In this project we worked on various classes and testing classes as a whole. One of the first that I worked on was the contact service class and its testing class. In my testing approach for these class I focused on testing the individual functionalities of the code. I did this by testing adding, updating, deleting, and retrieving contacts. Some examples of this are when I tested adding a new contact with a unique ID to make sure that the system stored the ID correctly. This was done in the testAddContact test. Another example would be for updating. testUpdateContact was in charge of that. It validated the update functionality by modifying contact fields like firstName, lastName, and phone. It verified each of the constraints of the fields. For adding , and deleting, I focused more on exception tests. I used assertThrows to validate error handling, and to test some of the edge cases such as attempting to add duplicate IDs or deleting non existent contacts. I also used tests like testAddContact, and testDeleteContact to test adding a contact, and deleting a contact.

Task service had a primary focus on validating the addition, update, and deletion of tasks. Things like adding tasks with unique IDs via were tested via my testAddTask method. This made sure that the service stored valid tasks while rejecting duplicates, which was also tested with the testAddTaskWithDuplicateId method. Task service and task were tested similarly as the contact and contact service. I also used exception tests to make sure that invalid operations, like updating the task were handled appropriately.

For appointment service the approach was to make sure that the constraints maintained integrity. I tested adding appointments through the testAddAppointment method. Through this method I validated that the dates were not in the past and that descriptions met the length requirements. Deleting appointments was testing via the tesDeleteAppointment, which testing that appointments were deleted. Like the other classes I also handled edge cases like adding duplicate IDs or invalid dates with exception tests.

My testing approach for each of the classes was aligned with the software requirements. Some evidence and examples I could provide are some that I listed above, but also some of these. One that comes to mind is the validation of a unique contactId. The requirement was that the contactId must be unique, not null, and up to 10 characters. In this example I used the addContact method to check for duplicate IDs and it would throw an exception if the ID already existed.

You can see that here:

if (contacts.containsKey(contact.getContactId())) {

throw new IllegalArgumentException("Contact ID already exists");

}

Next I tested using testAddContact and testContactInvalidId methods. They looked like this:

@Test  
 // Tests that a contact can be added successfully and retrieved by its ID.  
void testAddContact() {  
 ContactService service = new ContactService();  
 Contact contact = new Contact("12345", "John", "Doe", "1234567890", "123 Elm Street");  
 service.addContact(contact);  
 *assertEquals*(contact, service.getContact("12345"));  
}

And

@Test  
 // Tests that an exception is thrown if the contact ID is null or invalid.  
void testContactInvalidId() {  
 Exception exception = *assertThrows*(IllegalArgumentException.class, () -> {  
 new Contact(null, "John", "Doe", "1234567890", "123 Elm St");  
 });  
 *assertEquals*("Invalid contact ID", exception.getMessage());  
}

This adhered to the requirements because these methods ensured that duplicate Ids are not permitted, and that if the contact id is null or invalid it would not be excepted. Another example that I aligned with the software requirements was via testing the field constraints. One of the requirements of the contact class and many of the other classes like task and appointment was to ensure that the id length met the length requirements, and that the name would not be null or exceed a certain length, and that the description met certain length requirements. This is something that appointment, contact, and task all had in common. To meet those requirements I wrote tests that would test those requirements such as testContactInvalidId, or testContactUpdateMethods.

When talking about the overall quality of the Junit tests, my tests demonstrated high coverage, edge case validation, and error handling. In regards to coverage, all of the tests passed indicating that there is high coverage. For edge cases I checked for exact maximum lengths and invalid operations like updating non existent records or checking if things are null or not. Examples of this are sprinkled through the code classes. Some examples of this were adding duplicate tasks :

assertThrows(IllegalArgumentException.class, () -> {

service.addTask(new Task("123", "Task 1", "Description 1"));

service.addTask(new Task("123", "Task 2", "Description 2"));

});

Also deleting non existent appointments:

assertThrows(IllegalArgumentException.class, () -> {

service.deleteAppointment("99999");

});

My experience writing these unit tests were pretty straightforward. I have had experience writing all sorts of tests because of the software development bootcamp that I am in. But writing these tests have proven to be valuable practice. I was able to think critically about the different services and how they behaved under expected and unexpected conditions. By viewing the classes like this I was able to write my tests a little easier. When writing the tests I had to make sure that all of the functional requirements , validated constraints, and anticipated potential edge cases were met. This required planning, and frequently running the tests to confirm that they actually worked. I made sure that the code was technically sound by writing tests for all core functionalities, validating constraints and edge cases, and using assertions to verify correctness. One good example of showing this is my testContactInvalidId method:

@Test  
 // Tests that an exception is thrown if the contact ID is null or invalid.  
void testContactInvalidId() {  
 Exception exception = *assertThrows*(IllegalArgumentException.class, () -> {  
 new Contact(null, "John", "Doe", "1234567890", "123 Elm St");  
 });  
 *assertEquals*("Invalid contact ID", exception.getMessage());  
}

}

This code confirms that the contactId adheares to the constraints of being non null and <= 10 characters.

For efficient code I focused on reusing common setups. I initialized objects like Contact, Task, and Appointment in test methods. Overall I mainly looked to avoid redundancy if possible. I think in this project I was able to avoid redundant code while maintaining clarity.

