Corey Wallace

CS320 – Final Project 2

When developing I conducted a test on the mobile application process called JUnit testing, which involved checking that each feature's component parts and methods adhered to the requirements. As an end-user and a developer, I tested this. As soon as a working module was developed, I started weighing its benefits and drawbacks from the perspective of the final user. When I discovered any possible problems with the code, I took over as the developer and make them right. Throughout the project's development, I repeated this procedure to ensure the program functioned as planned.

One of the functional requirements for the contact functionality was that the name fields not include null values. I made two JUnit test cases with null values given to make sure to throw an exception. The following JUnit test case checks to see if the setLastName() appropriately using the Contact class's method raises an exception in the case of an incorrect null value is entered as an input. This test passed the test case because, as expected, it threw an error during runtime.

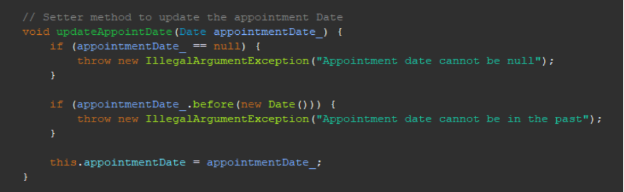
Text

Description automatically generated

The appointment features of the app were tested under comparable circumstances. For instance, one of the conditions was to not permit a date that had already been decided. I created a test case in JUnit that checks the entered date against throws an exception if the current date is past due. Once the test case has been validated, the Appointment setter method throws an error.

Text

Description automatically generated



Similarly, I created a JUnit test that raises a special case where a task ID exceeds 10 characters because the Task class required that the taskID not exceed 10 characters.

Text

Description automatically generated

I adhered to solid coding and design principles to make sure my code was technically sound while maintaining organization and readability. For instance, I verified that the input was null or greater than 50 characters as required before changing the description in the Appointment class below using the updateDescription setter function.

For my project, I employed JUnit testing as well as other unit testing methodologies. An open-source framework called Unit is used to test certain programs, to put it short. It reduces the amount of time needed for the project's final debugging. The constructor uses the "Assertions.assertThrows()" method to create a "IllegalArgumentException" based on the test's assertion when a null value is given. The predicted exception class and a "newList" object with a null value in the Name field and a non-null value in the Description he two inputs that the "assertThrows()" function uses are field requires.

At the end of the SDLC, finding defects early on with JUnit testing can help save time and money. This helps identify unreported issues that may arise following frequent changes to the code. Modern software's continuous integration and delivery cycles depend on this technique. Scalability is improved and development cycles are reduced.

I used caution when looking for any flaws and mistakes in the created code while assuming the role of a software tester. In order to consider every situation that an end user may experience, I often shifted between the mindsets of a developer and a tester. Because I understood the intricacy and interdependencies of the code being tested, I took care to test each method's output. When testing anything, it is useful to consider how it may affect other systems or programs. Additionally, you can see any application flaws that would require more extensive testing. Communicating with developers to improve the code is also made easier when you understand it.

The two classes' bequest qualities and their potential effects on things apart from the one being examined should be considered by the tester while evaluating the deleteAppointment function.

A screenshot of a computer

Description automatically generated with medium confidence

I made sure that every assertion was testing the desired behavior by closely examining the test cases. By following a rigorous process and closely examining the code, I made sure to mitigate the impact of bias, which can lead to gaps and make it more difficult to spot any problems. The Contact, Task, and Appointment classes all contain comparable JUnit test cases and the same field validation in their methods.

Software that has been poorly designed, written, or Tested systems may have serious issues including bugs, crashes, security flaws, and system outages. These problems may have adverse effect on end users and the reputation of a client, which might result in a drop in sales and client trust. When developers take short routes or fail to notice potential flaws, technical debt may quickly mount. Fixing technical debt takes more the longer it is neglected, the more time, effort, and resources are wasted.