Over the course of this class, we explored different software testing techniques and different libraries for testing, specifically in the Java language. In my files I imported JUnit testing libraries to test the different methods I implemented for the software program. The software consists of different methods like Contact and ContactService with their respective testing class names for a system that takes data and interacts with different data structures in the system, like HashMaps, and enters different information into these structures like contacts, appointments, and tasks. The system is being tested by validating the functionality of the code using JUnit testing with a goal of 80% coverage. By testing the units and edge cases that would happen in different scenarios, we can make sure we release a system that has minimum bugs and the desired functional outcome.

We validated data with our imported JUnit testing library to test the data from different angles testing invalid entries, valid entries, and different scenarios that could alter the data in different ways. By ensuring we have the proper testing coverage, we can ensure the method has the required functionality that the system needs. The requirements for the system were to be about 80% coverage of all our classes and methods inside of our program. This means the data and logic in our program needs to be all accounted for when constructing out tests. If we don’t cover parts of our code and make changes that reflect on that code, it can lead to unwanted changes and buggy code. Releasing a buggy codebase is asking for issues and will come back later in the form of technical debt.

We started with the Contact and ContactService class, which included data like first and last names, phones, and addresses. When reading through the requirements I concluded that we needed conditional statements when creating the objects so that these fields wouldn’t exceed certain numbers and letters, to trigger exception responses as well as prevent bad and duplicate data from being entered into the system. We also excluded the addition of some mutator methods for the final id that cannot be changed as for the requirements of the system. When making these objects, we made sure the proper error message was shown when the user entered data that was invalid. Making sure our error message was detailed really helps the user identify issues as to what they are doing wrong or what could be wrong with the system.

Furthermore, in the test cases the assertThrows and assertEquals method to check the Contact objects attribute’s stored in the exception variable, in which we stores the value of the data. The assert method is basically a validation checker, think of it as a checker for the test. If the condition is true then the test passes, and if the condition is false the test fails. This is a key concept for understanding the logic of your unit tests and how to implement the logic depending on the requirement for the code we are testing. This also helped in testing the software because I was the one who had constructed the codebase, so I already had a good understanding of what the system needed to be tested for and which edge cases and unit tests to cover.

The Task and Task service classes also had similar requirements and features. We validated certain lengths of the object’s attributes. For example, we added an exception conditional statement, like from before when initializing the exception variable for testing, except this time it was the actual conditional part of the code instead of a variable exception value. This tests the description by adding too long of a string and throws an exception if the length is exceeded. At the end we also validated the error message. This allowed us to have extensive coverage of the method, even checking for accurate exception conditional statements that were being thrown.

Moving onto the Appointment and Appointment service, this also had a similar approach structurally, as well as similar requirements in the object’s attributes field. The requirement that stuck out the most was utilizing a library to handle the date attributes. The required method before(newDate()) was implemented in the conditional when creating the setDate setter. We also had to utilize the CurrentTimeMillIs() method to capture the current time. I had began to research the library and the necessary methods I would need to implement in the program to meet the requirements and to also utilize that required method in the prompt. We then continued to implement the service testing classes which instantiated the Appointment object and tested adding and deleting appointments from the data structure from which we utilized.

The testing of the software was implemented in alignment of the system requirements. The tests that were designed tested the data from all angles, ensuring the data testing had full coverage. When creating the test classes, I thought of the different possibilities for the way the user would be interacting with the system. These interactions would be practical like adding contacts to a list, users adding duplicate contacts, and users trying to enter invalid data into different fields. By testing all possible scenarios, we gain more testing coverage in our system and can ensure our logic will be free of any major and moderate bugs.

Methodical approaches were taken by not only testing different invalid inputs, but also edge cases like overly long String values and null values. From my experience, you need a solid understanding of the codebase before we can implement testing for it. Fortunately, I am testing the same exact code that I built the logic for, so I have a comprehensive understanding of how my system is functioning and how the data is flowing in the system. For example, when designing the service classes, I implemented a data structure appropriate for storing the data, so I had to acknowledge that any changes made to the nonservice file would directly affect the code in the service file as well. The structure I utilized was a HashMap. This selection was appropriate due to the unique id value the attributes had for the object. This served as the key for the value of the HashMap. This way we had a key for the structure that was unique and immutable.

Moving forward, playing the role of the software tester means I must exercise a deal of caution. This included taking into consideration the various scenarios that could happen in the system. Our codebase was connected, for example the Contact and ContactService files were interdependent on one another. This means that sometimes certain implementation details can be different, and the logic of the system must be structured accordingly. In this scenario I also had to make sure I not only covered invalid creation of contacts, but also when characters were too long or null as specified in the conditional statement lines.

When we test code, we also need to take external factors like the person doing the system testing into consideration. Things like bias also pose a risk to the system. For example, when someone reviews their own code, they tend to overlook things. This is why having a peer review of your code is imperative, as they can help see things that you did not notice. During this project, I fully developed the backend of the application, as well as the corresponding unit tests and files for them, so I had to rely on ensuring I had the appropriate amount of testing coverage throughout the system. Since JUnit is automated, we can ensure its findings and readings are accurate and can also help us cover and test code that we might have overlooked or not tested properly.

As a developer, my commitment to quality is a top priority. I understand that proper and thorough testing of your system is an essential part of the development process, and for good reason. By ensuring our testing is properly implemented, we can help avoid technical debt in the future. This means that we can avoid future bugs and even emergency maintenance sessions in the future by squashing the bugs now. I have seen this plenty of times especially when companies must take over a codebase that consists of many bugs. They must specifically hire a prod/QA developer to help the frontend and backend tackle their bugs because the system was not developed and tested properly.

Moving forward, now we understand that testing is critical to the system. Now that we understand the criticality of testing the system, we must document our findings and all of the steps that we took to properly validate the applications functionality through testing. While I was developing the system, I was even documenting the process along the way, writing down requirements, steps taken, and reported bugs and errors to a log that I could reflect on later. Documentation is one of the most important things in software development, as having an intricate description and development process documented will help immensely when testing, designing, and maintaining a software system.

In conclusion, taking the proper steps to properly validate and test your software system is critical to the success and longevity of your software. Implementing the necessary logic and tests, understanding your codebase, and properly documenting the process is a crucial part of development. By using different tools and techniques like JUnit and constructing test files, we can ensure we are developing a system that is up to standard and properly secured, maintained, and avoids technical debt.