**Summary:**

**Unit Testing Approach for Each Feature:**

**ContactService**:

Unit tests were written for the ContactService class.

The tests covered adding, deleting, updating, and retrieving contacts using the service.

Tests covered scenarios such as adding a contact, adding a contact with a duplicate ID (expecting an exception), deleting a contact, deleting a nonexistent contact (expecting an exception), updating a contact's fields, updating a nonexistent contact (expecting an exception), and retrieving a contact by ID.

The approach was aligned with the software requirements of the ContactService class by testing its core functionality and exceptional cases.

**Example of technical soundness:** In the testAddContact method, a contact was added, and the service's getAllContacts method was used to verify that the contact was added successfully.

**Example of efficiency**: In the testDeleteContact method, after adding and deleting a contact, the getAllContacts method was used to ensure that the deleted contact was removed.

@Test

public void testDeleteContact() throws ContactNotFoundException {

Contact contact = new Contact("001", "Mahtab", "Beplab", "1234567890", "123 Main St");

try {

contactService.addContact(contact);

contactService.deleteContact("001");

List<Contact> allContacts = contactService.getAllContacts();

Assertions.assertTrue(allContacts.isEmpty());

} catch (DuplicateContactException e) {

Assertions.fail("Should not throw DuplicateContactException");

}

}

**TaskService:**

Unit tests were written for the TaskService class.

Tests included adding, deleting, updating, and retrieving tasks using the service.

Similar to the ContactService tests, these tests covered common scenarios and exceptional cases.

**Example of technical soundness:** In the testUpdateTask method, a task was added, then updated using the service's updateTask method, and finally, the task's fields were checked to ensure they were updated correctly.

@Test

@DisplayName("Test Update Task Fields by ID")

public void testUpdateTask() {

Task task1 = new Task("task1", "Task 1", "Description 1");

taskService.addTask(task1);

taskService.updateTask("task1", "Updated Task Name", "Updated Description");

Task updatedTask = taskService.getTaskById("task1");

assertNotNull(updatedTask);

assertEquals("Updated Task Name", updatedTask.getName());

assertEquals("Updated Description", updatedTask.getDescription());

}

Example of efficiency: In the testDeleteTask method, a task was added, deleted using the service's deleteTask method, and then its existence was checked using the getTaskById method.

**AppointmentService:**

Unit tests were written for the AppointmentService class.

Similar to the other services, tests covered adding and deleting appointments.

**Example of technical soundness:** In the testDeleteAppointment method, an appointment was added, deleted, and then an attempt to retrieve it using the getAppointmentById method verified that the appointment was indeed deleted.

@Test

public void testDeleteAppointment() {

Appointment appointment1 = new Appointment("app2", new Date(), "Description 2");

appointmentService.addAppointment(appointment1);

assertDoesNotThrow(() -> appointmentService.deleteAppointment("app2"));

assertNull(appointmentService.getAppointmentById("app2"));

}

Example of efficiency: Similar to the other tests, after adding and deleting an appointment, the getAppointmentById method was used to check if the appointment was deleted.

**Overall Quality of JUnit Tests:**

The JUnit tests are comprehensive and cover both normal and exceptional scenarios.

The tests use various assertion methods to ensure the correctness of the code under test.

Each test method is named descriptively, aiding in understanding their purpose.

The tests use setup and teardown methods (@BeforeEach and @AfterEach) to provide a clean state for each test, reducing interference between tests.

@BeforeEach

public void setUp() {

taskService = new TaskService();

}

The tests also make use of JUnit's @DisplayName annotation to improve the readability of the test results.

@Test

@DisplayName("Test Update Task Fields by ID")

public void testUpdateTask() {

Task task1 = new Task("task1", "Task 1", "Description 1");

taskService.addTask(task1);

// Update task1's name and description

taskService.updateTask("task1", "Updated Task Name", "Updated Description");

Task updatedTask = taskService.getTaskById("task1");

assertNotNull(updatedTask);

assertEquals("Updated Task Name", updatedTask.getName());

assertEquals("Updated Description", updatedTask.getDescription());

}

**Reflection:**

**Testing Techniques:**

**Employed Techniques:**

Black Box Testing: This type of testing ignores internal implementation details in favor of focusing on exterior behavior. An illustration would be any of the tests that add, remove, update, or retrieve items from the services.

Tests contain scenarios that are on the edge of what is acceptable input, according to boundary value analysis. As an illustration, consider the checks that look for names and descriptions that are too long.

Testing for exceptions verifies that the code raises exceptions as necessary. The tests that look for null values in constructor parameters, for instance.

**Equivalence Partitioning**: Tests are organized according to equivalence classes, grouping potential input values into sets that behave similarly. Examples are tests that include appointments with both correct and incorrect dates.

White box testing is not used since it does not rely on understanding the underlying organization of the code.

**Regression testing:** The provided code did not explicitly demonstrate any tests to make sure that new code modifications did not adversely affect current functionality.

**Applications and Consequences:**

Black Box Testing: This method is helpful for making sure that external behavior and requirements are in sync, making it appropriate for functional testing in many kinds of software projects.

Boundary Value Analysis: Useful in preventing edge situations and potential defects, especially in programs where input values must adhere to stringent restrictions, such submission of forms.

Testing for exceptions is crucial to ensure that code responds gracefully to unusual circumstances, which is essential for sustaining reliable software.

Equivalence Partitioning is useful for evaluating various types of input behavior, particularly in systems with intricate input needs, such financial calculations or user interactions.

**Mindset:**

Adopted Mindset: Caution: A wide range of scenarios, including both valid and invalid inputs as well as uncommon cases, were intended to be covered by the test cases. This strategy makes that the code functions as planned and helps find any issues.

Understanding Complexity: The complexity of the code was taken into consideration, notably when managing exceptions and input validation. For instance, correct dates must be verified and non-null values must be present in setters if runtime errors are to be avoided in the Appointment class.

Limiting Bias: Bias was reduced by rigorously following each class's and method's requirements and specifications.

Instead of favoring particular paths, test cases were created to assure complete coverage.

If the developer were in charge of testing their own code, bias might be an issue because they might have a propensity to ignore particular edge situations or extraordinary circumstances.

Commitment to Quality: Skimping on code creation or testing might result in subpar software, unanticipated defects, and expensive maintenance in the future.

For instance, robust input validation in the Contact class ensures that erroneous data doesn't jeopardize the dependability of the application.

For long-term success, avoiding technical debt is essential since it lessens the necessity of having to rework or rebuild portions of code in the future.