Національний технічний університет України «КПІ»

Факультет інформатики та обчислювальної техніки

Кафедра Інформаційних систем та технологій

Лабораторна робота №2

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

на тему: «Ознайомлення з засобами та практиками

модульного тестування»

Виконав:

студент гр. ІП-11

Веремчук Ігор

Викладач:

Бардін В.

2023 рік

**Мета**: навчитися створювати модульні тести для

вихідного коду розроблювального програмного забезпечення.

**Завдання:**

. Додати до проекту власної узагальненої колекції (застосувати

виконану лабораторну роботу No1) проект модульних тестів,

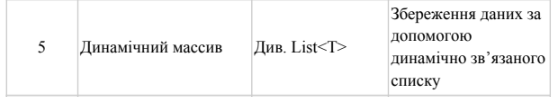
використовуючи певний фреймворк (Nunit, Xunit, тощо).

2. Розробити модульні тести для функціоналу колекції.

3. Дослідити ступінь покриття модульними тестами вихідного коду

колекції, використовуючи, наприклад, засіб AxoCover.

**Варіант 5:**



**Посилання на код GitHub:**

https://github.com/vrmchk/.NET  
Код основних класів:

**NodeEnumerator:**

public class NodeEnumerator<T> : IEnumerator<T>  
{  
 private int \_position = -1;  
 private readonly Node<T>? \_rootNode;  
 private Node<T>? \_currentNode;  
  
 public NodeEnumerator(Node<T>? rootNode)  
 {  
 \_rootNode = rootNode;  
 \_currentNode = \_rootNode;  
 }  
  
 public T Current => \_currentNode != null ? \_currentNode.Value : default;  
  
 object IEnumerator.Current => Current;  
  
 public bool MoveNext()  
 {  
 if (\_currentNode == null)  
 return false;  
  
 if (\_position == -1)  
 {  
 \_position++;  
 return true;  
 }  
  
 if (\_currentNode.Next == null)  
 return false;  
  
 \_currentNode = \_currentNode.Next;  
 \_position++;  
 return true;  
 }  
  
 public void Reset()  
 {  
 \_currentNode = \_rootNode;  
 \_position = -1;  
 }  
  
 public void Dispose() { }  
}

**MyList:**

public class MyList<T> : IList<T>, ICloneable  
{  
 private Node<T>? \_head;  
  
 public int Count { get; private set; }  
 public bool IsReadOnly => false;  
 private bool ShouldNotify => CollectionChanged != null;  
  
 public T this[int index]  
 {  
 get => GetNode(index).Value;  
 set  
 {  
 if (!ShouldNotify)  
 {  
 GetNode(index).Value = value;  
 return;  
 }  
  
 var node = GetNode(index);  
  
 var oldItem = node.Value;  
 var oldCollection = (MyList<T>)Clone();  
  
 node.Value = value;  
  
 OnCollectionChanged(CollectionChangeType.**Update**, oldItem, value, oldCollection);  
 }  
 }  
  
 public event Action<CollectionChangedEventArgs<T>>? CollectionChanged;  
  
 public void Add(T item)  
 {  
 if (!ShouldNotify)  
 {  
 PerformAdd();  
 return;  
 }  
  
 var oldCollection = (MyList<T>)Clone();  
 PerformAdd();  
 OnCollectionChanged(CollectionChangeType.**Add**, newItem: item, oldCollection: oldCollection);  
 return;  
  
 void PerformAdd()  
 {  
 if (\_head == null)  
 {  
 \_head = new Node<T>(item);  
 Count++;  
 return;  
 }  
  
 var current = \_head;  
 while (current is { Next: not null })  
 {  
 current = current.Next;  
 }  
  
 current.Next = new Node<T>(item);  
 Count++;  
 }  
 }  
  
 public void Clear()  
 {  
 if (!ShouldNotify)  
 {  
 PerformClear();  
 return;  
 }  
  
 var oldCollection = (MyList<T>)Clone();  
 PerformClear();  
  
