1. **Summary**
   1. Describe your unit testing approach for each of the three features.
      1. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.

**For each of the requirements I implemented both a positive and negative test. For instance, the Contact Service has a requirement that the contact ID must be no longer than 10 characters. So, I first implemented a positive test where I test a valid string (less than 10 characters) and ensured it completes without any errors.**

**@Test**

**public void testAddContact() {**

**assertTrue(testContactService.contactMap.containsKey("goodID"));**

**}**

**Then I implemented a negative test, where I intentionally create an ID with more than 10 characters, and ensure the correct error code is thrown.**

**@Test**

**void testBadIdTooLong() {**

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

**new Contact("badidtoolong", "goodfirst", "goodlast", "1234567890", "Good Address");**

**});**

**}**

**I did this for each of the requirements to ensure 100% coverage with both positive and negative testing.**

* + 1. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage?

**My tests were effective, they covered 100% of the lines of my code because I accounted for all possibilities in if statements. For instance see the following code:**

**public void deleteTask(String taskID) {**

**if (!taskMap.containsKey(taskID)) {**

**throw new IllegalArgumentException("Task not found");**

**}**

**taskMap.remove(taskID);**

**}**

**To cover all the lines of code in the above function, I needed to write a test that meets the contents of the IF statement, and another that doesn’t.**

**The first test runs the red code and verifies that deleting a non-existent task throws the Illegal Argument Exception:**

**@Test**

**void testDeleteNonExistent() {**

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

**testTaskService.deleteTask("NonExistentID");**

**});**

**}**

**The second test runs the green code and verifies that the taskMap.remove(taskID) works:**

**@Test**

**public void testDeleteTask() {**

**testTaskService.deleteTask("goodID");**

**assertTrue(!testTaskService.taskMap.containsKey("goodID"));**

**}**

* 1. Describe your experience writing the JUnit tests.
     1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.

**I used static testing to verify that I did not have any syntax errors in my code and used the IDE’s built-in code checker to verify that there were no red lines under any of my code. I also ensured I used the correct logic when adhering to the requirements. For example, the task service specifies that a taskID “cannot be longer than 10 characters.” So I had to be careful to choose the correct comparison operator when constructing my if statement.**

**if(taskID == null || taskID.length() > 10) {**

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

**}**

**You can see I chose “ > 10 “ and not “ >= 10 “ to throw the Invalid Task ID error, because the verbiage of the requirements implies that 10 characters is a valid length.**

* + 1. How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate.

**I ensured my code is efficient by removing unnecessary operations and choosing efficient data structures.**

**For instance, in ContactService.java I had the following line that checks for null values when setting first name:**

**if (firstName != null) searchContact.setFirstName(firstName);**

**However this was redundant because in Contact.java, the set method also checks for null:**

**public void setFirstName(String firstName) {**

**if (firstName == null…**

**So, in TaskService.java, I removed the if(name != null) and just implemented:**

**searchTask.setTaskName(taskName);**

**I also ensured efficiency by using a Map (HashMap in specific) to store the tasks, which averages O(1) time complexity for adding, removing, and retrieving. If I had used a list here, any of those operations would require looping through the entire list and checking for matching would take large amounts of time.**

**Additionally, if I were to spend additional time refactoring**

**this code, there are more ways I could increase efficiency. For example, all of the setters and constructors in each of the \*Service.java files checks for similar requirements and some of the code is redundant. I have several methods where I check for something like: if (taskName == null || taskName.length() > 20)   
And then later I check: if (taskDescription == null || taskDescription.length() > 50) {**

**These could be abstracted into a helper function where I run validateRequirements(taskDescription.length(), 50) for instance where I input a string and a length and return a bool**

1. **Reflection**
   1. Testing Techniques
      1. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details.

**Regression Testing - This is a type of testing where the developer tests the code after making changes. It’s important to make sure you didn’t break anything when going in and making changes.**

**I use this technique naturally when coding, perhaps I adopted the trait by learning from my mistakes. I used it in this project many times, Once when I realized that my variable names were not consistent across the three services. For instance, some of the function names for Contact.java were generic, like SetID, while function names for Task.java were more specific, like SetTaskID. This confused me while coding when I tried to implement SetID on a task object and it didn’t work. I ended up going in and rewriting all methods and variable names to match the same level of detail and specificity. After doing so, I needed to do some regression testing to make sure that I didn’t break all my code and, spoiler alert, I did, and needed to make fixes.**

**Positive Testing - This is a type of testing where you make sure that the program is working as you intend it to. For instance, the setTaskName function updates a non-null task name that is less than 20 characters. So to test that I inputted a non-null TaskName that’s less than 20 characters, and then ensured that it executed without error. Then separately verified that the Task Object was updated correctly.**

**Negative Testing - Negative testing is the opposite of positive testing, in that you’re testing to ensure your programming does not malfunction when *unexpected* actions occur. For instance, Contact.java has a phoneNumber attribute that is required to be 10 digits. You would expect the user to type in a 10 character all numeric phone number, but that does not always happen, so I implemented a check in the constructor and updatePhoneNumber method to ensure that the phone number is 10 digits and all numeric, otherwise the program would crash.**

**Boundary Value Analysis - When you use if statements to validate inputs, you may sometimes inherently create boundaries. For instance when checking lastName.length() > 10 the following boundaries are created:**

**(0 to 10) : valid input**

**[10] : valid input**

**(10 to ∞) invalid input**

**To participate in boundary value analysis, you would create tests for each of those ranges and validate that the expected behavior occurs.**

