Unit Testing

Estimated time for completion: 45 minutes

Overview:

In this lab you will create a prioritized queue using Test Driven Development

Goals:

- Understand how naming and structure of tests helps other developers
- Learn the cycle of Red-Green-Refactor
- See how comprehensive tests allow safe refactoring

Lab Notes:

N/A

Part 1: Writing your first tests

In the first part of the lab you will use Visual Studio's code generation features to build the code as you write the tests generating a set of simple initial tests

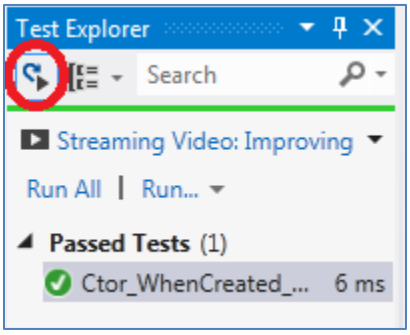
1. Open **Before/UnitTesting.sln**
2. Add a unit test project to the solution (under the project category of Test) calling it **DM.Collections.Test** (in other words the test project for the **DM.Collections** assembly)
3. In the test project add a reference to the **DM.Collections** project
4. Rename the **UnitTest1.cs** file to **PriorityQueueTests.cs** (this will also rename the class in the file). Open this file
5. Rename the existing test to `Ctor_WhenCreated_ShouldHaveCount0`
6. Writing the test now will not initially compile but with test driven development that's OK as the tools do the work for you. Write the test to create an instance of a generic class called `PriorityQueue`, specialized with an `int`, and then `Assert` that its `Count` property is 0

```
[TestMethod]
public void Ctor_WhenCreated_ShouldHaveCount0()
{
    var q = new PriorityQueue<int>();
    Assert.AreEqual(0, q.Count);
}
```

7. Click on the name `PriorityQueue` and open the smart tag (either with the mouse or Ctrl-.) and select *Generate new type*. In the dialog select **DM.Collections** as the

destination project. Do not use the *Generate class* option as this will create the class in the test project.

8. Click on the `Count` property and use the smart tag to generate the property in the `PriorityQueue` class
9. Run the test using the *Test Explorer* (select the *Test/Windows/Test Explorer* menu options to show it if it is not already visible. Notice that the test fails and if you look at the bottom of the Test Explorer you will see that the expected value was 0 but the actual value was null. This is due to the way Visual Studio generates the code and is a good thing as we always want to start with a failing test
10. Open the **PriorityQueue.cs** file and change the `Count` property type to an `int`
11. Press the *Run Tests After Build* button in the Test Explorer and build the project – you should see the test pass



12. Write a test using the same naming convention to verify that the `Enqueue` method increments the `Count` by 1. Compile the code and you will see the test fail as the generated method throws a `NotImplementedException`. Change the `Enqueue` method to pass the test. Remember only write enough code to pass the test

```
public void Enqueue(int p)
{
    Count++;
}
```

13. Refactor the code so that the `Enqueue` method parameter has a sensible name and compile (the test will pass again). Taking time at the end of each cycle to restructure the code is good practice to ensure the code retains a good structure. You ensure that all of the functionality still works by the fact that the tests still pass
14. Write a test to ensure that `Dequeue` decrements the `Count` again ensuring that you only write enough code to pass the test

```
[TestMethod]
public void Dequeue_WhenCalledAfterEnqueue_ShouldReturnCountToInitialValue()
{
    var q = new PriorityQueue<int>();
    int count = q.Count;
    q.Enqueue(5);

    q.Dequeue();

    Assert.AreEqual(count, q.Count);
}
```

15. Write a test to ensure that if an item is enqueued then `Dequeue` returns that item. We are now starting to implement the guts of the queue. You will most likely have to change the signature of the `Dequeue` method now as the generated version may well have a void return. Implement the functionality by having a variable of type `T` and storing the enqueued item on it then returning it in the `Dequeue`

```
public class PriorityQueue<T>
{
    public int Count { get; set; }
    private T item;

    public void Enqueue(T item)
    {
        Count++;
        this.item = item;
    }

    public T Dequeue()
    {
        Count--;
        return item;
    }
}
```

16. Write a test to ensure that if two items are enqueued then `Dequeue` returns the first item – the essential behavior of a queue. Implement the code by creating a `List<T>` to hold the items and return the item at index 0 in `Dequeue`.

```
public class PriorityQueue<T>
{
    public int Count { get; set; }
    private List<T> items = new List<T>();

    public void Enqueue(T item)
    {
        Count++;
        items.Add(item);
    }

    public T Dequeue()
    {
        Count--;
        return items[0];
    }
}
```

17. Refactor the code to make the `Count` property hand off to the `Count` of the List and remove the increment and decrement in the `Enqueue` and `Dequeue`. Compile and notice you now have a failing test as `Dequeue` doesn't remove the item currently. Change the code to remove the item

```
public T Dequeue()
{
    T item = items[0];
    items.RemoveAt(0);
    return item;
}
```

18. Add a test to ensure that `Dequeue` throws an `InvalidOperationException` if the queue is empty. For this you will need to use the `ExpectedException` attribute on the test

```
[TestMethod]
[ExpectedException(typeof(InvalidOperationException))]
public void Dequeue_WhenQueueIsEmpty_ShouldThrowAnExcedption()
{
    var q = new PriorityQueue<int>();
    q.Dequeue();
}
```