 OnCollectionChanged(CollectionChangeType.**Clear**, oldCollection: oldCollection);  
 return;  
  
 void PerformClear()  
 {  
 \_head = null;  
 Count = 0;  
 }  
 }  
  
 public bool Contains(T item)  
 {  
 return IndexOf(item) >= 0;  
 }  
  
 public void CopyTo(T[] array, int arrayIndex)  
 {  
 if (array == null)  
 throw new ArgumentNullException(nameof(array));  
  
 if (arrayIndex < 0)  
 throw new ArgumentOutOfRangeException(nameof(arrayIndex));  
  
 if (array.Length - arrayIndex < Count)  
 throw new ArgumentException("Destination array was not long enough.");  
  
 var current = \_head;  
 while (current != null)  
 {  
 array[arrayIndex++] = current.Value;  
 current = current.Next;  
 }  
 }  
  
 public bool Remove(T item)  
 {  
 var current = \_head;  
 Node<T>? previous = null;  
 while (current != null)  
 {  
 if (current.Value != null && current.Value.Equals(item))  
 {  
 if (!ShouldNotify)  
 {  
 Remove(previous, current);  
 return true;  
 }  
  
 var oldCollection = (MyList<T>)Clone();  
 Remove(previous, current);  
  
 OnCollectionChanged(CollectionChangeType.**Remove**, item, oldCollection: oldCollection);  
 return true;  
 }  
  
 previous = current;  
 current = current.Next;  
 }  
  
 return false;  
 }  
  
 public int IndexOf(T item)  
 {  
 var current = \_head;  
 var index = 0;  
 while (current != null)  
 {  
 if (current.Value != null && current.Value.Equals(item))  
 return index;  
  
 current = current.Next;  
 index++;  
 }  
  
 return -1;  
 }  
  
 public void Insert(int index, T item)  
 {  
 if (!ShouldNotify)  
 {  
 PerformInsert();  
 return;  
 }  
  
 var oldCollection = (MyList<T>)Clone();  
 PerformInsert();  
 OnCollectionChanged(CollectionChangeType.**Add**, newItem: item, oldCollection: oldCollection);  
 return;  
  
 void PerformInsert()  
 {  
 if (index == 0)  
 {  
 \_head = new Node<T>(item) { Next = \_head };  
 Count++;  
 return;  
 }  
  
 if (index == Count)  
 {  
 Add(item);  
 return;  
 }  
  
 var current = GetNode(index - 1);  
 var newNode = new Node<T>(item) { Next = current.Next };  
 current.Next = newNode;  
 Count++;  
 }  
 }  
  
 public void RemoveAt(int index)  
 {  
 var previous = index != 0 ? GetNode(index - 1) : null;  
 var toRemove = previous?.Next ?? \_head;  
 if (toRemove == null)  
 return;  
  
 if (!ShouldNotify)  
 {  
 Remove(previous, toRemove);  
 return;  
 }  
  
 var oldCollection = (MyList<T>)Clone();  
 var oldItem = toRemove.Value;  
 Remove(previous, toRemove);  
 OnCollectionChanged(CollectionChangeType.**Remove**, oldItem, oldCollection: oldCollection);  
 }  
  
 public IEnumerator<T> GetEnumerator()  
 {  
 return new NodeEnumerator<T>(\_head);  
 }  
  
 IEnumerator IEnumerable.GetEnumerator()  
 {  
 return GetEnumerator();  
 }  
  
 public object Clone()  
 {  
 return new MyList<T> { \_head = (Node<T>?)\_head?.Clone() };  
 }  
  
 private Node<T> GetNode(int index)  
 {  
 if (index < 0 || index >= Count)  
 throw new IndexOutOfRangeException();  
  
 var current = \_head;  
 for (int i = 0; i < index; i++)  
 {  
 if (current == null)  
 throw new IndexOutOfRangeException();  
  
 current = current.Next;  
 }  
  
 if (current == null)  
 throw new IndexOutOfRangeException();  
  