Some examples of this could be located in all of the classes, but one specific example of this is in the ContactServiceTest. Here I reused setup code for adding and retrieving a contqact. Here is the code:

@Test  
 // Tests that a contact can be added successfully and retrieved by its ID.  
void testAddContact() {  
 ContactService service = new ContactService();  
 Contact contact = new Contact("12345", "John", "Doe", "1234567890", "123 Elm Street");  
 service.addContact(contact);  
 *assertEquals*(contact, service.getContact("12345"));  
}

It is efficient because I am using the same Contact setup, and I avoid redundant code for initializing contacts.

In this project I mainly focused on using unit testing as the main mode of software testing within this project. By doing this I was able to really test our individual components of the code. This isolation made it easier to ensure that the program was working as expected. I like to look at it like you are breaking a problem down into smaller chunks. By doing so it makes it easier to tackle a big project or a bigger problem. By using Junit I was able to create tests that validated specific methods and classes. Some that come to mind are the add Contact, updateContact, and deleteContact in the Contact Service class. These tests really targeted specific functionality with that single class or method. This allowed me to identify issues at their source. An example of this was writing tests to confirm that invalid phone numbers or null values for firstName in the Contact class would throw exceptions, this ensured the input validation logic worked as required by the project guidelines. In my experience with unit testing, it is really good at snuffing out logic errors early on in the development process. I like this mode of testing too, because you can make sure each specific part is running smoothly before integrating the program with other components.

Another testing method I used was boundary testing. I did this to test edge cases. This involved testing the system with inputs at or near the limits of its constraints. An example of this would be with tasks. I tested whether the taskId in the Task class would reject values longer than 10 characters and ensured that the appointmentDate in the Appointment class wouldn’t not be set to a past date. The boundary tests helped a lot in identifying that these constraints and validation rules were implemented appropriately to all relevant fields.

While mainly I focused on unit testing, and a little boundary testing, there were a lot of techniques that I did not practice in this project. I did not use techniques like integration testing, system testing, or user acceptance testing. Integration testing involves identifying how different components of the application interact with each other. In the case of this project, an example of how this could be implemented would be if the ContactService had been integrated with a database or a front end interface. From here integration tests would have been necessary to make sure that there was communication between the layers. Another technique that I did not use was systems testing. In contrast with integration testing, systems testing examines the application as a whole, and makes sure that all the units and components are working together to meet the requirements of the program. UAT on the other hand, involves end users testing the application to ensure that it meets the requirements in a real world situation.

With these techniques in mind, they also have practical applications. Unit testing is ideal for early stage developments, where verifying if the individual methods and classes are working properly. This is the stage of the application we are in with this project. Integration testing becomes important when multiple things withing the application need to interact. Like I said above, this is when we would be adding a database or something to the program that would need to be able to communicate with each other. System testing makes sue that the entire application is meeting the functional and non functional requirements, which is very important for large and complex systems. Lastly UAT provides valuable feedback from the user actually using the application. This feedback is crucial because this is how the application will be used by many people. While unit testing is great for this project at this stage, if this project were to scale, and expand in terms of functionality, we would employe these kinds of tests to make sure that the program is meeting the requirements.

Wile working on this programing project, I was able to take on a mindset of thoroughness and caution as a developer and a software tester. I came to this project with the expectation that mistakes could exist. I think being cautions helped me slow down, and think about the potential edge cases and focus on properly validating the functionality of the code. For example in the ContactService class, I tested not only the happy path, such a successfully adding a contact, but also the edge cases, like attempting to add a duplicate contact or passing a null value of r a required field. By slowing down and moving cautiously I was able to make sure the system behaved predictably under varying circumstances. Another example would be the updateContact, where changes to one filed had to respect constraints without inadvertently breaking others. For instance, updating a contacts phone number to an invalid value had to correctly trigger and exception without affecting other fields.

When coding there is inevitable bias. To limit this I made a effort to step back and evaluate the implementation from the perspective of an external user or tester. I deliberately crafted tests for scenarios I didn’t initially consider. An example of this was in the Appointment class. I initially assumed users would always provide valid future dates. However, I wrote tests for invalid dates, including past dates and null values, to confirm the robustness of the validation logic. Bias is definitely a concern when testing your own code, because it can make it easy to overlook errors unconsciously. Especially when you think you fully understand how the code should behave. To also combat this bias, I went over the project requirements many times and compared them against the code and test cases to make sure they were actually aligned. Another thing that is required I think to overcome bias is discipline and commitment to quality in your code.

I think with this you can limit the amount of significant issues down the line. By going through the testing, and not skipping tests, and rigorously looking through your code, to make sure that it is working properly, and is being tested properly is all time worth spending, to save yourself time later down the line. For example if I had not tested one of my classes, lets just say the TaskService class, and a bug allowing duplicates might not have surfaced until the application was integrated with a data base or another component. This could have caused integrity issues in the program, and cause a bigger problem later down the line that I would have had to go back in and fix. This avoids technical debt. I plan to continue writing comprehensive unit tests, leveraging tools like code coverage analysis to ensue through testing, and regularly refactoring code to maintain its clarity and efficiency. By adhering to these practices, I can ensue the software I develop is maintainable, and reliable in real world environments.