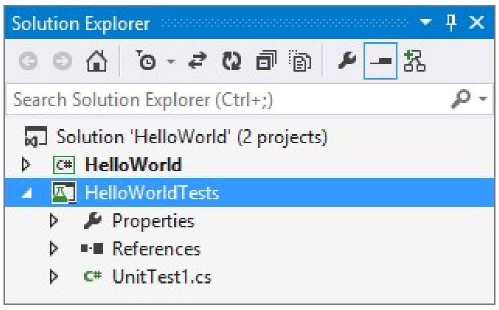
2. Name your project.



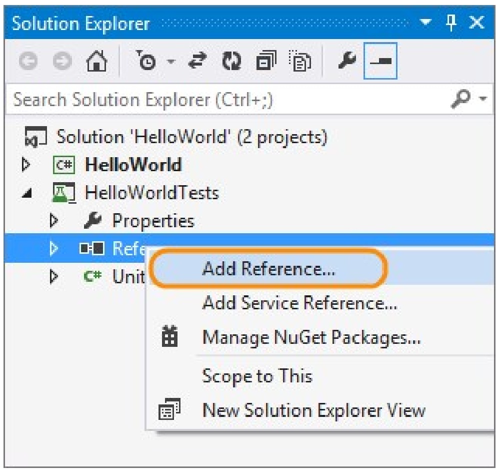
Name: HelloWorldTests

Location: C:\Users\JHartnett\Source\Workspaces\FabrikamFiber\HelloWorld\

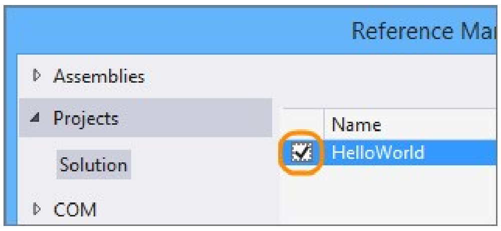
The project is added to your solution.



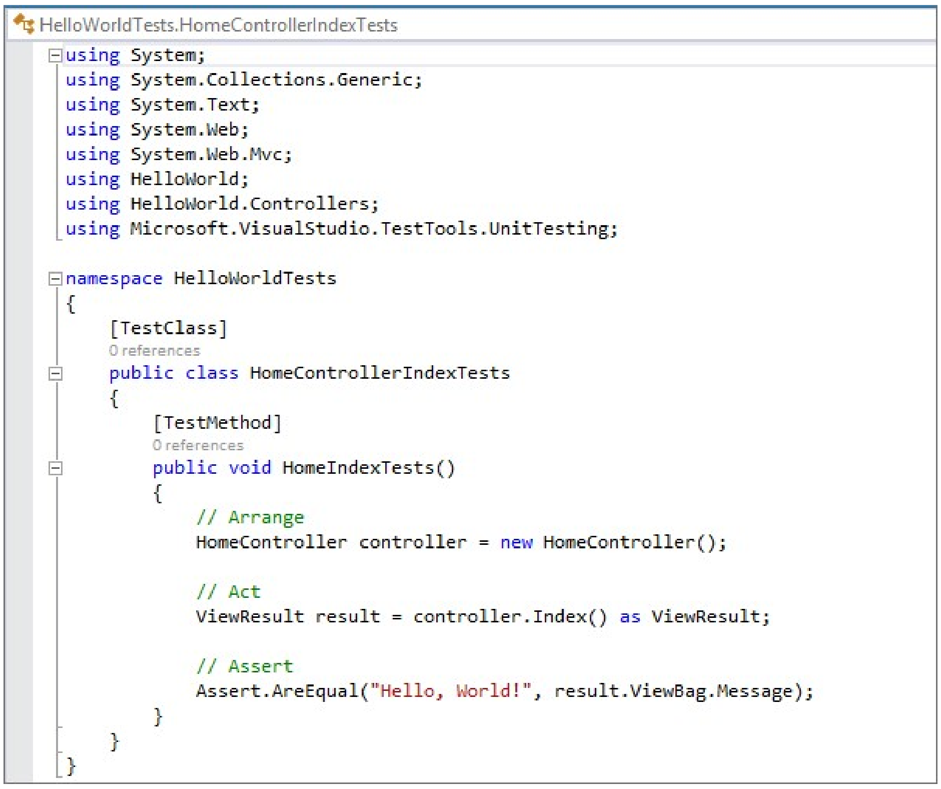
3. In the unit test project, add a reference to the project you want to test.



4. Select the project that contains the code you'll test.

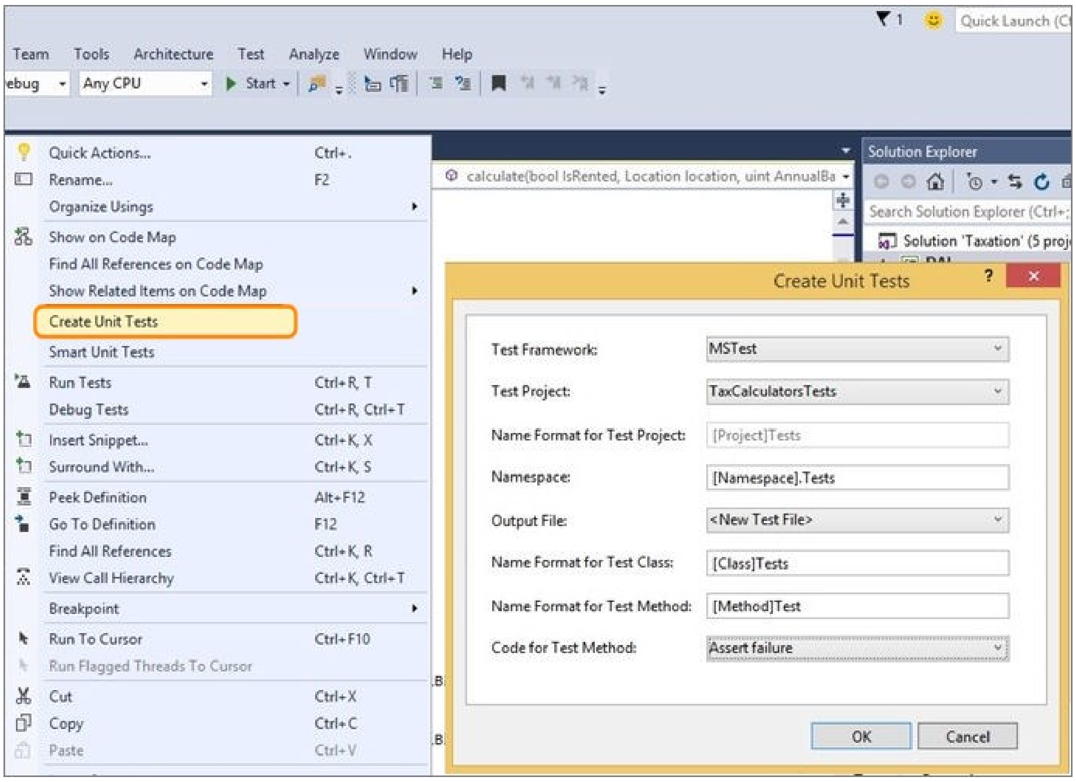


5. Code your unit test.



```
HelloWorldTests.HomeControllerIndexTests
using System;
using System.Collections.Generic;
using System.Text;
using System.Web;
using System.Web.Mvc;
using HelloWorld;
using HelloWorld.Controllers;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace HelloWorldTests
{
    [TestClass]
    0 references
    public class HomeControllerIndexTests
    {
        [TestMethod]
        0 references
        public void HomeIndexTests()
        {
            // Arrange
            HomeController controller = new HomeController();

            // Act
            ViewResult result = controller.Index() as ViewResult;

            // Assert
            Assert.AreEqual("Hello, World!", result.ViewBag.Message);
        }
    }
}
```

You can also create unit test method stubs with the **Create Unit Tests** command. To learn how, see Create unit test method stubs with the Create Unit Tests command.


Quick Launch (C
Team Tools Architecture Test Analyze Window Help
ebug Any CPU Start
Quick Actions... Ctrl+.
Rename... F2
Organize Usings
Show on Code Map
Find All References on Code Map
Show Related Items on Code Map
Create Unit Tests
Smart Unit Tests
Run Tests Ctrl+R, T
Debug Tests Ctrl+R, Ctrl+T
Insert Snippet... Ctrl+K, X
Surround With... Ctrl+K, S
Peek Definition Alt+F12
Go To Definition F12
Find All References Ctrl+K, R
View Call Hierarchy Ctrl+K, Ctrl+T
Breakpoint
Run To Cursor Ctrl+F10
Run Flagged Threads To Cursor
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
calculate(bool IsRented, Location location, uint AnnualBa
Solution Explorer
Search Solution Explorer (Ctrl+;
Solution 'Taxation' (5 proj

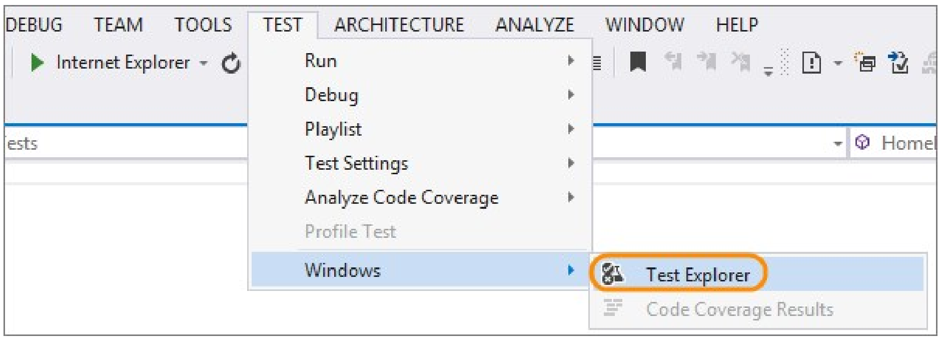
Create Unit Tests

Test Framework:	MSTest
Test Project:	TaxCalculatorsTests
Name Format for Test Project:	[Project]Tests
Namespace:	[Namespace].Tests
Output File:	<New Test File>
Name Format for Test Class:	[Class]Tests
Name Format for Test Method:	[Method]Test
Code for Test Method:	Assert failure

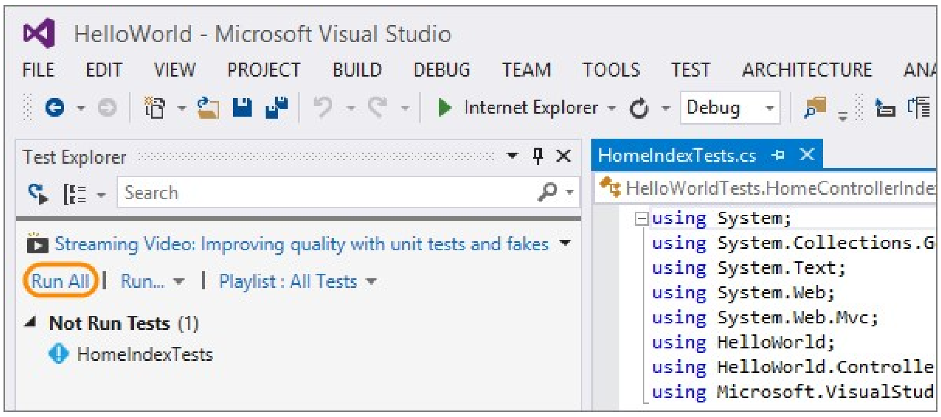
OK Cancel

Run unit tests

1. Open Test Explorer.

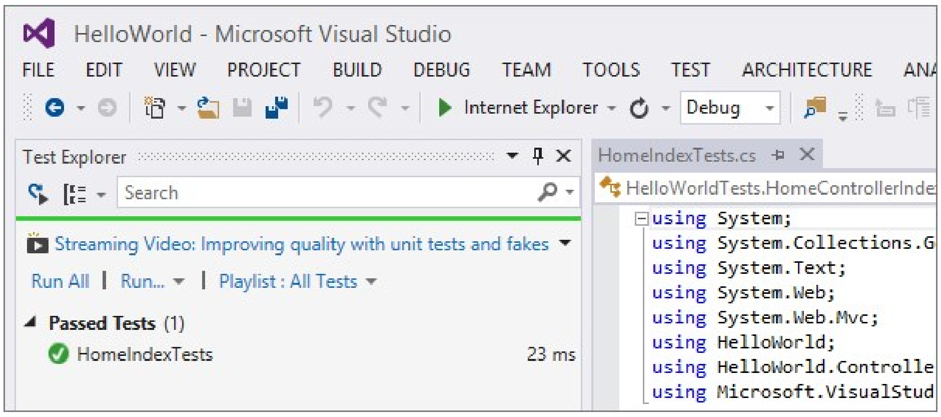


2. Run unit tests.


HelloWorld - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE
Internet Explorer
Debug
Test Explorer
Search
Streaming Video: Improving quality with unit tests and fakes
Run All | Run... | Playlist : All Tests
Not Run Tests (1)
HomeIndexTests
HomeIndexTests.cs
HelloWorldTests.HomeControllerInde

```
using System;
using System.Collections.G
using System.Text;
using System.Web;
using System.Web.Mvc;
using HelloWorld;
using HelloWorld.Controlle
using Microsoft.VisualStud
```

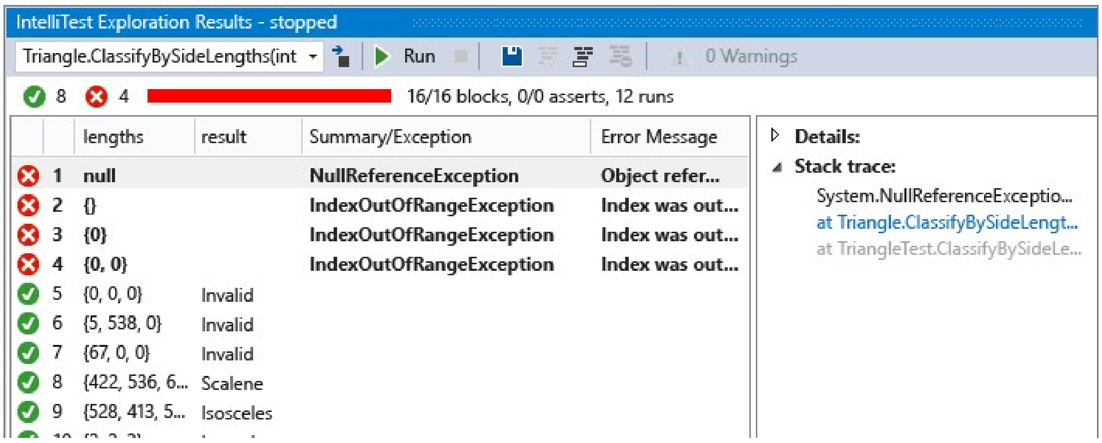
You can see the unit tests that passed or failed in Test Explorer.


HelloWorld - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE
Internet Explorer
Debug
Test Explorer
Search
Streaming Video: Improving quality with unit tests and fakes
Run All | Run... | Playlist : All Tests
Passed Tests (1)
HomeIndexTests 23 ms
HomeIndexTests.cs
HelloWorldTests.HomeControllerInde

```
using System;
using System.Collections.G
using System.Text;
using System.Web;
using System.Web.Mvc;
using HelloWorld;
using HelloWorld.Controlle
using Microsoft.VisualStud
```

Generate unit tests with IntelliTest

When you run IntelliTest, you can easily see which tests are failing and add any necessary code to fix them. You can select which of the generated tests to save into a test project to provide a regression suite. As you change your code, rerun IntelliTest to keep the generated tests in sync with your code changes. To learn how, see Generating unit tests for your code with IntelliTest.


