**НТУУ «КПІ» 2014**

**Java**

**Лабораторна робота №8:**

**Обробка виключних ситуацій та основи тестування в мові програмування Java**

**Факультет: ФІОТ**

**Група: ІО-34**

**Заліковка №3405**

**Виконав:**

**Власов М. Д.**

**Викладач:**

**Невдащенко М. В.**

**Мета**

Здобуття навичок у використанні механізму обробки виключних ситуацій та написанні тестів для перевірки працездатності методів в мові програмування Java.

**Завдання**

1. Модифікувати класи з попередніх лабораторних робіт (лабораторні роботи №6 та №7) таким чином, щоб обробка виключних ситуацій відбувалась за допомогою стандартних засобів мови програмування Java. Створити власний клас обробник виключних ситуацій.

2. Написати JUnit-тести для перевірки працездатності УСІХ методів та виключних ситуацій.

3. Всі початкові дані задаються у виконавчому методі. Код повинен відповідати стандартам JCC та бути детально задокументований.

**Опис програми**

Клас LinkedFlowersTest тестує всі методи класа LinkedFlowers на працездатність за допомогою J-Unit тестів. У класі LinkedFlowers обробка виключних ситуацій відбувається за допомогою стандартних засобів мови програмування Java.

**Код**

/\*\*

\* @author Maxym Vlasov

\* @version 1.0 06 June 2014

\*/

**import** java.util.Collection;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.ListIterator;

/\*\*

\* Реалізація інтерфейсу List

\*/

**public** **class** LinkedFlowers **implements** List<Flower> {

/\*\*

\* **@param** size розмір колекції

\* **@param** head перший елемент колекції

\* **@param** tail останный елемент колекції

\*/

**private** **int** size;

**private** Item head = **new** Item(**null**);

**private** Item tail = **new** Item(**null**);

/\*\*

\* Конструктор створює пусту колекцію

\*/

**public** LinkedFlowers() {

head = **new** Item(**null**);

tail = **new** Item(**null**);

}

/\*\*

\* Конструктор створюе колекцію з одним елементом

\* **@param** f об'єкт класу Flower

\*/

**public** LinkedFlowers(Flower f) {

**this**();

add(f);

}

/\*\*

\* Конструктор створює колекцію з переданою в нього колекцією елементів

\* **@param** c колекція об'єктів класу Flower

\*/

**public** LinkedFlowers(Collection<? **extends** Flower> c) {

**this**();

addAll(c);

}

/\*\*

\* **@return** розмір колекції

\*/

@Override

**public** **int** size() {

**return** size;

}

/\*\*

\* **@return** true, якщо колекція пуста

\*/

@Override

**public** **boolean** isEmpty() {

**return** size == 0;

}

/\*\*

\* **@return** true якщо даний елемент є в колекції

\*/

@Override

**public** **boolean** contains(Object o) {

**return** indexOf(o) != 0;

}

/\*\*

\* **@return** true якщо всі елементи з переданої колекції є в даній

\*/

@Override

**public** **boolean** containsAll(Collection<?> c) {

**boolean** res = **true**;

**for**(**int** i =0; i < c.size(); i++)

**if**(!**this**.contains(((LinkedFlowers)c).get(i)))

res = **false**;

**return** res;

}

/\*\*

\* **@return** ітератор для проходження по колекції

\*/

**private** **class** MyIterator **implements** Iterator<Flower> {

**private** **int** index = 0;

/\*\*

\* **@return** true, якщо в колекції є наступний елемент

\*/

@Override

**public** **boolean** hasNext() {

**return** index < size;

}

/\*\*

\* **@return** наступний елемент

\*/

@Override

**public** Flower next() {

**return** (Flower) get(index++);

}

@Override

**public** **void** remove() {

}

}

/\*\*

\* **@return** ListIterator для проходження по колекції

\*/

**private** **class** LIterator **implements** ListIterator<Flower> {

**private** **int** i = 0;

**public** LIterator() {

i = 0;

}

**public** LIterator(**int** index) {

i = index;

}

/\*\*

\* **@return** true, якщо в колекції є наступний елемент

\*/

@Override

**public** **boolean** hasNext() {

**return** i < size;

}

/\*\*

\* **@return** наступний елемент

\*/

@Override

**public** Flower next() {

**return** (Flower) get(i++);

}

/\*\*

\* **@return** true, якщо в колекції є попередній елемент

\*/

@Override

**public** **boolean** hasPrevious() {

**return** i >= 0;

}

/\*\*

\* **@return** попередній елемент

\*/

@Override

**public** Flower previous() {

**return** (Flower) get(i--);

}

/\*\*

\* **@return** позиція наступного елемента

\*/

@Override

**public** **int** nextIndex() {

**return** i + 1;

}

/\*\*

\* **@return** позиція попереднього елемента

\*/

@Override

**public** **int** previousIndex() {

**return** i - 1;

}

@Override

**public** **void** remove() {

}

@Override

**public** **void** set(Flower f) {

LinkedFlowers.**this**.set(i, f);

