Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

Лабораторная работа №11

По дисциплине «СПП»

за 6-й семестр

Выполнил:
студент 3 курса
группы ПО-3 (1)
Афанасьев В.В.

Проверил:
Крощенко А.А.

Брест, 2021

Цель работы: освоить приемы тестирования кода на примере использования библиотеки JUnit.

Вариант: 2

Задание1:

Создаете новый класс и скопируйте код класса Sum;

• Создаете тестовый класс SumTest;

• Напишите тест к методу Sum.accum и проверьте его исполнение. Тест должен проверять работоспособность функции accum.

• Очевидно, что если передать слишком большие значения в Sum.accum, то случится переполнение. Модифицируйте функцию Sum.accum, чтобы она возвращала значение типа long и напишите новый тест, проверяющий корректность работы функции с переполнением. Первый тест должен работать корректно.

Задание2:

Подготовка к выполнению:

• Создайте новый проект в рабочей IDE;

• Создайте класс StringUtils, в котором будут находится реализуемые функции;

• Напишите тесты для реализуемых функций.

Написать тесты к методу, а затем реализовать сам метод по заданной спецификации.

Вариант 2) Разработайте метод String repeat(String str, String separator, int repeat), который строит строку из указанного паттерна, повторённого заданное количество раз, вставляя строку-разделитель при каждом повторении.

Спецификация метода:

repeat ("e", "|", 0) = ""

repeat ("e", "|", 3) = "e|e|e"

repeat (" ABC ", ",", 2) = "ABC , ABC "

repeat (" DBE ", "", 2) = " DBEDBE "

repeat (" DBE ", ":", 1) = "DBE"

repeat ("e", -2) = IllegalArgumentException

repeat ("", ":", 3) = "::"

repeat (null , "a", 1) = NullPointerException

repeat ("a", null , 2) = NullPointerException

Задание3:

1) Импорт проекта Импортируйте один из проектов по варианту (2):

• Queue – содержит реализацию очереди на основе связного списка: Queue.java.

Разберитесь как реализована ваша структура данных. Каждый проект содержит:

• Клиент для работы со структурой данных и правильности ввода данных реализации (см.

метод main()).

• TODO-декларации, указывающие на нереализованные методы и функциональность.

• FIXME-декларации, указывающую на необходимые исправления.

• Ошибки компиляции (Синтаксические)

• Баги в коде (!).

• Метод check() для проверки целостности работы класса.

2) Поиск ошибок

• Исправить синтаксические ошибки в коде.

• Разобраться в том, как работает код, подумать о том, как он должен работать и найти допущенные баги.

3) Внутренняя корректность

• Разобраться что такое утверждения (assertions) в коде и как они включаются в Java.

• Заставить ваш класс работать вместе с включенным методом check.

• Выполнить клиент (метод main() класса) передавая данные в структуру используя включенные проверки (assertions).

4) Реализация функциональности

• Реализовать пропущенные функции в классе.

• См. документацию перед методом относительно того, что он должен делать и какие исключения выбрасывать.

• Добавить и реализовать функцию очистки состояния структуры данных.

5) Написание тестов

• Все функции вашего класса должны быть покрыты тестами.

• Использовать фикстуры для инициализации начального состояния объекта.

• Итого, должно быть несколько тестовых классов, в каждом из которых целевая структура данных создается в фикстуре в некотором инициализированном состоянии (пустая, заполненная и тд), а после очищается.

• Написать тестовый набор, запускающий все тесты.

Код программы:

