**PROJECT 2: REFLECTION & SUMMARY**

**Section I: Summary**

When writing any sort of automated testing, the intent is typically to ensure that the program’s output aligns with what is expected regarding the software requirements. In order to validate that the program meets (or even exceeds) the clients laid out requirements, edge cases and expected failures must be thoroughly tested along each guideline provided. I set out to do this in the ways I will be describing in the following paragraphs.

Starting out, I decided to implement tests for the Task, Appointment, and Contact classes. In order to ensure my software was technically sound, I set out to write extensive test cases to cover each pass, fail, and edge case I could imagine. First, I began implementing tests on the constructor by designing 2 tests per class attribute (see example in ContactTest.java lines 46-65). These tests consisted of instantiating an object in which a given variable would be tested along the requirements listed by attempting to pass either a null value or one that exceeded the specified maximum length (or specific length in the case of phone numbers). An additional test was implemented in which all object parameters were valid in order to ensure that the object was instantiating properly. Second, I decided to implement a single standard test for all getter methods for each non-service class I can verify that the function is returning the correct value as intended(see example in ContactTest.java lines 11 & 14-17). Third, I opted to implement 3 tests for each setter method closely aligned with the aforementioned constructor tests. Lastly, I added a single test for the objects toString method in which I simply compared a test objects output string with a hard-coded expected output(see example in ContactTest.java lines 39-41). For the non-service classes, I aimed to align as close to the software requirements via thorough testing of each expected pass and fail state as I have described above.

When moving to the TaskService, AppointmentService, and ContactService classes, I aimed to follow a similar testing methodology. The main difference when dealing with these tests was ensuring that invalid objects were not able to somehow slip through the previous classes’ tests in addition to testing the service classes’ tests. In order to efficiently implement these tests, each of the service-class functions were implemented to return Boolean values rather than void as the non-service classes had. This meant that each method would only need two tests apiece as they were simply pass/fail. I managed to make use of the efficiency of the Boolean return values by using assert methods for each test based on whether they were expected to pass or fail (see AppointmentServiceTest lines 20, 31, 36). While implementing these tests proved far simpler than the ones previously implemented, I believe I demonstrated proficiency here through efficient design. At the resolution of this project, I ended up with an 88% test coverage which, while I believe is adequate, shows that my work has much room for improvement.

**Section II: Reflection**

Through each milestone and at the culmination of the project, the techniques I employed would be commonly deemed VERY basic Black Box testing. Black Box testing is typically defined by its lack of information regarding the internal operations of a program or system.

RedStone Software, a leader in image-based automation and testing by their own description, states that the aim of the Black Box methodology is to “[look] at what are the available inputs for an application and what the expected outputs are that should result from each input.” (RedStone, 2008) Commonly stated advantages of this methodology include its ease of use in both implementation and execution, the short test case development period, and its overall simplicity. On the other hand, Black Box testing has several drawbacks in that it requires consistent script maintenance, has a high level of fragility as the code base evolves, and lacks the capability of fully testing the software due to its lack of introspection. This form of testing is very common in some form in almost every project, but is typically designed, implemented and maintained by dedicated software testers as the ease of integration is much greater than most other forms of testing.

I also employed some level of basic White Box testing, especially in the non-service classes. Again referring to RedStone’s piece on Black and White Box testing, the company states that the White Box testing “looks under the covers and into the subsystem of an application. Whereas black-box testing concerns itself exclusively with the inputs and outputs of an application, white-box testing enables you to see what is happening inside the application.” (RedStone, 2008) These tests are much more effective in getting a more complete image of the software’s functionality through its elevated introspection capabilities. Similar to Black Box tests this methodology struggles in its fragility during product evolution; however, White Box testing can also prove difficult due to its greater level of complexity and the difficulty that often comes with integrating the testing with the actual software. These tests are highly useful as they allow for great insight into potentially hidden errors that may cascade as development continues and areas that may be optimizable by the developer(s). White Box tests are generally designed and implemented by the actual development team as they require a greater understanding of the code base and are more complex to integrate.

Most of the techniques that I opted not to implement are reliant on a greater level of experience on the part of the developer. An example of such a methodology is error guessing. In the process of error guessing, the tester employs their understanding of both the code base and common areas of failure to determine (or rather guess) what sorts of tests may be relevant to the project. Another example of an experience-reliant testing methodology is exploratory testing. When the tester is designing and implementing exploratory test cases, they are essentially implementing some form of tests that examine portions of the program that definitive guidelines in the software requirements document. While both of these forms of testing are highly effective, it is likely that my lack of experience and the limited scope of the project would limit their necessity.

As a software tester, there will always be three aspects of my mindset that I will need to be mindful of: maintaining a healthy level of caution, minimizing unconscious bias, and maintaining discipline throughout the entire development cycle. In this project, I believe that I displayed an appropriate level of caution throughout development and testing as demonstrated by my error mitigation and thorough test cases. In particular, I am proud of my implementation of the service classes and in particular the error handling and method design that enabled my software to be both secure and efficient. In regards to managing bias, I feel I did an adequate job but have room to improve. Specifically, I would point to the lack of thorough testing during object instantiation in the service class methods as an example of an instance where I dropped the ball. In future work, I will need to work to further limit biases favoring my own work in order to ensure that I am not missing errors that I would have picked out of another developer’s work. Building on this sentiment, I believe that I allowed my bias to affirm my work and may have cut corners without even realizing it. Those test cases that I just mentioned could have easily led to substantial damage to either myself, my employer, or my customer if I had turned in that work for a project that handles sensitive data. In the future, I must be more disciplined in order to be able to be proud of the work I am doing and it will be at the front of my mind moving forward. Ensuring all of this will be key to maintain a firm grasp on the quality of my work and will prevent future technical debt. I will plan on improving in this area by continuing my studies on testing and the methodologies/principles that power it.

**Works Cited**

*Position Piece Black-box vs. White-box Testing: Choosing the Right Approach to Deliver Quality Applications*. (n.d.). Retrieved December 12, 2021, from https://www.cs.unh.edu/~it666/reading\_list/Defense/blackbox\_vs\_whitebox\_testing.pdf

‌