 return current;  
 }  
  
 private void Remove(Node<T>? previous, Node<T> current)  
 {  
 if (previous == null)  
 \_head = current.Next;  
 else  
 previous.Next = current.Next;  
 Count--;  
 }  
  
 private void OnCollectionChanged(CollectionChangeType changeType,  
 T? oldItem = default,  
 T? newItem = default,  
 ICollection<T>? oldCollection = null)  
 {  
 CollectionChanged?.Invoke(new CollectionChangedEventArgs<T>  
 {  
 ChangeType = changeType,  
 OldItem = oldItem,  
 NewItem = newItem,  
 OldCollection = oldCollection ?? new List<T>(),  
 NewCollection = this  
 });  
 }  
}

**Tests:**

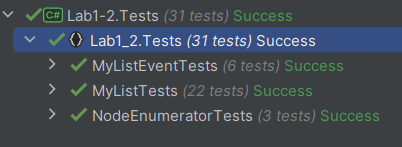
public abstract class MyListTestsBase  
{  
 protected static readonly int[] TestValues = Enumerable.Range(0, 5).ToArray();  
}

public class MyListTests : MyListTestsBase  
{  
 [Fact]  
 public void Add\_ShouldAddItemToList()  
 {  
 *// Arrange* var list = new MyList<int>();  
  
 *// Act* list.Add(1);  
  
 *// Assert* Assert.Contains(1, list);  
 }  
  
 [Fact]  
 public void Clear\_ShouldClearTheList()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* list.Clear();  
  
 *// Assert* Assert.Empty(list);  
 }  
  
 [Fact]  
 public void Contains\_WhenItemInTheList\_ShouldReturnTrue()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *//Act* var contains = list.Contains(TestValues.First());  
  
 *// Assert* Assert.True(contains);  
 }  
  
 [Fact]  
 public void Contains\_WhenItemNotInTheList\_ShouldReturnFalse()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *//Act* var contains = list.Contains(TestValues.Length + 1);  
  
 *// Assert* Assert.False(contains);  
 }  
  
 [Fact]  
 public void CopyTo\_ShouldCopyItemsToArray()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
 var array = new int[list.Count];  
  
 *// Act* list.CopyTo(array, 0);  
  
 *// Assert* Assert.Equal(TestValues, array);  
 }  
  
 [Fact]  
 public void CopyTo\_WhenNotEnoughSpaceInArray\_ShouldThrowException()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 var array = new int[list.Count - 1];  
  
 *// Act & Assert* Assert.Throws<ArgumentException>(() => list.CopyTo(array, 0));  
 }  
   
 [Fact]  
 public void CopyTo\_WhenArrayIsNull\_ShouldThrowException()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act & Assert* Assert.Throws<ArgumentNullException>(() => list.CopyTo(null!, 0));  
 }  
   
 [Fact]  
 public void CopyTo\_WhenNegativeIndex\_ShouldThrowException()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 var array = new int[list.Count];  
  
 *// Act & Assert* Assert.Throws<ArgumentOutOfRangeException>(() => list.CopyTo(array, -1));  
 }  
  
 [Fact]  
 public void Indexer\_ShouldReturnItemAtIndex()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act & Assert* for (int i = 0; i < list.Count; i++)  
 {  
 Assert.Equal(TestValues[i], list[i]);  
 }  
 }  
  
 [Fact]  
 public void Indexer\_ShouldSetItemAtIndex()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* list[1] = 100;  
  
 *// Assert* Assert.Equal(100, list[1]);  
 }  
  
 [Fact]  
 public void Indexer\_WhenInvalidIndex\_ShouldThrowException()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act & Assert* Assert.Throws<IndexOutOfRangeException>(() => list[-1]);  
 Assert.Throws<IndexOutOfRangeException>(() => list[TestValues.Length] = 1);  
 }  
  
 [Fact]  
 public void Remove\_ShouldRemoveItemFromList()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* var result = list.Remove(TestValues.First());  
  
