**CS 320 Project Two: Summary and Reflections Report**

**Summary**

**Unit Testing Approach**

For this project, I implemented unit tests using JUnit for three core features of the mobile application: **Contact Service, Task Service, and Appointment Service**. Each feature was tested independently to ensure that it met the software requirements outlined by the customer.

* **Contact Service:** The unit tests for this feature validated that contact information (e.g., name, phone number, and address) adhered to input constraints. Boundary value analysis was used to ensure that inputs met expected length and format restrictions.
* **Task Service:** The task service unit tests focused on verifying that tasks could be successfully created, updated, and deleted. Tests ensured that descriptions did not exceed character limits and that IDs remained unique.
* **Appointment Service:** For the appointment feature, unit tests ensured that appointment dates could not be scheduled in the past and that appointment IDs were correctly assigned and maintained.

**Alignment with Software Requirements**

My approach to unit testing closely aligned with the software requirements by adhering to **functional correctness, input validation, and boundary testing** principles. For example, the test testInvalidPhoneNumberFormat() in ContactServiceTest.java ensured that invalid phone numbers (such as those exceeding 10 digits) were correctly rejected. This supported the requirement that phone numbers must adhere to a fixed-length numeric format.

**Effectiveness of JUnit Tests**

The effectiveness of my unit tests was evaluated based on **code coverage metrics**, which indicated a high percentage of covered statements and branches. By executing the tests with the JUnit framework, I confirmed that edge cases were handled properly and that unexpected inputs resulted in the appropriate exceptions being thrown.

**Experience Writing JUnit Tests**

Writing JUnit tests for this project reinforced the importance of **modular testing and automated validation**. To ensure technical soundness, I used **assertions** to validate expected behavior, such as:

@Test

void testValidContactCreation() {

Contact contact = new Contact("12345", "Bradly", "Grigg", "6233417239", "2964 Anderson St");

assertEquals("Bradly", contact.getFirstName());

assertEquals("Grigg", contact.getLastName());

}

To ensure efficiency, I minimized redundancy in test cases by implementing **parameterized tests** where possible, reducing duplicated assertions while maintaining coverage.

**Reflection**

**Testing Techniques Used**

For this project, I employed **black-box testing, boundary testing, and exception testing**:

* **Black-box testing:** Focused on verifying expected outputs without analyzing internal code structure.
* **Boundary testing:** Ensured values at the edges of constraints (e.g., minimum/maximum character lengths) were handled correctly.
* **Exception testing:** Verified that invalid inputs triggered appropriate error handling (e.g., attempting to create a contact with a null ID).

**Testing Techniques Not Used**

Other testing methods, such as **white-box testing** and **mutation testing**, were not implemented in this project. White-box testing, which involves analyzing internal code structures, could have provided additional insights into branch coverage. Mutation testing, which involves intentionally modifying code to ensure test robustness, would have further strengthened test effectiveness.

**Practical Implications**

Different software projects require different testing strategies. For instance:

* **Black-box testing** is useful for user-facing applications where functional correctness is the priority.
* **White-box testing** is beneficial for security-critical applications, where developers need to examine internal code structures.
* **Mutation testing** helps in high-reliability applications, such as medical or financial software, where test case robustness is crucial.

**Mindset and Approach**

Throughout this project, I maintained a **cautious and methodical mindset**, recognizing that even minor errors could lead to unexpected failures. For example, I carefully reviewed how input validation logic was tested, ensuring that all invalid cases were accounted for before assuming correctness.

To limit bias, I performed **independent code reviews** after writing my unit tests, approaching the code from a tester’s perspective rather than assuming it was error-free. If I had been testing my own code in a professional setting, **confirmation bias** could have been a risk—potentially leading me to overlook assumptions that seemed correct at first glance.

**Commitment to Quality and Avoiding Technical Debt**

Ensuring **code quality and long-term maintainability** is a fundamental responsibility of software engineers. Cutting corners during testing can lead to costly defects in production. For example, neglecting boundary testing could result in unforeseen errors when users input unexpected values.

To avoid technical debt, I plan to:

* **Use version control and continuous integration (CI)** to catch issues early.
* **Write comprehensive unit tests** that prioritize edge cases and error handling.
* **Conduct code reviews** to gain diverse perspectives on software quality.

By maintaining disciplined testing practices, I can contribute to the development of more **reliable, maintainable, and efficient** software systems.

**Conclusion**

This project reinforced the importance of structured unit testing and the impact of different testing strategies on software quality. By applying **black-box, boundary, and exception testing**, I ensured that the contact, task, and appointment services met their functional requirements. Additionally, reflecting on my testing mindset highlighted the significance of **caution, thorough validation, and reducing bias** when evaluating code. Moving forward, I will continue to refine my approach by integrating additional testing techniques and automation tools to enhance software reliability and maintainability.

**References**

* Gamma, E., Beck, K. (2005). *JUnit Recipes: Practical Methods for Programmer Testing.* Addison-Wesley.
* Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code.* Pearson Education.