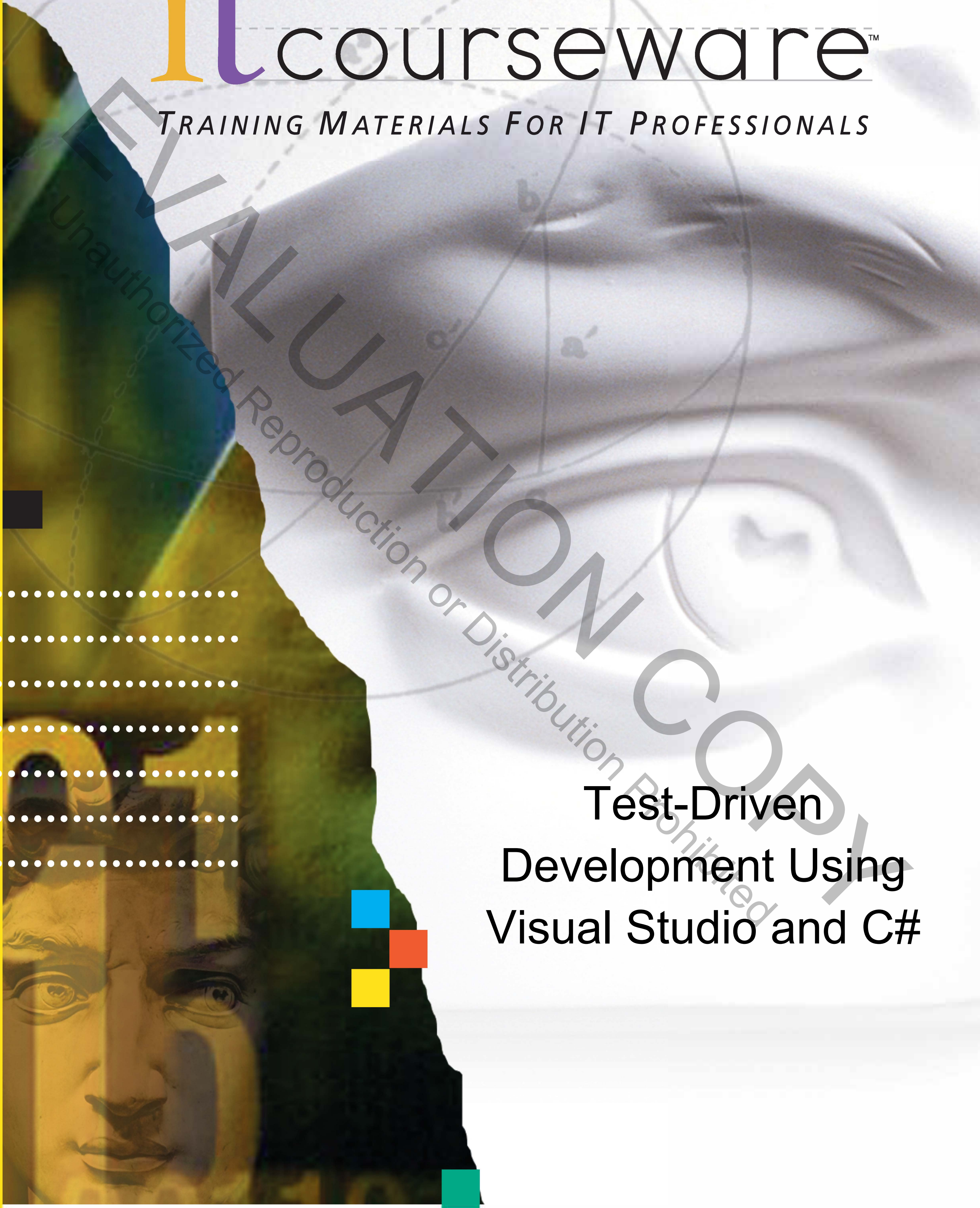
itcourseware™

TRAINING MATERIALS FOR IT PROFESSIONALS



Test-Driven Development Using Visual Studio and C#

Test-Driven Development Using Visual Studio and C#
Rev. 4.8.5

Student Guide

Author: Robert J. Oberg

Object Innovations
877-558-7246
www.objectinnovations.net

Printed in the United States of America.

Table of Contents (Overview)

Directory Structure

- **The course software installs to the root directory *C:\OIC\UnitCs*.**
  - Example programs for each chapter are in named subdirectories of chapter directories **Chap01**, **Chap02**, and so on.
  - A cumulative case study is provided in the directory **CaseStudy**.
  - The **Labs** directory contains one subdirectory for each lab, named after the lab number. Starter code is frequently supplied, and answers are provided in the chapter or case study directories.
  - The **Demos** directory is provided for in-class demonstrations led by the instructor.

Table of Contents (Detailed)

Chapter 1

Test-Driven Development

Test-Driven Development

Objectives

After completing this unit you will be able to:

- **Explain the principles of test-driven development or TDD.**
- **Describe the main types of tests pertaining to TDD:**
  - Functional tests, also known as customer tests
  - Unit tests, also known as programmer tests
- **Discuss the role of test automation in the development process.**
- **Outline the principles of simple design.**
- **Describe the use of refactoring in improving software systems and the role of test automation in support of refactoring.**
- **Describe the Unit Testing Framework in Visual Studio.**
- **Explain the use of TDD in working with legacy code.**

Test-Driven Development

- ***Test-driven development* (TDD) calls for writing test cases *before* functional code.**
  - You write no functional code until there is a test that fails because the function is not present.
- **The test cases embody the requirements that the code must satisfy.**
- **When all test cases pass, the requirements are met.**
- **Both the test cases and the functional code are incrementally enhanced, until all the requirements are specified in tests that the functional code passes.**
- **Functional code is enhanced for two reasons:**
  - To satisfy additional requirements
  - To improve the quality and maintainability of the code, a process known as **refactoring**.
- **Passing the suite of tests ensures that refactoring has not caused regression.**

Functional Tests

- **The best known type of tests is *functional tests*, which verify that functional requirements of the end system are satisfied.**
  - Such tests are also called **customer tests** or **acceptance tests**.
  - They are customer-facing tests.
- **Functional tests are run against the actual user interface of the running system.**
- **Functional tests may either be run manually by human testers, or they may be automated.**
- **Typical automation is to capture keystrokes and mouse movements, which can then be replayed.**
- **Various commercial test automation tools exist.**

Unit Tests

- ***Unit tests*** **are tests of specific program components.**
  - They are programmer-facing and are also called **programmer tests**.
- **Because there is no specific user interface for program components, testing requires some kind of test harness.**
  - This test harness must either be written specifically for the program, or a general purpose test harness may be used.
- **Besides the test harness, specific test cases must be written.**
- **Because these tests are programmer-facing, it is desirable if the tests can be specified in a familiar programming language.**
  - It is especially desirable if the test cases can be written in the same programming language as the functional code.
- **In this course we will write both functional code and test code in C#.**

Test Automation

- **A key success factor in using TDD is a system for test automation.**
- **Tests must be run frequently after each incremental change to the program, and the only way this is feasible is for the tests to be automated.**
- **There are many commercial and open source test automation tools available.**
- **A particular effective family of test automation tools are the unit test frameworks patterned after the original JUnit for Java:**

JUnit	Java
NUnit Visual Studio Unit Testing Framework	.NET
cppUnit	C++
PHPUnit	PHP
PyUnit	Python
Test::Unit	Ruby
JsUnit	JavaScript

Rules for TDD

- **Kent Beck, the father of eXtreme Programming (XP), suggested two cardinal rules for TDD:**
  - Never write any code for which you do not have a failing automated test.
  - Avoid all duplicate code.
- **The first rule ensures that you do not write code that is not tested.**
  - And if you provide tests for all your requirements, the rule ensures that you do not write code for something which is not a requirement.
- **The second rule is a cardinal principle of good software design.**
  - Duplicate code leads to inconsistent behavior over a period of time, as code is changed in one place but not in a duplicated place.

Implications of TDD

- **TDD has implications for the development process.**
  - You design in an organic manner, and the running code provides feedback for your design decisions.
  - As a programmer you write your own tests, because you can't wait for someone in another group to write frequent small tests for you.
  - You need rapid response from your development environment, in particular a fast compiler and a regression test suite.
  - Your design should satisfy the classical desiderata of highly cohesive and loosely-coupled components in order to make testing easier. Such a design is also easier to maintain.

Simple Design

- **Your program should both do *no less* and *no more* than the requirements demand.**
  - No less, because otherwise the program will not meet the functional requirements.
  - No more, because extra code imposes both a development and a maintenance burden.
- **You may find the following guidelines[1] useful:**
  - Your code is appropriate for its intended audience.
  - Your code passes all its tests.
  - Your code communicates everything it needs to.
  - Your code has the minimum number of classes that it needs.
  - Your code has the minimum number of methods that it needs.

[1] *Test-Driven Development in Microsoft .NET* by James V. Newkirk and Alexei A. Vorontsov.

Refactoring

- **The traditional waterfall approach to software development puts a great deal of emphasis on upfront design.**
  - Sound design is important in any effective methodology, but the agile approach emphasizes being responsive to change.
- **The *no more* principle suggests that you do not make your program more general than dictated by its current requirements.**
  - Future requirements may or may not come along the lines you anticipate.
- **The pitfall of incremental changes is that, if not skillfully done, the structure of the program may gradually fall apart.**
- **The remedy is to not only make functional changes, but when appropriate to *refactor* your program.**
  - This means to improve the program without changing its functionality.

Regression Testing

- **A pitfall of refactoring is that you may break something.**
  - A natural inclination is to follow the adage, “if it’s not broken, don’t fix it.”
- **But as we said, incremental changes to a program may lead to a deterioration of the program’s quality.**
- **So do go ahead and make refactoring improvements to your program, but be sure to test thoroughly after each change.**
- **Run the complete test suite to ensure that there has been no regression.**
- **As part of program maintenance, whenever you fix a bug, add a test to the test suite to test for this bug.**
  - Thus your test suite becomes gradually more and more robust, and you can have increased confidence that indeed your refactoring improvements will not break anything.

