My testing approach was completely aligned to the software requirements. For example, in task service, for the task class, requirements included the task object requiring a unique task id string that is 10 or less characters and that the task id cannot be null or not updatable. Accordingly, my code included the variable *private final String taskId;* where *final* causes the *taskId* variable to not be updatable and *string* making the variable a string. Additionally, the code *if (taskId == null || taskId.length() > 10) {throw new IllegalArgumentException("Invalid task ID");* stops the code if the *taskId* is missing using *taskId == null* or (||) the *taskId* is longer than 10 characters using *taskId.length() > 10).* Along with this, appropriate Junit tests were conducted to test that the exceptions were being thrown using *assertThrows(IllegalArgumentException.class, () -> new Task(null, "Name", "Description"));* and *assertThrows(IllegalArgumentException.class, () -> new Task("12345678901", "Name", "Description"));* where it purposefully uses a null in place of the *taskId* and then a string longer than 10 characters in place of the *taskId* to check if the *IllegalArgumentException* is thrown. Similar to this, other requirements for the Task class and TaskService class have been addressed.

As mentioned previously, JUnit tests were conducted for the various requirements put forward in the prompts for Task Service and Contact Service by using the exact input that is to be rejected so that an *IllegalArgumentException* is thrown. After the JUnit test was configured, Contact Service had a coverage percent of 83.8% while Task Service had a coverage percent of 79.1%. When looking at both of these coverages for the specific test cases, all were majorly green and passing. Based on these coverage percentages, I know my JUnit tests were effective because they covered all the necessary parts and each of those parts successfully threw the *IllegalArgumentException* for the intended wrong input.

I ensured my code was technically sound by organizing code in a way that is easy for anyone else to quickly understand. For example, I kept white spaces between lines of code and grouped together pieces of code doing the same functions. A direct example seen is in TaskTest.java where each of the tests are ordered in the same way as their respective variables are ordered in the other classes. Task Id validation (line 14) being first, followed by name validation (line 20) and description validation (line 26). Within these, the two tests for null and length are also grouped together (lines 15, 16 for *taskId*, lines 21, 23 for *name*, and lines 27, 28 for *description*).

I ensured my code is efficient by incorporating the four principles of Encapsulation, Abstraction, Inheritance, and Polymorphism as well as using coding best practices such as input validation. An example of this is data validation seen in Task.java lines 7 to 14 where if any of the variables are null or longer than the maximum assigned length, an exception is thrown and the code stops.

Software testing techniques that I employed for each of the milestones was unit testing with JUnit. Unit testing involves validating individual classes in the program and in this case, for the Contact, Task, and Appointment classes as well as their associated service classes. Specifically, *IllegalArgumentException*s were thrown when invalid input was received. Consequently, through the JUnit testing, assertions were used to prove that the code was indeed capturing this when incorrect or invalid input was received for the various components. This was also supported by both positive testing where the correct expected input was used and negative testing where the exact opposite of what was expected was used to confirm that the correct data was expected and wrong data was rejected. Going into more detail, in the module 3 milestone, contact object testing included creating a contact object using *public class contact* and including the variables *contactID, firstName, lastName, phone, and address*. Each of these variables were coded to reject any invalid input such as *contactId, firstName,* and/or *lastName* having more than 10 characters, *phone* having exactly 10 characters, or *address* having more than 30 characters. If the input data failed for any of the restrictions, *throw new IllegalArgumentException("Invalid");* was used to end the code and stop wastage of time in running the other code. When the JUnit test was run, *assertThrows(IllegalArgumentException.class, () -> new Contact("", "", "", "", ""));* in the ContactTest class was used to check that when invalid input was used, the contact class was capturing this wrong input and correctly throwing an exception. This method was used for all of the other variables in the contact class.

For the contact service testing, code involved adding a single contact and then multiple contacts successfully, handling duplicate contact IDs (checking with contacts list and throwing exception if new matches with an old entry), adding a contact and retrieving it, updating a contact, and deleting a contact. To test these, the ContactServiceTest class used *@BeforeEach* to start the test from scratch without previous data. This was followed by testing each of the actions involved such as adding, retrieving, and updating the variables for an entry. A direct example is ContactServiceTest.java lines 37 - 42 where a new contact is created (using *Contact contact = new Contact("3", "Java", "Joe", "1234567890", "130 Main St");* and *service.addContact(contact);*), the first name is updated (using *service.updateFirstName("3", "Python");*), and then this new contact is retrieved again and validated for the new value entered for the first name (Using *assertEquals("Python", service.getContact("3").getFirstName());*).

In the module 4 and 5 milestones, similar techniques were used. The main difference being that in the module 5 milestone, in addition to adding, retrieving, deleting, and updating data and validating length and constraints, the date had to also be verified using *before(new*

Date()). For all of these milestones, coverage tests were also run to check whether the unit testing was covering the important parts. This does not verify that each of the tests successfully detect errors, it only validates that it is being tested in some form.

Other software testing techniques not used for the milestones are integration testing, system testing, regression testing, performance testing, and exploratory testing. Integration testing focuses on verifying interactions between multiple components, which wasn't applicable since each of the milestones were tested in isolation individually. System testing evaluates the entire application against requirements, but there was no user interface or full system requested in the original prompts. Regression testing wasn't needed since it involves ensuring that changes aren't breaking the current code usually in very large projects instead of small ones like this. Performance testing measures behavior under heavy load, and exploratory testing involves informal, unscripted test cases which both aren't necessary for small, in-memory data services such as in the contact, task, and appointment service milestones. These techniques are more appropriate in later development stages or larger scale projects.

Practical uses and implications of unit testing include application in early stage development to ensure that individual modules/classes function correctly before integration. It is mostly used in modular, test-driven, or agile development projects where continuous feedback is used such as in the milestones. Integration testing is more useful and valuable with projects that involve separate components such as APIs or databases that are connected and required to communicate information correctly. System testing and regression testing are best used before deployment to make sure that a full application meets requirements and remains stable after updates, especially if the project is very large. Performance testing is ideal for high-traffic systems such as e-commerce or banking apps. Finally, exploratory testing is best for quickly learning and exploring a new software system and uncovering errors found along the way. Since the milestones are not very large and do not have much of an interface, this type of testing is unnecessary.

My mindset that I adopted when working on this project was making good quality code that is secure and especially works its intended purpose. In acting as a software tester, I amply employed caution by checking even small parts of the code that seem to be fine without errors because I knew that even small errors can be problematic and they aren't always easily visible. It is important to appreciate the complexity and interrelationships of the code being tested as it challenges the tester in finding errors and secures the code from potential hacks. Specific examples are the coverage test where although the coverage test was all green and hundred percent, when looked at deeper and further, it was realized that some functions were barely being tested.

To limit bias while reviewing code, I took the stance of a third person who has not programmed the code to better find errors. On the software developer side, bias would be a concern for checking own code because all the details about the code are known and so the same amount of focused attention would be absent in comparison to a tester who knows nothing about what is in the next corner. In my case, when coding, I know what I was coding and what errors I might make. When reviewing the code, I sometimes ignored checking thoroughly the parts of code I was confident in during coding.

When it comes to writing or testing code, it is important to not cut corners because small mistakes can have disastrous and even irreversible effects. I plan to avoid technical debt as a practitioner in the field by always reminding myself that the effort required for thorough coding and testing is significantly less than the amount of effort that would be required in the case of fixing mistakes after the code/system is released. Specific examples in practice include making sure that at least 80% of code is being covered by the Junit tests during coding itself. Doing this would mean that all the tests done would be worthwhile instead of realizing later that all the tests only verified very little of all the significant and important code.