Jacob C Wilson

08/13/2024

CS-320

Project Two

The software that I have provided included understanding requirements for these separate app components. This consisted of Tasks, Contacts, and Appointments along with the services provided to maintain a list of these respective items in our current application. Through these important components I have conducted numerous tests that allow the thorough testing of their adding, updating, deleting, and retrieval functionalities. The contacts service along with the contact items consisted of a unique id with no chance of having a null id through checking and ensuring these measures are accounted for. The null checking is also included for the various other attributes consisted in the Tasks and Appointment objects. I have conducted these checks for the individual objects along with the services that manipulate a list of these objects respectively. I have created measures that ensure there are no duplicate ids that would be in the list of objects for each section of the application. Coverage percentages are highly valuable percentages that help developers make sure they are testing every aspect of the code created for a program. Through these percentages, I provided evidence that my project code will be tested thoroughly and could have honestly created increased tests from edge cases found from tests I created for this program. These percentages also help ensure that all the methods in the program are being hit, along with every single line at play in the program. JUnit tests are meant to test the code given from a program and I created as many tests with edge cases as I could so that it can be as thorough as possible. By utilizing these percentages, I have been able to test whether the contact, task, and appointment service code will be covered or not in my tests so that I have tested every single piece of code at play.

In terms of making sure the pieces of the application were technically sound and contributed accurately towards the overall goal of the application, I had to make sure the checks for null ids, empty strings, and lengths of each string were concise with the requirements for each attribute. To create methods for the services portion of the application, I had to ensure they were technically sound by creating checks for the null and empty values from the start of the service process. I created separate methods that would allow for checking these status’s and throwing exceptions if they came back as being an empty string or null, or in some cases – too long or too short. Creating separate methods for checking the conditions for being null, empty or too long or too short allows for code efficiency by making the code more readable and read straight down the method as if it is a book. The efficiency in the methods created was from using a for each loop to determine whether I should return the found object with the similar id, or whether I should assign a new object to be returned later after I found the object. Well, from this contemplation I found that it is more efficient to return once you find the item in the list rather than continuing to loop through the whole list at once.

For this project, we employed unit testing and exception testing as our primary software testing techniques. Unit testing focuses on verifying the functionality of individual components in isolation. In our case, we thoroughly tested methods within the `ContactServices`, `TaskService`, and `AppointmentServices` classes to ensure they correctly added, retrieved, updated, and deleted contacts, tasks, and appointments, respectively. Each unit test was designed to cover various scenarios, including valid and invalid inputs, to confirm that the methods behaved as expected under different conditions. Exception testing was also crucial in this milestone. It involved validating those methods appropriately handled erroneous inputs by throwing specific exceptions. For instance, we tested how the `ContactServices`, `TaskService`, and `AppointmentServices` classes reacted to null or empty IDs and ensured that these cases were handled gracefully without causing the application to crash.

We did not use integration testing for this milestone. Integration testing aims to verify the interactions between different modules or services within an application. This technique is essential for identifying issues that may arise when individual components are combined and need to work together. In our project, integration testing would have involved checking how the `ContactServices`, `TaskService`, and `AppointmentServices` interacted with other potential modules, such as a database or external APIs, to manage contacts, tasks, and appointments. By not employing integration testing, we may have overlooked potential integration issues that could affect the system's overall functionality when these components are combined.

Another technique that was not employed is system testing. System testing involves evaluating the complete system to ensure it meets the specified requirements and functions correctly in a production-like environment. This technique is comprehensive and covers all aspects of the application, including performance, security, and usability. For our project, system testing would have been beneficial to confirm that the entire contact, task, and appointment management system worked as intended from end to end. By skipping system testing, we might miss identifying issues that only surface when the application is tested, potentially impacting the user experience and system reliability.

My mindset when it came to this project involved introducing ideas that the clients’ users would ultimately make mistakes or that mistakes could happen, and so I would try to prepare for every possible scenario that could arise from the application failing. For instance, I would check if the ids or any of the other attributes would be null, empty, or have too much length to them. Checking for overall sturdiness of the issues that could arise from technical errors on my end or from the user’s end is important for any compilation or runtime errors that could arise from the application. This could include users neglecting to put in a phone number, address, id, name, or title in any of the objects described by the classes in the application, or when adding in a new object through the respective services. I controlled bias by paying close attention to how I could control duplicate ids. If you look through each of the different sections of the application, I have three ways of taking care of this – one way was through checking in the services and making sure that the id did not match any of the instances in the list, another way was through having a static id in the object class itself to make sure that the object’s id needed to be higher or lower than the specified current id, and the last way was having the static id in the service class that would control this similarly to the previous way. I could imagine this would be a concern on the developer side because you need to find the best way to handle duplicate ids in a unique-id collection.