**Summary and Reflections Report**

**Christian Wallace**

**CS-320**

**08/25/2024**

**1. Summary**

**Unit Testing Approach**

**Contact Service:**

* **Approach:** I focused on testing that contact IDs are unique, non-null, and within the 10-character limit. I also ensured that the contact name and other fields met the length and non-null requirements.
* **Alignment with Requirements:** This approach directly ensured that the application met the requirements. For example, I verified that contact names did not exceed 50 characters, which aligned with the specified constraints.

**Task Service:**

* **Approach:** I tested task creation, ensuring valid IDs, non-null descriptions, and correct status updates.
* **Alignment with Requirements:** My tests ensured tasks met all functional requirements, such as preventing the creation of tasks with null descriptions.

**Appointment Service:**

* **Approach:** I validated that appointments had unique IDs, were not scheduled in the past, and adhered to all constraints.
* **Alignment with Requirements:** For instance, I wrote tests to ensure that appointments scheduled in the past were correctly rejected, aligning with the requirement only to allow future dates.

**Overall Quality of JUnit Tests**

* **Effectiveness of Tests:** The high-test coverage (over 90%) ensured that all critical scenarios were tested. This coverage effectively reduced the risk of undetected bugs.
* **Evidence from Code:** For example, my test cases confirmed that duplicate contact IDs were handled correctly, showing the robustness of the tests.

**Experience Writing JUnit Tests**

* **Technical Soundness:** I used assertions (e.g., assertEquals, assertThrows) to ensure expected outcomes, such as validating correct error handling.
* **Efficiency:** I optimized tests by using loops for repeated scenarios, reducing redundancy and increasing test efficiency.

**2. Reflection**

**Testing Techniques Employed**

**Unit Testing:**

* **Characteristics:** Tested individual components in isolation. Quick, automated, and focused on early bug detection.
* **Practical Uses:** Ideal for ensuring specific functions work correctly, particularly in projects with clearly defined logic.

**Test-Driven Development (TDD):**

* **Characteristics:** Wrote tests before implementation to define expected behavior.
* **Practical Uses:** Effective in projects where clear requirements and test coverage are critical.

**Other Software Testing Techniques Not Used**

**Integration Testing:**

* **Characteristics:** Tests the interaction between modules to ensure they work together.
* **Practical Uses:** Necessary for systems where multiple components or services must integrate seamlessly.

**System Testing:**

* **Characteristics:** Validates the entire system against requirements, including functional and non-functional aspects.
* **Practical Uses:** Critical in large-scale projects where end-to-end validation is required before deployment.

**Mindset**

**Caution in Testing:**

* **Approach:** I carefully considered how changes in one part of the code could impact others, such as how date validation in appointments could affect scheduling logic.
* **Example:** I added tests to ensure that deleting tasks did not affect other parts of the task list.

**Limiting Bias:**

* **Approach:** I focused on finding potential issues rather than confirming my code worked. I also used peer reviews to identify biases.
* **Example:** Peer review helped uncover potential blind spots in my tests.

**Commitment to Quality:**

* **Approach:** I maintained high standards to avoid technical debt, regularly refactoring code for clarity and efficiency.
* **Example:** I refactored the appointment service to improve readability and maintainability, ensuring long-term code quality.

**Conclusion**

In summary, the unit testing and TDD techniques were crucial in delivering a reliable application. Adopting a cautious and thorough approach helped ensure the final product met all requirements and maintained high quality.