Практична робота 7-8

ПРИЙОМИ РЕФАКТОРИНГУ ДЛЯ СКЛАДАННЯ ТА ПЕРЕМІЩЕННЯ МЕТОДІВ.

1. Попередні заходи

Практично, з огляду на часові та бюджетні міркування, неможливо провести вичерпне тестування для кожного набору даних тесту, особливо якщо існує велика кількість тестових зразків. Потрібний простий спосіб або спеціальні методики, які можуть вибирати тестові випадки інтелектуально з пулу test-case, таким чином, що всі тестові сценарії охоплюються. Для цього використовуються дві методики - ***Тестування граничного значення та тестування класів еквівалентності.***

1.1. Тестування граничного значення

"Досвід показує, що тестові випадки, які вивчають граничні умови, мають вищий виграш (payoff), ніж тестові випадки, які цього не роблять".

Часто трапляються невдачі на межі умовних виразів (conditional statements) та структур даних. При тестуванні з граничним аналізом значень тестові випадки призначені для тестування на краях вхідних і вихідних ділянок.

Наступні пункти - це пропозиції щодо розробки граничних тестів.

1) Якщо вхід може бути значенням між a та b, ми повинні протестувати значення a та b, а також значення, що знаходяться безпосередньо до і після a та b.

2) Якщо програма може приймати декілька входів (між максимумом та мінімумом), ми повинні протестувати її з максимальною та мінімальною кількістю входів, а також кількістю, що знаходяться безпосередньо вище і трохи нижче максимуму і мінімуму.

3) Попередні два моменти слід також застосовувати до вихідних значень.

4) Для структур даних конкретних розмірів ми повинні використовувати структуру на найменших і найбільших межах.

Тестування граничного значення є методом перевірки здорового глузду - програмісти інтуїтивно схильні виконувати цей тип тестування до певної міри при написанні програм.

Розглянемо **Example of Boundary Value Testing**. Припустимо, у вас дуже важливий інструмент в офісі, він приймає дійсне поле Ім'я користувача та пароль для роботи з цим інструментом і приймає мінімум 8 символів і

максимум 12 символів. Valid діапазон 8-12, Invalid діапазон 7 або менше 7 та Invalid діапазон 13 або більше 13.

Записати тестові випадки для Valid partition value, Invalid partition value та для граничного значення (exact boundary value).

• Тестовий випадок 1: Розглянемо довжину пароля менше 8.
• Тестовий випадок 2: Розглянемо пароль довжини рівно 8.
• Тестовий випадок 3: Розглянути пароль довжиною від 9 до 11.
• Тестовий випадок 4: Розглянути пароль довжини рівною 12.
• Тестовий випадок 5: розглянути пароль довжиною більше 12.

Розглянемо другий приклад **Example of Boundary Value Testing.** Тестові випадки для програми, чия область введення приймає числа від 1 до 1000. Valid range 1-1000, Invalid range 0 та Invalid range 1001 або більше.

Записати тестові випадки для Valid partition value, Invalid partition value та exact boundary value.

• Тестовий випадок 1: розглядайте дані тесту точно такі, як вхідні межі вхідного домену, тобто значення 1 та 1000.

• Тестовий випадок 2: розгляньте дані тесту зі значеннями, що знаходяться безпосередньо під крайньою межею вхідних доменів, тобто значеннях 0 та 999.

• Тестовий випадок 3: розглянути дані тесту з значеннями, що знаходяться безпосередньо за крайніми межами вхідного домену, тобто значення 2 та 1001.

1.2 Equivalence Class Testing

"Клас еквівалентності являє собою набір вхідних значень такий, що, якщо будь-яке значення обробляється правильно (неправильно), то припускається, що всі інші значення будуть оброблені правильно (неправильно)".