IntelliTest Exploration Results - stopped
Triangle.ClassifyBySideLengths(int
Run
0 Warnings
✓ 8 ✗ 4
16/16 blocks, 0/0 asserts, 12 runs

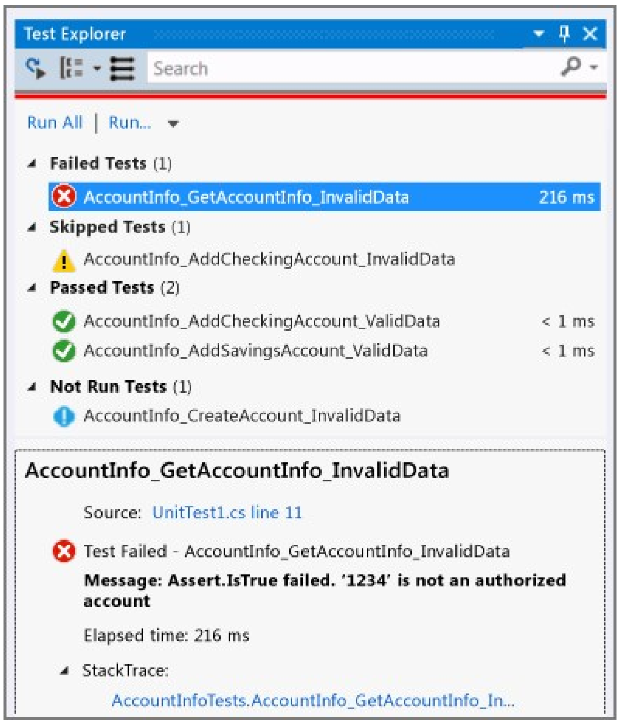
		lengths	result	Summary/Exception	Error Message
✗	1	null		NullReferenceException	Object refer...
✗	2	{}		IndexOutOfRangeException	Index was out...
✗	3	{0}		IndexOutOfRangeException	Index was out...
✗	4	{0, 0}		IndexOutOfRangeException	Index was out...
✓	5	{0, 0, 0}	Invalid		
✓	6	{5, 538, 0}	Invalid		
✓	7	{67, 0, 0}	Invalid		
✓	8	{422, 536, 6...	Scalene		
✓	9	{528, 413, 5...	Isosceles		

Details:
Stack trace:
System.NullReferenceExceptio...
at Triangle.ClassifyBySideLengt...
at TriangleTest.ClassifyBySideLe...

10 {2, 2, 3} Isosceles
11 {1, 512, 512} Isosceles
12 {512, 512, 5... Equilateral

Run unit tests with Test Explorer

Use Test Explorer to run unit tests from Visual Studio or third-party unit test projects, group tests into categories, filter the test list, and create, save, and run playlists of tests. You can also debug tests and analyze test performance and code coverage. To learn how, see Run unit tests with Test Explorer.

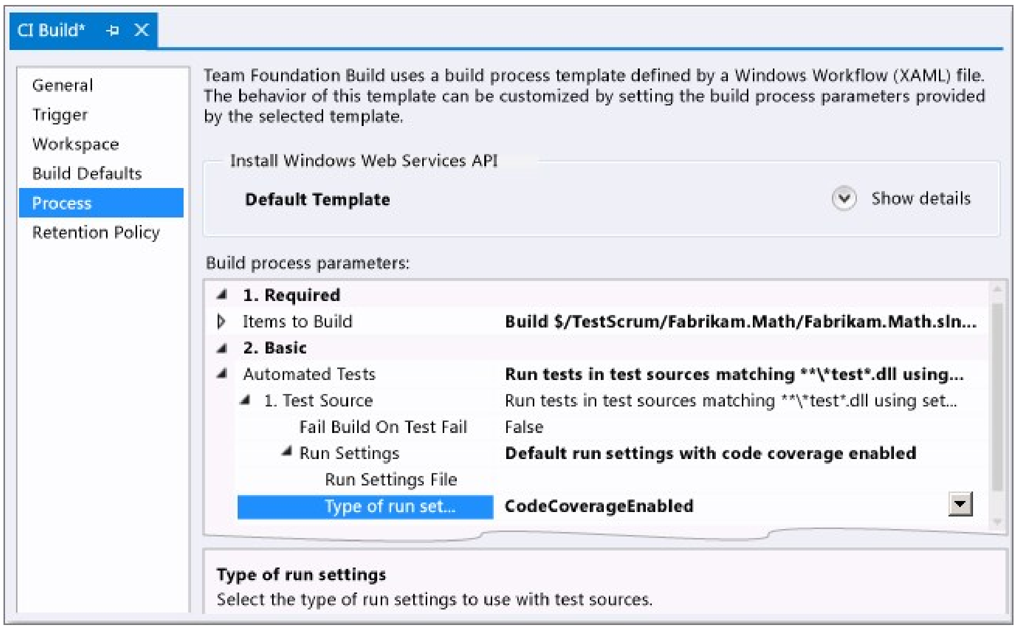


Failed Tests (1)
AccountInfo_GetAccountInfo_InvalidData 216 ms
Skipped Tests (1)
AccountInfo_AddCheckingAccount_InvalidData
Passed Tests (2)
AccountInfo_AddCheckingAccount_ValidData < 1 ms
AccountInfo_AddSavingsAccount_ValidData < 1 ms
Not Run Tests (1)
AccountInfo_CreateAccount_InvalidData

AccountInfo_GetAccountInfo_InvalidData
Source: UnitTest1.cs line 11
Test Failed - AccountInfo_GetAccountInfo_InvalidData
Message: Assert.IsTrue failed. '1234' is not an authorized account
Elapsed time: 216 ms
StackTrace:
AccountInfoTests.AccountInfo_GetAccountInfo_In...

Use code coverage to determine how much code is being tested

To determine what proportion of your project's code is actually being tested by coded tests such as unit tests, you can use the code coverage feature of Visual Studio. To guard effectively against bugs, your tests should exercise or 'cover' a large proportion of your code. To learn how, see Use Code Coverage to Determine How Much Code is being Tested.



Team Foundation Build uses a build process template defined by a Windows Workflow (XAML) file. The behavior of this template can be customized by setting the build process parameters provided by the selected template.

Install Windows Web Services API
Default Template Show details

Build process parameters:

1. Required	
Items to Build	**Build $/TestScrum/Fabrikam.Math/Fabrikam.Math.sln...**
2. Basic	
Automated Tests	**Run tests in test sources matching **\*test*.dll using...**
1. Test Source	Run tests in test sources matching **\*test*.dll using set...
Fail Build On Test Fail	False
Run Settings	**Default run settings with code coverage enabled**
Run Settings File	
Type of run set...	**CodeCoverageEnabled**

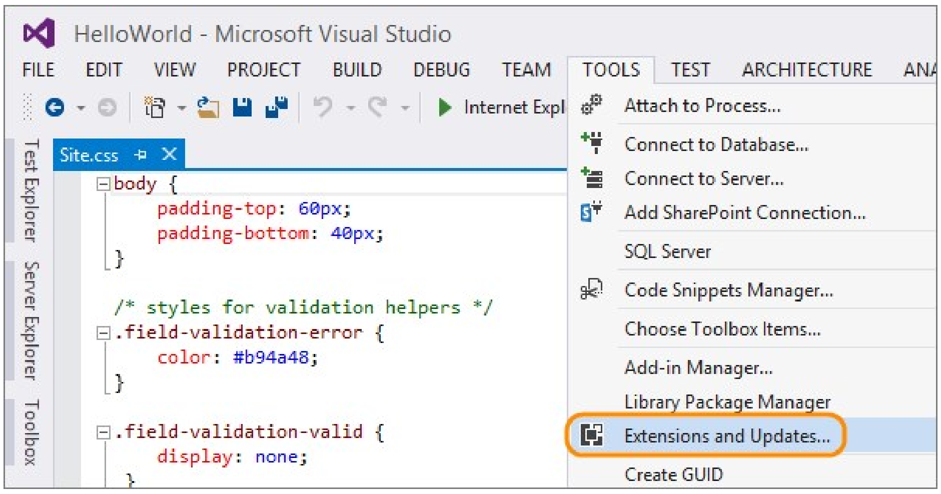
Type of run settings
Select the type of run settings to use with test sources.

Q & A

Q: Can I run unit tests in Visual Studio if I use a different unit test framework?

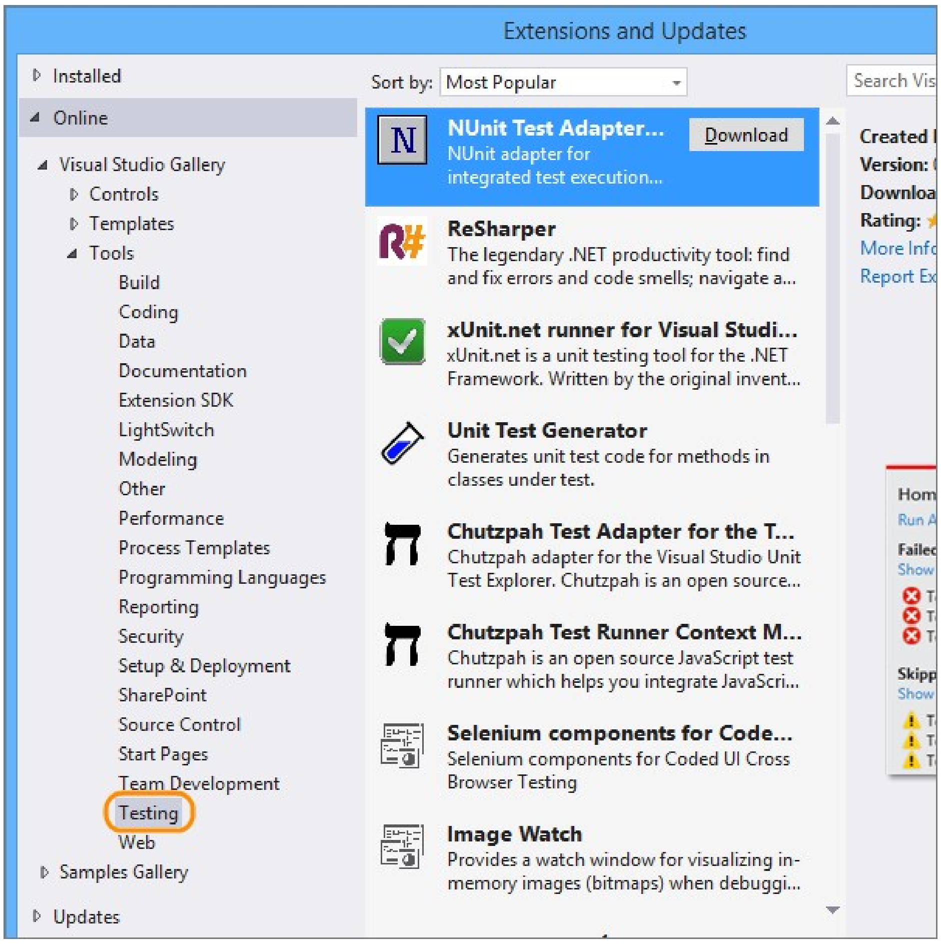
A: Yes, use the plug-in for that framework so that Visual Studio's test runner can work with that framework. Here are the unit testing framework plug-ins for Visual Studio that are available right now.

1. Use Visual Studio's extension manager to download your plug-in.



```
body {
    padding-top: 60px;
    padding-bottom: 40px;
}

/* styles for validation helpers */
.field-validation-error {
    color: #b94a48;
}

.field-validation-valid {
    display: none;
}
```

2. Download your plug-in from the Visual Studio Gallery under Tools/Testing, or search for it if you know the name.



NUnit Test Adapter... NUnit adapter for integrated test execution...

ReSharper The legendary .NET productivity tool: find and fix errors and code smells; navigate a...

xUnit.net runner for Visual Studi... xUnit.net is a unit testing tool for the .NET Framework. Written by the original invent...

Unit Test Generator Generates unit test code for methods in classes under test.

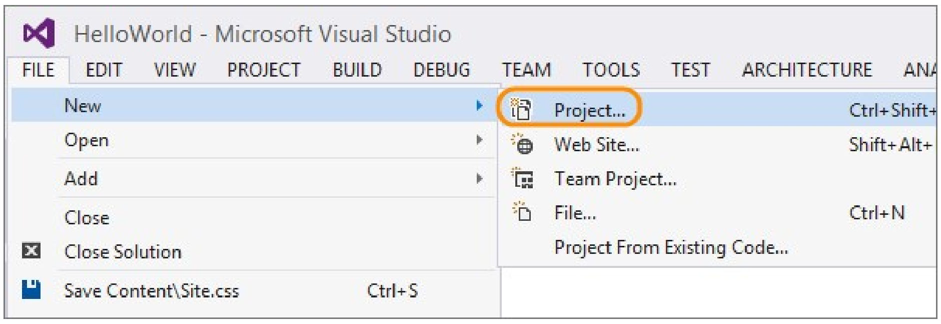
Chutzpah Test Adapter for the T... Chutzpah adapter for the Visual Studio Unit Test Explorer. Chutzpah is an open source...

Chutzpah Test Runner Context M... Chutzpah is an open source JavaScript test runner which helps you integrate JavaScri...

Selenium components for Code... Selenium components for Coded UI Cross Browser Testing

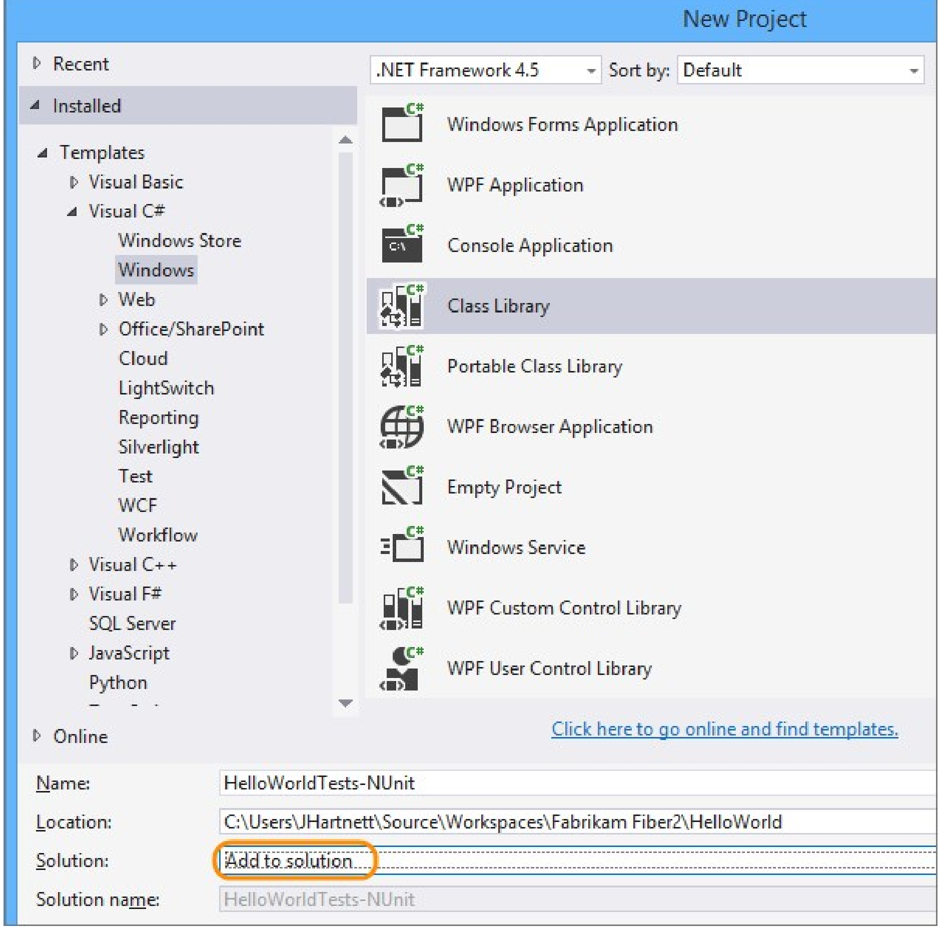
Image Watch Provides a watch window for visualizing in-memory images (bitmaps) when debuggi...

