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

Звіт
з лабораторної роботи №2 з дисципліни
«Сучасні технології розробки WEB-застосувань на платформі Microsoft.NET»

«Модульне тестування. Ознайомлення з засобами та практиками модульного тестування»

Варіант NV

Виконав студент ІП-13 Дем’янчук Олександр Петрович
(шифр, прізвище, ім'я, по батькові)

Перевірив Бардін В.
(прізвище, ім'я, по батькові)

Київ 2023

Лабораторна робота №2

Варіант 2

Тема: Модульне тестування. Ознайомлення з засобами та практиками модульного тестування.

Мета: навчитися створювати модульні тести для вихідного коду розроблювального програмного забезпечення.

Постановка задачі

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.

2	Черга	Див. Queue<T>	Збереження даних за допомогою динамічно зв’язаного списку

Виконання завдань

Код програми

```
using System.Collections;
using System.Diagnostics.CodeAnalysis;
using WebNetLab1.Collections.EventArgs;

namespace WebNetLab1.Collections;

public class MyQueue<T> : IEnumerable<T>, ICollection
{
    private MyQueueNode? _head;
    private MyQueueNode? _tail;

    public event EventHandler<PeekEventArgs<T>> PeekEvent;
    public event EventHandler<QueueEmptyEventArgs> QueueEmptyEvent;
```

```
public MyQueue()
{
    _head = null;
    _tail = null;
}

public MyQueue(IEnumerable<T> source)
{
    if (source is null)
    {
        throw new ArgumentNullException(nameof(source));
    }
    foreach (var item in source)
    {
        Enqueue(item);
    }
}

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

        return count;
```

```
        }
    }

    public bool IsSynchronized => false;
    public object SyncRoot => this;

    public void CopyTo(T[] array, int arrayIndex)
    {
        if (array is null)
        {
            throw new ArgumentNullException(nameof(array));
        }

        if (arrayIndex < 0 || arrayIndex > array.Length)
        {
            throw new ArgumentOutOfRangeException(nameof(arrayIndex),
arrayIndex,
                "Index is out of bounds of this array.");
        }

        if (array.Length - arrayIndex < Count)
        {
            throw new ArgumentException("There's not enough space to copy
into this range of an array.");
        }

        if (Count == 0)
        {
            return;
        }

        var current = _head;
```

```
        int index = arrayIndex;
        while (current is not null)
        {
            array[index] = current.Data;
            index++;
            current = current.Next;
        }
    }

    public void Clear()
    {
        _head = null;
        _tail = null;
        OnQueueEmpty(new QueueEmptyEventArgs("A queue was cleared."));
    }

    void ICollection.CopyTo(Array array, int arrayIndex)
    {
        if (array is null)
        {
            throw new ArgumentNullException(nameof(array));
        }

        if (array.Rank != 1)
        {
            throw new ArgumentException("The array has an invalid rank.");
        }

        if (array.GetLowerBound(0) != 0)
        {
            throw new ArgumentException("The array must have a lower bound
of 0.");
```

```
            }

            if (arrayIndex < 0 || arrayIndex > array.Length)
            {
                throw new ArgumentOutOfRangeException(nameof(arrayIndex),
arrayIndex,
                    "Index is out of bounds of this array.");
            }

            if (array.Length - arrayIndex < Count)
            {
                throw new ArgumentException("There's not enough space to copy
into this range of an array.");
            }

            if (Count == 0)
            {
                return;
            }

            var current = _head;
            int index = arrayIndex;
            while (current is not null)
            {
                array.SetValue(current.Data, index);
                index++;
                current = current.Next;
            }
        }

        public IEnumerator<T> GetEnumerator()
        {
```

```
    var current = _head;
    while (current is not null)
    {
        yield return current.Data;
        current = current.Next;
    }
}

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

public void Enqueue(T item)
{
    var newNode = new MyQueueNode(item);
    if (_head is null)
    {
        _head = newNode;
        _tail = _head;
        return;
    }

    _tail.Next = newNode;
    _tail = newNode;
}

public T Dequeue()
{
    if (_head is null)
    {
        throw new InvalidOperationException("The queue is empty.");
```

```
            }

            var removedData = HandleDequeue();

            return removedData;
        }

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

            result = HandleDequeue();
            return true;
        }

        public T Peek()
        {
            if (_head is null)
            {
                throw new InvalidOperationException("The queue is empty.");
            }

            var result = _head.Data;
            OnPeek(new PeekEventArgs<T>("An element was retrieved from Peek
method.", result));
            return result;
        }
```