Test List

- **TDD begins with a *test list*.**
  - A test list is simply a list of tests for a program component or feature, expressed in ordinary English.
- **The test list describes the program component's requirements unambiguously.**
- **The test list provides a precise definition of the completion criteria.**
  - The requirements are met when all the tests in the test list pass.

Red/Green/Refactor

- **You implement the tests in the test list by a process that is sometimes called *Red/Green/Refactor*.**
  - You work in small, verifiable steps that provide immediate feedback[2].

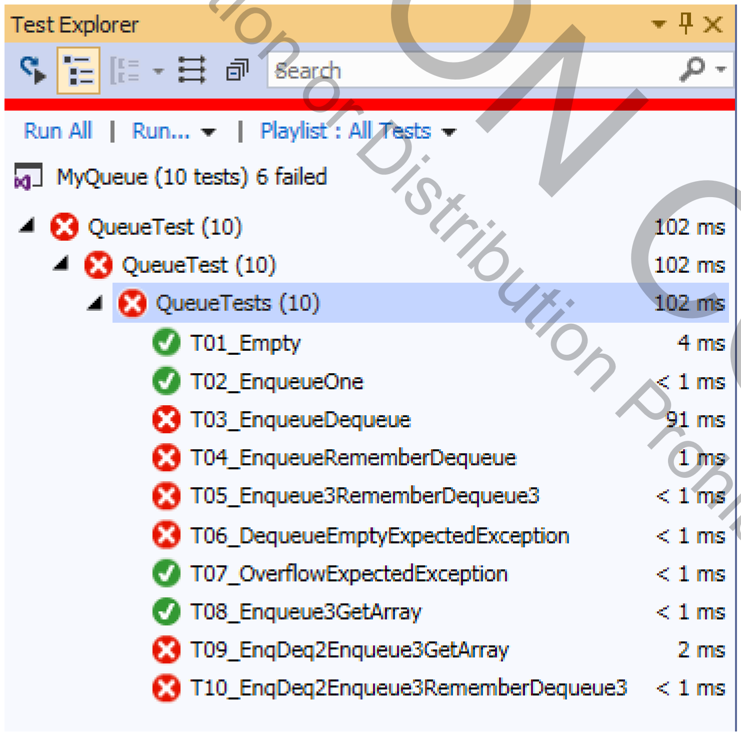
1. Write the test code.
2. Compile the test code. It should fail, because there is not yet any corresponding functional code.
3. Implement enough functional code for the test code to compile.
4. Run the test and see it fail (red).
5. Implement enough functional code for the test code to pass.
6. Run the test and see it pass (green).
7. Refactor for clarity and to eliminate duplication.
8. Repeat from the top.

- **Working in small steps enables you to immediately detect mistakes, and to see where the mistake occurred.**
  - You will rarely need the debugger!

[2] William Wake, *Extreme Programming Explored*.

Using the Unit Testing Framework

- **The Unit Testing Framework in Visual Studio provides an automated unit test facility for .NET languages such as C#.**
  - The framework comes with Visual Studio, including the free Visual Studio Community 2019.
- **It uses red (X) and green (check mark) to indicate failing and passing tests.**
  - The example shows the results of running a test suite for a Queue component, with Dequeue method not implemented.
  - The example is in **Chap01\MyQueue\NoDequeue**[3].

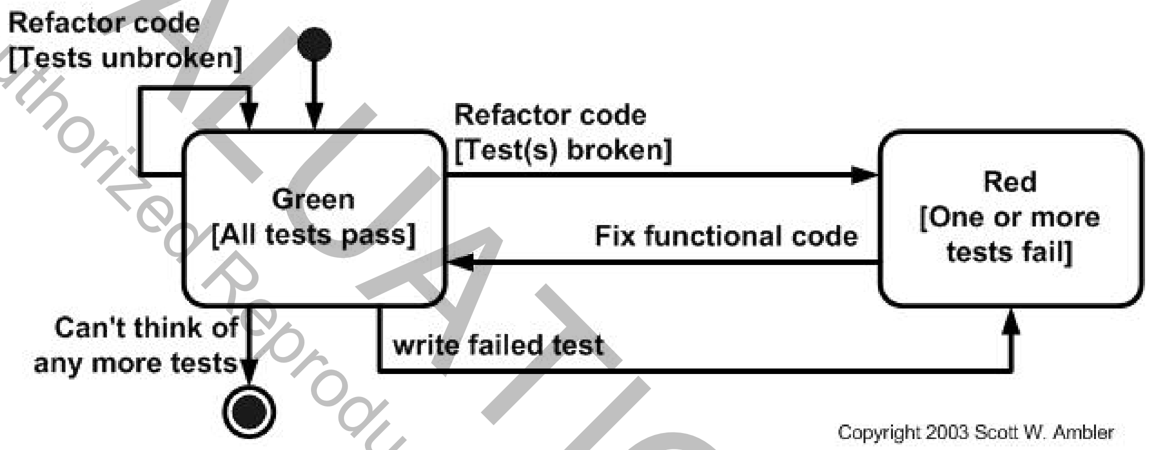


```
MyQueue (10 tests) 6 failed
◢ ✗ QueueTest (10)                                102 ms
  ◢ ✗ QueueTest (10)                              102 ms
    ◢ ✗ QueueTests (10)                           102 ms
        ✓ T01_Empty                                 4 ms
        ✓ T02_EnqueueOne                          < 1 ms
        ✗ T03_EnqueueDequeue                       91 ms
        ✗ T04_EnqueueRememberDequeue                1 ms
        ✗ T05_Enqueue3RememberDequeue3            < 1 ms
        ✗ T06_DequeueEmptyExpectedException       < 1 ms
        ✓ T07_OverflowExpectedException           < 1 ms
        ✓ T08_Enqueue3GetArray                    < 1 ms
        ✗ T09_EnqDeq2Enqueue3GetArray               2 ms
        ✗ T10_EnqDeq2Enqueue3RememberDequeue3     < 1 ms
```

[3] Open the Test Explorer window from the menu Test | Windows | Test Explorer. Run all the tests.

Testing with Unit Testing Framework

- **The diagram[4] illustrates how programmers doing TDD typically work using the Visual Studio Unit Testing Framework.**



1. Write a test case that will fail because functional code is not yet implemented (test first).
2. Run, and you will get red.
3. Fix the functional code and run again until you get green.
4. Keep writing test cases that will fail, implement the functional code, and get green.
5. At any point you may refactor for code improvements, and you need to make sure that you still get green.
6. When you can't think of any more tests, you are done!

[4] This diagram is reproduced by permission of the author, Scott Ambler. See http://www.agiledata.org/essays/tdd.html.

Unit Testing Framework Test Drive

- **Let's illustrate TDD by a simple example.**
  - Don't worry about the details of using the Unit Testing Framework but focus on the conceptual process of TDD.
- **Our program component is a FIFO (first-in, first-out) queue.**
  - The **Count** property returns number of elements in queue.
  - New items are inserted at the rear of the queue by the **Enqueue()** method.
  - Items are removed from the front of the queue by the **Dequeue()** method.
  - A method **ToArray()** returns all the items in the queue, with the front item at index 0.
- **We'll go through the following steps:**

1. Specify a .NET interface and provide a class with a stub implementation of the interface.
2. Create our test list, which is the specification of requirements.
3. Implement our first test and see it fail.
4. Implement the test code required to make the first test pass.
5. Implement the second test and see it fail.
6. Implement the test code to make the second test pass.
7. Repeat until all the tests pass.

IQueue Interface and Stub Class

– See the **QueueLib** class library project in the solution **Demos\MyQueue**, backed up in **Chap01\MyQueue\Step0**.

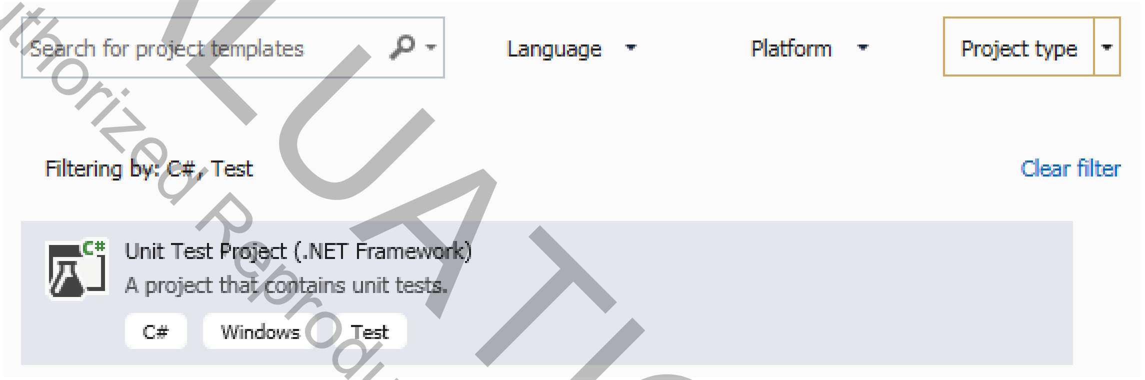
```
namespace QueueLib
{
   interface IQueue
   {
      int Count { get;}
      void Enqueue(int x);
      int Dequeue();
      int[] ToArray();
   }
   public class MyQueue : IQueue
   {
      public MyQueue(int size)
      {
      }
      public int Count
      {
         get
         {
            return -1;
         }
      }
      public void Enqueue(int x)
      {
      }
      public int Dequeue()
      {
         return 0;
      }
      public int[] ToArray()
      {
         return null;
      }
   }
}
```

Test List for Queue

1. Create a queue of capacity 3 and verify Count is 0. (All subsequent tests will also create a queue of capacity 3.)
2. Enqueue a number and verify that Count is 1.
3. Enqueue a number, dequeue it, and verify that Count is 0.
4. Enqueue a number, remember it, dequeue a number and verify that the two numbers are equal.
5. Enqueue three numbers, remember them, dequeue them, and verify that they are correct.
6. Dequeue an empty queue and verify you get an underflow exception.
7. Enqueue four numbers and verify you get an overflow exception.
8. Enqueue three numbers, get an array of numbers in queue and verify it is correct.
9. Enqueue two numbers, dequeue them. Enqueue three numbers, get an array of numbers in queue and verify it is correct.
10. Enqueue two numbers, dequeue them. Enqueue three numbers, remember them, dequeue them, and verify that they are correct.

