Walkthrough: Creating and Running Unit Tests for Managed Code

Visual Studio 2022

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:

1. Prepare the walkthrough
2. Create a unit test project
3. Create the test class
    - Test class requirements
4. Create the first test method
    - Test method requirements
5. Build and run the test
6. Fix your code and rerun your tests
7. Use unit tests to improve your code

Note
For information about how to run tests from a command line, see Walkthrough: Using the Command-line Test Utility.

Prerequisites

- In this lab we will work with the Bank Project. See the code below:

```
using System;

namespace BankAccountNS
{
    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);
        }
    }
}
```

1. Prepare the walkthrough

To 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 “Lab_3” and then click OK.

Note
If the name " Lab_3" is already used, choose another name for the project.

6. The new “Lab_3” 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.

7. Replace the original contents of Class1.cs with the code from the listing of the Bank project presented above.
8. Save the file as BankAccount.cs.
9. On the Build menu, click Build Solution.

You now have a project named Lab_3. It contains source code to test and tools to test it with. In this quick start, we focus on the method `Debit()` which is called when money is withdrawn from the account.

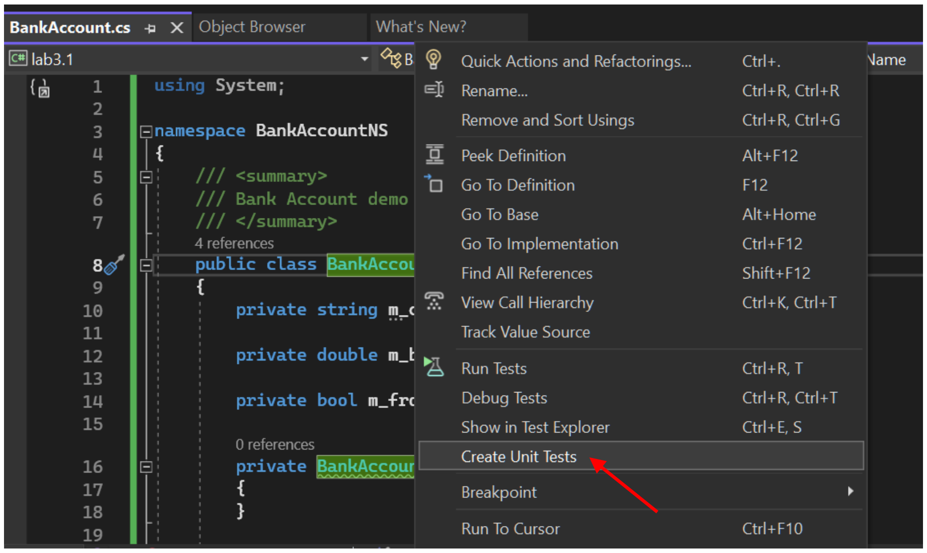
2. Create a unit test project

To create a unit test project – variant one

1. On the File menu, choose Add -> 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 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.

To create a unit test project – variant two

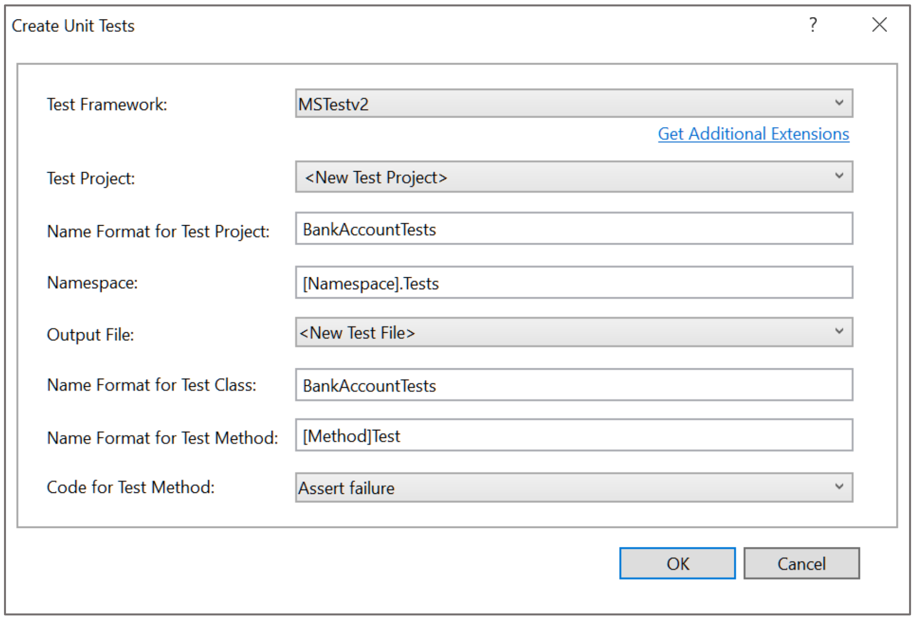
1. Put cursor on a class name and click the right mouse button. Choose an option Create Unit Tests:



```
using System;

namespace BankAccountNS
{
    /// <summary>
    /// Bank Account demo
    /// </summary>
    4 references
    public class BankAccou
    {
        private string m_c

        private double m_b

        private bool m_fro

        0 references
        private BankAccour
        {
        }
```

