CS-320

Professor Omar Toledo

Zeb Hawthorne

June 23, 2024

7-2 Project

Developing the code for the mobile application required a wide approach to unit testing to ensure reliable functionality. Each service's unit testing strategy was written to cover specific features and requirements. For the AppointmentService, the unit testing approach focused on the creation, updating, and deletion of appointments. This included ensuring that invalid inputs were handled properly and that the internal state of the service remained consistent after each operation. Specific methods tested included addAppointment, updateAppointmentDate, deleteAppointment, and validation for the Appointment constructor.

The ContactService testing strategy was centered around verifying the correct addition, updating, and deletion of contacts. Methods such as addContact, updateContactFirstName, updateContactLastName, updateContactPhoneNumber, updateContactAddress, and the Contact constructor were thoroughly tested. These tests were designed to ensure that invalid data was properly rejected.

Similarly, for the TaskService, the approach involved creating tests to add tasks, update their names and descriptions, and delete tasks. Each test was made to verify the correct handling of both valid and invalid inputs.

The unit testing approach was as closely aligned with the software requirements as possible, as each test was designed to validate specific functions described in the requirements. For instance, the requirement that an appointment date must be in the future was tested by attempting to set past dates and expecting exceptions. An example of this is the test for invalid appointment dates, which ensured that the application rejected dates set in the past:

@Test  
public void testInvalidAppointmentDate() {  
 Date pastDate = new Date(System.currentTimeMillis() - 10000); // 10 milliseconds in the past  
 assertThrows(IllegalArgumentException.class, () -> {  
 new Appointment("12345", pastDate, "Description");  
 });  
}

The overall quality of the JUnit tests can be defended based on their comprehensive coverage and validation of both normal and edge cases. The effectiveness of these tests was evident through high coverage percentages, over 80 percent in total, focusing on various inputs and states of the application. By testing both valid and invalid scenarios, the tests provided confidence in the correctness and reliability of the code.

Writing the JUnit tests was an insightful experience, reinforcing the importance of thorough testing in software development. Ensuring that the code was technically sound involved creating tests that covered a wide range of inputs and scenarios. For example:

@Test  
public void testUpdateAppointmentDateToInvalidDate() {  
 Date futureDate = new Date(System.currentTimeMillis() + 10000);  
 Date pastDate = new Date(System.currentTimeMillis() - 10000);  
 Appointment appointment = new Appointment("12345", futureDate, "Description");  
 service.addAppointment(appointment);  
 assertThrows(IllegalArgumentException.class, () -> {  
 service.updateAppointmentDate("12345", pastDate);  
 });  
}  
This test ensured that invalid updates were handled appropriately. Efficiency was maintained by minimizing redundancy and focusing on critical paths. For example:

@Test  
public void testAddAppointment() {  
 Date currentDate = new Date(System.currentTimeMillis());  
 Date afterToday = new Date(currentDate.getTime() + 10000); // 10 milliseconds in the future  
 Appointment appointment = new Appointment("12345", afterToday, "Description");  
 service.addAppointment(appointment);  
 assertEquals("12345", appointment.getAppointmentId());  
 assertEquals(afterToday, appointment.getAppointmentDate());  
 assertEquals("Description", appointment.getDescription());  
 assertEquals(appointment, service.getAppointment("12345"));  
}

Several testing techniques were employed in this project. Black-box testing was used to test functionalities without considering the internal code structure, ensuring that tests were written based on requirements and expected behavior. Boundary testing focused on testing the boundaries of input ranges, such as date values for appointments or the length of strings for IDs and descriptions. Exception testing was created to ensure that invalid inputs and operations threw the right exceptions.

Other software testing techniques not used in this project include white-box testing, which involves testing internal structures or workings of an application; integration testing, which tests multiple modules or services together to ensure they work correctly in combination; system testing, which evaluates the complete and integrated software's compliance with its specified requirements; and acceptance testing, which verifies if the system meets business requirements and is ready for deployment.

The practical uses and implications of these techniques vary. White-box testing is useful for ensuring code paths, conditions, and branches are executed and tested. Integration testing ensures that different modules or services interact correctly, which is crucial in microservices architectures. System testing validates the system’s overall behavior, essential before deployment. Acceptance testing ensures the final product meets business needs and is ready for production.

The mindset adopted during this project was one of caution and testing the waters, since this was the first time in using JUnit testing. Appreciating the inter-connections of the code was crucial to anticipate how changes might affect other parts of the system. For instance, ensuring that appointment dates were always in the future required understanding how date manipulations could impact other functionality.

Limiting bias in code review involved adopting an objective perspective and considering various user scenarios. Bias in self-testing could lead to overlooking edge cases or invalid inputs. For example, assuming valid inputs based on coding habits might overlook invalid and unexpected inputs. An example of this is testing for invalid first names:

@Test  
public void testInvalidFirstName() {  
 assertThrows(IllegalArgumentException.class, () -> {  
 new Contact("12345", null, "Doe", "1234567890", "123 Main St");  
 });  
}

Commitment to quality is crucial to avoid technical debt, which can hinder future development and maintenance. Cutting corners in testing can lead to undetected bugs, requiring more effort to fix later. To avoid technical debt, I plan to maintain high testing standards, ensure thorough documentation, and adopt best practices such as code reviews and continuous integration. Regularly running tests during development ensures immediate detection of issues.

The unit testing ensured the robustness and reliability of the code. Using various testing techniques and maintaining an objective mindset were key to the success of testing these services. This commitment to quality and thorough testing practices will guide future projects, ensuring the delivery of reliable and maintainable software.