Demo: Testing QueueLib

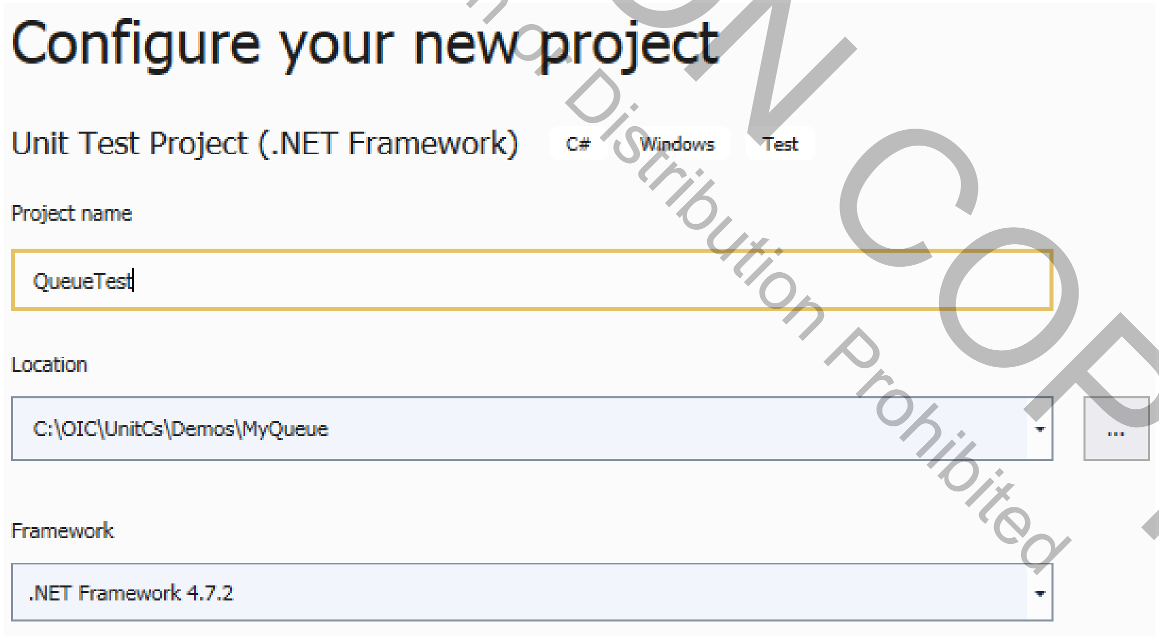
1. Open the **MyQueue** solution in **Demos\MyQueue**. Build the solution, which at this point consists only of a class library.
2. Find the template for Unit Test Project (.NET Framework). Click Next.



Filtering by: C#, Test

Unit Test Project (.NET Framework)
A project that contains unit tests.

3. Assign name **QueueTest** to the new project. Click Create.


Configure your new project

Unit Test Project (.NET Framework) C# Windows Test

Project name: QueueTest

Location: C:\OIC\UnitCs\Demos\MyQueue

Framework: .NET Framework 4.7.2

4. Change the name of the file **UnitTest1.cs** in the new project to **QueueTests.cs**.

Demo: Testing QueueLib (Cont'd)

5. Edit the supplied stub test method.

```
[TestMethod]
public void T01_Empty()
{
   MyQueue que = new MyQueue(3);
   Assert.AreEqual(0, que.Count);
}
```

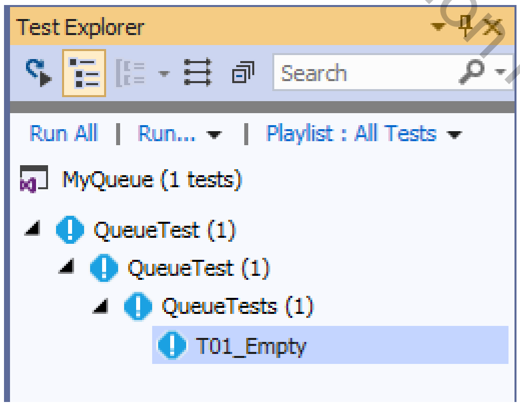
6. Build the solution. You will get a compile error, because the **MyQueue** class cannot be found by the test project.

7. In the **QueueTest** project add a reference to the **QueueLib** project.

8. In **QueueTests.cs** add a **using** statement to import the **QueueLib** namespace.

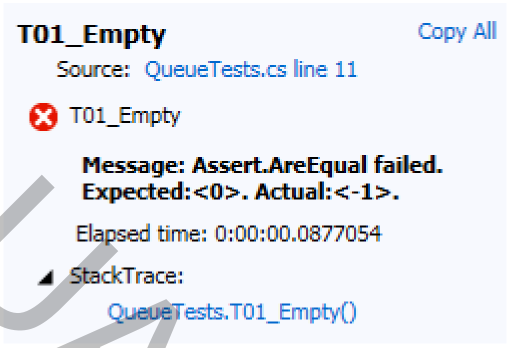
```
using QueueLib;
```

9. Build the solution. Bring up the Test Explorer window from the menu Test | Windows | Test Explorer. The blue icon shows that the test has not been run.



Demo: Testing QueueLib (Cont'd)

10. Click Run All. The test fails! Select the failed test in Text Explorer, and you will see details at the bottom of the window.



```
T01_Empty                                   Copy All
   Source:  QueueTests.cs line 11

   T01_Empty

      Message: Assert.AreEqual failed.
      Expected:<0>. Actual:<-1>.

      Elapsed time: 0:00:00.0877054

   StackTrace:
      QueueTests.T01_Empty()
```

11. The failure was expected, because we only have stub code for the implementation of the Queue.

12. Add code to **MyQueue.cs** to implement the **Count** property.

```
public class MyQueue : IQueue
{
   private int count;
   public MyQueue(int size)
   {
      count = 0;
   }
   public int Count
   {
      get
      {
         return count;
      }
   }
   ...
```

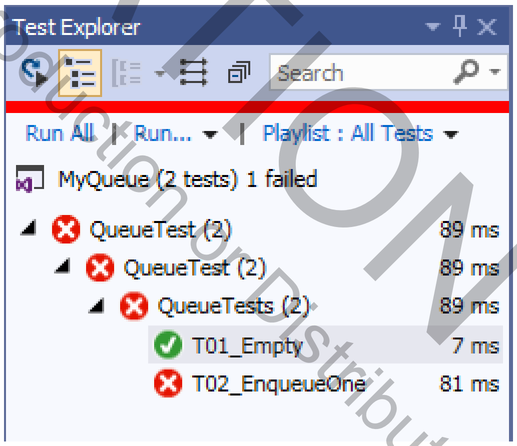
13. Rebuild the solution and run the test again. Now the test passes, showing green.

A Second Test

14. Add a second test to **QueueTests.cs**.

```
[TestMethod]
public void T02_EnqueueOne()
{
   MyQueue que = new MyQueue(3);
   que.Enqueue(17);
   Assert.AreEqual(1, que.Count);
}
```

15. Build the solution. Run all the tests. The first test passes (green), but the second test fails.



Test Explorer

Run All | Run... | Playlist : All Tests

MyQueue (2 tests) 1 failed

```
◢ ✗ QueueTest (2)              89 ms
  ◢ ✗ QueueTest (2)            89 ms
    ◢ ✗ QueueTests (2)         89 ms
        ✓ T01_Empty             7 ms
        ✗ T02_EnqueueOne       81 ms
```

More Queue Functionality

16. Add the following code to your **MyQueue** class.

```
public class MyQueue : IQueue
{
   private int count;
   private int[] data;
   private int front;
   private int rear;
   public MyQueue(int size)
   {
      count = 0;
      data = new int[size];
      front = 0;
      rear = -1;
   }
   public int Count
   {
      get
      {
         return count;
      }
   }
   public void Enqueue(int x)
   {
      rear += 1;
      data[rear] = x;
      count += 1;
   }
   ...
```

17. Run the tests again. Now both tests will pass (Step 1).

18. You could continue adding tests and functionality until the Queue is fully implemented and tested. We'll do that later. At this point we just want you to have a general idea of how unit testing in Visual Studio works.

TDD with Legacy Code

- **Our Queue example illustrates test-driven development with a brand new project, with tests developed before the code.**
- **But often, you may have existing legacy code and may wish to start employing TDD going forward.**
  - In this case you have a fully operational system, and you will begin by writing a test suite for the existing system.
  - Then as new features are to be added, you will first add appropriate tests to the test suite.
  - As bugs are discovered, you will also add test cases to the test suite to reproduce the failure.
  - As code is refactored, you will run the entire test suite to ensure that there is no regression.

