Name: Sam Brasher

Class: CS320-T4512

Assignment: Project Two

# **SUMMARY**

Given such clear requirements, I believe my testing was very well aligned. For example, each base class had fields with different input and, in one case, data type requirements. I made sure to test the boundaries of these requirements and ensured that tests were properly failed when given invalid input and succeeded when given valid input. Each of the service classes requested an in-memory database to manage a list of their respective objects. I used a list interface, and tested the functionality required of each class. Notably, the Appointment Service class did not require being able to update Appointment objects, where the Task and Contact Service class did.

Across all testing classes, I achieved 100% coverage of my code. Some instances were not quite as straightforward: consider the case of the Task Service needing to update a Task object. The update functionality is not part of the Task class, but the mutator functions lie within it. Therefore, the Task Test did not test the mutator functions, but those functions were tested by the Task Service Test. In the end, 100% coverage was achieved ensuring my code was effectively tested.

For each test I wrote, I tried to conform to the standards of JUnit testing. By using these standards and ensuring the entirety of my code base adheres to them, I can feel confident in a 100% test coverage rate. For example, I was consistent in using an InvalidArgumentException in the case of invalid input (too long, wrong character type, etc.). Because of this, I was able to check most of my tests with assertThrows while using invalid inputs. For checking successful update functions, I was able to use assertEquals to ensure that the update completed, and the new information was stored.

I believe my code was efficient because there was one test per requirement, but I also think there were opportunities to make it more efficient. There may have been unnecessary splitting of tests considering I split input requirements into a test for length and a test for null. There also might have been a chance for improvement because I noticed several components of my classes were tested more than once.

# **REFLECTION**

I employed several different testing techniques: input testing, boundary testing, state transition testing, and statement testing. Input testing, boundary testing, and state transition testing are forms of static testing known as Black Box testing (Hambling, Morgan, Samaroo, Thompson, & Williams, 2015) because we do not look at the inner workings of the program. I used input testing to verify that invalid inputs were handled correctly. For example, a null input should throw an InvalidArgument exception. Boundary testing was used to check that inputs within, at, and outside acceptable limits were all handled correctly. For example, ID fields could have no more than 10 characters. We test that an input of length 10 is accepted, and that a length of 11 throws an appropriate error. For the service classes that were responsible for adding and deleting objects from the in-memory database, I used state transition testing to ensure functionality was valid. For example, ensuring that a non-existent object cannot be deleted. Or that, provided an object exists in the database, another object cannot be added with that same ID. Finally, statement testing was used to ensure that 100% of my code was covered by the tests.

I did not use decision table testing, use case testing, or any flow chart testing techniques. Decision table testing involves creating a table of possible input conditions and crossing them with software requirements. For example, an input of 11 on the unique ID field should not be allowed. Use case testing diagrams the different ways different actors can interact with the system – we do not have multiple actors according to the requirements, so I found this to be an unnecessary testing technique. The various flow chart techniques diagram the flow of logic through the program, also being able to diagram the flow of control through various decisions.

For the techniques I did use, I believe they contribute to ensuring a program adheres to requirements. Additionally, the techniques can be used on variety of different scales and scopes. The techniques I did not use I found to add more work without providing additional benefits. I believe, in more complicated or larger scale programs, flow chart design can be of great benefit to ensuring the flow of the program is valid. Use case testing would be valid in a lot of other scenarios with different types of users. Websites are good example here, with admins having different functionality than regular users.

Caution played a big role in determining if I could consider testing “complete”. We know that testing does not prove a product defect-free, but merely reveals defects covered by the tests (7 Principles of Software Testing: Defect Clustering and Pareto Principle, 2022). I also wrote test code for each piece of functionality soon after coding, then revisited both the tests and the code later. A big part of my code was the choice to implement a list interface versus other types of in-memory databases. Proving a concept known as the Pareto Principle, most of my defects came from this system.

Bias can be a huge problem in writing your own test code, and there are many ways it can creep up (Shenk, 2018). There were a few ways I tried to eliminate bias in my coding, one of which was choosing the list interface over other options. As an example of congruence bias, I initially chose an array for my in-memory database. When I started testing the addition and deletion of objects, I repeatedly ran into problems dealing with location of objects in the array. When I reconsidered my options, a list interface was a clear better choice to me. I also came back to my test code several days in a row, considering what new tests I could add to ensure proper adherence to requirements.

Being disciplined in your coding is a cornerstone of good software development. A lack of discipline can lead to inconsistent or poor-quality code, which can lead to problems later – a concept known as technical debt. This can lead to minor to severe financial loss, or even more dire consequences. My biggest thought for avoiding technical debt, from personal experience in my career, is to clarify user requirements as early and as often as possible keeping in mind that users may not know or be able to verbalize what they want or need out of a piece of software (Hanna).

**Works Cited**

*7 Principles of Software Testing: Defect Clustering and Pareto Principle*. (2022, April 3). Retrieved from Software Testing Help: https://www.softwaretestinghelp.com/7-principles-of-software-testing/

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide.* BCS The Chartered Institute for IT.

Hanna, M. (n.d.). *Disciplined Software Testing Practices.* Retrieved from International Institute for Software Testing: https://testinginstitute.com/DST%20Free.pdf

Shenk, G. (2018, July 16). *The Impact of Cognitive Bias on Software Testing*. Retrieved from Functionize: https://www.functionize.com/blog/the-impact-of-cognitive-bias-on-software-testing