**Summary**

Most of my tests are aligned well with the software requirements. Specifically my tests for the Core classes of Contact, Task and Appointment are set up to test the specific input requirements of each class variable from the requirements document. Each of these classes has individual test cases covering the required input verification requirements for each class variable: No null variable can be allowed to pass in any variable, regardless of class; No String variable can exceed a specific length, such as 10 for each of the ID variables; and the ID variables must be immutable. In addition, the Date variable for a new appointment in the Appointment class must register as after the initial date value created when the program starts.

For this last issue, the initial Date value, I pre-set the date the program tests against to a date corresponding to the start of this class project. All of my remaining tests are set up to test individual null values for each function: for the Contact class this would be an individual test case for cID, firstName, lastName, contactNumber, and contactAddress. Additionally, there are individual test cases for overly long variables, resulting in the same number of tests as the null-entry case for each function. These tests satisfy the initial input requirements as specified in the program document.

For the functional requirements, such as the ContactUpdate feature of the ContactService class, I would say that more than 70% of my testing was directly related to the software requirements specified in the document. The reason for this discrepancy lies in the abstraction of how each function operates. For instance: for the in-class storage functionality, I could have created a Hash Table, a LinkedList, an ArrayList, or even an Array of Arrays. Each option varies in the complexity of its implementation, and thus will also vary in how tests must be implemented. I chose to use an ArrayList for each of my classes, both because I wanted to take advantage of the tools built into Java’s ArrayList class for item cleanup and sorting, and to keep my testing and general layout consistent across each stage of the project. This, and the necessity of testing Get and Set functionality, compose the remaining 30% of testing for the Service classes.

To test the ArrayList for each Service class, I needed to create mock data for each test case to manipulate. For the ContactService class, this meant that to check the removal function I would need to create a new ArrayList with two pre-loaded Contact objects before the test could begin. Once these two objects were instantiated, I could the perform the search function to look at each Contact object in the Arraylist for the one that contains the target cID. The function should the remove this object from the ArrayList using the built-in Remove function from the ArrayList library. To prove that the removal was successful, I then had it print the number of indices within the ArrayList and verify that there is only one entry. The test fails if any other number of objects are present in the List.

The integrity of my code is reliant on the results of these tests. Initial variable testing within the Core classes is thorough and ensures that when an object is created by the constructor, whether pre-loaded or created by the Service classes, it will only allow the input of valid strings to each variable. The testing of each function utilizing the ArrayList functions to Remove, Update and Add new Objects is also thorough, and additionally verifies that changes to existing objects will not violate the input requirements. Additionally, by separating the functionality of the Constructor and Manipulator functions into two different classes, and by integrating the ArrayList’s capability to reference entire objects, I can ensure a high level of efficiency in my code.

**Test Methods**

* Black Box:
  + Equivalence Partitioning: Covers the requirements for the Object variables in each class, testing for compliance. Does not check for the contents of the string, leaves the program open to nonsense data.
  + Boundary Value Analysis: Same as above.
  + State Transition: Covers the Update, Add and Removal functions, and ensures that updates remain in compliance with prior testing.
* White Box:
  + Statement and Decision Testing and Coverage are used in reference to the Service classes, as each Service class much abide by the requirements laid out by the Core Object class it references.
* Experience Based:
  + Error Guessing: I encountered specific issues with the Date class, and in find the correct method to test the ArrayList. I used print statements within the error codes to gauge where the communication issue between the Service and Object classes existed. I also attempted three methods of testing the ArrayList before I settled on the current design.
  + Checklist: Before writing the code for the test cases, I spent time creating a list of items that needed testing. During the coding process, I referenced this list for both the Object and Service class testing to ensure that I had adequately covered each potential case. I did have to add to this list as I went, however.

**Reflection**

Overall, I tried to approach the testing process with ample attention to detail and care for potential edge cases. While I tried to avoid the tendency for truly exhaustive testing, the programs created during this project were small enough that exhaustive testing was possible. I committed to extensive research towards differing methods of solving the problems presented in the requirements document, and asked for feedback regarding my code after reaching an impasse. While I ultimately did not choose to act on the feedback to alter my storage technique to utilize a Hash Table, I made this decision due primarily to a greater level of experience with ArrayLists. I do recognize that this is a bias I need to resolve, but it is one that can be resolved with further learning and experience.

I experienced technical debt during this project as the deadline closed in. Having a checklist certainly helped in keeping track of each task, and ensuring that individual cases were not ignored or forgotten. In the future however, I plan to incorporate more tools to manage this debt such as Microsoft ToDo and Trello. Both tools are easily accessible and comprehensive, with reminders that I have found to work in other areas of my life and are easy to adopt for development purposes. Cutting corners should never be an option, and these tools along with experience will help ensure it remains so.

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.