Joseph Grzywinski

10/08/2023

Project Two Summary Report

When developing the unit test for this mobile application, the approach that I used was to take the requirements that were provided by the customer and break down exactly how the unit test should be set up. The mobile application needed three different features to be implemented and tested before being released; these three features were the Contact, task, and appointment services and all of these needed to meet requirements to set in their user stories. I tested these to make sure all will work as necessary.

Regarding the Contact feature, the customer wanted contacts to be stored using an in-memory data structure, so we did not need to worry about a database. Each contact needed to have a unique ID, a first name, last name, phone number, and an address. Each of the properties of a contact object had their own length requirements; none of these properties were to be null and the application needed to be able to add, delete, and edit contacts. In the code I developed, a contact object could be instantiated and added to an Arraylist located in the contact service class. Also in the contact service class, I had three different methods, the first is to add a contact where it checks the Arraylist to make sure the id being used for the new contact object was not already in the Arraylist to ensure that each ID is unique. The other two methods were to either delete or edit a contact. Both methods implemented similar lambda functions to find the ID of the contact to be added or deleted.

The JUnit tests that I implemented for each of these methods were very effective based on the coverage percentage I received back from the program. The contact service class had 100% coverage according to the reports that were provided from the IDE and since all requirements were considered from the user story, I can confidently say that the contact feature for this application should work as required. One of the most important tests that was implemented was making sure that each contact had a unique ID so that no contact was the same. The way I implemented this test was by testing that the code was working if the ID did not match any of the others in our test data; I also checked that the program could recognize that (ContactServiceTest line 23), and testing if the ID was the same as one of the test contacts (ContactServiceTest line 30). By using ‘assertEquals()’ for both tests, I was able to make sure a contact being added would not have a unique ID.

The next feature that needed to be implemented into the mobile application was the task service feature. The task service requirements were that each task needed to have a unique ID, a task name, and a description. Each property of the task object had a character length limit, and, like the contact service, the task service needed to be able to add delete and edit a task from the in-memory database. This class was very similar to the Contact Service class, so the tests that were implemented for this feature were also very similar.

For the JUnit testing on the Task Service class, I made sure to check each method that was needed for the feature to be ready for launch. This included tests for the add, edit and delete methods of the feature as well as the Contact class methods to ensure that the max character lengths for each property were fulfilled. For these tests, I got one hundred percent coverage on the entire feature, so every method was accounted for and all tests passed including making sure no property was null; no property exceeded the length limit, and that there were no duplicate ID’s within the feature.

The final feature that I implemented for this mobile application was the appointment. The purpose of this new feature was similar to the two previous features, where an appointment needed to have a unique ID and needed to be capable of adding, editing, and deleting an appointment. The biggest difference of this feature and the part that I found most challenging was setting an appointment date. The user story specified that the application should not add a new appointment that has a past date. From the beginning, I needed to plan my tests around, reviewing that a user could not add a past date to the database. To achieve this, I used Java’s local data time utility which I found as an easier way to deal with dates.

The JUnit tests for the appointment class had a similar focus as the two previous features where I needed to make sure that the appointment properties did not exceed the max character length that was required, and that each new appointment did not have an Id that was matching an existing appointment. The biggest challenge and probably one of the most important parts of this new feature was checking the appointment date to ensure that it was not a date from the past. The test results from this class came back with 97.8% coverage which is well above passing for this program. Each method that was implemented was tested and works properly including testing if any inputs were null and if there were duplicate ID’s for the add an appointment method.

When I first started planning out my design for this mobile application, I first had to lay out the requirements set out in each of the user stories. I identified areas where I may encounter problems and prepared for them by researching exactly how I would be able to prepare for them before I started coding. This form of testing is static testing, by identifying areas that may need more attention before I even started coding; I was able to clear a problem before it even happened. Specifically, I did this with the date property in the appointment class. I realized that the provided method that they wanted me to use was an old Java utility and recognized that I should use the new Java localDateTime to achieve the requirements. Since I also knew the source code for the project, the JUnit tests that I created are a for of white-box testing. I was able to test the internal logic for each feature such as, checking to see if the max character length was reached and if the Id’s were already located within the ArrayList. I partially used black-box testing for this project as well, but I would say it wasn’t used primarily for the project. Since I knew the source code, I was able to do the checks on each specific method. I was not really focused on the input and output of the programs as much which is the basis of black-box testing.

When developing my program, I would say that I deployed a bit of caution when first started to plan my design for each feature, mostly because a lot of the methods did have to work together and since all of them were similar to each other. I know that if I had incorrect logic on one method, then the rest would most likely be incorrect as well. I also had to overcome the bias of me being the creator of the program. A lot of the time when I create an application I feel as though I have hit all the requirements and that the program should work as necessary, since I would have triple checked everything. In this case since, I also needed to create the tests for the application; I went in looking for errors in my code, assuming I missed something, which ultimately helped me. For example, I had initially finished the project and was sure that I had checked for every requirement that was expected, but I looked over the code towards the end and realized I did not have an expression for if the input for any of the properties was null, which was a big part of the assignment. This is where a lot of self-discipline comes in, since I must keep myself in check and not take the easy way out when it comes to programming. If I cut any corners in my application, a lot of the tests will come back and let me know that there are some cases in which my program will not work. Mistakes will ultimately cost me more time because I’d have to go back and change my code. Holding yourself accountable from the beginning is very important.