1) queue

Queue

```
package queue;

import java.util.NoSuchElementException;

/**
 * The <tt>Queue</tt> class represents a first-in-first-out (FIFO) queue of
 * generic items. It supports the usual <em>enqueue</em> and <em>dequeue</em>
 * operations, along with methods for peeking at the top item, testing if the
 * queue is empty, and iterating through the items in FIFO order.
 */

public class Queue<Item> {
    // the number of elements
    private int N;

    // the head
    private Node first;

    // the tail                    //fixed
```

```
private Node last;

// simple Node
private class Node {
    private Item item;
    private Node next;
}

/**
 * Create an empty queue.
 */
public Queue() {
    clear();                //fixed
}

/**
 * Clear queue.
 */
public void clear() {
    first = null;
    last = null;
    N = 0;
    assert check();
}

/**
 * Is the queue empty?
 */
public boolean isEmpty() {
    return first == null;               // != // ==
}

/**
 * Return the number of items in the queue.
 */
public int size() {
    return N;
}

/**
 * Return the item least recently added to the queue.
 *
 * @throws java.util.NoSuchElementException if queue is empty.
 */
public Item peek() {
    if (isEmpty()) {
        throw new NoSuchElementException();     // added
    }
    return first.item;
}

/**
 * Add the item to the queue.
 */
public void enqueue(Item item) {
    Node oldLast = last;             // oldlast // oldLast

    last = new Node();
    last.item = item;
    last.next = null;

    ++N;                             // added

    if (isEmpty()) {
        first = last;
    }
    else {
        oldLast.next = last;
    }

    assert check();
}

/**
```

```
 * Remove and return the item on the queue least recently added.
 *
 * @throws java.util.NoSuchElementException if queue is empty.
 */
// name fixed
public Item dequeue() {
    if (isEmpty()) {
        throw new NoSuchElementException();         // added
    }

    Item item = first.item;
    first = first.next;

    --N;

    if (isEmpty()) {
        last = null;                                // fixed
    }

    assert check();
    return item;
}

/**
 * Return string representation.
 */
public String toString() {
    StringBuilder s = new StringBuilder();

    for (Node x = first; x != null; x = x.next) {
        s.append(x.item).append(" ");               // added
    }

    return s.toString();
}

// internal invariants checking
private boolean check() {
    if (N == 0) {
        if (first != null) {
            return false;
        }
        return last == null;
    }
    else if (N == 1) {
        if (first == null || last == null) {
            return false;
        }
        if (first != last) {
            return false;
        }
        return first.next == null;
    }
    else {
        // become more wide
        if (first == null || last == null) {
            return false;
        }
        if (first == last) {
            return false;
        }
        if (first.next == null) {
            return false;
        }
        if (last.next != null) {
            return false;
        }

        // internal consistency of instance variable N checking
        int numberOfNodes = 0;

        for (Node x = first; x != null; x = x.next) {
            numberOfNodes++;
        }
```

```
            if (numberOfNodes != N) {
                return false;
            }

            // internal consistency of instance variable last checking
            Node lastNode = first;

            while (lastNode.next != null) {
                lastNode = lastNode.next;
            }

            return last == lastNode;
        }
    }
}
```

QueueClient

```
package queue;

import java.util.Scanner;

public class QueueClient {

    /**
     * A test client.
     */
    public static void main(String[] args) {
        Queue<String> q = new Queue<String>();

        Scanner scanner = new Scanner(System.in);

        System.out.println("Start...");
        System.out.println("Enter your values. If you entered '-', the element will be showed");

        while (scanner.hasNext()) {
            String item = scanner.next();

            if (!item.equals("-")) {
                q.enqueue(item);
                System.out.println("Elements has: "+q.size());
            }
            else if (!q.isEmpty()) {
                System.out.println(q.dequeue() + " ");
                System.out.println("Elements has: "+q.size());
            }
        }
    }
}
```

2) test->queue

QueueEmptyTest

```
package queue;

import org.junit.After;
import org.junit.Test;

import static org.junit.Assert.*;

public class QueueEmptyTest {
    Queue<String> queue = new Queue<>();

    @After
    public void queueClear() throws Exception {
        queue.clear();
    }

    @Test
    public void clear_when_queueIsCleaned_should_returnEmptyResult() {
        queue.clear();
        assertTrue(queue.isEmpty());
```

```
        assertEquals(0, queue.size());
        assertEquals("", queue.toString());
    }

    @Test
    public void isEmptyResult_when_queueIsCleaned_should_returnIsEmpty() {
        assertTrue(queue.isEmpty());
    }

    @Test
    public void size_when_queueIsCleaned_should_returnNullSize() {
        assertEquals(0, queue.size());
    }

    @Test(expected = java.util.NoSuchElementException.class)
    public void peek_when_queueIsCleaned_should_returnException() {
        queue.peek();
    }

    @Test
    public void enqueue_when_queueIsAddedWithOneElement_should_returnWorkingResult() {
        queue.enqueue("Test1");
        assertFalse(queue.isEmpty());
        assertEquals(1, queue.size());
        assertEquals("Test1 ", queue.toString());
    }

    @Test(expected = java.util.NoSuchElementException.class)
    public void dequeue_when_queueIsCleaned_should_returnException() {
        queue.dequeue();
    }

    @Test
    public void toString_when_queueIsCleaned_should_returnEmptyString() {
        assertEquals("", queue.toString());
    }
}
```