3. Create a class library project.



HelloWorld - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANA
New
Open
Add
Close
Close Solution
Save Content\Site.css Ctrl+S
Project... Ctrl+Shift+
Web Site... Shift+Alt+
Team Project...
File... Ctrl+N
Project From Existing Code...

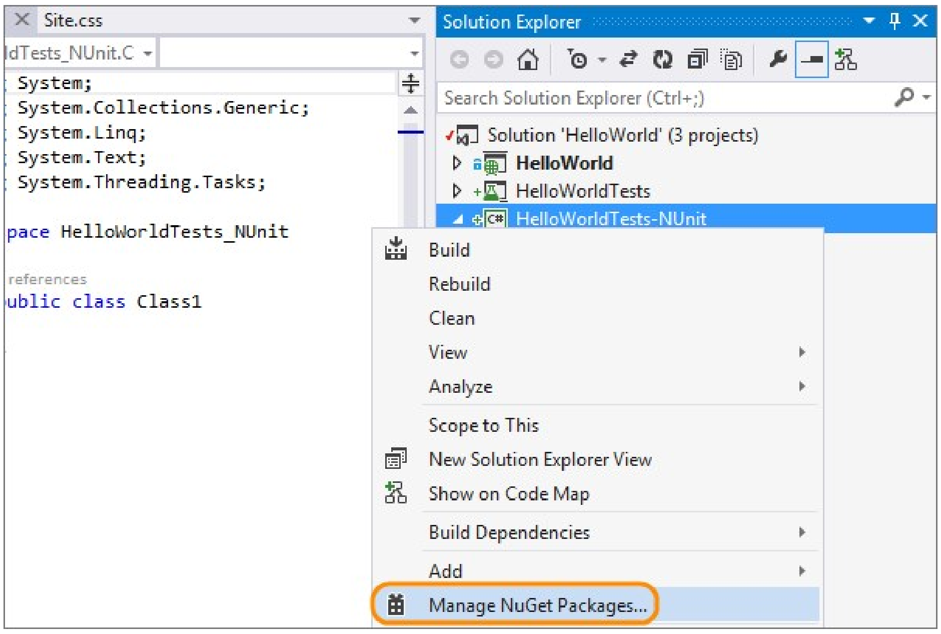
Add the project to your solution.



.NET Framework 4.5 Sort by: Default

Name: HelloWorldTests-NUnit
Location: C:\Users\JHartnett\Source\Workspaces\Fabrikam Fiber2\HelloWorld
Solution: Add to solution
Solution name: HelloWorldTests-NUnit

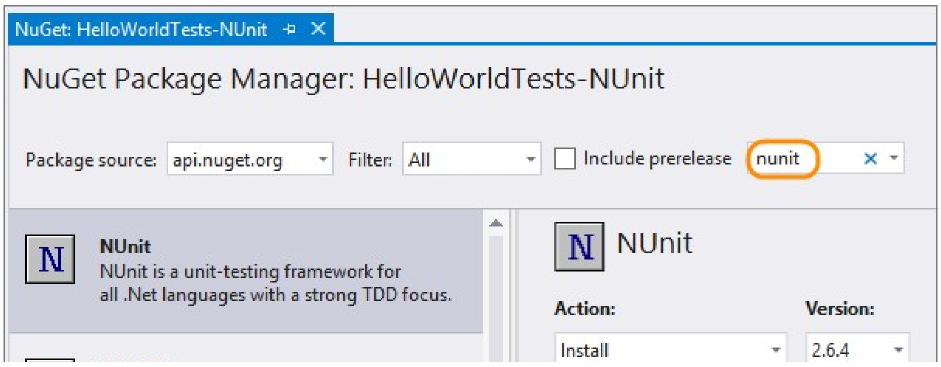
4. In the class library project, run NuGet to install the plug-in.



```
ldTests_NUnit.C
 System;
 System.Collections.Generic;
 System.Linq;
 System.Text;
 System.Threading.Tasks;

pace HelloWorldTests_NUnit

references
ublic class Class1
```

NuGet is an extension of Visual Studio that you can use to add and update libraries and tools for your projects.

5. Install your plug-in. If you know the name, you can search for it online.

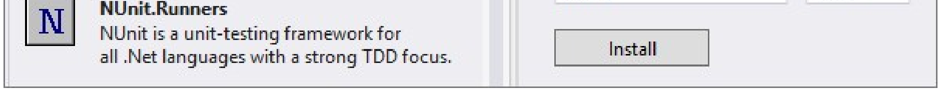


NuGet Package Manager: HelloWorldTests-NUnit

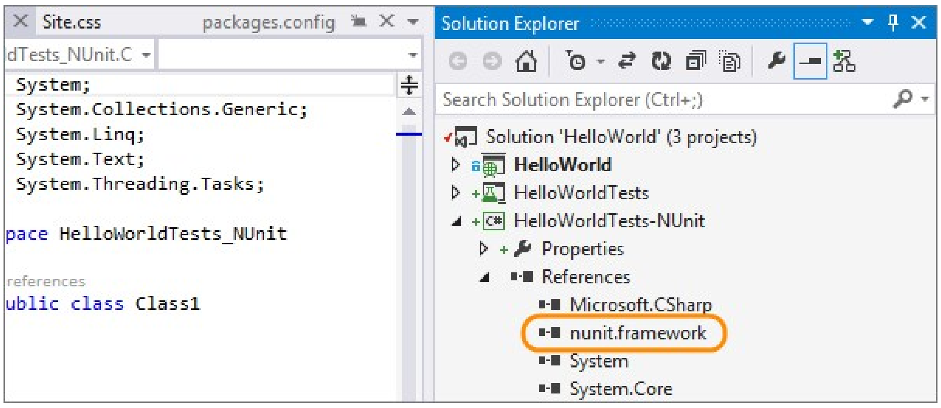
Package source: api.nuget.org Filter: All nunit

NUnit
NUnit is a unit-testing framework for all .Net languages with a strong TDD focus.

NUnit
Action: Install Version: 2.6.4

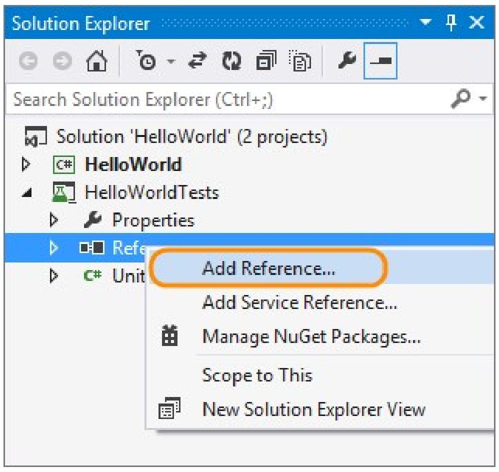

NUnit.Runners
NUnit is a unit-testing framework for all .Net languages with a strong TDD focus.
Install

The framework is referenced in your project.

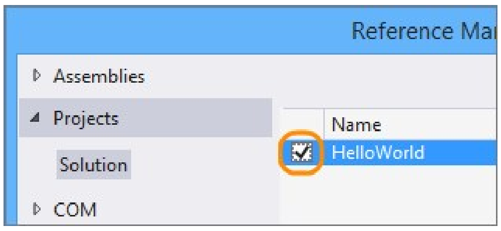


```
 System;
 System.Collections.Generic;
 System.Linq;
 System.Text;
 System.Threading.Tasks;

pace HelloWorldTests_NUnit

references
ublic class Class1
```

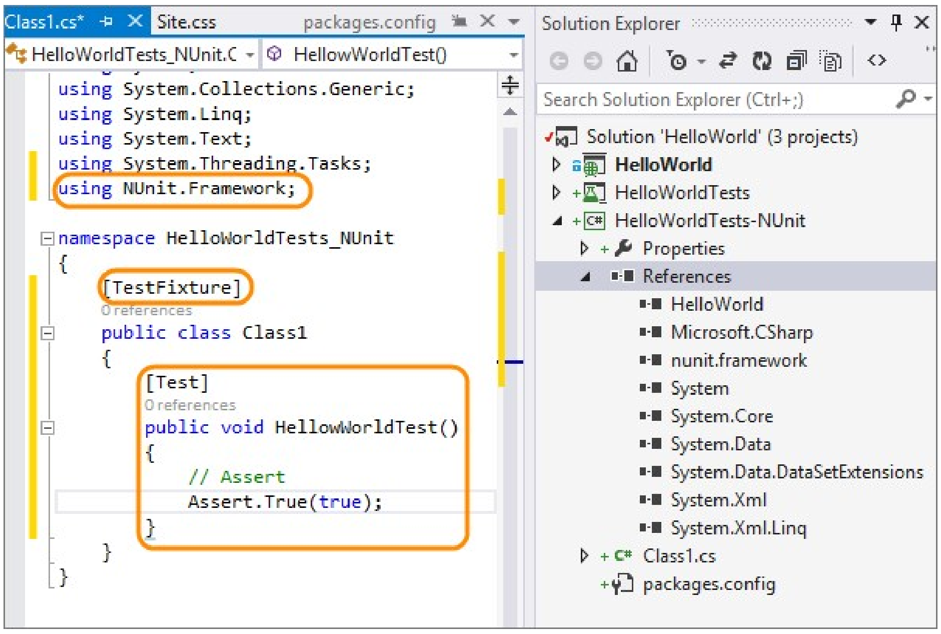
6. In the class library project, add a reference to the project you want to test.



7. Select the project that contains the code you'll test.



8. Code your unit test.



```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using NUnit.Framework;

namespace HelloWorldTests_NUnit
{
    [TestFixture]
    0 references
    public class Class1
    {
        [Test]
        0 references
        public void HelloWorldTest()
        {
            // Assert
            Assert.True(true);
        }
    }
}
```

See also

- Create Unit Tests command
- Generate tests with IntelliTest
- Run tests with Test Explorer
- Determine code coverage
- System test with Visual Studio
- Create system tests with VS
- Run system tests with VS

Help and support

Submit bugs through Connect, make suggestions on Uservoice, and send quick thoughts using the Send-a-Smile link in the Visual Studio, Team Services, or TFS title bar. We look forward to your feedback.

Table of contents

Testing scenarios

- Manual and exploratory testing
  - Overview
  - Get started with manual testing
  - Advanced manual testing techniques
    - User acceptance testing
    - Share steps between test cases
    - Repeat a test with different data
    - Manage test results
  - Manual testing with Microsoft Test Manager
  - Get started with exploratory testing
  - Advanced exploratory testing techniques
  - Exploratory testing with Microsoft Test Manager
- Performance testing
- Continuous testing
- Developer testing tools
- Test lab management

API reference for test tools

User acceptance testing

Last Updated: 8/30/2016

IN THIS ARTICLE +

Team Services | TFS 2015

Today's faster development pace requires tools that enable test teams to more easily verify value based on business requirements, and the high quality software demanded by customers. This type of testing is often referred to as *user acceptance testing* and is available as a feature in Visual Studio Team Services and Team Foundation Server.

Typically you create a Test Suite using a formal requirement work item type. However, today’s agile teams often prefer to work from User Stories or Product Backlog items as their requirements.

Before you start

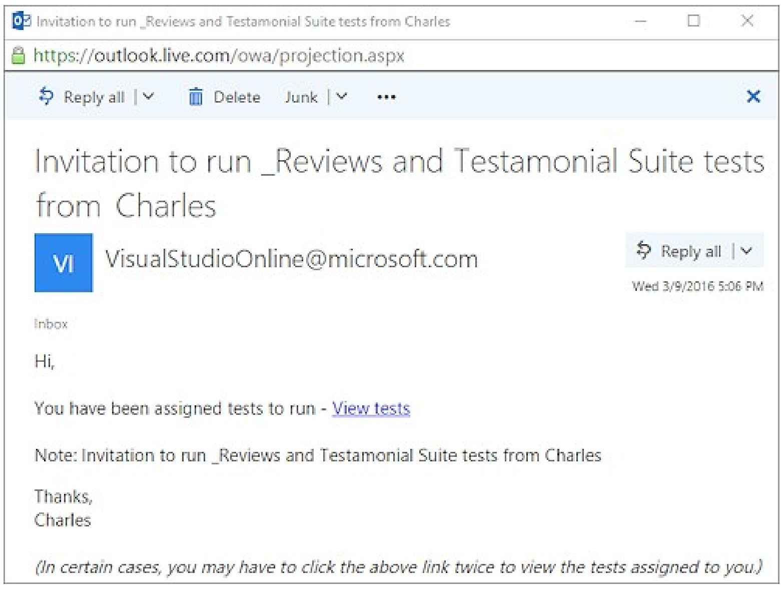
You must have already created work items and a test plan. If not, follow the steps in:

- Create your backlog
- Create a test plan

Assign and invite testers

Visual Studio Team Services makes it easy to assign testers to individual test cases. A typical scenario for user acceptance testing is the ability to not just assign one tester to a test case (see Search for and assign testers) but assign multiple testers an entire set of tests.

This can also be accomplished by selecting the suite and choosing **Assign testers to run all tests**. This option also enables the sending of emails to the testers assigned to the tests.


Invitation to run _Reviews and Testamonial Suite tests from Charles
https://outlook.live.com/owa/projection.aspx
Reply all
Delete
Junk
VI
Reply all
Inbox

Invitation to run _Reviews and Testamonial Suite tests from Charles

VisualStudioOnline@microsoft.com

Wed 3/9/2016 5:06 PM

Hi,

You have been assigned tests to run - View tests

Note: Invitation to run _Reviews and Testamonial Suite tests from Charles

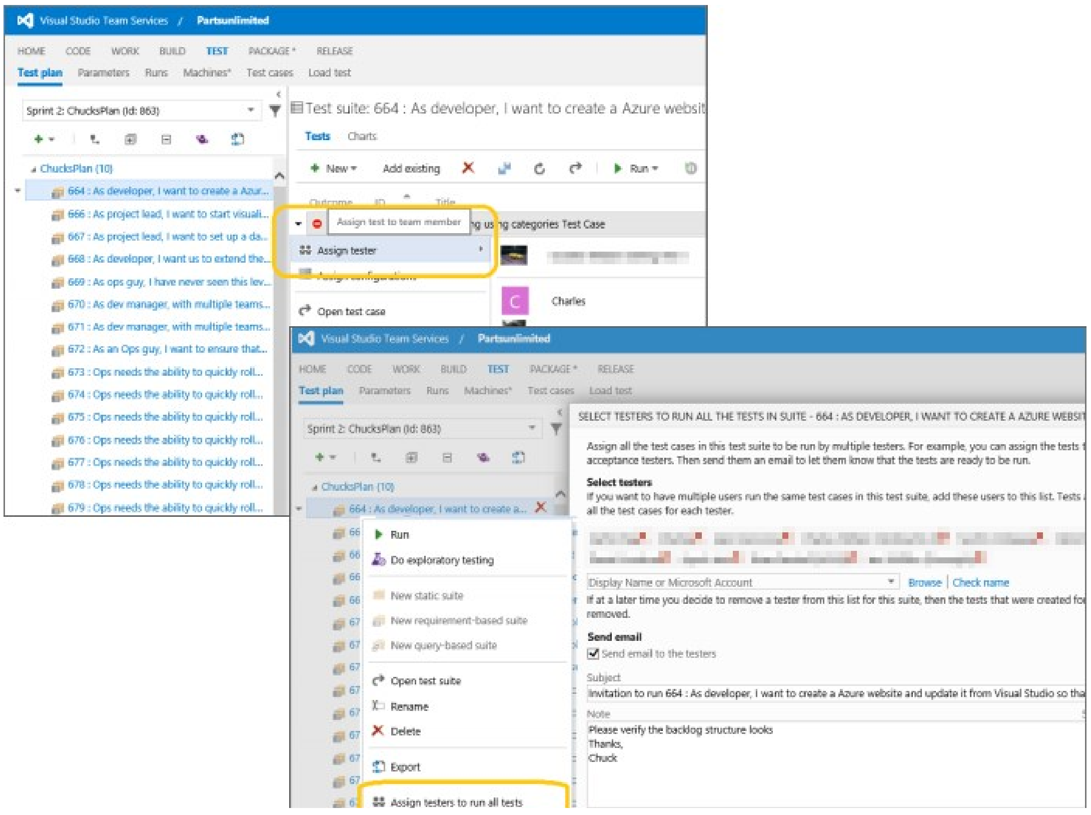
Thanks,
Charles

(In certain cases, you may have to click the above link twice to view the tests assigned to you.)

