МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Практикум №2

з курсу «Сучасні технології розробки WEB-застосувань на платформі Microsoft.NET»

на тему: «Модульне тестування. Ознайомлення з засобами та практиками модульного тестування »

Виконав:
студент 3 курсу
групи ІП-11 ФІОТ
Рябов Ю. І.

Київ-2023

Варіант 2

Мета лабораторної роботи – навчитися створювати модульні тести для вихідного коду розроблювального програмного забезпечення.

Завдання:

1. Додати до проекту власної узагальненої колекції (застосувати виконану лабораторну роботу No1) проект модульних тестів, використовуючи певний фреймворк (Nunit, Xunit, тощо).
2. Розробити модульні тести для функціоналу колекції.
3. Дослідити ступінь покриття модульними тестами вихідного коду колекції, використовуючи, наприклад, засіб AxoCover.

Колекція згідно варіанту – черга.

Гіт репозиторій лабораторної роботи:

https://github.com/YuraRiabov/WebNet1Collections

Код лабораторної роботи:

```
using System.Collections;
using System.Diagnostics.CodeAnalysis;

namespace Collections;

public class MyQueue<T> : IEnumerable<T>, ICollection
{
    private MyQueueNode? _first;
    private MyQueueNode? _last;

    public event Action? LastElementRemoved;
    public event Action? LastElementLeft;

    public int Count
    {
        get
        {
            var count = 0;
            var currentItem = _first;
            while (currentItem is not null)
            {
                count++;
                currentItem = currentItem.Next;
            }

            return count;
        }
    }
    public bool IsSynchronized => false;
    public object SyncRoot => this;
```

```
public MyQueue()
{
    _first = null;
    _last = null;
}

public MyQueue(IEnumerable<T> source)
{
    foreach (var value in source)
    {
        Enqueue(value);
    }
}

public void Enqueue(T value)
{
    var newNode = new MyQueueNode(value);

    if (_first is null)
    {
        _first = newNode;
        return;
    }

    if (_last is null)
    {
        _first.Next = newNode;
        _last = newNode;
        return;
    }

    _last.Next = newNode;
    _last = newNode;
}

public T Dequeue()
{
    if (_first is null)
    {
        throw new InvalidOperationException();
    }

    return DequeueInternal();
}

public bool TryDequeue([MaybeNullWhen(false)] out T value)
{
    if (_first is null)
    {
        value = default;
        return false;
    }

    value = DequeueInternal();
    return true;
}

public T Peek()
{
    if (_first is null)
    {
        throw new InvalidOperationException();
    }

    return _first.Value;
}
```

```
    public bool TryPeek([MaybeNullWhen(false)] out T value)
    {
        if (_first is null)
        {
            value = default;
            return false;
        }

        value = _first.Value;
        return true;
    }

    public void Clear()
    {
        _first = null;
        _last = null;
        LastElementRemoved?.Invoke();
    }

    public bool Contains(T item)
    {
        var currentItem = _first;
        while (currentItem is not null)
        {
            if (currentItem.Value is null && item is null || currentItem.Value is
not null && currentItem.Value.Equals(item))
            {
                return true;
            }

            currentItem = currentItem.Next;
        }

        return false;
    }

    public T[] ToArray()
    {
        var array = new T[Count];
        CopyToInternal(array, 0);
        return array;
    }

    public void CopyTo(T[] array, int index)
    {
        if (!IsValidArrayLengthIndexForCopy(array, index))
        {
            throw new ArgumentOutOfRangeException();
        }

        CopyToInternal(array, index);
    }

    void ICollection.CopyTo(Array array, int index)
    {
        if (!IsValidArrayLengthIndexForCopy(array, index))
        {
            throw new ArgumentOutOfRangeException();
        }

        if (!IsValidArrayTypeForCopy(array))
        {
            throw new ArgumentException(nameof(array));
        }

        CopyToInternal(array, index);
```

```
    }

    public IEnumerator<T> GetEnumerator()
    {
        var currentNode = _first;
        while (currentNode is not null)
        {
            yield return currentNode.Value;
            currentNode = currentNode.Next;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    private T DequeueInternal()
    {
        var first = _first;
        _first = _first!.Next;

        if (_first is not null && !_first.HasNext)
        {
            _last = null;
            LastElementLeft?.Invoke();
        }

        if (_first is null)
        {
            LastElementRemoved?.Invoke();
        }

        return first!.Value;
    }

    private void CopyToInternal(Array array, int index)
    {
        var currentIndex = index;
        var currentItem = _first;
        while (currentItem is not null)
        {
            array.SetValue(currentItem.Value, currentIndex);
            currentItem = currentItem.Next;
            currentIndex++;
        }
    }

    private bool IsValidArrayLengthIndexForCopy(Array array, int index)
    {
        if (array is null)
        {
            return false;
        }

        if (index < 0 || index >= array.Length)