}

@Override

**public** **void** add(Flower f) {

LinkedFlowers.**this**.add(i, f);

}

}

/\*\*

\* Метод записує колекцію в масив

\* **@return** масив об'єктів класу Flower

\*/

@Override

**public** Flower[] toArray() {

Flower[] f = **new** Flower[size];

**for**(**int** i = 0; i < size; i++)

f[i] = **this**.get(i);

**return** f;

}

/\*\*

\* Метод записує колекцію у визначений масив

\* **@return** масив об'єктів класу Flower

\*/

@Override

**public** Object[] toArray(Object[] arr) {

Flower[] f = **new** Flower[arr.length];

**for**(**int** i = 0; ((i < size) && (i < arr.length)); i++)

f[i] = (Flower) **this**.get(i);

**for**(**int** i = size; i < arr.length; i++)

f[i] = **null**;

**return** f;

}

/\*\*

\* Метод додає в кінець колекції переданий йому об'єкт

\* **@return** true

\*/

@Override

**public** **boolean** add(Flower f) {

**if**(size == 0) {

head.setData(f);

tail = head;

}

**else** {

tail.setNext(**new** Item(f));

tail.getNext().setPrev(tail);

tail = tail.getNext();

}

size++;

**return** **true**;

}

/\*\*

\* Метод видаляє переданий йому елемент з колекції

\* **@return** true якщо елемент було знайдено і видалено

\*/

@Override

**public** **boolean** remove(Object o) {

Item current = head;

**boolean** res = **false**;

**if**(current.getValue().getName().equals(((Flower)o).getName())) {

head = current.getNext();

head.setPrev(**null**);

res = **true**;

size--;

}

**else** {

**while**(current.getNext() != **null**) {

Item next = current.getNext();

**if**(next.getValue().getName().equals(((Flower)o).getName())) {

current.setNext(next.getNext());

current.getNext().setPrev(current);

size--;

res = **true**;

}

**else**

current = current.getNext();

}

}

**return** res;

}

/\*\*

\* Метод додає об'єкт до колекції у визначеному місці

\*/

@Override

**public** **void** add(**int** index, Flower f) {

**if** (index == 0) {

head.setPrev(**new** Item(f));

head.getPrev().setNext(head);

head = head.getPrev();

}

**else** {

**int** i = 0;

Item current = head;

**for**( i= 0; i < index; i++)

current = current.getNext();

current.getPrev().setNext(**new** Item(f));

current.getPrev().getNext().setPrev(current.getPrev());

current.setPrev(current.getPrev().getNext());

current.getPrev().setNext(current);

}

size++;

}

/\*\*

\* Метод видаляє з колекції елемент, що знаходиться на визначеному місці

\* **@return** значення видаленого елемента

\*/

@Override

**public** Flower remove(**int** index) {

Flower res = **null**;

**if** (index == 0) {

head.getNext().setPrev(**null**);

head = head.getNext();

}

**if** ((index != 0)&&(index != **this**.size() - 1)) {

Item current = head.getNext();

**for**(**int** i = 1; i < index; i++)

current=current.getNext();

res = current.getValue();

current.getPrev().setNext(current.getNext());

current.getNext().setPrev(current.getPrev());

}

**if** ( index == **this**.size() - 1) {

tail.getPrev().setNext(**null**);

tail = tail.getPrev();

}

size--;

**return** res;

}

/\*\*

\* Метод додає передану йому колекцію до даної, починаючи з визначеної позиції

\* **@return** true

\*/

@Override

**public** **boolean** addAll(**int** index, Collection <? **extends** Flower> c) {

**if**(index < 0)

index = 0;

**if** (index >= **this**.size())

**for** (**int** i = 0; i < c.size(); i++)

add(((LinkedFlowers) c).get(i));

**else**

**for** (**int** i = 0; i < c.size(); i++)

add(index++, ((LinkedFlowers) c).get(i));

