Miguel Hilario

CS-320

Module 7

Project Two: Reflection

**Summary**

Unit Testing Approach for Each Feature

For the Contact service, I wrote tests to cover all key functions, like creating, updating, and deleting contacts. The tests made sure that the contact ID was unique, the first and last names were not empty and had the right number of characters, the phone number was exactly 10 digits, and the address was not empty and had the right number of characters.

Example:

@Test

void createContactWithInvalidPhone() {

assertThrows(ValidationException.class, () -> {

new Contact("12345", "First", "Last", "123", "1234 Loblolly Lane");

});

}

For the Task service, I did the same. The tests checked that the task ID was unique, the name was not empty and had a 20-character limit, and the description was not empty and had a 50-character limit.

Example:

@Test

void createTaskWithLongDescription() {

assertThrows(ValidationException.class, () -> {

new Task("12345", "Test Task", "this description is way too long to be valid as it exceeds fifty characters.");

});

}

For the appointment tests, the tests made sure the appointment ID was unique, the date was valid (not empty and not in the past), and the description had a 50-character limit.

Example:

@Test

void createAppointmentWithPastDate() {

Date pastDate = new Date(System.currentTimeMillis() - 100000);

assertThrows(ValidationException.class, () -> {

new Appointment("12345", pastDate, "Test appointment description.");

});

}

Alignment with Software Requirements

My approach matched the requirements well. For example, in the `Contact` service, I made sure the phone number had to be exactly 10 digits. This was tested in the `createContactWithInvalidPhone` test, which checked that a phone number with fewer than 10 digits would throw a `ValidationException`. Similarly, for the `Appointment` service, the `createAppointmentWithPastDate` test made sure the appointment date was not set in the past, as required.

Quality of JUnit Tests

The JUnit tests were good because they covered over 80% of the code. This high coverage means the tests checked most parts of the code, catching potential problems. For example, tests for creating, updating, and deleting tasks in `TaskServiceTest` covered all actions on tasks, ensuring thorough checks.

Experience Writing JUnit Tests

Writing the JUnit tests was a learning experience. It showed me the importance of checking edge cases and thorough validation. For example, ensuring a contact ID was unique needed checking against an in-memory database, as shown in the `addSuccess` test in `ContactServiceTest`.

To make sure the code was sound, I used validation methods to check input constraints, as seen in the `Contact` constructor:

Validation.validateLength(contactId, "id", 1, 10);

Validation.validateNotBlank(firstName, "firstName");

Validation.validateLength(firstName, "firstName", 1, 10);

To keep the code efficient, I used simple validation methods. For instance, checking if a date was in the past:

if (appointmentDate == null || appointmentDate.before(new Date())) {

throw new ValidationException("appointmentDate cannot be in the past or null");

}

**Reflection**

Testing Techniques Employed

The main techniques I used were unit testing and validation testing. Unit testing checked individual methods and classes to make sure they worked right. Validation testing checked if the input data met certain criteria before being processed.

Other techniques not used included integration testing, which tests how different parts work together, and system testing, which tests the whole system. These are useful in larger projects with many parts.

Integration testing makes sure different parts of the system work together correctly, while system testing checks if the whole system meets the requirements.

Mindset

Being careful was key. Understanding the complexity and how different parts of the code interact helped me write better tests. For example, knowing how contact updates could affect the database helped me write more thorough tests.

To avoid bias, I reviewed the code with the idea that mistakes could be anywhere, even in simple logic. This helped me not assume the code was correct.

Bias can be a problem when developers test their code because they might avoid testing known weak spots. To avoid this, I wrote tests for all edge cases and reviewed the code critically.

Commitment to Quality

Staying committed to quality is important in software engineering. Cutting corners can lead to technical debt, which makes maintenance harder and reduces reliability. To avoid this, I plan to follow best practices, like writing thorough tests and doing regular code reviews.

For example, making sure all input validations are tested prevents future bugs and reduces debugging time. By consistently following these practices, I aim to deliver high-quality, reliable software.

Conclusion

In this project, I used different testing strategies to meet the requirements and ensure good code quality. The JUnit tests were effective in finding errors and checking the functionality of the `Contact`, `Task`, and `Appointment` services. By being thorough and careful, I made sure the code was strong and reliable, which is crucial for professional software development.