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

Лабораторна робота № 2

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

Викладач:

Бардін В.

Виконав:

студент 3 курсу

групи ІП-15 ФІОТ

Плугатирьов Д.В.

Київ – 2023

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

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.

Варіант	**Опис узагальненої колекції**	**Функціонал**	**Реалізація**
1	Стек	Див. Stack<T>	Збереження даних за допомогою динамічно зв’язаного списку

В якості фреймворку для створення тестів мною було обрано xUnit. Основною підставою мого вибору стала порада зі сторони викладачів курсу .NET. Ось його переваги:

1. **Проста архітектура**: xUnit має лаконічну та структуровану архітектуру, яка робить його простим у використанні та зрозумілим для новачків;
2. **Параметризовані тести**: можливість створювати тести з різними вхідними даними дозволяє ефективно перевіряти різні сценарії і полегшує тестування;

3. **Атрибути для позначення тестів**: xUnit використовує атрибути для позначення тестів і їх характеристик, забезпечуючи зручний та легкий підхід до організації тестів;
4. **Асинхронне тестування**: фреймворк підтримує асинхронні тести, що дозволяє перевіряти асинхронний код та його взаємодію;
5. **Паралельне виконання тестів**: xUnit автоматично розподіляє тести на виконання в різних потоках, що сприяє швидкому виконанню тестів;
6. **Розширюваність**: фреймворк дозволяє розширювати свою функціональність за допомогою розширень та сторонніх бібліотек.

Програмний код

CustomLinkedListNodeTests.cs

```
namespace CustomCollectionLib.Tests;

public class CustomLinkedListNodeTests
{
    [Fact]
    public void Constructor_InitializesListCorrectly()
    {
        var initialList = new CustomLinkedList<int>();

        var node = new CustomLinkedListNode<int>(initialList, default);

        var actual = node.List;
        Assert.Equal(initialList, actual);
    }

    [Fact]
    public void Constructor_InitializesValueCorrectly()
    {
        var initialList = new CustomLinkedList<int>();
        const int initialValue = 10;

        var node = new CustomLinkedListNode<int>(initialList,
initialValue);

        var actual = node.Value;
        Assert.Equal(initialValue, actual);
    }
```

```
    [Fact]
    public void Clear_SetsReferencesToNull()
    {
        var list = new CustomLinkedList<int>();
        var node = new CustomLinkedListNode<int>(list, default);
        node.Previous = node;
        node.Next = node;

        node.Clear();

        Assert.Null(node.List);
        Assert.Null(node.Next);
        Assert.Null(node.Previous);
    }

    [Fact]
    public void Value_SetsCorrectly()
    {
        var list = new CustomLinkedList<int>();
        var node = new CustomLinkedListNode<int>(list, default);
        const int expected = 10;

        node.Value = expected;

        var actual = node.Value;
        Assert.Equal(expected, actual);
    }

    [Fact]
    public void ValueRef_ReturnsCorrectly()
    {
        var list = new CustomLinkedList<object?>();
        var node = new CustomLinkedListNode<object?>(list, default);
        var expected = new object();

        node.ValueRef = expected;

        var actual = node.ValueRef;
        Assert.Equal(expected, actual);
    }
}
```

CustomLinkedListEnumeratorTests.cs

```
namespace CustomCollectionLib.Tests;

public class CustomLinkedListEnumeratorTests
{
    [Fact]
    public void MoveNext_ReturnsFalseForEmptyList()
    {
        var list = new CustomLinkedList<int>();
```

```
        var enumerator = new CustomLinkedListEnumerator<int>(list);

        var movingResult = enumerator.MoveNext();

        Assert.False(movingResult);
    }

    [Fact]
    public void MoveNext_IteratesOverList()
    {
        var list = new CustomLinkedList<int>();
        list.AddLast(1);
        list.AddLast(2);
        list.AddLast(3);
        var enumerator = new CustomLinkedListEnumerator<int>(list);
        var elementsCount = 0;

        while (enumerator.MoveNext())
        {
            elementsCount++;
        }

        var actual = list.Count;
        Assert.Equal(elementsCount, actual);
    }

    [Fact]
    public void Reset_ResetsEnumerator()
    {
        var list = new CustomLinkedList<int>();
        list.AddLast(1);
        list.AddLast(2);
        var enumerator = new CustomLinkedListEnumerator<int>(list);

        enumerator.MoveNext();
        enumerator.Reset();

        const int expected = 0;
        var actual = enumerator.Current;
        Assert.Equal(expected, actual);
    }
}
```