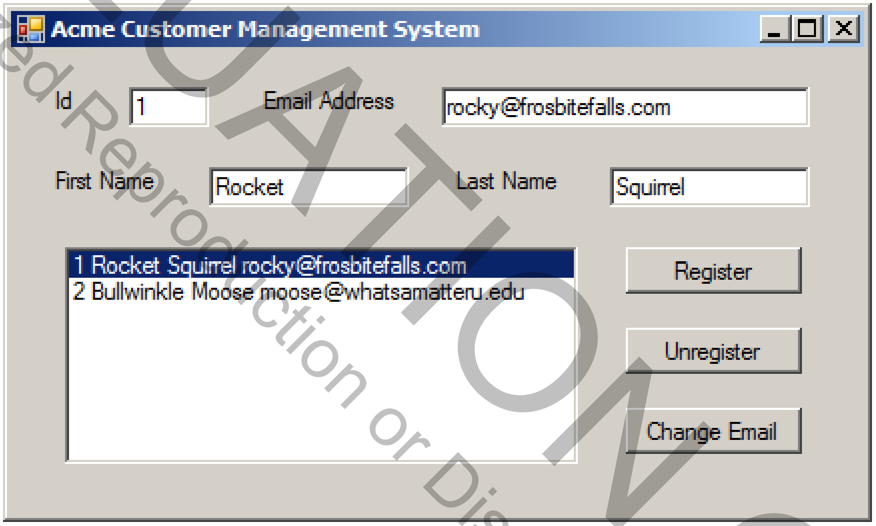
Acme Travel Agency Case Study

- **The Acme Travel Agency has a simple customer management system to keep track of customers who register for its services.**
- **Customers supply their first and last name and email address. The system supplies a customer ID.**
- **The following features are supported:**
  - Register a customer, returning a customer id.
  - Unregister a customer.
  - Obtain customer information, either for a single customer or for all customers (pass the customer id, and for customer id of –1 return all customers).
  - Change customer's email address.

```
public interface ICustomer
{
   int RegisterCustomer(string firstName,
      string lastName, string emailAddress);
   void UnregisterCustomer(int id);
   Customer[] GetCustomer(int id);
   void ChangeEmailAddress(int id,
      string emailAddress);
}
```

Acme Example Program

- **The Acme Customer Management System comes as a solution with two projects.**
  - See **CaseStudy\Acme\Step0**.
  - The solution contains a class library project **AcmeLib** and a Windows Forms client program **AcmeClient**.


Acme Customer Management System
Id
1
Email Address
rocky@frosbitefalls.com
First Name
Rocket
Last Name
Squirrel
1 Rocket Squirrel rocky@frosbitefalls.com
2 Bullwinkle Moose moose@whatsamatteru.edu
Register
Unregister
Change Email

- **To create unit tests, we will add a third project, *AcmeTest*, so as not to perturb the released class library, *AcmeLib*.**
  - See **CaseStudy\Acme\Step1**.

Lab 1

Testing the Customer Class

In this lab, you will begin the Acme Travel Agency case study by implementing simple tests for the **Customer** class. You are provided with starter code that provides implementation of classes **Customer** and **Customers** in a class library. You are also provided with a GUI client program. Your job is to create a third project for testing the **Customer** class with the Unit Testing Framework and to provide simple tests. You will exercise your tests using Visual Studio.

Detailed instructions are contained in the Lab 1 write-up at the end of the chapter.

Suggested time: 45 minutes

Summary

- ***Test-driven development* (TDD) calls for writing test cases *before* functional code.**
- **The test cases embody the requirements that the code must satisfy.**
- **There are two main types of tests pertaining to TDD:**
  - Functional tests, also known as customer tests
  - Unit tests, also known as programmer tests
- **Test automation is essential in TDD because many tests have to be frequently run.**
- **Simple design dictates that your program should both do *no less* and *no more* than the requirements demand.**
- **Refactoring provides continuous improvements in a software system, and automated tests ensure that no regression occurs.**
- **The Unit Testing Framework in Visual Studio 2019 simplifies writing and running tests in a .NET environment.**
- **TDD can drive a new project from start to finish, and it can also be used with legacy projects.**

Lab 1

Testing the Customer Class

Introduction

In this lab, you will begin the Acme Travel Agency case study by implementing simple tests for the **Customer** class. You are provided with starter code that provides implementation of classes **Customer** and **Customers** in a class library. You are also provided with a GUI client program. Your job is to create a third project for testing the **Customer** class with the Unit Testing Framework and to provide simple tests. You will exercise your tests using Visual Studio.

Suggested Time: 45 minutes

Root Directory: **OIC\UnitCs**

Directories:	**Labs\Lab1**	(Do your work here)
	CaseStudy\Acme\Step0	(Backup of starter files)
	CaseStudy\Acme\Step1	(Answer)

Instructions

1. Build the starter solution. This will build a class library **AcmeLib.dll** and a client Windows program **AcmeClient.exe**.

2. Exercise the client program by registering and unregistering a few customers, and changing the email addresses of some customers. Note that the ID of a new customer is automatically generated.

3. Examine the code of the class library project. Make sure you understand both the **Customer** class and the **Customers** class. Note the simple array implementation in this version of **Customers**.

4. Add a new Test Project **AcmeTest** to your solution. You can do this by right-clicking over the solution in Solution Explorer and choosing Add | New Project from the context menu. The project should have template Unit Test Project in the Test project type.

5. Examine the generated file **UnitTest1.cs** in the test project. Change the name of the supplied test method to **OneCustomer()**. This method should instantiate a customer object and make assertions that the three fields of the new object have the proper data.

```
[TestMethod]
public void OneCustomer()
{
   Customer cust = new Customer("Joe", "Blow", "foo@bar.com");
```

```
    Assert.AreEqual("Joe", cust.FirstName);
    Assert.AreEqual("Blow", cust.LastName);
    Assert.AreEqual("foo@bar.com", cust.EmailAddress);
}
```

6. Build the solution. The build fails, because the test project cannot find the Customer class.

7. In the **AcmeTest** project, add a project references to **AcmeLib** project. Now the build should be successful.

8. You should see the new test show up in Test Explorer with blue icon showing it has not run.

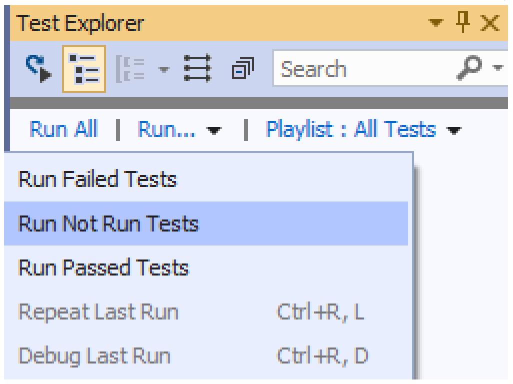
9. Click Run All. The test should succeed!

10. Examine the documentation of the **Assert** class, and notice the various overloads of the static **AreEqual()** method. Provide a second test method **OneCustomerMessage()** that performs the same test as the first test method but in addition displays a designated error message, which you can use to specify which field failed. In the assertion for the first name, test against the first name with all upper case characters.

```
[TestMethod]
public void OneCustomerMessage()
{
    Customer cust = new Customer("Joe", "Blow", "foo@bar.com");
    Assert.AreEqual("JOE", cust.FirstName, "FirstName not equal");
    Assert.AreEqual("Blow", cust.LastName, "LastName not equal");
    Assert.AreEqual("foo@bar.com", cust.EmailAddress,
       "EmailAddress not equal");
}
```

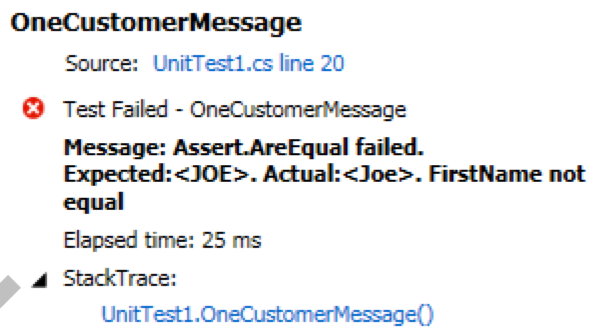
11. Build the solution. You will now see the second test in Test Explorer.

12. You could select Run All Tests again. But it will be more efficient to just Run Not Run Tests, because nothing has changed to make the Passed Test fail this time.



13. The new test fails.

14. Click on the failed test to see the details at the bottom of the window. You will see the error message that you supplied.



OneCustomerMessage
Source: UnitTest1.cs line 20
Test Failed - OneCustomerMessage
Message: Assert.AreEqual failed. Expected:<JOE>. Actual:<Joe>. FirstName not equal
Elapsed time: 25 ms
StackTrace:
UnitTest1.OneCustomerMessage()

15. Provide a third variation of the one customer test in which you specify that case should be ignored when doing a comparison.

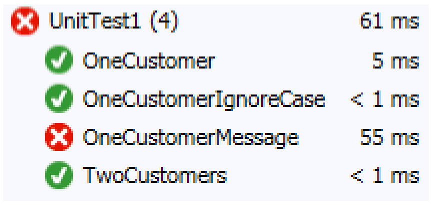
```
[TestMethod]
public void OneCustomerIgnoreCase()
{
   Customer cust = new Customer("Joe", "Blow", "foo@bar.com");
   Assert.AreEqual("JOE", cust.FirstName, true);
   Assert.AreEqual("Blow", cust.LastName, true);
   Assert.AreEqual("foo@bar.com", cust.EmailAddress, true);
}
```

16. Build the solution and run all the tests. Now the first and third tests should succeed while the second test continues to fail.