19. Finally notice all of your tests have the same line of code creating the queue. Test code is no different from your application code – you need to make it as maintainable as possible. Create a method called `Setup` in the test class and annotate it with the `[TestInitialize]` attribute. You can now move the creation of the queue into this method making `q` a member variable in the class. Delete the common code from the tests.

```
private PriorityQueue<int> q;
[TestInitialize]
public void Setup()
{
    q = new PriorityQueue<int>();
}
```

Compile and verify all of your tests pass. You have now implemented a queue

---

Part 2: Implementing the Prioritisation handing

In the last part of the lab you will implement the priority basis of the PriorityQueue completing the functionality. You wikll see that we can refactor, even quite aggressively, knowing we haven't broken our functionality

1. Add a test to verify that a high priority enqueued item enqueued after a normal one is Dequeued first. You will need to generate a `Priority` enum along the way and cfreate the new functionality as an overload of `Enqueue` that takes a `Priority`

```
[TestMethod]
public void
   Dequeue_WhenEnqueueWithHighPriority_ShouldDequeueHighPriorityFirst()
{
    int val = 42;
    q.Enqueue(5);
    q.Enqueue(val, Priority.High);

    int result = q.Dequeue();

    Assert.AreEqual(val, result);
}
```

2. To make this pass will involve a bit of surgery in the **DM.Collections** project. Create a new generic class called `PriorityItem` that have a readonly property of type `T` and a readonly property of type `Priority`

```
class PriorityItem<T>
{
    public PriorityItem(T item, Priority priority)
    {
        Priority = priority;
        Item = item;
    }

    public T Item { get; private set; }
    public Priority Priority { get; private set; }
}
```

3. Change the List in the `PriorityQueue` to be a `List<PriorityItem>`.
4. If you haven't already done so add a `Normal` value to the `Priority enum`.
5. In the `Enqueue` method with no parameters add the item to the list with a priority of `Normal`.

6. Change the `Dequeue` method to use `PriorityItem<T>` in its implementation. Compile your code and see that all tests pass except the new one. We have refactored the code ready for the new implementation.
7. In the new `Enqueue` method add the item to the list with the passed priority
8. Change `Dequeue` to look through the list for an item with high priority. If one exists then remove it and pass back its value, otherwise return the item at index 0 as before

```
public T Dequeue()
{
    if (Count == 0)
    {
        throw new InvalidOperationException();
    }

    int? itemToReturnIndex = null;

    for (int i = 0; i < items.Count; i++)
    {
        if (items[i].Priority == Priority.High)
        {
            itemToReturnIndex = i;
            break;
        }
    }
    if (itemToReturnIndex == null)
    {
        itemToReturnIndex = 0;
    }

    PriorityItem<T> itemToReturn = items[(int) itemToReturnIndex];
    items.RemoveAt((int)itemToReturnIndex);
    return itemToReturn.Item;
}
```

9. Now write a test to ensure that a low priority item enqueued before a normal item is dequeued after it

```
[TestMethod]
public void Dequeue_WhenEnqueueWithLowPriority_ShouldDequeueAfterNormal()
{
    int val = 42;
    q.Enqueue(5, Priority.Low);
    q.Enqueue(val);

    int result = q.Dequeue();

    Assert.AreEqual(val, result);
}
```

10. Add another loop in the `Dequeue` to look for a normal priority item if no high priority item has been found and to dequeue that in preference to a low priority item at index 0

```
for (int i = 0; i < items.Count; i++)
{
    if (items[i].Priority == Priority.High)
    {
        itemToReturnIndex = i;
        break;
    }
}
if (itemToReturnIndex == null)
{
    for (int i = 0; i < items.Count; i++)
    {
        if (items[i].Priority == Priority.Normal)
        {
            itemToReturnIndex = i;
            break;
        }
    }
    if (itemToReturnIndex == null)
    {
        itemToReturnIndex = 0;
    }
}
```

11. Compile and you should see all of the tests pass
12. However, our current implementation is not good for large queues as we traverse the entire list twice. We should be able to optimize this by watching out for a normal priority item as we traverse the list for high priority items. Fortunately we have unit tests to verify we don't break anything as we make this change

```
public T Dequeue()
{
    if (Count == 0)
    {
        throw new InvalidOperationException();
    }

    int? itemToReturnIndex = null;
    int? firstNormalPriorityIndex = null;

    for (int i = 0; i < items.Count; i++)
    {
        if (items[i].Priority == Priority.High)
        {
            itemToReturnIndex = i;
            break;
        }
        if (firstNormalPriorityIndex == null &&
            items[i].Priority == Priority.Normal)
        {
            firstNormalPriorityIndex = i;
        }
    }
```



```
    if (itemToReturnIndex == null)
    {
        itemToReturnIndex = firstNormalPriorityIndex ?? 0;
    }

    PriorityItem<T> itemToReturn = items[(int) itemToReturnIndex];
    items.RemoveAt((int)itemToReturnIndex);
    return itemToReturn.Item;
}
```

13. Compile and verify your test still pass

Solutions

after\UnitTesting.sln