Hanna Parham

Professor Phillips

CS-320

22 December 2024

**Project Two**

During Project One, my unit testing approach was methodical and thorough, focusing on satisfying the requirements outlined in the rubric. For each feature, I carefully analyzed the criteria and implemented tests aligned with the given specifications. After writing the tests, I added detailed comments to my code to explain the purpose of each section, ensuring clarity and demonstrating how the requirements were met. For example, I verified the task deletion functionality through the deleteTask(String taskID) method, ensuring that tasks could only be deleted if their corresponding taskID existed thus meeting the criteria of: “The task service shall be able to delete tasks per task ID.”

By thoroughly testing every function and edge case, I ensured that all of the necessary functions in my application were well tested. This comprehensive approach provided confidence in the reliability of my application’s behavior while also making sure that each and every requirement was considered, met, and properly executed. My experience writing JUnit tests required a meticulous attention to detail, especially when it came to defining variables and methods to eliminate any potential execution errors. Initially, I encountered challenges with importing the JUnit package into my project, but after researching and adjusting my configuration, I resolved the issue which turned out to be not so hard after all. Once the setup was complete the process became more streamlined, enabling me to develop a system for testing efficiently with each part of Project One.

To ensure my code was technically sound, I adhered to the ebay practices and rigorously validated each feature. For instance, the testAddContactWithExistingID test case verified that attempting to add a contact with an existing contactID would throw an exception. This test ensured that duplicate contact IDs could not be added, preventing data conflicts. The use of a HashMap for storing contact information further enhanced efficiency by allowing quick retrieval and management while maintaining simplicity in the implementation.

Code efficiency was another key focus throughout the project. I achieved this by incorporating encapsulation, reusing code wherever possible, and implementing coding callbacks to streamline the process. Encapsulation helped centralize operations within the TaskService class, reducing redundancy and improving maintainability while modular methods such as deleteTask and addTask were designed for reusability, minimizing duplication, and making any further updates or auditions more straightforward. By incorporating callbacks in appropriate scenarios, I further simplified the implementation with the intention to keep the code both clean and efficient.

Overall, my approach to unit testing and code development ensured that the mobile application met the customer’s requirements while maintaining high quality and adhering to best practices. This experience highlighted the importance of rigorous testing, detailed documentation, and efficient coding techniques in delivering reliable software solutions.

In this project, I employed JUnit testing to validate individual components, ensuring robust functionality through test automation and assertions. For example, the testAddContactWithExistingID test ensured duplicate contact IDs were not allowed, maintaining data integrity. Encapsulation improved testability and efficiency by organizing task and contact management into modular service classes with controlled interactions. Error handling and edge case testing, such as verifying inputs and duplicates, further ensured reliability.

Other techniques not used include UI testing, which validates visual elements and user interactions, and Agile practices like user stories, sprints, and Scrum. These are useful for iterative, collaborative projects requiring constant updates or user feedback but were beyond the scope of this project.

JUnit testing is ideal for projects requiring precision and reliability, such as backend systems or APIs as it allows developers to isolate and test individual methods, quickly identifying issues and improving code quality. Its integration with CI/CD pipelines ensures rapid feedback during development, streamlining the debugging process. Encapsulation helps maintain and secure complex projects by organizing functionality in well-defined classes, reducing redundancy, simplifying updates, and ensuring controlled data access. UI testing, although not used in this project, is crucial for user-facing applications to validate visual design and interactive elements, ensuring a seamless user experience.

Agile practices like user stories, sprints, and Scrum are highly effective for dynamic, collaborative projects with evolving requirements. User stories help align features with user goals, sprints deliver iterative progress, and Scrum fosters team collaboration and adaptability, making these approaches essential for managing complex projects with multiple stakeholders or dependencies. Employing these techniques strategically allows developers to address diverse project needs, improve quality, and ensure successful delivery.

Working on this project required both a cautious and detail oriented mindset. I made sure to double check my work, compared my code to other sections and past projects, and conducted thorough research to ensure accuracy. By carefully cross referencing code and requirements, it allowed me to identify potential issues and maintain consistency. As noted by Wang et al. (2022), "Mature test automation is key for achieving software quality at speed." For instance, while implementing the testAddContactWithExistingID functionality, I ensured duplicate entries were handled correctly across all scenarios. I made sure to test the entire project to avoid conflicts between modules like the Contact and Task management systems.

To limit bias, I reviewed code critically and aligned it with established best practices and requirements rather than personal preferences. Bias is a significant concern when testing one’s own code, as one’s own familiarity may lead to overlooked errors due to personal habits and crutches. As Katalon (2023) notes, "Cognitive biases in software testing can cause testers to overlook important flaws, leading to higher defect rates and missed issues. To mitigate this, I approached the testing process as if I were reviewing someone else’s code, carefully questioning each assumption and verifying all functionality in a checklist-like manner. This objective perspective helped ensure thorough testing and prevented complacency, ultimately leading to more reliable results.

Being disciplined in your commitment to quality is essential for long term success in software engineering. Cutting corners in writing or testing code may provide short term relief but in the end can cause significant issues down the line such as bugs or scalability problems. It is vital to view a project as a whole to ensure everything functions together, not just in individual parts. To avoid technical debt, it’s important to write clean, maintainable code and be sure to conduct consistent testing early on. Regularly recycling code and using automated tools like JUnit helps catch issues early, ensuring a stable and sustainable product for the future.

**References**

Team, K. (2024, September 17). Cognitive biases in software testing: A guide to overcome. Katalon AI-augmented Test Automation Platform. https://katalon.com/resources-center/blog/cognitive-biases-in-software-testing

Wang, Y., Mäntylä, M. V., Liu, Z., Markkula, J., & Raulamo-jurvanen, P. (2022, February 18). Improving test automation maturity: A multivocal literature review. arXiv.org. https://arxiv.org/abs/2202.09076