QueueManyElementsTest

```
package queue;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class QueueManyElementsTest {
    Queue<String> queue = new Queue<>();

    @Before
    public void init() throws Exception {
        queue.enqueue("Test1");
        queue.enqueue("Test2");
    }

    @After
    public void queueClear() throws Exception {
        queue.clear();
    }

    @Test
    public void clear_when_queueIsFilledWithTwoElements_should_returnEmptyResult() {
        queue.clear();
        assertTrue(queue.isEmpty());
        assertEquals(0, queue.size());
        assertEquals("", queue.toString());
    }

    @Test
    public void isEmpty_when_queueIsFilledWithTwoElements_should_returnIsNotEmpty() {
        assertFalse(queue.isEmpty());
    }
```

```
    @Test
    public void size_when_queueIsFilledWithTwoElements_should_returnResultTwo() {
        assertEquals(2, queue.size());
    }

    @Test
    public void peek_when_queueIsFilledWithTwoElements_should_returnWorkingResult() {
        assertEquals(2, queue.size());
        assertEquals("Test1", queue.peek());
        assertEquals(2, queue.size());
    }

    @Test
    public void
enqueue_when_queueIsAddedWithOneElement_should_returnWorkingResultAndDecreasedSize() {
        queue.enqueue("Test3");
        assertFalse(queue.isEmpty());
        assertEquals(3, queue.size());
        assertEquals("Test1 Test2 Test3 ", queue.toString());
    }

    @Test
    public void
dequeue_when_queueIsErasedWithOneElement_should_returnWorkingResultAndErasedSize() {
        assertEquals(2, queue.size());
        assertEquals("Test1", queue.dequeue());
        assertEquals(1, queue.size());
    }

    @Test
    public void toString_when_queueIsFilledWithTwoElements_should_returnFilledString() {
        assertEquals("Test1 Test2 ", queue.toString());
    }
}
```

QueueOneElementTest

```
package queue;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class QueueOneElementTest {
    Queue<String> queue = new Queue<>();

    @Before
    public void init() throws Exception {
        queue.enqueue("Test1");
    }

    @After
    public void queueClear() throws Exception {
        queue.clear();
    }

    @Test
    public void clear_when_queueIsFilledWithOneElement_should_returnEmptyResult() {
        queue.clear();
        assertTrue(queue.isEmpty());
        assertEquals(0, queue.size());
        assertEquals("", queue.toString());
    }

    @Test
    public void isEmpty_when_queueIsFilledWithOneElement_should_returnIsNotEmpty() {
        assertFalse(queue.isEmpty());
    }

    @Test
    public void size_when_queueIsFilledWithOneElement_should_returnResultOne() {
```

