Ethan Fancher

Project 2

The software had limited features, which made compliance difficult to ensure. While the quality of JUnit tests can be challenging to measure, there are ways to assess their effectiveness. For example, the coverage percentage can reveal how much code has been executed by the tests, indicating whether all necessary functionality has been tested. In this project, the code coverage is currently at 100% for methods and classes and 92% for lines of code tested. The developer's understanding of the requirements and ability to implement them also contributes to the effectiveness of the tests.

Writing tests with JUnit was straightforward, with assertion being a useful element for creating concise tests that can detect bugs or enforce requirements. It's crucial for developers to write technically sound code by accounting for all possible scenarios and avoiding programming errors. Unit tests are an effective way to achieve this, as they test small parts of the software and can be executed during development, even before code is committed to version control.

For this project, unit testing was the primary testing technique employed, as it focused on testing individual units of code to ensure they performed as expected. Although other techniques such as integration testing, system testing, and acceptance testing could be employed in more complex projects, unit testing was sufficient for this project's scope.

I adopted a Test-Driven Development approach for this project, where I analyzed each user requirement for each module to extract necessary test cases. After compiling a list of test cases for a specific function, I would write the function and ensure that all cases passed. Input validation was a specific software testing technique I used extensively to check the input values for the modules before instantiating objects to prevent the entry of bad values. To automate testing and catch any errors after changing a certain function, I utilized @Test annotations for JUnit testing. Integration tests, system tests, and blackbox tests were not used for this project. All tests were whitebox, and the codebase was written alongside the test cases.

In accordance with the ISTQB-BCS Certified Tester Foundation readings, boundary value analysis involves testing at the input boundaries, while equivalence partitioning involves dividing the input set into different partitions of testable data. Integration testing entails integrating individual modules/functions with other modules/functions to ensure compatibility in the system, while system testing involves validating the software works, and all specifications are met in terms of the whole physical computer hardware and software system.

JUnit testing can be adopted in any incremental development environment, such as GitHub or source control, and its uses involve catching bugs early by testing early. Regression testing is also beneficial in catching bugs before they appear late in the Software Development Life Cycle, where fixing them can be significantly more expensive. Automation techniques ensure these tests can run regularly to test modules, while integration testing helps ensure that each of the modules sends and inputs data correctly between each other. System testing can ensure that each physical computer, operating system, and browser is working as per the system's specifications. Any form of black box testing can complement white box testing to help test performance issues or break the code at certain input boundaries without requiring access to a codebase.

I shifted my mindset from that of a developer frantically pushing out working code to that of an overly analytical tester in working on this project. I framed each test case in terms of how I could break the functionality and tested each user requirement with high attentiveness. For each class, I methodically checked off each requirement on the list and ensured that each assertion was tested, as well as object creation and method testing. Because services used functionality from the respective object classes, attention to creating object data to be used for the services was necessary in the test cases.

While I coded these services, I acknowledge that bias can play a role in testing these functions. However, to achieve good bug-free software, I overcame that bias by over-testing. Although some of my object's code coverage is <80%, I still attempt to test the most important targeted requirement cases and functionality. Nonetheless, bias is always a consideration, and this is why test teams are necessary in larger environments.

To become a stronger software engineering professional, I appreciate the importance of not cutting corners in testing the codebase from day one, even during the requirements gathering stage. For example, communicating with the source of the requirements, whether customers or stakeholders, can clear up any ambiguity in requirements from the start. Catching bugs later on in the lifecycle becomes significantly more expensive, and having a Test-Driven mindset with the discipline to test every edge case and function can result in more bug-free code and potentially a lot less work in the end.