        {
            return false;
        }

        return array.Length - index >= Count;
    }

    private bool IsValidArrayTypeForCopy(Array array)
    {
        return array.Rank == 1 && array.GetLowerBound(0) == 0;
    }
```

```
    private class MyQueueNode
    {
        public T Value { get; }
        public MyQueueNode? Next { get; set; }
        public bool HasNext => Next is not null;

        public MyQueueNode(T value)
        {
            Value = value;
        }
    }
}
```

```
using Xunit;

namespace Collections.Test.Unit.Abstract;

public abstract class MyQueueTestsBase
{
    private static readonly int[] NumberArray = { 1, 2, 3, 4 };
    private static readonly TestStructure[] StructureArray = { new(1), new(2),
new(3), new(4) };
    private static readonly TestClass[] ObjectArray = { new(1), new(2), null,
new(3), new(4) };

    public static IEnumerable<object[]> GetEmptyQueuesTestData()
    {
        yield return new object[] { new MyQueue<int>() };
        yield return new object[] { new MyQueue<TestStructure>() };
        yield return new object[] { new MyQueue<TestClass>() };
    }

    public static IEnumerable<object[]> GetTestDataForQueueFill()
    {
        yield return new object[] { NumberArray };
        yield return new object[] { StructureArray };
        yield return new object[] { ObjectArray };
    }

    protected static void AssertEqualCollections<T>(IEnumerable<T> expected,
IEnumerable<T> actual)
    {
        using var enumerator = actual.GetEnumerator();
        foreach (T value in expected)
        {
            enumerator.MoveNext();
            var currentValue = enumerator.Current;

            Assert.Equal(value, currentValue);
        }
    }

    protected class TestClass
    {
        public int Value { get; set; }

        public TestClass(int value)
        {
            Value = value;
        }
    }

    protected struct TestStructure
    {
```

```
        public int Value { get; set; }

        public TestStructure(int value)
        {
            Value = value;
        }
    }
}
```

```
using Collections.Test.Unit.Abstract;
using Xunit;

namespace Collections.Test.Unit;

public class MyQueueCollectionTests : MyQueueTestsBase
{
    [Theory]
    [MemberData(nameof(GetEmptyQueuesTestData))]
    public void
GetEnumerator_WhenEmptyQueue_ShouldReturnMoveNextFalse<T>(MyQueue<T> queue)
    {
        using var enumerator = queue.GetEnumerator();

        var moveNextResult = enumerator.MoveNext();
        var current = enumerator.Current;

        Assert.False(moveNextResult);
        Assert.Equal(default, current);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void Clear_ShouldMakeQueueEmpty<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);

        queue.Clear();

        Assert.Empty(queue);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void Contains_WhenHasElement_ShouldBeTrue<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);

        foreach (var value in values)
        {
            var containsResult = queue.Contains(value);

            Assert.True(containsResult);
        }
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void Contains_WhenHasNoElement_ShouldBeFalse<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);

        queue.Dequeue();
        var containsResult = queue.Contains(values[0]);

        Assert.False(containsResult);
    }
```

```
    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void Count_WhenElementsChange_ShouldBeCorrect<T>(T[] values)
    {
        var queue = new MyQueue<T>();
        for (int i = 0; i < values.Length; i++)
        {
            queue.Enqueue(values[i]);

            Assert.Equal(i + 1, queue.Count);
        }

        queue.Clear();
        Assert.Equal(0, queue.Count);
    }

    [Theory]
    [MemberData(nameof(GetEmptyQueuesTestData))]
    public void ToArray_WhenEmptyQueue_ShouldReturnEmptyArray<T>(MyQueue<T> queue)
    {
        var array = queue.ToArray();

        Assert.Empty(array);
    }
}
```

```
using System.Collections;
using Collections.Test.Unit.Abstract;
using Xunit;

namespace Collections.Test.Unit;

public class MyQueueCopyToTests : MyQueueTestsBase
{
    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void ToArray_WhenNotEmptyQueue_ShouldReturnArray<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);

        var array = queue.ToArray();

        AssertEqualCollections(array, values);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void CopyTo_WhenValidParams_ShouldCopy<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);

        var array = new T[values.Length];

        queue.CopyTo(array, 0);

        AssertEqualCollections(array, values);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void CopyTo_WhenLongerArray_ShouldCopy<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);

        var array = new T[values.Length];
```

```
        queue.Dequeue();

        queue.CopyTo(array, 1);

        Assert.DoesNotContain(values[0], array);
        Assert.Equal(default, array[0]);
        for (var i = 1; i < values.Length; i++)
        {
            Assert.Equal(values[i], array[i]);
        }
    }

    [Theory]
