Liu, Kevin

9 October 2024

Professor Kalysa Wilson

CS-320 Software Test, Automation

CS 320 Project Two

In Project One, I developed unit tests for the contact, task, and appointment services to ensure the features worked as intended. For the Contact Service, my tests focused on making sure user input was valid, ensuring that contacts were correctly created, and flagging errors when information like names or email addresses were missing or improperly formatted. For example, I tested edge cases like maximum field lengths and required fields. Similarly, for the Task Service, I verified that tasks could be added, updated, or deleted smoothly. I also accounted for scenarios like what happens when there are no tasks available. For the Appointment Service, my tests checked whether appointments could be created, modified, or deleted correctly, while also looking for issues like scheduling conflicts or invalid appointment times. By testing these services across a range of situations, I could ensure that they behaved correctly.

My unit testing approach closely followed the project’s software requirements, making sure each feature was covered thoroughly. For instance, the software required that users be able to add, update, and delete contacts, tasks, and appointments. My tests were built to ensure these actions worked as specified. An example is the test I wrote for contact creation, which included checks for valid email addresses. I tested scenarios like missing symbols or incorrect domains to make sure the system handled errors as expected, meeting the requirement exactly.

The quality of my JUnit tests can be judged by how well they covered all the key functionalities and edge cases. I didn’t just test successful actions; I also made sure to test failure cases, like trying to add a task with missing details. I used assertions effectively to compare expected and actual outcomes, which helped ensure my results were accurate. By designing these tests with such a wide range of inputs, I’m confident that they provided strong, reliable feedback on how well the code was working.

Writing JUnit tests was a great learning experience that helped me think carefully about both correctness and efficiency. To ensure the tests were technically sound, I followed best practices, like keeping the tests focused on small, specific functions. For example, on line X of my task creation test, I used the assertEquals() method to check that tasks were saved correctly in memory. I also structured my tests to avoid redundancy, focusing on edge cases like boundary conditions for appointment start and end times to maximize efficiency.

In this project, I used units, boundaries, and negative testing. Unit testing allowed me to test individual components like contact creation in isolation from the rest of the system. This technique is great for catching issues early and ensuring each piece of code works correctly before it interacts with other parts of the application. Boundary testing is used to test input limits, such as the minimum and maximum lengths allowed for text fields. Finally negative testing ensures that the application could handle invalid inputs and unexpected scenarios gracefully to prevent unintended behaviors or crashes. These techniques helped me identify potential problems early in the process, but there were also other testing techniques I could have considered, such as integration and system testing. Integration testing would have been useful to check how the different services interacted with each other and the user interface. However, since the project was mainly focused on back-end services, integration testing wasn’t necessary. System testing looks at the entire system, which would have been more relevant during the later stages of development which didn’t apply to this project that focuses on core functionalities.

Throughout the project, I took a cautious approach, always keeping in mind the complexity of the code and how different parts might interact. For example, when testing the Appointment Service, I knew that scheduling conflicts could occur, so I specifically wrote tests to check for overlapping appointments. This careful approach was essential because even minor oversights can lead to significant issues in real-world applications. By staying cautious, I made sure that my tests covered potential problems that might not have been obvious at first glance.

One challenge of testing my own code is personal bias. Sometimes I avoid testing areas where I assume everything will work fine but ends up being a vulnerability. To counter this, I focused on creating tests that would challenge any assumptions. For example, I intentionally wrote tests that fed incorrect data into the system, trying to create contacts without required fields. This helped me discover edge cases I hadn’t thought of initially. Revisiting my test cases frequently also helped ensure that I stayed objective and didn’t miss critical issues.

In software development, sticking to high standards and avoiding shortcuts is essential for preventing future problems. Throughout this project, I made sure to avoid taking the easy way out, especially when it came to testing. I carefully tested boundary conditions, such as the length limits on contact names and appointment times, to ensure everything was thoroughly validated. By maintaining this level of discipline, I reduced the chances of accumulating technical debt. Moving forward, I plan to keep following the best practices in both development and testing to make sure my code remains high quality and avoids future issues.