Mark Turner – 4/20/2025

CS 320 – Project 2

**Describe your unit testing approach for each of the three features.**  
In the Task service, I tested constraints with unit tests, such as unique, non-null IDs limited to 10 characters, ensuring the ID remained immutable. For example, testAddTask confirmed task creation, while testUpdateExistingTask ensured that updates did not modify the task ID. In the Contact service, I validated the addition, updating, and deletion of contacts while enforcing uniqueness of IDs and field constraints. Tests like testDeleteExistingContact and testUpdateContact helped verify this functionality. The Appointment service had limited unit tests but included checks for valid, non-null future dates and proper data input, laying the groundwork for more testing in future iterations.

**To what extent was your approach aligned to the software requirements?**   
My testing approach was closely aligned to the defined software requirements. For instance, in the Task class, I validated non-null and length-constrained fields using assertions like:  
assertThrows(IllegalArgumentException.class, () -> new Task(null, "Name", "Description"));  
and assertEquals("1234567890", task.getTaskId());  
I validated the ContactService using similar assertions like:  
assertThrows(IllegalArgumentException.class, () -> contactService.deleteContact("nonexistentId"));

**How do you know your tests were effective based on the coverage percentage?**  
I achieved 100% coverage for the Task and TaskService classes, and 100% coverage for the Contact class, with 94**%** for the ContactService. high coverage levels indicate that nearly all methods and logic branches were covered by tests, minimizing the risk of untested logic.

**Describe your experience writing the JUnit tests.**  
Initially, I missed some negative scenarios—such as deleting non-existent contacts or handling null updates—but writing tests helped me identify and resolve these gaps. Over time, I developed a more defensive and structured approach to testing, treating it as a core part of the software design process rather than a post-development task.

**How did you ensure that your code was technically sound?**  
I ensured technical soundness by writing clear, testable logic and validating them through specific assertions in unit tests. For example, I used:  
assertThrows(IllegalArgumentException.class, () -> taskService.addTask(new Task("123", null, "desc"))); to test invalid data input, and assertEquals("desc", taskService.getTask("123").getDescription()); to confirm that values were set and retrieved as expected. These tests confirmed that the code behaved correctly under valid and invalid conditions.

**How did you ensure that your code was efficient?**   
By using data structures like Hash tables and testing for edge cases, I became confident in my code’s efficiency. Assertions in my unit tests like: assertTrue(contactService.deleteContact("001")); allowed me to confirm success with a single, readable assertion, and assertThrows(IllegalArgumentException.class, () -> contactService.updateContact("invalidId", "New", "Last", "1234567890", "Address"));  
tested exception handling without extra setup. Reusing test data and keeping tests small also contributed to faster execution and easier maintenance.

**What were the software testing techniques that you employed in this project?**   
I used static testing, automated unit testing, and automated code coverage analysis. Static testing involved reviewing code structure and logical consistency before execution. Automated unit testing allowed me to test each service class (Task, Contact, and Appointment) in isolation, verifying valid operations and edge cases. Automated code coverage tools helped identify which parts of the code were not being tested, leading to more complete and confident test coverage.

**What are the other software testing techniques that you did not use for this project?**   
I did not use integration testing, which would check how the Task, Contact, and Appointment services interact as a system. I also skipped end-to-end (E2E) testing, which simulates real user workflows through the entire application, and acceptance testing, which verifies the software meets business needs—often involving stakeholders or users in the validation process.

**For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**  
Static testing is practical early in development for maintaining code quality and consistency. Unit testing is ideal for modular systems where features evolve independently. Code coverage tools help assure developers their software is appropriately tested. Integration testing is useful for projects where services share data or functionality, such as APIs or microservices. End-to-end testing is critical in full-stack development where user experience and interface reliability matter. Acceptance testing ensures alignment with stakeholder goals, making it essential in client-driven projects or software delivered under contract.

**In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing?**  
I adopted a cautious, detail-focused mindset throughout the project. I constantly questioned how the code might fail, especially in cases of invalid input, empty data, or repeated operations. For example, I tested deleting tasks that didn’t exist or updating fields with null values. This caution helped me uncover logic gaps and understand how the services could be misused or produce incorrect results, especially where input validation and ID uniqueness were involved.

**On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code?**   
To limit bias, I wrote tests to simulate misuse and invalid input, even when I felt confident about the logic. For example, I initially overlooked a null check in the Contact address field but caught the issue during execution of the unit tests.

**Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field?**  
Cutting corners in testing may save time upfront but leads to bugs, regressions, and higher maintenance costs. I ensured test completeness before marking features as done, including failure tests like those for updateContact. To avoid technical debt as a practitioner in the field, I’ll follow TDD, use CI test automation, and commit to regular reviews and refactoring.