Software testing

Unit test design - Exercise solutions

> **Exercise 1:**
>
> Design the white box test set for the following piece of code, using the *Basic path test* approach explained above. Determine the corresponding paths to be tested, and the test cases to test each path.

```
if (num1 > 10)
{
    if (num2 > 10)
        System.out.println("Both are greater");
    else
        System.out.println("First is greater");
} else {
    if (num2 > 10)
        System.out.println("Second is greater");
    else
        System.out.println("None is greater");
}
```

First, we add comments in each line indicating the possible bifurcations (1, 2, 3...) and sentences run (F1, F2, F3...):

```
if (num1 > 10) // 1
{
    if (num2 > 10) // 2
        System.out.println("Both are greater"); // F1
    else
        System.out.println("First is greater"); // F2
} else {
    if (num2 > 10) // 3
        System.out.println("Second is greater"); // F3
    else
        System.out.println("None is greater"); // F4
}
```

Then, the possible paths for this piece of code are:

1. 1, 2, F1
2. 1, 2, F2
3. 1, 3, F3

4. 1, 3, F4

We need to provide a set of input values to run each path. This can be a valid set of input values:

1. num1 = 20, num2 = 30
2. num1 = 20, num2 = 5
3. num1 = 5, num2 = 30
4. num1 = 5, num2 = 5

> **Exercise 2:**
>
> Repeat previous exercise using now the *Condition tests* approach.

Let's have a look again at the code, and identify every possible condition:

```
if (num1 > 10) // C1
{
    if (num2 > 10) // C2
        System.out.println("Both are greater");
    else
        System.out.println("First is greater");
} else {
    if (num2 > 10) // C3
        System.out.println("Second is greater");
    else
        System.out.println("None is greater");
}
```

Now, let's build the truth table for these conditions:

N	C1	C2	C3	Result
1	true	true	true	"Both are greater"
2	true	true	false	"Both are greater"
3	true	false	true	"First is greater"
4	true	false	false	"First is greater"
5	false	true	true	"Second is greater"
6	false	true	false	"None is greater"
7	false	false	true	"Second is greater"
8	false	false	false	"None is greater"

As you can see, we can get the same result from many pairs of cases:

- 1 and 2
- 3 and 4
- 5 and 7
- 6 and 8

So we can simplify previous table with just 4 different cases (we leave original numbers in first column to clarify this point):

N	C1	C2	C3	Result
1	true	true	true	"Both are greater"
3	true	false	true	"First is greater"
5	false	true	true	"Second is greater"
6	false	true	false	"None is greater"

> **Exercise 3:**
>
> The following piece of code checks if a number has its digits in ascending order:

```
boolean result = true;
while (number >= 10 && result)
{
    int lastDigit = number % 10;
    number /= 10;
    int newLastDigit = number % 10;
    if (lastDigit < newLastDigit)
        result = false;
}
```

> You are asked to design a test case table for every possible loop iteration, according to previous example.

In this case, we need to detect the limits of the loop, and design test cases for these limits. We assume we are only working with positive numbers. Otherwise, we could get the absolute value to discard negative numbers from the test cases.

- Loop is never iterated. This can be done with any number < 10. For instance, `number = 7` (result = *true*)
- Loop is iterated once. We can do this with either a two-digit number (`number = 23`) or with a number whose two last digits are not in ascending order, such as `number = 3352`. In this case, both tests could be interesting, so that we can check the two possible exits of the loop (according to the *while* condition)
- Loop is iterated twice. For instance, with `number = 234`
- Loop is iterated *m* times < *N*. We can do this with a number that, in the middle of its digits, breaks the ascending order, such as `number = 23452345`

- Loop is iterated *N-1* times. This can be the case of a number whose second digit is lower that its first digit, such as `43456`.
- Loop is iterated *N* times. We can get any number with ascending digits, such as `123456`.