CustomLinkedListTests.cs

```
namespace CustomCollectionLib.Tests;

public class CustomLinkedListTests
{
    [Fact]
    public void RemoveLast_RemovesCorrectly()
    {
```

```
        var list = new CustomLinkedList<int>();
        const int firstNodeValue = 1;
        list.AddLast(firstNodeValue);
        list.AddLast(2);

        list.RemoveLast();

        var actual = list.Last!.Value;
        Assert.Equal(firstNodeValue, actual);
    }

    [Fact]
    public void
RemoveLast_HeadPreviousIsNull_ThrowsInvalidOperationException()
    {
        var list = new CustomLinkedList<int>();
        list.AddLast(1);
        list.Head!.Previous = null;

        Assert.Throws<InvalidOperationException>(() => list.RemoveLast());
    }

    [Fact]
    public void RemoveLast_HeadIsNull_ThrowsInvalidOperationException()
    {
        var list = new CustomLinkedList<int>();

        Assert.Throws<InvalidOperationException>(() => list.RemoveLast());
    }

    [Fact]
    public void RemoveFirst_RemovesCorrectly()
    {
        var list = new CustomLinkedList<int>();
        const int secondNodeValue = 2;
        list.AddLast(1);
        list.AddLast(secondNodeValue);

        list.RemoveFirst();

        var actual = list.First!.Value;
        Assert.Equal(secondNodeValue, actual);
    }

    [Fact]
    public void RemoveFirst_HeadIsNull_ThrowsInvalidOperationException()
    {
        var list = new CustomLinkedList<int>();

        Assert.Throws<InvalidOperationException>(() =>
list.RemoveFirst());
    }

    [Fact]
```

```
    public void Remove_RemovesSuccessfully()
    {
        var list = new CustomLinkedList<int>();
        const int nodeValueToRemove = 2;
        list.AddLast(nodeValueToRemove);

        var removeResult = list.Remove(nodeValueToRemove);

        var removedNode = list.Find(nodeValueToRemove);
        Assert.Null(removedNode);
        Assert.True(removeResult);
    }

    [Fact]
    public void Remove_HeadIsNull_ThrowsInvalidOperationException()
    {
        var list = new CustomLinkedList<int>();
        const int nodeValueToRemove = 10;

        var removeResult = list.Remove(nodeValueToRemove);

        Assert.False(removeResult);
    }

    [Fact]
    public void Remove_NodeToRemoveIsNull_ReturnsFalse()
    {
        var list = new CustomLinkedList<int>();
        const int nodeValueToRemove = 10;

        var removeResult = list.Remove(nodeValueToRemove);

        Assert.False(removeResult);
    }

    [Theory]
    [InlineData(0, new []{1, 2, 3, 0, 0})]
    [InlineData(1, new []{0, 1, 2, 3, 0})]
    public void CopyTo_CopiesCorrectly(int index, int[] expected)
    {
        var list = new CustomLinkedList<int>();
        list.AddLast(1);
        list.AddLast(2);
        list.AddLast(3);
        var destination = new int[expected.Length];

        list.CopyTo(destination, index);

        Assert.Equal(expected, destination);
    }

    [Fact]
    public void CopyTo_HeadIsNull_InterruptsCopying()
    {
```

```
        var list = new CustomLinkedList<int>();
        var expected = new int[5];
        var destination = new int[expected.Length];
        const int index = 0;

        list.CopyTo(destination, index);

        Assert.Equal(expected, destination);
    }

    [Fact]
    public void CopyTo_IndexIsEqualToArrayLength_ThrowsArgumentException()
    {
        var list = new CustomLinkedList<int>();
        list.AddLast(1);
        var destination = new int[list.Count];
        var index = destination.Length;

        Assert.Throws<ArgumentException>(() => list.CopyTo(destination,
index));
    }

    [Fact]
    public void
CopyTo_IndexIsBiggerThanArrayLength_ThrowsArgumentOutOfRangeException()
    {
        var list = new CustomLinkedList<int>();
        list.AddLast(1);
        var destination = new int[list.Count];
        var index = destination.Length + 1;

        Assert.Throws<ArgumentOutOfRangeException>(() =>
list.CopyTo(destination, index));
    }

    [Fact]
    public void
CopyTo_IndexIsLessThanZero_ThrowsArgumentOutOfRangeException()
    {
        var list = new CustomLinkedList<int>();
        list.AddLast(1);
        var destination = new int[list.Count];
