David Hughes

2/21/24

CS-320 Project 2

The unit testing approach for the mobile application services—contact, task, and appointment—was thoughtfully devised to align with the software requirements. My strategy involved methodical creation of tests that evaluated each feature's functionality and boundary conditions. For the Contact Service, I concentrated on ensuring the fields adhered to the required character limits. An example is the test case shouldThrowExceptionForLongContactId(), which validated that contact IDs exceeded no more than ten characters, directly reflecting the software's stipulations. The Task Service demanded a focus on the uniqueness of task IDs and constraints on the name and description lengths, culminating in tests like testAddTaskWithDuplicateId() that verified the rejection of redundant IDs. Lastly, the Appointment Service required diligence in handling date-related validations. The test shouldThrowExceptionWhenSettingPastAppointmentDate() ensured no appointments were set in the past, in compliance with the project's requirements. The overall quality of my JUnit tests was demonstrated through a coverage metric exceeding 80%. This number was not just a quantitative measure but a testament to the tests' ability to systematically exercise the code. High coverage indicated a broad and deep validation of the services' functionality, increasing confidence in the reliability and robustness of the application.

Writing JUnit tests was an exercise in balancing precision with efficiency. I ensured technical soundness by crafting test scenarios that directly corresponded with the functional requirements. For example, the line assertEquals("1234567890", contact.getPhone()); from the ContactTest suite confirmed the exactitude of the phone number format as required. Efficiency stemmed from writing streamlined tests that avoided unnecessary duplication. A single test case like assertThrows(IllegalArgumentException.class, () -> new Contact("12345678901", ...)); was sufficient to assert the constraints on the contact ID, eliminating the need for multiple similar test cases.

In the project, I employed several software testing techniques that were crucial in ensuring the robustness of the application. Boundary Value Analysis (BVA) was a primary technique, which involved testing at the edges of input ranges. For example, if a field required a maximum of 10 characters, I tested with 10 and 11 characters to check the application's response. Equivalence Partitioning was another technique used to divide inputs into logically similar groups, thereby reducing the number of test cases while still thoroughly testing functionality. An instance of this was grouping all valid phone numbers to test the 'add contact' feature, assuming they should all yield the same result—a successfully added contact. There were also techniques I didn't employ; State Transition Testing wasn't used due to the lack of complex state changes in our application. This technique is often characterized by the need to test different state sequences that an application can go through. Similarly, Decision Table Testing wasn't applicable, as our application didn't have complex business rules that would require assessing combinations of inputs leading to different outcomes. Each of these techniques has its place in software development. BVA is particularly useful when testing user input fields, while Equivalence Partitioning can simplify testing by reducing redundant checks. State Transition Testing has its practical use in applications like online banking systems where account states may change, and Decision Table Testing fits well in complex business environments where multiple conditions influence the flow of operations.

Throughout the project, caution was my guiding principle. I recognized early on that appreciating the complexity and interconnectedness of the code was essential. For instance, while testing the updateContact function, I had to consider how changes to one attribute might affect the retrieval of another contact attribute, given the object-oriented nature of the application. To limit bias, I approached my code as if it were written by someone else. I questioned every function and challenged every assumption, acknowledging that bias could lead to overlooking potential defects. For example, even if I believed my implementation of the deleteAppointment method was correct, I would still test scenarios where it could fail, like attempting to delete an appointment that didn't exist. Being disciplined in my commitment to quality was a non-negotiable aspect of my work. The importance of not cutting corners became clear when I realized that any oversight could later manifest as a defect, impacting the user experience or leading to technical debt. For example, ensuring that the setAppointmentDate method correctly handled future dates was not just about meeting a requirement but also about future-proofing the application. As a software engineering student and aspiring professional, my goal is to continue practicing thorough testing and thoughtful code review, striving for clarity, maintainability, and quality in every line of code I write. This commitment, I believe, is foundational to avoiding technical debt and delivering software that stands the test of time and use.