**CS320 Project Two**

1. **Summary**

My testing approach directly aligned with the designated software requirements for the product. The testing addressed creating contacts, removing contacts, updating contacts, and ensuring that each field abided by the proper parameters. When checking the addition or removal of contacts, I would employ two different methods within the test: search of ID and check of list size. First, I would search the Contact List for the added/removed ID and depending on if it was found, I would throw a test failure. Secondly, I would capture the Contact List’s size before adding/removing a contact and then measure it after the method was enacted. Both methods allowed me to verify that the correct actions were being taken within the code.

I know that for the contact service tests, my testing code did not have a high coverage percentage. However, that can be attributed to me creating helper methods or code to assist in the ease and functionality of the code which was not required. Thus, when the testing code was developed to only test the required elements, some pieces of code were left out. However, for the task tests, my code covered around 86.5% of the code between the two classes. This was accomplished because I did not create additional code or method to help me during the development process.

To ensure that my code was technically sound I added multiple different task objects into the list and would test deleting which ones from the list. From doing this, I discovered that the objects I created and added to the list from previous testing methods still existed within the list. Therefore, to ensure that each test was conducted in a controllable environment, the list was cleared at the start of each testing method. This can be observed in lines 28, 45, and 59 in the TaskServiceTest class.

A great example of how I made my code efficient can be seen within the updateTask method in the TaskService class. Since the requirements expressed that two fields must be updatable (name and description) and instead of creating two methods with essentially the same code, I added a parameter into the method. The parameter is a Boolean which controls whether the new value inputted updates the name or description field. With a simple if statement, I can check the value of the Boolean and update the appropriate field. If this logic was not added, there would be two methods instead of one with the same basic code to perform the same function but on different fields.

1. **Reflection**

For each of the milestone I employed dynamic testing techniques to analyze and ensure that the product was achieving the desired objectives. Specifically, I utilized the System under Test (SUT) concept when planning and developing my testing. For each of the basic classes (Contact, task, appointment), I created a unit test to ensure that the construction of those classes occurred within the given parameters. For all of the service classes for the previous units, I developed integration testing units that ensured that they units worked together and performed the required functions.

I did not use any static software testing techniques within my milestones. Static software testing requires that the code is analyzed and tested without the execution of the code. However, for each of the milestone tests, the code was executed to fulfill the testing objectives. If I had to implement a static testing technique in my milestones, I would implement the OWASP dependency checker which analyzes the code and the dependencies for any known vulnerabilities and provides their criticality ranking.

Unit testing ensures that each of the basic classes are performing correctly and ensures that the rest of the application has a solid foundation to build off. If unit tests are poorly or not implemented, then the basic classes could be the reason for error messages further down in the code or other tests. The time and effort needed to backtrack the error to the basic class costs money and could result in the reconstruction of the basic class and its dependent classes. Integration testing ensures that all the units are interacting together according to the program requirements. A lapse in the development of integration testing could result in values being overwritten in basic classes as the service possess too much access. Additionally, the service classes could outright fail to fulfill the specified objectives.

Throughout the semester, I employed caution in a reasonable manner during all of my development assignments. While caution is a great trait and concept to have on the top of your mind while creating software, too much can paralyze development (and developer) in an unattainable pursuit of perfection; thus, I developed with reasonable expectations of caution on my mind. To develop with caution, I guaranteed that I would test all potential tests cases to verify that the proper error message was occurring. After creating one test, I would run it and ensure that all the correct outcomes occurred rather than waiting to run all the tests at one time. It is important to appreciate how complex interactions that occur within the code will pave the way for more effective testing units. A lack of appreciation might lead to basic tests that do not fully capture the complexity and interactions that occur across the code. By appreciating the complexity within the code, I created two different update test methods within the AppointmentServiceTest class as the two variables employed different data types.

For the assignments in this course, I was both the developer and tester which is an easier way for bias to sneak in and corrupt the final product. In order the curb the amount of bias that was built into the final product, I developed and tested the product with strict guidance from the given requirements. By ensuring that the code and test aligned with the requirements, I limited any form of bias that could have been introduced. My commitment to fight bias in my code is exemplified through testing the prescribed attribute length in all the different classes.

Being disciplined is one of the most important characteristics of a software engineering professional. Exhibiting discipline can be observed through genuinely progressing through each of the testing phases and making a sincere effort to develop congruent testing units. Without this discipline, ineffective tests will be the final stage before product release, resulting in damaged business reputation and customer satisfaction from subpar products.