**Describe your unit testing approach for each of the three features.**

The requirements that were provided for this assignment directly align with the tests provided in this assignment submission. All of the test cases were derived from a specific software requirement. An example includes the several test cases that check if an exception is thrown when a null value is presented as an argument to any of the string parameters of Contact.java and Task.java. Examples include testPhoneNotNull(), testLastNameNotNull(), and testTaskIDNotNull(). These test cases were directly derived from the requirement to prevent the input of null into the Contact.java method. Similar test cases involving management of a null argument are present in AppointmentTest.java and TaskTest.java. Another example is the ability to create objects from the Contact class with input arguments that are the correct length. In ContactTest.java’s testContact() method, the length a typical case and the length of a border case are checked to ensure only valid lengths are allowed. Similar test methods are included in AppointmentTest.java and TaskTest.java. There are many other test cases that could be created involving performance and other aspects of each class, however, it is important to focus on testing the requirements given.

The test coverage percentage revealed that all of the relevant sections (100%) of the classes being tested had been exercised. This metric ensures that regression tests done on the code will check all of the code relevant to the given requirements. This metric does not ensure that every possible issue is covered, however, it does ensure that regression tests will exercise the entire code base. In combination with relevant equivalency testing/boundary testing, the test coverage demonstrates that each portion of the code is tested for all scenarios relevant to requirements. When new code is added to these classes, these tests can be used to ensure the standard of quality is held to the entire code base.

**Describe your experience writing the JUnit tests.**

When creating the majority of the code, I used automation from the Eclipse IDE to write most of the object oriented code. This process helps reduce the number of potential bugs the software needs to deal with and ensure the code is technically sound. A specific example includes the getter/setter methods inside the Contact.java/Task.java’s classes. Automation reduces that amount of potential error involved when a human writes the same codel. Another way I ensured the code was technically sound was doing regression tests for each individual class. This allows each issue to be visible at its source. An example is the construction of a Contact object through the ContactServices’s addContact() method and the constructor in the Contact class. If there is an issue creating a Contact object, the regression test methods testAddContact() and testContact() check where the issue is located. This structure for regression tests allows for a streamline debugging process and ensures errant bugs do not go unnoticed.

Instead of exhaustively testing areas that have input equivalency, the software tested a few normal test cases and boundary cases. This method allows all of the locations in the code that congregate issues to be tested without creating endless test cases. An example of this includes the testContactIDTooLong() test case which checks if an exception is thrown if the inputted contactID was too long. This checks the typical case and the border case so that all relevant locations are checked without creating excessive test cases. There are no non-functional requirements for the code since it is a simple class that holds string data at this point.

**Testing Techniques**

The software testing techniques that I am employed for each milestone typically included equivalency tests and border testing. Both of these strategies fall under the black box testing techniques. Throughout all three of the milestones, I used equivalency testing to determine what lengths of input strings are considered out of bounds and within bounds. An example of this would be the length of an ID being less than or equal to 10 characters. Anything within this bound is considered valid and anything outside this bound is considered invalid and leads to and an IllegalArgumentException. Considering this example, I also used border testing to check if values close to 10 behaved correctly. Another example of these concepts is the testAppointmentDateTooEarly() method inside the AppointmentTest.java file.The method checks a typical date that is too early for input as an appointment and a border case is tested where the date is a minute away from being considered valid. Border testing and equivalence testing are used extensively in the code.

There are other black box testing concepts that I did not use including decision tables, state diagrams, and use case diagrams. These methods are best used when trying to understand all the different possibilities for user interaction with the system. The decision table uses a row of combinations that can lead to specific outcomes. State diagrams apply to programs that change between different ‘states’, which leads to different outcomes from user interaction. Use case diagrams are useful for testing possible interactions with the system. Other testing techniques included white box testing. A form of white box testing focuses on dividing the program into units that can be tested independently and checked for proper interface functionality between units. There are a host of other testing technique s that could have been used. The requirements for this project simply needed equivalency and border testing to ensure code quality.

The tests that I created using techniques such as equivalence testing and border testing, can be used for regression checking while adding new code to the existing project. This also applies to other techniques that I didn't use for this project. Specifically, unit testing is useful for ensuring the system works during regression checks. Techniques that include the use case, state diagram, and decision table help develop test cases out of existing requirements. These types of techniques allow developers to develop an understanding of the different scenarios that are possible defined under requirements. These techniques help create structure for further testing.

**Mindset**

For this project, I had to employ caution when deciding the amount of tests I wanted to create. I could have easily started creating numerous different tests for issues that I felt were important, but were not necessarily a part of the requirements. If I were creating tests without a requirements document, I would start thinking about all of the ways I could ‘break’ the code. This is not helpful when trying to finish the project in a timely manner. An example of my use of caution includes limiting the number of tests I do within an equivalency range. Another important aspect of working with modular code is thinking about testing in the context of debugging. Tests should be written in a way where finding issues with the code should be as easy as possible. Writing separate tests for the Appointment.java and AppointmentService.java classes allows issues to be traced back to the appropriate file.

I think it is tough to limit bias on code when the individual who writes the code also tests the code. The main concept that limits my bias is focusing on the requirements given to me. Examples of writing code specifically for testing requirements is shown all of the most of the test cases. A direct association can be drawn between each test case and a given requirement. An example is the testAppointmentIDTooLong() method inside the AppointmenTest.java class. The requirement expects arguments for an appointmentID outside the length of 10 to be invalid. The test case demonstrates this fact about the code. If I were personally working on a large project, I would separate the testers from the people who write the code.

The value of regression tests and planning the direction of your code is immeasurable. Writing code that is bug free is good, however, writing code that knows how to deal with new bugs is better. Having the foresight to plan for issues in newer iterations of code allows new code to be written better and allows it to be more accessible for more developers. Writing proper tests for code now, reduces the amount of technical debt that will accumulate. I plan to work on small modules of a project to be able to write code with excellent coverage. As a project progresses, I will take notes on common issues and write regression tests for those issues in new code. I am already dealing with this issue with a video game engine I am writing in Java. As the program becomes more complex, the need for decent regression testing becomes increasingly apparent.