Тест класів еквівалентності є методом тестування чорної коробки. Ми використовуємо особливості програми для визначення класів входів. Ідея полягає в тому, що ми групуємо входи в набори valid and invalid вхідних значень. Тоді ми можемо вибрати одне значення з кожного класу для тестування програми. Це мінімізує кількість тестів, які нам потрібно виконати, але водночас має хороші шанси знайти помилки.

Ми можемо використовувати наступні інструкції для вибору класів еквівалентності:

• Якщо вхід може бути одним із діапазону значень, ми визначаємо один valid (min <x <max) та два invalid (x<min та max>x) класи еквівалентності.

• Якщо програма приймає певну кількість значень як вхідних даних (наприклад, між 1 і 3 значеннями), ми повинні мати один valid і два invalid (немає даних, більше 3 вхідних даних) класів еквівалентності.

• Якщо вхід може бути встановленим значенням, кожен з яких може розглядатися по-різному в програмі, повинен бути один valid клас еквівалентності для кожного члена набору, а один invalid клас еквівалентності - елемент, який не входить до набору.

• Якщо вхід має бути певним значенням, ми повинні мати один дійсний клас еквівалентності, де значення правильне, і один недійсний клас еквівалентності, де значення не є правильним.

• Якщо програма не обробляє всі елементи класу еквівалентності, клас повинен бути розділений на менші класи еквівалентності.

Давайте розглянемо приклад тестування класу еквівалентності.

• Текстове поле допускає лише цифрові символи

• Довжина повинна бути 6-10 символів. Розбиття відповідно до вимоги має бути таким:

Під час оцінки Equivalence partitioning значення в усіх розділах є еквівалентними, тому 0-5 еквівалентні, 6 - 10 еквівалентні та 11 - 14 еквівалентні.

Під час тестування тести 4 та 12 вважаються invalid, а 7 - valid. Легко перевірити вхідні діапазони 6-10, але важче тестувати діапазони вхідних даних

2-600. Тестування буде легким у випадку менших випробувань, але ви повинні бути дуже обережні. Припускаючи, valid вхід становить 7. Це означає, що ви вважаєте, що розробник кодував правильний діючий діапазон (6-10).

1.3 Параметризовані випробування

Junit 4 представила нову функцію – параметризовані тести. Параметризовані тести дозволяють розробникам проводити один і той же тест знову і знову, використовуючи різні значення. Є п'ять кроків, які потрібно виконати, щоб створити Parameterized tests.

• Анотація тестового класу @RunWith (Parameterized.class)

• Створіть загальнодоступний статичний метод із приміткою @Parameters, який повертає Колекцію об'єктів (як масив) як набір тестових даних.

• Створіть загальнодоступний конструктор, який приймає в еквіваленті один "рядок" тестових даних.

• Створіть змінну екземпляра для кожного "стовпця" даних тесту.

• Створіть тестовий приклад, використовуючи змінні екземпляра як джерело даних тесту. Тестовий випадок буде викликати один раз на кожен рядок даних.

Розглянемо, наприклад, клас PrimeNumberChecker та відповідні параметризовані тести.

```
public class PrimeNumberChecker {
   public Boolean validate(final Integer primeNumber) {
      for (int i = 2; i < (primeNumber / 2); i++) {
         if (primeNumber % i == 0) {
            return false;
         }
      }
      return true;
   }
}
```

Давайте розглянемо Parameterized tests.

```java
import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.Before;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertEquals;

@RunWith(Parameterized.class)
public class PrimeNumberCheckerTest {
   private Integer inputNumber;
   private Boolean expectedResult;
   private PrimeNumberChecker primeNumberChecker;

   @Before
   public void initialize() {
      primeNumberChecker = new PrimeNumberChecker();
   }

   // Each parameter should be placed as an argument here
   // Every time runner triggers, it will pass the arguments
   // from parameters we defined in primeNumbers() method
   public PrimeNumberCheckerTest(Integer inputNumber,
      Boolean expectedResult) {
      this.inputNumber = inputNumber;
      this.expectedResult = expectedResult;
   }
```

```java
   @Parameterized.Parameters
   public static Collection primeNumbers() {
      return Arrays.asList(new Object[][] {
         { 2, true },
         { 6, false },
         { 19, true },
         { 22, false },
         { 23, true }
      });
   }

   // This test will run 4 times since we have 5 parameters defined
   @Test
   public void testPrimeNumberChecker() {
      System.out.println("Parameterized Number is : " + inputNumber);
      assertEquals(expectedResult,
      primeNumberChecker.validate(inputNumber));
   }
}
```