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

            result = _head.Data;
            OnPeek(new PeekEventArgs<T>("An element was retrieved from
TryPeek method.", result));
            return true;
        }

        public bool Contains(T item)
        {
            var current = _head;
            while (current is not null)
            {
                if (current.Data.Equals(item))
                {
                    return true;
                }

                current = current.Next;
            }

            return false;
        }

        public T[] ToArray()
        {
```

```
            if (Count == 0)
            {
                return Array.Empty<T>();
            }

            var array = new T[Count];

            CopyTo(array, 0);

            return array;
        }

        private T HandleDequeue()
        {
            var dequeuedData = _head.Data;

            if (_head == _tail)
            {
                _head = null;
                _tail = null;

                OnQueueEmpty(new QueueEmptyEventArgs("The last element was
dequeued."));
            }
            else
            {
                _head = _head.Next;
            }

            return dequeuedData;
        }
```

```
    private void OnQueueEmpty(QueueEmptyEventArgs e)
    {
        QueueEmptyEvent?.Invoke(this, e);
    }
    private void OnPeek(PeekEventArgs<T> e)
    {
        PeekEvent?.Invoke(this, e);
    }
    private class MyQueueNode
    {
        public T Data { get; }
        public MyQueueNode? Next { get; internal set; }

        public MyQueueNode(T data)
        {
            Data = data;
            Next = null;
        }
    }
}

namespace WebNetLab1.Collections.EventArgs;

public class QueueEmptyEventArgs : System.EventArgs
{
    public string Message { get; }

    public QueueEmptyEventArgs(string message)
    {
        Message = message;
    }
```

```
}
namespace WebNetLab1.Collections.EventArgs;

public class PeekEventArgs<T> : System.EventArgs
{
    public string Message { get; }
    public T Data { get; }

    public PeekEventArgs(string message, T data)
    {
        Message = message;
        Data = data;
    }

}
```

Код модульних тестів:

```
using WebNetLab1.Collections;
using WebNetLab1.Tests.ClassData;
using Xunit;

namespace WebNetLab1.Tests;

public class ContainsTests
{
    [Theory]
    [ClassData(typeof(MultipleItemsQueueData))]
    public void Contains_WhenHasElement_ThenReturnTrue<T>(T[] items)
    {
        var queue = new MyQueue<T>(items);

        var contains = queue.Contains(items[0]);
```

```
            Assert.True(contains);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void Contains_WhenHasNoElement_ThenReturnFalse<T>(T[]
items)
        {
            var queue = new MyQueue<T>(items);

            queue.Dequeue();
            var contains = queue.Contains(items[0]);

            Assert.False(contains);
        }
    }

    using System.Collections;
    using WebNetLab1.Collections;
    using WebNetLab1.Tests.ClassData;
    using Xunit;

    namespace WebNetLab1.Tests;

