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

**Summary**

1. **Unit Testing Approach for Each Feature**
   * **Contact Feature:** I wrote JUnit tests to validate adding, editing, and deleting contacts. For example, I tested invalid inputs like a name exceeding the maximum allowed characters and ensuring that a null phone number threw an exception.
   * **Task Feature:** My unit tests focused on edge cases such as task descriptions being too short or too long. I also tested scenarios where tasks had overlapping deadlines.
   * **Appointment Feature:** For appointments, I tested valid and invalid dates, ensuring that past dates could not be added and overlapping appointments were flagged.
2. **Alignment to Software Requirements**
   * My approach directly followed the project requirements. For instance, the requirement that a contact's phone number must be 10 digits long was tested using testInvalidPhoneNumberLength(). This ensured compliance with the specification.
3. **Overall Quality of JUnit Tests**
   * I used code coverage tools to measure test effectiveness and achieved over 90% coverage. For example, my test testAppointmentOverlap() identified an issue where overlapping appointments were not handled correctly. This was fixed and retested to confirm functionality.
4. **Experience Writing JUnit Tests**
   * **Technical Soundness:** I ensured correctness by using assertions like assertEquals and assertThrows. For example:

@Test

void testInvalidPhoneNumber() {

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

contactService.addContact("John", "Doe", "123");

});

assertEquals("Invalid phone number", exception.getMessage());

}

* + **Efficiency:** I avoided redundant tests by parameterizing inputs. For instance, a loop was used to test multiple invalid phone numbers:

@Test

void testMultipleInvalidPhoneNumbers() {

String[] invalidNumbers = {"123", "abc", ""};

for (String number : invalidNumbers) {

assertThrows(IllegalArgumentException.class, () -> contactService.addContact("John", "Doe", number));

}

}

**Reflection**

1. **Testing Techniques**
   * **Techniques Used:**
     + Unit Testing: Focused on small, isolated code units to ensure reliability.
     + Negative Testing: Tested invalid inputs to ensure robustness.
     + Boundary Testing: Checked edge cases like maximum and minimum field lengths.
   * **Techniques Not Used:**
     + Integration Testing: While helpful for checking interactions between modules, it wasn’t necessary since the project scope was limited to unit testing.
     + Stress Testing: Typically used to test system behavior under load, which was irrelevant for a mobile application backend.
   * **Practical Uses:**
     + Unit Testing is essential for early-stage development to catch bugs in isolated components.
     + Integration Testing is more suitable for complex systems, such as multi-module applications or microservices.
2. **Mindset**
   * **Caution in Testing:** I took a systematic approach to testing, ensuring edge cases and dependencies were considered. For example, while testing task deadlines, I ensured that invalid dates did not corrupt other tasks in the system.
   * **Limiting Bias:** To avoid bias, I reviewed the test cases with a peer and followed the requirements document rather than assumptions about how the code should behave. For instance, I initially overlooked testing for null input until a peer pointed it out during review.
   * **Commitment to Quality:** Cutting corners often leads to bugs that are harder to fix later. I ensured that all tests were modular and well-documented to minimize future technical debt. For example, my testAddContact method was broken into smaller test cases to make debugging easier.

**Citations and Attributions**

1. **JUnit Documentation**  
   The JUnit framework was instrumental in developing and executing unit tests for the application. The official JUnit documentation provided detailed guidance on assertions, annotations, and test structures.
   * JUnit 5 Documentation: https://junit.org/junit5/docs/current/user-guide/
2. **Course Materials**  
   Concepts and techniques for unit testing were referenced from course lectures and assigned readings provided in CS 320. Specific references include examples of boundary and negative testing strategies.
3. **Textbooks and Learning Resources**  
   Additional insights on testing techniques were derived from:
   * Beck, K. (2003). *Test Driven Development: By Example.* Addison-Wesley.
   * Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.
4. **IDE and Tools Used**
   * **IntelliJ IDEA:** Used for coding and testing. Features such as built-in JUnit integration streamlined the development process.
   * **JaCoCo:** Used for measuring test coverage to ensure comprehensive testing of the codebase. Documentation: <https://www.eclemma.org/jacoco/>