An important feature of user acceptance testing is that the testers may in fact be the business owners who created the original business requirements.

Search for and assign testers

In scenarios where you have large development teams the ability search for an individual is also important. Choose **Assign tester** from the drop-down menu. In the shortcut menu, choose **Assign testers to run all tests** and select the testers you want to include.


Visual Studio Team Services / Partsunlimited
HOME CODE WORK BUILD TEST PACKAGE* RELEASE
Test plan Parameters Runs Machines* Test cases Load test
Sprint 2: ChucksPlan (Id: 863)
ChucksPlan (10)
664 : As developer, I want to create a Azur...
666 : As project lead, I want to start visuali...
667 : As project lead, I want to set up a da...
668 : As developer, I want us to extend the...
669 : As ops guy, I have never seen this lev...
670 : As dev manager, with multiple teams...
671 : As dev manager, with multiple teams...
672 : As an Ops guy, I want to ensure that...
673 : Ops needs the ability to quickly roll...
674 : Ops needs the ability to quickly roll...
675 : Ops needs the ability to quickly roll...
676 : Ops needs the ability to quickly roll...
677 : Ops needs the ability to quickly roll...
678 : Ops needs the ability to quickly roll...
679 : Ops needs the ability to quickly roll...
Test suite: 664 : As developer, I want to create a Azure websit
Tests Charts
New Add existing Run
Outcome ID Title
Assign test to team member
Assign tester
Assign configuration...
Open test case
categories Test Case
C Charles
Run
Do exploratory testing
New static suite
New requirement-based suite
New query-based suite
Open test suite
Rename
Delete
Export
Assign testers to run all tests

SELECT TESTERS TO RUN ALL THE TESTS IN SUITE - 664 : AS DEVELOPER, I WANT TO CREATE A AZURE WEBSIT

Assign all the test cases in this test suite to be run by multiple testers. For example, you can assign the tests t
acceptance testers. Then send them an email to let them know that the tests are ready to be run.

Select testers

If you want to have multiple users run the same test cases in this test suite, add these users to this list. Tests
all the test cases for each tester.

[redacted]

Display Name or Microsoft Account | Browse | Check name

If at a later time you decide to remove a tester from this list for this suite, then the tests that were created fo
removed.

Send email

[✓] Send email to the testers

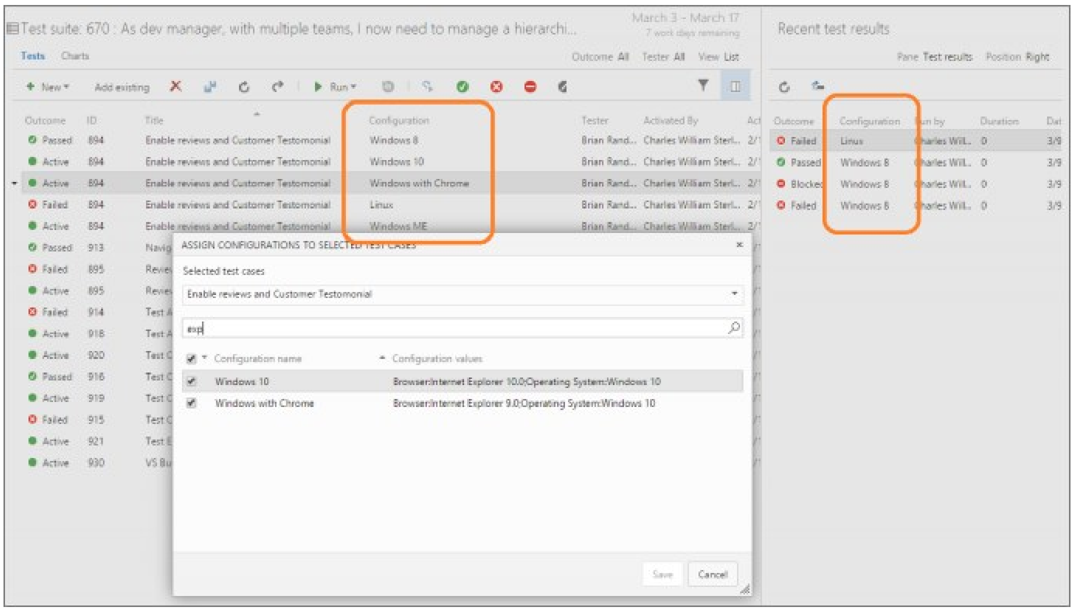
Subject: Invitation to run 664 : As developer, I want to create a Azure website and update it from Visual Studio so tha

Note:
Please verify the backlog structure looks
Thanks,
Chuck

Set the **Send email** option to send all of them a notification email.

Assign configurations to test suites and test cases

You will often want to verify requirements by configuration. Do this by opening the shortcut menu for a test case and choosing **Assign Configurations**.



Outcome	ID	Title	Configuration	Tester	Activated By
Passed	894	Enable reviews and Customer Testomonial	Windows 8	Brian Rand...	Charles William Sterl...
Active	894	Enable reviews and Customer Testomonial	Windows 10	Brian Rand...	Charles William Sterl...
Active	894	Enable reviews and Customer Testomonial	Windows with Chrome	Brian Rand...	Charles William Sterl...
Failed	894	Enable reviews and Customer Testomonial	Linux	Brian Rand...	Charles William Sterl...
Active	894	Enable reviews and Customer Testomonial	Windows ME	Brian Rand...	Charles William Sterl...

Outcome	Configuration	Run by	Duration
Failed	Linux	Charles Will...	0
Passed	Windows 8	Charles Will...	0
Blocked	Windows 8	Charles Will...	0
Failed	Windows 8	Charles Will...	0

ASSIGN CONFIGURATIONS TO SELECTED TEST CASES

Selected test cases: Enable reviews and Customer Testomonial

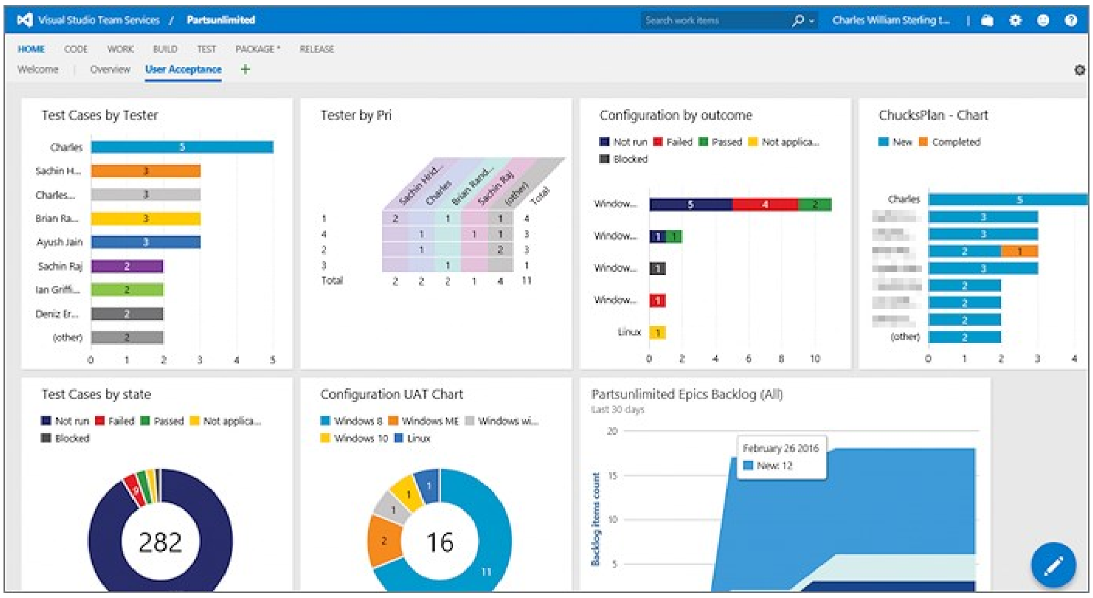
Configuration name	Configuration values
✓ Windows 10	Browser:Internet Explorer 10.0;Operating System:Windows 10
✓ Windows with Chrome	Browser:Internet Explorer 9.0;Operating System:Windows 10

Save Cancel

The test case will then be listed for each configuration. The test results will indicate which configuration was run.

Easily track results

A key principle of good user acceptance testing practice is to minimize the effort required to determine whether a requirement has been achieved. There are two ways this can be achieved, you can focus on individual test runs and tests in the **Test** hub to see which failed or use the charts views make it much easy and accessible to all members of Visual Studio Team Services makes this much easier.



Note: The dashboard display show here is also used for other types of testing such as continuous testing.

If you don't see the data or information you expect in the dashboard charts, verify that the columns in your data have been added to the Tests view. For details see this blog post.

Help and support

Submit bugs through Connect, make suggestions on Uservoice, and send quick thoughts using the Send-a-Smile link in the Visual Studio, Team Services, or TFS title bar. We look forward to your feedback.

Table of contents

Testing scenarios

Manual and exploratory testing

Overview

Get started with manual testing

Add, run, and track tests from the Kanban board

Create test plans

Create test cases

Run manual tests

Track test status

Advanced manual testing techniques

Manual testing with Microsoft Test Manager

Get started with exploratory testing

Advanced exploratory testing techniques

Exploratory testing with Microsoft Test Manager

Performance testing

Continuous testing

Developer testing tools

Test lab management

API reference for test tools

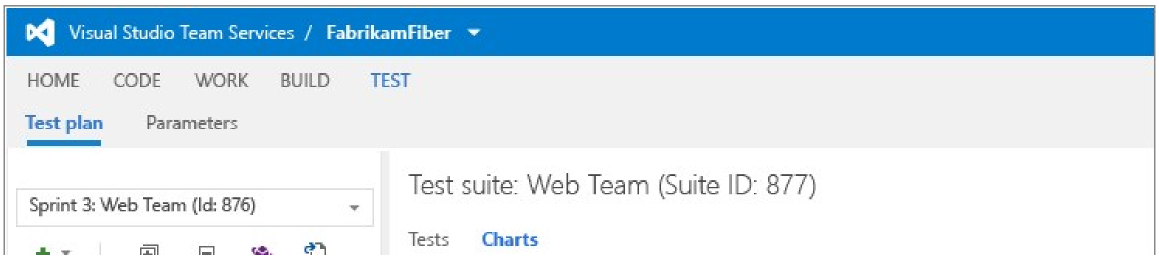
Track test status

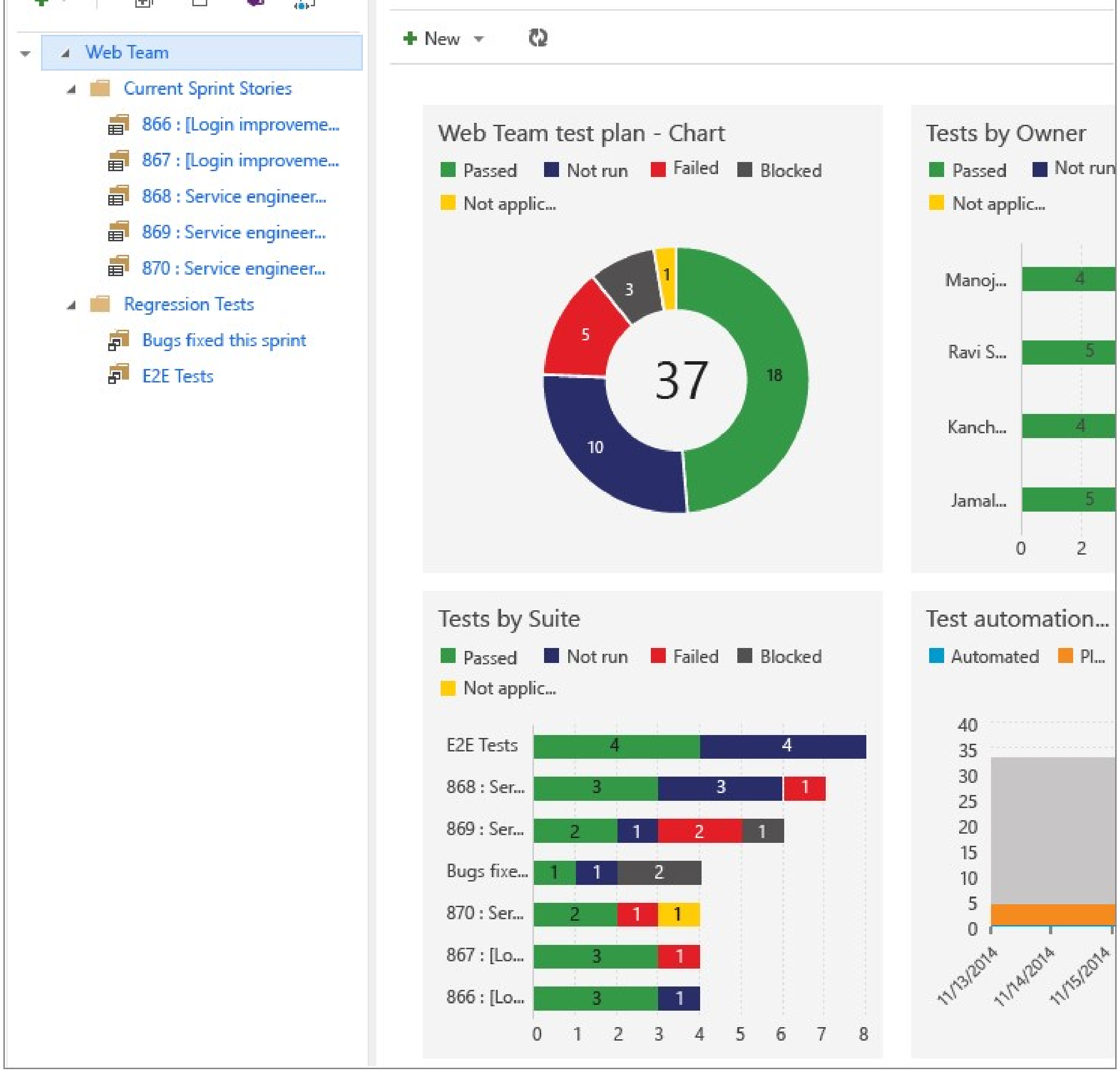
Last Updated: 8/4/2016

IN THIS ARTICLE +

Visual Studio 2015 | TFS 2015 | Previous version

Quickly view the status of your testing using lightweight charts. For example, find out how many test cases are ready to run, or how many tests are passing and failing in each test suite. You can pin these charts to your home page, then all the team can see the progress at a glance.

