7-1 Project Two

Gavin Bish

SNHU

CS 320: Software Test, Automation QA

October 17th, 2024

In the development of the contact, task, and appointment services for the mobile application, I employed a structured unit testing approach that focused on the core functionality of each feature. For the contact service, I used white-box testing, which involved thorough validation of contact creation, ensuring that all fields met the specified criteria such as the contact ID being non-null and not exceeding 10 characters. Similarly, for the task and appointment services, my approach followed the software requirements by implementing test cases to validate that inputs adhered to constraints like task IDs and appointment dates.  
 My testing approach was closely aligned with the software requirements. For instance, the task ID validation logic was directly based on the requirement that IDs should not exceed 10 characters. I incorporated assertions in my tests, which validated that any incorrect input resulted in error flags, while valid inputs proceeded successfully. These unit tests covered the essential functionality, ensuring that the software behaved as expected under normal conditions.  
 The quality of the JUnit tests was validated through coverage reports, which showed that a high percentage of the code was covered by the tests, particularly in the service and contact classes. While some areas, like the task class, had lower coverage, the overall tests were robust in identifying and confirming that critical functionalities worked as required. The positive coverage percentages were a strong indicator of effectiveness, ensuring that the majority of the code was thoroughly tested.  
 Writing JUnit tests for this project allowed me to refine my testing strategies. One of the main challenges was ensuring that all edge cases were covered. For example, in the contact service, I included test cases to validate deletion functionality by asserting the size of the contact list before and after each deletion operation:

service.deleteContact("1234");

assertEquals(2, service.getContactList().size());

This code snippet illustrates how I ensured the code’s correctness. I also applied efficient coding techniques by avoiding redundant code structures and utilizing arrays and lists to optimize test operations.  
 I employed several software testing techniques, including unittesting and white-box testing, which involved testing the internal structure of the code by validating various inputs and outputs. The unit tests focused on individual functions and their adherence to specifications, while white-box testing ensured that internal logic and code paths were executed correctly.

I did not use static testing, a technique where code is reviewed without execution, typically through code inspections or walkthroughs. Static testing can identify potential issues before running the code, but I preferred dynamic testing (running code) to verify functionality.

The use of unit testing was practical for validating specific functions, especially in a project with clear requirements. White-box testing helped ensure that my code met internal logic expectations, while static testing, though unused, would have been beneficial in catching issues early.  
 Throughout this project, I adopted a mindset of careful attention to detail. As a software tester, caution was essential to ensure that even small mistakes did not propagate through the system. Understanding the interrelationships between code components was key to this project’s success, especially in scenarios where changes in one module could affect others. For instance, changes to the task service required updates to the corresponding JUnit tests to ensure alignment.

Bias in testing can be a significant concern, especially when testing your own code. To limit bias, I took a step back after writing the code and approached the tests with fresh eyes, focusing on the requirements rather than my assumptions about the code. For example, while testing the task service, I initially overlooked boundary conditions but revisited the code to ensure all edge cases were accounted for.

Being disciplined in writing and testing code is critical in software engineering. Cutting corners can lead to technical debt, which can accumulate over time and lead to bigger issues. As I continue to grow in this field, I will avoid technical debt by following best practices such as code reviews and continuous integration to ensure that my code is not only functional but maintainable in the long term.

**Resources:**

Fowler, M. (2004). Continuous integration: Improving software quality and reducing risk. Addison-Wesley Professional.

Beck, K. (2003). Test-driven development: By example. Addison-Wesley Professional.