CS-320 – Project Two

Shawn Henly

06/20/24

**Contact Feature:** For the Contact feature, my unit testing approach focused on validating the constraints and properties of the Contact class. Each test case was designed to ensure that invalid data, such as null values or data exceeding the defined length, would trigger appropriate exceptions. For instance, in ContactTest.java, the tests testContactIDConstraints and testFirstNameConstraints are designed to check these constraints:

@Test

public void testContactIDConstraints() {

assertThrows(IllegalArgumentException.class, () -> new Contact(null, "John", "Doe", "1234567890", "123 Main St"));

assertThrows(IllegalArgumentException.class, () -> new Contact("12345678901", "John", "Doe", "1234567890", "123 Main St"));

}

**ContactService Feature:** For the ContactService feature, my approach was to validate the core CRUD (Create, Read, Update, Delete) operations. The tests ensured that contacts were correctly added, updated, and removed from the service, while also checking for edge cases such as duplicate IDs and non-existent IDs. An example from ContactServiceTest.java is the testAddContact method:

@Test

public void testAddContact() {

Contact contact = new Contact("1234567890", "John", "Doe", "1234567890", "123 Main St");

contactService.addContact(contact);

assertEquals(contact, contactService.getContact("1234567890"));

}

**AppointmentService Feature:** For the AppointmentService feature, the focus was on ensuring that appointments were added and deleted correctly, with checks for unique IDs and valid dates. The test testAddAppointment in AppointmentServiceTest.java checks these requirements:

@Test

public void testAddAppointment() {

Date futureDate = new Date(System.currentTimeMillis() + 86400000);

Appointment appointment = new Appointment("1234567890", futureDate, "Description");

assertTrue(appointmentService.addAppointment(appointment));

assertNotNull(appointmentService.getAppointment("1234567890"));

}

**Alignment with Requirements:** My approach was tightly aligned with the software requirements by ensuring that all functional requirements were validated through tests. The specific constraints and behaviors defined in the requirements were directly reflected in the test cases, ensuring comprehensive coverage.

**JUnit Test Quality:** The overall quality of the JUnit tests is demonstrated by the high coverage percentage. The tests effectively covered various scenarios, ensuring that both typical and edge cases were considered. This extensive coverage is indicative of effective tests that minimize the risk of undetected bugs.

**Technical Soundness:** To ensure technical soundness, each test case was designed to cover specific functionality and constraints. For example, the testInvalidPhoneConstraints method ensures that phone numbers must be exactly 10 digits:

@Test

public void testPhoneConstraints() {

assertThrows(IllegalArgumentException.class, () -> new Contact("1234567890", "John", "Doe", "123456789", "123 Main St"));

assertThrows(IllegalArgumentException.class, () -> new Contact("1234567890", "John", "Doe", "12345678901", "123 Main St"));

}

**Efficiency:** Efficiency was achieved by eliminating redundant tests and concentrating on crucial paths and edge cases. This strategy reduced the total number of tests while ensuring maximum coverage and effectiveness. For example, each constraint was tested with only the essential number of cases needed to validate the behavior.

### Reflection

**Techniques Used:**

* **Unit Testing:** Focused on individual classes and methods, ensuring they function correctly in isolation.
* **Boundary Testing:** Tested the boundaries of input constraints, such as string length and date validity, to ensure the system handles edge cases correctly.

**Techniques Not Used:**

* **Integration Testing:** Could test interactions between multiple components.
* **System Testing:** Validates the complete system for compliance with requirements.

**Practical Uses and Implications:**

* **Unit Testing** is ideal for early detection of issues within individual components, making it a staple in Test-Driven Development (TDD).
* **Integration Testing** ensures that different parts of the system work together correctly, important in projects with multiple interacting modules.
* **System Testing** validates the entire application against the requirements, crucial before deployment to ensure everything functions as expected.

#### Mindset

Working on this project, I took a careful approach, paying close attention to the details of the code and its dependencies. For example, making sure the Contact class handled invalid input correctly meant thinking about how different constraints interacted. To avoid bias, I tested the functionality objectively, not letting personal assumptions influence how I thought the code should work. This objectivity was vital, especially for catching edge cases that might be missed. To prevent technical debt, I will keep prioritizing thorough testing and regular code reviews to identify issues early. For instance, I will ensure all new features come with complete tests.