    [MemberData(nameof(GetCopyToGenericTestData))]
    public void CopyTo_WhenInvalidParams_ShouldThrow<T>(T[] values, T[] array, int
index, Type exceptionType)
    {
        var queue = new MyQueue<T>(values);

        var code = () => queue.CopyTo(array, index);

        Assert.Throws(exceptionType, code);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void CopyToNonGeneric_WhenValidParams_ShouldCopy<T>(T[] values)
    {
        ICollection queue = new MyQueue<T>(values);

        Array array = new T[values.Length];

        queue.CopyTo(array, 0);

        Assert.Equal(array.Length, values.Length);
        Assert.Equal(array.GetValue(0), values[0]);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void CopyToNonGeneric_WhenLongerArray_ShouldCopy<T>(T[] values)
    {
        ICollection queue = new MyQueue<T>(values);

        Array array = new T[values.Length + 1];

        queue.CopyTo(array, 1);

        Assert.Equal(default(T), array.GetValue(0));
        for (var i = 0; i < values.Length; i++)
        {
            Assert.Equal(values[i], array.GetValue(i + 1));
        }
    }

    [Theory]
    [MemberData(nameof(GetCopyToNotGenericTestData))]
    public void CopyToNotGeneric_WhenInvalidData_ShouldThrow<T>(T[] values, Array
array, int index, Type exceptionType)
    {
        ICollection queue = new MyQueue<T>(values);

        var code = () => queue.CopyTo(array, index);

        Assert.Throws(exceptionType, code);
    }
```

```
    public static IEnumerable<object[]> GetCopyToGenericTestData() =>
GetCopyToTestData(true);

    public static IEnumerable<object[]> GetCopyToNotGenericTestData() =>
GetCopyToTestData(false);

    private static IEnumerable<object[]> GetCopyToTestData(bool isGeneric)
    {
        var types = new[] { typeof(int), typeof(TestStructure), typeof(TestClass)
};
        var currentTypeIndex = 0;
        foreach (var values in GetTestDataForQueueFill())
        {
            var array = values[0] as Array;
            var type = types[currentTypeIndex];
            yield return new object[] { array, null, 0,
typeof(ArgumentOutOfRangeException) };
            yield return new object[] { array, Array.CreateInstance(type,
array.Length - 1), 0, typeof(ArgumentOutOfRangeException) };
            yield return new object[] { array, Array.CreateInstance(type,
array.Length ), 1, typeof(ArgumentOutOfRangeException) };
            yield return new object[] { array, Array.CreateInstance(type,
array.Length ), -1, typeof(ArgumentOutOfRangeException) };
            yield return new object[] { array, Array.CreateInstance(type,
array.Length ), array.Length + 1, typeof(ArgumentOutOfRangeException) };
            if (!isGeneric)
            {
                yield return new object[] { array, Array.CreateInstance(type, new
[]{ array.Length, array.Length }, new []{ 0, 0 } ), 0, typeof(ArgumentException)
};
                yield return new object[] { array, Array.CreateInstance(type, new
[]{ array.Length }, new []{ 1 } ), 0, typeof(ArgumentException) };
            }
            currentTypeIndex++;
        }
    }
}
```

```
using Collections.Test.Unit.Abstract;
using Xunit;

namespace Collections.Test.Unit;

public class MyQueueCoreTests : MyQueueTestsBase
{
    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void Enqueue_WhenEmptyQueue_ShouldAddElements<T>(T[] values)
    {
        var queue = new MyQueue<T>();
        foreach (var value in values)
        {
            queue.Enqueue(value);
        }

        AssertEqualCollections(values, queue);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void Enqueue_WhenQueueCleared_ShouldAddElements<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);

        queue.Clear();
```

```
        foreach (var value in values)
        {
            queue.Enqueue(value);
        }

        AssertEqualCollections(values, queue);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void Enqueue_WhenNotEmptyQueue_ShouldAddElements<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);
        foreach (var value in values)
        {
            queue.Enqueue(value);
        }

        Assert.Equal(values.Length * 2, queue.Count);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void EnumerableConstructor_ShouldAddElements<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);

        AssertEqualCollections(values, queue);
    }


    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void Dequeue_WhenNotEmptyQueue_ShouldReturnAndRemove<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);

        foreach (T value in values)
        {
            var currentValue = queue.Dequeue();

            Assert.Equal(value, currentValue);
        }

        Assert.Empty(queue);
    }

    [Theory]
    [MemberData(nameof(GetEmptyQueuesTestData))]
    public void Dequeue_WhenEmptyQueue_ShouldThrow<T>(MyQueue<T> queue)
    {
        var call = () =>
        {
            queue.Dequeue();
        };

        Assert.Throws<InvalidOperationException>(call);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void TryDequeue_WhenNotEmptyQueue_ShouldReturnTrueAndRemove<T>(T[]
values)
    {
        var queue = new MyQueue<T>(values);

        foreach (T value in values)
```