    public class CopyToTests
    {
        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
CopyToGeneric_WhenNonEmptyQueue_ThenCopyToNewArray<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
```

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

            queue.CopyTo(array, 0);

            Assert.Equal(items, array);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
CopyToGeneric_WhenArrayIsNull_ThenThrowArgumentNullException<T>(T[]
items)
        {
            var queue = new MyQueue<T>(items);

            Assert.Throws<ArgumentNullException>(() => queue.CopyTo(null!,
0));
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
CopyToGeneric_WhenIndexOutOfRange_ThenThrowArgumentOutOfRangeExce
ption<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
            var array = new T[items.Length];

            Assert.Throws<ArgumentOutOfRangeException>(() =>
queue.CopyTo(array, -1));
            Assert.Throws<ArgumentOutOfRangeException>(() =>
queue.CopyTo(array, items.Length + 1));
```

```
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
CopyToGeneric_WhenInsufficientSpace_ThenThrowArgumentException<T>(T[]
items)
        {
            var queue = new MyQueue<T>(items);
            var array = new T[items.Length - 1];

            Assert.Throws<ArgumentException>(() => queue.CopyTo(array, 0));
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void CopyToGeneric_WhenEmptyQueue_ThenQuit<T>(T[] items)
        {
            var queue = new MyQueue<T>();
            var array = new T[items.Length];

            var arrayBeforeCopy = (T[])array.Clone();
            queue.CopyTo(array, 0);

            Assert.Equal(arrayBeforeCopy, array);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
CopyToNonGeneric_WhenNonEmptyQueue_ThenCopyToNewArray<T>(T[]
items)
```

```
        {
            var queue = new MyQueue<T>(items);
            Array array = new T[items.Length];

            ((ICollection)queue).CopyTo(array, 0);

            Assert.Equal(items, array);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
CopyToNonGeneric_WhenArrayIsNull_ThenThrowArgumentNullException<T>(
T[] items)
        {
            var queue = new MyQueue<T>(items);

            Assert.Throws<ArgumentNullException>(() =>
((ICollection)queue).CopyTo(null!, 0));
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
CopyToNonGeneric_WhenArrayIsMultiDimensional_ThenThrowArgumentExcept
ion<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
            Array array = new T[items.Length, 1];

            Assert.Throws<ArgumentException>(() =>
((ICollection)queue).CopyTo(array, 0));
```

```
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
CopyToNonGeneric_WhenIndexOutOfRange_ThenThrowArgumentOutOfRangeE
xception<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
            Array array = new T[items.Length];

            Assert.Throws<ArgumentOutOfRangeException>(() =>
((ICollection)queue).CopyTo(array, -1));
            Assert.Throws<ArgumentOutOfRangeException>(() =>
((ICollection)queue).CopyTo(array, items.Length + 1));
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
CopyToNonGeneric_WhenInsufficientSpace_ThenThrowArgumentException<T>(
T[] items)
        {
            var queue = new MyQueue<T>(items);
            Array array = new T[items.Length - 1];

            Assert.Throws<ArgumentException>(() =>
((ICollection)queue).CopyTo(array, 0));
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
```

```
        public void CopyToNonGeneric_WhenEmptyQueue_ThenQuit<T>(T[]
items)
        {
            var queue = new MyQueue<T>();
            Array array = new T[items.Length];

            var arrayBeforeCopy = (T[])array.Clone();
            ((ICollection)queue).CopyTo(array, 0);

            Assert.Equal(arrayBeforeCopy, array);
        }
    }

    using WebNetLab1.Collections;
    using WebNetLab1.Tests.ClassData;
    using Xunit;

    namespace WebNetLab1.Tests;

    public class DequeueTests
    {
        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
DequeueOne_WhenNonEmptyQueue_ThenRemoveAndReturnAndDecreaseCount
<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);

            var dequeuedItem = queue.Dequeue();

            Assert.Equal(items[0], dequeuedItem);
```

```
            Assert.Equal(items.Length - 1, queue.Count);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
DequeueMany_WhenNonEmptyQueue_ThenRemoveAndReturnAndQueueEmpty
<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);

            foreach (var item in items)
            {
                var dequeued = queue.Dequeue();
                Assert.Equal(item, dequeued);
            }

            Assert.Empty(queue);
        }

        [Fact]
        public void
Dequeue_WhenEmptyQueue_ThenThrowInvalidOperationException()
