Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота № 2

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

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

Виконав:
студент групи ІА-13
Прізвище Ім’я.
Дата здачі _____
Захищено з балом _____

Перевірила:
ст. вик. кафедри ІПІ
Крамар Ю. М.

Київ, 2023

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

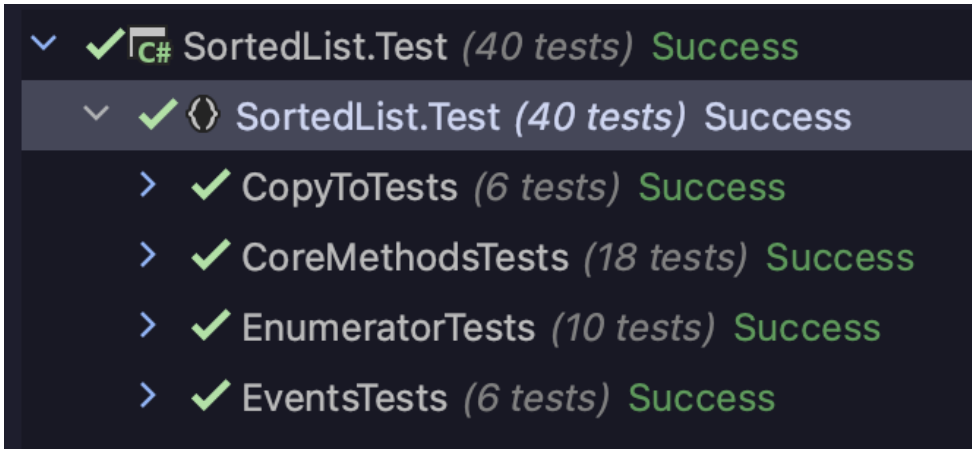
Завдання:

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.

