**Project 2: Summary and Reflections Report**

Jaden B. Knutson  
CS 320 – Grand Strand Systems  
Date: 2/23/25

**Summary - Alignment to Requirements and Coverage Percentage:**

I used JUnit 5 for unit testing my Contact, Task, and Appointment service classes. JUnit 5 was used to test functionality, enforce constraints, and handle exceptions as required. My approach aimed to ensure a fully functional deliverable that met all assignment and project requirements by using JUnit tests, assertions, and test coverage analysis to verify that critical parts of my files were thoroughly tested.

My overall test coverage was above 80%, with most of the untested code consisting of output messages and test data used to check whether the correct exceptions were being thrown. For example, in TaskServiceTest.java, I validated adding, deleting, and updating tasks, ensuring each method had direct assertions and exception handling.

My tests aligned with software requirements by verifying valid input constraints and expected behaviors. One example is in AppointmentTest.java, where I ensured that IDs were unique, dates weren’t in the past, and that descriptions followed character limits. I also validated exceptions in ContactServiceTest.java, ensuring that duplicate contacts couldn’t be added. These tests confirmed that all major constraints were enforced, preventing invalid data entry and maintaining system integrity.

assertThrows(IllegalArgumentException.class, () -> new Appointment("J11223", pastDate, "Past appointment"));

assertThrows(IllegalArgumentException.class, () -> contactService.addContact(contact2));

**Summary - Experience Writing JUnit Tests, Technical Soundness and Efficiency of Code:**

To ensure my code was technically sound, I structured my JUnit tests using @Test and @Order to execute them systematically and efficiently. I used assertEquals, assertThrows, and assertNull to confirm that the functionality and results matched expectations. I also added debug statements throughout my code to track test executions and ensure that all testing scenarios, including valid, invalid, and edge cases, were properly covered.

System.out.println("Task Added: " + task);

To go through my code and ensure it was technically sound, I used a consistent strategy of validation checks, exception handling, and immutable IDs in constructors. For instance, in Contact.java, constraints prevented null values and incorrect data lengths:

if (phone == null || phone.length() != 10) {

throw new IllegalArgumentException("Invalid Phone Number!");}

This ensured strict data integrity across all objects.

I prioritized efficiency in my code by using HashMaps for quick lookups and modifications, reducing the need to search through lists. I also automated and streamlined the logic with exception-based validation checks instead of manual error checking, which made the code and testing process simpler, more concise, and easier to maintain.

private final Map<String, Task> taskMap = new HashMap<>();

**Reflection - Testing Techniques Used, Practical Uses, and Techniques Not Used**

For my testing strategy, I used various testing techniques to validate functionality and catch errors early. I used unit testing with JUnit 5 to verify individual components and ensure they met expected behaviors. Exception testing was crucial for handling invalid inputs and preventing issues like duplicate accounts. I applied boundary testing to check edge cases, such as minimum and maximum lengths for phone numbers, addresses, names, descriptions, and ID numbers. Exception testing helped isolate and confirm that invalid inputs triggered the correct errors. Boundary testing is especially useful in user input-heavy applications. I also used test coverage analysis to measure how much of my code was tested across my class and test files. This helped assess the effectiveness of my tests and identify areas for improvement. Test coverage analysis is especially useful in large-scale projects, where untested code can lead to unexpected bugs (Garcia, 2017).

While I used various testing techniques, there were several I did not incorporate. Integration testing was not used, but it would have been helpful in verifying how my services interacted and worked together. Since my files were module-based, system testing was also not used, as it focuses on full system validation rather than individual components. I also did not use performance testing, which would have measured the speed and scalability of my files. My focus was on functionality and validation rather than optimizing for performance (Garcia, 2017).

**Reflection - Mindset During Testing**

With learning and implementing these testing approaches, I approach them with a great deal of caution and curiosity. I like to tinker with things just to see what works and what doesn’t, so I’m constantly testing my code in small sections as I write it. Early on, I realized how important it is to pay attention to detail, knowing that even the smallest bug could break functionality now or cause bigger issues later if not caught early. As I’ve learned in this class, it’s always best to **find bugs and errors early** rather than later when they would take more time and effort to fix. One example is preventing **duplicate immutable IDs** from being created. If duplicates were allowed, it could create integrity issues, such as two different people or appointments being assigned the same ID. This could lead to errors or even **two different people showing up for the same doctor’s appointment**.

assertThrows(IllegalArgumentException.class, () -> taskService.addTask("12333", "Task Two", "Description Two"));

**Reflection - Limiting Bias in Testing, Code quality, and Avoiding Technical Debt**

Bias can be a concern when testing your own code since you might assume it works as intended, which can lead to overlooked areas and potential bugs. To avoid this, I used predefined test cases instead of ad-hoc testing, ensuring consistency in my approach. I also tested edge cases, such as empty descriptions or null values, to catch unexpected behavior. Additionally, I added negative tests to confirm that the program handled errors correctly and prevented invalid operations.

Cutting corners leads to long-term issues. Technical debt builds up when errors aren’t fully addressed, or tests aren’t thorough. To avoid this, I fully implemented validation checks in constructors, used exceptions instead of silent failures, ensured high test coverage to catch edge cases early. For example, enforcing immutable IDs in Appointment.java prevents accidental modifications:

private final String appointmentId;

The biggest thing to remember is that cutting corners leads to bugs and errors, and if those errors aren’t caught early, they become more costly and time-consuming to fix as development progresses. By following these principles, I ensure code reliability while minimizing future maintenance costs.

**References:**

García, B. (2017). Mastering software testing with JUnit 5. Packt Publishing.

Hambling, B., Morgan, P., & Samaroo, A. (2019). Software testing: An ISTQB-BCS certified tester foundation guide (4th ed.). BCS Learning & Development Ltd.