        {
            var queue = new MyQueue<int>();

            Assert.Throws<InvalidOperationException>(() => queue.Dequeue());
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
```

```
Dequeue_WhenQueueCleared_ThenThrowInvalidOperationException<T>(T[]
items)
        {
            var queue = new MyQueue<T>(items);
            queue.Clear();

            Assert.Throws<InvalidOperationException>(() => queue.Dequeue());
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
TryDequeueOne_WhenNonEmptyQueue_ThenRemoveAndReturnTrueAndDecrea
seCount<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);

            var dequeuedResult = queue.TryDequeue(out var dequeuedItem);

            Assert.True(dequeuedResult);
            Assert.Equal(items[0], dequeuedItem);
            Assert.Equal(items.Length - 1, queue.Count);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
TryDequeueMany_WhenNonEmptyQueue_ThenRemoveAndReturnTrueAndQueu
eEmpty<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
```

```
            foreach (var item in items)
            {
                var dequeuedResult = queue.TryDequeue(out var dequeuedItem);
                Assert.True(dequeuedResult);
                Assert.Equal(item, dequeuedItem);
            }

            Assert.Empty(queue);
        }

        [Fact]
        public void
TryDequeue_WhenEmptyQueue_ThenReturnFalseAndDefault()
        {
            var queue = new MyQueue<int>();

            var dequeuedResult = queue.TryDequeue(out var dequeuedItem);

            Assert.False(dequeuedResult);
            Assert.Equal(default, dequeuedItem);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
TryDequeue_WhenQueueCleared_ThenReturnFalseAndDefault<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
            queue.Clear();

            var dequeuedResult = queue.TryDequeue(out var dequeuedItem);
```

```
            Assert.False(dequeuedResult);
            Assert.Equal(default, dequeuedItem);
        }
    }

    using WebNetLab1.Collections;
    using WebNetLab1.Tests.ClassData;
    using Xunit;

    namespace WebNetLab1.Tests;

    public class EnqueueTests
    {

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void Enqueue_WhenEmptyQueue_ThenAddItems<T>(T[] items)
        {
            var queue = new MyQueue<T>();

            foreach (var item in items)
            {
                queue.Enqueue(item);
            }

            Assert.Equal(items, queue);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void Enqueue_WhenNonEmptyQueue_ThenAddItems<T>(T[]
items)
```

```
    {
        var queue = new MyQueue<T>(items);

        foreach (var item in items)
        {
            queue.Enqueue(item);
        }

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

    [Theory]
    [ClassData(typeof(MultipleItemsQueueData))]
    public void Enqueue_WhenQueueCleared_ThenAddItems<T>(T[] items)
    {
        var queue = new MyQueue<T>(items);
        queue.Clear();

        foreach (var item in items)
        {
            queue.Enqueue(item);
        }

        Assert.Equal(items, queue);
    }
}

using WebNetLab1.Collections;
using WebNetLab1.Tests.ClassData;
using Xunit;
using FakeItEasy;
using WebNetLab1.Collections.EventArgs;
```

```
namespace WebNetLab1.Tests;

public class EventsTests
{
    [Theory]
    [ClassData(typeof(MultipleItemsQueueData))]
    public void
OnQueueEmpty_WhenLastElementDequeued_ThenOneCallbackHappened<T>(T[
] items)
    {
        var queue = new MyQueue<T>(items);
        var eventHandler = A.Fake<ITestEventHandler>();
        queue.QueueEmptyEvent += eventHandler.Callback;

        foreach (var item in items)
        {
            queue.Dequeue();
        }

        A.CallTo(() => eventHandler.Callback(A<object?>._,
A<QueueEmptyEventArgs>._))
            .MustHaveHappenedOnceExactly();
    }

    [Theory]
    [ClassData(typeof(MultipleItemsQueueData))]
    public void
QueueEmptyEventArgs_WhenLastElementDequeued_ThenMessageIsNotNull<T>
(T[] items)
    {
        var queue = new MyQueue<T>(items);
```

```
            var eventHandler = A.Fake<ITestEventHandler>();
            queue.QueueEmptyEvent += eventHandler.Callback;

            foreach (var item in items)
            {
               queue.Dequeue();
            }

            A.CallTo(() => eventHandler.Callback(A<object?>._,
A<QueueEmptyEventArgs>._))
                .WhenArgumentsMatch(args =>
args.Get<QueueEmptyEventArgs>(1)?.Message is not null)
                .MustHaveHappened();
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
OnQueueEmpty_WhenQueueCleared_ThenOneCallbackHappened<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
            var eventHandler = A.Fake<ITestEventHandler>();