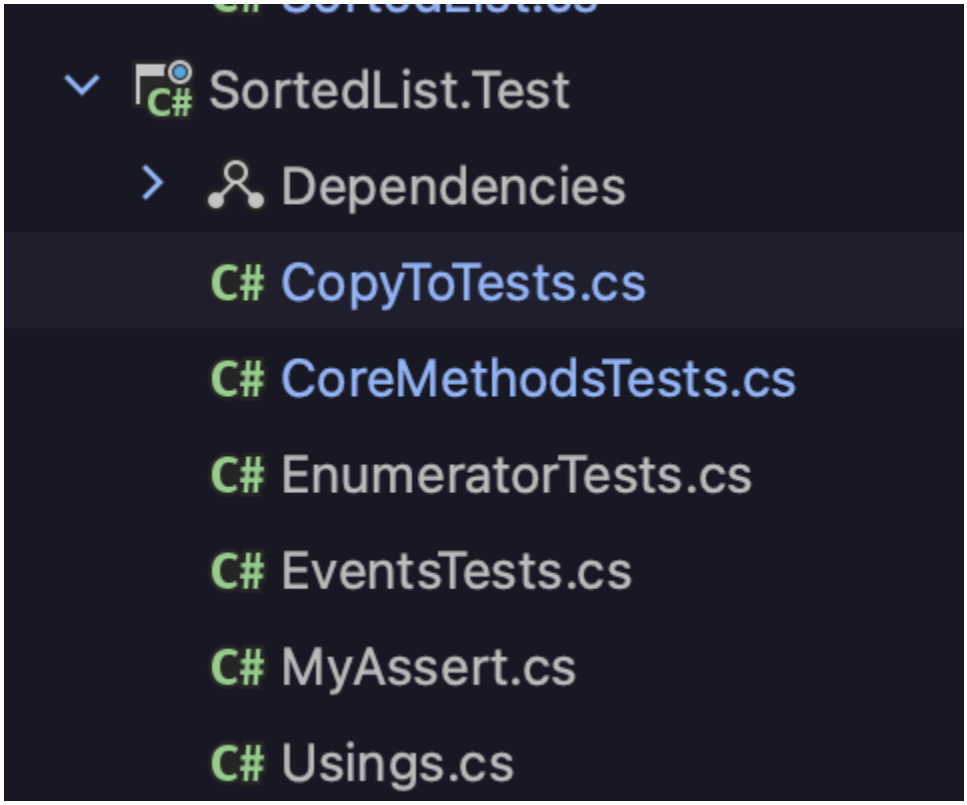
Результат Роботи Програми

Покриття 95 процентів

Увсі тести працюють



Код програми:




```
namespace SortedList.Test;

public class CopyToTests
{
    public static IEnumerable<object[]> CopyToData_ValidArgs()
    {
        yield return new object[]
        {
            new SortedList<int>() { 1, 2, 3, 4 },
            1,
            new int[5],
            new int[] { 0, 1, 2, 3, 4 }
        };
    }

    public static IEnumerable<object[]>
CopyToData_InvalidStartingIndex()
    {
        yield return new object[]
        {
```

```
            new SortedList<int>() { 1, 2, 3, 4 },
            7,
            new int[10],
        };
        yield return new object[]
        {
            new SortedList<int>() { 1, 2, 3, 4 },
            0,
            new int[3],
        };
        yield return new object[]
        {
            new SortedList<int>() {},
            -1,
            new int[] { 1, 2, 3, 4 },
        };
    }

    [Theory]
    [MemberData(nameof(CopyToData_ValidArgs))]
    public void CopyTo_ValidArgs_ExpectedArray<T>(SortedList<T> list,
int index, T[] arr, T[] expectedArr) where T : IComparable<T>
    {
        list.CopyTo(arr, index);

        MyAssert.Equal(arr, expectedArr);
    }

    [Theory]
    [MemberData(nameof(CopyToData_InvalidStartingIndex))]
    public void
CopyTo_InvalidStartingIndex_ArgumentException<T>(SortedList<T>
list, int index, T[] arr) where T : IComparable<T>
    {
        Action action = () => list.CopyTo(arr, index);

        Assert.Throws<ArgumentException>(action);
```

```
    }

    [Fact]
    public void
CopyTo_ArrayLengthListLengthAreZero_ReturnEmptyArray()
    {
        var list = new SortedList<int>();
        var arr = new int[] { };

        list.CopyTo(arr, 0);

        Assert.Empty(arr);
    }

    [Fact]
    public void CopyTo_ListIsEmpty_ArrayIsTheSame()
    {
        var list = new SortedList<int>();
        var arr = new int[] { 1, 2, 3, 4 };

        var cloneArr = (int[])arr.Clone();
        list.CopyTo(arr, 0);

        MyAssert.Equal(cloneArr, arr);
    }
}
namespace SortedList.Test;

public class CoreMethodsTests
{
    public static IEnumerable<object[]>
Data_List_NewItem_ExpectedAfterAdd()
    {
        yield return new object[]
        {
            new SortedList<int> { 1, -100, 200, 87, -600, 3 },
            10,
```

```
            new [] { -600, -100, 1, 3, 10, 87, 200 }
        };
    }

    public static IEnumerable<object[]> Data_List_Duplicate()
    {
        yield return new object[]
        {
            new SortedList<int> { 1, -100, 200, 87, -600, 3, 10 },
            10
        };
    }

    public static IEnumerable<object[]>
Data_List_Duplicate_ExpectedAfterRemove()
    {
        yield return new object[]
        {
            new SortedList<int> { -600, -100, 1, 3, 10, 87, 200 },
            10,
            new [] { -600, -100, 1, 3, 87, 200 },
        };
        yield return new object[]
        {
            new SortedList<int> { -100, 1, 3, 10, -600, 87, 200 },
            -600,
            new [] { -100, 1, 3, 10, 87, 200 },
        };
    }

    public static IEnumerable<object[]> Data_List_NewItem()
    {
        yield return new object[]
        {
            new SortedList<int> { 1, -100, 200, 87, -600, 3 },
            10,
        };
```

```
        yield return new object[]
        {
            new SortedList<int> { 1, -100, 200, 87, -600, 3 },
            1992,
        };
    }

    public static IEnumerable<object[]> Data_NotEmptyList()
    {
        yield return new object[]
        {
            new SortedList<int> { 1, 2, 3, 4 },
        };
    }

    [Theory]
    [MemberData(nameof(Data_List_Duplicate))]
    public void
OrderRetain_RemoveThenAdd_OrderIsRetained<T>(SortedList<T> list, T
item) where T : IComparable<T>
    {
        var copy = list.ToArray();

        list.Remove(item);
        list.Add(item);

        MyAssert.Equal(list, copy);
    }

    [Theory]
    [MemberData(nameof(Data_List_NewItem_ExpectedAfterAdd))]
    public void Add_NoDuplicates_SortedList<T>(SortedList<T> list, T
item, T[] expOutput) where T : IComparable<T>
    {
        list.Add(item);

        MyAssert.Equal(list,expOutput);
```

```
    }

    [Theory]
    [MemberData(nameof(Data_List_Duplicate))]
    public void Add_Duplicates_ArgumentException<T>(SortedList<T>
list, T item) where T : IComparable<T>
    {
        Action action = () => list.Add(item);

        Assert.Throws<ArgumentException>(action);
    }

    [Theory]
    [MemberData(nameof(Data_List_Duplicate_ExpectedAfterRemove))]
    public void Remove_ItemInTheList_True<T>(SortedList<T> list, T
item, IEnumerable<T> expected) where T : IComparable<T>
    {
        bool isDeletionSuccessful = list.Remove(item);

        Assert.True(isDeletionSuccessful);
        MyAssert.Equal(list, expected);
    }

    [Theory]
    [MemberData(nameof(Data_List_NewItem))]
    public void Remove_ItemNotInTheList_False<T>(SortedList<T> list,
T item) where T : IComparable<T>
    {
        bool isDeletionSuccessful = list.Remove(item);

        Assert.False(isDeletionSuccessful);
        MyAssert.DoesNotContain(item, list);
    }

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

