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 11 **Analytical quality assurance (2)** for 2023-07-03

Task 11-1: Screening

On the last page you will find a checklist for reviewing Java code.

a) Which of the *checks* listed there do you yourself occasionally violate in your programming work?

b) List at least one item that cannot be checked automatically and explain why.

c) Name at least one point that reveals defects whose potential impact (= failure modes) is difficult to detect by testing and explain it.

d) Research other code review checklists and add at least three items that you consider important to the checklist here. Give reasons for your choice.

e) What are the advantages of screening in general compared to dynamic me- thods, i.e. tests? Describe at least three such advantages.

Task 11-2Æ: Apply and evaluate test techniques

In 1968 Edsger Dijkstra wrote a short but famous note about the danger of jump instructions in programming languages under the title *"Goto considered harmful".*

a) An Internet search will let you find this article quickly. Read it.

b) Characterize his argumentation and give it in broad outline. Do you agree with it and find it convincing?

Task 11-3: Test case creation with equivalence classes

One form of black box tests are tests with equivalence classes. In software testing, equivalence classes are *disjunctive* sets of program inputs whose respective values lead to a *similar* behavior of the software. Since these are black-box tests, similar *presumed* behavior is meant here.

Software has already been developed for the following problem. You are tasked with a black box test. Since the total number of all possible test cases is too high for you, you decide to make a selection by means of equivalence class formation.

> The grading of a university course is made up of two exams. The first exam was worth a maximum of 40 points, the second 60 points, for a total of 100 points. The following grading scheme applies:
>
> - Students who score less than 20 points in one of the two exams will fail (grade F).
> - Students who score at least 20 points in each of the two exams will receive at least a grade of D. (There is no grade of E.)
> - A grade of C is awarded for 60 percent or more of the total points.
> - A grade of B is awarded for 75 percent or more of the total points.
> - The top grade A is awarded to those who achieve at least 90 percent of the total points.
>
> Only whole points are awarded. Students always take both exams.

The software system expects two integer inputs for the scores of the two exam results and returns a grade.

a) Stick to the definition above and define equivalence classes for this problem. In particular, make *explicit* your underlying *assumptions* about the internal behavior of the program.

Make sure that your equivalence classes completely cover the input space. If you are unsure, make a graphical sketch of the two-dimensional space spanned by the two points.

You can assume that only valid entries (e.g. no more than the respective maximum score, no negative entries) are made.

b) Formulate exactly one test case for each of your equivalence classes.

An extension of the equivalence class method is the *boundary value approach*. At the boundary between two equivalence classes, the (presumed) behavior of the softwa- re "flips". In a one-dimensional boundary case approach, two test cases would arise: one each to the left and right of the boundary. With *multidimensional* equivalence classes (in this task: two dimensions) the boundaries become more complicated.[1]

Instead of a complete limit analysis, we restrict ourselves here to the simple transitions between two equivalence classes each.

c) Get an overview of the boundary courses of your equivalence classes from task **a)**. Define exactly two test cases for each uninterrupted boundary between two classes: One for the first class, one for the second class.

For illustration purposes, here is a representation of two-dimensional equi- valence classes (rectangles) with the test cases highlighted (circles):

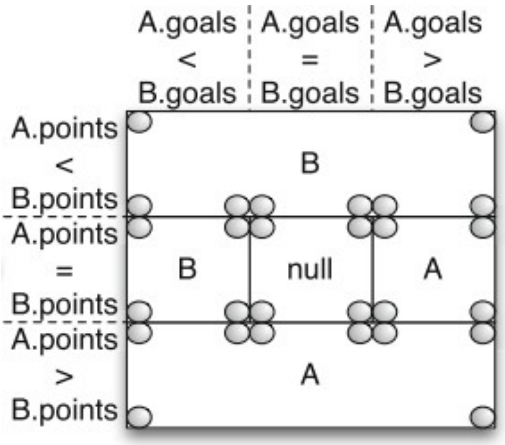
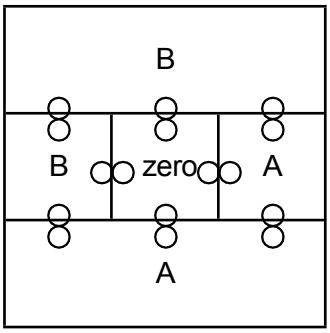


Figure 4.17 from Hoffmann (2013), with complete boundary case analysis



Analog scenario, with simplified limit case consideration

d) Install the "Eclipse IDE for Java Developers" (http://www.eclipse.org) on your computer. Download the prepared Java project `grader-project.zip` from KVV and import it into Eclipse ("Import existing project" and select the ZIP file). The functionality described above is implemented in the method `Grader.grade(int firstExam, int secondExam)`; it is available only as a library, not in source code.

Implement the test cases of tasks **b)** and **c)** as JUnit test cases in the existing test class `GraderTest.java`. Group your test cases sensibly into test methods and name them meaningfully. Include your test class.

[1] If necessary, read e.g. Section 4.3.1 of Hoffmann's "Software-Qualität" (2013); however, the aspect of dependent parameters relevant here is excluded there. If you are interested, you can close this gap with Section 4.2 of Kleuker's "Qualitätssicherung durch Softwaretests" (2013). Both books are available in full text from the FU network at http://link.springer.com.

Run the test cases: Did they reveal any failures? If so, make at least two hypotheses about the underlying defect(s). Use the items on the checklist below as a guide: Your two hypotheses must refer to two different points; please indicate the corresponding points.

Java program review checklist

Arithmetic

1. Are overflow or underflow possible during the calculation?
2. For expressions with more than one operator: Has the execution order been respected?
3. Were parentheses used to avoid ambiguity?
4. Is "Division by Zero" possible?
5. Do we wrongly assume that floating point arithmetic is exact?

Grinding

6. Are all variables involved correctly initialized before a loop?
7. Are all loops finished in any case?

Branches

8. Are the comparison operators correct? (<, <=, >, >=)
9. Are all `else` branches handled correctly? If an if-condition is missing the else-branch, is the case handled correctly if the `if-condition` is not fulfilled?
10. Does every switch `statement` have a `default` case?
11. Are missing break `statements` in switch `blocks` correct and commented separately?

Data flow

12. Can a variable be `zero` under certain circumstances and is this case caught?
13. Are parameter values checked for valid value ranges (according to preconditions)?
14. Do objects have to be compared with `equals()` or directly with `==`?
15. Can type matching (casting) fail?

Arrays

16. Is indexing of arrays outside the valid range possible?

Function calls

17. Does the caller of a method respond to all possible values that can be returned, including exceptions?
18. Does the caller of a method meet the prerequisites for the method's parameters?
19. Can stack overflow occur with recursive functions?

Multithreading

20. Are all accesses from multiple threads to the same variables synchronized? Does the variable have to be defined as `volatile` be declared?
21. Is there a risk of jamming?
22. Do threads waiting with `wait()` get woken up at some point?
23. Are `InterruptedExceptions` handled?
24. Is accessing a variable in two separate `synchronized` blocks from the same thread and is that correct?
25. Is an object shared (set to `null`) by one thread and accessed by another?

Exception handling

26. Does the control flow continue properly when an exception occurs or is caught?
27. Are any exceptions caught when accessing external resources?
28. Are all exceptions that can be raised by called methods caught?

External resources

29. Are streams and external resources (database connections, sockets, etc.) closed again?
30. Is buffered data "flushed"?
31. Are exceptions handled correctly during input and output?