            queue.QueueEmptyEvent += eventHandler.Callback;

            queue.Clear();

            A.CallTo(() => eventHandler.Callback(A<object?>._,
A<QueueEmptyEventArgs>._))
                .WhenArgumentsMatch(args =>
args.Get<QueueEmptyEventArgs>(1)?.Message is not null)
                .MustHaveHappenedOnceExactly();
        }
```

```
        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
QueueEmptyEventArgs_WhenQueueCleared_ThenMessageIsNotNull<T>(T[]
items)
        {
            var queue = new MyQueue<T>(items);
            var eventHandler = A.Fake<ITestEventHandler>();
            queue.QueueEmptyEvent += eventHandler.Callback;

            queue.Clear();

            A.CallTo(() => eventHandler.Callback(A<object?>._,
A<QueueEmptyEventArgs>._))
                .WhenArgumentsMatch(
                    args => args.Get<QueueEmptyEventArgs>(1)?.Message is not
null)
                .MustHaveHappened();
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
OnPeek_WhenPeekCalledNTimes_ThenNCallbacksHappened<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
            var eventHandler = A.Fake<ITestEventHandler>();
            queue.PeekEvent += eventHandler.Callback;
            var numberOfPeeks = Random.Shared.Next(1, 10);

            for (var i = 0; i < numberOfPeeks; i++)
```

```
            {
                queue.Peek();
            }

            A.CallTo(() => eventHandler.Callback(A<object?>._,
A<PeekEventArgs<T>>._))
                .MustHaveHappened(numberOfPeeks, Times.Exactly);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void PeekEventArgs_WhenPeekCalled_ThenValidArgs<T>(T[]
items)
        {
            var queue = new MyQueue<T>(items);
            var eventHandler = A.Fake<ITestEventHandler>();
            queue.PeekEvent += eventHandler.Callback;

            queue.Peek();

            A.CallTo(() => eventHandler.Callback(A<object?>._,
A<PeekEventArgs<T>>._))
                .WhenArgumentsMatch(args =>
                    (args.Get<PeekEventArgs<T>>(1)?.Data.Equals(items[0]) ??
false)
                    && args.Get<PeekEventArgs<T>>(1)?.Message is not null)
                .MustHaveHappened();
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
```

```
OnPeek_WhenTryPeekCalledNTimes_ThenNCallbacksHappened<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
            var eventHandler = A.Fake<ITestEventHandler>();
            queue.PeekEvent += eventHandler.Callback;
            var numberOfPeeks = Random.Shared.Next(1, 10);

            for (var i = 0; i < numberOfPeeks; i++)
            {
                queue.TryPeek(out _);
            }

            A.CallTo(() => eventHandler.Callback(A<object?>._,
A<PeekEventArgs<T>>._))
                .MustHaveHappened(numberOfPeeks, Times.Exactly);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void PeekEventArgs_WhenTryPeekCalled_ThenValidArgs<T>(T[]
items)
        {
            var queue = new MyQueue<T>(items);
            var eventHandler = A.Fake<ITestEventHandler>();
            queue.PeekEvent += eventHandler.Callback;

            queue.TryPeek(out _);

            A.CallTo(() => eventHandler.Callback(A<object?>._,
A<PeekEventArgs<T>>._))
                .WhenArgumentsMatch(args =>
                    (args.Get<PeekEventArgs<T>>(1)?.Data.Equals(items[0]) ??
```

```
false)
                && args.Get<PeekEventArgs<T>>(1)?.Message is not null)
            .MustHaveHappened();
        }
    }

    public interface ITestEventHandler
    {
        void Callback(object? sender, EventArgs e);
    }

    using WebNetLab1.Collections;
    using Xunit;

    namespace WebNetLab1.Tests;

    public class InitializationTests
    {
        [Fact]
        public void ParameterlessCtor_WhenCalled_ThenEmptyQueue()
        {
            var queue = new MyQueue<int>();

            Assert.Empty(queue);
        }

        [Fact]
        public void
CtorWithSource_WhenNonEmptySource_ThenQueueEqualsSource()
        {
            var source = Enumerable.Range(1, 5);
            var queue = new MyQueue<int>(source);
```

```
            Assert.Equal(source, queue);
        }

        [Fact]
        public void
CtorWithSource_WhenEmptySource_ThenThrowArgumentNullException()
        {
            Assert.Throws<ArgumentNullException>(() =>
            {
                var queue = new MyQueue<int>(null);
            });
        }
    }

    using WebNetLab1.Collections;
    using WebNetLab1.Tests.ClassData;
    using Xunit;

    namespace WebNetLab1.Tests;

    public class PeekTests
    {
        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void Peek_WhenNonEmptyQueue_ThenReturnItem<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);

            var peekedItem = queue.Peek();

            Assert.Equal(items[0], peekedItem);
```