```
        var list = new SortedList<string> {};

        string nullString = null;
        Action action = () => list.Add(nullString);

        Assert.Throws<ArgumentNullException>(action);
    }

    [Fact]
    public void Remove_Null_ArgumentNullException()
    {
        var list = new SortedList<string>();

        string nullString = null;
        Action action = () => list.Remove(nullString);

        Assert.Throws<ArgumentNullException>(action);
    }

    [Theory]
    [MemberData(nameof(Data_List_Duplicate))]
    public void Contains_ItemInTheList_True<T>(SortedList<T> list, T
item) where T : IComparable<T>
    {
        bool contains = list.Contains(item);

        Assert.True(contains);
        MyAssert.Contains(item, list);
    }

    [Theory]
    [MemberData(nameof(Data_List_NewItem))]
    public void Contains_ItemNotInTheList_False<T>(SortedList<T>
list, T item) where T : IComparable<T>
    {
        bool contains = list.Contains(item);
```

```
        Assert.False(contains);
        MyAssert.DoesNotContain(item, list);
    }

    [Fact]
    public void Contains_Null_ArgumentNullException()
    {
        var list = new SortedList<string> {};

        string nullString = null;
        Action action = () => list.Contains(nullString);

        Assert.Throws<ArgumentNullException>(action);
    }

    [Theory]
    [MemberData(nameof(Data_NotEmptyList))]
    public void
Count_NotEmptyList_Clear_CountChangesToZero<T>(SortedList<T> list)
where T : IComparable<T>
    {
        var EMPTY_LIST_COUNT = 0;
        var countBeforeClear = list.Count;

        list.Clear();
        var countAfterClear = list.Count;

        Assert.NotEqual(EMPTY_LIST_COUNT, countBeforeClear);
        Assert.Equal(EMPTY_LIST_COUNT, countAfterClear);
    }

