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].

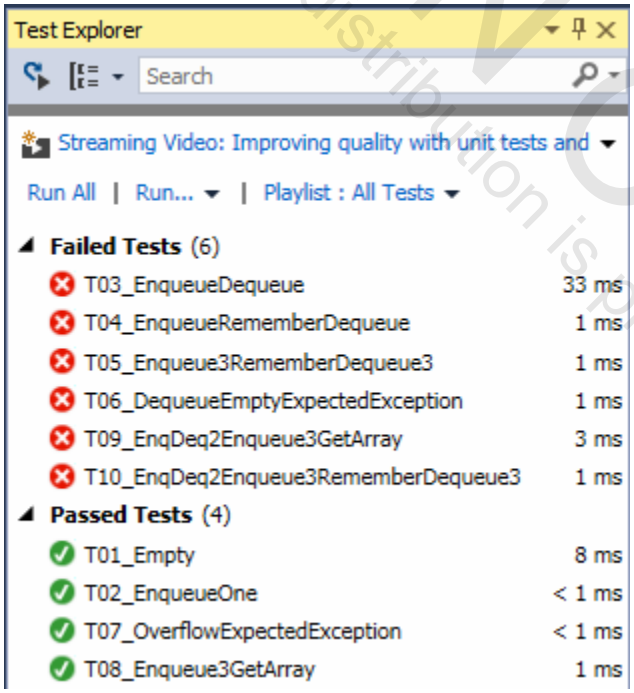
1. Write the test code.
2. Compile the test code. It should fail, because there is not yet any corresponding functional code.
3. Implement enough functional code for the test code to compile.
4. Run the test and see it fail (red).
5. Implement enough functional code for the test code to pass.
6. Run 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*.

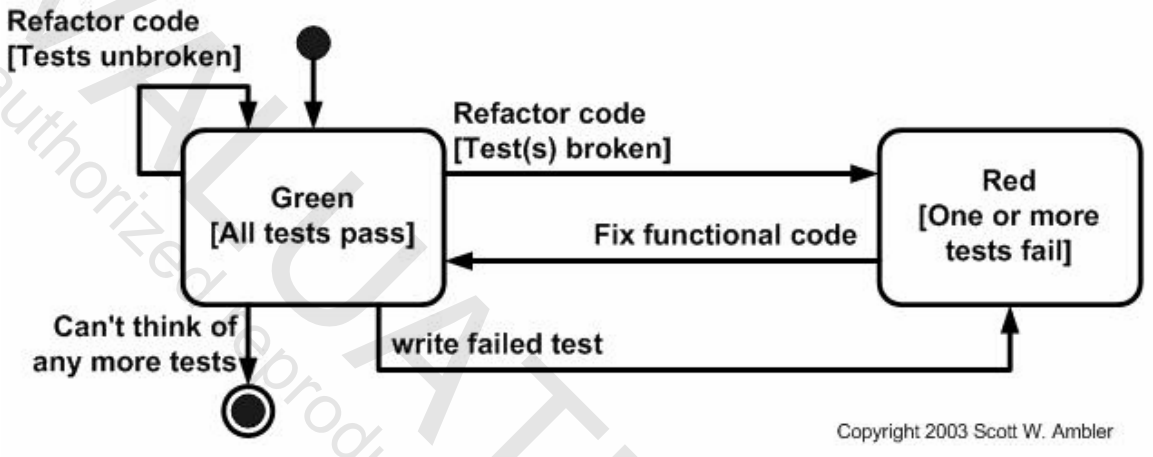
Using the Unit Testing Framework

- **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 2015.
  - Visual Studio 2012 introduced a pluggable architecture for test frameworks, enabling integration with third-party test frameworks such as NUnit.
- **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**.



Testing with Unit Testing Framework

- **The diagram[3] 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!