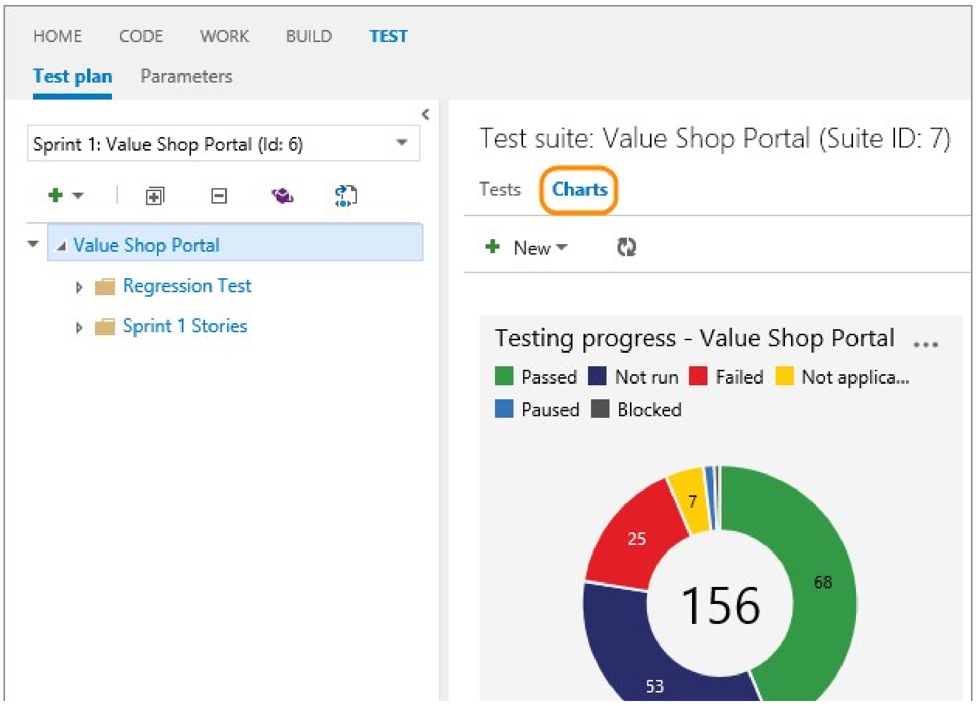

Visual Studio Team Services / FabrikamFiber
HOME
CODE
WORK
BUILD
TEST
Test plan
Parameters
Sprint 3: Web Team (Id: 876)
Test suite: Web Team (Suite ID: 877)
Tests
Charts



Track testing progress

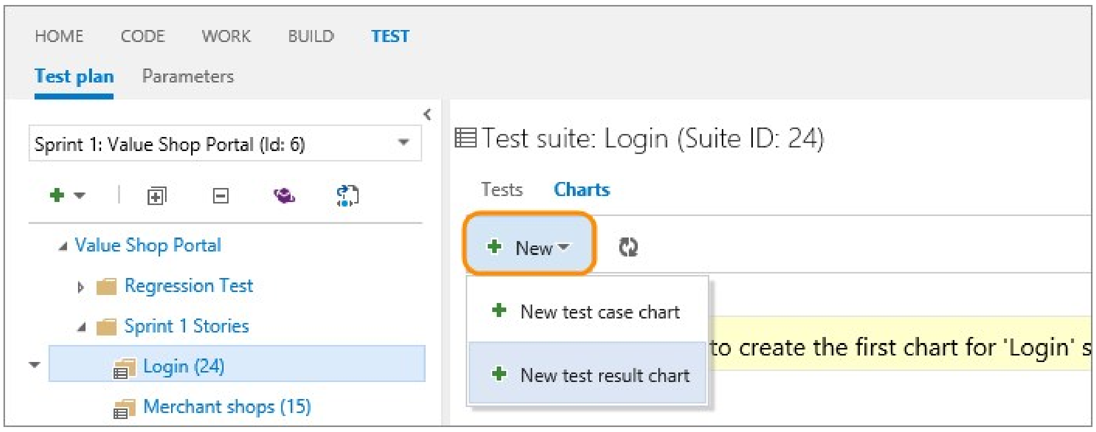
Use test results charts to track how your testing is going. Choose from a fixed set of pre-populated fields related to results. By default, a pie chart is created for each test plan. This chart is grouped by the outcome field to show the latest results for all the tests in the test plan.

View this default chart from the Charts tab.

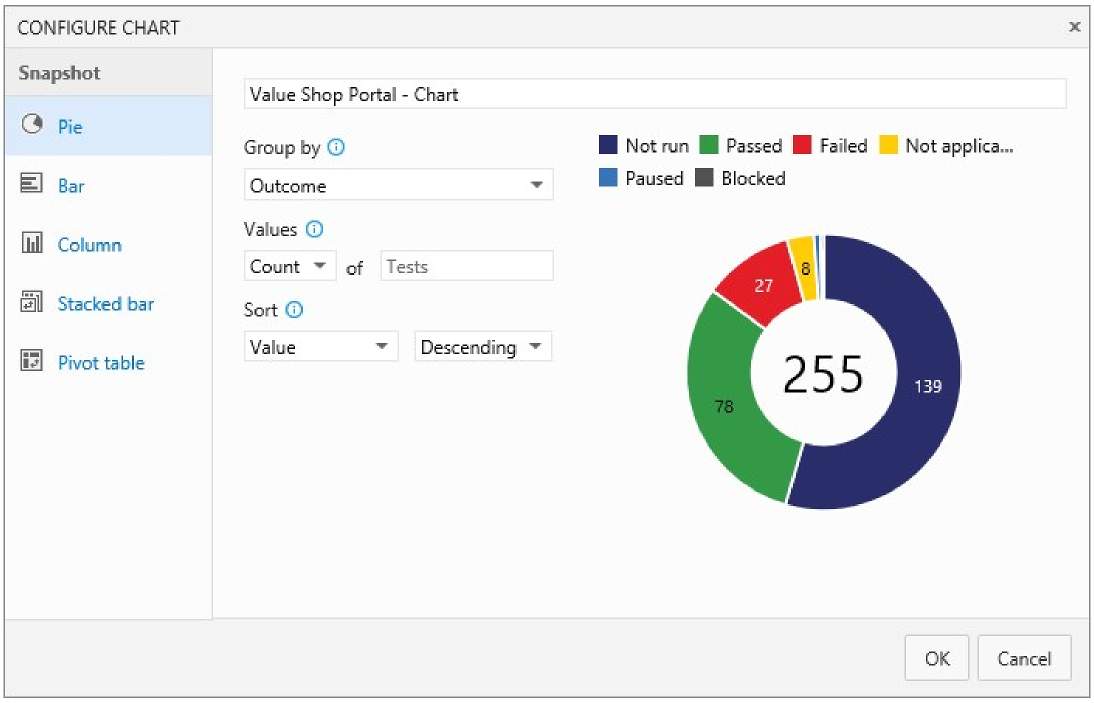


Add your own charts for test results to visualize what's important for your team. If you already know how to add a chart, jump to the examples below of charts that you can create.

1. Select the test plan or test suite for your chart in the Test plan tab. Then create a new chart.



2. Select the chart type. Based on the chart, configure the fields that you want to use to group by, or for rows and columns.



CONFIGURE CHART

Snapshot: Pie, Bar, Column, Stacked bar, Pivot table

Value Shop Portal - Chart

Group by: Outcome

Values: Count of Tests

Sort: Value, Descending

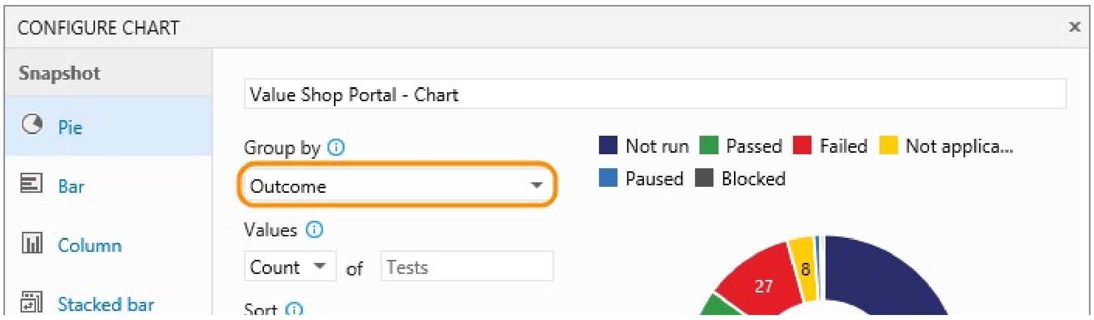
All charts roll up the information for any child test suites of the test plan or test suite that you selected.

3. Save the chart. Now it will be displayed in the charts tab for the test plan or test suite that you selected.

Test results examples

What's the test status for a specific test suite?

Select the test suite from the Test plan tab and add a test results pie chart. Group by outcome.



CONFIGURE CHART

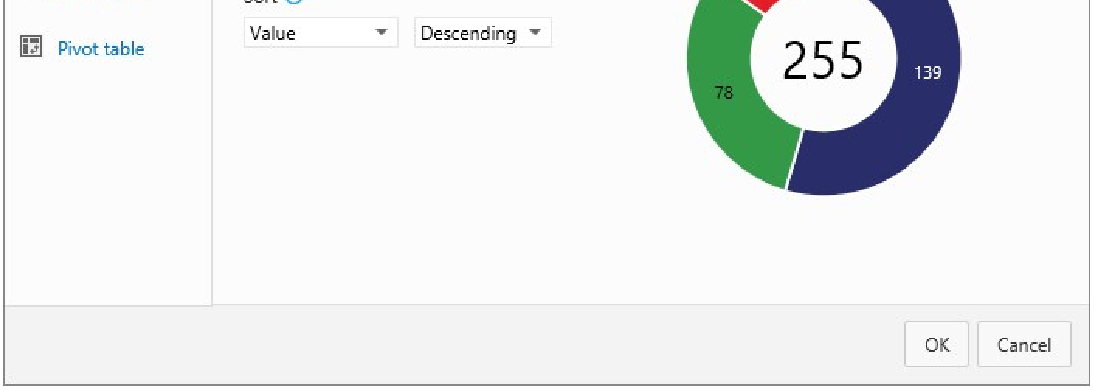
Snapshot: Pie, Bar, Column, Stacked bar

Value Shop Portal - Chart

Group by: Outcome

Values: Count of Tests

Sort

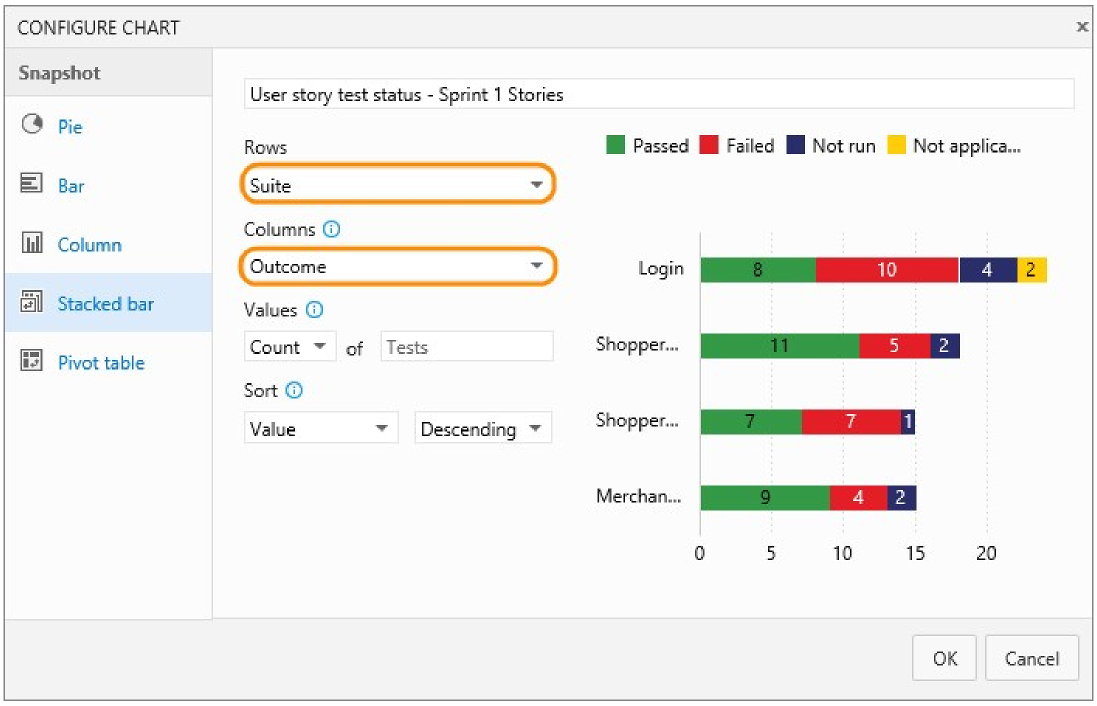

Pivot table

Value | Descending

What's the test status for user stories that my team's testing this sprint?

If you have created requirement-based test suites in your test plan for your user stories, you can create a chart for this.

1. Group these requirement-based test suites together in a static test suite.
2. Select this static test suite in the Test plan tab.
3. Add a test results stacked bar chart. Choose Suite as the rows pivot and Outcome as the columns pivot.


CONFIGURE CHART

Snapshot: Pie, Bar, Column, Stacked bar, Pivot table

User story test status - Sprint 1 Stories

Rows: Suite

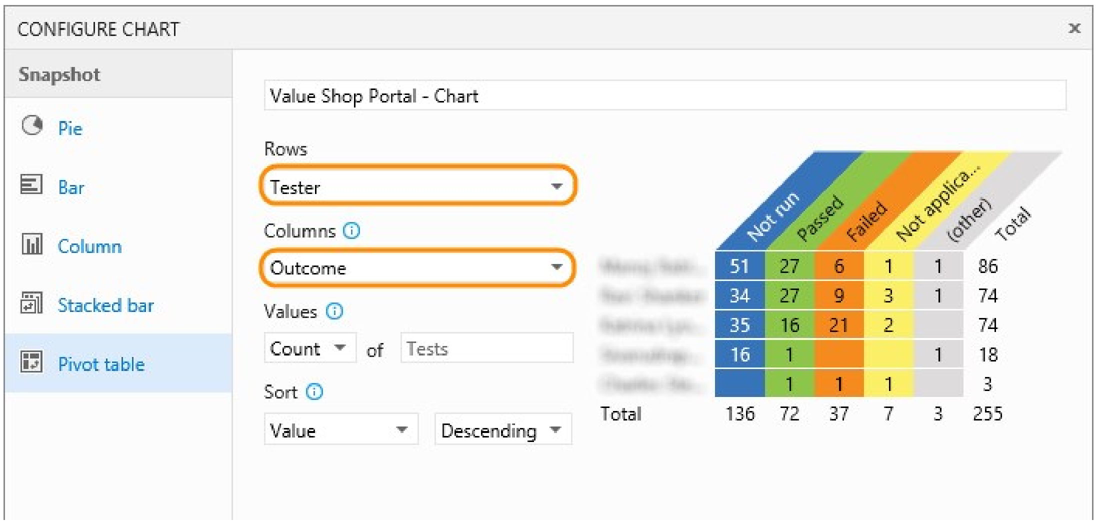
Columns: Outcome

Values: Count of Tests

Sort: Value | Descending

How many tests has each tester left to run?