        const int index = -5;

        Assert.Throws<ArgumentOutOfRangeException>(() =>
list.CopyTo(destination, index));
    }

    [Fact]
    public void FindLast_SearchesCorrectly()
    {
        var list = new CustomLinkedList<int>();
        list.AddLast(1);
        const int nodeValueToFind = 3;
```

```
        list.AddLast(nodeValueToFind);
        var expected = list.AddLast(nodeValueToFind);
        list.AddLast(4);

        var actual = list.FindLast(nodeValueToFind);

        Assert.Equal(expected, actual);
    }

    [Fact]
    public void FindLast_HeadIsNull_ReturnsNull()
    {
        var list = new CustomLinkedList<int>();
        const int nodeValueToFind = 10;

        var foundNode = list.FindLast(nodeValueToFind);

        Assert.Null(foundNode);
    }

    [Fact]
    public void Clear_ClearsCorrectly()
    {
        var list = new CustomLinkedList<int>();
        list.AddLast(10);
        list.AddLast(20);

        list.Clear();

        Assert.Null(list.Head);
    }

    [Fact]
    public void AddFirst_HeadIsNull_AddsCorrectly()
    {
        var list = new CustomLinkedList<int>();
        const int nodeValueToAdd = 10;

        var actual = list.AddFirst(nodeValueToAdd);

        var expected = list.Head;
        Assert.Equal(expected, actual);
    }

    [Fact]
    public void AddFirst_AddsNodeAsHead()
    {
        var list = new CustomLinkedList<int>();
        var lastNode = list.AddLast(1);
        const int nodeValueToAdd = 10;

        var actual = list.AddFirst(nodeValueToAdd);

        var expected = list.Head;
```

```
        Assert.Equal(expected, actual);
        Assert.Equal(list.Last, lastNode);
    }

    [Fact]
    public void AddBefore_AddsBeforeHeadCorrectly()
    {
        var list = new CustomLinkedList<int>();
        var nodeToAddBefore = list.AddLast(1);
        const int valueToAddBefore = 10;

        list.AddBefore(nodeToAddBefore, valueToAddBefore);

        var actual = list.Find(valueToAddBefore);
        var expected = list.Head;
        Assert.Equal(expected, actual);
    }

    [Fact]
    public void AddBefore_AddsBeforeExistingNodeCorrectly()
    {
        var list = new CustomLinkedList<int>();
        list.AddLast(1);
        list.AddLast(2);
        var nodeToAddBefore = list.AddLast(3);
        const int nodeValueToAdd = 10;

        list.AddBefore(nodeToAddBefore, nodeValueToAdd);

        var actual = list.Find(nodeValueToAdd);
        var expected = nodeToAddBefore.Previous;
        Assert.Equal(expected, actual);
    }

    [Fact]
    public void Constructor_AddCollectionElementsToLast()
    {
        var list = new List<int> { 1, 2, 3 };

        var customList = new CustomLinkedList<int>(list);

        Assert.All(list, item => Assert.Contains(item, customList));
    }

    [Fact]
    public void
Constructor_PassedNullCollection_ThrowArgumentNullException()
    {
        Assert.Throws<ArgumentNullException>(() => new
CustomLinkedList<int>(null!));
    }

    [Fact]
    public void Find_NodeNotFound_ReturnsNull()
```

```
    {
        var list = new CustomLinkedList<int?>();
        list.AddLast(1);

        var actual = list.Find(null);

        Assert.Null(actual);
    }

    [Fact]
    public void Find_SearchedValueIsNull_ReturnsNodeWithNullValue()
    {
        var list = new CustomLinkedList<int?>();
        list.AddLast(1);
        var expected = list.AddLast(null);

        var actual = list.Find(null);

        Assert.Equal(expected, actual);
    }

    [Fact]
    public void Find_ListIsEmpty_ReturnsNull()
    {
        var list = new CustomLinkedList<int>();
        const int nodeValueToFind = 10;

        var actual = list.Find(nodeValueToFind);

        Assert.Null(actual);
    }

    [Fact]
    public void AddAfter_AddsAfterExistingNodeCorrectly()
    {
        var list = new CustomLinkedList<int>();
        var nodeToAddAfter = list.AddLast(1);
        list.AddLast(2);
        const int nodeValueToAdd = 10;

        list.AddAfter(nodeToAddAfter, nodeValueToAdd);

        var actual = list.Find(nodeValueToAdd);
        var expected = nodeToAddAfter.Next;
        Assert.Equal(expected, actual);
    }
}
```