17. Provide a fourth test that will create two customers and verify that the generated CustomerIds are not equal.

```
[TestMethod]
public void TwoCustomers()
{
   Customer cust1 = new Customer("Joe", "Blow", "foo@bar.com");
   Customer cust2 = new Customer("Amy", "Smith", "amy@foo.com");
   Assert.AreNotEqual(cust1.CustomerId, cust2.CustomerId);
}
```

18. Run all the tests. All tests should pass except one. You are now at Step 1.



```
UnitTest1 (4)              61 ms
   OneCustomer              5 ms
   OneCustomerIgnoreCase  < 1 ms
   OneCustomerMessage      55 ms
   TwoCustomers           < 1 ms
```

19. If you have time, provide some tests for the **Customers** class.

Chapter 2

Visual Studio Unit Testing Fundamentals

Visual Studio Unit Testing Fundamentals

Objectives

After completing this unit you will be able to:

- **Describe the general structure of unit tests.**
- **Outline the features of the Unit Testing Framework in Visual Studio that support TDD.**
  - Assertions
  - Test Cases
  - Test Classes
  - Test Runners
- **Provide for initialization and cleanup of tests.**
- **Use Visual Studio as an integrated environment for creating and running unit tests.**

Structure of Unit Tests

- **As we saw in Chapter 1, *unit tests* are tests of specific program components.**
- **Test code is for internal use only and is separate from the production code being tested.**
- **Test code is responsible for doing several things:**

1. Set up resources needed for the test.
2. Call the function or method to be tested.
3. Verify that the called function behaved as expected.
4. Clean up after itself.
   - For simple tests, no special setup or cleanup may be required, but steps 2 and 3 are always performed.

Assertions

- **A central requirement of unit tests is that they must be self-verifying.**
  - It would be very inefficient to require a separate process or human intervention to examine the output of the tests to determine whether or not they passed.
- **We need a mechanism to support self-verification.**
- **This mechanism is called an *assertion*.**
  - An assertion is a statement that some condition is true, and a report will be made if the condition is not true.
- **The notion of assertion is common in many programming languages and frameworks.**
- **The ANSI C runtime library has an *assert()* method that can be used in the C language.**

Assert Example

- **A simple example illustrates the C *assert()* function.**
  - See **Chap02\CMax\Step1**[1].
  - This example provides unit tests for a **findmax()** function that finds the maximum of three integers.

```
// CMax.c

#include <stdio.h>
#include <assert.h>

int findmax(int x, int y, int z)
{
   int max = x;
   if (y > x)
      max = y;
   if (z > x)
      max = z;
   return max;
}

int main()
{
   assert(findmax(4, 3, 2) == 4);
   printf("Test 1 passed\n");
   assert(findmax(3, 4, 2) == 4);
   printf("Test 2 passed\n");
   assert(findmax(3, 2, 4) == 4);
   printf("Test 3 passed\n");
   assert(findmax(2, 4, 3) == 4);
   printf("Test 4 passed\n");
}
```

[1] You will need to install the Desktop development for C++ workload. Visual Studio provides a link to conveniently do the installation and then restart Visual Studio.

Assert Example (Cont'd)

- **The *findmax()* function has a bug. The first three tests pass, but the fourth one fails.**

```
Test 1 passed
Test 2 passed
Test 3 passed
Assertion failed: findmax(2, 4, 3) == 4, file
c:\oic\unitcs\chap02\cmax\step1\cmax.c, line 24
```

- **Step 2 fixes the bug.**

```
int findmax(int x, int y, int z)
{
   int max = x;
   if (y > max)
      max = y;
   if (z > max)
      max = z;
   return max;
}
```

  – Here is the output:

```
Test 1 passed
Test 2 passed
Test 3 passed
Test 4 passed
```

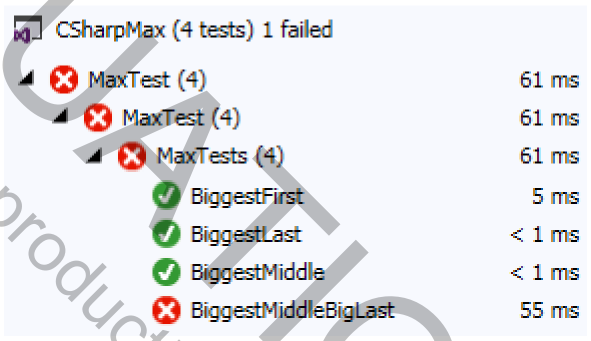
Unit Testing Framework

- **Although completely hand-written unit tests using only primitive library features are feasible, it is not efficient.**
- **Visual Studio's Unit Testing Framework that we introduced in Chapter 1 provides many features to simplify writing and running unit tests:**
  - An **Assert** class with a variety of methods for testing assertion conditions.
  - A custom attribute **[TestMethod]** to designate a method as a test case.
  - A custom attribute **[TestClass]** to designate a class that encapsulates a group of test methods sharing a common set of run-time resources. Such a class is sometimes called a "test fixture."
  - A test runner that automates the running of all the tests.
  - Some additional infrastructure that is transparent to the programmer but enables Visual Studio to seamlessly run the tests.

Lab 2A

Visual Studio Unit Tests of Maximum Method

In this lab, you will develop a C# version of the function to find the maximum of three integers. You will then review use of the Visual Studio Unit Testing Framework by developing and running test cases.



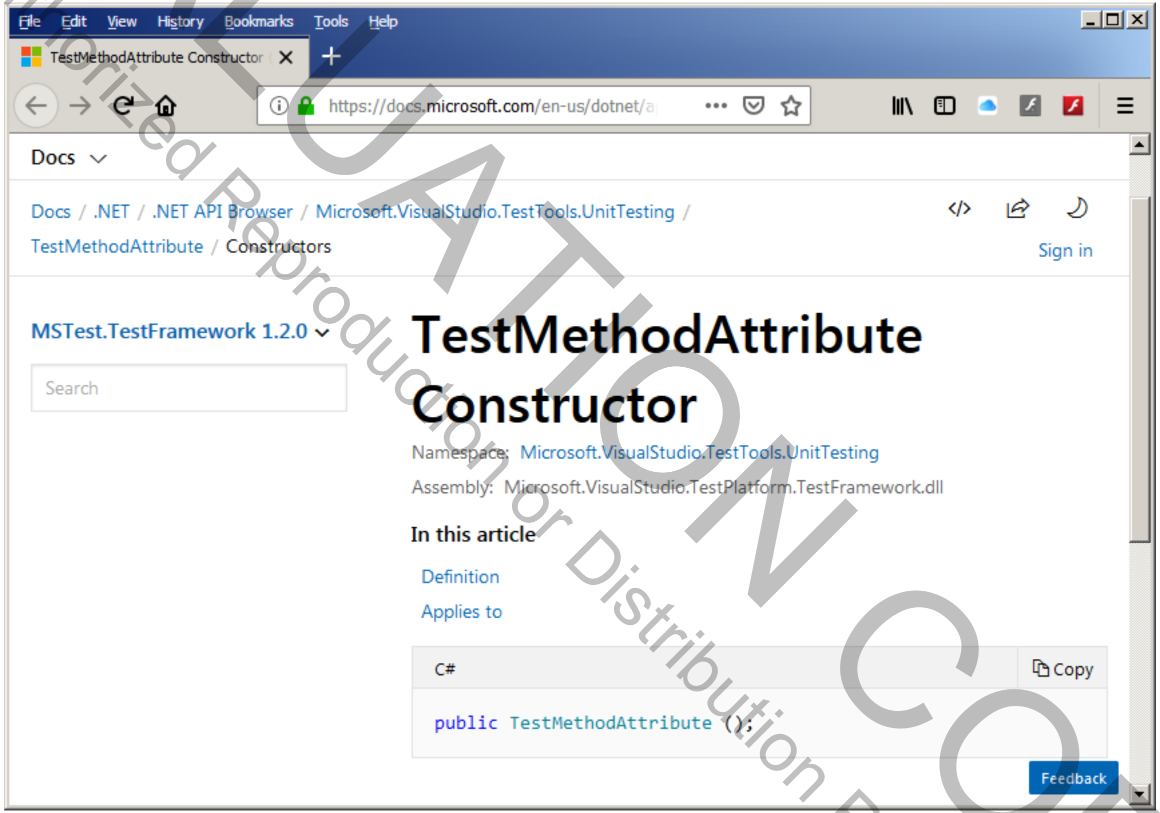
```
CSharpMax (4 tests) 1 failed
  MaxTest (4)                    61 ms
    MaxTest (4)                  61 ms
      MaxTests (4)               61 ms
        BiggestFirst              5 ms
        BiggestLast             < 1 ms
        BiggestMiddle           < 1 ms
        BiggestMiddleBigLast     55 ms
```

Detailed instructions are contained in the Lab 2A write-up at the end of the chapter.

Suggested time: 30 minutes

Unit Testing Framework Namespace

- **The classes supporting unit testing are in the *Microsoft.VisualStudio.TestTools.UnitTesting* namespace.**
  - MSDN provides documentation.


File Edit View History Bookmarks Tools Help
TestMethodAttribute Constructor
https://docs.microsoft.com/en-us/dotnet/a
Docs
Sign in
Search
Copy
Feedback

Docs / .NET / .NET API Browser / Microsoft.VisualStudio.TestTools.UnitTesting / TestMethodAttribute / Constructors

MSTest.TestFramework 1.2.0

TestMethodAttribute Constructor

Namespace: Microsoft.VisualStudio.TestTools.UnitTesting

Assembly: Microsoft.VisualStudio.TestPlatform.TestFramework.dll

In this article

Definition

Applies to

C#

```
public TestMethodAttribute ();
```

  - You can navigate to appropriate help pages from Visual Studio by using context-sensitive help: place the cursor over an appropriate keyword such as **TestMethod** and hit the F1 key.

Assert Class

- **The *Assert* class in this namespace has a number of static methods.**
  - These methods can help you determine whether the test passed or failed.
  - They record failures (when an assertion is false) and errors (when an unexpected exception occurs).
- **Failures and errors are reported through Visual Studio.**
  - You won’t see a system-generated exception message as in a failure of the **assert()** method of the C runtime library.
  - In Visual Studio you will see a red X displayed, plus explanatory text.
- **When a failure or error occurs, the current test method is aborted, and execution continues with the next method in the test class.**

Assert.AreEqual()

- **There are many overloaded versions of the *AreEqual()* method, such as:**

```
Assert.AreEqual(expected, actual)
Assert.AreEqual(expected, actual, message)
```

  - Typically **expected** is a hard coded value representing the value you expect to see.
  - **actual** is the value actually produced by the code that you are testing.
  - The optional parameter **message** is a string which will be displayed by Visual Studio upon failure.