Select your test plan from the Test plan tab and add a test results pivot table chart. Choose Tester as the rows pivot and Outcome as the columns pivot.


CONFIGURE CHART

Snapshot: Pie, Bar, Column, Stacked bar, Pivot table

Value Shop Portal - Chart

Rows: Tester

Columns: Outcome

Values: Count of Tests

Sort: Value | Descending

	Not run	Passed	Failed	Not applica...	(other)	Total
[redacted]	51	27	6	1	1	86
[redacted]	34	27	9	3	1	74
[redacted]	35	16	21	2		74
[redacted]	16	1			1	18
[redacted]		1	1	1		3
Total	136	72	37	7	3	255

How can I check quality based on the configuration?

Use either a stacked bar chart or a pivot table chart. Choose Configuration as the rows pivot and Outcome as the columns pivot.

How can I track why tests are failing for my team?

For failure analysis, use either a stacked bar chart or a pivot table chart. Choose Tester for the rows and Failure type for the columns. (Failure type for test results can only be set using Microsoft Test Manager.)

How can I track the resolution for failing tests for my team?

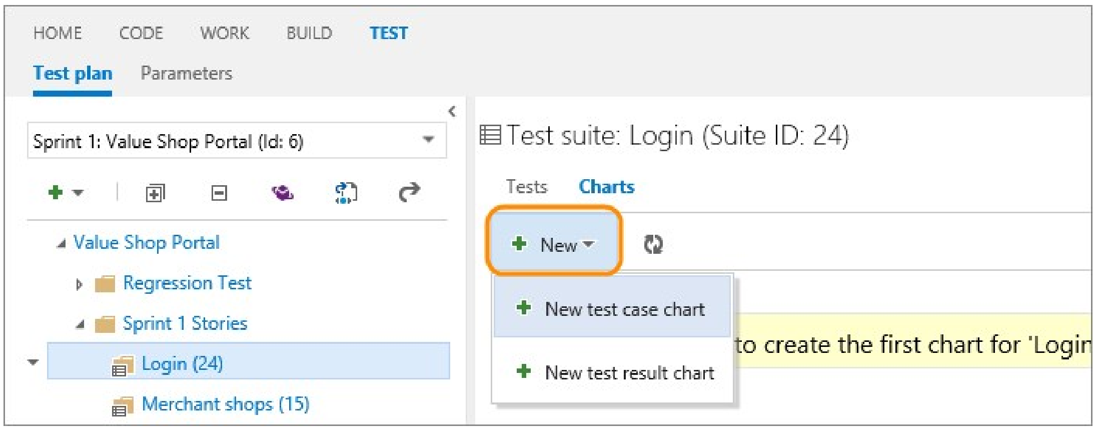
For resolution analysis, use either a stacked bar chart or a pivot table chart. Choose Tester for the rows and Resolution for the columns. (Resolution type for test results can only be set using Microsoft Test Manager.)

Track test case status

Use test case charts to find out the progress of your test case authoring. The charts for test cases give you the flexibility to report on columns that you add to the Tests tab. By default, test case fields are not added to the view in the Tests tab.

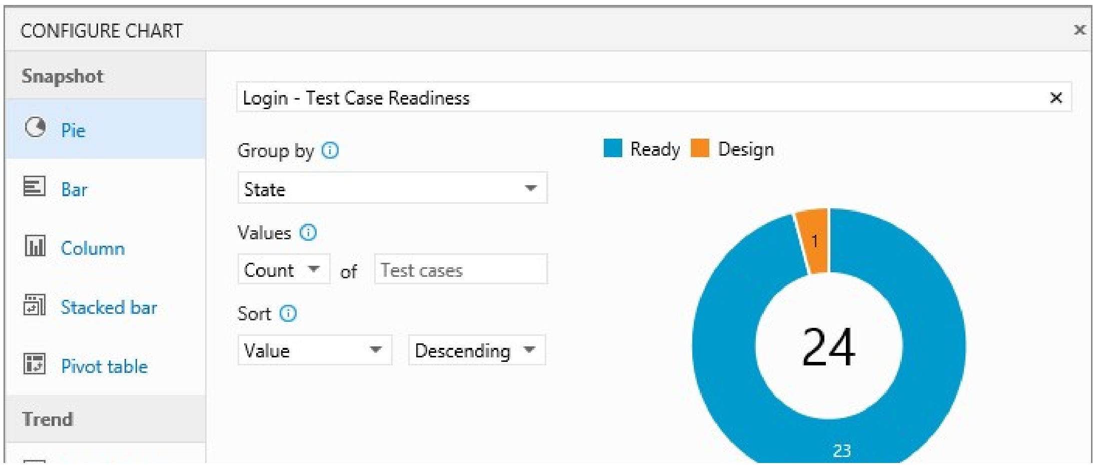
If you already know how to add a chart, jump to the examples below of charts that you can create for test cases.

1. Add any fields you want to use for your test case chart from the Tests tab with Column options. Then the fields will appear as choices in the drop-down lists for grouping for your test case charts.
2. Select the test plan or test suite for your chart in the Test plan tab. Then add a test case chart.



All charts roll up the information for any child test suites of the test plan or test suite that you selected.

3. Select the chart type. Based on the chart, configure the fields that you want to use to group by, for rows and columns, or the range (trend charts only).



Login - Test Case Readiness

Group by: State

Values: Count of Test cases

Sort: Value, Descending

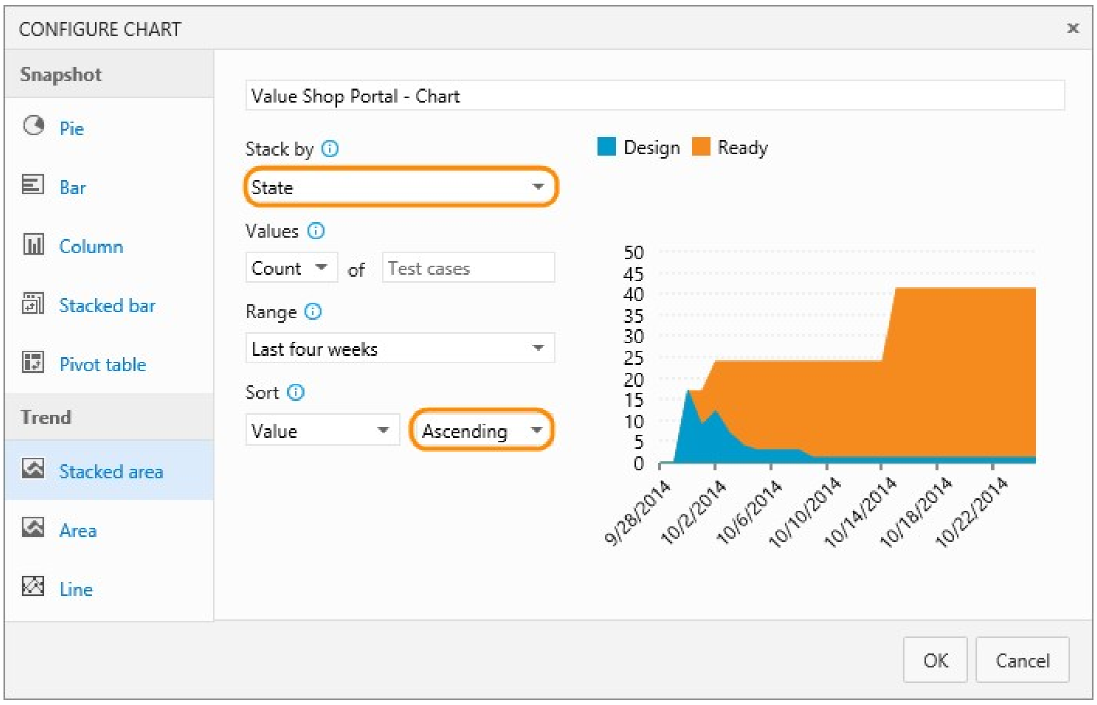
You can't group by test suite for the test case charts.

4. Save the chart. Now it will be displayed in the charts tab for the test plan or test suite that you selected.

Test case examples

How can I track burndown for test case creation?

Use a stacked area trend chart to view the burndown for how many test cases are ready to be run. Choose State for the stack by field and Ascending for the sort field.



Value Shop Portal - Chart

Stack by: State

Values: Count of Test cases

Range: Last four weeks

Sort: Value, Ascending

How can I track burndown for automation status?

Use a stacked area trend chart to view the burndown for how many test cases have been automated. Choose Automation status for the stack by field and Ascending for the sort field.

If multiple teams own test cases in my test plan, can I see how many each team owns and the priorities of the tests?

If your teams are organized by area path, then your can use a test case pie chart. Choose Area path for the group by field.

If you want to know the priorities of these tests, then create a stacked bar chart. Choose Area path for rows and priority for the columns.

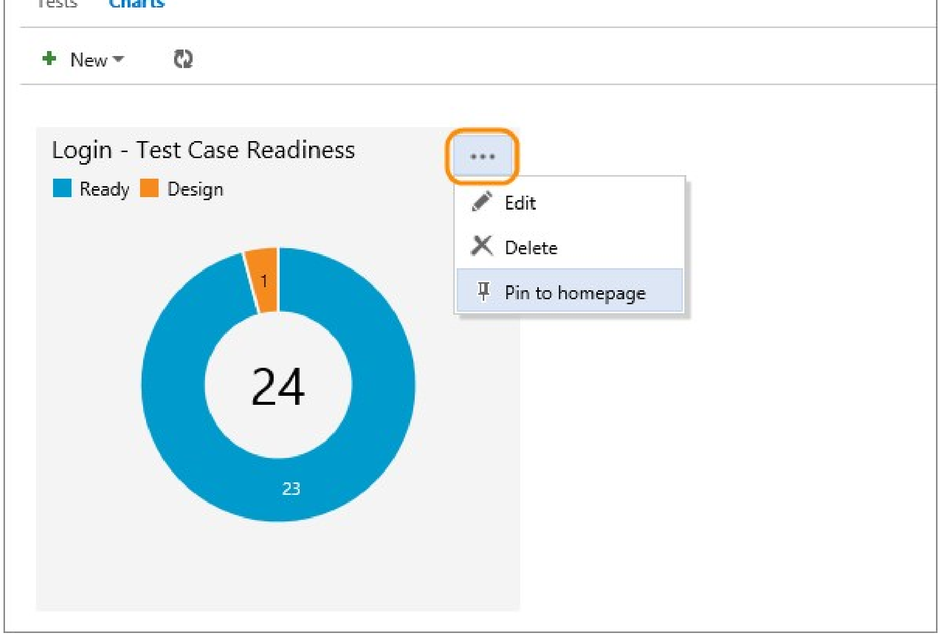
How can I track test creation status by team members?

Test case owners are tracked by the Assigned to field. Use a stacked bar chart or a pivot table chart. Choose Assigned to for rows and status for the columns.

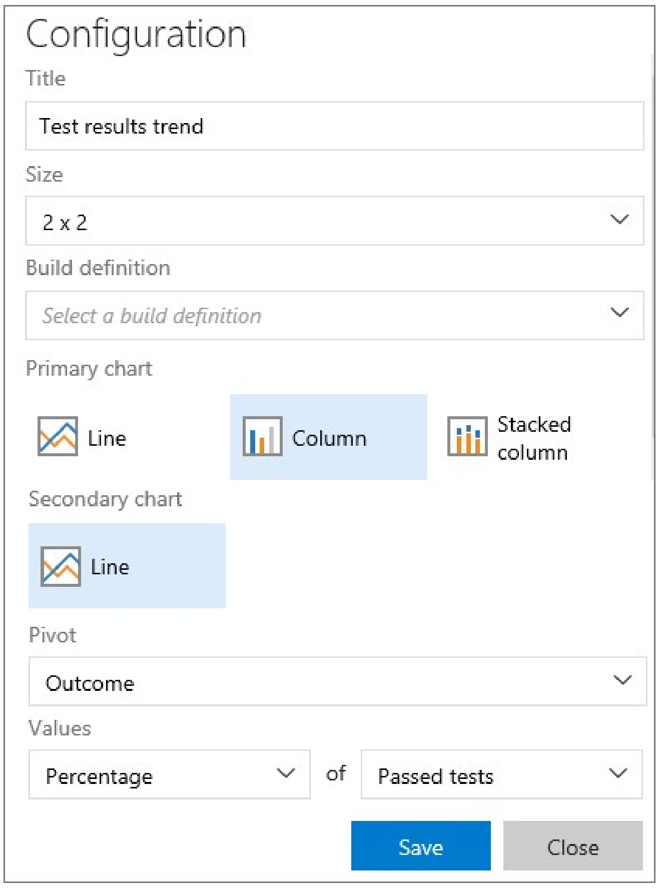
Share charts on your team's dashboard

Pin a chart to your team's dashboard for all the team to view. Use the chart's context menu.

Test suite: Login (Suite ID: 24)



You can configure the dashboard widget to show a range of chart types.



Title: Test results trend
Size: 2 x 2
Build definition: Select a build definition
Primary chart: Column
Secondary chart: Line
Pivot: Outcome
Values: Percentage of Passed tests

You must be a team administrator to do this, but team members with Stakeholder access can view the charts on the dashboard. Learn more about dashboards. Or learn more about team administration.

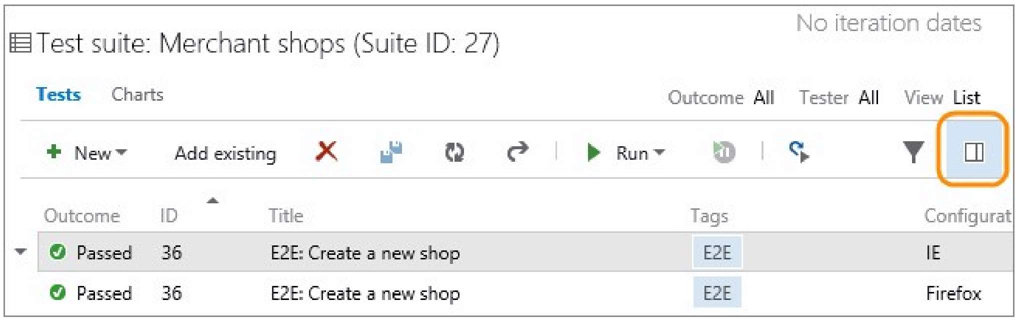
Try this next

- Control how long to keep test results

Q&A

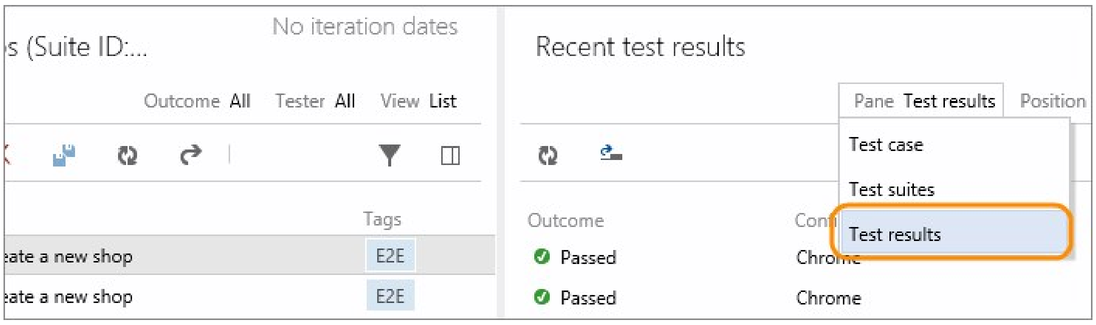
Q: Can I view the recent test results for an individual test case?

