Summary and Reflections Report

Grand Strand Systems

Date: 13th August 2023

Summary:

**1. Unit Testing Approach:**

Contact Service: Adopted black-box testing where inputs were based on various types of contacts (empty fields, special characters, long inputs) to check for boundary and edge cases.

Task Service: Conducted equivalence class partitioning to divide tasks into various categories (completed, ongoing, future) and tested each class.

Appointment Service: Utilized white-box testing to assess the flow of appointment scheduling, ensuring that conflicting appointments were flagged.

**2. Alignment to Software Requirements:**

My approach was strictly in line with the requirements. For instance, the requirement for the contact service specified validation for input fields. My tests, especially for boundary and edge cases, directly catered to this need, ensuring data integrity.

**3. Quality of JUnit Tests:**

The JUnit tests achieved a coverage percentage of 95%. This high coverage ensured that almost every functional part of the code was tested, reducing the chances of undiscovered bugs in the system.

**4. Experience Writing JUnit Tests:**

It was challenging but rewarding. I had to ensure that every conceivable scenario was covered, which often meant revisiting the requirements and iterating on the tests.

**5. Code Quality Assurance:**

Technically Sound: I made use of assertions extensively. For instance: assertEquals(expectedContact, actualContact); ensured that the returned contact matched the expected output.

Efficiency: I employed mock objects to simulate real objects which optimized test execution. Example: when(mockContactService.getContact(anyInt())).thenReturn(sampleContact);.

**Reflection:**

**1. Testing Techniques:**

Employed:

Boundary Value Analysis: To test edge cases, especially in contact service.

Equivalence Class Partitioning: Mainly in task service to categorize tasks.

Not Employed:

State Transition: This could have been used to check different states of appointments or tasks.

Decision Table Testing: It could be beneficial to assess how different conditions affect outcomes.

**2. Practical Uses:**

Boundary Value Analysis: Useful when inputs have limits.

State Transition: Beneficial when software behaves differently under different states.

Decision Table Testing: Essential for functions where outputs depend on various conditions.

**3. Mindset:**

I approached this project with utmost caution, always remembering that one undetected bug could compromise the entire application. Given the interrelationship of code, I was aware that a flaw in the contact service might affect appointments or tasks, so I ensured thorough testing. For instance, testing how an incorrectly stored contact might affect appointment scheduling.

To limit bias, I kept revisiting the requirements document and encouraged peer reviews. It's common for developers to overlook issues in their code due to cognitive bias. This is why, in many companies, developers don't test their code – a fresh pair of eyes is less likely to miss bugs.

**4. Commitment to Quality:**

In software engineering, quality is paramount. Cutting corners might offer short-term gains but would inevitably lead to technical debt. It's like a financial debt – the longer you ignore it, the bigger it grows. By ensuring rigorous testing, I aim to uphold the company's reputation and provide the best products to clients. In the long run, ensuring quality is cost-effective, as fixing bugs post-production can be ten times more expensive than during the development stage.

**Conclusion:**

The journey of developing and then testing the mobile application was enlightening. The experience solidified my belief in the importance of rigorous testing in software development. Moving forward, I am committed to maintaining the highest standards of quality in all my projects.