 *// Assert* Assert.True(result);  
 Assert.Equal(TestValues.Length - 1, list.Count);  
 Assert.DoesNotContain(TestValues.First(), list);  
 }  
  
 [Fact]  
 public void Remove\_WhenItemNotInList\_ShouldReturnFalse()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* var result = list.Remove(TestValues.Length + 1);  
  
 *// Assert* Assert.False(result);  
 Assert.Equal(TestValues.Length, list.Count);  
 }  
  
 [Fact]  
 public void GetEnumerator\_ShouldReturnEnumerator()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* var enumerator = list.GetEnumerator();  
  
 *// Assert* Assert.NotNull(enumerator);  
 }  
  
 [Fact]  
 public void IndexOf\_WhenItemInList\_ShouldReturnIndexOfItem()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* int index = list.IndexOf(TestValues[1]);  
  
 *// Assert* Assert.Equal(1, index);  
 }  
  
 [Fact]  
 public void IndexOf\_WhenItemNotInList\_ShouldReturnNegativeOne()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* int index = list.IndexOf(TestValues.Length + 1);  
  
 *// Assert* Assert.Equal(-1, index);  
 }  
  
 [Fact]  
 public void RemoveAt\_WhenValidIndex\_ShouldRemoveItemAtIndex()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* list.RemoveAt(1);  
  
 *// Assert* Assert.Equal(TestValues.Length - 1, list.Count);  
 Assert.DoesNotContain(TestValues[1], list);  
 }  
  
 [Fact]  
 public void RemoveAt\_WhenInvalidIndex\_ShouldThrowException()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act & Assert* Assert.Throws<IndexOutOfRangeException>(() => list.RemoveAt(-1));  
 Assert.Throws<IndexOutOfRangeException>(() => list.RemoveAt(TestValues.Length));  
 }  
  
 [Fact]  
 public void Insert\_WhenValidIndex\_ShouldInsertItemAtIndex()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* list.Insert(1, 100);  
  
 *// Assert* Assert.Equal(TestValues.Length + 1, list.Count);  
 Assert.Contains(100, list);  
 }  
  
 [Fact]  
 public void Insert\_WhenEmptyList\_ShouldInsertItemAtStart()  
 {  
 *// Arrange* var list = new MyList<int>();  
  
 *// Act* list.Insert(0, 100);  
  
 *// Assert* Assert.Single(list);  
 Assert.Contains(100, list);  
 }  
  
 [Fact]  
 public void Insert\_WhenInsertingToEnd\_ShouldInsertItemAtEnd()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* list.Insert(TestValues.Length, 100);  
  
 *// Assert* Assert.Equal(TestValues.Length + 1, list.Count);  
 Assert.Contains(100, list);  
 }  
  
 [Fact]  
 public void Insert\_WhenInvalidIndex\_ShouldThrowException()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act & Assert* Assert.Throws<IndexOutOfRangeException>(() => list.Insert(-1, 1));  
 Assert.Throws<IndexOutOfRangeException>(() => list.Insert(TestValues.Length + 1, 1));  
 }  
}

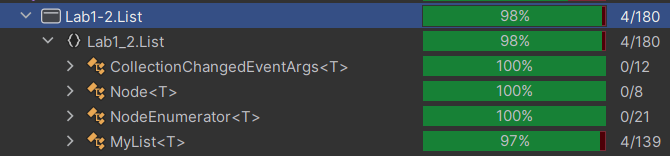
public class NodeEnumeratorTests : MyListTestsBase  
{  
 [Fact]  
 public void MoveNext\_ShouldEnumerateItemsInOrder()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
  
 *// Act* using var enumerator = list.GetEnumerator();  
 var items = new List<int>();  
 while (enumerator.MoveNext())  
 {  
 items.Add(enumerator.Current);  
 }  
  
 *// Assert* Assert.Equal(TestValues, items);  
 }  
  
 [Fact]  
 public void MoveNext\_WhenEmptyList\_ShouldReturnFalse()  
 {  
 *// Arrange* var list = new MyList<int>();  
  
