Project 2

Summary:

Unit Testing Approach for Each Feature:

ContactService:

The ContactService class's unit tests were created.

The tests covered using the service to add, delete, update, and retrieve contacts.

Tests included scenarios like adding contacts, adding contacts with duplicate IDs (expecting an exception), deleting contacts, updating contacts' fields, updating contacts that didn't exist (expecting an exception), and retrieving contacts by ID.

By evaluating the ContactService class's core functionality and unusual cases, the strategy was in line with the software requirements for that class.

Technically sound example: The getAllContacts method of the service was used to confirm that the contact was added successfully after it was added in the testAddContact method.

@Test

public void testAddContact() {

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

try {

contactService.addContact(contact);

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

Assertions.assertEquals(1, allContacts.size());

Assertions.assertEquals(contact, allContacts.get(0));

} catch (DuplicateContactException e) {

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

}

}

Efficiency example: In the testDeleteContact function, the deleted contact was removed by using the getAllContacts method after a contact had been added and deleted.

@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:

The TaskService class's unit tests were created.

Tasks using the service were added, deleted, updated, and retrieved during tests.

These tests included typical scenarios as well as exceptional cases, just like the ContactService tests.

The testUpdateTask method added a task, updated it using the service's updateTask method, and then examined the task's fields to make sure the update was successful. This is an example of excellent technical design.

@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());

}

A task was added, deleted using the service's deleteTask function, and then its existence was verified using the getTaskById method in the testDeleteTask method.

AppointmentService:

AppointmentService class unit tests were created.

The testing included both adding and canceling appointments, much like the other services.

In the testDeleteAppointment method, an appointment was added, removed, and then an attempt was made to retrieve it using the getAppointmentById method, which confirmed that the appointment had in fact been deleted. This is an example of sound technical practice.

@Test

public void testDeleteAppointment() {

// Test deleting an existing appointment

Appointment appointment1 = new Appointment("app1", new Date(), "Description 1");

appointmentService.addAppointment(appointment1);

assertDoesNotThrow(() -> appointmentService.deleteAppointment("app1"));

assertNull(appointmentService.getAppointmentById("app1"));

}

Efficiency example: In line with the other tests, the getAppointmentById method was used to determine whether an appointment had been deleted after being added and deleted.

Overall Quality of JUnit Tests:

Both typical and extraordinary circumstances are covered by the JUnit tests, which are exhaustive.

To guarantee that the code being tested is valid, the tests employ a variety of assertion techniques.

Each test method has a descriptive name that makes it easier to comprehend what it is meant to do.

The tests reduce interference between tests by providing a clean state for each test using setup and teardown methods (@BeforeEach and @AfterEach).

@BeforeEach

public void setUp() {

appointmentService = new AppointmentService();

}

Additionally, the tests make advantage of JUnit's @DisplayName annotation to make the test results easier to read.

@Test

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

public void testUpdateNonexistentTask() {

// Attempt to update a task that does not exist should throw an exception

assertThrows(IllegalArgumentException.class, () -> taskService.updateTask("nonexistentTaskId", "Updated Name", "Updated Description"));

}

Reflection:

Testing Techniques:

Employed Techniques:

Black Box Testing: Tests are focused on external behavior without concern for internal implementation details. Example: The various tests that add, delete, update, and retrieve items from the services.

Boundary Value Analysis: Tests include scenarios that are at the boundaries of valid input. Example: The tests that check for invalid lengths of names and descriptions.

Exception Testing: Tests validate that the code correctly throws exceptions when appropriate. Example: The tests that check for null values in constructor parameters.

Equivalence Partitioning: Tests are structured based on equivalence classes, dividing possible input values into subsets with similar behavior. Example: The tests that add appointments with valid and invalid dates.

Not Used Techniques:

White Box Testing: No tests are based on knowledge of the internal code structure.

Regression Testing: Tests to ensure that new code changes do not negatively impact existing functionality were not explicitly shown in the provided code.

Practical Uses and Implications:

Black Box Testing: Useful for ensuring the external behavior aligns with requirements, making it suitable for functional testing in various types of software projects.

Boundary Value Analysis: Valuable in preventing edge cases and potential bugs, particularly in applications where input values have strict limits, such as form submissions.

Exception Testing: Essential for ensuring that code handles exceptional scenarios gracefully, which is critical for maintaining robust software.

Equivalence Partitioning: Effective in validating different classes of input behavior, especially in systems with complex input requirements, such as financial calculations or user interactions.

Mindset:

Adopted Mindset:

Caution: The test cases were designed to cover a wide range of scenarios, including both valid and invalid inputs, as well as exceptional cases. This approach helps uncover potential bugs and ensures the code behaves as expected.

Appreciating Complexity: The complexity of code was considered, especially in input validation and handling exceptions. For example, in the Appointment class, checking for valid dates and ensuring non-null values in setters is crucial to prevent runtime errors.

Limiting Bias:

Bias was minimized by adhering strictly to the requirements and specifications of each class and method.

Test cases were designed to ensure thorough coverage, rather than favoring certain paths.

Bias could be a concern if the developer were responsible for testing their own code, as there might be a tendency to overlook certain edge cases or exceptional scenarios.

Commitment to Quality:

Cutting corners in writing or testing code can lead to poor software quality, unexpected bugs, and costly maintenance later on.

For example, in the Contact class, thorough input validation ensures that invalid data doesn't compromise the application's stability.

Avoiding technical debt is crucial for long-term success, as it reduces the likelihood of having to refactor or rewrite sections of code in the future.