* Describe your unit testing approach for each of the three features.
* To what extent was your approach aligned to the software requirements?

When writing my unit tests for each of the different classes and services I would strictly follow the requirements and documentation provided to me ensuring that each information field would not only require a user response but expect a response that aligns with certain parameters provided to me by my client’s requirements. For example, for my client’s customer class the user was required to include an id, first name, last name, phone number and address of the recipient they wish to add to their contacts. I then implemented validation logic in the form of functions on lines 29-50 that would pass along the specific information fields like for first name the requirements asked that I ensure that the first name could not be blank, so I ensured that the system would trim the users input of any blank space and throw a validation message if the input field contained a fist name that was less than 1 character in length which is present on lines 18-24 in my validation file. The other validation logic that was applied to the first name according to the requirements was that it shall not be null so I checked for if the user input was equal to null lines 12-16, and according to the requirements the first name must be no longer than 10 characters in length so if the first name was less than 1 character or more than 10 characters it would throw out a validation exception with a message detailing that the first name can not be more than 10 characters lines 26-30. I of course did this for all the other contact input fields such as id could not be null, blank or longer than 10 characters. Lats name could not be null, blank or longer than 10 characters, phone number could not be blank, null or longer/shorter than 10 characters, and address could not be blank, null or longer than 30 characters strictly following the requirements provided to me. I of course not only implemented this validation logic for the contact class but for the task class and appointment class. Through the creation of a validation file, I was able to hold all my validation logic in one area that can be called and reused in all the different classes rather than rewriting the same logic handling in each individual class boosting efficiency and cleanliness of my code base. As far as the task class goes a lot of the same validation logic was applied to the information fields as the contact class with id not being blank, null or longer than 10 characters, the name field could not be blank, null or longer than 20 characters in length, and finally the description could not be blank, null or longer than 50 characters in length all present in lines 26-42 in my task.java file. The class that has some minor changes and additions to its validation is the appointment class in which I have the same id and description validation but includes the additional date validation in lines 38-40. The date, according to the requirements, could not be null and must be a date that is either present or in the future, meaning that a date provided must not be a past date. This is validated in my validation file in lines 32-37 in which my code checks for the current date and compares the date provided by the user to it to check if it is before the current date and if it is throws a validation exception message informing the user that the date must be current or in the future.

* 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?

Overall, my jUnit tests covered 91.5% of my code base exceeding the requirements 80% coverage percentage. I obtained this percentage through eclipse’s coverage as tool in which it runs several instructions through my code and analyzes how many of the instructions are covered by my tests and how many of the instructions are missed. The coverage as tool also offers some insight as to which of my files were covered the most by my tests which were my exceptions file having 100% coverage, my validation logic file also having 100% coverage, and my contact files bosting a 100% and 94.3% coverage for the contact service file. Overall, I believe that my jUnit tests cover a wide variety of instructions and run exceptionally well, exceeding the 80% requirement. My jUnit tests were effective in thoroughly covering most of my code with the high percentage indicating that a high percentage of my code was executed during the tests I also know that my jUnit tests were effective in handling edge cases, acknowledging correct outcomes through assertion, expecting negative outcomes through assertions, and covering each individual user input field to test that when one is incorrect the input is rejected. For my contact class I reached 100% coverage meaning that 100% of my code present in my contact class was executed and for my contact service 94.3% of my code was executed during the tests still offering a high percentage of coverage. For my task class 100% of my code was executed during the tests and my task service had an 82.8 percent execution. Finally for my appointment class my coverage percentage was 90% and my appointment service was 77% which is my lowest coverage percentage. To gain this high overall coverage percentage I strictly followed the requirements turning each requirement into a jUnit test to assert that positive tests were successful and negative tests were rejected and a validation exception was thrown. I ensured my jUnit tests were effective with the use of assert statements to test that the result of certain user scenarios resulted in the correct outcomes such as a contact being added or rejected due to an invalid input or multiple.

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

I ensured that my code was technically sound by creating many test cases to cover a wide range of user given scenarios and deployed error handling for scenarios that would reject the users request based on improper inputs. An example of this is displayed in my test cases for my contact class in which I first test a successful path given a valid id, first name, last name, etc.… then I test for multiple unexpected paths for each given input like first name asserting that each improper input results in a certain validation message being deployed. A specific example is in lines 33-37 in my contact test file in which I test for when the first name is null and assert that a validation exception is thrown that is equal to “firstName must not be null” comparing it to my exception.getMessage(). This is an example of how I ensured my code was technically sound because I not only handled an improper input due to validation rejection, I also ensured that when it failed the user would receive a message informing them why their request was rejected. When originally writing my jUnit tests I went for only checking if a validation exception was thrown which they did for the most part but I believed this to be not effective enough and wanted to ensure that my validation exception was throwing out the correct expected message and not an error that was not being handled by my code which is why I threw and assert equals to ensure that the message obtained was the message I had expected the input the result in. To deal with dependency and to ensure that each individual test was strictly only testing particular parts at a time I wrote numerous tests to test for each individual outcome of each individual input to ensure that even if for example most of the contacts inputs are valid if one is invalid the contact is not created. To avoid duplicating code I used the @BeforeEach to initialize my service instance before each test to avoid having to initialize the service instance each time a test ran. Although I did not use @ParamterizedTest I did use @Test and assert all methods to assert that all improper input requests for each variable resulted in correct validation exceptions.

