Justin Osman

CS 320 Software Test Automation

Professor Prasad

12/10/2023

*Summary & Reflections Report*

When developing the mobile application for the customer, a thorough unit testing approach was used for Contact, Task, and Appointment Services features. I set out to diligently design tests that validated every requirement that was given for each service. Each individual service had its own set of tests that covered the various scenarios. In doing this I was able to ensure functionality, correctness, and reliability. The testing chosen was meant to align with the software requirements, ensuring that each service met its designated specifications. For example, in Contact Service, tests were crafted to validate the uniqueness and length constraints of contact ID, first and last name, phone number and address. Not all the requirements were the same for each variable, so a test needed to be created for each one. This guaranteed no overlap in testing which made it easier to achieve 100% coverage in all tests in all services. Aligning the tests to the requirements was important to delivering a mobile application that fulfilled the needs of the client.

The JUnit tests exhibited a high level of quality, offering comprehensive coverage across diverse scenarios and functionalities. Positive and negative test cases, along with edge case testing, were systematically integrated to ensure the correctness of the implemented code. Being able to reach 100% coverage on the test cases in each service gives confidence to the application that we have covered almost all the bases that the client asked for.

When writing the JUnit tests, it took me multiple attempts to ensure the technical soundness of the code. Take In Appointment Service, the `setAppointmentDateTest` stands out as an example. This test not only verified the functionality of setting an appointment date but also ensured technical correctness by validating that the appointment date is always set in the future. The mentioned example is below:

**@Test**

**public void setAppointmentDateTest() {**

**// Arrange**

**Appointment appointment = new Appointment();**

**// Act**

**appointment.setAppointmentDate(new Date());**

**// Assert**

**assertTrue(appointment.getAppointmentDate().after(new Date()));**

**}**

Efficiency in code implementation was always in the back of my mind when writing the JUnit tests. An example is found in the `deleteTask` method within Task Service. This method illustrates an efficient deletion process by shifting the array left. This approach minimizes unnecessary iterations and ensures that redundant elements are set to null, contributing to optimal memory utilization. By trying to use coding practices like these I believe it contributes not only to the confirmation of functionality but also to the overall performance of the program. The deleteTask method example is seen below:

**public void deleteTask(String taskId) {**

**int index = findIndexWithId(taskId);**

**if (index != -1) {**

**for (int i = index; i < numOfTasks - 1; i++) {**

**tasks[i] = tasks[i + 1];**

**}**

**tasks[--numOfTasks] = null;**

**}**

**}**

Various testing techniques were applied throughout the project, including boundary value analysis, equivalence partitioning, positive and negative testing, and date validations. These techniques were chosen based on the specific requirements of each feature, systematically uncovering potential issues in different scenarios.

Stress and performance testing were some of tests that I did not use when making this application. That does not mean that they are not valuable techniques for projects with higher user loads and scalability concerns. Stress testing, for instance, can reveal how the system behaves under extreme conditions, providing insights into potential bottlenecks and areas for improvement.

In acting as a software tester, I made sure to have a very cautious mindset. Understanding the complexity and interrelationships of the code was something I had to take my time with, in order to design effective tests and ensure the overall reliability of the mobile application. For example, when assessing coverage in the JUnit tests, it was sure to recognize that shaded red code indicated portions not covered by the test suite. This awareness ensured a better examination of potential vulnerabilities and gaps in the testing strategy.

I made every effort to limit bias in the code review by adopting an objective and systematic testing approach. I used various input scenarios to assess the system's behavior without any preconceived notions, ensuring a more accurate evaluation of the software's functionality.

Making sure I stayed disciplined in code writing and testing is important for giving the user high quality software. The `updateTaskDescription` method in the Task Service serves as a testament to this discipline, demonstrating adherence to coding standards and best practices. This method allows updating the description of a task, ensuring that the new description meets specified requirements. By validating the input parameters and enforcing length constraints. Going back and reviewing the code that I already created, such as method `updateTaskDescription`, played a large role in upholding quality, and creating code that is not only functional but also resistant to potential technical debt.

In conclusion, the unit testing approach, alignment with software requirements, quality of JUnit tests, and the disciplined mindset collectively come together to deliver strong and reliable services for the mobile application. The comprehensive testing strategies highlight the commitment to quality, with an emphasis on efficient, technically sound, and bias-free code. These practices will aid in future efforts to avoid technical debt and deliver software solutions that meet and exceed client expectations.