- **Any object may be tested for equality.**
  - The **Equals()** method of the **Object** class will be used for comparison.
  - There are special overloads for the built-in data types of **int**, **uint**, **decimal**, **float**, and **double**.
  - For **float** and **decimal** there is an overload available that takes a parameter **delta** that may be used as a tolerance, specifying how close to equals the result should be.

```
Assert.AreEqual(expected, actual, delta)
```

- **In Visual Studio you can use Intellisense to see all the possible overloaded methods.**

More Assert Methods

- **There are many other Assert methods, which can be viewed via Intellisense.**
  - Many come in pairs with a **Not** variant, and there is always an optional string **message** parameter.

AreNotEqual	**expected** does not equal **actual**
IsNull IsNotNull	Given object is **null** (or is not **null**)
AreSame AreNotSame	**expected** and **actual** refer (or do not refer) to the same object
IsTrue	Given Boolean condition is true
IsFalse	Given Boolean condition is false
Fail	Fail the test immediately
IsInstanceOfType IsNotInstanceOfType	**actual** object is (or is not) of **expected** type

CollectionAssert Class

- **The *CollectionAssert* class can be used to compare collections of objects and to verify the state of one or more collections.**
  - As with **Assert**, many methods come in pairs with a **Not** variant, and there is always an optional string **message** parameter.

AreEqual AreNotEqual	Collections are (or are not) equal in having the same elements in the same order
AreEquivalent AreNotEquivalent	Collections are (or are not) equal in having the same elements in any order
AllItemsAreInstancesOfType	All items in a collection are instances of a particular type
AllItemsAreNotNull	All items in a collection are not **null**
AllItemsAreUnique	All items in a collection are unique
IsSubsetOf IsNotSubsetOf	One collection is (or is not) a subset of another collection
Contains DoesNotContain	Collection contains (or does not contain) a specified element

StringAssert Class

- **The *StringAssert* class can be used to compare strings.**

Matches	String matches a regular expression
DoesNotMatch	String does not match a regular expression
EndsWith	String ends with a specified substring
StartsWith	String starts with a specified substring
Contains	String contains a specified substring

Test Case

- **The fundamental unit of testing with the Visual Studio Testing Framework is a *test case*.**
  - A test case is a programmer test, which is a low-level test intended to verify behavior at the method or class level.
  - A test case is **self-validating**, having a built-in mechanism to report success or failure.
  - A test case can be automatically discovered by a test runner.
  - A test case can be automatically executed by a test runner.
  - A test case executes independently of other test cases; one test case should not produce any side effects that could change the results from other test cases.

Test Methods

- **In Visual Studio, test cases are specified by *test methods*, which are methods of a *test class*.**
- **A test method is marked by the *[TestMethod]* attribute.**

```
[TestMethod]
public void BiggestFirst()
{
   Assert.AreEqual(4, Find.Max(4, 3, 2));
}
```

- **A test method must have the following features:**
  - It is declared as **public**.
  - It is an instance method (not static).
  - It returns **void**.
  - It takes no parameters.

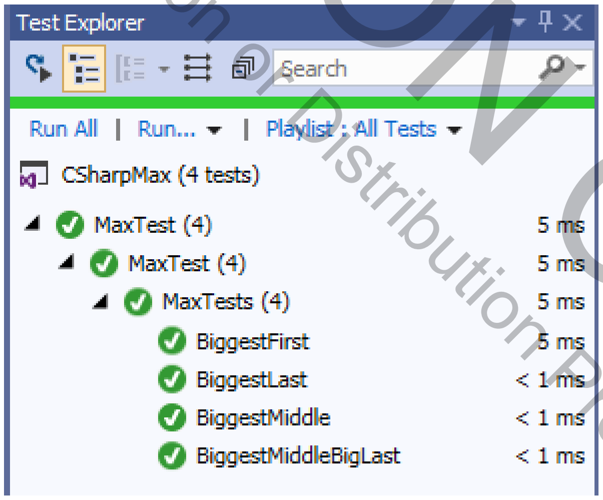
Test Class

- **In Visual Studio, test methods are encapsulated in a *test class*.**
  - In the similar NUnit unit test framework, a test class is called a test fixture.
- **The test methods in a test class share a common set of resources.**
- **The test class is marked with the *[TestClass]* attribute.**

```
[TestClass]
public class MaxTests
{
    ...
    [TestMethod]
    void BiggestFirst()
    {
        Assert.AreEqual(4, Find.Max(4, 3, 2);
    }
    ...
 }
```

Test Runner

- **An essential component of an effective unit testing system is a facility to automatically run the tests.**
- **A *test runner* is a program that automatically discovers test cases, runs them, and reports on the results.**
- **An example of a test runner is Visual Studio.**
- **A test runner may use .NET reflection to dynamically discover and execute test methods.**
- **The Visual Studio test runner shows test methods in a Test Explorer Window.**



```
CSharpMax (4 tests)
◢ MaxTest (4)                      5 ms
  ◢ MaxTest (4)                    5 ms
    ◢ MaxTests (4)                 5 ms
        BiggestFirst               5 ms
        BiggestLast              < 1 ms
        BiggestMiddle            < 1 ms
        BiggestMiddleBigLast     < 1 ms
```

Command Line Test Runner

- **You can also run tests at the command line using the tool *MSTest.exe*.**

```
mstest /testcontainer:<path to test DLL>
```

- **For an example, do the following:**

1. Build the **CSharpMax\Step3** example.
2. Run the Developer Command Prompt for Visual Studio 2019.
3. Navigate to the directory **C:\OIC\UnitCs\Chap02\CSharpMax\Step3**.
4. Enter the following command (which has also been provided in the batch file **RunMSTest.bat**):

```
mstest /testcontainer:MaxTest\bin\debug\maxtest.dll
```

5. You will obtain this output:

```
Loading MaxTest\bin\debug\maxtest.dll...
Starting execution...
Warning: The disabled test 'BiggestMiddle' was removed from
the test run.

Results               Top Level Tests
-------               ---------------
Passed                MaxTest.DivTests.SimpleDivide
Passed                MaxTest.MaxTests.BiggestFirst
Passed                MaxTest.MaxTests.BiggestLast
Passed                MaxTest.MaxTests.BiggestMiddleBigLast
4/4 test(s) Passed
```

Ignoring Tests

- **A test case can be made non-runnable by marking it with the *[Ignore]* attribute.**

```
[Ignore]
[TestMethod]
public void BiggestMiddle()
{
   Assert.AreEqual(4, Find.Max(3, 4, 2));
}
```

- **This feature may be useful during development to temporarily disable running certain tests which will be known to fail.**
  - The system being tested may have a known bug or have not yet implemented certain functionality.
- **It may also be useful to temporarily disable certain long-running tests that are known to succeed.**

Demo: Multiple Test Classes

- **Let's illustrate a scenario in which there are two test classes.**
  - We'll start with an extension of the **CSharpMax** program you worked with in the lab. We'll add a stub second method **Div()** but no test code yet.
  - In accord with test-driven development, we will provide the test before the implementation code.
  - See **Demos\CSharpMax**, which is backed up in the directory **Chap02\CSharpMax\Step2**.

1. Examine the code in **Find.cs**. In the **Find** class add a stub method **Div()**.

```
public static int Div(int x, int y)
{
   return 0;
}
```

2. To add a second test class, right-click the **MaxTest** project and choose Add | Unit Test from the context menu.
3. Change the file name of the new test class to **DivTests.cs**.

Demo: Multiple Test Classes (Cont'd)

4. Provide the following test method. Import **MaxLib**.

```
using MaxLib;
...
   [TestClass]
   public class DivTests
   {
      [TestMethod]
      public void SimpleDivide()
      {
         Assert.AreEqual(5, Find.Div(10, 2));
      }
   }
```

5. Run the tests. As expected, the SimpleDivide test fails.

6. Implement the **Div()** method.

```
public static int Div(int x, int y)
{
   return x / y;
}
```

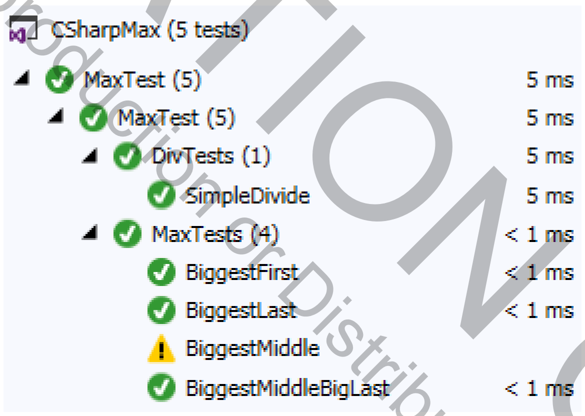
7. Now all the tests pass.

Using the Ignore Attribute

8. In **MaxTests.cs** try placing the **[Ignore]** attribute in front of the **BiggestMiddle()** test method.

```
[Ignore]
[TestMethod]
public void BiggestMiddle()
{
   Assert.AreEqual(4, Find.Max(3, 4, 2));
}
```

9. Run the tests again. Now the ignored test method is skipped and shown with a yellow icon.



```
CSharpMax (5 tests)
◢ ✓ MaxTest (5)                         5 ms
  ◢ ✓ MaxTest (5)                       5 ms
    ◢ ✓ DivTests (1)                    5 ms
        ✓ SimpleDivide                  5 ms
    ◢ ✓ MaxTests (4)                  < 1 ms
        ✓ BiggestFirst                < 1 ms
        ✓ BiggestLast                 < 1 ms
        ⚠ BiggestMiddle
        ✓ BiggestMiddleBigLast        < 1 ms
```

10. The program at this point is saved in **Chap02\CSharpMax\Step3**.

Test Initialization and Cleanup

- **The Visual Studio Unit Testing Framework provides attributes that you can use to set up and tear down tests.**
- **You can ensure that all tests are initialized in the same manner by means of the *[TestInitialize]* attribute.**
  - Place this attribute before a method, which will then be called prior to the execution of each test in the test class.
- **You can ensure that all tests are cleaned up in the same manner by means of the *[TestCleanup]* attribute.**
  - Place this attribute before a method, which will then be called immediately after the execution of each test in the test class.

