Project Two by Aqbah Aamir

I ensured that I conformed strictly to the software requirements for both services. We were required to provide CRUD operations, enforcing data integrity, and validation of data input so invalid input would be rejected. I will share some examples which display how I followed these requirements.

In the Task Service, I added several tests one of which was “taskAddTest”. The test made sure that new tasks could only be added using unique IDs, as per the requirements.

A computer screen shot of text

Description automatically generated

Due to this test, I was able to make sure that there were no duplicate IDs and was able to fulfill the criteria set out for us.

Similarly, I created a test in Contact Service which validated the contact creation input and ensured that no invalid data was input for the contact. These requirements included no null input for the first and last name of the contact, and that a phone number must contain exactly 10 digits.



As for the quality of Junit tests, I can back it up by the fact of how thoroughly they are testing the code. The tests I created handled both happy paths as well as edge cases for both services and this made sure that the required validation and functionality was achieved by my code. I also made sure that exception handling was robust for the code.

As an example, the “taskDelTest” in the Task Service makes sure that the user is not attempting to delete a task ID that does not exist, in which case it throws an exception, and when the ID is found, the task is deleted in its entirety.

A computer screen shot of text

AI-generated content may be incorrect.

The fact that this test deals with the standard as well as scenarios where an exception might occur, shows the extent of the coverage of the tests.

A black screen with colorful text

AI-generated content may be incorrect.

For the Contact Service, I set out tests to verify that if any phone number was to be updated, it had to be of exactly 10 digits, and if invalid data was added then an exception would be triggered. These tests for both services ensured that all the different scenarios were covered, and exceptions were triggered in cases of invalid input.

I achieved the technical quality of my code by performing extensive validation logic and encapsulation of fields. To maintain data integrity, I implemented constraints in both constructors and setters.

I used the constructor in the Task class to validate data for all the fields:

A screen shot of a computer program

Description automatically generated

This test ensures that only valid data is passed when creating an object.

A computer screen shot of a black screen

AI-generated content may be incorrect.

Similarly, I used the constructor in the Contact class to make sure that every contact had a unique ID that followed the requirements, and even the setters for every field checked the data to meet all requirements otherwise, the data would not be updated.

During the development of each service, there was a particular focus on optimal utilization of resources. Adding, deleting, and updating tasks and contacts through a HashMap took O(1) time, so storing them as such achieved perfect time complexity.

In the Task Service, I used HashMap in the “taskAdd” method:

A screen shot of a computer program

AI-generated content may be incorrect.

While checking for existing task IDs when adding a new task, the use of HashMap allowed me to allocate the resources efficiently.

For the milestones I submitted in Modules Three, Four, and Five, I made use of different software testing methods to make sure that the Task Service, Contact Service, and Appointment Service were functioning as required. Using these tests ensured that the features that I implemented were functional and reliable and stopped invalid data from being entered and processed by the code.

The testing method which was repeatedly used across the assignments was Unit Testing. I coded Junit tests to validate if all the components of my code were performing correctly. Different tests were used to validate functionalities such as data validation, exception handling, and creating objects. In the Task Service, I used the following test to validate that every task was created with a unique task ID:

A computer screen shot of text

AI-generated content may be incorrect.

Similarly, I developed the following test for Appointment Service to make sure that only dates from the future were used when adding an appointment:

A screenshot of a computer code

AI-generated content may be incorrect.

Another testing technique that I used in the milestones was Boundary testing. I used this technique to meet the constraints set for us including the length of the strings and components where null values were not to be processed. This was especially useful for Contact Service and Appointment Service because we were required to validate the data for the right dates, descriptions, and lengths of IDs. As an example, I developed this test for the Appointment Service to validate that the description entered would not exceed the 50 characters limit:

A screenshot of a computer

AI-generated content may be incorrect.

I also implemented exception testing which ensured that any data input which did not meet the requirements threw an exception so the data which entered the system would be completely in line with the constraints. For this purpose, I developed the following test for the Appointment Service which throws an exception if a duplicate appointment ID is used:

A screenshot of a computer code

AI-generated content may be incorrect.

As for the testing techniques that were not used in the milestones, I can think of the following ones:

* System Testing: System Testing is implemented to make sure that the whole application is working as intended. This approach is different from the unit tests that I used in the milestones because, instead of individual parts, system testing focuses on the application.
* Integration Testing: Integration testing is used to verify the seamless functioning of distinct parts of an application altogether. This ensures that appropriate data is exchanged between different modules and aims to identify problems that may present in situations when various parts of the application interact with each other. Such issues are not identifiable with unit testing alone.
* Performance Testing: Performance testing uses metrics like stability, response time, and scalability to assess how well software would run under different levels of loads. This aids in discovering areas where software may bottleneck and what areas would need further optimization. When done correctly, this ensures a seamless user experience when handling operations with high traffic.

Practical Uses:

Unit testing is crucial for identifying errors early on and making sure that each component functions as intended. Utilized in practically every software project.

Boundary Testing: By making sure that restrictions (such as string length limits) are appropriately applied, boundary testing helps to avoid mistakes. Beneficial for apps' form validation.

System Testing: System error handling is ensured by exception testing. Essential for enterprise apps' input validation.

Integration Testing: Applications involving the interaction of various services must undergo integration testing.

System Testing: System testing is essential for end-to-end validation since it guarantees that the complete system functions as intended.

Performance Testing: Performance testing is crucial for apps that are supposed to manage heavy user traffic or volumes of data.

As a software tester, I approached this assignment with caution and attention to detail. It was essential to approach testing with attention to detail because the mobile application comprised three fundamental functionalities: contact, task, and appointment management. I made sure to thoroughly review the application's edge cases, boundary values, and possible weak points. For instance, I purposefully tried to establish contacts with empty values or very long names when testing the contact service to make sure the system applied constraints effectively. To make sure the system handled invalid inputs gracefully, I also tested instances in the task service where due dates were set in the past.

Effective testing requires an understanding of the code's interrelationships and complexity. Because the application's three features were interdependent, a flaw in one module may influence the others. For example, a contact may be mentioned in an appointment, and a contact creation error may result in unexpected action from the appointment service. Because of this interdependence, an organized testing strategy was necessary to make sure that modifications made in one area wouldn't have unforeseen consequences. To confirm that the system operated as intended when several components interacted, I used integration testing in addition to unit testing.

I approached testing objectively, concentrating only on the expected behaviors specified in the criteria, to reduce prejudice in my evaluation of the code. I relied on written requirements rather than speculating about how the system "should" operate based on my past development experience. I also occasionally asked myself whether there were any scenarios I had missed by looking at my test cases and outcomes from an external viewpoint. When developers test their code, bias is a typical problem since they could unconsciously refrain from testing edge scenarios they believe would work. For instance, I thought the task ID generation logic was sound when I first developed the task service.

In software engineering, discipline and dedication to quality are crucial. Skimping on testing might result in hidden flaws that raise maintenance costs and negatively impact user experience. For instance, the appointment scheduling service's dependability may be compromised if concurrency problems are not tested, which could result in duplicate bookings. I want to continuously follow best practices, such as creating thorough test cases, performing frequent code reviews, and utilizing automated testing tools to guarantee ongoing validation of code quality, to prevent technical debt. I can reduce long-term risks and provide software solutions that are more reliable by integrating quality into every stage of development.