Joey Paul E. Haynes

CS 320 – Software Testing, Automation, and Quality Assurance

December 7th, 2022

**Summary**

In reflection of my work in the milestone activities, I believe I aligned my testing approach with the software requirements to a reasonably high extent by designing the tests to be relevant to all the requirements, individually and cumulatively, and by ensuring adequate test coverage. For instance, I tested requirements for each field in the contact class individually for their specific length and nullity requirements, as well as uniqueness in the case of the ID field. Likewise, all functions in the Contact Service class were tested individually to ensure that adds, deletes, and updates were performed according to requirements. This approach also extends to the task service, where the requirements of each field and task function were tested individually.

The JUnit tests specifically appear to be effective on the basis of test coverage because the overall coverage percentage is high. For example, the Contact and Contact Service tests provide coverage of 88.6% or greater on the target classes. Likewise, the Task and Task Service tests provide over 90% coverage on the Task and Task Service classes, respectively. This high test coverage indicates that a large majority of the source code is tested, and can therefore provide confidence in the overall quality of the tests.

I ensured that my code was technically sound by designing each test to focus only on one particular requirement and that the tests are able to function independently. For instance, lines 14-17 in ContactServiceTest.java ensure that an object is created, updated, and compared without relying on other tests to be run first. I followed similar practices on line 24 in TaskServiceTest.java, where I ensure that a key exists before confirming that its absence in the database after the delete method is called actually proves a deletion occurred.

In addition to technical requirements, I ensured that the code was efficient by isolating tests and providing static access to the Contact Service and Task Service databases. For example, lines 9, 15, 17, 24, and 26 in TaskServiceTest.java all use static access to the Task Service database without requiring the instantiation of new objects or temporary databases. Although I did not implement SetUp methods in my tests, static access of the class fields and methods was equally effective and efficient for testing each requirement and provided greater isolation of the functionality of each test.

**Reflection**

Across all of the milestones, I employed equivalence partitioning, boundary value analysis, and use case testing techniques. Equivalence partitioning simplifies the testing process by dividing the possible input values into similar partitions (Hambling et al., 2019, p. 102). Boundary value analysis tests edge cases in the program by testing extreme ends of possible input values (p. 105). Finally, use case testing specifies the program’s functionality and identifies test cases that are relevant to how the software will be used (p. 113-114). These techniques worked synergistically in the milestone projects and ensured that all requirements were verified and validated.

Software testing techniques that I did not use for this project include state transition testing and decision table testing. State transition testing is a technique where we consider a change of state, that is, behaviors in which outputs are triggered by changes in inputs, as modeled by a state transition diagram (Hambling et al., 2019, p. 109). Decision table testing instead examines how the system reacts to various input combinations (p. 106-107). These testing techniques are especially useful for complex systems and would likely be helpful if the milestone projects were to be deployed on a larger scale.

Each software testing technique described above has its own set of practical uses for different projects and situations. Equivalence partitioning can make helpful inferences to reduce redundancy in testing and increase focus on the requirements. Boundary value analysis ensures that software responds as expected to boundaries and edge cases, which are areas where the software is most likely to fail. Use case testing emulates scenarios of how the system will be used to ensure that it functions correctly, is accepted, and meets business needs. State transition testing is especially useful when various system behaviors or transitions need to be tested. Finally, decision table testing can model more complex system behavior and logical relationships. Overall, Hambling et al. (2019) note that the choice of which test techniques to use comes down to, among other factors, the type of system under test, regulatory standards, customer requirements, and risk level (p. 100).

My overall mindset in testing this project was to employ a high degree of caution and critical analysis to ensure that requirements were verified and validated. For example, I wrote all requirements down before designing tests and ensured that the tests can catch different failure points. Furthermore, the cumulative testing strategy across multiple files ensured that test coverage was high overall. Taking note of the relationships within and between different files and system components, in my experience, helped in the design of effective and versatile tests.

I attempted to limit bias in my review of the code by writing down requirements and critically analyzing how to design test cases that account for different failure points. For instance, I designed the tests based on the requirements, rather than on what cases would be easiest to write. I attempted to think from the user’s perspective of potential ways the code could be broken and remedy them. Overall, bias could be a very pertinent concern as a software developer; because bias can lead to errors and defects being missed, ensuring that development and testing are done objectively can increase the chance of product success and a reliable reputation.

Overall, it is important to keep a commitment to quality and not cut corners in the development or testing process because software failure can have consequences at an individual and societal level. For an individual user, software failure can result in program crashes, critical data loss, or inconvenient user experience. Software failure can have massive financial costs, too; problems in the manufacturing of the Airbus A380 aircraft, for example, resulted in direct or indirect costs of approximately 6.1 billion US dollars (Hambling et al., 2019, p. 109). These examples illustrate the importance of keeping a commitment to software quality. Going forward, I plan to avoid technical debt in this field of work by critically analyzing problems, collaborating with my peers, and maintaining a commitment to lifelong learning.

**References**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing:*

*An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.