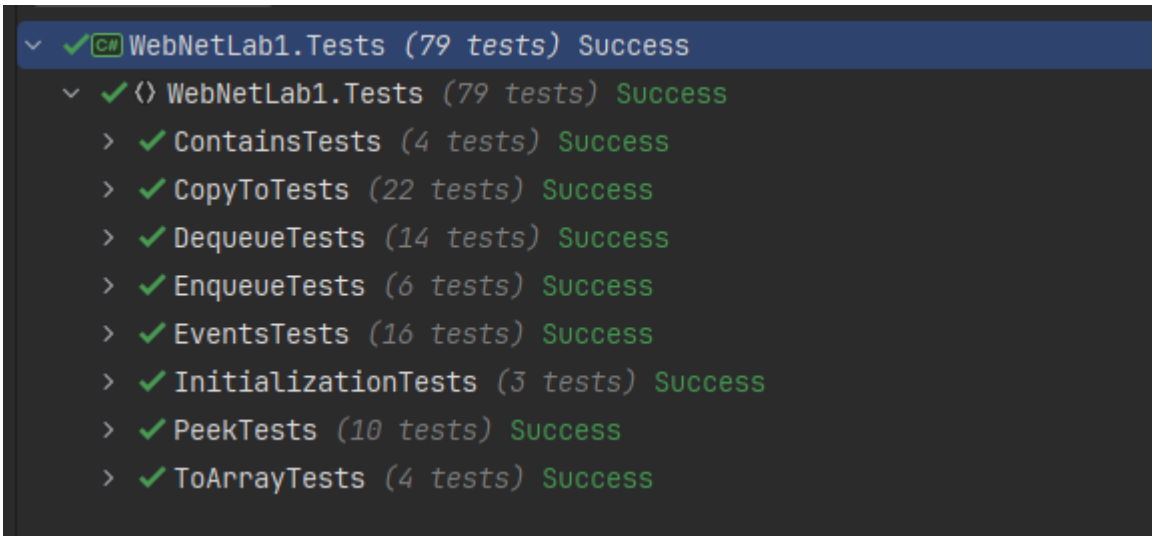
```
        }

        [Fact]
        public void
Peek_WhenEmptyQueue_ThenThrowInvalidOperationException()
        {
            var queue = new MyQueue<int>();

            Assert.Throws<InvalidOperationException>(() => queue.Peek());
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
Peek_WhenQueueCleared_ThenThrowInvalidOperationException<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
            queue.Clear();

            Assert.Throws<InvalidOperationException>(() => queue.Peek());
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
TryPeek_WhenNonEmptyQueue_ThenReturnTrueAndItem<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);

            var peekedResult = queue.TryPeek(out var peekedItem);

            Assert.True(peekedResult);
```

```
            Assert.Equal(items[0], peekedItem);
        }

        [Fact]
        public void TryPeek_WhenEmptyQueue_ThenReturnFalseAndDefault()
        {
            var queue = new MyQueue<int>();

            var peekedResult = queue.TryPeek(out var peekedItem);

            Assert.False(peekedResult);
            Assert.Equal(default, peekedItem);
        }

        [Theory]
        [ClassData(typeof(MultipleItemsQueueData))]
        public void
TryPeek_WhenQueueCleared_ThenReturnFalseAndDefault<T>(T[] items)
        {
            var queue = new MyQueue<T>(items);
            queue.Clear();

            var peekedResult = queue.TryPeek(out var peekedItem);

            Assert.False(peekedResult);
            Assert.Equal(default, peekedItem);
        }
    }

    using WebNetLab1.Collections;
    using WebNetLab1.Tests.ClassData;
    using Xunit;
```

