Brendan Clarke

CS 320

02/20/25

Project 2

Software testing is one of the most important steps in the SDLC, it will ensure that your code not only functions well enough not to crash and run into app breaking errors, but keeps your users perspectives intact with each use case of your code. The approach I had used for my JUNIT testing was starting with understand how my code will be used. Finding each possible input that user will have and then try to shape the tests to be what would happen if the user entered in an input which is not was expected, as well as attempting to test what was expected. This is so the code I wrote will have the ability to give the system some freedom by deciding how to handle what type of input the user will give. By creating this sort of soft release, it gives me better understanding that the system can handle complex procedures that may occur when a user is interacting with my app.

The software requirements had requested several tasks for my code to be able to fulfill, one of such tasks was the ability for the user to input personal information with varying interpretations of these inputs. Also, for the Task objects in the task class requirements, it needed to have a unique task ID which was no more than 10 characters, which we were able to achieve, and validate through our JUNIT tests. In such, I think my JUnit tests found nearly every justified input through each of the tests, and was able to confirm that the system would be able to contend with these inputs, and not run into any issues and be able to give the user a meaningful enough output for them to understand why their input was either successful or not compatible with what was expected of the input. In my appointment service class, I had code that set appointment date constraints

assertThrows(IllegalArgumentException.class, () -> new Appointment("12345", pastDate, "Test"));

assertThrows(IllegalArgumentException.class, () -> new Appointment("12345", null, "Test"));

This ensures that the appointment cannot be made for a past date, and that the system would prevent such creation of appointments. And for the efficiency of my code, I set redundant checks to make sure that an appointment ID does not exist before attempting to add. This will make sure there is unnecessary insert operations and that any data for an id is not over written.

if (appointments.containsKey(appointment.getAppointmentId())) {

throw new IllegalArgumentException("ppointment ID must be unique");

I had employed the use of JUNIT for my unit tests that help verify each individual input or component of my app were able to function without issue. Unit tests work by isolating each section of code to set it to achieve one objective and ensure that they meet the software requirements. Boundary testing was also one of the types of test I utilized. This checks specific edge cases such as the description not exceeding a certain text limit. One testing type I did not employ was the use of acceptance testing, which will evaluate the application from the users perspective, this is because we only really focused on backend tests to achieve functionality without crashes and not necessarily tests that focus on whether the objective user use case goals were achieved. In most practical scenarios, unit testing will be critical for ensuring base level of code is functional, whereas integration testing is valuable when we have multiple apps working together, or maybe a database and a user end app to interact with it. Acceptance testing is truly just ensure that a user is able to use the application as they might expect to use it.

When working on my apps, I had chosen to follow a detail oriented and cautious mindset, this is most likely because of the way we had testing to run after word that weren’t just looking for errors in compiling. Doing such though forced me to ensure specific details of my code were able to run and accept the input and process it as it would need to be utilized in my system. Without such an approach many of my tests would have run into issues on the logic standpoint or on how efficient my app is. In order to limit bias while reviewing my code, I had to take a tactical structured approach, by following the predefined requirements word for word instead of assuming how I felt the code should run. I could have assumed my implementation of appointment deleting had worked without testing, but in order to ensure it was operable, I had run tests to validate error handling where the id for the appointment did not exist.

Overall, my commitment to the quality of code will be integral to my coding career, as software engineering will always give me complex objectives for issues that can occur long-term that I may not be able to see at that very moment. Through the practice of code review and the various methods of testing my code, I can ensure that the software I am creating is able to function in the meantime, and be scalable for the future of its lifetime. Testing, is one of the most important steps to software development, and so long as I follow this, my intention to keep my code robust, will come true.