Abbigale Junker

Professor Kuang-Jung Huang

CS-320

18 April, 2025

Software Testing Summary and Reflection Report

In this project I developed and tested six classes for a mobile application service. The service classes include ContactService, TaskService, and AppointmentService and their object classes include Contact, Task, and Appointment. Each service handled object creation, updates, and deletions, and the object classes carried the constraints of required fields, maximum character lengths, and non-editable IDs. The testing classes for all six classes are used to verify that each object and service meet specifications. These tests verify valid behavior, invalid input handling, and exception throwing occur when expected to ensure the application logic is reliable.

My approach to writing unit tests was directly aligned with the software requirements. I focused on validating the required field constraints like maximum character limits, non-null and null values, and behavior when adding, updating, and deleting objects through the service. I wrote both positive and negative tests to verify the expected behavior. Each class was tested for its specific validation rules. For example, the Task object has a name field limited to 20 characters and a description limited to 50. I wrote tests like testNameTooLong() and testSetDescriptionNull() to confirm that the constructor and setter methods threw IllegalArgumentException when the constraints were not followed. I took a similar approach in the other object classes like ContactTest and testSetNameNull() or AppointmentTest’s testSetNullDate().

I know the tests are well written because the coverage of all classes averaged over 80%. The TaskService project reached 86.5%, the ContactService project reached 80%, and the AppointmentService project reached 84.9% coverage. These numbers confirm that my tests reached nearly all logic and exception paths across the application.

Writing JUnit tests is a work of being intentional with what I was testing. To ensure my code was technically sound, I utilized assertEquals and assertThrows to verify that each object behaved as expected when given valid and invalid input. I wrote tests to target the rules in the project requirements. For example, I tested that null IDs were rejected with:

assertThrows(IllegalArgumentException.class, () ->{

new Task(null, "Name", "Description");

});

I also used update tests to confirm that the service modified the test fields, like this one:

service.updateName("123", "New Name");

assertEquals("New Name", task.getName());

In addition to writing JUnit tests, I ensured my code was efficient by utilizing @BeforeEach to set up reusable service and object instances. This helped me keep from rewriting redundant code, which contributes to the organization of the project.

The software testing techniques I used in this project include boundary testing, which tests values at the limit for example, I tested a 10-character ID just over the limit to ensure an error was thrown. Another technique I used was using clear naming conventions for the tests to make it easier to describe what each test was testing, like testSetDescriptionNull which tests if an error is thrown when a description is null. I didn’t use techniques like system testing or user acceptance testing in this project, since neither was necessary. System testing wasn’t possible because the application doesn’t include a complete system or user interface, it focused on backend functionality. User acceptance testing is another technique I did not use since the project did not involve real end users for feedback. My testing focused on verifying that the code met the specified requirements for each feature and handled input as expected.

While working on this project I have adopted a detail-oriented mindset. There were presented paths in the code that needed to be tested, and it was important for me to get the coverage needed without jeopardizing the quality of the code. I was incredibly cautious while writing my tests because I wrote the code and should not assume it works perfectly. I needed to verify that the code worked by writing tests outside of the requirements, like invalid input and boundary testing. It's important to appreciate the complexity of the code because each piece has the potential to work together while still being modular. Like the TaskService utilizes a Task object and each of those have their own classes and test classes.

Bias can be dangerous in testing your own code. I tried to eliminate bias by writing tests that expect failure, not just success. I did not assume a field would always be valid and tested what would happen if it were null or too long. One example involves the number in Contact, it needs exactly 10 characters, and I tested the edge cases of it being too long and too short. As a developer it is easy to trust your own work, but it can lead to missed edge cases. I had to be intentional with my tests while assuming the worst to keep me on track.

Being disciplined with testing is essential for writing reliable code. Cutting corners might save time in the short-term, but it creates bugs that will cost a lot more to fix in the future. In this project, I tested all required constraints and also tested edge cases like testAddDuplicateTaskID(). I used coverage tools to confirm I wasn’t missing logic. Technical debt can be prevented with adequate testing; this is something I have learned and will continue to do by testing input boundaries, reviewing coverage, and fixing logic early.