    [Theory]
    [MemberData(nameof(Data_List_Duplicate))]
    public void
Count_NotEmptyList_RemoveItem_CountDecreasesByOne<T>(SortedList<T>
list, T item) where T : IComparable<T>
    {
```

```
        var countBeforeRemove = list.Count;
        var EXPECTED_DECREASE = 1;

        list.Remove(item);
        var actualDecrease = countBeforeRemove - list.Count;

        Assert.Equal(EXPECTED_DECREASE, actualDecrease);
    }

    [Theory]
    [MemberData(nameof(Data_List_NewItem))]
    public void
Count_NotEmptyList_AddItem_CountIncreasesByOne<T>(SortedList<T>
list, T item) where T : IComparable<T>
    {
        var countBeforeRemove = list.Count;
        var EXPECTED_INCREASE = 1;

        list.Add(item);
        var actualIncrease = list.Count - countBeforeRemove;

        Assert.Equal(EXPECTED_INCREASE, actualIncrease);
    }

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

        var isReadOnly = list.IsReadOnly;

        Assert.False(isReadOnly);
    }
}
namespace SortedList.Test;

public class EnumeratorTests
```

```
{
    public static IEnumerable<object[]>
IteratorData_ListWithExpectedSequence()
    {
        yield return new object[]
        {
            new SortedList<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
            new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
        };
    }

    [Theory]
    [MemberData(nameof(IteratorData_ListWithExpectedSequence))]
    public void
Enumerator_SortedListItems_ReturnsItemsInExpectedOrder<T>(SortedLis
t<T> list, List<T> expected) where T : IComparable<T>
    {
        var expectedEnumerator = expected.GetEnumerator();

        var actualEnumerator = list.GetEnumerator();

        MyAssert.Equal(expectedEnumerator, actualEnumerator);
    }

    [Theory]
    [MemberData(nameof(IteratorData_ListWithExpectedSequence))]
    public void
ReversedEnumerator_SortedListItems_ReturnsItemsInExpectedOrder<T>(S
ortedList<T> list, List<T> expected) where T : IComparable<T>
    {
        expected.Reverse();
        var expectedEnumerator = expected.GetEnumerator();

        var actualEnumerator = list.Reversed();

        MyAssert.Equal(expectedEnumerator, actualEnumerator);
    }
```

```
    [Fact]
    public void Enumerator_Reset_NotSupportedException()
    {
        var list = new SortedList<int>();

        var enumerator = list.GetEnumerator();
        var action = () => enumerator.Reset();

        Assert.Throws<NotSupportedException>(action);
    }

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

        var enumerator = list.Reversed();
        var action = () => enumerator.Reset();

        Assert.Throws<NotSupportedException>(action);
    }

    [Fact]
    public void
Enumerator_MoveNext_GetEnumeratorWhenCollectionIsEmpty_MoveNextRetu
rnsFalse()
    {
        var list = new SortedList<int>();

        var enumerator = list.GetEnumerator();
        var hasNext = enumerator.MoveNext();

        Assert.False(hasNext);
    }

    [Fact]
```

```
    public void ReversedEnumerator_MoveNext_GetEnumeratorWhenCollectionIsEmpty_MoveNextReturnsFalse()
    {
        var list = new SortedList<int>();

        var enumerator = list.Reversed();
        var hasNext = enumerator.MoveNext();

        Assert.False(hasNext);
    }

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

        var enumerator = list.GetEnumerator();
        object Action() => enumerator.Current;

        Assert.Throws<InvalidOperationException>(Action);
    }

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

        var enumerator = list.Reversed();
        var current = enumerator.Current;

        Assert.Equal(default,current);
    }
```

```
    [Fact]
    public void
ReversedEnumerator_Current_EnumerationIsAlreadyFinished_ReturnsFirs
tItemInCollection()
    {
        var list = new SortedList<int>() { 1, 2, 3, 4 };
        var FIRST_ITEM_IN_LIST = 1;

        var enumerator = list.Reversed();
        while (enumerator.MoveNext()) { }

        var current = enumerator.Current;

