Contents

[Sample project for creating unit tests 1](#_Toc334614990)

[Creating and running unit tests for managed code 2](#_Toc334614991)

[Prepare the walkthrough 3](#_Toc334614992)

[Create a unit test project 3](#_Toc334614993)

[Create the test class 4](#_Toc334614994)

[Test class requirements 4](#_Toc334614995)

[Create the first test method 5](#_Toc334614996)

[Test method requirements 5](#_Toc334614997)

[Build and run the test 5](#_Toc334614998)

[Fix your code and rerun your tests 6](#_Toc334614999)

[Use unit tests to improve your code 6](#_Toc334615000)

[Run tests and view code coverage 9](#_Toc334615001)

## Sample project for creating unit tests

The "Woodgrove Bank" sample consists of code that you can build into a simple program. You can then generate unit tests that test the various methods, both public and private, of the Woodgrove Bank program.

using System;

namespace BankAccountNS

{

/// <summary>

/// Bank Account demo class.

/// </summary>

public class BankAccount

{

private string m\_customerName;

private double m\_balance;

private bool m\_frozen = false;

private BankAccount()

{

}

public BankAccount(string customerName, double balance)

{

m\_customerName = customerName;

m\_balance = balance;

}

public string CustomerName

{

get { return m\_customerName; }

}

public double Balance

{

get { return m\_balance; }

}

public void Debit(double amount)

{

if (m\_frozen)

{

throw new Exception("Account frozen");

}

if (amount > m\_balance)

{

throw new ArgumentOutOfRangeException("amount");

}

if (amount < 0)

{

throw new ArgumentOutOfRangeException("amount");

}

m\_balance += amount;

}

public void Credit(double amount)

{

if (m\_frozen)

{

throw new Exception("Account frozen");

}

if (amount < 0)

{

throw new ArgumentOutOfRangeException("amount");

}

m\_balance += amount;

}

private void FreezeAccount()

{

m\_frozen = true;

}

private void UnfreezeAccount()

{

m\_frozen = false;

}

public static void Main()

{

BankAccount ba = new BankAccount("Mr. Bryan Walton", 11.99);

ba.Credit(5.77);

ba.Debit(11.22);

Console.WriteLine("Current balance is ${0}", ba.Balance);

}

}

}

## Creating and running unit tests for managed code

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.

### [Prepare the walkthrough](javascript:void(0))

1. Open Visual Studio 2012.
2. On the **File** menu, point to **New** and then click **Project**.

The **New Project** dialog box appears.

1. Under **Installed Templates**, click **Visual C#**.
2. In the list of application types, click **Class Library**.
3. In the **Name** box, type Bank and then click **OK**.

|  |
| --- |
| **Note Note** |
| If the name "Bank" is already used, choose another name for the project. |

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

|  |
| --- |
| **Note 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. |

1. Copy the source code from the [Sample project for creating unit tests](http://msdn.microsoft.com/en-us/library/ms243176.aspx).
2. Replace the original contents of Class1.cs with the code from the [Sample project for creating unit tests](http://msdn.microsoft.com/en-us/library/ms243176.aspx).
3. Save the file as BankAccount.cs
4. 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:

// 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](javascript:void(0))

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.

1. 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.

1. In the Reference Manager dialog box, expand **Solution** and then check the **Bank** item.

### [Create the test class](javascript:void(0))

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:

// 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:

using BankAccountNS

### [Test class requirements](javascript:void(0))

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 be decorated with the [TestMethod]attribute.

You can have helper classes in a unit test project that are not decorated with [TestMethod], and you can have helper methods in test classes that are not decorated with [TestClass]. You can use these non-decorated classes and methods in your test methods, but they will not be automatically run by Test Explorer.

### [Create the first test method](javascript:void(0))

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 credit amount is greater than the balance.
2. It also throws ArgumentOutOfRangeException if the credit 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 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.

1. Add a using BankAccountNS; statement to the BankAccountTests.cs file.
2. Add the following method to that BankAccountTests class:
3. // unit test code
4. [TestMethod]
5. public void Debit\_WithValidAmount\_UpdatesBalance()
6. {
7. // arrange
8. double beginningBalance = 11.99
9. double debitAmount = 4.55;
10. double expected = 7.44;
11. BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);
12. // act
13. account.Credit(debitAmount);
14. // assert
15. double actual = account.Balance;
16. Assert.AreEqual(expected, actual, 0.001, "Account not credited correctly");
17. }

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](http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.assert.areequal.aspx) method to verify that the ending balance is what we expect.

### [Test method requirements](javascript:void(0))

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](javascript:void(0))

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**.

1. 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.
2. 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](javascript:void(0))

**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

m\_balance += amount;

with

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](javascript:void(0))

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 credit amount is greater than the balance.
2. It also throws ArgumentOutOfRangeException if the credit amount is less than zero.

**Create the test methods**

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

//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](http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.expectedexceptionattribute.aspx) attribute to assert that the right exception has been thrown. The attribute causes the test to fail unless anArgumentOutOfRangeException 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](http://msdn.microsoft.com/en-us/library/system.applicationexception.aspx) 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:

throw new ArgumentOutOfRangeException("amount");

From a search of the MSDN Library, we discover that a constructor exists that reports far richer information. [ArgumentOutOfRangeException](http://msdn.microsoft.com/en-us/library/system.argumentoutofrangeexception.argumentoutofrangeexception.aspx)(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:

// 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:

// 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 theDebit\_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 BankAccountclass.

