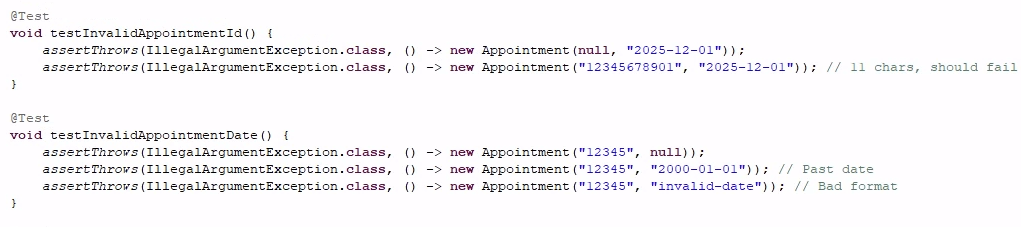
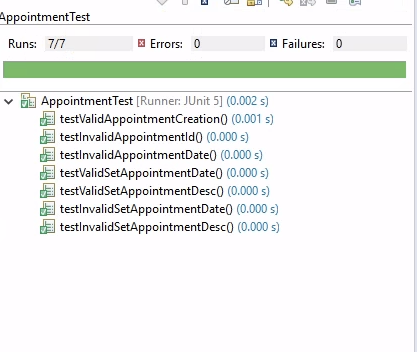
Unit testing approach

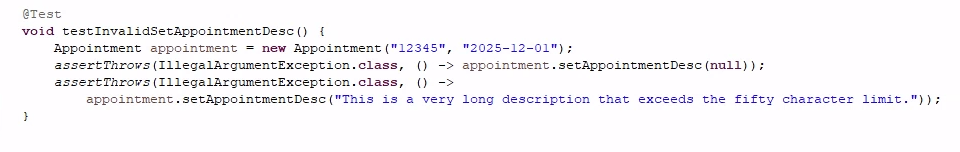
While developing the unit tests for our mobile application which primarily focuses on adding deleting and updating contact, tasks, and appointments I devised individual tests for each feature to ensure functional accuracy, edge case handling and robustness against unexpected inputs. As evident with this screen shot:

Similar tests were developed for the other features. In this particular screenshot we test for the software requirement of 10 characters or less for appointment ID, set by the client. Likewise we are testing for the correct format for the appointment date inputted and also consider the requirements of testing if the date is in the past and therefore invalid.

Again as per the screenshot, all test ran effectively without failure with coverage running as high as 79.5% thus ensuring adherance to requirements. 

The percentage figure represents the percentage of code that was executed while running our tests as the test ensure soundness by validating input, and adhering to the clients expectations.

Here is an added example of boundary testing for maximixing efficiency:



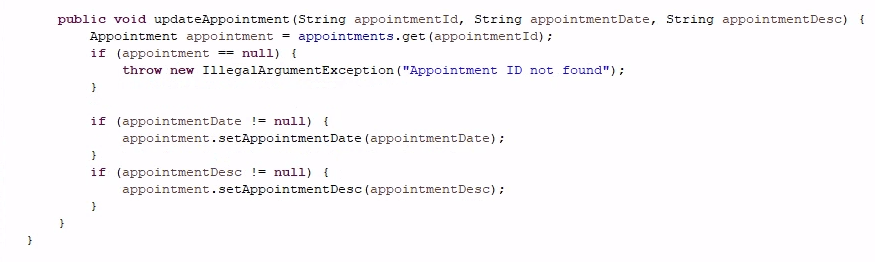
Reflection

The testing techniques utilized in this project were JUNIT testing, which includes black-box testing, boundary value analysis, exception handling.

For blackbox testing, the logic is evaluated without taking any account into the purpose of the application. In this instance when we are evaluating a date for our appointment, we insure that the date is not in the past. Boundary testing whenever we are validating that certain stored or inputted values are exactly 10 characters. And lastly exception handling when if there is a blank or null value, we don’t accept the input as valid.

Blackbox’s practical use is to run the code and observe the behavior that the client expects. Boundary testing once again puts the limits of the clients predetermined requirements for their use case which can be implemented and scaled. Lastly exemption handling prevents the program from crashing when an event that is not allowed or expected occurs.

For this particular project I implemented a proactive mindset. Here is an example of providing feedback to user on an appointment that does not exists:



This is considering that the purpose of this particular part of the overall program is to set appointments.

Bias can definitely creep in when you aren’t objective enough. The help I received was reviewing my feedback after submitting my milestones/project 1. In addition, not testing for when the code works, but also when it does not. Once again we refer to my image of when code is tested for a specific amount of characters in order to be accepted as valid.

Being disciplined not only makes me a better team player, but it also avoids costly rollbacks or even refactoring large portions of the code simply due to faulty logic or assumptions. In the field I intend to not only request outside evaluation of my code worthiness but also building with regression in mind as the code evolves. The above screenshot updatedAppointment throws an exemption when an Appointment ID is not found. In future when this code is used, we can make sure that every Appointment ID is found and safeguarded against invalid lookups.