Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Владимирский государственный университет

имени Александра Григорьевича и Николая Григорьевича Столетовых»

(ВлГУ)

Кафедра информационных систем и программной инженерии

Лабораторная работа №4

по дисциплине "Технологии программирования"

ТЕМА РАБОТЫ:

Разработка Unit-тестов с помощью фреймворка XUnit

Выполнил:

студент гр. ПРИ-120

Парахин К.В.

Принял:

Ассистент кафедры ИСПИ

Данилов В.В.

Владимир 2022 г.

Цель работы:

Изучить возможности Unit тестирования .NET приложений с использованием framework XUnit

Выполненине работы:

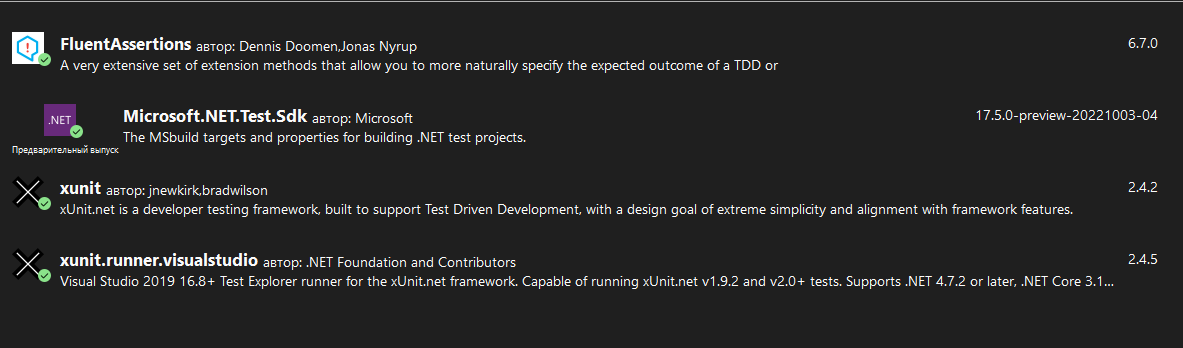


Рисунок 1. Установка пакетов NuGet

В данной работе я проведу примеры тестирования различных методов работы с очередью. Для этого я сам реализую обобщенный класс MyQueue<T> и напишу для него такие базовые методы, как: Dequeue, Enqueue, а также подсчет кол-ва элементов (Count) и базовый итератор с методами MoveNext, Current и Reset.

Листинг программы:

namespace QueueChecking

{

public class TestClass

{

public int Value { get; set; }

}

public class MyQueue<T> : IQueue<T>

{

private static T[] \_array;

private int \_head;

private int \_tail;

private int \_size;

private static int \_version;

public MyQueue()

{

\_array = Array.Empty<T>();

\_head = 0;

\_tail = 0;

\_size = 0;

\_version = 0;

}

public int Count { get { return \_size; } private set { } }

public T Dequeue()

{

if (\_size == 0)

throw new InvalidOperationException(nameof(\_size));

T element = \_array[\_tail++];

SetCapacity(--\_size);

\_head--;

return element;

}

public void Enqueue(T item)

{

SetCapacity(++\_size);

\_array[++\_head] = item;

}

private void SetCapacity(int capacity)

{

T[] newarray = new T[capacity];

Array.Copy(\_array, \_tail, newarray, 0, capacity);

\_array = newarray;

\_version++;

}

public Enumerator GetEnumerator()

{

return new Enumerator();

}

public class Enumerator

{

private int \_position;

private readonly int \_curVersion;

public Enumerator()

{

\_position = -1;

\_curVersion = \_version;

}

public bool MoveNext()

{

if (\_curVersion != \_version)

throw new InvalidOperationException(nameof(\_version));

return (++\_position < \_array.Length);

}

public void Reset()

{

if (\_curVersion != \_version)

throw new InvalidOperationException(nameof(\_version));

\_position = -1;

}

public object Current

{

get { return \_array[\_position]!; }

}

}

}

public interface IQueue<T>

{

public T Dequeue();

public void Enqueue(T item);

public int Count { get; }

}

}

Далее приступим к написанию Unit-тестов (5 методов) по методологии цикла TDD:

1. Метод, в котором мы попытаемся вызвать метод Dequeue (который берет и удаляет элемент из начала очереди) для пустой очереди.

На 1 этапе мы напишем вот такой код тестового метода:

[Fact]

public void QueueDequeueEmpty()

{

var queue = new MyQueue<TestClass>();

var ex = Record.Exception(() => queue.Dequeue());

ex.Should().NotBeNull().And.BeOfType<InvalidOperationException>();

}

То есть мы предполагаем логически, что будет выброшено исключение при попытке что-то взять из пустой очереди, но пока этот тест не будет пройден.

