Pravishna Nand

CS-320-X6131 Software Test Automation& QA 23EW6

8/13/23

7-2 Project Two Submission

**Summary**

As I worked on Project One, I adopted a comprehensive unit testing approach to ensure the quality and functionality of the contact, task, and appointment services. For the Contact Service, I designed test cases to verify successful contact retrieval, update, and deletion, covering various input combinations. For the Task Service, I wrote tests to validate task creation, modification, completion status changes, and deletion. For the Appointment Service, I crafted tests to ensure accurate appointment scheduling, modification, and cancellation. By focusing on distinct functionalities and crafting test cases to cover a wide range of scenarios, my unit testing approach aimed to thoroughly validate each feature’s behavior.

My unit testing approach was closely aligned with the software requirements for each feature. In accordance with the requirement that a contact ID cannot exceed 10 characters, I designed a test scenario where an attempt was made to add a contact with an ID longer than 10 characters. The test ensured that an exception was thrown, indicating the correct enforcement of the length constraint. Additionally, I created a test case where a contact’s phone number was provided with less than or more than 10 digits, verifying that only phone numbers with exactly 10 digits were accepted. These specific tests underscored the alignment of my approach with the requirement of contact ID length and phone number format validation.

@Test

**public** **void** testInvalidContactId() {

assertInvalidContactId(**null**);

assertInvalidContactId("");

assertInvalidContactId("12345678901");

}

@Test

**public** **void** testSetInvalidPhone() {

Contact contact = **new** Contact("1234567890", "Ashley", "Brown", "1234567890", "123 Luxerone St");

*assertThrows*(IllegalArgumentException.**class**, () -> contact.setPhone("123456789")); // Phone number with 9 digits

}

To align with the stipulation that task names should not exceed 20 characters, I developed a test case where a task was added with a name longer than 20 characters. The test confirmed that the system correctly rejected the task addition and enforced the character limit. Similarly, in line with the requirement to update task names, I created a test scenario where a task's name was modified and verified that the updated name was accurately stored in the system. These targeted tests showcased the direct alignment between my testing approach and the requirement of name length validation and update functionality.

@Test

public void testCreateTaskWithLongValues() {

assertThrows(IllegalArgumentException.class, () -> new Task("12345678901", "Task Name", "Task Description")); // 11 characters in taskID

assertThrows(IllegalArgumentException.class, () -> new Task("1", "Task Name that exceeds 20 chars", "Task Description")); // 31 characters in name

assertThrows(IllegalArgumentException.class, () -> new Task("1", "Task Name", "Task Description which is too long for this test case")); // As per the description length constraint

}

@Test

public void testAddTask() {

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

Task addedTask = taskService.addTask(task);

assertEquals(task, addedTask); // Verifying that the task has been added to the service

}

In accordance with the requirement for appointment date validation, I designed a test case where an appointment with a past date was attempted to be added. The test verified that the system properly rejected the addition of appointments with dates in the past, aligning with the specified requirement. Additionally, to ensure appointments could be updated and deleted, I created test cases to delete appointment and update description. These concrete examples of tests demonstrated how my approach directly matched the requirement of appointment date validation and appointment update and deletion.

@Test

public void testSetAppointmentInPast() {

Appointment appointment = new Appointment("F123456789", new Date(), "Valid appointment");

Date currentDate = new Date();

Date pastDate = new Date(currentDate.getTime() - 86400000); // One day before the current date

assertThrows(IllegalArgumentException.class, () -> {

appointment.setAppointmentDate(pastDate);

});

}

@Test

public void testUpdateAppointment() {

// Add an appointment and then update its description

appointmentService.addAppointment(appointment1);

String newDescription = "Updated Appointment 1";

appointment1.setDescription(newDescription);

appointmentService.updateAppointment(appointment1);

Appointment updatedAppointment = appointmentService.getAppointmentByID(appointment1.getAppointmentID());

assertEquals(newDescription, updatedAppointment.getDescription());

}

@Test

public void testDeleteAppointment() {

// Add an appointment and then delete it to test deletion functionality

appointmentService.addAppointment(appointment1);

int initialSize = appointmentService.getNumberOfAppointments();

appointmentService.deleteAppointment(appointment1.getAppointmentID());

int newSize = appointmentService.getNumberOfAppointments();

assertFalse(appointmentService.appointmentExists(appointment1.getAppointmentID()));

assertEquals(initialSize - 1, newSize);

}

The overall quality of my Junit tests is well supported by the achieved coverage percentage of above 80%. This high coverage percentage indicates that a substantial portion of the codebase was exercised by my tests, providing strong evidence of their effectiveness in thoroughly validating the software’s functionality. The coverage implies that a significant number of code paths, including various scenarios and edge cases, were tested. By covering a wide range of possible inputs, conditions, and branches, I minimized the likelihood of undetected defects slipping through the testing process. The coverage percentage reflects my commitment to producing robust tests that verify different aspects of the software's behavior. I also want to emphasize that coverage alone does not guarantee absolute correctness. It’s possible to have high coverage but still miss specific tests, such as delete contact in my code. I didn’t have any code to find out if the contact even existed before deleting it.

