Lecture "Software Engineering SoSe 2023

Free University of Berlin, Institute of Computer Science, Software Engineering Group Lutz Prechelt, Linus Ververs, Oskar Besler, Tom-Hendrik Lübke, Nina Matthias

Exercise sheet 10 **Analytical quality assurance (1)** for 2023-06-26

Task 10-1: Terms

a) Explain the difference between *verify* and *validate*.

b) What does a *test case* consist of? When is a test case called *successful* and what is the idea behind it?

c) How are *failure*, *error* and *defect* related?

d) Explain in each case the similarities and differences between

1. Structure test and review
2. Load test and stress test
3. Testing and debugging
4. Function test and acceptance test
5. Top-down testing and bottom-up testing

Task 10-2: Apply and evaluate testing techniques

Assume the following natural language interface description of the opera- tion `classify triangle:`

> **classifyTriangle(int side1, int side2, int side3)** determines for a triangle described by its integer edge lengths whether it is **equilateral**, **right-angled**, **isosceles,** or **normal.** The strictest possible characterization is used.

Consider your *definitions of terms* from Task **10-1 -** especially the definition of a *test case* - when working on the following subtasks. Assume that both the system and the test cases are implemented in Java.

a) Specify the precondition for the `classifyTriangle` operation in OCL.

b) **Black box**: Create at least seven *test cases* based on the interface description of the operation alone by considering different cases which differ in that you expect a *different* behavior of the system in each case; avoid unnecessary duplications (keywords: equivalence classes and edge cases). Do not restrict yourself to valid inputs only (keyword: error cases).

List the test cases in a table. For each test case, briefly describe why you included it in the set of test cases.

c) **White-box**: Now create test cases based on the *structure of* the program, with the goal of achieving *branch coverage* (C_1), i.e. that at each branch in the program both branches were executed.

On the second page of this exercise sheet you will find an implementation for `classifying triangles` in Java: Since for a complete branch coverage each control condition must have been false at least once and must become false once, first determine those places in the program where there are branches and list them.

Now search for test cases so that each branch in the program becomes active once. Again, create a tabular overview of the test cases and indicate in each case why you have included this test case in the set.

d) Add all the test cases created in **b)** and **c)** together to form a test case set and mentally execute the test cases. Which of your test cases are *successful*?

Describe any defects you found in the program this way.

e) Have you achieved *statement coverage* (C_0)? How suitable do you consider the two coverage criteria C_0 and C ?$_1$

f) How suitable do you think the two applied test case generation methods functional test and structural test are?

g) Are there other failure modes that the test cases did not uncover? How did you discover them? What is this "discovery technique" called?

Task $^{10\text{-}3Æ}$: Test your own code?

a) One of the "golden" rules in the field of software development is:

> *"A programmer should not test his/her own code."*

What problem would you face if you did not follow this rule? Explain two different problems.

Discuss: Does this rule make sense?

b) Research and explain what is meant by *Test-Driven Development* (TDD).

Explain how TDD helps with the issue raised in **a).**

Implementation for task 10-2, subtask c)

```
class Math {
  enum TriangleType { Right Angle, Isosceles, Equilateral, Normal }

  public TriangleType classifyTriangle (int side1, int side2, int side3) {
    if ((page1  ==  page2)  ||
        (page2  ==  page3)  ||
        (page1 == page3))
          return TriangleType.Isosceles;

    if ((page1  ==  page2)  &&
        (page2 == page3))
        return TriangleType.Equilateral;

    int  quad1  =  page1  *  page1;
    int  quad2  =  page2  *  page2;
    int quad3 = page3 * page3;

    if  ((quad3  +  quad2  ==  quad3)
        ||   (quad1    +   quad3   ==
        quad2)  ||  (quad3  +  quad2
        == quad1))
        return TriangleType.Rectangular;

    return TriangleArt.Normal;
  }
}
```