While developing this mobile application, JUnit tests were employed extensively to ensure that the software was always aligned with the requirements. JUnit tests were written for each specific function, testing for outliers (boundary value testing), acceptance, and functionality. In doing so, these tests covered over 85% of the overall code, a great indicator that most of this code meets the client’s specific requirements with little extraneous code.

In the code itself, I hard-coded specific print line statements that the system would output once an action was done correctly. For example, once a contact was accepted (passed all criteria for adding a contact), the system would output “Contact with ID (contact ID here) was added successfully”. So, even though there was a specific test in place to ensure that correct contacts were added successfully, the system would also inform the user that it had done so. This was to make sure the code was technically sound. Even though tests were being performed, I wanted to make sure the code was functional, which it is. Efficiency during this project was not overlooked, however I believe my code can be refactored to different variable names. The variable names as they are now are a bit lengthy. This was done for testing purposes. Matching the variables to the corresponding JUnit test was done by naming the variables’ names that matched the tests. This resulted in some variable names that are a bit longer than necessary. Refactoring to a single character or shorter variable names can be done to improve efficiency. However, I stand by the names as they are now because testing was the primary focus for me during this project and I did what was necessary to ensure the tests and requirements were met (longer variable names to match with the tests).

As mentioned earlier, I employed a couple of different testing techniques throughout the development process. One specific technique was boundary value testing. I made sure that variables such as contact ID were not replicated and were unique, and that other variables, such as Appointment Description, did not exceed the specified number of characters. I also ensured that required variables were never null. All of this ensures that boundaries, whether they are the minimum or maximum boundary, were all tested. Acceptance testing was also a technique that ties in with boundary testing. I needed to make sure minimum requirements were met and that the correct type of input was entered. A further requirement test that I could have made better was the contact phone number. A string of no less than 10 characters is accepted, but this leaves room for a person to enter characters, not just numbers. A further step to validate this number would be to change the type to integers only. However, this was not a specific requirement outlined by the client, so I kept to the script and only tested the client’s acceptance requirements.

Regression testing was another technique I employed while in development. Any time a change was made to the code, I simply reran all unit tests to ensure everything still worked and passed tests. This is simply doing my due diligence as a programmer, however, this is an important and sometimes overlooked testing technique. In running unit tests after each code change, errors and bugs are easier to track. I did not test performance, as the specific scale and speed needed by the client was not specified. I also did not employ any security testing techniques for this project, as there was no security details provided. I also did not use any databases, so a RESTful API was not employed during any part of my development process. In other circumstances and projects, security would be at the forefront of the development process, but again, the client for this project did not list any security needs or concerns for any particular method or object. Practically speaking, I’m sure another part of this project will include some sort of security protocol, since a user must be validated to access their specific tasks, contacts, and appointments, but I digress.

Within this project, I tried to eliminate any bias I felt in regards to my own code. Testing to ensure every requirement was met was done objectively. I failed many tests many times, and in doing so, I was able to create cleaner, more efficient code. Once all tests passed, my code had undergone massive transformations. This is great, in my opinion, and this mindset of everchanging and optimizing code helped me in eliminating any bias. I wrote all functions first to meet all requirements, laying out each method and object how I initially thought it should be done. Once everything was written, testing began. Immediately, tests began to fail, and I employed a grateful mindset. This meant that my initial code was sloppy. Once I changed one thing, another part of my old code would fail, naturally. I was able to move on and optimize the next section. I thought of testing as a great way to find my own failures and shortcomings, and learned a lot throughout the testing process on how code SHOULD be.

Testing is a touchy subject to some developers, as they are married to their code and work hard to create something beautiful. When a test says that code is wrong, a developer may become angry or flustered that all of that hard work has to be changed. This is bias. However, I find the chance to become a better developer overall exciting. This eliminates any bias I may feel towards changing my code. I do not get emotionally attached when writing the first draft, so to speak, of functions. I try to get all basics covered, and through testing, begin to create more robust connections and code between all of my objects and functions. This is the importance of staying disciplined as a developer. I always want to get better, to create functional, optimal code. If there is a better way in doing something, I believe unit tests reveal them. Through testing your own code, you become a much better developer.

There is a biased downside to unit testing as a developer, and that is cutting corners and writing code that you know will pass your self-written tests, but not meeting all requirements. In my opinion, this is just a sign of a lazy person in general, and I am not a lazy person. I would rather have both great accuracy and coverage rather than 100% testing coverage with 0 tests passed. I don’t change the tests to fit the code, as I could see some developers doing, as unit tests are much easier to write and manipulate than developing better code. This is a very lazy way to develop software, however, and I do not advise this. Bigger problems will arise, such as requirements not being met, if unit tests are written to just fit sloppy code. Always change the code and keep the stringent tests as they are. This is how I completed testing for this project.

Citations

*Software Testing Techniques*. (2021, February 22). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/software-testing-techniques/

‌