Hiep Ha

CS-320

Professor Kalysa Wilson

Dec. 14, 2024

7-2 Project Two Submission

**Summary and Reflections Report**

In this project, I implemented comprehensive unit testing for three critical backend services: **Contact**, **Task**, and **Appointment**. I ensured that the services met the specified requirements by testing key features such as valid and invalid input handling, boundary cases, and exception handling. Each service was tested using **JUnit** testing frameworks to ensure they functioned correctly under various conditions.

For the **Contact** service, I focused on testing the addition, deletion, and retrieval of contacts, ensuring that invalid input scenarios (e.g., invalid contact IDs, invalid name or phone number lengths) were properly handled. The tests for the **Task** service focused on verifying task creation, duplication handling, and deletion. Similarly, the **AppointmentService** tests ensured that appointments were only added with valid dates (future dates only) and unique IDs.

I aligned my testing approach with the requirements by validating that:

* Each service (Contact, Task, Appointment) handled valid and invalid input correctly.
* All input constraints, such as unique IDs, valid names, and valid dates, were enforced.
* Deletion and retrieval operations worked correctly even when attempting to delete non-existent entities.

For **Contact**:

*@Test*

*public void testAddContact() {*

*Contact contact = new Contact("12345", "John Doe", "123-456-7890");*

*service.addContact(contact);*

*assertEquals(contact, service.getContact("12345"));*

*}*

For **Task**:

*@Test*

*public void testAddTask() {*

*Task task = new Task("12345", "Task Name", "Task Description");*

*service.addTask(task);*

*assertEquals(task, service.getTask("12345"));*

*}*

For **Appointment**:

*@Test*

*public void testAddAppointment() {*

*Date futureDate = new Date(System.currentTimeMillis() + 86400000); // 1 day in the future*

*Appointment appointment = new Appointment("12345", futureDate, "Meeting");*

*service.addAppointment(appointment);*

*assertEquals(appointment, service.getAppointment("12345"));*

*}*

The quality of the tests was determined by achieving high code coverage (using tools like **JaCoCo**) and by ensuring thorough edge case validation. For example, I tested cases where the appointment date was in the past, ensuring that an exception was thrown:

*@Test*

*public void testInvalidAppointmentDate() {*

*Date pastDate = new Date(System.currentTimeMillis() - 86400000); // 1 day in the past*

*assertThrows(IllegalArgumentException.class, () -> new Appointment("12345", pastDate, "Meeting"));*

*}*

The tests covered all the necessary functionality, such as creating and retrieving appointments, tasks, and contacts, ensuring they met the business rules.

The software testing techniques I employed included **unit testing**, **boundary testing**, and **exception testing**.

* **Unit Testing**: I wrote unit tests for individual methods within the services. Each unit test verified that the methods worked as intended, such as adding and deleting contacts, tasks, and appointments.
* **Boundary Testing**: I tested the boundaries for input lengths, such as name and description lengths in the **Contact** and **Task** services, and appointment date validation in the **AppointmentService**.
* **Exception Testing**: I ensured that invalid inputs (e.g., a past date for an appointment or a duplicate ID for a task) triggered the correct exceptions. For instance:

*assertThrows(IllegalArgumentException.class, () -> service.addTask(new Task("12345", "Task Name", "Task Description")));*

While I did not use techniques like **integration testing** or **system testing**, these could have been beneficial for verifying how the **Contact**, **Task**, and **Appointment** services interact with each other and with a database. **Integration testing** ensures that data flows between the services correctly, while **system testing** could validate the entire mobile application’s functionality.

The testing techniques I used are applicable in various types of software projects. **Unit testing** helps ensure that each component is correct, while **boundary testing** guarantees that edge cases (e.g., maximum length for task descriptions) are handled correctly. **Exception testing** is essential to catch errors and ensure robust error handling. These techniques are particularly useful in backend services where validation and data integrity are key.

For example, in a larger-scale project with a complex database or user interface, **integration testing** would help verify that data flows correctly across the system. **System testing** would allow us to check that the entire application, including all services, works as expected in real-world scenarios.

In completing this project, I maintained a mindset of caution and thoroughness. I understood that small errors in input validation could result in significant issues in production. For instance, ensuring that appointments could only be scheduled in the future was crucial, as any appointment with a past date would have caused confusion. As a result, I wrote extensive tests for invalid data inputs.

To limit bias, I actively avoided assumptions that my code would work as expected. Instead, I wrote tests that were independent and based on input-driven behavior. For example, when testing the **TaskService**, I did not assume that the code would handle duplicate tasks correctly without first verifying it with explicit tests like:

*assertThrows(IllegalArgumentException.class, () -> service.addTask(new Task("12345", "Task Name", "Task Description")));*

As a software engineer, maintaining high-quality code and testing is vital. Cutting corners can lead to **technical debt**, where poor-quality code accumulates over time, resulting in difficult-to-fix bugs and a less maintainable codebase. By ensuring thorough test coverage and paying attention to edge cases, I avoided introducing such debt.

For example, I ensured that all edge cases were covered, such as testing for description length constraints in the **Task** service:

*assertThrows(IllegalArgumentException.class, () -> new Task("12345", "Task Name", "This description is far too long to be valid in this context."));*

In the future, I plan to continue adopting disciplined practices by revisiting my code to refactor and optimize it regularly, and to keep test coverage high to ensure a stable, maintainable codebase.

**References**

Hobbs, B. (2022). *Software testing techniques for professionals*  
JaCoCo. (2023). *JaCoCo code coverage tool*. <https://www.jacoco.org>  
Smith, J. (2023, June 25). *Understanding unit testing in Java*