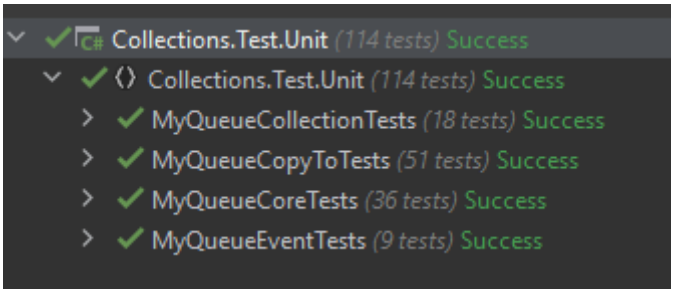
```
        {
            bool tryDequeueResult = queue.TryDequeue(out var currentValue);

            Assert.Equal(value, currentValue);
            Assert.True(tryDequeueResult);
        }

        Assert.Empty(queue);
    }

    [Theory]
    [MemberData(nameof(GetEmptyQueuesTestData))]
    public void TryDequeue_WhenEmptyQueue_ShouldReturnFalse<T>(MyQueue<T> queue)
    {
        var tryDequeueResult = queue.TryDequeue(out var currentValue);

        Assert.Empty(queue);
        Assert.False(tryDequeueResult);
        Assert.Equal(default, currentValue);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void Peek_WhenNotEmptyQueue_ShouldReturnAndNotRemove<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);

        foreach (T value in values)
        {
            var currentValue = queue.Peek();

            Assert.Equal(value, currentValue);

            queue.Dequeue();
        }

        Assert.Empty(queue);
    }

    [Theory]
    [MemberData(nameof(GetEmptyQueuesTestData))]
    public void Peek_WhenEmptyQueue_ShouldThrow<T>(MyQueue<T> queue)
    {
        var call = () =>
        {
            queue.Peek();
        };

        Assert.Throws<InvalidOperationException>(call);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void TryPeek_WhenNotEmptyQueue_ShouldReturnTrueAndNotRemove<T>(T[]
values)
    {
        var queue = new MyQueue<T>(values);

        foreach (T value in values)
        {
            bool tryDequeueResult = queue.TryPeek(out var currentValue);

            Assert.Equal(value, currentValue);
            Assert.True(tryDequeueResult);

            queue.Dequeue();
        }
```

```
        Assert.Empty(queue);
    }

    [Theory]
    [MemberData(nameof(GetEmptyQueuesTestData))]
    public void TryPeek_WhenEmptyQueue_ShouldReturnFalse<T>(MyQueue<T> queue)
    {
        var tryDequeueResult = queue.TryPeek(out var currentValue);

        Assert.Empty(queue);
        Assert.False(tryDequeueResult);
        Assert.Equal(default, currentValue);
    }
}
```