So we could build this table for the test cases:

ID	Name	Data	Expected result	Actual result
U1	NoIterations	7	true	
U2	OneIterationTrue	23	true	
U2b	OneIterationFalse	3352	false	
U3	TwoIterations	234	true	
U4	MIterations	23452345	false	
U5	N-1Iterations	43456	false	
U6	NIterations	123456	true	

Exercise 4:

You have been asked to implement the tests for a class called *SalesList*, whose attribute is a `HashMap<String,Double>`. The string is the product description, and the number is the total amount of sales over this product. The class has the following methods:

- `addSale(String concept, int amount, double price)`: it adds a new element to the *HashMap* with the specified concept as product description. The incomes will be calculated by multiplying the amount and the price. It will return 0 if everything is OK, and -1 if there is any error. We will not be able to add sales with amount = 0 or price < 0, but we can add sales with negative amounts (but not negative prices).
- `getTotal()`: it will return the total sum of the incomes of the HashMap.
- `getAverage()` : it will return the income average.

Design the possible test cases for every method of the class. Regarding `addSale` method, you just have to complete the table shown in previous example. For `getTotal` and `getAverage` methods, you just need to set the preconditions to get the desired result, since they have no parameters.

Regarding `addSale` method, we go on with the table started in the unit contents. Again, we omit *Precondition* (which is the same for every test case) and *Actual result* columns for simplicity:

ID	Name	Steps	Data	Expected result
U1	Valid	Enter valid classes for concept, amount and price	concept="screw", amount=2,price=2	0, a new element is added
U2	NotValidConcept1	Enter empty string as concept	concept="", amount=2, price=2	-1, no element added
U3	NotValidConcept2	Enter string starting with number	concept="2screw", amount=2, price=2	-1, no element added
U4	NotValidConcept3	Enter string starting with special char	concept="@screw", amount=2, price=2	-1, no element added
U5	ValidAmount	Enter negative amount	concept="screw", amount=-2, price=2	0, a new element is added
U6	NotValidAmount	Enter amount of 0	concept="screw", amount=0, price=2	-1, no element added
U7	ValidPrice	Enter zero as price	concept="screw", amount=2, price=0	0, a new element is added
U8	NotValidPrice	Enter a negative price	concept="screw", amount=2, price=-2	-1, no element added

We could even go on and check values that are not numbers for price and amount, depending on the programming language on which we will implement the tests later.

Regarding `getTotal` and `getAverage` methods, we can use a table like this one (for both):

ID	Name	Precondition	Expected result
U1	ValidEmpty	*SalesList* object exists, but it is empty	0
U2	ValidNotEmpty	*SalesList* object exists, with at least one entry	The corresponding total/average
U3	NotValidNull	*SalesList* object is null	Exception thrown

> **Exercise 5:**
>
> Let's test a function that gets as input the day of a month (integer between 1 and 31) and a month number (integer between 1 and 12) and returns how many days are left in this month (an integer between 1 and 30, depending on the month).

```
int getDaysLeft(int dayOfMonth, int monthNumber) { ... }
```

> Think of the possible test cases to cover all the limit values.

Regarding the **input values**, we need to take into account both parameters (day and month):

- For the day, we need to check values 0, 1, 2 and also 30, 31, 32
- For the month, we need to check values 0, 1, 2 and also 11, 12, 13.

For the **output values** we need to take care of:

- Values -1, 0, 1
- Values 29, 30, 31

So we build the test case table considering all the combinations of these limits:

A possible test case table for this example could be this one:

ID	Name	Data	Expected result	Actual result
TC1	Day0	Day = 0, Month = 1	Error	
TC2	Day1	Day = 1, Month = 1	30	
TC3	Day2	Day = 2, Month = 2	26	
TC4	Day30	Day = 30, Month = 5	1	
TC5	Day31	Day = 31, Month = 6	Error	
TC6	Day32	Day = 32, Month = 7	Error	
TC7	Month0	Day = 1, Month = 0	Error	
TC8	Month1	Day = 10, Month = 1	21	
TC9	Month2	Day = 30, Month = 2	Error	
TC10	Month11	Day = 30, Month = 11	0	
TC11	Month12	Day = 20, Month = 12	11	
TC12	Month13	Day = 30, Month = 13	Error	
TC13	Output-1	Day = 31, Month = 9	Error	
TC14	Output0	Day = 30, Month = 9	0	
TC15	Output1	Day = 29, Month = 4	1	
TC16	Output29	Day = 1, Month = 4	29	
TC17	Output30	Day = 1, Month = 7	30	
TC18	Output31	Day = 0, Month = 5	Error	

Note that some of these test cases can be joint. For instance, if we use *Month = 1* for test cases 4 and 17 (and some others) we could simplify previous table.