## Summary

# Alignment to Requirements

My approach to ensuring that the software meets all the requirements is by first creating the structure of the classes and then testing the classes against the requirements, I can ensure that the software meets the expected functionality. The first step of creating the class structure is important because it establishes the foundation of the software. By defining the structure of the classes, I identified the fields that are required and ensured that they are present. For example, in the task class, by including a unique id field, an object field, and a description field in the task structure, I ensured that these fields are present. Next, I created functions that validate the inputs to ensure that they fall within the acceptable range. This is an important step because it ensures that the inputs are valid and that the software will function correctly. By validating the inputs, you can prevent errors and bugs from occurring. Once the class structure and input validation functions have been created, I created JUnit tests to ensure that the software meets the requirements. JUnit tests are an essential part of software development because they allow developers to test the software in a controlled environment. By testing the software against the requirements, I identified any issues that may arise and fixed them before they become bigger problems. Overall, my approach to ensuring that the software meets all the requirements is thorough and effective. By creating a solid foundation, validating inputs, and testing the software against requirements, I was able to ensure that the software functions correctly.

# Effective Tests

The Junit test doesn’t prove that they are completely effective because there may be errors in the test them self or there are false positives or negatives. On the other hand, the coverage of the inputs was around 100% where the test checked all the code had a corresponding check. For example the setName accessor has attached test that correspond to the if the name is null, a viable name or if its two long. There is a balancing act when determining how much of the code should be covered because if you spend too much time testing superfluously can be costly but not testing enough can lead to errors.

# Technically Sound Code

To ensure the technical soundness of the code, I primarily focused on creating unit tests. I used the assertEquals function to compare input values with expected output values in most of the tests. For example, I tested whether the getID function returned the expected output value of "1" when the input was set to "1". Additionally, I used assertThrows to test the program's ability to catch errors when values were out of bounds, such as null or character limit exceeding ID values. Apart from unit tests, I also performed static analysis to review the code myself and ensure that there were no mistakes in the program. By combining unit tests and static analysis, I aimed to ensure that the code was technically sound and free of errors.

# Efficient Code

When working on a coding assignment, I realized that there were certain checks that needed to be repeated for constructors and mutators. To make my code more efficient, I decided to create a function that could handle these checks and avoid having to rewrite them separately for each function. While this was a step in the right direction, in hindsight, I realize that I could have made my code even more efficient by creating a more generalized function that could handle validation for multiple functions. For example, it would take in the lower character count and the upper and through an exception if value is not between the two. This could be done for any of the constructors and mutators. By reviewing the code that I have written and doing some checks on in I ensured efficient code.

# Reflection

# Techniques Employed

For my project, I utilized both static and dynamic testing techniques. In terms of static testing, I reviewed my own code to ensure that it met the necessary requirements and was free of errors. Additionally, I sought technical reviews from experts, such as my professor, who could provide an independent evaluation of the project's technical aspects based on the documentation (the grading of the work). During a designated inspection period, any identified issues were addressed, and formal discussions were held regarding past problems that had been fixed from the results grading.

As for dynamic testing techniques, I employed structure-based testing. This involved testing each method within the class to ensure that all units or pieces of the program functioned as expected. The methods were setters, getters, and constructors. Furthermore, the testing covered most of the possible inputs to ensure that most inputs resulted in the correct outputs. For example, in the appointment class, I tested for appointments that occurred in the past. Specification-based techniques were used for non-functional specifications only because they were the only specifications that were necessary. Overall, by utilizing a combination of static and dynamic testing techniques, I was able to thoroughly evaluate my project and ensure that it met the necessary requirements.

# Other Techniques

In my project, there were several techniques that I did not utilize. For static testing, I did not conduct peer review, which involves having classmates review the code for errors. I also did not conduct a walkthrough to explain the code's logic to stakeholders since there were no stakeholders involved in the project. In addition, I did not analyze the data flow, as the tasks were independent, and the flow did not affect the outcome of the project. The customer task and appointment classes were independent.

Regarding dynamic testing techniques, I did not employ experience-based techniques. This technique involves using prior knowledge to predict where errors may occur during testing. However, since I lacked the necessary experience to make these predictions, I did not use this method. Additionally, I did not test for functional requirements as this was not necessary according to the Specification-based techniques. Overall, despite not using some techniques, I was still able to effectively test my project and ensure compliance with the required standards.

# Uses and Implications of Techniques

In a real-life scenario, more comprehensive static testing would involve setting design specifications, conducting reviews throughout the software development lifecycle, and analyzing data flow. Peer reviews would occur after a small amount of code has been written, and walkthroughs would occur when stakeholders need to confirm logic or intent. Analysis of data flow and structure would also be necessary to gain insights into how data is collected, processed, stored, retrieved, transformed, and used by different applications and systems. For dynamic testing, gaining more experience in testing would help determine what needs to be tested and what doesn't, resulting in faster and better tests. By utilizing these techniques, the software development team can ensure high-quality software products that meet user requirements and are compliant with necessary standards.

# Caution

To be cautious, I focused on problem areas and avoided unnecessary complexity. For instance, instead of using a for loop to create a 50-character string, I used a simple sequence of characters, like "123456", to easily verify the string length and prevent mistakes. This approach prevented errors like using the 0 as the start of a for loop, which would result in a string that is 51 characters long. In addition, I prioritized my efforts by focusing on the areas that required attention, rather than testing every possible case. For example, if the validation requires a string length of 1 to 50 characters, testing for 2 to 48 characters is unnecessary. By spending more time on the edge cases, I ensured that my solution was robust and able to handle a wide range of scenarios. Lastly, I took the time to review my work thoroughly to ensure that there were no errors. This step was crucial in identifying any mistakes and making sure that my solution met the requirements. By employing these cautionary measures, I was able to develop a solution that was both efficient and accurate.

# Bias

To limit bias in my code, I followed several strategies. Firstly, I wrote the code in pieces and revisited the old code with fresh eyes to ensure that no mistakes were made. Secondly, I cross-checked the written code against the requirements to ensure that I had not deviated from the original objectives. Thirdly, I incorporated feedback from another person who reviewed the code. Fourthly, I employed good coding practices, such as using classes to ensure that the objective was consistent with normal practices. Lastly, I used Eclipse's IDE to catch any typos and class creation errors with the debugger. By following these steps, I aimed to minimize bias in my code and ensure its accuracy and reliability.

# Disciplined

Being disciplined in commitment to quality is an essential trait for a software engineering professional, as it not only ensures reliable and effective software. To ensure that the software is important not to cut corners because being lacs can lead to higher cost and efficiency. The costs can also snowball where being undisciplined at the start can cause further problems as the project continues. I plan to avoid technical debt by following the prosses, habits, and best practices that have been established. For example, using the software development life cycle. I would not just jump into any issues as they arise but instead plan out each stage of the development process. For instance, I look back at the code to see what changes need to be made to improve the code like making it more efficient by adding some more automation to the error checking by creating a validate function. By adhering to established processes, implementing best practices, and taking a disciplined approach to software engineering, I will ensure that the software I develop is reliable, efficient, and effective, without accumulating technical debt.