Summary

Each of the three features had its own set of required information to initialize an object of the class. Every feature also has the option to add and delete an object in the array. The Contact and Task services also had the additional option to update their respective item in the array.

The Contact class required an object to hold an ID, first name, last name, phone, and address. I ensured that it matched the requirements by putting in checks (if statements) to throw an error if the member is null or out of the bounds that were set. I then created getter and setter methods to retrieve the fields as needed. The contact service class is where I created the add/delete/update methods. Separating these into two files helps when JUnit testing, to focus on specific parts. For the JUnit tests themselves, I made sure to test each bound. In the Contact test, I ensured that adding a contact with the correct parameters worked. I also tested to make sure an error was thrown if an invalid member was inputted. To help determine where errors may originate from, each bound error was separated into its own test. For example: when testing the member ID, I could have separated contactID based on if null was input or if a string that was too long was input. In the Contact services, I made sure each method was able to function as it was supposed to. When ran, the JUnit tests came back green across the board.

The Appointment class had a different set of parameters to include in its object. This consisted of an ID, date, and description. New if statements were created with new bounds to ensure that the members were within set limits and would throw an error accordingly. The appointment services class had only an add and a delete method. The JUnit tests looked very similar to the other two function’s tests. The tests all came back correct when ran.

The Task class object took in the parameters ID, name, and description using the same error bounds as they did in the appointment and contact class. If the member was null or too long then an error would be thrown. The service class contained the methods to add/delete/update. The JUnit tests were set up the same as the other two functions and returned green when run.

Reflection

Testing Techniques

For this project, I used JUnit tests to guarantee that my code threw the errors when it was supposed to and did not throw errors when correct. There are a variety of testing techniques I implemented for each function of the project.

The first one was the separation of files. Keeping the object separated from its methods helps when testing. Since the object declaration was in a unique file, I was able to run JUnit tests to specifically target how the object is created. This improves reusability and creates an easier understanding of what is being tested.

The second technique was descriptions. When creating the throw new exception errors, I specified which member was not input correctly so that the end user could update accordingly. I do think that I should have specified more clearly why the input was incorrect. This could have been done by throwing a different output in the two bound cases: when the member input is null or when the input is too long.

The technique that I used in the JUnit tests themselves was exception testing. I used assertions to determine if the code reacted the way I meant it to. I mainly used assert true and assert throws.

Many other testing techniques were not required for this project, so I did not utilize them. One that sticks out to me is that I did not test how the parts worked together. I determined each method worked but did not extensively look at how they interact with each other. I only used the add method once in each test when I should have seen if it could handle more.

Mindset

When creating each of the three functions I broke it down by what object did I need to create and then what actions did I need to take for them to be useful. The creation of the object was easy, input each member into the class definition parameter. Then I created another file to hold the action methods. These action methods need to interact with the class object. To do so, I realized that the methods needed some to access the object’s members. That is why I created the getter and setter methods in the object file. I employed caution when creating these getter/setter methods by making it impossible to set the ID for each function. The ID must be unique and non-updatable in order to use that member as a key.

After creating my code and then testing it myself I can see that bias could be prevalent. It is easy to be overconfident in the code that you create because you’ve done it to the best of your ability. Another person might be able to look at it and notice where details could be added or removed. A unique perspective may think of another test that could be added to ensure that nothing is missed. If developers were responsible for testing their own code, they could falsify results or not test extensively enough.

This project helped me learn and practice the coding principle of method naming conventions. I originally did not name the JUnit individual tests to what they did. Instead of a simple testfirstName, I changed it to testFirstNameTooLong. This more accurately describes how this test checks that the member first name, is not too long. Along with naming each test to highlight what it is testing, I also documented information using in-line comments. For example, “// test first name is too long”. This increases reusability and helps other programmers understand the function of the project.