[3] 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;
      }
   }
}
```

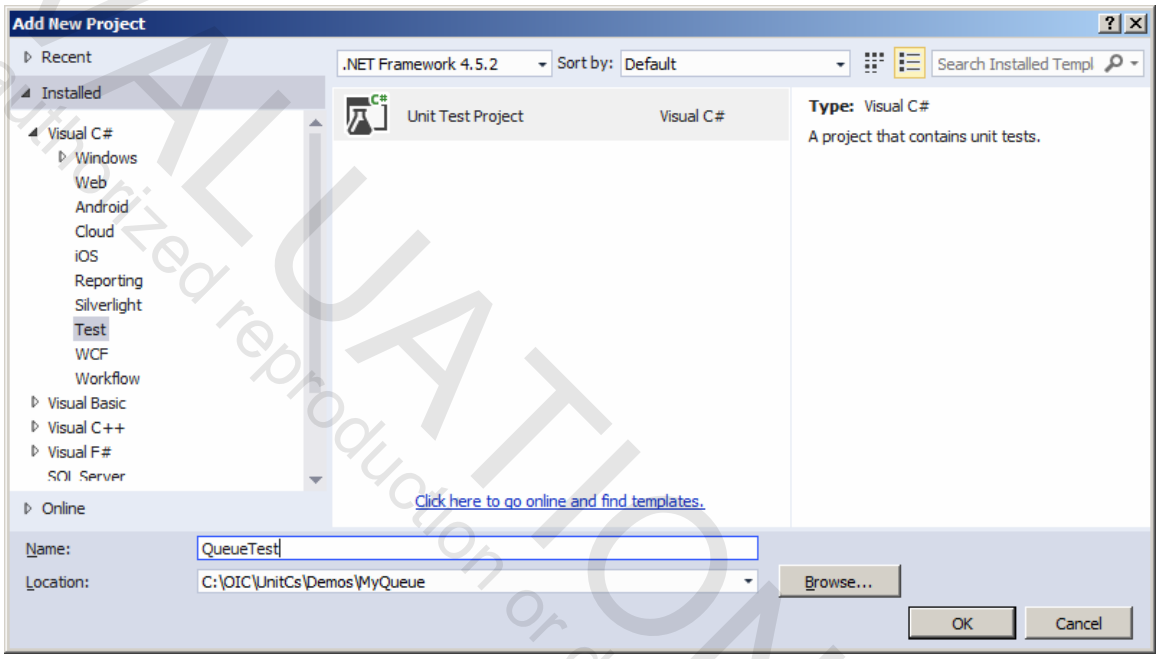
Test List for Queue

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. Add a new test project **QueueTest** to the solution.



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

4. Edit the supplied stub test method.

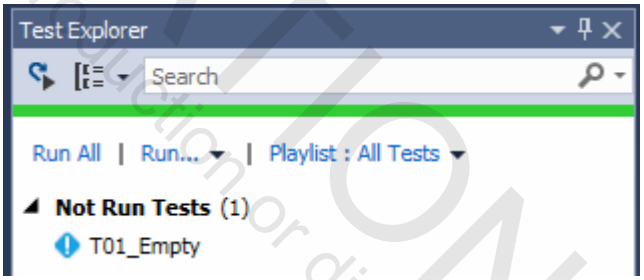
```
[TestMethod]
public void T01_Empty()
{
    MyQueue que = new MyQueue(3);
    Assert.AreEqual(0, que.Count);
}
```

Demo: Testing QueueLib (Cont'd)

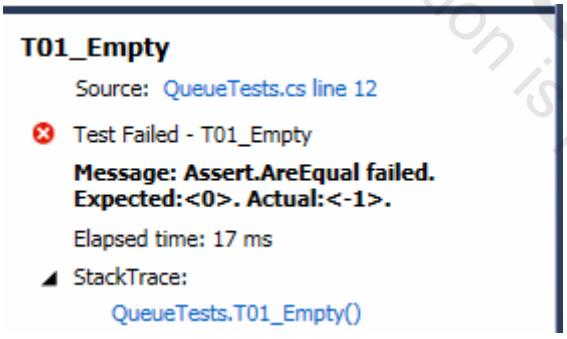
5. Build the solution. You will get a compile error, because the **MyQueue** class cannot be found by the test project.
6. In the **QueueTest** project add a reference to the **QueueLib** project.
7. In **QueueTests.cs** add a **using** statement to import the **QueueLib** namespace.

