SOUTHERN NEW HAMPSHIRE UNIVERSITY

Summary and Reflections Report

Submitted to Prof. Kraya,

in partial fulfillment of the requirements for the completion of

CS-320

Software Test Automation & QA

by

Anthony Lee Hancock

December 14, 2023

I tested the Task class meticulously, ensuring the generation of unique task IDs, adherence to task name and description length constraints, and overall functionality. Test cases were thoughtfully designed to evaluate expected results based on Task class restrictions, instantiated with specific inputs. For example, assertions like assertNotEquals(task1.getTaskID(), task2.getTaskID(), "Test Failed: Task IDs are not unique"); validated the uniqueness of task IDs. Similarly, in the Contact class, I applied constraints related to contact IDs, first and last names, phone numbers, and addresses to validate the testing approach. Test scenarios involved creating contact instances and confirming that restrictions, such as contact ID length or phone number format, were correctly applied. The statement assertNotNull(contact.getContactID(), "Test Failed: contactID is null"); is an illustrative example. For the Appointment class, the testing strategy aimed at ensuring proper scheduling and representation of appointments. Test cases included creating Appointment instances and validating scheduling constraints. Assertions like assertEquals("2023-12-20 14:30", appointment.getFormattedDateTime(), "Test Failed: Incorrect date and time representation"); checked the accuracy of date and time representation.

With regards to following software requirements, I believe that my unit testing methods closely aligned with software requirements. This can be seen by looking at assertions like assertNotEquals(task1.getTaskID(), task2.getTaskID(), "Test Failed: Task IDs are not unique");. Furthermore, the Task class underwent extensive testing to confirm the uniqueness requirement for task IDs. Similar validation occurred in the Contact class, where predefined requirements were compared against limitations on contact IDs, names, phone numbers, and addresses. The code consistently complied with criteria thanks to assertions like assertNotNull(contact.getContactID(), "Test Failed: contactID is null");. Similarly, the quality of JUnit tests was supported by the achieved coverage percentage, resulting from collective testing of Task, Contact, and Appointment classes. High coverage indicated a thorough examination of the codebase, with comprehensive test cases, including boundary cases and error handling, contributing to the effectiveness of JUnit tests. Positive outcomes validated that the code behaved as expected across diverse scenarios.

Crafting JUnit tests involved a systematic exploration of each class's functionality, identification of potential edge cases, and the creation of test cases to address them. This experience provided valuable insights into the code's behavior under different conditions, instilling confidence in the software's reliability. When it came to implementing these tests, a significant emphasis was placed on boundary testing to explore the extremes of the input space. This involved creating test cases that pushed the limits of task name and description lengths, contact details, and appointment scheduling parameters. By assessing the behavior of the software at these boundaries, the testing process aimed to uncover potential vulnerabilities or unexpected behaviors that might arise in real-world usage. Efficiency was also a focal point during this project, with data structures like ConcurrentHashMap used for managing task IDs efficiently. In TaskServiceTest, the deleteTask method strategically used the break statement (taskList.remove(counter); break;), improving overall method efficiency.

To prevent the accrual of technical debt, I used a proactive strategy to try and resolve any faults found. This included examining code areas that might need optimization or reworking in addition to correcting errors. I wasn’t able to optimize everything as much as I would have like to, so when working on future projects I would ensure that these problems are given priority in next development cycles by documenting possible technological debt—such as regions with less-than-ideal efficiency—are also kept up to date. The goal of this proactive management approach is to keep technical debt from interfering with the software's long-term maintainability.

In conclusion, although I think I did a good job on this project, there are some things that I think could have been improved. To start, I could have made a lot of the code more efficient and not as dense; I think with time and practice I’ll be able to write more efficient code. Additionally, I would like to get better with writing comments throughout my code, especially as I start to work on bigger and more complex projects that may need often referencing. I also want to place a big emphasis on my continuous learning approach and making an effort to keep up with new testing methodologies and technological developments. With the aforementioned in mind, I believe that the collective strategies I used, applied to Task, Contact, and Appointment features, result in robust, technically sound, and efficient JUnit tests, reinforcing the reliability and functionality of the implemented classes.