Lasupe Xiong

CS-320

Instructor Angelo Luo

04/18/2025

Project Two: Summary and Reflections Report

In order to satisfy the client's requirement for a mobile application, I implemented and tested three main service classes for Project One: ContactService, TaskService, and AppointmentService. Each of the services, the aim was to test data against stringent constraints during the act of creating, editing, and deleting entities. Such functional needs were exactly matched by my unit testing approach.

For example, phone numbers had to be precisely 10 digits, and the Contact class needed a contact ID of no more than 10 characters. If the phone number is less than ten digits, the program raises an IllegalArgumentException, as my test function testAddContactPhoneTooShort in ContactServiceTest clearly confirmed. Similar to this, the AppointmentServiceTest's testUpdateAppointmentPastDate function made sure that customers couldn't make an appointment in the past, satisfying the need for future dated appointment dates. These examples show how, in order to guarantee alignment, each validation rule in the requirements was supported by a matching unit test.

I created my tests using JUnit in the Apporto Eclipse environment, and I utilized the code coverage percentage that Eclipse has built in to make sure that all branches were covered. Eclipse gave me enough information to verify that my tests covered more than 80% of the code, even though I didn't utilize JaCoCo.

For instance, my ContactServiceTest had boundary/negative route tests like testUpdateContactFirstNameTooLong in addition to happy path methods like testAddContactSuccess. These were essential for verifying how the program behaved in both edge case and real-world situations. At least one test method exercised every logical branch, including those that resulted in exceptions, confirming the thoroughness and efficacy of my unit tests.

My approach to building tests was to validate expected behavior or output for each form of input, including null, invalid, and valid. For instance, I made that TaskServiceTest sound by using:

@Test

void testUpdateTaskNullName() {

assertThrows(IllegalArgumentException.class, () -> {

service.updateTask("12345", null, "New description");

});

}

Strong defensive coding techniques are demonstrated by this test, which verifies that null values are rejected, and exceptions are raised correctly. To guarantee technically good code, I adhered to this structure uniformly across all services.

By utilizing Java collections like HashMap for constant-time access to service objects and minimizing repeated logic, I was also able to optimize my code. Each service class, for instance, featured:

* private final Map<String, Contact> contacts = new HashMap<>();

This made it possible for me to directly reference item IDs and carry out add, update, and delete operations quickly. The tests were also effective since they built up test situations without needlessly repeating code logic by utilizing helper methods like service.addContact(). This method improved test speed and maintainability.

I mostly used boundary value analysis, black-box testing, and unit testing. Individual class methods were isolated for validation with the use of unit testing. For instance, in AppointmentServiceTest, I tested every update function individually to make sure they didn't conflict with one another.

I was able to concentrate on input-output behavior through black box testing, which was crucial for evaluating exception handling. I only verified that incorrect inputs produced the right outputs or exceptions; I didn't need to understand how internal validation was carried out.

Analyzing boundary values was essential for determining the maximum string length. I was able to confirm accurate cutoff enforcement, for instance, by testing a first name with precisely 10 characters and subsequently one with 11 characters.

I did not employ equivalency partitioning, integration testing, or test-driven development (TDD).

* Equivalency partitioning entails classifying inputs as either valid or invalid. I did not technically categorize them, but I did cover both inferentially.
* Verifying the interoperability of components is the goal of integration testing. Project One concentrated on discrete services, so I didn't evaluate how they interacted with one another.
* TDD mandates that tests be written prior to code implementation. Although I occasionally had a similar approach, I did not adhere to the TDD technique exactly.

Unit testing may be automated for regression testing and guarantees method-level dependability in real-world situations. When several services interact, like in microservices or REST APIs, integration testing becomes essential. By describing behavior up front, TDD enhances code quality and can be especially helpful in agile settings. Systems that rely heavily on user input, such as online forms or data entry apps, benefit from boundary value and equivalency class testing.

By purposefully testing faulty inputs and foreseeing potential errors, I was cautious. To make sure null values are never allowed, for instance, I created tests like testAddContactNullId in ContactServiceTest. This type of prudence increases program resiliency and lowers runtime exceptions. I was able to comprehend the interrelationships within the system by seeing how limitations in one class (contact ID length, for example) may impact service behavior (ID lookups, for example).

I refrained from presuming that my code will function in order to reduce prejudice. As an alternative, I built tests to disprove my assumption that it would fail. I became conscious of developer bias,the propensity to have faith in your implementation when I tested my own code. I was reminded to think like an end-user and consider how the system can malfunction by writing tests like testDeleteNonexistentContact. This kind of thinking was essential to creating trustworthy, objective testing.

Cutting shortcuts was never an option when it came to testing discipline. After the majority of tests were successful, it was tempting to omit the edge cases, but I made a commitment to complete coverage. I understood that omitting a single length validation or null check may result in issues later, particularly when additional features are implemented. Maintaining high-quality tests, recording test cases, and modifying when necessary are all part of avoiding technical debt. For example, in order to keep the code DRY (Don't Repeat Yourself), I rebuilt redundant test logic in helper methods.

Through this project, I developed a great deal of experience in developing and implementing thorough unit testing to verify back-end service logic. By following instructions, maintaining coverage above 80%, and cultivating a purposeful and methodical mindset, I ensured the efficacy and caliber of my code. The skills I've learned here will serve as the foundation for my future work as I delve deeper into increasingly complex and interconnected software systems.

References  
JUnit 5 User Guide. (n.d.). <https://junit.org/junit5/docs/current/user-guide/>  
Oracle. (n.d.). Java Platform, Standard Edition. <https://docs.oracle.com/javase/8/docs/>