Summary and Reflections Report: Unit Testing for Contact, Task, and Appointment Services

Mubasil Choudhry  
Southern New Hampshire University  
CS-320: Software Testing  
Professor Antoun  
October 19, 2025

Summary

For this project, I developed and tested three separate services: the Contact Service, Task Service, and Appointment Service. My unit testing approach for each feature was designed to ensure that all customer requirements were satisfied. For the Contact Service, I implemented tests that validated each field’s constraints. Specifically, the contact ID was tested to ensure it was unique, non-null, and no longer than ten characters. Additional tests confirmed that firstName and lastName fields were non-null and no longer than ten characters, the phone field consisted of exactly ten digits, and the address was non-null and limited to thirty characters. The JUnit test class ContactTest verified these constraints, while ContactServiceTest validated that the service correctly added, updated, and deleted contacts.

For the Task Service, I took a similar approach. Tests were implemented to ensure the task ID was non-null, not updatable, and shorter than or equal to ten characters. Tests also verified that the task name was non-null and no longer than twenty characters, while the description field was non-null and no longer than fifty characters. TaskTest ensured that validation logic was enforced, and TaskServiceTest confirmed that tasks could be added, updated, and deleted correctly while preventing duplicates.

For the Appointment Service, my tests placed special emphasis on date validation. The requirements specified that an appointment date could not be null or set in the past. Therefore, I implemented tests that created both past and future dates to confirm that only future dates were accepted. AppointmentTest validated these date constraints, tested description limits, and ensured that appointment IDs met the required constraints. AppointmentServiceTest then confirmed that appointments could be added with unique IDs and deleted when requested.

Overall, my testing approach aligned closely with the requirements. Each requirement listed in the rubric was explicitly translated into test cases, ensuring comprehensive coverage. For example, the requirement that phone numbers must be exactly ten digits was tested using both valid and invalid inputs, such as nine-digit, eleven-digit, and alphanumeric strings. This demonstrates a clear alignment between the customer’s requirements and my testing strategy.

The quality of my JUnit tests can be defended based on their thoroughness and coverage. With over 80% test coverage across all classes, the tests effectively validated both happy paths and failure cases. For instance, the following test ensured that invalid phone numbers were correctly rejected: assertThrows(IllegalArgumentException.class, () -> c.setPhone("12345"));. Such lines confirm that validation was not only implemented but also enforced through testing.

My experience writing JUnit tests was positive and reinforced the importance of testing as a parallel activity to development. The tests helped me identify edge cases early, such as overly long strings or null values, which otherwise might have gone unnoticed. Writing these tests also gave me practice in applying exception-based assertions such as assertThrows, which are essential for validating input constraints.

To ensure that my code was technically sound, I wrote tests that attempted to break the program through invalid inputs. For example, in ContactServiceTest, I verified that duplicate IDs were rejected by calling addContact with two contacts that shared the same ID. By doing this, I confirmed that the logic preventing duplicates was functioning correctly. Additionally, I ensured efficiency by writing helper methods such as futureDate() in AppointmentTest to generate consistent test data, reducing redundancy and improving clarity.

Reflection

The primary software testing techniques that I employed in this project included unit testing, boundary testing, and exception testing. Unit testing was the foundation, focusing on small, isolated components such as Contact, Task, and Appointment classes. Boundary testing was used to ensure that field limits, such as maximum lengths for names or descriptions, were properly enforced. Exception testing was critical to confirm that invalid inputs threw the appropriate errors, thereby preventing invalid data from entering the system.

Other testing techniques were not employed in this project but are worth noting. Integration testing, for example, focuses on verifying that separate modules work together as expected. System testing validates the application as a whole, typically including UI and database layers, which were not part of this assignment. Performance testing examines scalability and response times under load, while regression testing ensures that new changes do not break previously working features. Each of these techniques has practical implications: performance testing is crucial for high-traffic web applications, and regression testing is essential in long-term software maintenance.

The mindset I adopted while working on this project was one of caution and thoroughness. As a tester, I approached the services as though they were written by someone else, adopting a black-box perspective. This mindset helped me remain objective and prevented me from assuming that my code was correct without validation. For example, I tested scenarios where null values were passed into constructors, even though I believed my code handled these cases. This demonstrates the importance of appreciating the complexity and interrelationships of the code.

I also worked to limit bias in my testing. When developers test their own code, there is often a subconscious bias toward validating what they expect to work. To avoid this, I intentionally attempted to break my own code by designing tests that simulated real-world invalid inputs. For instance, I tested whether an Appointment could be created with a past date, even though I knew the validation logic should prevent it. This helped me identify areas where additional defensive programming was necessary.

Finally, this project underscored the importance of being disciplined in my commitment to quality. Cutting corners in testing might save time initially but creates technical debt, leading to larger problems later in the development cycle. For example, failing to test boundary conditions for string lengths might result in runtime exceptions when unexpected input is encountered in production. As a practitioner, I plan to avoid technical debt by adhering to best practices such as writing unit tests alongside new features, conducting peer reviews, and using continuous integration tools to automatically run tests with every code change.

References

García, B. (2022). \*JUnit 5: Basics and best practices\*. Springer.  
Myers, G. J., Sandler, C., & Badgett, T. (2011). \*The art of software testing\* (3rd ed.). Wiley.  
IEEE. (2017). \*IEEE standard for software and system test documentation (IEEE 829-2017).\*

IEEE Computer Society.