Project 2

Summary

1. Contact Management Feature:

Approach: Used JUnit to create tests for adding, updating, and deleting contacts. Tests verified correct handling of invalid input and maintained data integrity.

Alignment to Requirements: It adhered to requirements by ensuring each contact's unique constraints (unique ID, valid phone number, etc.). The test `testAddContact` confirms the addition of a new contact, aligning with the requirement to manage contacts.

2. Task Management Feature:

-Approach: Created JUnit tests for task creation, updating, and deletion. Tests included validation of task properties such as ID length, name length, and description length.

- Alignment to Requirements: This approach ensured that tasks met specified constraints, directly supporting requirements. For example, `testAddTask` ensures that tasks cannot exceed specified character limits.

3. Appointment Management Feature:

- Approach: Developed JUnit tests for adding and deleting appointments. Tests included date validation to ensure appointments are set in the future.

- Alignment to Requirements: The tests made sure the appointment management system operated as needed. `testAddAppointment` validates that the appointment is correctly added and retrieved.

Ensuring Technical Soundness

The technical soundness was ensured by:

- Strict input validation in constructors, as shown in the `Contact` class:

```java

if (contactId == null || contactId.length() > 10) {

throw new IllegalArgumentException("Invalid contact ID");

}

```

Ensuring Code Efficiency

Efficiency was ensured by optimizing data structures and methods. For example, using a `HashMap` in `ContactService` for O(1) average time complexity for add, update, and delete operations:

```java

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

```

Reflection

Testing Techniques Employed

- Unit Testing: Focused on validating individual units of code (classes, methods, etc.). Ensured each class functioned correctly.

Testing Techniques Not Used

- Integration Testing: Tests the interaction between different modules. It ensures that combined parts of the application work together.

- System Testing: Validates the complete and integrated software. Ensures the application meets the specified requirements.

Practical Uses and Implications

- Unit Testing: Ideal for early development stages. Helps identify bugs at an early stage, reducing the cost of fixing them later in development.

- Integration Testing: Crucial when different modules are developed by separate teams. Ensures smooth communication between modules.

- System Testing: Essential before product release. Validates the software’s compliance with requirements set by team.

Mindset Adopted

- Cautious Mindset: Employed caution to ensure all possible scenarios were tested and nothing was missed.

- Example: Validating the date in `Appointment` to ensure it is set in the future.

Commitment to Quality

Quality was essential. Cutting corners can lead to technical debt, which increases maintenance costs in the future.

- Avoiding Technical Debt: Regularly refactoring code and writing comprehensive tests help avoid technical debt.