**Lab 62: black box testing, unit**

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 following structure to document the test cases, defining clearly the criteria, the conditions on the criteria (partition), the test cases per each partition.

This lab should be done individually (not in teams).

**Documentation structure**

Criteria

|  |  |
| --- | --- |
| Criterion id | description |
| Criterion 1 | C1 |
| Criterion 2 | C2 |
| … |  |

Predicates

|  |  |
| --- | --- |
|  | Predicate |
| Criterion1 | C1 == true |
|  | C1 == false |
| Criterion2 | C2 < 0 |
|  | C2 > 0 |
| … |  |

Boundaries

|  |  |
| --- | --- |
| Criterion | Boundary |
| C2 | C2 == 0 |
|  |  |

Equivalence classes and tests

|  |  |  |  |
| --- | --- | --- | --- |
| C1 | C2 | Valid invalid | Test case |
| true | < 0 |  | T1 = |
|  | > 0 |  | T2 =  T3B =  (B indicates boundary test case) |
| False | < 0 |  |  |
|  | > 0 |  |  |

**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)

criteria

type of input (carb) : int, float, double, char not testable (see later integration testing)

~~number of parameters no, checked by compiler~~

~~order of parameters , not testable~~ (see later integration testing)

sign of carb

sign of protein

sign of fat

total calories (formula 1)

proportion (formula 2)

predicates

|  |  |  |
| --- | --- | --- |
| Sign of carb | Positive |  |
|  | negative |  |
| Sign of protein | Pos |  |
|  | Neg |  |
| Sign of fat | Pos |  |
|  | Neg |  |
| Formula 1 | True |  |
|  | False |  |
| Formula 2 | True |  |
|  | false |  |

DEFECT on function design: no way to express invalid conditions (needs an exception)

Overal32 combinations

Partitions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sign of Carb | Sign of protein | Sign of fat | Formula 1 | Formula 2 | Valid invalid |
| negative | Negative | Negative | True | True | I |
|  |  |  | false | True | I |
|  |  |  | false | False | I |
|  |  |  | true | F | I |
|  |  | positive | True | True | I |
|  |  |  | false | True | I |
|  |  |  | false | False | I |
|  |  |  | true | F | I |
|  | positive | Negative | True | True | I |
|  |  |  | false | True | I |
|  |  |  | false | False | I |
|  |  |  | true | F | I |
|  |  | positive | True | True | I |
|  |  |  | false | True | I |
|  |  |  | false | False | I |
|  |  |  | true | F | I |
| positove | Negative | Negative | True | True | I |
|  |  |  | false | True | I |
|  |  |  | false | False | I |
|  |  |  | true | F | I |
|  |  | positive | True | True | I |
|  |  |  | false | True | I |
|  |  |  | false | False | I |
|  |  |  | true | F | I |
|  | positive | Negative | True | True | I |
|  |  |  | false | True | I |
|  |  |  | false | False | I |
|  |  |  | true | F | I |
|  |  | positive | True | True | V |
|  |  |  | false | True | V |
|  |  |  | false | False | V |
|  |  |  | true | F | v |

Cutting the combination

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sign of Carb | Sign of protein | Sign of fat | Formula 1 | Formula 2 |  |  |
| negative | Negative | Negative | - | - | i | T1(-1, -1, -1; error) |
| positive | positove | P | True | True | V | T2 (2,2,1; true)  TB(0,0,0; error)  TB(maxint, maxint, maxint; false) |
|  |  |  | false | True | V |  |
|  |  |  | false | False | V |  |
|  |  |  | true | F | v |  |

Missing boundary test cases on the formulas

**Exercise 2**

*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 3**

*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

**Exercise 4**

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 Item(String itemCode,int quantity); // creates a new item

}

public class Inventory{

void addItem(Item i) throws ItemAlreadyExists // adds new descriptor

Item searchItem (String itemCode) throws ItemNotExists; // returns item with given code

int availabilityItem (String itemCode) throws ItemNotExists; // returns availability of item

void subtractItem (String itemCode) throws ItemNotExists, ItemNotAvailable;

// subtracts 1 to availability

void addQtyToItem(String itemCode, int qty\_to\_add) throws ItemNotExists;

void subtractQtyToItem(String itemCode, int qty\_to\_sub) throws ItemNotExists;

}

addItem()

criteria: item already exists or not

item type

conditions

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Exists or not |  | Test cases |
| Not null | Y | I | T1  Inventory I = new Inventory();  Item i1 = new Item( “milkOneLiter”, 10);  I.addItem(i1);  I.addItem(i1) 🡪 ItemAlreadyExists  I.searchItem(“milkOneLiter”) 🡪 returns i1 |
|  | N | V | Inventory I = new Inventory();  Item i1 = new Item( “milkOneLiter”, 10);  I.addItem(i1);  I.searchItem(“milkOneLiter”) 🡪 returns i1; |
| null | Y | I | Inventory I = new Inventory();  I.addItem(null) 🡪 ?? |
|  | N | I | Same as above |

Caller -- callee

// some where in the program, CALLER

Inventory I = new Inventory(); Item it;

if (it != null) I.addItem(it) ;

// implementation of addItem , CALLEE

Void addItem(Item i) {

If (I == null ) -🡪

else

}

Who does the check?

searchItem()

criteria

item is in inventory or not

parameter is null or not

many items or one

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Is in inventory or not | Many items | Valid invalid | Test cases |
| Not null | Y | n | I | T1  Inventory I = new Inventory();  Item i1 = new Item( “milkOneLiter”, 10);  I.addItem(i1);  I.searchItem(“milkOneLiter”) 🡪 returns i1 |
|  | N | n | V | Inventory I = new Inventory();  Item i1 = new Item( “milkOneLiter”, 10);  I.addItem(i1);  I.searchItem(“orangeJuice”) 🡪 ItemNotExists |
|  | Y | Y | V | Inventory I = new Inventory();  Item i1 = new Item( “milkOneLiter”, 10);  I.addItem(i1);  I.addItem(new Item(“bread”, 5);  // here add many more different items  I.searchItem(“milkOneLiter”) 🡪 i1 |
|  | N | Y | I | Inventory I = new Inventory();  Item i1 = new Item( “milkOneLiter”, 10);  I.addItem(i1);  I.addItem(new Item(“bread”, 5);  // here add many more different items but not orange juice  I.searchItem(“OrangeJuice”) 🡪 ItemNotExists |
| null | Y |  | I | Inventory I = new Inventory();  I.searchItem(null) 🡪 ?? |
|  |  |  |  |  |

Subtract()

void subtractItem (String itemCode) throws ItemNotExists, ItemNotAvailable;

criteria

itemCode null or not

itemCode exists or not

itemcode has availability or not

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ItemCode null or not | itemCode exists | available | V / I |  |
| null | - | - | I | Inventory I = new Inventory();  I.subtractItem(null) 🡪 ?? |
| Not null | y | Y | V | Inventory I = new Inventory();  Item i1 = new Item( “milkOneLiter”, 10);  I.addItem(i1);  I.subtractItem(“milkOneLiter”);  I.availabilityItem() == 9; |
|  |  | N | I | Inventory I = new Inventory();  Item i1 = new Item( “milkOneLiter”, 1);  I.addItem(i1);  I.subtractItem(“milkOneLiter”);  I.subtractItem(“milkOneLiter”) 🡪 ItemNotAvailable |
|  | N |  | I | Inventory I = new Inventory();  Item i1 = new Item( “milkOneLiter”, 10);  I.addItem(i1);  I.subtractItem(“orange”) 🡪 ItemNotExists |