2. Завдання

2.1 Створіть новий проект "Java Application" під назвою "ShoppingCart" та додайте дані класу з файлу ShoppingCart.java.

2.2. Визначити класи еквівалентності та створіть відповідні unit tests для методу addItem, використовуючи наступні специфікації:

1) назву об'єкта з 1 до 32 символів вважається дійсним;

2) ціна товару в центрах,> 0, <1000 вважається дійсною;

3) кількість предметів від 1 до 1000 вважається дійсною;

4) Item.Type з набору { ITEM_REGULAR, ITEM_DISCOUNT, ITEM_SECOND_FREE, ITEM_FOR_SALE } вважається дійсним;

5) якщо загальна кількість доданих елементів більше 99 - тIndexOutOfBoundsException

6) якщо деяке значення неправильне - IllegalArgumentException

2.3 Створіть параметризований тест для методу ShoppingCart.calculateDiscount наступні специфікації:

1) Для знижки ITEM_REGULAR - 0%;

2) Для знижки ITEM_SECOND_FREE знижка 50%, якщо кількість > 1;

3) Для знижки ITEM_DISCOUNT - 10% + 10% для кожної повної 10 одиниці, але не більше 50% від загальної суми;

4) Для ITEM_FOR_SALE знижка 90%

5) За кожен повний 100й предмет одержуються додаткові 10%, але не більше 80% від загальної суми

Загальна ціна обчислюється за формулою total_price:= price* quantity * discount.

