Dawson Kennedy

06/21/2024

CS-320-11104-M01 Software Test, Automation QA 2024 C-3

7-2 Project Two Submission

For the contact, task, and appointment services, my unit testing approach focused on verifying individual functionalities in isolation to ensure correctness and reliability. Each feature was tested using JUnit tests to cover a wide range of scenarios, including edge cases and typical use cases. Starting with the contact services case the approach I used to make the tests was to validate the adding, updating and deleting of the contacts. These tests ensure that each contact has unique identifiers and correct information fields. An example of this in my JUnite tests would be found in the tests codes for the contact service. To highlight one I used a test case on the AddContact method to assert a NotNull to a new contact which I named and gave an ID to make sure that the code to handle the adding of a new contact. Moving on to the task service method. The tests created here were written to handle task creation, completion, and deletion. The tests verified that task deadlines and priority settings were functioning correctly. An example of this would be the code to test a CompletedTask where in it it defines a task from the string of tasId and calls it by name and description of task to then assert that it is True to mean that the task is complete and would show up as marked complete. In the last method, appointment, the focus here was on creating, updating, and canceling appointments. These tests were made to ensure that the appointment conflicts and reminders worked as expected. An example of the test I used in the tests case would be that of the AddAppointment where I set the start time and end time in local time, added the appointment with a name of the appointment, the start and end time, and asserted it as a false to show that the appointment had not yet started but it was made and added to the appointments list.

With all that I would say that my approach was closely aligned with the software requirements provided by the customer. The tests were derived from the functional specifications and acceptance criteria for each service. For instance there was a requirement that stated that they prevent overlap of appointments in which I directly tested by creating conflicting appointments and ensuring that the system would reject them. Furthermore I would say that my overall quality of my JUnite tests is high. I would say that I achieved a wide code coverage from my tests that included both positive and negative tests to ensure that I was making a robust code.

When writing these JUnite tests I took a more methodical approach and paid a little more attention to detail so I would get the test cases correct. I ensured my code was technically sound by writing tests that covered different input scenarios and edge cases. For example , in the tasks service, I checked for invalid tasks IDs to ensure that there was proper error handling. Maintaining efficiency was also a concern when making these codes. To make sure that I did maintain efficiency while making these I optimized my tests to avoid redundancy. An example of this would be when I used parameterized tests in my JUnit tests to test multiple input values in a single test method reducing code duplication.

Reflecting on what I did through this project and looking back at the testing techniques I did use, and why, versus the ones I did not use, any why. I want to start with the ones I did use. Starting out with the Unit testing. This one is quite obvious but still nonetheless a testing technique I employed through this course. Unit testing focuses more on testing of the individual components or functions in isolation which can be essential for ensuring the correctness of individual parts of the application before integration. Another technique I used was Integration testing. Integration testing tests the interaction between integrated units and ensures that combined parts of the system work together as expected. The last technique that I used that I would like to talk about is system testing. In system testing it tests the complete system as a whole to validate the end to end scenarios and is useful for validating the overall functionality and user workflows.

Moving on to touch on some techniques that I did not employ in my project. Starting off with regression testing. Regression testing re-tests the system to ensure that new changes have not introduced new bugs and is important for maintaining stability in ongoing development. There is only one other testing that I want to talk about that I did not use and that is Performance Testing. Performance testing is a test that assesses the speed, responsiveness and stability of the system under load and is crucial for applications with high traffic or performance critical operations.

The practical uses and implications that come from these tests, both used and not, is that each test has its place in the different stages of the software development lifecycle. Unit testing is foundational and is used early in the development process. Integration and system testing follow, ensuring that modules work together and that the entire system meets requirements. Regression and performance testing are often part of an ongoing maintenance and optimization efforts.

This project to get us to put on a few hats, mainly software tester. and software developer. While earring the software developer hat, I employed a cautious mindset, meticulously verify the correctness of each function. Appreciating the complexity and interrelationships of the code was crucial to avoid overlooking subtle bugs. For example, ensuring that the appointment service correctly handled overlapping times required understanding the interaction between different parts of the scheduling logic. Switching hats to the software developer, limiting bias was essential to maintain objectivity. I approached the code as if I were an external reviewer, questioning assumptions and validating behaviors from a user's perspective. Bias would indeed be a concern since I was testing my own code, as familiarity might lead to overlooking potential issues.

Commitment to quality is vital in software engineering. Cutting corners can lead to technical debt, which accumulates and becomes harder to manage over time. To avoid this, I plan to adopt practices like continuous integration and regular code reviews. Learning how to automate tests would be a great way to help me maintain high code quality and help me catch bugs and issues early in the process, rather than later, to effectively make strong and robust code more efficiently.