CustomStackTests.cs

```
namespace CustomCollectionLib.Tests;
```

```
public class CustomStackTests
{
    [Fact]
    public void Push_PushesValueToEmptyStackCorrectly()
    {
        var stack = new CustomStack<int>();
        const int valueToPush = 1;

        stack.Push(valueToPush);

        var actualPushedValue = stack.Peek();
        Assert.Equal(valueToPush, actualPushedValue);
    }

    [Fact]
    public void Pop_RaisesItemPoppedEvent()
    {
        var stack = new CustomStack<int>();
        stack.Push(1);
        var eventRaised = false;
        stack.ItemPopped += (_, _) => eventRaised = true;

        stack.Pop();

        Assert.True(eventRaised, "The ItemPopped event should be
raised.");
    }

    [Fact]
    public void Push_RaisesItemPushedEvent()
    {
        var stack = new CustomStack<int>();
        var eventRaised = false;
        stack.ItemPushed += (_, _) => eventRaised = true;

        stack.Push(1);

        Assert.True(eventRaised, "The ItemPushed event should be
raised.");
    }

    [Fact]
    public void Clear_RaisesStackClearedEvent()
    {
        var stack = new CustomStack<int>();
        stack.Push(1);
        stack.Push(2);
        var eventRaised = false;
        stack.StackCleared += (_, _) => eventRaised = true;

        stack.Clear();

        Assert.True(eventRaised, "The StackCleared event should be
raised.");
```

```
    }

    [Fact]
    public void GetEnumerator_IteratesOverStack()
    {
        var stack = new CustomStack<int>();
        stack.Push(1);
        stack.Push(2);
        stack.Push(3);
        using var enumerator = stack.GetEnumerator();
        var result = new List<int>();

        while (enumerator.MoveNext())
        {
            result.Add(enumerator.Current);
        }

        var expected = new[] { 1, 2, 3 };
        Assert.Equal(expected, result);
    }

    [Fact]
    public void TryPop_ReturnsTrueAndCorrectValue()
    {
        var stack = new CustomStack<int>();
        const int valueToPop = 1;
        stack.Push(valueToPop);

        var isPopped = stack.TryPop(out var poppedValue);

        Assert.Equal(valueToPop, poppedValue);
        Assert.True(isPopped);
    }

    [Fact]
    public void TryPop_StackIsEmpty_ReturnsFalseAndDefaultValue()
    {
        var stack = new CustomStack<int>();

        var isPopped = stack.TryPop(out var poppedValue);

        const int expectedPoppedValue = default;
        Assert.Equal(expectedPoppedValue, poppedValue);
        Assert.False(isPopped);
    }

    [Fact]
    public void Pop_StackIsEmpty_ThrowsInvalidOperationException()
    {
        var stack = new CustomStack<int>();

        Assert.Throws<InvalidOperationException>(() => stack.Pop());
    }
```

```
    [Fact]
    public void Pop_PopsLastElementCorrectly()
    {
        var stack = new CustomStack<int>();
        const int expectedPoppedValue = 3;
        stack.Push(expectedPoppedValue);

        var actualPoppedValue = stack.Pop();

        const int expectedElementsCount = 0;
        var actualElementsCount = stack.Count;
        Assert.Equal(expectedElementsCount, actualElementsCount);
        Assert.Equal(expectedPoppedValue, actualPoppedValue);
    }

    [Fact]
    public void Pop_PopsCorrectly()
    {
        var stack = new CustomStack<int>();
        const int expectedPoppedValue = 3;
        stack.Push(1);
        stack.Push(expectedPoppedValue);

        var poppedValue = stack.Pop();

        const int expectedElementsCount = 1;
        var actualElementsCount = stack.Count;
        Assert.Equal(expectedElementsCount, actualElementsCount);
        Assert.Equal(expectedPoppedValue, poppedValue);
    }

    [Fact]
    public void Clear_ClearsCorrectly()
    {
        var stack = new CustomStack<int>();
        stack.Push(1);
        stack.Push(3);

        stack.Clear();

        const int expected = 0;
        var actual = stack.Count;
        Assert.Equal(expected, actual);
    }

    [Fact]
    public void Contains_WorksCorrectly()
    {
        var stack = new CustomStack<int>();
        const int elementValueToCheck = 2;
        stack.Push(1);
        stack.Push(elementValueToCheck);

        var elementIsFound = stack.Contains(elementValueToCheck);
```

```
        Assert.True(elementIsFound);
    }

    [Theory]
    [InlineData(0, new []{1, 2, 3, 0, 0})]
    [InlineData(1, new []{0, 1, 2, 3, 0})]
    public void CopyTo_CopiesCorrectly(int index, int[] expected)
    {
        var stack = new CustomStack<int>();
        stack.Push(1);
        stack.Push(2);
        stack.Push(3);
        var destination = new int[expected.Length];

        stack.CopyTo(destination, index);

        Assert.Equal(expected, destination);
    }

    [Fact]
    public void Peek_PeeksCorrectly()
    {
        var stack = new CustomStack<int>();
        const int pushedValue = 1;
        stack.Push(pushedValue);

        var peekedValue = stack.Peek();

        Assert.Equal(pushedValue, peekedValue);
    }

    [Fact]
    public void Peek_StackIsEmpty_ThrowsInvalidOperationException()
    {
        var stack = new CustomStack<int>();

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

    [Fact]
    public void TryPeek_PeeksCorrectlyAndReturnsTrue()
    {
        var stack = new CustomStack<int>();
        const int pushedValue = 1;
        stack.Push(pushedValue);

        var isPicked = stack.TryPeek(out var pickedValue);

        Assert.True(isPicked);
        Assert.Equal(pushedValue, pickedValue);
    }

    [Fact]
```

```
    public void TryPeek_StackIsEmpty_PeeksDefaultAndReturnsFalse()
    {
        var stack = new CustomStack<int>();

        var isPicked = stack.TryPeek(out var pickedValue);

        Assert.False(isPicked);
        Assert.Equal(default, pickedValue);
    }
}
```

