Software Quality 2018/2019

Software Testing

Notes on answer format:

Start by providing a short rationale for the way you are building up your test suites (e.g. explain how you come up with your equivalence classes).

When presenting your test suites, please create a unique identifier for each test (e.g. an integer value) and present the test suite in a table with the following columns: **test id**, **test data** (you may split this one in as many as you need), **test expected outcome(s)** (split in as many as you need), and **test classes covered** by test case.

Edit: While preparing their reports, some students noticed a couple of problems with the information provided in questions 3 and 6. The changed information is highlighted in red.

Part 1 – Black box test cases generation

Challenges

1. Your favourite supermaket has some great new promotions for fruit: for each 500g of fruit, up to a total of 2kg – over that, the 2kg discount applies – the supermarket offers a 1% discount on the selling price. Also, to promote the selling of local products, the supermarket sells fruits produced within a radius of up to 20km with a "proximity" discount of 5%. If the fruits were produced within a radius of up to 50km, this proximity discount is of 2%. Fruits produced over more than 50km away do not receive this discount. The weight and the distance discounts can be accumulated. For example, if the customer gets a 3% discount based on weight and a 5% discount based on proximity, the total discount is 8% on the base value. You were hired to implement and test an operation to compute the correct value to charge the customer for buying fruit at that supermarket. This is the operation signature:

```
/**
 * Computes the actual price to pay for fruit in this supermarket.
 * @param value – the value to pay, in cents, before applying discounts
 * @param weight – the weight of the fruit being bought, in grams
 * @param distance – the distance from the fruit origin, in kms
 * @return – the actual price, in cents, once the discount is applied
 */
public int computeFruitPrice(int value, int weight, int distance)
```

Considering this specification of the `computeFruitPrice` function, please use the **Equivalence Partitioning Testing** approach and:

a) Compute the required number of weak equivalence class tests.
b) Create a set of test cases that fulfills the weak equivalence criteria.
c) Compute the required number of strong equivalence class tests.
d) Create a set of test cases that fulfills the strong equivalence criteria.
e) Implement the test cases for covering the weak equivalence criteria using JUnit.
f) Implement the `computeFruitPrice` function in Java and test it with your JUnit test suite.

2. Consider a configurator for buying a computer monitor in a computer shop. You can select one of 10 monitor brands (Acer, Alienware, AOC, Asus, Benq, Dell, HP, Lenovo, LG, Samsung). There are 4 different panel types to consider: TN, VA, IPS, and OLED panels. We should also consider 5 resolution alternatives: HD, FullHD, QHD, UHD and 5k. Monitors may be flat or curved. We must also consider the refresh rate: 60hz, 75hz, 144hz, 180hz, 240hz Taking all this information into account, please use the Pairwise testing to design test cases for testing this application.

3. Next to your favourite supermarket, there is a household appliances megastore. They sell computers, tvs, refrigerators, hovens, hairdriers, smartphones… you name it, they have it. Now, they are about to start their Christmas/new year campaign, under the motto "Watch Home Alone in true UHD and start the new year eating really fresh shrimps!". What they are really having is a campaign on household appliances, excluding products from the informatics section of the store. You are asked to build a test battery for this campaign, so that they are confident the following set of business rules is correctly implemented:

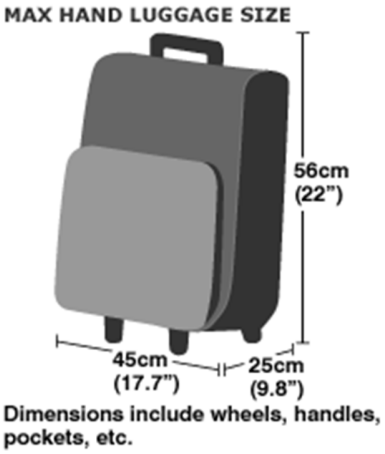
- Customers buying products worth 500€ have a 10% discount
- The promotion does not include informatics products from the store
- For purchases over 1000€, the store offers free transportation of the purchased products
- For purchases up to 1000€, the transportation fee is 10€. (new!)
- Purchases with a low accumulated weight (less than 5kg) are not eligible for free transportation
- If the customer's birthday is on the same month of the purchase, the customer has a special discount of 10€, for purchases over 200€, regardless of the kind of products (so, only for this part of the campaign, informatics products are included).

