Binaya Rimal

02/22/2024

CS 305: Project-2

Due to the overlap of functionalities and structure between the three features I approached each of them in a similar manner. I first created a class to store and retrieve data. These are the Task, Appointment, and Contact classes. Each of these classes have methods to get and set data based on requirements. Each data being inputted is validated before being assigned to an attribute. The services classes, which are the TaskService, ContactService, and AppointmentService, have functionalities to add, update and delete objects from a list. While writing the Junit test for these functionalities, I started with a valid test (Line 12 ContactTest.java), where I tested each class with expected inputs to make sure it returns expected values. After this, I tested with values that are invalid, or values that might trigger edge cases. These invalid inputs include testing when values go beyond the appropriate length, when values are null or empty, or when values don’t meet specific functionalities mentioned in the requirements. Some of the edge cases include, retrieving object from a list when it’s empty, updating data when a value does not exist, adding appointment dates that is before the current date, or adding appointment date on the same day as the current date (Line 68 AppointmentTest.java). My approach was essentially to try to break the code, upon doing so, I was able to find bugs and edge cases and fix them.

There were several approaches I took to make sure that my code was technically sound. A primary goal was to ensure that nearly every line of code was executed. If a decision statement, a line in a loop or an error message is not being triggered when it’s expected to, there is likely a bug in the code. My coverage for each feature is 99%, this level of coverage isn’t always viable with time constraints, and large codebases that require configurations to access certain functionalities, but for this project, it wasn’t an unreasonable task.

A grid of white paper with black and blue symbols

Description automatically generated with medium confidence

As mentioned earlier, I made sure to validate each user input and test them. For each input validation, I added proper error handling, with the inclusion of try catch exceptions. For efficiency I made sure to follow standard coding practices like not repeating any code, emission of nested loops, and making sure each function has one purpose. An example of one of my techniques for efficiency is where I write commonly used inputs as global variable, so I don’t have to write it on each test (Line 12 TaskTest.java).

I primarily relied on White Box testing techniques and Experience-based testing techniques. I implemented Statement and Decision Testing Techniques to try to trigger every condition and statement. I used inputs that triggered every if and else statements, and every loop. After making sure I covered nearly every line of code, I used various Experience-based testing techniques. I checked if there are any inputs that will prevent the application from triggering any functionality of the app, for instance, in appointmentTest class, I check how the application will function if date is formatted wrong. Another example of this is in appointmentServiceTest where I test if I can delete appointment when the list is empty (Line 63 AppointServiceTest.java).

I didn’t use any Blackbox techniques; although, some of these techniques could’ve been useful. I could’ve used decision tables to define how the program should run based on various user interactions; however, the application was simple enough that I didn’t think it was necessary. I did not use any use case technique as there aren’t multiple actors and the requirements for each assignment were clear enough that I didn’t deem it to be necessary.

Black box techniques are useful while designing tests and testing against the end-product. They are useful when we are dealing with large scale projects with many objects, actors, and scenarios. For instance, in an application where there are multiple users with different levels of authorities, it could be useful to draw a use case diagram to visualize the roles of each type of users. Similarly, implementing a decision table can be useful to visualize how each decision should impact the program. White box testing techniques are useful for developers to write tests. Using decision and statement techniques allows developers to trigger every aspect of the program which helps ensure that the code is bug free. Experience based techniques can also be very useful to determine common cases where a program might fail. Based on testing experience and experience using similar applications developers can create commonly used scenarios to test against the program to make sure the program runs as intended. Developers can also test with rarer scenarios which might be missed like using incredibly large inputs to see if the program runs efficiently or checking with invalid input types.

While writing tests, I was cautious on being as thorough as possible to consider every possible type of input. A concern I had in my approach was towards making sure that the tests are implemented properly without any false positives and negatives. While running the tests, there were cases where the tests passed even if they were not supposed to, which I was able to luckily catch and fix. I believe this is where a tester should really show caution, because it defeats the purpose of testing for flaws and bugs if the tests themselves are flawed. I believe developers should also run tests frequently because the tests might pass at one time, while failing other times, this is especially true with anything that involves dates. In my code, my test to create valid appointment was passing one day; however, it failed the next day, this was due to my current date returning values that weren’t accurate. If I didn’t test it frequently, this might not have been caught. As one method impacts the other, it is important to have that reflected in your tests as well. In this project, all the services methods rely on objects being created and validated properly, therefore, it is important to test the original class before we can test any of the services classes. For instance, to test deletion of a task object, first there must be a task object created and then added to the list. While testing we must mimic this by manually adding task object to the list in our task deletion test.

Despite being cautious and thorough, my tests are unlikely to be perfect due to personal bias. Because I wrote my code and tested them as well, they both will go through a similar process of thinking. There might be cases that I’m not considering because I’ve never come across it. To tackle this bias, I tried my best to look at each code and dissect it as much as possible to generate user inputs and behaviors that might cause it to break. Based on this I wrote my tests with the intention of triggering every functionality in each method. I tried my best to access every if statement, try/catch blocks, loops etc. and tested them with values that might cause them to break. For instance, I tested my deleteTask() function with valid ID, invalid ID, and when the list is empty.

Testing is the last checkpoint before the client can access the product; therefore, it is of utmost importance to be disciplined and not cut corners. Early detection of bugs saves companies from financial loss, security breaches, data loss etc. Testing helps ensure software to be on par with industry quality standards and checks if software meets all requirement. To avoid technical debt, it is important to gain feedback, perform best coding practices, consider edge cases, compare tests with peers, and consistently learn and develop your skills to adapt to the changing nature of software development.

Citations

Atlassian. (n.d.). *The different types of testing in software*. https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

*What is software testing and how does it work?*. IBM. (n.d.). https://www.ibm.com/topics/software-testing