Maraiah Frelix

CS 320

August 15th, 2025

Project Two: Summary and Reflection

\*\*Summary\*\*

The testing approach was aligned with the software requirements, focusing on the necessary methods implemented within the base code. In the Contact class, the requirements specified that both the first name and last name must not exceed ten characters in length. Accordingly, the ContactTest JUnit incorporated the assertion "Assertions.assertThrows(IllegalArgumentException.class," reflecting this restriction. Furthermore, the requirements for the Task class indicated that the task ID should also not exceed ten characters. Consequently, the TaskTest JUnit included the method "void testTaskIdTooLong()," which raises an illegal argument exception if the task ID exceeds the specified limit.

The quality of the JUnit tests has been gradually improving, with the resources provided in the modules proving to be significantly beneficial. The coverage percentage was notably higher in the contact tests compared to the task tests. However, both functionalities revealed that the service tests demonstrated a much higher coverage percentage. The positive coverage results indicated the extent to which the tests addressed the functions. I plan to utilize these insights more frequently to ensure that each code statement is tested in at least one corresponding test case.

In order to guarantee the technical soundness of the code, I employed various measures, including the use of data structures such as arrays. For instance, in the Contact class, I implemented "private static List<String> CONTACT\_IDS = new ArrayList<String>();" to create a list for storing strings. I also utilized common algorithms such as equals, add, and length. Examples include "taskId.length()" in the Task class, "getFirstName().equals("Jane");" in the ContactTest JUnit, and "contactService.add(c1);" in the ContactService JUnit. Additionally, I incorporated assertions such as "assertEquals," "assertTrue," and "assertThrows" to validate outcomes.

Given that programming is not my area of expertise, I adopted multiple strategies to ensure the efficiency of the code. I engaged with numerous tutorials, utilized the provided resources, and addressed coding errors incrementally through trial and error. I ensured that all variables were declared prior to initialization. In the TaskService JUnit, the statement "if (!alreadyPresent) { tasks.add(task); System.out.println("Contact added successfully!"); return true;" was used to verify whether the task ID was already in use before permitting its addition. In the ContactService test, I assessed both valid and invalid contact inputs. For instance, the assertion "assertEquals(false, contactService.update("22222", "Adam", "Mike", "8888888888", "8 More Rd. San Diego, CA 88888"));" was employed to examine outcomes for invalid update inputs.

\*\*Reflection on Software Testing Techniques\*\*

The software testing techniques I employed throughout the project milestones fall within the categories of black box and white box testing, as they are fundamentally specification-based and structure-based. According to Knovel, black box testing techniques derive test cases directly from the specifications or models that define the system's expected behavior. These techniques include equivalence partitioning to assess both valid and invalid inputs, decision tables to evaluate conditions and actions, state transition testing to analyze events that alter the system state or generate outputs, use case testing derived from test scenarios, and boundary value analysis to examine the limits of inputs.

In contrast, structure-based testing was prominently utilized in the coverage tests to dissect components and analyze conditional statements such as "if-then" statements. This approach facilitates the segmentation of tests into specific sections for thorough examination. Structure-based testing techniques encompass statement coverage, path coverage, and branch coverage, all of which are designed to explore the architecture of the system or components at various levels.

I did not utilize experience-based testing techniques for the milestones. These techniques leverage the users' and testers' prior experiences to identify critical areas of the system that require testing and to exercise these areas in accordance with expected usage patterns and potential misuse, which are often the sites of errors. Specific experience-based techniques include error guessing, where previous knowledge informs the selection of tests for code evaluation, and exploratory testing, which serves to assess areas that lack comprehensive specifications. Due to my limited experience with testing, I opted to omit these techniques.

The aforementioned techniques include black box, white box, and experience-based approaches. Black box techniques are appropriate when code segments have clearly defined functionalities and are commonly applied in outsourced testing scenarios. Conversely, white box techniques are applicable when there is a profound understanding of the product's desired outcomes. Experience-based techniques are crucial for identifying specialized tests that may not be easily captured by more formal methods, particularly when specifications are inadequate or when time constraints are severe. Each of these techniques is employed based on their practical applications and implications within various software development projects and contexts.

The mindset I adopted while working on this project was one of analysis, experimentation, and a focus on growth. I approached the testing process with caution, engaging in extensive research and practical trials, including the review of tutorial materials and the implementation of numerous iterative tests. It is essential to comprehend the complexity and interrelationships of the code, as these elements significantly influence the quality and performance of the final product. For instance, my initial coverage tests yielded a low coverage percentage. However, after augmenting the number and scope of tests, the coverage percentage exceeded the 80% threshold requirement, thereby enhancing the technical integrity of the code and ensuring a high-quality final product.

I made a concerted effort to mitigate bias in my code review process by conducting multiple tests across various scenarios, regardless of my expectations regarding their outcomes. I aimed to form hypotheses rather than assumptions, recognizing the potential pitfalls of the latter. This approach highlighted the risk of bias, particularly in situations where I was responsible for testing my own code. For example, had I solely tested the function responsible for checking the length of a first name while neglecting to evaluate the ID function, I might have overlooked the failure of the ID validation due to an omitted line of code. Testing a range of inputs, both valid and invalid, further contributed to the reduction of bias.

Finally, I recognize the importance of disciplined commitment to quality within the field of software engineering. Such a commitment upholds and advances the integrity and reputation of the profession. It is imperative to avoid shortcuts that could compromise the quality and performance of the final product.