Software Engineering WS 2022/23

Assignment 10: Quality Assurance 1

Hand-in via Moodle: Jan. 10, 14:00

Task 1–3 should also be implemented in source code and pushed into your GitHub repo, with unit testing using GitHub actions enabled (to realize continuous integration). Read the section Deliverables below for further information.

Task 1: White-Box Testing (1)

The following code excerpt shows a method for calculating scores in an imaginary software for assessing job applicants.

```
1   public static double calculateBaseScore(int[] scores, int base) {
2      double baseScore = 0.00;
3      int counter = 0;
4
5      //sum up the scores for each category
6      while (counter < scores.length) {
7         if (base > 0 && scores[counter] > 0) {
8               baseScore += scores[counter] / base;
9            }
10           counter++;
11        }
12     return baseScore;
13  }
```

(a) Describe meaningful test cases for this method. Create at least one test case for each of these four categories:

1. a positive logical test case
2. a negative logical test case
3. a positive concrete test case
4. a negative concrete test case

Describe the test cases using the template (slide “Test case: definition and description”) from the lecture slides, so come up with meaningful pre-/post-conditions, inputs and outputs. You can use the examples of logical and concrete test cases in the slides for inspiration.

(b) Implement the concrete test cases using JUnit. For the negative ones, invert the result, such that the test cases are also green.

(c) Create a control-flow graph for the method. Use the line numbers as labels.

(d) What is the minimum number of different test cases needed to achieve 100 % statement coverage (C_0)?

(e) What is the minimum number of different test cases needed to achieve 100 % branch coverage (C_1)?

(f) Calculate the statement and branch coverage for each of the two concrete test cases from subtask (a). Also specify the paths traversed from your control flow graph from task b).

Task 2: White-Box Testing (2)

Below is a method that calculates a number based on two parameters.

```
1   public static int number (int x, int y) {
2       int e;
3       if (x > y) {
4           e = x-y;
5       } else {
6           e = y-x;
7       }
8
9       if (e == 0) {
10          return 0;
11      }
12
13      int z = 0;
14      int i = 1;
15      while(i <= e) {
16          z += i;
17          i++;
18      }
19      return z;
20  }
```

(a) Create a control-flow graph for this method. Use the line numbers as labels.

(b) What is the minimum number of different test cases needed to achieve 100 % statement coverage (C_0)?

(c) What is the minimum number of different test cases needed to achieve 100 % branch coverage (C_1)?

(d) Describe these test cases using the template (slide “Test case: definition and description”) from the lecture slides. Implement them using JUnit.

(e) Calculate the statement and branch coverage for each test case. Also specify the paths traversed from your control flow graph.

(f) What does the method number() calculate? Write it down as a mathematical expression.

Task 3: Black-Box Testing

```
1  public class StringAnalyzer {
2      public static String convertAlphaNumeric(String input) {
3          // The code of this method is unknown...
4      }
5  }
```

The method takes arbitrary strings as a parameter. It examines which letters from [a-z,A-Z] (the letters from a–z in upper or lower case) and which numbers from [0-9] are present in the input string. All other input characters are ignored. The return string is sorted, such that the numbers are listed first in ascending order followed by the letters in alphabetical order. Each element is only listed once, i.e. duplicate elements are ignored, and uppercase letters become lowercase letters.

a) Come up with 5 equivalence classes that would be wise to test for this method.

b) For each equivalence class, specify at least one input string and the target result in a table. The table should have the following columns: Test case number, input, equivalence class, and output (expected result).

c) Implement suitable test cases in JUnit for each equivalence class.

d) After you created the test cases, they will obviously fail. Implement the method until all tests are green.

Task 4: Testing at Google

Read chapters 11 and 12 of the book *Software Engineering at Google*. Prepare a presentation of 5min length. In the first 2–3 slides you should summarize the testing situation at Google (chapter 11). Here, you have a lot of freedom, try to make it interesting for the audience. In the rest of the slides, you should briefly discuss the unit-testing principles from chapter 12 (each of which has a small subsection) and whether/how they apply to the test cases you implemented in Tasks 1–3. If you violate a principle, briefly explain why. Be prepared to present the slides and for a discussion on efficient testing in the tutorial.

Deliverables

Upload the following file in Moodle by the deadline:

- A single PDF document containing your solutions for all tasks. Use one page for each diagram. The document should not contain more than 6 pages (use font size 12).

Push your solution for Tasks 1–3 to Github (https://classroom.github.com/a/eYa_y3a6) by the deadline. Implement the given code snippets and your tests (using JUnit) in a Gradle project. Implement each task in different classes for the implementation and tests. Configure GitHub Actions in your repository to automatically run your test cases when you make a push. *Make sure that all your test cases succeed by the deadline.*

Note:

- The PDF document header needs to contain the author names with student IDs (“Matrikelnummern”).
- Add a README.md to your repository that contains the author names with student IDs (“Matrikelnummern”).