Thomas Cheek

Prof. Luo

CS 320

February 23, 2025

**Summary**

For the Contact Service, Task Service, and Appointment Service features, I employed a comprehensive unit testing approach to verify that each service met its specified requirements. My strategy involved testing individual components in isolation, focusing on ensuring that each method behaved correctly with valid inputs and appropriately handled invalid inputs by throwing exceptions. Using JUnit to automate these tests provided consistent and repeatable results, aligning with best practices in software testing as outlined by the IEEE Computer Society (2023).

My testing approach closely aligned with the software requirements. For example, the Task Service required that task names be no longer than twenty characters, and descriptions be no longer than fifty characters. To verify compliance, I wrote tests that provided valid inputs within these limits as well as invalid inputs exceeding them. Similarly, for the Appointment Service, I ensured that appointments could not be scheduled in the past by using Java's Date class with before(new Date()) to check for invalid dates. In the Contact Service, I verified that phone numbers were exactly ten characters long, with tests to catch both shorter and longer inputs. These targeted tests ensured that each validation rule was strictly enforced, aligning with the project specifications.

The overall quality of my JUnit tests is demonstrated by their comprehensive coverage of both valid and invalid scenarios. By testing both positive and negative cases, I ensured that each method functioned correctly and responded appropriately to errors. The use of assertions like assertEquals and assertThrows allowed me to verify expected outputs and exceptions. For instance, in the Appointment Service, the test assertThrows(IllegalArgumentException.class, () -> new Appointment("12345", pastDate, "Checkup")) confirmed that creating an appointment with a past date correctly threw an exception. This aligns with best practices in software testing, which emphasize identifying and addressing issues early in the development process to prevent defects from reaching production (Rouse, 2023).

Writing the JUnit tests was a valuable experience that helped me appreciate the importance of thorough testing in software development. I ensured that my code was technically sound by using clear, concise assertions to verify each method's behavior. For example, in the Task Service, the test assertEquals("Updated Description", task.getDescription()) verified that the setDescription method correctly updated a task’s description. Similarly, in the Contact Service, I confirmed that invalid phone numbers were rejected using assertThrows(IllegalArgumentException.class, () -> new Contact("John", "Doe", "1234", "123 Main St")). These tests provided concrete evidence that my code met all functional requirements.

To ensure that my code was efficient, I wrote streamlined tests that focused on essential functionality without unnecessary complexity. For example, in the Appointment Service, I added and deleted appointments using minimal code while still verifying that each action succeeded. The test assertEquals(0, service.getAppointments().size()) confirmed that deleting an appointment removed it from the list, while assertEquals(1, service.getAppointments().size()) verified that adding an appointment increased the list size. This approach kept the tests concise and easy to maintain, promoting long-term code quality and aligning with industry standards for efficient software testing (IEEE Computer Society, 2023).

**Reflection**

Throughout the project, I used several software testing techniques to ensure that the services met their requirements. Unit testing was my primary technique, as it allowed me to test individual methods in isolation. By using JUnit assertions, I could verify that each method produced the expected output and handled errors correctly. Boundary value testing was also essential, particularly for fields with length constraints. For example, I tested task names with exactly twenty characters and descriptions with exactly fifty characters to ensure that the upper limits were enforced. Additionally, I performed exception handling testing by providing invalid inputs to verify that the code threw appropriate exceptions. This technique ensured that errors were caught and handled gracefully, preventing unexpected crashes (Rouse, 2023).

There were some testing techniques that I did not use in this project. Integration testing, which verifies that different components of a system work together, was not necessary since each service was tested independently. System testing, which evaluates the entire software system, was also not required because the project focused on individual services rather than a complete application. Performance testing, which measures how well a system performs underload, was not applicable since the services used simple in-memory data structures and did not require optimization for large datasets. However, in more complex systems, integration and performance testing would be essential to ensure that all components interact correctly and that the system can handle expected workloads (IEEE Computer Society, 2023).

Each testing technique has practical applications in different software development contexts. Unit testing is essential for verifying that individual components work as intended, making it a fundamental part of any development process. Boundary value testing is particularly useful for applications that enforce strict input constraints, ensuring that edge cases are handled correctly. Exception handling testing is crucial for systems that must gracefully handle invalid inputs, improving reliability and user experience. Integration testing becomes important when multiple components interact, ensuring that they function correctly as a whole. System testing is essential before deploying a complete application, as it verifies that all components work together as expected. Performance testing is critical for applications that must handle high loads, ensuring that they remain responsive and reliable under stress (Rouse, 2023).

The mindset I adopted while working on this project was one of caution and attention to detail. As a software tester, it was essential to anticipate potential errors and verify that the code handled them appropriately. For example, I was careful to test both valid and invalid inputs for each method, ensuring that the code behaved correctly in all scenarios. This cautious approach helped me identify and fix potential issues before they could cause problems in production. It was also important to appreciate the complexity and interrelationships of the code I was testing. For instance, deleting a contact from the Contact Service needed to be verified to ensure that it did not affect other contacts in the list. Similarly, updating a task's description in the Task Service had to be tested to ensure that it did not inadvertently alter other task properties. Understanding these relationships helped me write more comprehensive and effective tests.

To limit bias in my code review, I approached testing with a mindset of critical evaluation. I deliberately tried to break the code by providing invalid inputs, ensuring that it could handle unexpected scenarios. This approach helped me identify edge cases that might not have been apparent during development. I also considered the potential for bias if a developer were responsible for testing their own code. For example, a developer might assume that their input validation is sufficient and fail to test edge cases like excessively long task names or past appointment dates. By approaching testing from an objective perspective, I was able to identify and address these issues, improving the overall quality of the code (IEEE Computer Society, 2023).

Maintaining a commitment to quality is essential in software engineering. Cutting corners during development or testing can lead to bugs and system failures, which can have serious consequences in production. For example, failing to validate appointment dates in the Appointment Service could result in scheduling errors that disrupt users' plans. Similarly, allowing duplicate task IDs in the Task Service could cause data corruption and make it difficult to track individual tasks. To avoid technical debt, I plan to follow best practices for writing clean, maintainable code and thoroughly testing each component. By writing comprehensive unit tests and verifying that they cover all possible scenarios, I can ensure that my code is reliable, maintainable, and free of unexpected errors (Rouse, 2023).

In conclusion, the software testing techniques I used in this project provided a solid foundation for verifying the functionality and reliability of the Contact Service, Task Service, and Appointment Service. By using unit testing, boundary value testing, and exception handling testing, I was able to ensure that each service met its requirements and handled errors gracefully. Maintaining a cautious mindset and limiting bias helped me identify and address potential issues, while my commitment to quality ensured that the code was both reliable and maintainable. As I continue my career in software engineering, I will continue to apply these principles to deliver high-quality software that meets users' needs and performs reliably in real-world scenarios.

**References**

IEEE Computer Society. (2023). *The importance of software testing.* Computer.org. <https://www.computer.org/resources/importance-of-software-testing>

Rouse, M. (2023). *Software testing.* TechTarget. <https://www.techtarget.com/whatis/definition/software-testing>