**State transition Testing - This testing technique is used to ensure that things in the code still work after a change takes place in the code, like for instance, does the home button on a webpage still work after the user has logged in. I used this in my project in my addTask, addContact, and addAppointment methods. When a task, contact, or appointment is added, the state of the program changes because now an object is stored in memory, and the chance of a duplicate object is possible.**

**Black box testing - Black box testing is a type of testing where the internal structure of the code is not known. This is typically the perspective of the user, where they just want things to work, and don’t care about what’s going on with the backend. I tried to take the mindset of the user by implementing input validation on phone numbers to ensure that ONLY numbers are inputted, avoiding issues like users entering the letter O instead of the number zero.**

* + 1. What are the **other software testing techniques** that you did not use

**Decision Table Based Testing - In decision table testing a table is created where conditions and actions are displayed alongside each other to create an easy to read, visual representation of how things might play out, making it easier to make a decision.**

**Flow charts - Flow charts are another type of visual representation testing where steps in the process are shown and logically represented with trees and branches.**

**Statement Testing - Statement testing is where you ensure that every single statement, or line of code, is run and tested in the program. I believe I may have done this inadvertently and achieved 100% code run, but I did not explicitly set out to track that every line of code was run and tested.**

**Stress Testing - Stress testing puts the system under heavy load and ensures that things don’t break under high usage. Since we did not have any central infrastructure in this project, it was outside the scope and irrelevant to do this type of testing.**

**Security Testing - Software security testing checks for vulnerabilities or security weaknesses that may exist in the code. Again, we did not have any infrastructure to test on and the application with not complex, so things like cross site scripting, privilege escalation, or brute force attacks did not need to be addressed. However, the input validation I implemented can be considered a form of security testing because I prevented users from being able to intentionally crash the program.**

* + 1. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.

**Regression Testing - This is very useful and should be implemented often because any time you make changes or add features to code, it’s possible that other things break. It’s also important to do this frequently especially if you are making lots of changes. You don’t want to go and make 100 changes at once without testing, and then all of a sudden users can no longer log in, because then you’d have 100 things to go through individually and it would take a long time.**

**Positive Testing - This type of testing is arguably one of the most important because it ensures things are working *as intended*. The intentions of the app are what the users actually want. If a customer wants an app that can find hot dog shops in the area. It should be tested and ensured that hot dog shops in the area are listed, otherwise the app is useless.**

**Negative Testing - Negative testing might be slightly less important than positive testing in the grand scheme of things, it should definitely not ever be skipped. Negative testing is responsible for ensuring your apps are not a buggy mess with unintentional actions happening all over the place.**

**Boundary Value Analysis - This is important to ensure your code is robust and accounts for specific cases and ensures programs do not fail.**

**State transition Testing. - This is important because in the real world, many applications are dynamic, constantly updating and changing based on user input. If a program works initially, the programmer might think they’ve done their job, but changes like login states should not impact the functionality of the app.**

**Black box testing** - This is important because of the gap in knowledge between the developer that created an app, and anyone else potentially using or modifying the app. Say a developer creates a secure app that requires you to jump up and down three times and recite the national anthem before you can attempt to log in. The developer will know to do this because they are the ones that created it, but to an uninformed user or even an uninformed developer, the process makes no sense and they’d never be able to log in.

**Decision Table Based Testing - This is useful when dealing with very complex functions or requirements and it’s hard to visualize what might need to be done. Taking the time to display actions and reactions in front of you can make it easier to determine how to implement a certain feature in software instead of leaving the complexity to whirl around in your head.**

**Flow charts - Doing testing while following a flow chart is another great way to visually keep track of how a program should be behaving and minimize the risk of losing track of complex processes.**

**Statement Testing - It’s important to create tests of *all*** **your code so a user doesn’t potentially uncover an issue that you forgot to test and fix.**

**Stress Testing - In the real world, successful applications get a lot of traffic, and infrastructure and code may not be able to handle when the load gets heavy. For instance, a particular backend list may have only been created to handle 100 user objects in it, but if more than 100 users are using the app at once, the program would crash.**

**Security Testing - Security flaws can potentially crash your program, release private data, or ruin your reputation, so it’s important to test accordingly.**

* 1. Mindset
     1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

**I was very cautious in my programming and made sure to not make too many changes at once or try to reinvent the wheel. I searched online numerous times to see if other people had solutions for issues I was coming across and I made sure I fully understood something before implementing it. I also reduced complexity by reusing code I had already written. Code for Contact.java was very similar to the code needed for Task.java and Appointment.java, so I was able to reuse most of the code. However, after copying and pasting I needed to double check and make sure that I didn’t leave any variable or method names from Contact.java in the other two files.**

* + 1. Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

**It’s important to consider bias because of the wide range of people that could potentially use your application. Not every will have the exact background knowledge of the app or even computers in general to be able to succeed in using your app. I attempted to keep my code simple and well-commented to account for potential new developers in the project, and kept all user interactions simple and input-validated.**

* + 1. Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

**Discipline is important because in order for testing to be effective, it needs to be thorough. The amount of testing that can and should be done on a system increases exponentially as the complexity of the system increases. So, for a fairly complex app, creating a testing plan and actually following through with it will uncover lots of bugs and avoid issues down the road. The alternative to discipline, being sloppy and cutting corners, leads to bugs and unhappy customers.**