BankAccount.cs
Object Browser
What's New?
lab3.1
Quick Actions and Refactorings... Ctrl+.
Rename... Ctrl+R, Ctrl+R
Remove and Sort Usings Ctrl+R, Ctrl+G
Peek Definition Alt+F12
Go To Definition F12
Go To Base Alt+Home
Go To Implementation Ctrl+F12
Find All References Shift+F12
View Call Hierarchy Ctrl+K, Ctrl+T
Track Value Source
Run Tests Ctrl+R, T
Debug Tests Ctrl+R, Ctrl+T
Show in Test Explorer Ctrl+E, S
Create Unit Tests
Breakpoint
Run To Cursor Ctrl+F10

2. In the opened window choose Test Framework as “MSTestv2”, and then press “Ok”. By default, a test class for each class in the project will be created. Namely, after this step we will have the test file BankAccountTests.cs.



Create Unit Tests

Test Framework: MSTestv2

Test Project: <New Test Project>

Name Format for Test Project: BankAccountTests

Namespace: [Namespace].Tests

Output File: <New Test File>

Name Format for Test Class: BankAccountTests

Name Format for Test Method: [Method]Test

Code for Test Method: Assert failure

3. Create the test class

Let’s consider the automatically generated test class `BankAccountTests` for verifying the class `BankAccount`. It contains the following code:

```
using ...

namespace BankAccountNS.Tests
{
    [TestClass()]
    public class BankAccountTests
    {
        [TestMethod()]
        public void BankAccountTest()
        {
            Assert.Fail();
        }

        [TestMethod()]
        public void DebitTest()
        {
            Assert.Fail();
        }
    ...
}
```

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

```
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.

4. 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 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.

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:

```
// 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.

5. Build and run the test

To build and run the test:

1. On the Build menu, choose Build Solution. In *Output* window you will see an information about success of the process.
2. Choose Test -> Run All to run the test. Results will be presented in *Test Explorer*. If *Test Explorer* does not appear after a successful build, choose Test -> Test Explorer to open it. 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.

6. Fix your code and rerun your tests

In the written test we succeed to find a bug. To fix the found bug we need to 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). Based on this information and description/specification of the method under test we can correct the code of this method to fix a bug.

After fixing the bug, we need to rerun the test. If the correction was precise, now the test has to pass. We can see in *Test Explorer* that the red/green bar turns green, and the test is moved to the Passed Tests group.

7. 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 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 `ExpectedException` 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 sure 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 there exists a constructor 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:

```
// 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 the `Debit_WhenAmountIsLessThanZero_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 less than zero.
- Assert that the message (the third parameter of the constructor) includes the `DebitAmountLessThanZeroMessage` 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:

```
 [TestMethod]
public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange ()
{
    // arrange
    double beginningBalance = 11.99;
    double debitAmount = -100.0;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);\

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

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 `debitAmount` that is less than zero, the `Contains` assert passes, the exception is ignored, and so the test method passes. This is the behavior we want.
2. If the `debitAmount` is greater than the current balance, 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_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange` looks as follows:

```
[TestMethod]
public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange()
{
    // arrange
    double beginningBalance = 11.99;
    double debitAmount = -100.0;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);\

    // act
    try
    {
        account.Debit(debitAmount);
    }
    catch (ArgumentOutOfRangeException e)
    {
        // assert
        StringAssert.Contains(e.Message, BankAccount.DebitAmountLessThanZeroMessage);
        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. However, more importantly, the extra analysis also led to **better code in our project under test**.