Isaac Ocegueda

CS 320

February 21, 2024

Project 2

My testing approach was based on ensuring that each variable and method of the contact, contact service, task, task service, appointment, and appointment service classes met their specific requirements. In the contact class, for example, the requirements were that each variable was to not be null or over a certain number of characters. To test this, I first tested the normal state in which the variable was within the acceptable parameters. Then I tested with the variable being null, and finally with the variable being too long. I did this test for each variable in the contact and task classes. I also tested the setter method, and in extension the getter method, for each variable in a similar way. I first tested updating the variable with an acceptable variable, then with a null variable, and finally with a variable that was too long. Running a coverage test ensured that the contact class and task class variables and methods were fully tested.

To ensure that my code was technically sound, I used clear variable and method names. For example, in the contact class, in lines 11-15, I used the variable names contactId, firstName, lastName, phone, and address. In the contact service class, in lines 18, 24, 35, 44, 53, 62, and 71, I used the method names addContact, deleteContact, updateFirstName, updateLastName, updatePhone, Update Address, and searchContact. I used a similar process in the task and task service classes. In addition, the JUnit tests also helped me test that the code was working as intended. When I ran the tests, I would slightly change each variable to ensure that each method was given new test data. I would also check the tests with different variables to ensure that the test would fail when it was supposed to. For example, in the task service test class, lines 36-41 test the updateTaskName method. In line 38, I initially set the task name to “Task3”. Then in line 39, I updated the task name to “3Task”. Line 40 tests if that change was made correctly by checking the task name with the task Id. Once that test passed, I went back and changed the name of the task in line 40 to something different and tested it again to make sure it failed. This told me that the method worked and that the test was comparing the task name and not something else.

Testing the code in an efficient way is important for continuous testing since it keeps tests short and quick. I did struggle with this aspect of the program as I think I could have possibly written shorter tests for the contact and task classes. I prioritized code coverage over efficiency and ended up with long contact test and task test classes. However, my contact service and task service classes were very efficient with each test consisting of mainly 4 lines each and each test class being a total of 75 lines or less. Something that could help build more efficient tests is using more advanced testing techniques, such as beforeEach, beforeAll, afterEach, and afterAll. These could be used to quickly create test instances using less code lines.

For these milestones, I used several of the testing techniques present in the JUnit framework. Some of these were assertions and assertion exceptions. In the first two milestones, I used the assertion assertTrue to ensure that the constructor properly created the contact, task, and appointment objects. In all three milestones, I used assertThrows to test if a null or too long variable was tried on the constructor. I also used the beforeEach annotation in the third milestone to simplify the test and make it more efficient. Using these techniques will be very helpful in future projects and development situations since they test the basics of whether or not the program works properly and can easily be repeated throughout development.

The testing techniques I did not use as much or at all are the other assertions and annotations, such as beforeAll, afterEach, afterAll, or assertNull. While a lot of these were not very necessary for these milestones, I could have used assertNull or assertNotNull in this milestone since I was testing for a null variable. These other testing tools in the JUnit framework will be helpful in future projects and development situations. Another feature that I did not use was the display name feature. This feature can be used to identify tests and their purpose more easily.

This project helped me find and build the right mindset for future projects. Employing caution was very important and involved closely testing each of the classes to make sure they worked independently so that they could work together in a larger system. For example, the appointment and task classes relied on the contact class to be correctly designed as their functionality depends on the contact class correctly creating customer profiles. If there was an error in the contact class, then the other classes would most likely also be affected which could end with an unprocessed task or appointment for the customer. This could lead to customer dissatisfaction and loss of business for the company.

Bias in code could lead to improper testing and lead to a bad app with nonfunctional code. Bias could occur when writing one’s own code since the developer could wrongly only create tests that they know the code will pass. However, an unbiased developer would test against all scenarios and look for ways to break the code. This is also where being disciplined as a software engineer is important. Taking the appropriate amount of time to write and test code can be difficult, especially when a deadline or weekend approaches. However, rushing things will only create more work to fix the problems found later. When working in the field, I plan to code a few lines and test those few lines often to ensure that the code is being build on a solid foundation. I also think it is a good idea to let other developers test the code to eliminate any personal bias.