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

By approaching the wish list items laid out for each feature, I believe I created the software encompassing all the software requirements. Looking at each class contained within the software, all requirements have been achieved.

First, in the contact class, the requirements were to create a contact object that shall require a unique ID string no longer than 10 characters, cannot be null, and non-updatable. Other requirements in this class were ensuring the contact object included a first name, last name, phone, and address all of which cannot be null. Characters included are no longer than 10, except for which address can not be more than 30. The contact service feature can add, delete, and update these contacts with a unique ID. Each item is checked to be within requirements before being recorded into the system:

*if (contactID.length() <= 10 && contactID != null) {*

*if (firstName.length() <= 10 && firstName != null) {*

*if (lastName.length() <= 10 && lastName != null) {*

*if (phone.length() <= 10 && phone != null) {*

*if (address.length() <= 30 && address != null) {*

Secondly, the task class holds a task object having a unique task ID, a name, and a description. All of which shall not be null and hold a maximum of 10, 20, and 50 characters long, respectively. Within the task service, this is to be able to add, delete, and update via the unique task ID associated with them. Each item is checked to be within requirements before being recorded into the system, such as:

*if (name != null && name.length() <= 20) {*

*if (description != null && description.length() <= 50) {*

Thirdly, the appointment class holds an appointment object with a unique appointment ID that can be no longer than 10 characters, not null, and non-updatable. This object also requires an appointment date field that cannot be in the past nor null and incorporates a required description string field, no longer than 50 characters, and not null.

Each of these three classes, their methods, and inputs have been tested with Junit5 version code, each with one input within and one input beyond the boundaries set. I know my Junit5 tests were effective because of high coverage percentages that were captured amongst these areas of code that were shown to be conclusive.

* 1. Describe your experience writing the JUnit tests.

I ensured my code was technically sound by providing each test one path to probe, both one for within and one beyond the boundaries set to gauge how the code reacted to each input:

*@Test*

*void testValidAppointmentId() {*

*Date date = new Date();*

*Appointment appointment = new Appointment("1", date, "AppointmentDescription");*

*String appointmentId = appointment.getAppointmentId();*

*assertEquals("1", appointmentId);*

*}*

*@Test*

*void testInvalidAppointmentId() {*

*Date date = new Date();*

*assertThrows(IllegalArgumentException.class, () -> {*

*new Appointment(null, date, "AppointmentDescription");*

*}*

*}*

Efficiency within the code was achieved using getters and setters keeping the code lines clear and concise and as minimally cluttered as possible:

*public contact(String contactID, String firstName, String lastName, String phone, String address) {*

*if (contactID.length() <= 10 && contactID != null) {*

*this.contactID = contactID;*

*}*

*else {*

*throw new IllegalArgumentException("Invalid ID");*

*}*

*setFirstName(firstName);*

*setLastName(lastName);*

*setPhone(phone);*

*setAddress(address);*

*}*

*public String getFirstName() {*

*return firstName;*

*}*

1. Reflection
   1. Testing Techniques

Some of the software testing techniques employed within the Junit5 testing were analyzing the boundaries of input to ensure the system would only record expected values, positive and negative testing to ensure the system behaves as expected when inputs are within and beyond parameters, and exception handling to ensure the system handles exceptions and errors correctly.

A few techniques that were not used in testing include security testing for vulnerabilities within the code making sure the system is resistant to various security threats. Nor was load testing carried out to ensure the system operates as intended under peak usage.

All the described testing techniques have extremely practical uses. Boundary testing input ensures that systems will not take in anything other than it has to to function. Taking input without boundaries is a huge security flaw and hazardous for a program. Positive and negative testing also goes hand-in-hand with this to identify the system is operating as intended. Exception handling helps the system from crashing and is used as a security net to avoid security issues from various attacks. Security testing all around buttons up systems from all known attacks and creates a robust and healthy starting point for anything new that undoubtedly shows up using something like Maven. Finally, load testing is incredibly valuable because, though a system may operate as intended through low usage, the system could be clogged when at high usage due to memory issues or unidentified bugs that can trip up a system under high volumes.

* 1. Mindset

The mindset that I adopted working on this project started out as less invasive. This, I am thinking, is because it can be hard to test your own product to the limit, as this can show deficiencies in one’s own work. I guess you can say that I proceeded with a little too much caution at first. In the later works and going back to make changes, the caution was lowered substantially to envelop the testing to each individual item. It was very important to appreciate the complexity and interrelationships of the code because testing completely requires one to understand what the code is trying to do. With this, the output can be expected. In the beginning I accompanied the code with testing that sought only the right input but failed to test further to see input outside the parameters. Later, I went back and wrote more tests to cover each happy path and only one per test.

The way I tried to limit my bias when reviewing my code was simply to remove myself from the title of author, only making sure to connect testing to each piece of code. I can see on the software developer side that review my own code could come with biases, but I think I can be fair with myself and do an all-encompassing job as well, but I understand that not everyone can. It is simply easier, when reviewing code you written, to test small as many people like to feel accomplished rather than pushing to find the few areas where errors my exist.

As with anything you produce, I think your character shines through any product. Therefore, being disciplined with commitment to quality is highly important as this ensures more things are looked at knowing you want to put forth a good product. When cutting corners while writing code, this can lead to deficiencies and unexpected behaviors within the system. Depending on what you are writing for, this could impacted reputations and even impact lives with an untold reach. I plan to avoid technical debt by testing and seeking out a “proof-reader” to double check when able. Testing is truly the only way to know what a system is capable of before deploying and it is critical to treat it as such.