Project Two

Summary

My approach to unit testing was directly aligned with the software requirements. Utilizing testing for the get and set methods for the different fields needed for the contact, task, and appointment projects helped to ensure that the fields could not be created with or updated to contain invalid information. This can also be seen in each Service class, where I tested updating each field in an existing and new contact, task and appointment. This aligned with the software requirements by ensuring that invalid values would not be allowed in any field, and that each constructed class was created with a unique identifier.

Based on the system requirements my JUnit tests had as close to 100% coverage percentage as possible. Testing was used to confirm that every restriction that needed to be followed was followed. Each getter and setter were tested individually with both valid and invalid input. The creation of each class was tested to ensure invalid input was not allowed. These tests provided reassurance that each new contact, task or appointment could only be created with valid information for each field and that each also had a unique identifier that could be used to search for the correct entry.

When writing JUnit tests, I tested each field individually. This can be seen in all packages. For example, when testing for updating contact fields, I tested the first name, last name, phone number, and address all separately. This allowed me to test each field without worrying if another change could influence the results. I tested each restriction that each field could have individually. For example, when testing a task’s description, I performed separate tests for both null values and invalid lengths.

To ensure that my code was efficient I avoided using repetitive code. This can be seen in the AppointmentServiceTest, where I first wrote a setup block of code that was performed before each test. This allowed me to reuse variables for different tests without having to create the variables each time. I also utilized an After Each code block that cleared the hash map so that the map was empty before each test was performed. Using these methods I was able to create efficient code that meets all expectations.

Reflection

The software techniques that I employed for this project included unit testing and white-box testing. Using unit testing I isolated individual components of the program to ensure they had proper functionality and behaved as expected. Since I was the one who created the packages, I was able to utilize white-box testing to test specific functionality of the program. Knowing the restrictions that the classes held, I used white box testing to test specific faults to confirm that they would not be accepted by the systems.

Since this program was self-contained and did not use any form of external systems, integration testing was not used. Integration testing is used when determining if outside systems are communicating as expected with the internal systems. The tests I performed also did not test the performance of the systems. While the projects were using small sets of data compared to other systems, during a performance test the limits and speed of the system would be pushed to its limits by throwing using large amounts of data and testing how long it would take for the programs to complete the tasks.

Unit and white-box testing can be used frequently and early to help catch faults quickly. They also allow for development to be quickly tested by writing tests as you create the program, ensuring that each new function that is added behaves as expected. Integration testing is integral when using resources from outside of the program such as databases or payment processing to ensure data integrity and communication is working correctly. In a real-world environment, users expect software to work quickly and reliably, that is where performance testing would come in. Pushing a system to its limits to determine if it can handle large amounts of data and multiple users at once can be done by performance testing to help verify that the system meets user expectations.

When operating as a software tester it was important to adopt a cautionary mindset. I needed to make sure that any possible fault that could be present was tested to verify that proper error handling was in place. Testing for all forms of invalid input allowed me to proceed with confidence when the tests gave the expected results. This can be seen when testing for invalid dates. Since an appointment date could not have been in the past, I had to ensure that if the date was before the current date that it would cause an error. Deliberately testing with invalid values helps to confirm that the error handling was properly implemented.

Limiting bias is needed to properly test your own code. Even if you are certain that your code will behave as expected testing can reveal unexpected faults and can give insight into an issue you may have overlooked when developing the code. Even if I know that I have implemented a duplicate id check, I still need to test that functionality to know that it is working properly. Having proper testing also allows for future changes that may be made by another developer to see how the older systems were expected to behave. I overlooked some issues with the initial contact testing, such as the phone number. While I did test for invalid length and null values, I failed to test if the phone number would accept non-numbers. This would allow user to input letters instead of an actual phone number if it fit within the length. This shows that even when trying to be cautious and removing bias, some aspects may slip through the cracks if you are not careful.

Being disciplined in your testing means creating tests even for minor issues. A non-disciplined approach may not verify the limits of fields such as the address or appointment description. This may seem like small issues, but if multiple fields allowed for extremely long entries, then performance and the user experience could suffer. To avoid technical debt continuous testing should be used. While testing can seem like a waste of time and monotonous, proper testing can help avoid headaches in the future. Testing frequently can catch errors early on, allowing you to implement a fix before developing a large system. The larger the system, the more complex the code becomes and the harder it is to address any issues that may appear. Implementing comprehensive tests will help to make any future changes easier. Making changes to existing code that could make it more efficient or cleaner can quickly be checked to see if it breaks anything else by rerunning the existing tests.