```
using QueueLib;
```

8. Build the solution. Bring up the Test Explorer window from the menu Test | Windows | Test Explorer.



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



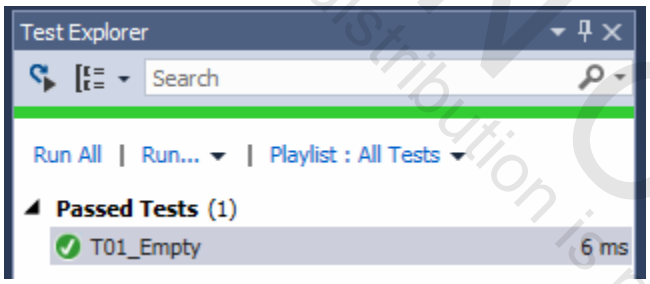
10. The failure was expected, because we only have stub code for the implementation of the Queue.

Demo: Testing QueueLib (Cont'd)

11. 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;
      }
   }
   ...
```

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

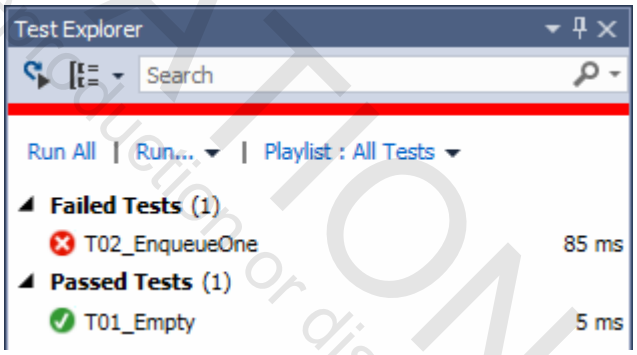


A Second Test

13. 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);
}
```

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



More Queue Functionality

15. 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;
   }
   ...
```

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

17. 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.

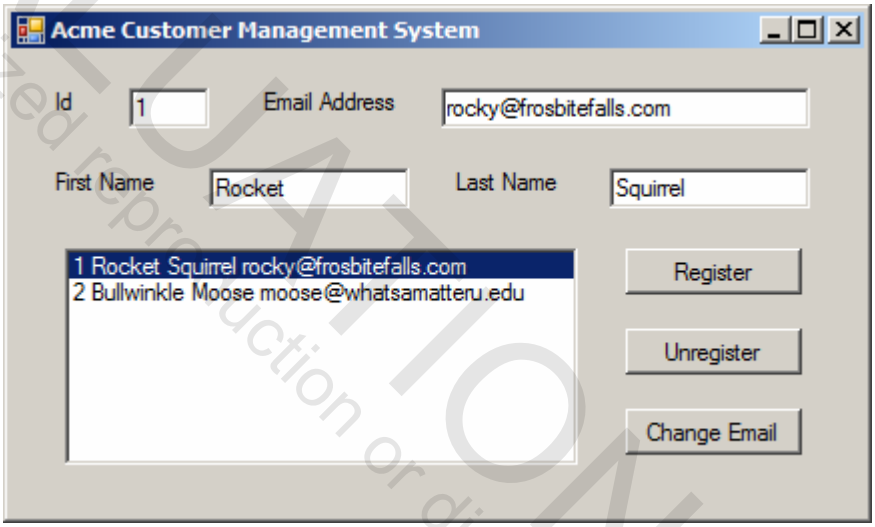
Acme Travel Agency Case Study

- **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**.



- **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 2008 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.**