Project Assignment 3 & 4: Class Testing, Test Case Implementation, and Results

Due: Friday, 12/02/2022, 23:59h

In this project milestone, you will add appropriate test levels for class testing to your existing Quality Management Plan. Herein, you will amend your report to include justifications, like:

- What testing techniques are you applying?
- Why are you applying the techniques in the chosen order?
- What techniques from the quality management plan could not be applied? Why?
- What techniques did you apply in addition to those stipulated by the assignments of the previous milestones? Why?
- Did you find any defects? What are these defects and how can they be fixed?
- What other information should you provide to make a third, uninformed party understand what you did and why you are confident that testing is sufficient?

Task 1

Find the file you created for Project 2 containing the description of your project. Append the solutions for this project into the same file.

Task 2

Correct all the mistakes and incorporate all the feedback you received on Project 2.

Task 3

Add another section to the existing Quality Management Plan titled “Class Test Strategy,” e.g. after the section on source-code testing techniques. To the introduction of this section (i.e., before the first subsection), add some text outlining the following:

- Think about the modality of the class your component is implemented by. Is your class non-modal, uni-modal, quasi-modal, or modal? State the answer and explain why.
- Given your modality, what test techniques do you recommend to apply, and in which order? For example, you could apply the Class Test Strategy according to Binder, as discussed in class. Or, if that is not applicable (e.g., because there are no class variables, polymorph calls, etc.), you can come up with your own strategy. When selecting a strategy, make sure it is (a) useful, and (b) makes optimal use of your findings from Milestones 1 and 2.

Task 4

Now derive a set of test cases for your component to in order to be able to implement an Object-Oriented Test according to your strategy from Task 3. Do the following:

a) ***Method Scope Test***. For each method, create a subsection. List test cases to satisfy:

   I. **Category Partition Test.** Identify primary functions and potential secondary functions (side effects) and write them down. Identify interface parameters and class variables. Reuse the equivalence classes and Boundary Value Analysis from Project 1. For interface parameters, then generate equivalence classes and Boundary Value Analysis for class variables, choose representatives, and create test cases accordingly.

   II. **Source-Code Test**. You may want to reuse the test cases for statement, branch, path, and all-uses coverage as well as Boundary Interior Test from from Project 2. List the test cases (including parameter values and expected outputs) you want to apply. For each test case, list which coverage criteria are being achieved. Reference your findings in previous sections for proof.

   III. **Polymorphism Test** for each non-primitive parameter, test the appropriate polymorphism, if any (this of course is not applicable, if there is no known sub- or super-class). Extend your control flow graph with polymorph calls as decisions, and derive test cases for each polymorph call.

b) ***Class Scope Test.*** Add another subsection and model the component state-space as an automaton. Apply state-based testing to find an execution order that satisfies "all events" for the modality property of your class according to your finding from Task 3. Write down the execution order as a sequence of method calls. You may use parameters from test cases from 4.a) or other milestones, if needed. If your class is non-modal, use the "alpha-omega" execution order.

c) ***Flattened Class Scope Test and Class Interaction Test***. If your method calls to other classes/methods, write down call sequences that satisfy the flattened class scope and class interaction test.

If your component does not allow for any of these tests to be conducted, write down why it can't be done.

Task 5

In every milestone thus far, you created a number of test cases, and – more importantly – a OO-test strategy for your class and method.

a) **Implement** your test cases from past projects honoring that test strategy. Use JUnit and make special use of `setUp`, `tearDown` and `assert`. Be sure to implement all test cases – that means you might need to create multiple test methods per method under test, i.e. one for each test case.

b) **Create** a test suite and **execute** all test cases. Provide a screen shot of the test summary JUnit gives you.

Task 6

Critically analyze the output.

a) Which test cases passed, which failed? Did you expect the test cases which passed to pass or the ones that failed to fail? Make a note of any irregularity you might find.

b) If some test case output didn't meet your expectation, first double check your test case and make sure it is correct. If you're convinces the test case is correct, then the code must have a defect.

c) For each detected defect, write down the failure, the possible fault, and the underlying error. Provide a suggestion how it could be fixed.

Task 7

Make a brief slide presentation consisting of 5-7 slides and be ready to justify why you successfully and sufficiently tested your project in class. Show defects that you found and how they can be fixed. Don't panic. Submit the slide show together with your Project 3&4 PDF as well as the JUnit code as a ZIP.