A: Yes, select the test case within a test suite and then choose to view the test details pane.



Outcome	ID	Title	Tags	Configurat
Passed	36	E2E: Create a new shop	E2E	IE
Passed	36	E2E: Create a new shop	E2E	Firefox

View the recent test results for this test case.



Q: How is data shown in the charts for test cases that are in multiple test suites?

A: For test case charts, if a test case has been added to multiple test suites in a plan then it's only counted once.

For test result charts, each instance of a test that is run is counted for each of the test suites separately.

Q: Who can create charts?

A: You need at least a Basic access to create charts.

Q: How can I edit or delete a chart?

A: Select the option you want from the chart's context menu.

Q: How do I control how long I keep my test data?

A: Learn more here.

Help and support

Submit bugs through Connect, make suggestions on Uservoice, and send quick thoughts using the Send-a-Smile link in the Visual Studio, Team Services, or TFS title bar. We look forward to your feedback.

Table of contents

Testing scenarios

Manual and exploratory testing

Overview

Get started with manual testing

Add, run, and track tests from the Kanban board

Create test plans

Create test cases

Run manual tests

Track test status

Advanced manual testing techniques

Manual testing with Microsoft Test Manager

Get started with exploratory testing

Advanced exploratory testing techniques

Exploratory testing with Microsoft Test Manager

Performance testing

Continuous testing

Developer testing tools

Test lab management

API reference for test tools

Run manual tests

Last Updated: 8/4/2016

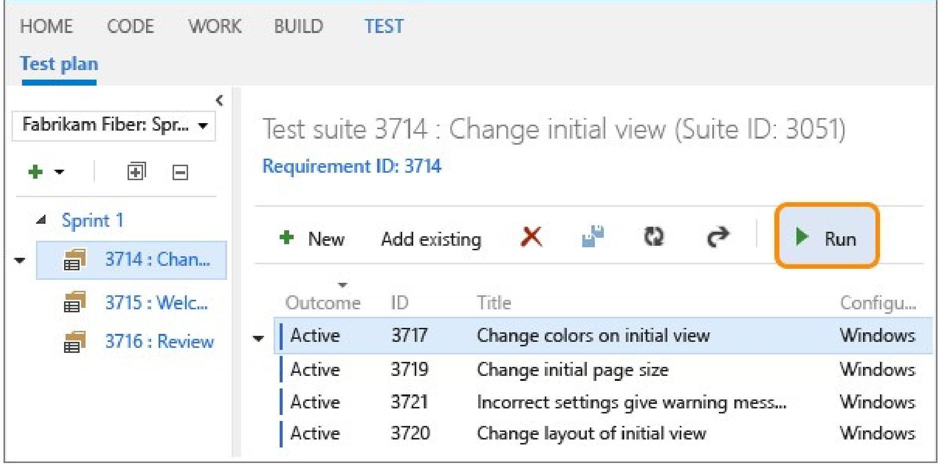
IN THIS ARTICLE +

Visual Studio 2015 | TFS 2015 | Previous version

Run your manual tests and record the test results for each test step using Microsoft Test Runner. If you find an issue when testing, use Test Runner to create a bug. Test steps, screenshots, and comments are automatically included in the bug.

You just need Basic access to run tests that have been assigned to you with Visual Studio Team Services. Learn more about the access that you need for more advanced testing features.

1. If you haven't already, create your manual tests.
2. Select a test from a test suite and run it.


HOME CODE WORK BUILD TEST

Test plan

Fabrikam Fiber: Spr...

Sprint 1
- 3714 : Chan...
- 3715 : Welc...
- 3716 : Review

Test suite 3714 : Change initial view (Suite ID: 3051)

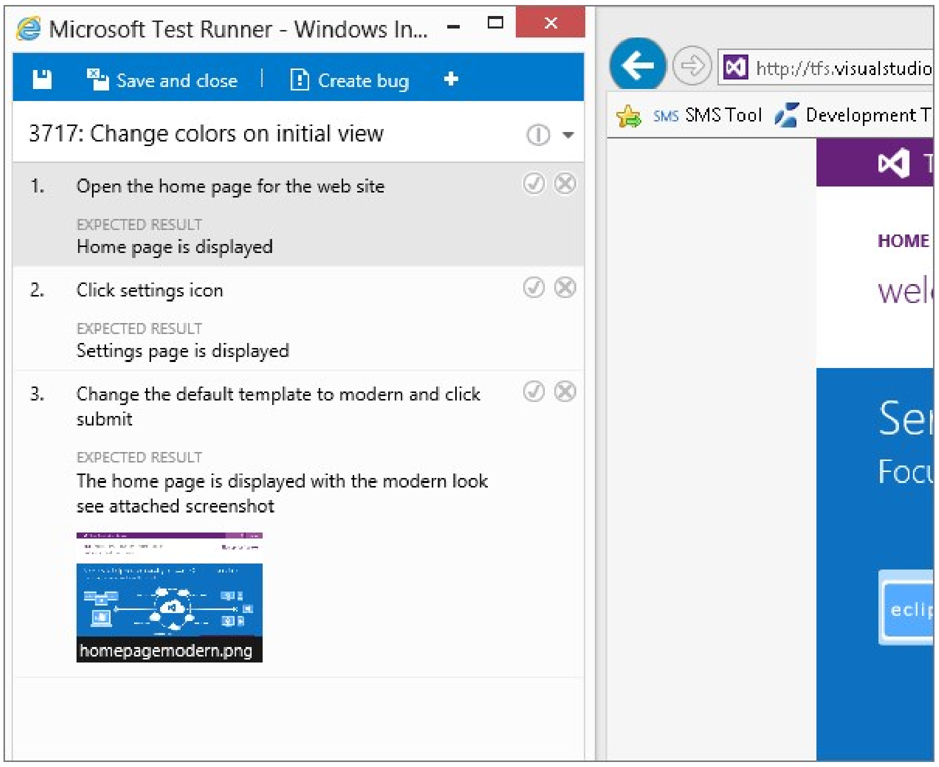
Requirement ID: 3714

New | Add existing | Run

Outcome	ID	Title	Configu...
Active	3717	Change colors on initial view	Windows
Active	3719	Change initial page size	Windows
Active	3721	Incorrect settings give warning mess...	Windows
Active	3720	Change layout of initial view	Windows

Microsoft Test Runner opens and runs in a new browser.

3. Start the app that you want to test. Your app doesn't have to run on the same computer as Test Runner. You just use Test Runner to record which test steps pass or fail while you manually run a test. For example, you might run Test Runner on a desktop computer and run your Windows 8 store app that you are testing on a Windows 8 tablet.


Microsoft Test Runner - Windows In...

Save and close | Create bug

3717: Change colors on initial view

1. Open the home page for the web site

EXPECTED RESULT
Home page is displayed

2. Click settings icon

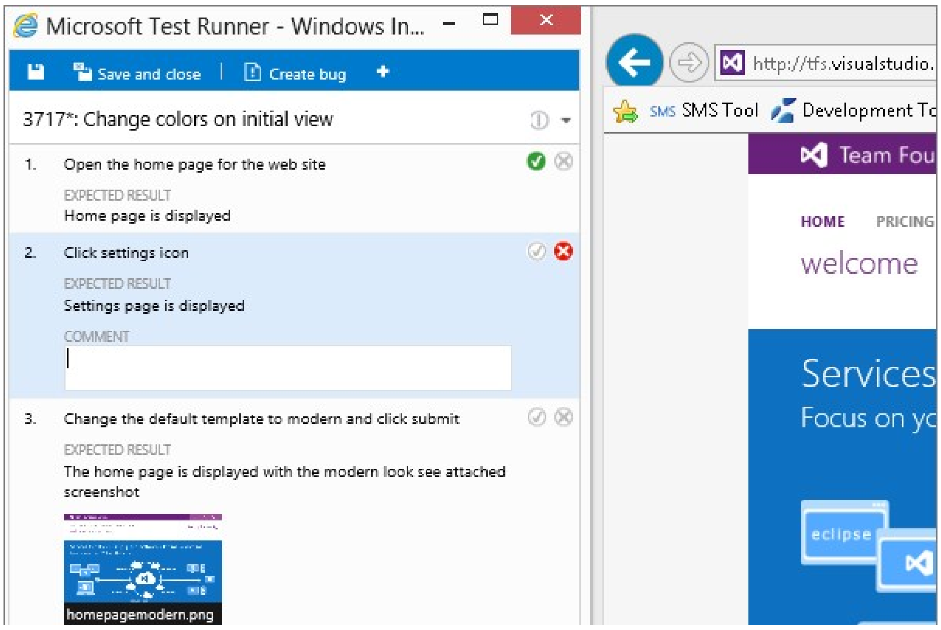
EXPECTED RESULT
Settings page is displayed

3. Change the default template to modern and click submit

EXPECTED RESULT
The home page is displayed with the modern look see attached screenshot

homepagemodern.png

4. Mark each test step as either passed or failed based on the expected results. If a test step fails, you can enter a comment on why it failed.


Microsoft Test Runner - Windows In...

Save and close | Create bug

3717*: Change colors on initial view

1. Open the home page for the web site

EXPECTED RESULT
Home page is displayed

2. Click settings icon

EXPECTED RESULT
Settings page is displayed

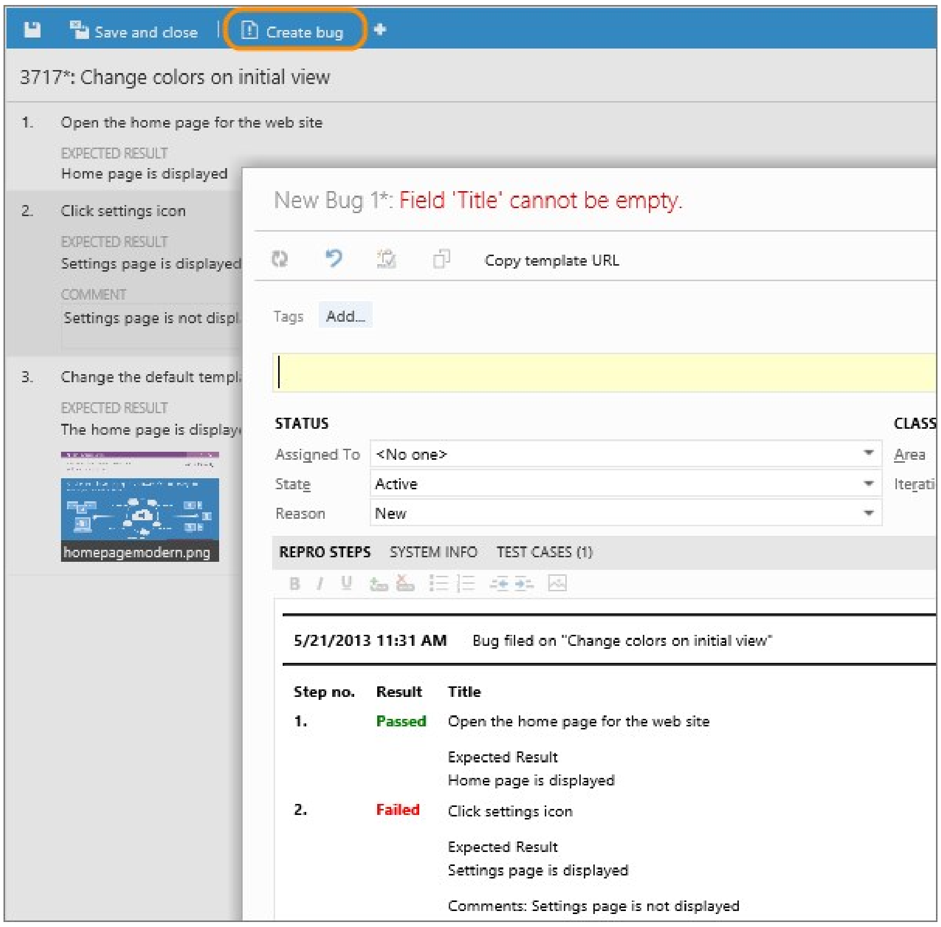
COMMENT

3. Change the default template to modern and click submit

EXPECTED RESULT
The home page is displayed with the modern look see attached screenshot

homepagemodern.png

5. Create a bug to describe what failed.



3717*: Change colors on initial view

1. Open the home page for the web site
EXPECTED RESULT
Home page is displayed

2. Click settings icon
EXPECTED RESULT
Settings page is displayed
COMMENT
Settings page is not displ

3. Change the default templ
EXPECTED RESULT
The home page is display

homepagemodern.png

New Bug 1*: Field 'Title' cannot be empty.

STATUS

Assigned To <No one>
State Active
Reason New

CLASS
Area
Iterati

REPRO STEPS SYSTEM INFO TEST CASES (1)

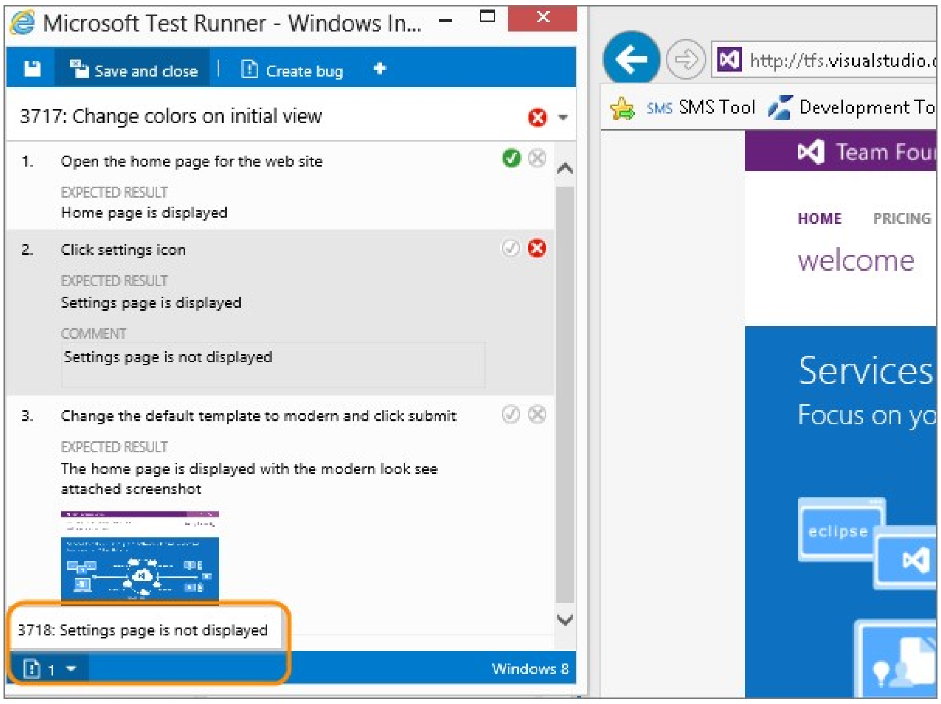
5/21/2013 11:31 AM Bug filed on "Change colors on initial view"

Step no.	Result	Title
1.	Passed	Open the home page for the web site Expected Result Home page is displayed
2.	Failed	Click settings icon Expected Result Settings page is displayed Comments: Settings page is not displayed

The steps and your comments are automatically added to the bug. Also, the test case is linked to the bug.

If Test Runner is running in an Internet Explorer 11 or a Chrome window, you can copy a screenshot from the clipboard directly into the bug.