Test Initialization Example

- **A version of the tests for our queue class illustrates test initialization and cleanup.**
  - See **Chap02\QueueInit**.

```
[TestClass]
public class QueueTests
{
   ...
   private MyQueue que;
   private static int SizeQueue;

   [TestInitialize]
   public void InitQueue()
   {
      que = new MyQueue(SizeQueue);
      Debug.WriteLine("Initialize queue of size " +
         SizeQueue);
   }
   [TestCleanup]
   public void CleanupQueue()
   {
      Debug.WriteLine("Cleanup queue");
   }
   ...
```

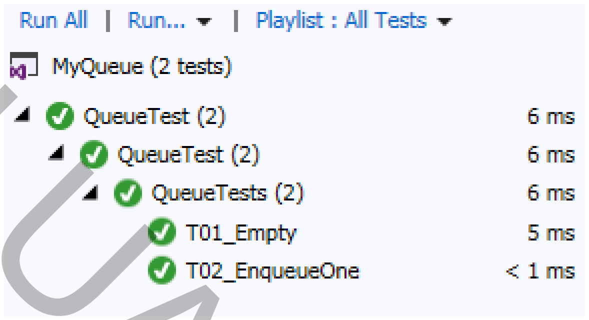
Class Initialization and Cleanup

- **The Visual Studio Unit Testing Framework also provides attributes that you can use to initialize and cleanup test classes.**
  - While test initialization and cleanup are done on a per-method basis, class initialization and cleanup are performed on a per-class basis.
  - A static method marked with the **[ClassInitialization]** attribute is called once for the entire test class, before any of the test cases are executed.
  - A static method marked with the **[ClassCleanup]** attribute is called once for the entire test class, after all of the test cases are executed.

```
[TestClass]
public class QueueTests
{
   ...
   [ClassInitialize]
   public static void InitTestClass(
      TestContext testContext)
   {
      SizeQueue = 3;
      Debug.WriteLine("Initialize test class");
   }
   [ClassCleanup]
   public static void CleanupTestClass()
   {
      Debug.WriteLine("Cleanup test class");
   }
   ...
```

Running Test Initialization Example

- **Here is the result of running these tests in Visual Studio.**



```
MyQueue (2 tests)
  QueueTest (2)                 6 ms
    QueueTest (2)               6 ms
      QueueTests (2)            6 ms
        T01_Empty               5 ms
        T02_EnqueueOne        < 1 ms
```

- **You can examine the debug output from the initialization and cleanup methods by running under the debugger and examining the Output window.**
  - From the Test menu select Debug. You can also right-click in the code window for the tests and select Debug Tests from the context menu.

```
Initialize test class
...
Initialize queue of size 3
Cleanup queue
Initialize queue of size 3
Cleanup queue
Cleanup test class
```

Lab 2B

Testing the Customers Class

In this lab, you will create a list of test cases for the **Customers** class. You will then implement these test cases using the Unit Testing Framework and run them using Visual Studio.

Detailed instructions are contained in the Lab 2B write-up at the end of the chapter.

Suggested time: 90 minutes

Summary

- **Unit tests have a basic structure to set up needed resources, call the test method, verify the result, and then clean up.**
- **The Unit Testing Framework provides these facilities:**
  - Assertions
  - Test Methods
  - Test Classes
  - Test Runners
- **You can mark methods with attributes to initialize and cleanup tests.**
- **Unit tests can be run both from within Visual Studio and at the command line.**

Lab 2A

Visual Studio Unit Tests of Maximum Method

Introduction

In this lab, you will develop a C# version of the function to find the maximum of three integers. You will then review use of the Visual Studio Unit Testing Framework by developing and running test cases..

Suggested Time: 30 minutes

Root Directory: **OIC\UnitCs**

Directories:	**Labs\Lab2A**	(Do your work here)
	Chap02\CSharpMax\Step1	(Answer to Part 1)
	Chap02\CSharpMax\Step2	(Answer to Part 2)

Part 1

1. Use Visual Studio to create a new empty solution **CSharpMax** in the working directory.

2. Right-click over the solution to add a new class library project **MaxLib** to the solution.

3. Change the name of the file **Class1.cs** to **Find.cs**. This will also change the name of the class to **Find**.

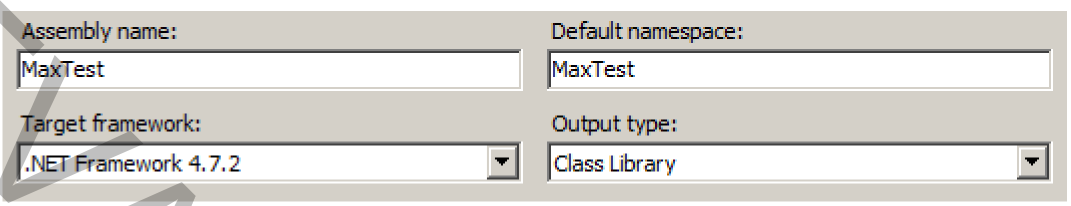
4. Add a public static method **Max()** to the **Find** class that attempts to find the maximum of three integers using this code:

```
public static int Max(int x, int y, int z)
{
   int max = x;
   if (y > x)
      max = y;
   if (z > x)
      max = z;
   return max;
}
```

5. Right-click over the solution to add a new Unit Test Project **MaxTest** to the solution. In this project add a reference to the **MaxLib** project.

6. In The standard output types of projects in Visual Studio are Console Application, Windows Application and Class Library. What is the output type of the new **MaxTest** project? How can you find out?

7. You can right-click over the project and choose Properties from the context menu. The Application tab will show the Assembly name, the Output type, and so on. The Output type is Class Library. Building the project will generate a DLL.



Assembly name: MaxTest
Default namespace: MaxTest
Target framework: .NET Framework 4.7.2
Output type: Class Library

8. Change the name of the file **UnitTest1.cs** to **MaxTests.cs**. This will also change the name of the test class to **MaxTests**.

9. Add a reference to the **MaxLib** project and provide a **using** statement to import the **MaxLib** namespace.

10. Examine the code for the **MaxTests** class. The key features are:

    a. The namespace **Microsoft.VisualStudio.TestTools.UnitTesting**.

    b. The attribute **[TestClass]** in front of the class.

    c. The attribute **[TestMethod]** in front of each test method.

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

namespace MaxTest
{
    [TestClass]
    public class MaxTests
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

11. Add a test method to the **MaxTests** class corresponding to the first assertion in the C language test program. Be sure to provide the **[TestMethod]** attribute in front of the test method.

```
 [TestClass]
 public class MaxTests
 {
     ...
     [TestMethod]
     void BiggestFirst()
     {
         Assert.AreEqual(4, Find.Max(4, 3, 2);
     }
```

```
   }
```

12. Examine Test Explorer[2]. You will see the stub test method **TestMethod1()** but not your own test method. Why?

13. The **BiggestFirst()** test method was not public. Make it public.

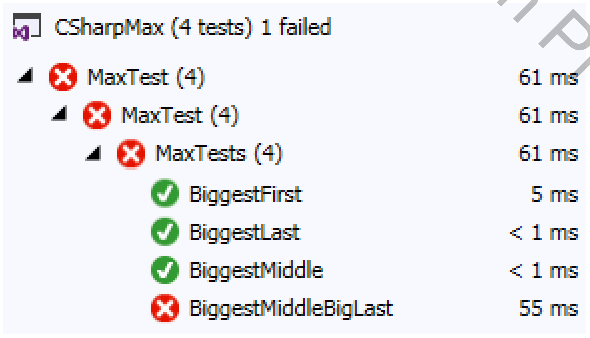
```
      [TestMethod]
      public void BiggestFirst()
      {
          Assert.AreEqual(4, Find.Max(4, 3, 2));
      }
```

14. Rebuild the solution. Now the new test shows up in Test Explorer in the Not Run Tests. Run all the tests. Both tests pass.

15. Implement the remaining test methods corresponding to the assertions in the C version of the program. Also, delete the stub **TestMethod1()** test method.

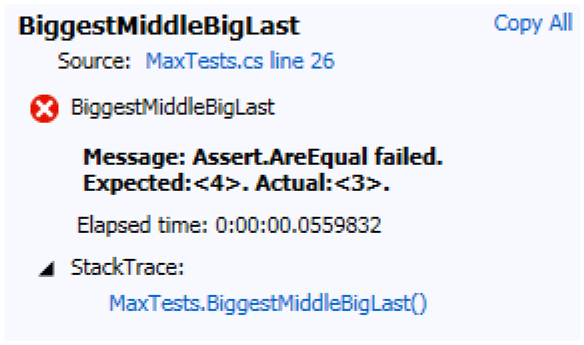
```
[TestMethod]
public void BiggestMiddle()
{
   Assert.AreEqual(4, Find.Max(3, 4, 2));
}
[TestMethod]
public void BiggestLast()
{
   Assert.AreEqual(4, Find.Max(3, 2, 4));
}
[TestMethod]
public void BiggestMiddleBigLast()
{
   Assert.AreEqual(4, Find.Max(2, 4, 3));
}
```

16. Rebuild the solution and run the tests. The first three tests pass, but the last test fails. The project is now at Step 1.



```
CSharpMax (4 tests) 1 failed
◢ ✗ MaxTest (4)                    61 ms
   ◢ ✗ MaxTest (4)                 61 ms
      ◢ ✗ MaxTests (4)             61 ms
           ✓ BiggestFirst            5 ms
           ✓ BiggestLast           < 1 ms
           ✓ BiggestMiddle         < 1 ms
           ✗ BiggestMiddleBigLast   55 ms
```

[2] If the Test Explorer window is closed, you can get it back by running tests from the menu Test | Run.