Knowing all this:

a) Analyse the specification and produce a cause-effect graph for it. Clearly identify the causes and the effects you identified, as well as their relationships
b) Derive a decision table from your cause-effect graph. You may use the value "indifferent", if necesssary, while building your decision table.
c) Specify adequate test cases for the decision table you built.
d) Implement this business rule and the corresponding junit tests in Java. Hint: assume that the purchase list is provided as a List<Purchase> where Purchase is a Java interface defined as follows:

```
/**
 * A purchase in the home appliances store.
 */
public interface Product {
    /**
     * Tests whether the product is from the informatics section or not.
     * @return - true, if the product is from the informatics section, false otherwise
     */
     public boolean isInformatics();
     /**
      * Returns the value of the product.
      * @return - the value of the product, in cents
      */
     public int valueInCents();
     /**
      * Returns the weight of the product. (new!)
      * @return - the weight of the product, in grams
      */
     public int weight();
     /**
      * Returns whether it is the customer's birthday or not
      * @return - true if it is the customer's birthday, false otherwise
      */
     public boolean isBirthday();
}
```

4. When travelling in commercial flights, passengers may carry a small cabin bag. This bag must comply with the following restrictions on its dimensions. Consider the operation `isCabinBag`:




```
/**
 * Returns true if the dimensions comply with the allowed limits,
 * false otherwhise. The dimensions are expressed in centimeters.
 * @param width – the width of the bag in centimeters
 *                (should be <= 45)
 * @param height – the height of the bag in centimeters
 *                (should be <= 56)
 * @param depth – the depth of the bag in centimeters
 *                (should be <= than 25)
 * @return – true if the bag is a cabin bag, false otherwise
 */
boolean isCabinBag(int width, int height, int depth)
```

a) Define adequate equivalence classes for conducting tests using the equivalence classes partition approach.
b) How many tests must you create for achieving *weak equivalence class testing*?
c) How many tests must you create for achieving *strong equivalence class testing*?

In the following questions, please build, for each of them, a table with a test battery. For each test, please use an integer identifier, the inputs and the expected result.

d) Build a test battery for the weak equivalence class testing criterion.
e) Build a test battery for the strong equivalence class testing criterion.
f) Build a test battery using boundary value analysis
g) Build a test battery using robust boundary value analysis
h) Build a test battery using worst case boundary value analysis
i) Build a test battery using robust worst case boundary value analysis
j) Please implement a Bag class containing the `isCabinBag(int, int, int)`. Then, create a JUnit test suite for this method covering the simple case specified in f).

5. Mr. X. is a big fan of his soccer club. He is very keen on watching it playing in the UEFA Champions League final, in May. Unfortunately, Mr. X. is facing some problems with justice right now, and unemployed. But all is not lost. He does have a friend who could get him an invitation (including the ticket and the flight to Madrid, where the final will be held), provided that:

- Mr. X.'s team qualifies to the UEFA Champions League Final.
- Mr. X.'s friend is still in a position of getting Mr. X. an invitation by then, or
- Mr. X. has found a new job so that he can pay for attending the final himself
- Mr. X. is, in May:
    - Cleared of all his problems with justice, or,
    - Still waiting for his trial without any travel restrictions

Using the **decision table testing** technique to devise tests to check whether Mr. X is going to the final.