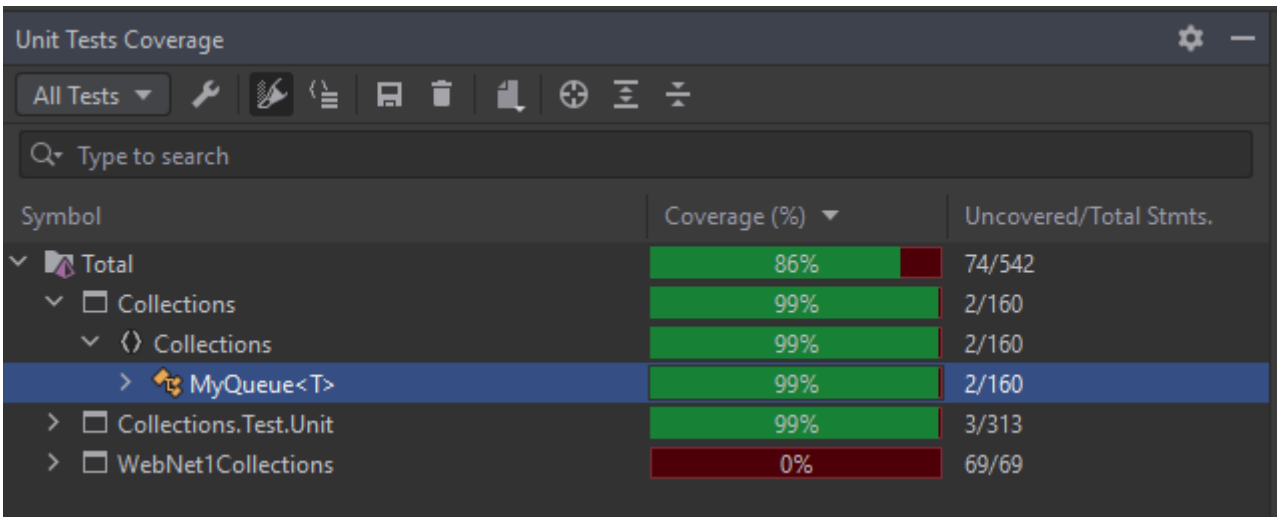
```
using Collections.Test.Unit.Abstract;
using FakeItEasy;
using Xunit;

namespace Collections.Test.Unit;

public class MyQueueEventTests : MyQueueTestsBase
{
    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void LastElementRemoved_WhenLastItemRemoved_ShouldBeCalled<T>(T[]
values)
    {
        var queue = new MyQueue<T>(values);
        var eventHandler = A.Fake<ITestEventsHandler>();
        queue.LastElementRemoved += eventHandler.Callback;
        queue.LastElementRemoved += eventHandler.Callback;

        foreach (var _ in values)
        {
            queue.Dequeue();
        }

        A.CallTo(() => eventHandler.Callback).MustHaveHappened(2, Times.Exactly);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void LastElementRemoved_WhenQueueCleared_ShouldBeCalled<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);
        var eventHandler = A.Fake<ITestEventsHandler>();
        queue.LastElementRemoved += eventHandler.Callback;

        queue.Clear();

        A.CallTo(() => eventHandler.Callback).MustHaveHappened(1, Times.Exactly);
    }

    [Theory]
    [MemberData(nameof(GetTestDataForQueueFill))]
    public void LastElementLeft_WhenLastItemLeft_ShouldBeCalled<T>(T[] values)
    {
        var queue = new MyQueue<T>(values);
        var eventHandler = A.Fake<ITestEventsHandler>();
        queue.LastElementLeft += eventHandler.Callback;

        for (var i = 0; i < values.Length - 1; i++)
        {
            queue.Dequeue();
```

```
        }

        A.CallTo(() => eventHandler.Callback).MustHaveHappened(1, Times.Exactly);
    }

    public interface ITestEventsHandler
    {
        public Action Callback { get; set; }
    }
}
```

Скріншот екрану виконання тестів:



Скріншот покриття проекту тестами(за допомогою dotCover):



Висновок:

Отже, ми освоїли фреймворк xUnit та техніки модульного тестування та навчились перевіряти покриття проекту тестами за допомогою dotCover.