Devin Bashaw

4/25/2024

CS 320

Project Two Journal

In my unit testing approach for project one, my approach was aligned with the software requirements probably 80/100. I had a hard time with the boundary tests and was not able to gain 100% JUNIT coverage for all of my classes after all. Some of this was due to eclipse being generally buggy, and the fact that my laptop is awful and outdated (new laptop ordered for the new semester thank God). However, in the 80 out of 100 I managed by ensuring that each function I created in my class files was thoroughly tested, double checked, tweaked, and reviewed so that errors were resolved, and if errors persisted, at the very least I had an explanation for them. This involved testing how the code reacted to multiple different inputs in the test classes, with tests including correct inputs, length inputs, and null inputs. Ensuring my code was properly tested allowed me to pertain to the software requirements.

Ultimately, the JUNIT tests I ran, shown via my coverage report, successfully displays that my code ran effectively enough and that I did proper testing, aligning with the software requirements. To achieve this, I ensured their technical soundness by following the method of copying the class I wanted to test and then adjusting the written code piece by piece, checking for errors along the way, until I gained a run with zero errors or minimal errors. I did this by removing the class formatting and instead adding a test in the code. As an example, if a function in the class expected input, I crafted JUnit tests to cover both correct and incorrect inputs. Once I had a test for every logic branch, the cloned class transformed into my JUnit test class, and then it was ready for execution, adding efficiency. By testing every code section, I ensured a wide amount of coverage and avoided faulty tests. Additionally, I utilized a shared test instance of the class, enabling testing of change and setter functions with the same code base, which saved more space and helped to process each line.

Overall, the testing techniques I commonly used were black box testing techniques with equivalence partitioning, which ensured that only correct input types were accepted by the class objects where I specified each input and output type. Additionally, boundary value analysis was incorporated to ensure the code rejected values outside the input scope requirements during the testing phase. Use case testing was also utilized to ensure specific requirements were met, outlined in the use cases and requirements for the project. I also utilized white box testing techniques, such as statement and coverage testing to ensure comprehensive testing of all project classes, including the communication between service and object classes. Thus, using these testing techniques and the prior knowledge I had of coding with java and eclipse, I was able to align fairly well with the software requirements for this project.