**Summary and Reflections Report**

**Summary**

My approach was based strongly on the software requirements. As both the code and the tests were being written, they were compared to the given software requirements to ensure they were all met. For example, each object was required to have a unique ID under 10 characters, and the field could not be null or be edited. To ensure the value would be unique and remain unchanged, a map was chosen to store the object. Input values for the ID were verified that they were not null and 10 characters or less before being added to the map. In the JUnit tests, the field was tested to ensure that null input, a value of more than 10 characters, as well as attempting to add two objects with identical ID values would all throw an exception. Once code was complete for both Java classes and JUnit tests, and they were manually inspected to confirm all software requirements were met, a coverage test was run. Each of the six Java classes involved were reported to have 100% coverage. This reflects comprehensively effective coverage of the functionality elements in the classes was captured in the JUnit tests.

Object oriented programming was adhered to consistently throughout the code. For instance, an example of encapsulation and abstraction is employing the object model classes to hold the structure for the object, and only exposing the necessary information to the service classes. For instance, when using maps to store data for object fields for which the software requirements specified a maximum length, a method was created to validate the length of input. This method was not useful outside of this class, so it was made private. Additionally, all variables being used within the classes were made private. Most methods were used in the Service classes and were made public. This is an example of inheritance. The input length validation method is also an example of efficiency. Rather than writing code to check the length of each input field, the method was written and simply called each time it was necessary.

**Reflection**

For dynamic testing strategies, I ran a coverage test and ensured that any portion of code that was not covered did not need to be directly tested. These lines resided solely in the test classes. The programs had 100% coverage in both the model and service classes. This speaks to the reliability of the code that was produced. I also performed a static analysis of my code, reviewing it to identify any data structure, data flow, or control flow issues. Eclipse also does some of this for you. For instance, in the Appointment Service assignment, I imported many date-related libraries to figure out the best way to ensure an input date was not set in the past. By the time I figured out how I wanted to implement it, I had several unused libraries that I needed to delete. I also had some variables that I ended up not using. This is an example of data flow inconsistencies. I was also informed by my instructor after submitting the Module Three assignment that using Maps, rather than Linked Lists, was a more efficient way to store my data. I adjusted future assignments accordingly. This is an example of a peer review, which is also a static testing technique.

There was no formal review of my applications, so no other static testing techniques, such as a technical review or inspection, were used. I still struggle sometimes with the most efficient way to structure my code. For instance, in the first assignment, I check the requirements for each field individually within the constructor. After being provided example code, I realized it was more technically sound to verify these aspects in a separate method that returns a Boolean, and then use this information in the constructor. This is an issue that may have been identified in a technical review. I also did not have access to someone who has significant experience in the field, so any experience-based dynamic techniques like error was not possible. In other words, I did not have someone who could evaluate my code and point out any weaknesses that may exist. These assignments are simple and consistent with the logic and requirements, so I don’t feel that I would have benefited significantly from this strategy. However, in larger or more intricate projects, this would be very helpful.

I tried to be wary of any bias that might be present by testing my own code. After being involved in the development and looking at the same lines of code for hours, it can be easy to overlook issues that are present. I relied heavily on static testing done by Eclipse by making sure no warnings or suggestions were present. I utilized example code provided by the professor and compared it with what I had written as another form of static testing to double-check my code for any conciseness errors and that testing for any functionalities were not missed. The code coverage was probably the most important part of avoiding any bias. I was confident when I saw 100% code coverage for model and service classes that I had sufficiently tested my program. However, if I had not put significant thought into the interrelationships of the code, I could have spent a significant amount of time attempting to identify and handle any portions of code that were not being tested. For example, it was imperative that each entry have a unique identification term so that entries were not updated or deleted incorrectly. This means not only testing that identical identification terms could not be used, but that the correct entry was being retrieved via that identification term.

While the repercussions of not being diligently committed to software quality can be as great as loss of life, sometimes on a massive scale, it is important even in situations that don’t carry as much risk. Not only can mistakes lead to major financial ramifications for the company responsible for the program, including additional time allocated to the project and loss of business for the company, it can also lead to personal consequences. Your integrity and trust may be lost, and you may even lose employment for professionals who continue to make these mistakes. This can be avoided by not being afraid to fail, identifying technically proficient coworkers and asking questions often, being willing to acknowledge areas that you have opportunity to grow and being accepting of constructive criticism, and not getting too comfortable in your abilities.