Jessica McAlum

SNHU CS 320

December 11, 2022

**Project Two**

**Summary**

**Describe your unit testing approach for each of the three features:**

My testing approach was directly aligned with the software requirements. The requirements helped me to better understand what I needed to test for, as far as how the code should function. Each JUnit test I created was to test that the code was functioning correctly to meet the requirements. For example, in the module four assignment, one of the requirements for the Task class was that a task object name cannot be longer than 20 characters. Therefore, I created a JUnit test that ensured that my code was behaving correctly by testing to see if an exception had been thrown if the object name was longer than 20 characters.

I would say that my JUnit tests were effective on the basis of coverage percentage. My JUnit test cases for each of my classes had a one hundred percent coverage for their associated class. This tells me that each instruction in my code had been tested for each class.

**Describe your experience writing the JUnit tests:**

To ensure that my code was technically sound, I included comments that explained the functionality of my code. I also made sure to name my JUnit tests that depicted what the tests were testing for. For example, to test that my code was behaving as expected when the task name was too long, I named the test, “testTaskNameTooLong()”.

To ensure that my code was efficient, I made sure that my JUnit test coverage covered all the instructions that were within the class it was testing. I also made sure to include assertions to assess every instruction included in each class. For example, my test for the method that deleted tasks was the following:

*@Test*

void testDeleteTask() {

TaskService taskService = new TaskService();

Task task1 = new Task("taskID", "taskname", "task description");

taskService.addTask(task1);

*assertEquals*(true, taskService.deleteTask("taskID"));

*assertEquals*(false, taskService.deleteTask("IDnotfound"));

}

The two assertions within this test were to test each possible instruction that was included in that method. These were to test that that the task had been deleted if the task ID already existed, and that the method could not delete a task if the task ID provided did not exist.

**Reflection**

**Testing Techniques:**

I employed white-box test techniques in this project through JUnit tests and analyzing the coverage of those tests. White-box test techniques allowed me to test that every instruction of code working as I had intended. I could also view my test coverage to ensure that every instruction of code was being executed. Deviations from the project design can be identified through white-box testing (Hambling et al., 2019).

I also employed specification-based techniques to validate the information being entered into the application. For example, in the Contact class, the requirements were that the first name could not be null or longer than 10 characters. I employed equivalence portioning techniques through JUnit tests by creating test cases to ensure that the program was accepting valid inputs and rejecting invalid inputs. “The equivalence partitioning technique takes advantage of the properties of equivalence partitions to reduce the number of test cases we need to write. Since all the values in an equivalence partition are handled in exactly the same way by a given program, we need only test one of them as a representative of the partition” (Hambling et al., 2019).

Some other techniques that were not used in for this project were decision table testing and state transition testing. Decision table testing is typically utilized when there are “business rules to define the functions of the system and the conditions under which each function operates” (Hambling et al., 2019). Decision table testing tests the behavior of the system resulting from various input combinations. The requirements for our system did not involve this type of functionality, so it was not utilized. State transition testing is used to evaluate if certain state transitions are taking place in response to certain conditions. State transition testing was not needed for this project because we were not required to develop functions that would trigger state changes.

**Mindset:**

The mindset I adopted working on this project was very focused on meeting the specified requirements. In acting as a software tester, I employed caution when writing my JUnit tests to ensure that I was including all the test cases I needed to check that my code was meeting the specified requirements. It was important to appreciate the complexity and interrelationships of the code I was testing, because having a strong understanding of them allows you to better determine what you should test and the best testing techniques to use. For this project, in order to write the tests, I had to first know how my code was intended to function and what inputs I needed to include in my tests so that I could properly test every instruction in my code. For example, one project requirement for the Contact class was that the address should not exceed 30 characters and it could not be null. Therefore, I wrote tests that ensured that the system was rejecting any input for an address that was longer than 30 characters or null, as well as accepting inputs for addresses that met the specifications.

When it comes to limiting bias, I tried to eliminate biases by trying to include as many tests as I could to test for any defects in my code. On the software developer side, I can definitely see how bias can be a concern if I were responsible for testing my own code. For example, I know how my code is intended to function and the inputs that are needed to produce certain outputs. In my mind, I only have what I think are the possible inputs, but that doesn’t mean that another person will have the same ideas as I do. It’s always advantageous to get an outsider’s perspective when it comes to your own work, because everyone has their own unique perspective that could provide valuable information to how you could improve your work.

It's important to be disciplined in your commitment to quality as a software engineering professional and to not cut corners when it comes to writing or testing code. Cutting corners in these areas will greatly increase the risk of defects and diminish the quality of the software. Cutting corners could lead to defects/bugs that are very costly to fix. I plan to avoid technical debt by doing my best to write clean code, continuously test my code throughout the development process, and collaborate with others to eliminate any bias. For example, in this project I practiced writing clean code by writing code in a way that was consistent, easy to understand and maintain. I also included helpful comments that explained what my code was doing.

**References**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.