BiggestMiddleBigLast Copy All

Source: MaxTests.cs line 26

BiggestMiddleBigLast

Message: Assert.AreEqual failed. Expected:<4>. Actual:<3>.

Elapsed time: 0:00:00.0559832

StackTrace:

MaxTests.BiggestMiddleBigLast()

Part 2

1. Fix the bug in the **Find.Max()** method.

2. Rebuild the solution and run the tests. Now all the tests should pass. The project is now at Step 2.

Lab 2B

Testing the Customers Class

Introduction

In this lab, you will create a list of test cases for the **Customers** class. You will then implement these test cases using the Unit Testing Framework and run them using Visual Studio.

Suggested Time: 90 minutes

Root Directory: **OIC\UnitCs**

Directories:	**Labs\Lab2B\Acme**	(Do your work here)
	CaseStudy\Acme\Step1	(Backup of starter files)
	CaseStudy\Acme\Step2	(Answer)

Part 1. Create Test List

In this part you will review the Acme Customer Management System case study, and you will create a list of test cases for testing the **Customers** class.

1. If you prefer, replace the supplied starter code with your own code from Lab 1.

2. Build and run the starter project, and review the code.

3. Create a list of test cases that you feel will adequately test the **Customers** class.

4. When you are done, compare your list with the list on the next page.

Part 1 Answer. Test cases for the Customers class.

1. Register one customer, note id, obtain info for this customer and verify.
2. Register two customers, obtain info for all customers and verify.
3. Register one customer, change email, and verify.
4. Verify that customers list is initially empty (null).
5. Register one customer, unregister, and verify list is empty.
6. Register three customers, unregister first customer, and verify remaining two customers are what expected.
7. Register three customers, unregister last customer, and verify remaining two customers are what expected.
8. Register 100 customers and verify that count of registered customers is 100.
9. Register 100 customers, unregister all of them, and verify that customer list is empty.

Part 2. Implement Test Cases for the Customers Class

In this part you will write test methods for all of your test cases and run them using Visual Studio. You should at least implement the test cases shown above.

1. Build the solution and run it using Ctrl + F5. This will run the four tests for the **Customer** class. Verify that three of them succeed.
2. Delete the test methods **OneCustomerMessage()** and **OneCustomerIgnoreCase()**. These were only present to demonstrate additional **Assert.AreEqual()** overloads and have no bearing on testing the functionality of the **Customer** class.
3. Change the name of the file **UnitTest1.cs** to **CustomerTests.cs**. This will also change the name of the test class.
4. Build and run the tests. There should now be two tests, and both succeed.
5. Add a new test: right-click over the **AcmeTest** project and choose Add | New Test from the context menu. In the Add New Test dialog, select the Basic Unit Test template, and assign Test Name **CustomersTests.cs**. This will create a new test class **CustomersTests**. (In Visual Studio Community there is only one kind of test, and from the context menu you will choose Add | Unit Test.)
6. Provide code to do the following;

a. Import the **System.Diagnostics** namespace.

b. Provide a private data member **custs** of type **Customers**.

c. Provide a helper method **ShowCustomerArray(Customer[] arr)** to display an array of **Customer** objects at the debug Output window..

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

namespace AcmeTest
{
    [TestClass]
    public class CustomersTest
    {
        private Customers custs;

        private void ShowCustomerArray(Customer[] arr)
        {
            if (custs == null)
            {
                Debug.WriteLine("<null>");
                return;
            }
            for (int i = 0; i < arr.Length; i++)
            {
                string id = arr[i].CustomerId.ToString();
                string first = arr[i].FirstName.PadRight(12);
                string last = arr[i].LastName.PadRight(12);
                string email = arr[i].EmailAddress.PadRight(20);
                string str = id + "   " + first + "   " + last + "   "
                             + email;
                Debug.WriteLine(str);
            }
        }
```

7. Build the solution to make sure that you get a clean compile.

8. Run the tests. You should see the two **Customer** tests run successfully and there is the stub test **TestMethod1** in the **Customers** test class.

9. Implement the first test method to register one customer, note id, obtain info for this customer and verify. You may call it **OneCustomer()**. Note that it is perfectly legal for test methods under different test fixtures to have the same name. This will replace the stub test.

```
[TestMethod]
public void OneCustomer()  //1
{
   int id = custs.RegisterCustomer("Joe", "Blow", "foo@bar.com");
   Customer[] arr = custs.GetCustomer(id);
   ShowCustomerArray(arr);
   Assert.AreEqual(arr[0].FirstName, "Joe");
   Assert.AreEqual(arr[0].LastName, "Blow");
   Assert.AreEqual(arr[0].EmailAddress, "foo@bar.com");
}
```

10. Build and run all tests from Test Explorer.

11. The **OneCustomer** test **in CustomersTests.cs** fails, because the **custs** object has not been initialized.

12. Provide an initialization method to initialize the **Customers** object **custs**.

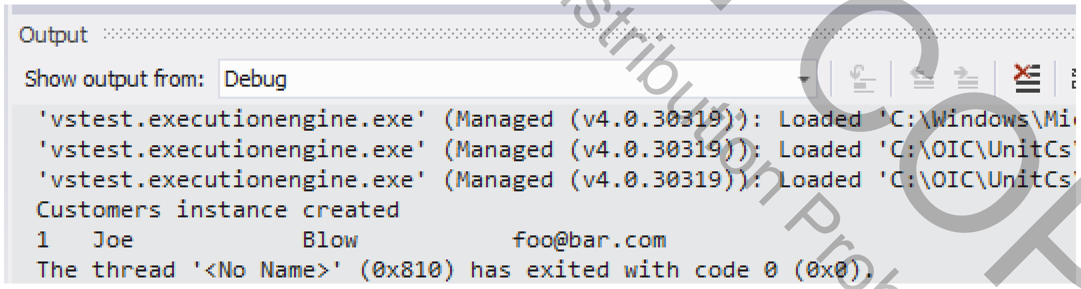
```
[TestInitialize]
public void InitCustomers()
{
   custs = new Customers();
   Debug.WriteLine("Customers instance created");
}
```

13. All tests should succeed.

14. Although it is legal to use the same name **OneCustomer** for the test of the **Customers** class, the output in Test Explorer is confusing. Let's adapt a naming convention in which we will put **Customers** at the beginning of the names of the test methods of the **Customers** class. The test method is now **CustomersOneCustomer**.

15. Debug the **Customers** tests. (Right-click over the **CustomersTests.cs** window and choose Debug Tests from the context menu.)

16. Look at the Output window to see the debug output that you've generated in your test code.



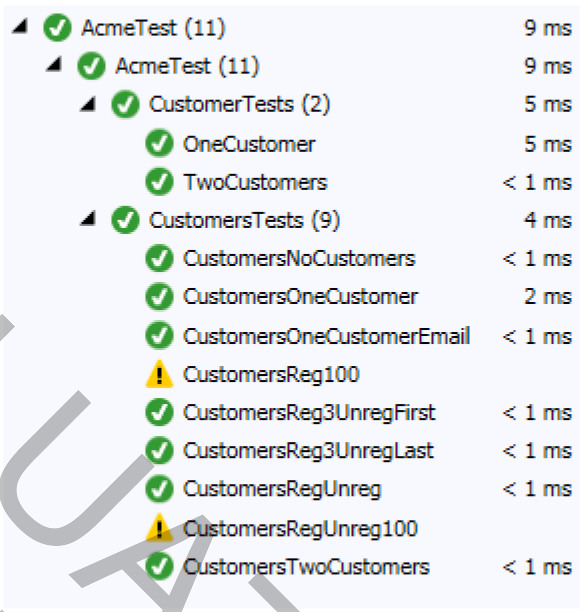
```
'vstest.executionengine.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Mi
'vstest.executionengine.exe' (Managed (v4.0.30319)): Loaded 'C:\OIC\UnitCs
'vstest.executionengine.exe' (Managed (v4.0.30319)): Loaded 'C:\OIC\UnitCs
Customers instance created
1   Joe          Blow         foo@bar.com
The thread '<No Name>' (0x810) has exited with code 0 (0x0).
```

17. Implement the remaining tests. Follow the naming convention by beginning the name of each test method with **Customers**. If you have questions about any of the test method implementations, don't hesitate to consult the supplied answer.

18. Run all your tests. They should all succeed except for the two tests in which you register 100 customers. (We'll fix our class in the next chapter!)

19. Since you expect those two tests to fail at this point, due to the implementation of the list of customers by an array of limited size, mark these two tests with the **[Ignore]** attribute.

20. Run the tests again. The two tests that previously failed now are skipped.



```
◢ ✓ AcmeTest (11)                          9 ms
  ◢ ✓ AcmeTest (11)                        9 ms
    ◢ ✓ CustomerTests (2)                  5 ms
        ✓ OneCustomer                      5 ms
        ✓ TwoCustomers                   < 1 ms
    ◢ ✓ CustomersTests (9)                 4 ms
        ✓ CustomersNoCustomers           < 1 ms
        ✓ CustomersOneCustomer             2 ms
        ✓ CustomersOneCustomerEmail      < 1 ms
        ⚠ CustomersReg100
        ✓ CustomersReg3UnregFirst        < 1 ms
        ✓ CustomersReg3UnregLast         < 1 ms
        ✓ CustomersRegUnreg              < 1 ms
        ⚠ CustomersRegUnreg100
        ✓ CustomersTwoCustomers          < 1 ms
```

itcourseware
TRAINING MATERIALS FOR IT PROFESSIONALS

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com

9-08-00401-000-06-18-19