Ступінь покриття коду тестами

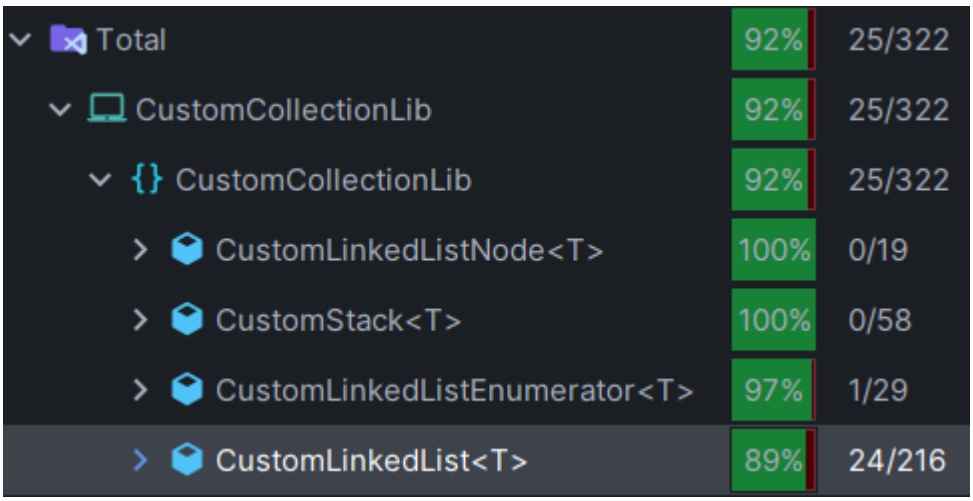

Рисунок 1 – Ступінь покриття коду тестами

Висновок

Під час написання тестів для реалізованої колекції мені вдалося виявити помилки, допущені в момент її розробки. Це доводить користь TDD підходу, дозволяючи виявити потенційні недоліки програми на етапі написання коду. Проте, цей підхід може стати на заваді, якщо дизайн проєкту динамічний, на розробку відведено небагато часу або код не потребує тестування.

Також варто зазначити про важливість існування тестів перед рефакторингом, бо це дозволить набагато швидше зрозуміти розробнику, що внесені ним зміни не впливають на коректність роботи програми.