**return** **true**;

}

/\*\*

\* Метод видаляє всі елементи переданої йому колекції з даної

\* **@return** true, якщо всі елементи переданої колекції були видалені

\*/

@Override

**public** **boolean** removeAll(Collection<?> c) {

**boolean** res = **true**;

**for**(**int** i = 0; i < c.size(); i++)

**if** (**this**.indexOf(((LinkedFlowers)c).get(i)) != -1)

**this**.remove(**this**.indexOf(((LinkedFlowers)c).get(i)));

**else**

res = **false**;

**return** res;

}

/\*\*

\* Метод додає передану йому колекцію в кінець даної

\* **@return** true

\*/

@Override

**public** **boolean** addAll(Collection<? **extends** Flower> c) {

**for** (**int** i = 0; i < c.size(); i++)

add(((LinkedFlowers) c).get(i));

**return** **true**;

}

/\*\*

\* Метод очищує колекцію

\*/

@Override

**public** **void** clear() {

tail = **new** Item(**null**);

head = tail;

size = 0;

}

/\*\*

\* Метод видаляє з даної колекції всі елементи, яких немає в переданій

\* **@return** true якщо дана колекція має в собі всі елементи переданої

\*/

@Override

**public** **boolean** retainAll(Collection<?> c) {

**for**(**int** i = **this**.size() - 1; i >= 0; i--)

**if** (((LinkedFlowers)c).indexOf(**this**.get(i)) == -1)

**this**.remove(i);

**if** (**this**.containsAll(c))

**return** **true**;

**else**

**return** **false**;

}

/\*\*

\* Метод повертає елемент колекції, який знаходиться на визначеній позиції

\* **@return** значення знайденого елемента

\*/

@Override

**public** Flower get(**int** index) {

Flower result = **null**;

Item current = head;

**try**{

**for**(**int** i = 0; i < index; i++)

current = current.getNext();

result = current.getValue();

}**catch** (NullPointerException e){

System.*out*.println("Index is larger than the size");

}

**return** result;

}

/\*\*

\* Метод вставляє елемент в колекцію на визначену позицію

\* **@return** значення елемента, який стояв на цій позиції до виконяння метода

\*/

@Override

**public** Flower set(**int** index, Flower f) {

Flower res = **null**;

Item current = head;

**for**(**int** i = 0; i < index; i++)

current = current.getNext();

res = current.getValue();

current.setData(f);

**return** res;

}

/\*\*

\* **@return** ітератор для проходження по колекції

\*/

@Override

**public** Iterator<Flower> iterator() {

**return** **new** MyIterator();

}

/\*\*

\* **@return** позиція першого входження переданого елемента, -1, якщо такого елемента в колекції немає

\*/

@Override

**public** **int** indexOf(Object o) {

**int** res = -1;

**for** (**int** i = 0; (i < size) && (res == -1); i++)

**if** (**this**.get(i).getName().equals(**new** Item((Flower)o).getValue().getName()))

res = i;

**return** res;

}

/\*\*

\* **@return** позиція останнього входження переданого елемента, -1, якщо такого елемента в колекції немає

\*/

@Override

**public** **int** lastIndexOf(Object o) {

**int** res = -1;

**for** (**int** i = size - 1; (i >= 0) && (res == -1); i--)

**if** (**this**.get(i).getName().equals(**new** Item((Flower)o).getValue().getName()))

res = i;

**return** res;

}

/\*\*

\* **@return** ListIterator для проходження по колекції

\*/

@Override

**public** ListIterator<Flower> listIterator() {

**return** **new** LIterator();

}

/\*\*

\* **@return** ListIterator для проходження по колекції, починаючи з певноі позиції

\*/

@Override

**public** ListIterator<Flower> listIterator(**int** index) {

**return** **new** LIterator(index);

}

/\*\*

\* **@return** колекція з елементів даної, починаючи і закінчуючи певною позицією

\*/

@Override

**public** LinkedFlowers subList(**int** fIndex, **int** lIndex) {

LinkedFlowers res = **new** LinkedFlowers();

**for**(**int** i = fIndex; i < lIndex; i++)

res.add(**this**.get(i));

**return** res;

}

}

/\*\*

\* Ermolenko Misha

\* Copyright (c) 2014

\*/

**import** java.util.Iterator;

**import** java.util.ListIterator;

**import** org.junit.\*;

**import** **static** org.junit.Assert.\*;

/\*\*

\* Клас тестує всі методи класа LinkedFlowers

\*/

**public** **class** LinkedFlowersTest {

**private** Flower f1 ;

**private** Flower f2 ;

**private** Flower f3 ;

**private** LinkedFlowers c;

**private** LinkedFlowers t;

**private** LinkedFlowers t1;

Iterator<Flower> iter;

ListIterator<Flower> it;