        Assert.Equal(FIRST_ITEM_IN_LIST,current);
    }

    [Fact]
    public void
Enumerator_Current_EnumerationIsAlreadyFinished_InvalidOperationExc
eption()
    {
        var list = new SortedList<int>() { 1, 2, 3, 4 };

        var enumerator = list.GetEnumerator();
        while (enumerator.MoveNext()) { }
        object Action() => enumerator.Current;

        Assert.Throws<InvalidOperationException>(Action);
    }
}
namespace SortedList.Test;

public class EventsTests
{
    [Fact]
```

```
    public void ItemAdded_ItemAdded_EventInvoked()
    {
        var list = new SortedList<int>();


        var isInvoked = false;
        list.ItemAdded += (o, e) => isInvoked = true;
        list.Add(default);

        Assert.True(isInvoked);
    }

    [Fact]
    public void ItemRemoved_ItemRemovedReturnsTrue_EventInvoked()
    {
        var list = new SortedList<int>() {default};

        var isInvoked = false;
        list.ItemRemoved += (o, e) => isInvoked = true;
        list.Remove(default);

        Assert.True(isInvoked);
    }

    [Fact]
    public void
ItemRemoved_ItemRemovedReturnsFalse_EventIsNotInvoked()
    {
        var list = new SortedList<int>() {};

        var isInvoked = false;
        list.ItemRemoved += (o, e) => isInvoked = true;
        list.Remove(default);

        Assert.False(isInvoked);
    }
```

```
    [Fact]
    public void ListCleared_RemoveLastElement_EventIsInvoked()
    {
        var list = new SortedList<int> {default};

        var isInvoked = false;
        list.ListCleared += (o, e) => isInvoked = true;
        list.Remove(default);

        Assert.True(isInvoked);
    }
    [Fact]
    public void
ListCleared_ListClearedWhileBeingNotEmpty_EventIsInvoked()
    {
        var list = new SortedList<int> {default};

        var isInvoked = false;
        list.ListCleared += (o, e) => isInvoked = true;
        list.Clear();

        Assert.True(isInvoked);
    }

    [Fact]
    public void
ListCleared_ListClearedWhileBeingEmpty_EventIsNotInvoked()
    {
        var list = new SortedList<int> { };

        var isInvoked = false;
        list.ListCleared += (o, e) => isInvoked = true;
        list.Clear();

        Assert.False(isInvoked);
    }
}
```

```
namespace SortedList.Test;

public static class MyAssert
{
   public static void Equal<T>(IEnumerable<T> expected,
IEnumerable<T> actual) where T : IComparable<T>
   {
       var expEnum = expected.GetEnumerator();
       var actEnum = actual.GetEnumerator();

       while (actEnum.MoveNext() | expEnum.MoveNext())
       {
           Assert.Equal(0,
expEnum.Current.CompareTo(actEnum.Current));
       }

       expEnum.Dispose();
       actEnum.Dispose();
   }

   public static void Equal<T>(IEnumerator<T> expected,
IEnumerator<T> actual) where T : IComparable<T>
   {
       while (actual.MoveNext() | expected.MoveNext())
       {
           Assert.Equal(0,
expected.Current.CompareTo(actual.Current));
       }

       expected.Dispose();
       actual.Dispose();
   }

   public static void Contains<T>(T item, SortedList<T>list) where T
: IComparable<T>
   {
       var contains = list.Any(i => item.CompareTo(i) == 0);
```

```
        Assert.True(contains);
    }

    public static void DoesNotContain<T>(T item ,SortedList<T>list)
where T : IComparable<T>
    {
        foreach (var i in list)
        {
            Assert.False(item.CompareTo(i) == 0);
        }
    }
}
global using Xunit;
global using SortedList;
```

Висновок:

Виконуючи лабороторну роботу я написав тести для своеї колекції за допомогою XUnit та за допомогою dotCover дослідив процент покриття тестами кода моеї колекції