Miguel Ornelas-Morales

Instructor Ramsey Kraya

CS-320 Software Test Automation & QA

December 18, 2022

Project Two

Summary

My testing approach for Contact, Task, and Appointment was very similar. After declaring all the required variables in each category, I utilized specifically “if” statements to ensure that set requirements were met. In my Task.java class, the variables being assessed were taskId, taskName, and taskDesc. Each had a specific length restriction and could also not be null. With TaskTest.java, I designed a total of seven different tests to ensure this requirement. When further analyzing the coverage of both classes, I had 100% code coverage on Task.java, 100% coverage on each designed test, and 60.9% of the covered instructions in TaskTest.java.

With this being my first experience in JUnit testing, the ability to run the applications as a JUnit Test and with coverage allows me to see where specific lines of code were missed due to subtle errors in my code or errors in my logic. In one instant, I copied and pasted the previous test and changed the title. However, I had accidentally forgotten to change the specific variables. Upon running my test, the Eclipse JUnit test analyzed this as a passing run, but on the coverage side, lines of code were highlighted red as an indicator that the branches were missed. These queues helped ensure that my code was technically sound.

During each feature's development phase, I knew I had to fix some redundant code. In ContactService.java, I consolidated some of my code by providing a method that could be used to search apptIds within the arrays. This method allowed me to reduce the number of lines to add and delete an appointment with a unique Id. Going back and forth between JUnit and Coverage allowed me to see what was working and what needed extra attention quickly.

Reflection

The software testing techniques employed in Project One would fall under the black box and white box testing categories. The black box tests the functions or behaviors by studying the input and corresponding outputs. This method does not require knowledge of the internal design. A tester can use the software from the perspective of the end-user. White box testing examines the structure and internal logic for faults. The tests are more thorough and can be run earlier in development.

A software testing technique that I did not use for Project One was regression and experience-based testing. Regression testing ensures that the latest enhancements, fixes, or patches do not affect the system functionally. Experience-based testing is based on the tester’s experience with testing and developing similar systems or previous releases. One technique with this method is exploratory testing. In exploratory testing, the planning and execution of the test are done simultaneously and documented. This option provides a dynamic experience where the tester can react and adjust the test as needed.

With this being my first attempt at JUnit testing, my mindset was growth and extreme caution. I took extra time to play with the code, to assess how Eclipse handled specific lines. This approach was constructive, in one instance, when I had an assertEquals that failed, I could target the logic of specific lines in the class being tested and the test itself. After further review, most of the errors were due to either logic errors or the wrong type of assertion being utilized.

               Boni Garcia's book, Mastering Software Testing with JUnit 5, listed software testing principles and anti-patterns that helped limit personal bias and improve my discipline. When designing my test, I referred to two best practices: "tests should be simple" and "we should test positive and negative scenarios." Garcia discusses,

"Our test logic should be simple enough to avoid any kind of meta-testing, since this would lead to a recursive problem out of any logic. Indirectly, if we keep tests simple, we also obtain another desirable feature: tests will be easy to maintain." Furthermore, "We should test positive and the negative scenarios: This mean that we need to write tests with for input condition that assess the expected outcome, but we also need to verify what the program is not supposed to do. In addition to meet its requirements, programs must be tested to avoid unwanted side effects." (p. 286-287, 2017)

By following these practices, we can ensure that tests are guided and limited to a specific outcome to keep them simple and assessed to achieve the desired result. When testing Contact.java, each test was to anticipate one expected result. Weather to determine if a variable was null, the specificity of a length, or the validity of an array entry.

               Two anti-patterns I avoided for disciplined testing practices were "the test with no name" and "the secret catcher." For the test with no name, I ensured that each was self-explanatory and that each method name was, in laymen's terms, about what was being tested. So in ContactTest.java, my "void testContactIdToLong()" tested if the entry for uniqueId was greater than ten characters. The test naming scheme was simple and efficient. As for the secret catcher, this refers to "a test that is not making any assertion, relying on an exception to be thrown and reporting by the testing framework as a failure" (Garcia, p. 289, 2017). By avoiding this practice, we can ensure that each test is clear, concise, and repeatable.

We must provide quality code and unbiased reviews as software developers and testers. Although testing one's own code is not desirable, it can sometimes be unavoidable. We can ensure a better ROI and eliminate technical debt by not cutting corners. Code becomes more easily refactorable and in better condition, as changes and upgrades are introduced.

References

Garica, B. (2017). *Mastering Software Testing with JUnit 5.* Birmingham: Packt Publishing Ltd.