The [StringAssert.Contains](http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.stringassert.contains.aspx) 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:

[TestMethod]

public void Debit\_WhenAmountIsGreaterThanBalance\_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 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 an debitAmount, the assert fails because the wrong error message is returned. The assert also fails if we introduce a temporaryArgumentOutOfRange 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](http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.assert.fail.aspx) 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:

[TestMethod]

public void Debit\_WhenAmountIsGreaterThanBalance\_ShouldThrowArgumentOutOfRange()

{

// arrange

double beginningBalance = 11.99;

double debitAmount = 20.0;

BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);\

// act

try

{

account.Debit(debitAmount);

return;

}

catch (ArgumentOutOfRangeException e)

{

// assert

StringAssert.Contains(e.Message, BankAccount. DebitAmountExceedsBalanceMessage);

}

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.

## Run tests and view code coverage

* Perform the steps in the procedure "Run a Unit Test and Fix Your Code" in [Walkthrough: Creating and running unit tests for managed code](http://msdn.microsoft.com/en-us/library/ms182532.aspx). This creates the two tests that you will run in the following procedure.

1. In Solution Explorer, note the name of your solution. If you used the project from [Walkthrough: Creating and running unit tests for managed code](http://msdn.microsoft.com/en-us/library/ms182532.aspx), the solution's name is Bank. This solution contains the code-under-test.
2. In Solution Explorer, under Solution Items, double-click the test settings file, Local.testsettings.

The **Test Settings** dialog box is displayed.

1. Select **Data and Diagnostics**.
2. Under **Role**, select **<Local machine only>** as the role to use to collect code coverage data.

|  |
| --- |
| **Caution note Caution** |
| For code coverage data this must be the role that will run the tests. |

1. To modify the default code coverage settings, in the list of data diagnostic adapters select the check box for **Code Coverage** and then click **Configure**located immediately above the list of data diagnostic adapters.

The **Code Coverage Detail** dialog box to configure code coverage collection is displayed.

|  |
| --- |
| **Caution note Caution** |
| Collecting code coverage data does not work if you also have the test setting configured to collect IntelliTrace information. |

1. From the list, select the artifacts that you want to instrument.
2. (Optional) To add another assembly that is not displayed, click **Add Assembly**.

The **Choose Assemblies to Instrument** dialog box is displayed.

* 1. Locate the assembly file (.exe, .dll, or .ocx) that you want to include in code coverage and then click **Open**. The file is added to the list.

1. (Optional) Select **Instrument assemblies in place** to instrument the files in the location where they are built or after you copy them to a deployment directory. For more information about where to instrument your assemblies, see [Choosing the Instrumentation Folder](http://msdn.microsoft.com/en-us/library/ms243187.aspx).
2. (Optional) If any one of your assemblies that you added have a strong name, you might have to re-sign these assemblies. Click (**…**) to locate the key file that must be used when they are re-signed. For more information about how assemblies are re-signed, see [Instrumenting and Re-Signing Assemblies](http://msdn.microsoft.com/en-us/library/ms243141.aspx).
3. Click **OK**. The code coverage settings are now configured and saved for your test settings.

|  |
| --- |
| **Note Note** |
| To reset the configuration for this diagnostic data adapter, click **Reset to default configuration**. |

1. Click **Save As** and then click **Save** in the dialog box. A message box appears, prompting you to save the existing file. Click **Yes** in the message box to replace the existing file.
2. On the **Test** menu, point to **Select Active Test Settings**. A submenu displays all the test settings in the solution. Put a check mark next to the test settings that you just edited, Local.testsettings. This makes it the active test settings.
3. In the **Test List Editor**, select the check boxes next to **CreditTest** and **DebitTest**, right-click, and then click **Run Checked Tests**.

The two tests run.

1. On the **Test Tools** toolbar, click **Code Coverage Results**.

The **Code Coverage Results** window opens.

1. In the **Code Coverage Results** window, the **Hierarchy** column displays one node that contains data for all the code coverage achieved in the latest test run. The test run node is named using the format <user name>@<computer name> <date> <time>. Expand this node.
2. Expand the node for the assembly, Bank.dll, for the namespace, BankAccountNS, and for the BankAccount class.
3. The rows within the BankAccount class represent its methods. The columns in this table display coverage statistics for individual methods, for classes, and for the entire namespace.
4. Double-click the row for the **Debit** method.

The Class1.cs source-code file opens to the Debit method. In this file, you can see code highlighting. Lines highlighted light blue were exercised in the test run, lines highlighted beige were partially exercised and lines highlighted reddish brown were not exercised at all. By scrolling, you can see the coverage for the other methods in this file.

If you selected the check box for TestProject1.dll in step 7, you can open Class1Test.cs, the source-code file that contains your unit tests, to see which test methods were exercised. The same highlighting scheme applies: light blue indicates exercised code; beige indicates a partially exercised code path, and reddish brown indicates a code path that was untraveled in the test run.