Lab 6: black box testing

Goal of this lab is to practice black box testing of small software modules. For each of the following modules define test cases applying equivalence classes partitioning, and boundary conditions.

Use the structure of the document provided at https://git-softeng.polito.it/se-2021/softeng_blackbox_lab/-/blob/master/blackboxtesttemplate.md , for each exercise define clearly the criteria, the conditions on the criteria (partition), the test cases per each partition. This lab should be done individually (not in teams).

For each test case defined with the black box approach, define a JUnit test case executing the corresponding input to the function and verifying the output through assertions.

You can clone the project with the library to test from the following repository: https://git-softeng.polito.it/se-2021/softeng_blackbox_lab

Exercise 1

boolean acceptableToEat(int carb, int protein, int fat);

The function *acceptableToEat* receives the weight in grams of, respectively, carbohydrates, proteins, fats in a serving of food. It returns true if

- the total amount of calories is < 1000

- (carb + protein) / fat > ½

ex. acceptableToEat (100,100,100) -> false (tot amount of calories = 100*4 + 100*4 + 100*9 > 1000)

acceptableToEat (1,1,10) -> false (carb + protein / fat = 2/10)

acceptableToEat (1,1,1) -> true (carb + protein / fat = 2/1)

Exercise 2

A retail support system manages an inventory of items. Each item has a descriptor and the number of available items.

```
public class Item { // descriptor of items in inventory
        private String itemCode // aka bar code, unique identifier of item
        private int availability; // number of items available
        private String description; // description of item
        private String name; // name of item
}
```

```
public class Inventory{
        public void addItem(Item i) throws ItemAlreadyExists// adds new item
        public Item searchItem(String itemCode) throws ItemNotExists; // returns item with given code
        public int availabilityItem(String itemCode) throws ItemNotExists; // returns availability of item
        public void subtractItem(String itemCode) throws ItemNotExists, ItemNotAvailable;
                        // subtracts 1 to availability
        public void addQtyToItem(String itemCode, int qty_to_add) throws ItemNotExists;
        public void subtractQtyToItem (String itemCode, int qty_to_subtract) throws ItemNotExists,
ItemNotAvailable;
}
```

Exercise 3

double computeFee (int duration, int minRate, int minRate2);

This function computes (in euros) the fee for a bicycle rental, using these parameters

- duration: minutes the bicycle has been used
- minRate: cost per minute, in cents of euro
- minRate2: cost per minute, in cents of euro

The fee is computed as follows: free the first 30 minutes. minRate per min for the first hour exceeding the first 30 min (30 to 90 minutes), minRate2 after 90 minutes

Ex. computeFee(35, 10, 20) □ = (35-30) * 10

computeFee(65, 10, 20) □ = (65-30) * 10

computeFee(95, 10, 20) □ = (90-30) * 10 + (95-90) * 20

Exercise 4

double computeFee(double basePrice, int n_passengers, int n_over18, int n_under15);

A railway company offers the possibility to people under 15 to travel free. The offer is dedicated to groups from 2 to 5 people travelling together.

For being eligible to the offer, at least a member of the group must be at least 18 years old. If this condition applies, all the under 15 members of the group travel free, and the others pay the Base Price.

The function computeFee receives as parameters basePrice (the price of the ticket), n_passengers (the number of passengers of the group), n_over18 (the number of passengers at least 18 old), n_under15 (the number of passengers under 15 years old). It gives as output the amount that the whole group has to spend. It gives an error if groups are composed of more than 5 persons.

Examples:

computeFee(20.0, 3, 0, 1) -> 60.0;

computeFee(30.0, 5, 1, 2) -> 150.0