ShoppingCart.java

```
import java.util.*;
import java.text.*;
/**
* Containing items and calculating price.
*/
public class ShoppingCart
{
public static final int ITEM_REGULAR = 0;
public static final int ITEM_DISCOUNT = 1;
public static final int ITEM_SECOND_FREE = 2;
public static final int ITEM_FOR_SALE = 3;
/**
* Tests all class methods.
*/
public static void main(String[] args)
{
// TODO: add tests here
ShoppingCart cart = new ShoppingCart();
cart.addItem("Apple", 0.99, 5, ITEM_REGULAR);
cart.addItem("Banana", 20.00, 4, ITEM_SECOND_FREE);
cart.addItem("A long piece of toilet paper", 17.20, 1, ITEM_FOR_SALE);
cart.addItem("Nails", 2.00, 500, ITEM_REGULAR);
System.out.println(cart.toString());
}
/**
```

```
* Adds new item.
*
* @param title item title 1 to 32 symbols
* @param price item ptice in cents, > 0, < 1000
* @param quantity item quantity, from 1 to 1000
* @param type item type, on of ShoppingCart.ITEM_* constants
*
* @throws IndexOutOfBoundsException if total items added over 99
* @throws IllegalArgumentException if some value is wrong
*/
public void addItem(String title, double price, int quantity, int type)
{
if (title == null || title.length() == 0 || title.length() > 32)
throw new IllegalArgumentException("Illegal title");
if (price < 0.01 || price >= 1000.00)
throw new IllegalArgumentException("Illegal price");
if (quantity <= 0 || quantity > 1000)
throw new IllegalArgumentException("Illegal quantity");
if (items.size() == 99)
throw new IndexOutOfBoundsException("No more space in cart");
Item item = new Item();
item.title = title;
item.price = price;
item.quantity = quantity;
item.type = type;
items.add(item);
}
/**
* Formats shopping price.
*
 * @return  string as lines, separated with \n,
 *       first line:   # Item                Price Quan. Discount     Total
 *       second line: ------------------------------------------------------------
 *       next lines:  NN Title              $PP.PP   Q     DD%    $TT.TT
 *                 1 Some title            $.30   2       -     $.60
 *                 2 Some very long ti... $100.00   1     50%    $50.00
 *                ...

 *                31 Item 42           $999.00 1000       - $999000.00
 *       end line:   ------------------------------------------------------------
 *       last line:  31                                    $999050.60
 *
 *       Item title is trimmed to 20 chars adding '...'
 *
 *       if no items in cart returns "No items." string.
 */
public String toString()
```

```
{
StringBuffer sb = new StringBuffer();
if (items.size() == 0)
return "No items.";
double total = 0.00;
sb.append(" # Item                Price Quan. Discount Total\n");
sb.append("-------------------------------------------------------\n");
for (int i = 0; i < items.size(); i++) {
Item item = (Item) items.get(i);
int discount = calculateDiscount(item);
double itemTotal = item.price * item.quantity * (100.00 - discount) / 100.00;
appendPaddedRight(sb, String.valueOf(i + 1), 2);
sb.append(" ");
appendPaddedLeft(sb, item.title, 20);
sb.append(" ");
appendPaddedRight(sb, MONEY.format(item.price), 7);
sb.append(" ");
appendPaddedRight(sb, String.valueOf(item.quantity), 4);
sb.append(" ");
if (discount == 0)
sb.append(" -");
else {
appendPaddedRight(sb, String.valueOf(discount), 7);
sb.append("%");
}
sb.append(" ");
appendPaddedRight(sb, MONEY.format(itemTotal), 10);
sb.append("\n");
total += itemTotal;
}
sb.append("-------------------------------------------------------\n");
appendPaddedRight(sb, String.valueOf(items.size()), 2);
sb.append(" ");
appendPaddedRight(sb, MONEY.format(total), 10);
return sb.toString();
}
// --- private section ------------------------------------------------
private static final NumberFormat MONEY;
```

```
static {
DecimalFormatSymbols symbols = new DecimalFormatSymbols();
symbols.setDecimalSeparator('.');
MONEY = new DecimalFormat("$#.00", symbols);
}
/**
* Adds to string buffer given string, padded with spaces.
* @return " str".length() == width
*/
private static void appendPaddedRight(StringBuffer sb, String str, int width)
{
for (int i = str.length(); i < width; i++)
sb.append(" ");
sb.append(str);
}
/**
* Adds string to buffer, wills spaces to width.
* If string is longer than width it is trimmed and ends with '...'
*/
private static void appendPaddedLeft(StringBuffer sb, String str, int width)
{
if (str.length() > width) {
sb.append(str.substring(0, width-3));
sb.append("...");
}
else {
sb.append(str);
for (int i = str.length(); i < width; i++)
sb.append(" ");
}
}
/**
* Calculates item's discount.
* For ITEM_REGULAR discount is 0%;
* For ITEM_SECOND_FREE discount is 50% if quantity > 1
* For ITEM_DISCOUNT discount is 10% + 10% for each full 10 items, but not more than 50% total
* For ITEM_FOR_SALE discount is 90%
```

```
* For each full 100 items item gets additional 10%, but not more than 80% total
*/
private static int calculateDiscount(Item item)
{
int discount = 0;
switch (item.type) {
case ITEM_SECOND_FREE:
if (item.quantity > 1)
discount = 50;
break;
case ITEM_DISCOUNT:
discount = 10 + item.quantity / 10 * 10;
if (discount > 50)
discount = 50;
break;
case ITEM_FOR_SALE:
discount = 90;
}
discount += item.quantity / 100 * 10;
if (discount > 80)
discount = 80;
return discount;
}
/** item info */
private static class Item
{
String title;
double price;
int quantity;
int type;
}
/** Container for added items */
private List items = new ArrayList();
}
```