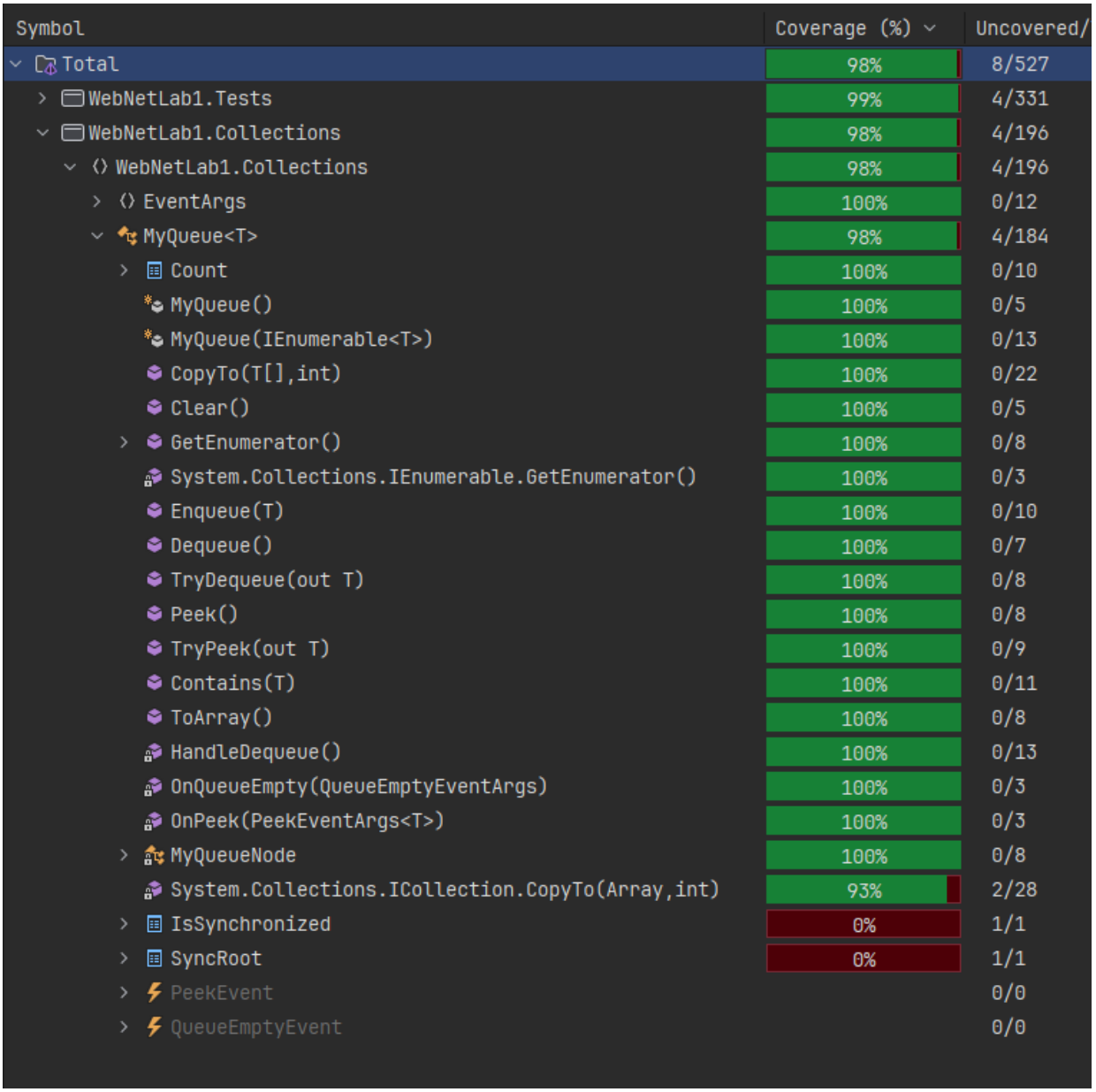
```
namespace WebNetLab1.Tests;

public class ToArrayTests
{
    [Theory]
    [ClassData(typeof(MultipleItemsQueueData))]
    public void
ToArray_WhenEmptyQueue_ThenReturnEmptyArray<T>(T[] items)
    {
        var queue = new MyQueue<T>();

        var array = queue.ToArray();

        Assert.Empty(array);
    }

    [Theory]
    [ClassData(typeof(MultipleItemsQueueData))]
    public void
ToArray_WhenNonEmptyQueue_ThenReturnNewArray<T>(T[] items)
    {
        var queue = new MyQueue<T>(items);

        var array = queue.ToArray();

        Assert.Equal(items, array);
    }
}

using System.Collections;
```

```
namespace WebNetLab1.Tests.ClassData;

public class MultipleItemsQueueData : IEnumerable<object[]>
{
    private static readonly int[] IntArray = { 1, 2, 3, 4, 5, 6, 7 };
    private static readonly string[] StringArray = { "a", "b", "c", "d", "e", "f",
"g" };
    public IEnumerator<object[]> GetEnumerator()
    {
        yield return new[] { IntArray };
        yield return new[] { StringArray };
    }

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

Скріншот запуску модульних тестів:



Скріншот покриття модульних тестів:



Symbol	Coverage (%)	Uncovered/