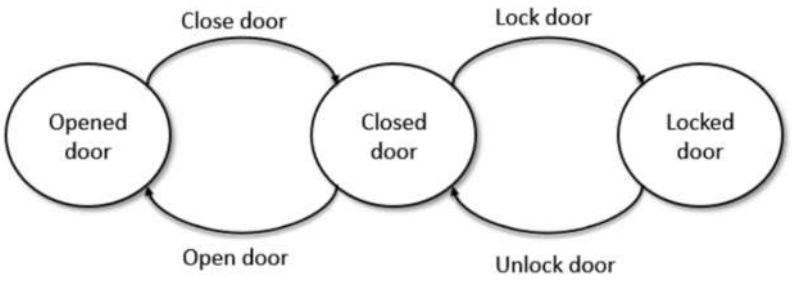
6. You have been hired to test the software in a clothes megastore. Consider a software module that is intended to accept the code identifier of clothes and a list of the different sizes those clothes come in, specified in integer values. The specifications state that the clothes name is represented by a string containing from 2 to 20 alphabetic characters, in length. This is followed by a string containing a comma-separated list of sizes. Each size may be a value in the range of 32 to 54, whole even numbers only. The sizes are to be entered in ascending order (smaller sizes first). A maximum of eight sizes may be entered for each clothes model. The clothes model name is to be entered first, followed by a comma, and followed by a list of sizes. A comma will be used to separate each size. Spaces (blanks) are to be ignored anywhere in the input. Here is a sample description of clothes (here, a shirt with all sizes from 32 to 40, except 38):

"shirt", "32, 34, 36, 40"

Create a set of test cases that fulfills these requirements using the **boundary value testing** approach (note: you may want to combine this with a contribution of equivalence class partitioning to weed out odd numbers).

a) Your first test battery should support simple boundary value analysis
b) Your second test battery should support robust boundary value analysis
c) Your third test battery should support worst case testing
d) Your fourth test battery should support robust worst case testing

7. Consider the following state machine, representing a door.



Use **state machine testing** to build a test battery for this door. Implement a door class and the corresponding test battery for it.

Part 2 – White box test cases generation

8. Consider the Java method `processGrades`. This operation receives a reference in of type `Scanner`, to read data from the standard input. The method reads a sequence of positive real numbers representing student grades, adding them, until it reads an invalid grade (a valid grade is a grade that is between 0 and 20 points: 0.0 <= validGrade <= 20.0). Then, it writes to the standard output the average grade, the average grade of the passing students, the number of passing students and the total number of students.

```
1  public static final double PASSING_GRADE = 9.5;
2  public static void processGrades(Scanner in) {
3       double totalGrades = 0.0;
4       double totalPassingGrades = 0.0;
5       int totalStudents = 0;
6       int passingStudents = 0;
7       double nextGrade = in.nextDouble();
8       while (nextGrade >= 0.0 && nextGrade <= 20.0) {
9          totalGrades = totalGrades + nextGrade;
10         totalStudents++;
11         if (nextGrade >= PASSING_GRADE) {
12            totalPassingGrades = totalPassingGrades + nextGrade;
13            passingStudents++;
14         }
15         nextGrade = in.nextDouble();
16      }
17      if (totalStudents > 0)
18         System.out.println("Average grade: " + totalGrades/totalStudents);
19      else
20         System.out.println("Average grade: 0.0");
21      if (passingStudents > 0)
22         System.out.println("Average passing grade: " + totalPassingGrades/passingStudents);
19      else
20         System.out.println("Average passing grade: 0.0");
17      System.out.println("Passing students: " + passingStudents);
17      System.out.println("Total students: " + totalStudents);
19 }
```

Using the white-box testing techniques covered in class, please:

a) Build a control flow diagram for the `applyDiscount` method.
b) Build a test battery (or the closest you can get to, and explain why it is impossible to do so, if necessary), for:
    1. Statement/node coverage
    2. Condition coverage
    3. Decision coverage
    4. Modified condition/decision coverage
    5. Independent path coverage
c) Build a data flow diagram for the `applyDiscount` method.
d) Create a test battery that ensures (or explain why it is impossible to do so, if necessary – in that case, create a test battery that is as close as possible to ensuring):
    1. All definitions coverage
    2. All uses
    3. All def-uses