ListIterator<Flower> it2;

@Before

**public** **void** initialize(){

f1 = **new** Flower("Rose");

f2 = **new** Flower("Lily");

f3 = **new** Flower("Anemone");

c = **new** LinkedFlowers();

t = **new** LinkedFlowers();

t1 = **new** LinkedFlowers();

iter = c.iterator();

it = c.listIterator();

it2 = c.listIterator(1);

}

@Test

**public** **void** testSize(){

c.add(f1);

c.add(f2);

c.add(f3);

*assertTrue*(c.size() == 3);

}

@Test

**public** **void** testIsEmpty(){

*assertTrue*(c.size() == 0);

}

@Test

**public** **void** testContains(){

*assertTrue*(c.indexOf(f1) == -1);

c.add(f1);

*assertFalse*(c.indexOf(f1) == -1);

}

@Test

**public** **void** testToArray(){

LinkedFlowers c = **new** LinkedFlowers();

c.add(f1);

c.add(f2);

Flower[] f = **new** Flower[2];

f[0] = f1;

f[1] = f2;

*assertArrayEquals*(f, c.toArray());

}

@Test

**public** **void** testAdd(){

*assertTrue*(c.add(f1));

}

@Test

**public** **void** testRemove(){

c.add(f1);

c.add(f2);

c.add(f3);

c.remove(f2);

*assertTrue*(c.indexOf(f3) == 1);

}

@Test

**public** **void** testAddInd(){

c.add(f1);

c.add(f2);

c.add(1,f3);

*assertTrue*(c.indexOf(f3) == 1);

}

@Test

**public** **void** testRemoveInd(){

c.add(f1);

c.add(f2);

c.add(f3);

c.remove(1);

*assertTrue*(c.indexOf(f3) == 1);

}

@Test

**public** **void** testAddAll(){

c.add(f1);

t.add(f2);

t.add(f3);

*assertTrue*(c.addAll(t));

}

@Test

**public** **void** testContainsAll(){

t.add(**new** Flower("Rose"));

t.add(**new** Flower("Lily"));

c.add(**new** Flower("Rosa"));

c.add(**new** Flower("Lilу"));

*assertTrue*(c.containsAll(t));

}

@Test

**public** **void** testRemoveAll(){

c.add(f1);

c.add(f2);

c.add(f3);

t.add(f2);

t.add(f3);

c.removeAll(t);

*assertTrue*(c.indexOf(f2) == -1);

}

@Test

**public** **void** testAddAllInd(){

c.add(f1);

c.add(f2);

t.add(f2);

t.add(f3);

*assertTrue*(c.addAll(0, t));

}

@Test

**public** **void** testClear(){

c.add(f1);

c.add(f2);

c.clear();

*assertTrue*(c.size() == 0);

}

@Test

**public** **void** testRetainAll(){

c.add(f1);

c.add(f2);

c.add(f3);

t.add(f2);

t.add(f3);

c.retainAll(t);

*assertTrue*(c.size() == 2);

}

@Test

**public** **void** testGet(){

c.add(f1);

c.add(f2);

c.add(f3);

*assertTrue*(c.get(1) == f2);

}

@Test

**public** **void** testSet(){

c.add(f1);

c.add(f2);

c.add(f3);

c.set(1, f3);

*assertTrue*(c.get(1) == f3);

}

@Test

**public** **void** testIterator(){

c.add(f1);

c.add(f2);

*assertTrue*(iter.hasNext());

}

@Test

**public** **void** testIndexOf(){

c.add(f1);

c.add(f2);

*assertTrue*(c.indexOf(f1) == 0);

}

@Test

**public** **void** testLastIndexOf(){

c.add(f2);

c.add(f2);

*assertTrue*(c.lastIndexOf(f2) == 1);

}

@Test

**public** **void** testListIterator(){

c.add(f1);

c.add(f2);

*assertTrue*(it.hasNext());

}

@Test

**public** **void** testListIterarorInd(){

c.add(f1);

c.add(f2);

c.add(f3);

*assertTrue*(it2.hasNext());

}

@Test

**public** **void** testsublist(){

c.add(f1);

c.add(f2);

c.add(f3);

t.add(f1);

t.add(f2);

*assertTrue*(c.subList(0, 2).get(1) == f2);

}

}

**Висновки**

У даній лабораторній роботі ми здобули навички у використанні механізму обробки виключних ситуацій та написанні тестів для перевірки працездатності методів в мові програмування Java, створивши клас, що тестує методи іншого класу, а також обробивши виключні ситуації допомогою стандартних засобів мови програмування Java.