Randy Ortiz

Professor Gudivada

CS-320 Software Test, Automation QA

August 16, 2024

As a software engineer for Grand Strand Systems, I was responsible for the development and testing of the back-end services of a mobile application. My focus was on the contact, task, and appointment services. In this report, I will be explaining my approach to unit testing, and my overall experience with testing.

For the Contact Service, I concentrated on verifying core CRUD operations: creating, reading, updating, and deleting contacts. I identified key functions such as adding new contacts, preventing duplicates, and validating input data. For instance, the ‘testAddContact()’ method ensured a contact was correctly added and retrievable using ‘assertEquals(expectedContact, actualContact)’. The ‘testAddDuplicateContact()’ method checked that adding a duplicate contact threw an IllegalArgumentException using ‘assertThrows(IllegalArgumentException.class, () -> contactService.addContact(duplicateContact))’. Boundary testing handled edge cases like empty fields or excessive lengths. In ‘testContactCreationWithEmptyFields()’, I verified that missing required fields threw an IllegalArgumentException. For the Task Service, I focused on testing task management functionalities. Methods like ‘testAddTask()’ verified task creation and retrieval. ‘assertNotNull(taskService.getTask(taskId))’ confirmed tasks were correctly managed. The ‘testDeleteTask()’ method ensured tasks were deleted with ‘assertNull(taskService.getTask(taskId))’ after deletion. In the Appointment Service, I tested the creation and deletion of appointments. The ‘testAddAppointment()’ method verified that an appointment could be added and retrieved using ‘assertEquals(expectedAppointment, actualAppointment)’. The ‘testDeleteAppointment()’ method ensured removed appointments were no longer retrievable with ‘assertNull(appointmentService.getAppointment(appointmentId))’.

My testing approach aligned with the software requirements of the project. For the Contact Service, I implemented measures to prevent duplicate contacts as specified. ‘testAddDuplicateContact()’ confirmed that duplicate entries were correctly identified and rejected. In the Task Service, I enforced constraints in the Task constructor, such as if ‘(taskId == null || taskId.length() > 10)’ throw new IllegalArgumentException("Invalid task ID"), catching invalid IDs early. For the Appointment Service, I ensured appointment IDs did not exceed 10 characters and were not null, as validated in ‘testAddAppointment()’.

I assessed the quality of my JUnit tests based on the code coverage percentages. I maintained coverage above 80% for all services. For example, ‘testContactCreation()’ validated attributes and constraints in Contact Service, ensuring fields like contact ID and address met required formats. In Task Service, ‘testAddTask()’ ensured tasks were correctly managed. In Appointment Service, ‘testAddAppointment()’ confirmed successful addition and retrieval of appointments. These tests validated the code’s reliability.

Writing the JUnit tests involved ensuring technical soundness and efficiency. In Task Service, ‘assertThrows(IllegalArgumentException.class, () -> new Task(null, "Test Task", "Description"))’ confirmed proper error handling. In Contact Service, ‘testContactCreation()’ validated that contact IDs, names, and addresses were correctly set with assertions like ‘assertEquals("ExpectedName", contact.getName())’. Efficient testing was achieved by focusing on core functionalities and edge cases.

The primary testing technique I used was unit testing. This approach involved testing individual components in isolation to ensure they worked as intended. Tests like ‘testAddAppointment()’ and ‘testDeleteAppointment()’ ensured that appointments could be managed correctly. I chose to focus on unit testing for this project because it allows for early detection of bugs and is particularly effective for testing individual functionalities. Boundary testing was also employed, as seen in ‘testContactCreationWithEmptyFields()’, to handle input constraints and validate edge cases. Techniques not used include integration testing, which verifies module interactions, and system testing, which evaluates the entire system. These were not the focus of this project but were essential for validating software interactions and overall functionality.

For this project, I adopted a cautious mindset as a software tester considering that the code interactions required careful examination and testing. When working on Appointment Service, I thought how adding and deleting appointments could impact the system, ensuring the service managed appointments correctly. By focusing on the criteria and requirements, I minimized bias and ensured that the code met its requirements and functioned properly. The requirements dictated my testing approach, maintaining objectivity throughout.

I’ve learned the importance of discipline and commitment to quality in software engineering. Cutting corners or skipping tests can lead to software issues and increase technical debt. By thoroughly testing the code for this project and covering all relevant scenarios, I've demonstrated my commitment to quality. Moving forward, I will continue to avoid technical debt by adhering to the best coding and testing practices, ensuring the code remains reliable and robust.