На 2 этапе реализуем метод T Dequeue для нашего класса MyQueue, добавив условие проверки величины очереди и выброс исключения

if (\_size == 0)

throw new InvalidOperationException(nameof(\_size));

Затем проверяем работу теста и видим, что он успешно проходит.

На 3 этапе можно провести небольшой рефакторинг, например, создать метод InitQueue<T>, который будет создавать нам пустую очередь (чтобы мы каждый раз не обращались к конструктору)

private MyQueue<T> InitQueue<T>() => new MyQueue<T>();

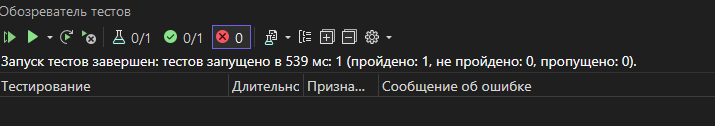


Рисунок 2. Скриншот прохождения первого теста

1. Метод, в котором мы вызываем несколько раз методы Dequeue (который удаляет элемент из начала очереди) и Enqueue (который добавляет элемент в конец очереди), а затем сравниваем размер очереди с тем, который должен получиться

На 1 этапе пишем вот такой тестовый метод для Xunit:

[Fact]

[Trait("DisplayName", "Class Count Not empty")]

public void ClassQueue\_Count\_NotEmpty()

{

var q = InitQueue<TestClass>();

foreach (var i in Enumerable.Range(0, 10))

q.Enqueue(new TestClass { Value = i });

foreach (var i in Enumerable.Range(0, 7))

q.Dequeue();

q.Count.Should().Be(3);

}

На 2 этапе реализуем метод Enqueue и проверим работоспособность нашего тестового метода:

public void Enqueue(T item)

{

SetCapacity(++\_size);

\_array[++\_head] = item;

}

private void SetCapacity(int capacity)

{

T[] newarray = new T[capacity];

Array.Copy(\_array, \_tail, newarray, 0, capacity);

\_array = newarray;

\_version++;

}

На 3 этапе проведем рефакторинг методов Dequeue и Enqueue и для дальнейших тестовых операций в этой лабораторной работе создадим отдельный метод SetCapacity, который будет пересоздавать массив на новое нужное число элементов.

1. Метод, в котором мы проверим простое использование итерационного цикла foreach с использованием уже реализованных методов Dequeue и Enqueue

[Fact]

[Trait("DisplayName", "Class Foreach simple")]

public void ClassQueue\_Foreach\_Simple()

{

var testItem1 = new TestClass { Value = 1 };

var testItem2 = new TestClass { Value = 2 };

var testItem3 = new TestClass { Value = 3 };

var testItem4 = new TestClass { Value = 4 };

var q = InitQueue<TestClass>();

q.Enqueue(testItem1);

q.Enqueue(testItem2);

q.Enqueue(testItem3);

q.Enqueue(testItem4);

var arr = new[]

{

testItem1,testItem2,testItem3,testItem4

};

int k = 0;

foreach (var item in q)

item.Should().BeSameAs(arr[k++]);

}

1. Метод, в котором мы попытаемся проводить изменение очереди прямо во время выполнения цикла foreach (c помощью метода Dequeue)

А затем логикой запретим это делать, и изменим, соотвественно, логику тестового метода, добавив проверку на выброс исключения (проверка будет реализована в методе MoveNext, который используется при итерациях foreach)

[Fact]

[Trait("DisplayName", "Class Foreach invalidation Dequeue")]

public void ClassQueue\_Foreach\_InvalidationDequeue()

{

var q = InitQueue<TestClass>();

foreach (var i in Enumerable.Range(0, 10))

q.Enqueue(new TestClass { Value = i });

void a()

{

foreach (var item in q)

q.Dequeue();

}

var ex = Record.Exception(a);

ex.Should().NotBeNull().And.BeOfType<InvalidOperationException>();

}

1. Метод, в котором мы проверим корректность работы реализованных методов интерфейса Ienumerator, то есть логику методов MoveNext, Current и Reset.

[Fact]

[Trait("DisplayName", "Class Foreach Reset")]

public void ClassQueue\_Iterator\_Reset()

{

var q = InitQueue<TestClass>();

var arr = new[]

{

new TestClass{ Value = 1 },

new TestClass{ Value = 2 },

new TestClass{ Value = 3 },

};

foreach (var i in Enumerable.Range(0, 3))

q.Enqueue(arr[i]);

int k = 0;

var iterator = q.GetEnumerator();

while (iterator.MoveNext())

iterator.Current.Should().BeSameAs(arr[k++]);

k.Should().Be(3);

k = 0;

iterator.Reset();

while (iterator.MoveNext())

iterator.Current.Should().BeSameAs(arr[k++]);

k.Should().Be(3);

}