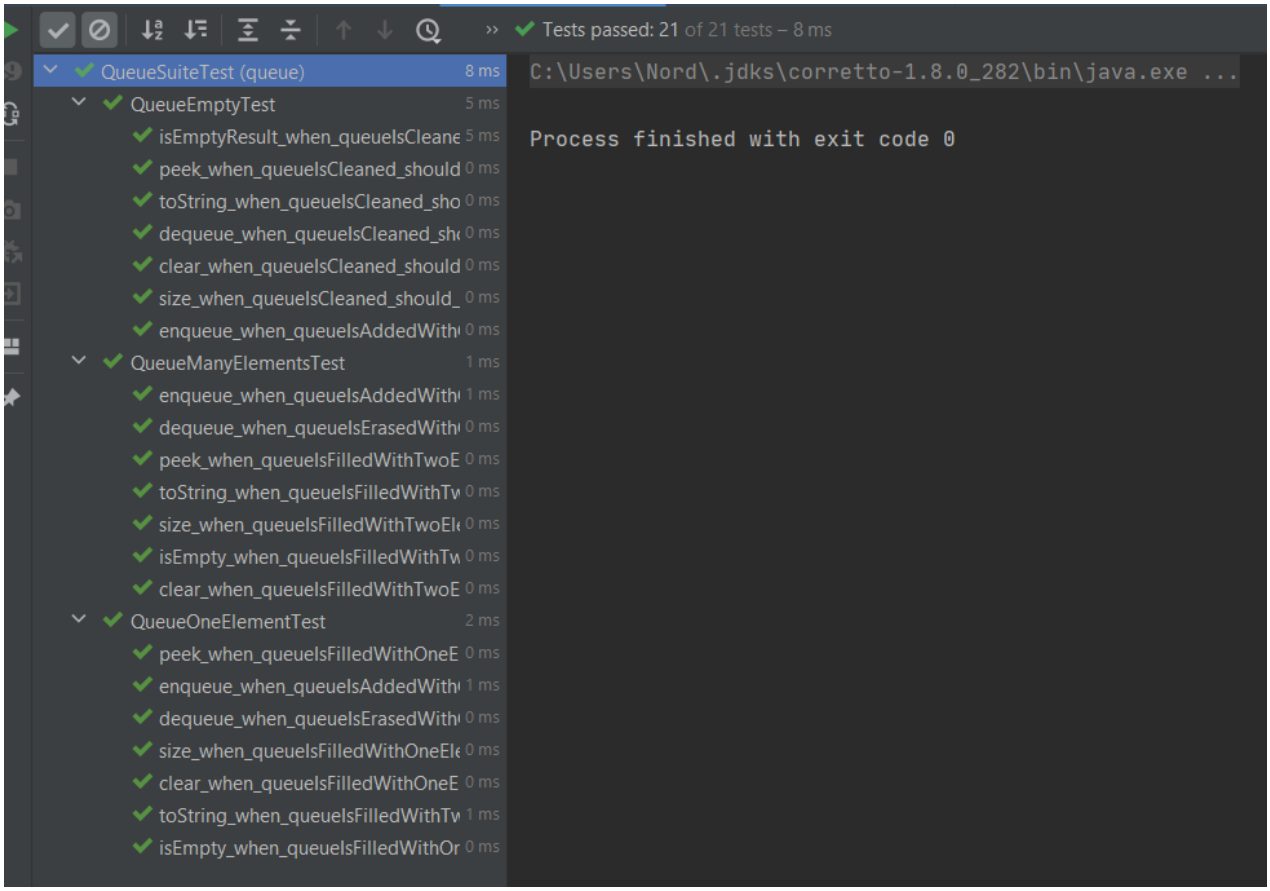
```
        assertEquals(1, queue.size());
    }

    @Test
    public void peek_when_queueIsFilledWithOneElement_should_returnWorkingResult() {
        assertEquals(1, queue.size());
        assertEquals("Test1", queue.peek());
        assertEquals(1, queue.size());
    }

    @Test
    public void
enqueue_when_queueIsAddedWithOneElement_should_returnWorkingResultAndDecreasedSize() {
        queue.enqueue("Test2");
        assertFalse(queue.isEmpty());
        assertEquals(2, queue.size());
        assertEquals("Test1 Test2 ", queue.toString());
    }

    @Test
    public void
dequeue_when_queueIsErasedWithOneElement_should_returnWorkingResultAndErasedSize() {
        assertEquals(1, queue.size());
        assertEquals("Test1", queue.dequeue());
        assertEquals(0, queue.size());
        assertTrue(queue.isEmpty());
    }

    @Test
    public void toString_when_queueIsFilledWithTwoElements_should_returnFilledString() {
        assertEquals("Test1 ", queue.toString());
    }
}
```

QueueSuiteTest

```
package queue;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses ({QueueEmptyTest.class, QueueManyElementsTest.class,
QueueOneElementTest.class})
public class QueueSuiteTest {

}
```

Результаты работы:

```
Start...
Enter your values. If you entered '-', the element will be showed
5 9 8 7
Elements has: 1
Elements has: 2
Elements has: 3
Elements has: 4
-
5
Elements has: 3
-
9
Elements has: 2
-
8
Elements has: 1
-
7
Elements has: 0
-
```



Выводы: в ходе выполнения лабораторной работы были освоены приемы тестирования кода на примере использования библиотеки JUnit.