Total	98%	8/527
WebNetLab1.Tests	99%	4/331
WebNetLab1.Collections	98%	4/196
WebNetLab1.Collections	98%	4/196
EventArgs	100%	0/12
MyQueue<T>	98%	4/184
Count	100%	0/10
MyQueue()	100%	0/5
MyQueue(IEnumerable<T>)	100%	0/13
CopyTo(T[],int)	100%	0/22
Clear()	100%	0/5
GetEnumerator()	100%	0/8
System.Collections.IEnumerable.GetEnumerator()	100%	0/3
Enqueue(T)	100%	0/10
Dequeue()	100%	0/7
TryDequeue(out T)	100%	0/8
Peek()	100%	0/8
TryPeek(out T)	100%	0/9
Contains(T)	100%	0/11
ToArray()	100%	0/8
HandleDequeue()	100%	0/13
OnQueueEmpty(QueueEmptyEventArgs)	100%	0/3
OnPeek(PeekEventArgs<T>)	100%	0/3
MyQueueNode	100%	0/8
System.Collections.ICollection.CopyTo(Array,int)	93%	2/28
IsSynchronized	0%	1/1
SyncRoot	0%	1/1
PeekEvent		0/0
QueueEmptyEvent		0/0

Висновки:

Висновок: на лабораторній роботі №2 вивчили теоретичні та практичні поняття в написанні модульних тестів, застосували знання на практиці, створюючи власні модульні тести для коду колекції, створеної в рамках лабораторної роботи №1. Мною було обрано фреймворк xUnit для написання модульних тестів, також допоміжний фреймфорк FakeItEasy для створення моків і використовувався плагін dotCover для визначення рівня покриття коду модульними тестами.