 *// Act* using var enumerator = list.GetEnumerator();  
  
 *// Assert* Assert.False(enumerator.MoveNext());  
 }  
  
 [Fact]  
 public void Reset\_ShouldResetToStart()  
 {  
 *// Arrange* var list = new MyList<int>(TestValues);  
 using var enumerator = list.GetEnumerator();  
  
 *// Act* enumerator.MoveNext();  
 enumerator.Reset();  
 enumerator.MoveNext();  
  
 *// Assert* Assert.Equal(TestValues.First(), enumerator.Current);  
 }  
}

public class MyListEventTests : MyListTestsBase  
{  
 [Fact]  
 public void CollectionChanged\_WhenItemAdded\_ShouldBeCalled()  
 {  
 var newItem = 100;  
 TestCollectionChangedEvent(CollectionChangeType.**Add**,  
 list => list.Add(newItem),  
 eventArgs => Assert.Equal(newItem, eventArgs.NewItem));  
 }  
  
 [Fact]  
 public void CollectionChanged\_WhenItemInserted\_ShouldBeCalled()  
 {  
 var newItem = 100;  
 TestCollectionChangedEvent(CollectionChangeType.**Add**,  
 list => list.Insert(1, newItem),  
 eventArgs => Assert.Equal(newItem, eventArgs.NewItem));  
 }  
  
 [Fact]  
 public void CollectionChanged\_WhenCleared\_ShouldBeCalled()  
 {  
 TestCollectionChangedEvent(CollectionChangeType.**Clear**,  
 list => list.Clear(),  
 eventArgs => Assert.Empty(eventArgs.NewCollection));  
 }  
  
 [Fact]  
 public void CollectionChanged\_WhenSetWithIndexer\_ShouldBeCalled()  
 {  
 var oldItem = TestValues[1];  
 var newItem = 100;  
  
 TestCollectionChangedEvent(CollectionChangeType.**Update**,  
 list => list[1] = newItem,  
 eventArgs =>  
 {  
 Assert.Equal(oldItem, eventArgs.OldItem);  
 Assert.Equal(newItem, eventArgs.NewItem);  
 });  
 }  
  
 [Fact]  
 public void CollectionChanged\_WhenItemRemoved\_ShouldBeCalled()  
 {  
 var oldItem = TestValues.First();  
 TestCollectionChangedEvent(CollectionChangeType.**Remove**,  
 list => list.Remove(oldItem),  
 eventArgs => Assert.Equal(oldItem, eventArgs.OldItem));  
 }  
  
 [Fact]  
 public void CollectionChanged\_WhenItemRemoveAt\_ShouldBeCalled()  
 {  
 var oldItem = TestValues[1];  
 TestCollectionChangedEvent(CollectionChangeType.**Remove**,   
 list => list.RemoveAt(1),  
 eventArgs => Assert.Equal(oldItem, eventArgs.OldItem));  
 }  
  
 private void TestCollectionChangedEvent(CollectionChangeType changeType,  
 Action<MyList<int>> listAction,  
 Action<CollectionChangedEventArgs<int>> assertAction)  
 {  
 *//Arrange* var list = new MyList<int>(TestValues);  
 var oldList = (MyList<int>)list.Clone();  
   
 CollectionChangedEventArgs<int>? eventArgs = null;  
  
 list.CollectionChanged += args => eventArgs = args;  
  
 *//Act* listAction(list);  
  
 *//Assert* Assert.NotNull(eventArgs);  
 Assert.Equal(eventArgs.ChangeType, changeType);  
 Assert.Equal(eventArgs.OldCollection, oldList);  
 Assert.Equal(eventArgs.NewCollection, list);  
 assertAction(eventArgs!);  
 }  
}

**Екран виконання тестів:**



**Покриття тестами:**



**Висновки:** під час виконання лабораторної роботи я навчився писати модульні тести за допомогою фреймворку xunit та перевіряти покриття проекту тестами за допомогою dotCover.