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

The software testing techniques that I employed in this project were use case testing, boundary value analysis, and test coverage percentage analysis. Use case testing is when you deploy example cases or scenarios that you might expect a user of your software to try and utilize your software for, an example would be testing for if a user were to input 18 as their age. Boundary value analysis is when you test for values that are near your limit, an example would be testing that any number before 10 is rejected and testing for the case of 9 which is right before the boundary or limit. Coverage percentage analysis although not strictly a software testing technique I decided to mention it because it demonstrates the effectiveness and percentage of executed code in my code base by the test cases which is equally important when testing software. An example of where I employed boundary test analysis was in my appointment test file in which I covered for even if the date was in the past by just 2 days that test case would fail because the date cannot be before the current date, this is present in lines 95-98. I decided to go with 2 days in my final project to demonstrate that any date in the past would not work but previously I also tested for if it was 1 day in the past here:

*@*Test

void failTestValidateDate\_Past() {

assertThrows(ValidationException.class,

() -> new Appointment("1", LocalDate.now().minusDays(1), "Description"));

}

Which also failed due to the date still being in the past not meeting the requirements provided to me and implemented in my validation logic. Although this example is also relevant for use case testing, I will demonstrated another example of how I used the software testing technique use case testing for my contact service testing. My contact service allows the user to add, remove, and update a contact information in their contacts, in my service tests I conducted both successful and unsuccessful additions, deletions and updates of example contacts that a user might try to utilize my software for. In lines 27-40 I demonstrate a certain use case that I believe a user will try to utilize my software for and that is adding a contact with the id 12, first name of first name, last name of last name, phone number of 7183708967, and address of address. In my test case I expect that within the service repository that stores all the successfully added contacts there will exist a contact with the key or id of 12 which means that the use case was successful. Lastly, I utilized eclipse’s coverage as tool to identify whether my software tests were executing a large amount of my code also meaning that a large amount of my code is being tested by the test files which demonstrates the effectiveness of my tests and helps me identify files that lack the proper testing.

* What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details.

Other software testing techniques that I did not use for this project include equivalence partitioning, which divides inputs such as id’s into groups of similar values that expect similar results, and statement testing which are tests that ensure all lines from a function are covered by the tests. A specific example of equivalence partitioning is testing for if the user inputs ages 18 and up the test passes and if lower than 18 fails and then testing for cases like ages 1-5, 6-10, 16-17 then 18-22 which the first 3 cases should fail and last one should pass. Statement testing could be ensuring that all functions work as intended for a phone to add an app, delete and app and update an app also ensuring a large coverage percentage for each function showing that the functions completely and steps were not being utilized.

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

Use case testing

Practical use: A website might have a form that requires user input for name and phone number that only will allow a name of 10 characters in length and a phone number that’s exactly 10 characters in length you will want to ensure that cases in which the user enters name and phone numbers of different varying lengths are denied or pass like a name of 15 characters being denied.

Boundary value analysis

Practical use: An app requires that you must be 18 or older to use the app you want to ensure that if the user were to input the age 17 that it fails, and you do not have an off by one issue with your age validation logic allowing a 17-year-old to enter an 18+ app. You may also want to try the ages 18 or 19 to ensure that 18 and one above 18 are allowed.

Equivalence partitioning

Practical use: Test for groups of inputs that should produce the same output behaviors. An example of this could be a website age limit being 16 and up and testing for groups of age inputs like 1-5. 6-10, 11-15, and 16-22 the last group being the only one that passes into the website.

Statement testing

Practical use: Tests that measure the execution of statements percentage within functions. An example could be a phone programmed to allow the user to add, delete and update apps and having tests that tests that the add function runs all of its statements, the delete app runs all of its statements and update runs all of its statements.

* 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.

When working on this project I adopted the mindset of a software tester who wants to limit the number of bugs that end up in production, boosting user experience. I employed a lot of caution with my code ensuring that every input field was thoroughly tested, and edge cases were handled in which I could see my end user trying to enact. It is important to appreciate the complexity and interrelationship of the code I was testing because it ensures that I cover a wide variety of scenarios specifically at critical points in my code and test for how each part of my code is going to impact and interact with other portions of my code. For example my contact service relies heavily on my contact class to employ proper validation logic so if my contact class were to pass along a contact with improper input fields this would drastically decrease the effectiveness of my contact service and even may cause code breaking bugs to effect the performance and wellbeing of my code. Understanding these connections allowed me to take understand why it is so important to thoroughly test every part of your code and relay exception messages that detail what part of the code is causing the problem to occur so that I can easily go in, fix it and retest that portion.

* 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.

When testing my code I went in with the mindset that my code would not work and expected the opposite outcome then what should happen and to my surprise that actually rarely was the case but because I removed the bias that my code was flawless but was able to prove that my code was actually exceptional I was able to see my code for what it really was rather than what I had hoped for. On the developer side bias can definitely be a concern if you are responsible for testing your own code because you might overlook places which you believed to work perfectly because of your own bias.

* 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.

It is very important to be disciplined in my commitment to quality as a software engineering professional because in the end I am constructing a piece of software that is going to be used by someone else to solve a problem that they have and if the software is of lesser quality the user is going to not want to use my product and it would affect the trust clients and users have with me to make a product that solves their problem. It is important to not cut corners when it comes to writing and testing code because it can leave problems and vulnerabilities in your code that may affect you, your company, and or clients beyond the piece of software such as financially or socially. I plan to avoid technical debt as a practitioner in the field by always striving for excellence and high quality so that I don’t fall victim to more work and money problems later in production of a product.