Writing JUnit tests for Project One was both challenging and rewarding. It demanded a deep understanding of the features and their requirements. Each test required thoughtful consideration of inputs, expected outputs, and potential interactions. It was very time-consuming, and I had to be super organized in making sure I didn’t miss anything. Each time I ran the coverage test, it allowed me to keep increasing the coverage and understand what areas didn’t have any validation tests.

I ensured the technical soundness of my code by closely adhering to best practices in test design and validation. For instance, in the Contact Service tests, I employed assertions to validate specific attributes of the contact object after updating its first name. The use of assertions allowed me to confirm that the contact’s attributes were accurately updated and aligned with the expected values. This approach ensures proper data manipulation and integrity in the system.

@Test

public void testSetValidFirstName() {

Contact contact = new Contact("1234567890", "Ashley", "Brown", "1234567890", "123 Luxerone St");

contact.setFirstName("James");

assertEquals("James", contact.getFirstName());

}

In this example, the test initializes a contact with the provided attributes and then updates its first name using the setFirstName method. The subsequent assertion using assertEquals confirms that the updated first name matches the expected value "James". This rigorous validation validates that the method behaves correctly and maintains the consistency and integrity of the contact's data, demonstrating my commitment to technical soundness in my JUnit tests.

I ensured the efficiency of my code through a focused and streamlined approach to test design, as demonstrated by the following example from the Task Service tests. In this test case, I directly validate the addition of a task to the task service while minimizing unnecessary steps.

@Test

public void testAddTask() {

Task task = new Task("1", "Task Name", "Task Description");

Task addedTask = taskService.addTask(task);

assertEquals(task, addedTask); // Verifying that the task has been added to the service

}

The test directly adds a task to the task service using the addTask method and assigns the returned task to the addedTask variable. The subsequent assertion checks if the added task matches the original task, ensuring that the task was successfully added to the service.

**Reflection**

I extensively used unit testing to validate individual components of the software. Each unit test focused on a specific function or behavior and ensured that it worked as expected. For instance, in the Contact Service, I tested the creation, retrieval, update, and deletion of contacts independently. I performed integration testing to verify interactions between different components of the software. In the Appointment Service, I ensured the accurate interaction between the appointment scheduling mechanism and the participant management functionality. This ensured that appointments were correctly scheduled with the intended participants, validating the integration of these two critical components.

While I conducted unit and integration testing, I did not perform acceptance testing in this project. Acceptance testing involves validating the software's adherence to business requirements from a user perspective. In the context of Project One, this would involve running the entire application and testing its features to ensure it meets user needs. Performance testing involves assessing the system's responsiveness and scalability under different loads. Since Project One focused on developing and testing back-end services, I did not directly address performance testing in this context. Performance testing would be more relevant for applications with heavy user traffic, such as web applications or services with high transaction volumes.

Each of the testing techniques discussed holds practical significance and carries specific implications tailored to distinct software development projects and scenarios. For instance, in a financial application where accuracy is paramount, unit testing would be indispensable for rigorously verifying calculations in isolated modules, such as tax calculations or interest rates. In contrast, in a large-scale e-commerce platform, integration testing becomes critical to ensure that user authentication, inventory management, and payment processing components seamlessly collaborate to provide a smooth shopping experience. This is very simple to our Project, as the contact creation was crucial to setting the appointment date.

Throughout this project, I adopted a cautious yet curious mindset, embodying the role of a diligent software tester. I acknowledged the immense importance of meticulous testing, fully aware of the potential consequences of undetected defects on the application's overall functionality. A prime example of this mindset came when testing the Contact Service. By thoroughly understanding the intricate interplay between the attributes of the Contact object, such as the unique contact ID, first name, and phone number, I ensured comprehensive test coverage. This careful approach was aimed at preventing any potential inconsistencies or data integrity issues that could adversely impact user experiences or data management.

To limit bias in code review, I maintained a neutral standpoint, focusing solely on the software's adherence to requirements. I steered clear of personal preferences or assumptions about the code's behavior. There were many times when I wanted to modify the code according to how I wanted it. But I had to make sure I followed the requirements. If I were responsible for testing my own code, biases could creep in, affecting my judgment. For instance, if I assumed that my implementation was correct without robust testing, I might overlook critical defects or vulnerabilities. I do believe this is one of the biggest mistakes developers make. We trust our code too much and forget the importance of testing.

Being disciplined in commitment to quality is paramount to ensure a reliable software product. Cutting corners in writing or testing code can lead to undetected defects, resulting in maintenance challenges and customer dissatisfaction. For instance, if I rushed through testing and overlooked boundary cases in the Contact Service, it could lead to incorrect data storage or retrieval, affecting user experiences. As a practitioner, I plan to avoid technical debt by consistently following best practices, conducting code reviews, and investing time upfront to create thorough test cases. This will prevent the accumulation of unresolved issues and enhance the long-term maintainability of the software.