Project 2 CS-320

Daniel Dobbs

SNHU

For the Contact Services, a white-box approach was employed. Every function, from CRUD operations to more subtle functionalities like preventing the addition of duplicate contacts and handling requests for non-existent contacts, was meticulously tested. Mock data was used to simulate a variety of contact scenarios to validate each function against both typical and edge cases. For task Services, given the time-sensitive nature of tasks a special emphasis was placed on boundary conditions. The system was rigorously tested to ensure that tasks with deadlines that fell out of permissible limits were appropriately flagged. Additionally, scenarios with overlapping tasks were played out to ensure that the application’s scheduling algorithms were functioning optimally. For appointment services, a mix of positive and negative testing strategies was integrated into the approach for this service. The system was not only tested to ensure it functioned correctly under normal circumstances but also to ensure it appropriately handled and flagged erroneous inputs, such as attempts to book appointments in the past or exceed description length constraints.

An intricate understanding of the software requirements for each module guided the entire testing process. Each test case was crafted in tandem with the associated requirement it was meant to validate. For instance, one of the key requirements for appointments were that they had to be scheduled in a present or future time. To validate this, a test (‘testInvalidAppointmentDate()’) was created to simulate attempts to create an appointment at a past date. The expected outcome was valid as no appointment can be set in the past as said in the requirements. The fact that such explicit tests consistently returned expected results speaks volumes about the alignment of the testing approach with the software requirements.

Using the tools of Eclipse IDE, we determine a commendable coverage of approximately 95%+. Such extensive coverage ensures that most of the application’s functions, branches, and logical paths were under scrutiny and validated. The journey of writing Junit tests, were enlightening, was also filled with its own set of challenges. Understanding the software requirements in their entirety was the first hurdle. However, as the project progressed, translating these requirements into viable test cases became increasingly intuitive. To ensure the technical soundness of the code, assertions like ‘assertEquals()’, ‘assertTrue()’, ‘assertNull()’, ‘assertThrows()’, and ‘assertNotNull()’ were utilized ubiquitously. One illustrative example would be the ‘testAddTask()’, where adding a new tasks was expected and validated to return true. Efficiency was another cornerstone of our testing philosophy. By ensuring tests were atomic and each one focused on validating a single functionality, any breaking changes could be identified and rectified faster. The atomic nature of tests like ‘testDeleteTask()’, which solely verify the task deletion process, embodies this principle.

A wide spectrum of testing techniques found their way into this project. Boundary Value Analysis or BVA was paramount, especially for tasks like date and length validations in tasks and appointments. Equivalence Partitioning was another staple, which helped categorize similar input sets expected to be handled in a uniform fashion. Given the intimate knowledge of the code’s internal logic, White-Box Testing was the method of choice to ensure every nook and cranny was put to the test. While the above techniques were actively utilized, some like Black-Box Testing and Stress Testing didn’t find much utility given the project’s scope and nature.

Throughout the project, a constant state of caution was adopted. The sheer interconnectedness and complexity of the code meant that even minor oversights could cascade into significant functionality issues. One prime lesson from this project was the importance of perspective and objectivity. While it’s tempting to assume our code to be devoid of flaws, it’s a bias that testers need to shed. This was circumvented through methods like peer reviews where I got my friend to go over the code to make sure I was doing everything right. Lastly, the most resonant lesson from this endeavor was the irreplaceability of quality. In software development, shortcuts might lead to transient success but invite long-term complications. Staying updating, rigorous testing, and periodic refactoring are a few methodologies I plan to adopt to uphold the highest standards of quality and avoid the pitfalls of technical debt.

In conclusion, project one was a rich tapestry of technical challenges and invaluable lessons. While the immediate objective was the successful development and validation of the three services, the broader takeaway was a reinforced respect for quality and the intricate processes that ensures it.