6. You can see any bugs that you have reported during your test session.



3717: Change colors on initial view

1. Open the home page for the web site
EXPECTED RESULT
Home page is displayed

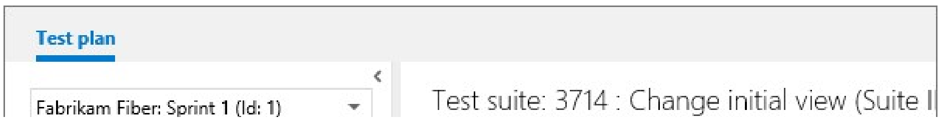
2. Click settings icon
EXPECTED RESULT
Settings page is displayed
COMMENT
Settings page is not displayed

3. Change the default template to modern and click submit
EXPECTED RESULT
The home page is displayed with the modern look see attached screenshot

3718: Settings page is not displayed

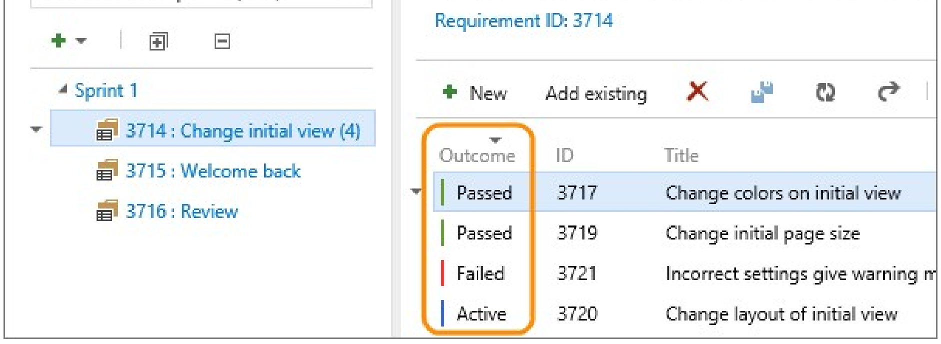
7. When you've run all your tests, save the results and close Test Runner. Now, all the test results are stored in Visual Studio Team Services.

8. View the testing status for your test suite.



Fabrikam Fiber: Sprint 1 (Id: 1)

Test suite: 3714 : Change initial view (Suite I


Requirement ID: 3714

Outcome	ID	Title
Passed	3717	Change colors on initial view
Passed	3719	Change initial page size
Failed	3721	Incorrect settings give warning m
Active	3720	Change layout of initial view

You see the most recent results for each test.

Try this next

- View your test progress with lightweight charts
- Control how long to keep test results

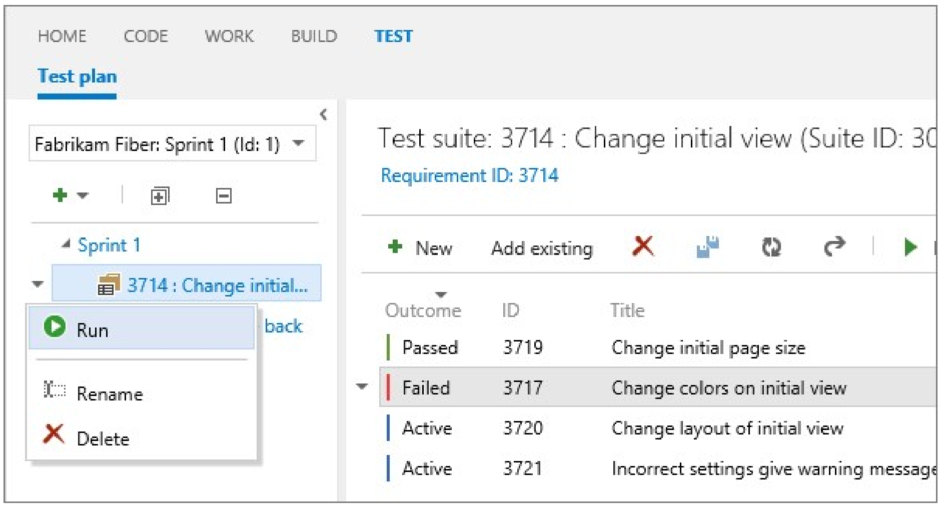
Q&A

Q: How do I rerun a test?

A: Just select any test and choose Run.

Q: Can I run all the tests in a test suite together?

A: Yes, select a test suite and choose Run. This runs all the active tests in the test suite. If you haven't run a test yet, its state is active. You can reset the state of a test to active if you want to rerun it.


Test suite: 3714 : Change initial view (Suite ID: 30

Requirement ID: 3714

Outcome	ID	Title
Passed	3719	Change initial page size
Failed	3717	Change colors on initial view
Active	3720	Change layout of initial view
Active	3721	Incorrect settings give warning message

Q: Can I choose a build to run tests against?

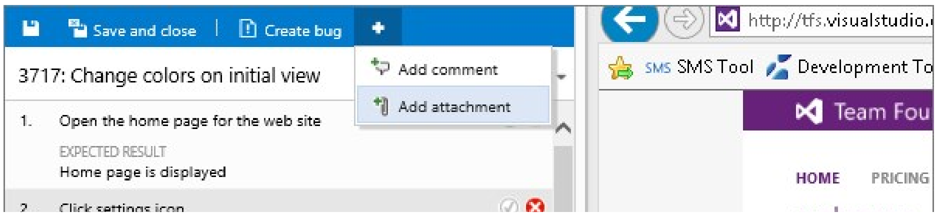
A: Yes, Choose **Run** and then select **Run with options**. Any bug filed during the run will automatically be associated with the selected build, and the test outcome will be published against that build.

Q: I want to do some exploratory testing before I create manual test cases. Can Test Runner help with this?

A: Not from the Test hub. But if you use Microsoft Test Manager, it will record your actions, screenshots and other data while you're exploring your app. If you create a bug, all this data is included automatically.

Q: Can I add a screenshot to the test results when I am running a test?

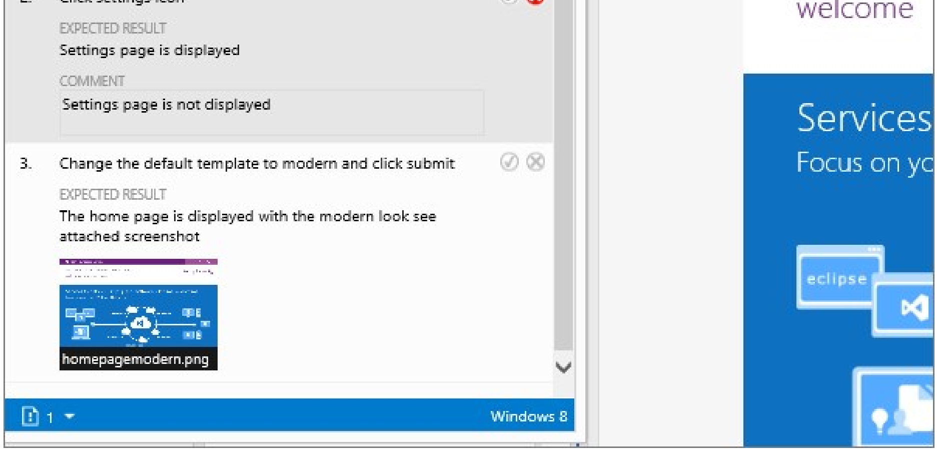
A: Yes, take a screenshot, save it to a file and add the attachment. The file is stored with the test results.


3717: Change colors on initial view

1. Open the home page for the web site

EXPECTED RESULT
Home page is displayed

2. Click settings icon


EXPECTED RESULT
Settings page is displayed

COMMENT
Settings page is not displayed

3. Change the default template to modern and click submit

EXPECTED RESULT
The home page is displayed with the modern look see attached screenshot

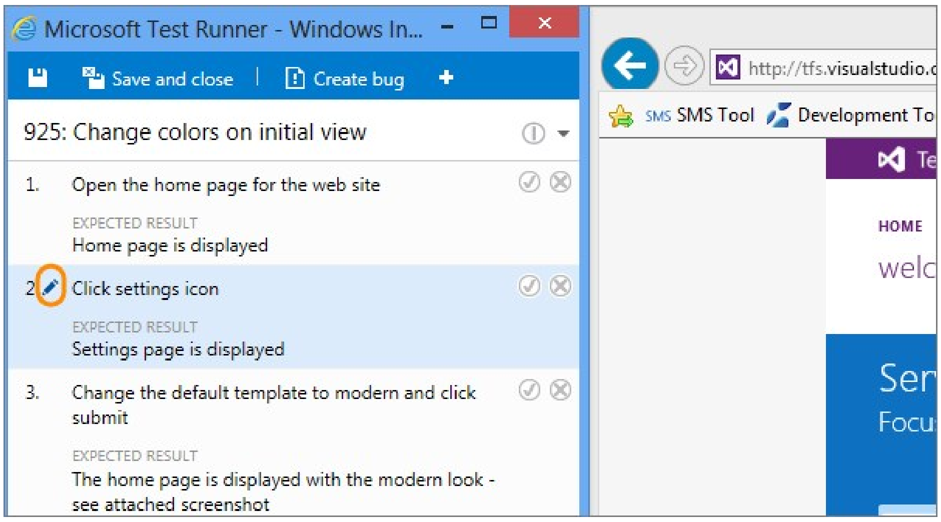
homepagemodern.png

Q: Can I add a screenshot to a bug when I am running a test?

A: Yes, if Test Runner is running in an Internet Explorer 11 or a Chrome window, you can copy a screenshot from the clipboard directly.

Q: Can I fix my test steps while I'm running a test?

A: Yes, if you have the Test Manager for Visual Studio Team Services. You can insert, move, or delete steps. Or you can edit the text itself. Use the edit icon next to the test step number to do this.


Microsoft Test Runner - Windows In...

925: Change colors on initial view

1. Open the home page for the web site

EXPECTED RESULT
Home page is displayed

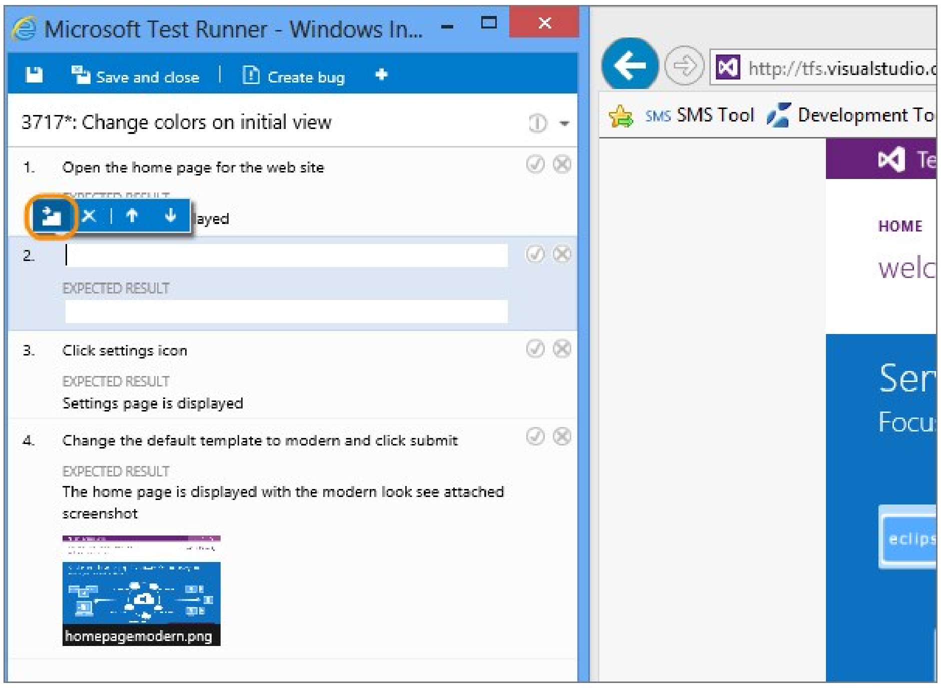
2. Click settings icon

EXPECTED RESULT
Settings page is displayed

3. Change the default template to modern and click submit

EXPECTED RESULT
The home page is displayed with the modern look - see attached screenshot

The tool to edit the test steps is shown.


Microsoft Test Runner - Windows In...

3717*: Change colors on initial view

1. Open the home page for the web site

EXPECTED RESULT
...ayed

2.

EXPECTED RESULT

3. Click settings icon

EXPECTED RESULT
Settings page is displayed

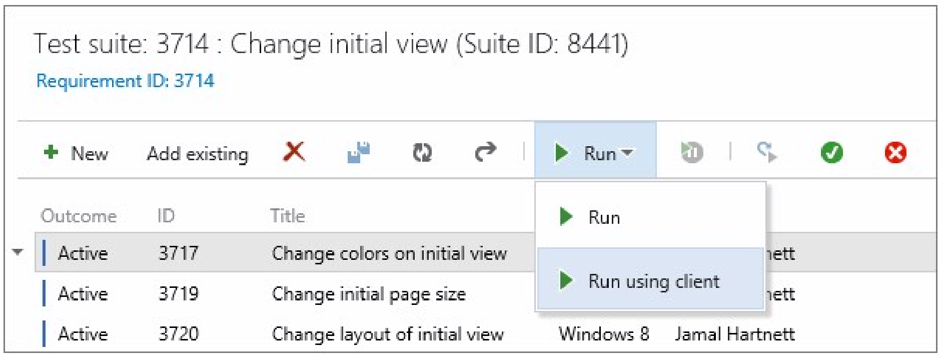
4. Change the default template to modern and click submit

EXPECTED RESULT
The home page is displayed with the modern look see attached screenshot

homepagemodern.png

Q: Can I collect additional data while I'm running a test?

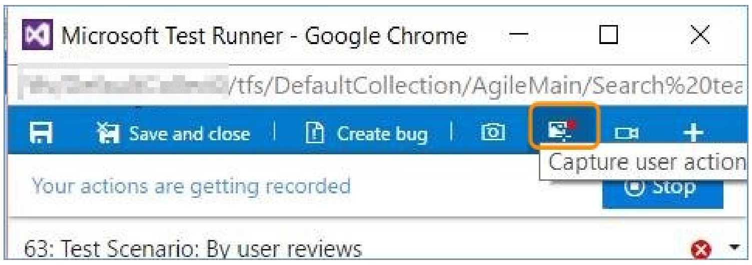
A: If you use Microsoft Test Manager to run your tests, you can collect user actions, system logs, screen and audio recordings and other additional data. If you're using Visual Studio 2015, Visual Studio 2013, or Visual Studio 2012 Update 3, you can run a test using Microsoft Test Manager from the Test hub. (The most recently installed version of MTM will launch.)


Test suite: 3714 : Change initial view (Suite ID: 8441)
Requirement ID: 3714

Outcome	ID	Title		
Active	3717	Change colors on initial view		…nett
Active	3719	Change initial page size		…nett
Active	3720	Change layout of initial view	Windows 8	Jamal Hartnett

Q: Can I capture on-demand image actions?

A: Yes, in addition to screenshots and screen recordings you can capture an on-demand image action log from your web apps. You specify the browser window on which to capture your actions � all actions on that window (any existing or new tabs you open in that window) or any new child browser windows you launch, will automatically be captured and correlated against the steps being tested in the Web Runner. These image action logs are then added to any bugs you file during the run and also attached to the current test result.


Microsoft Test Runner - Google Chrome
/tfs/DefaultCollection/AgileMain/Search%20tea
Your actions are getting recorded
63: Test Scenario: By user reviews

Q: How do I control how long I keep my test data?

A: Learn more here.

Help and support

Submit bugs through Connect, make suggestions on Uservoice, and send quick thoughts using the Send-a-Smile link in the Visual Studio, Team Services, or TFS title bar. We look forward to your feedback.

YES NO