Project 2 Summary and Reflection

In module three, I created a Contact and Contact Service java program which closely resembled the Task and Task Service java program in module four and the Appointment and Appointment Service java programs in module five. Each program required the setup of an object and to create basic setters and getters for that object. In module three the object was a Contact, in module four it was a Task, and module five it was an Appointment. These objects have specific attributes such as a unique ID, name, description, phone number, date etc. Also, these attributes have specific requirements to be valid, such as a phone number must be exactly 10 digits long, a description must be less than 50 characters long and an appointment date must be in the future.

To test these requirements, a Junit test was used for each setter and getter method by setting the attribute to a string that was either within the parameters (valid) or outside the parameters (invalid). By using the “Assertions.asserEquals() for the former and Assertions.assertThrows() for the latter methods, the Junit tests were able to prove that the attributes were coded correctly. JUnit testing also has a coverage percentage feature which shows how much of the tested classes code is covered. This is why we were tasked with creating a specific test class for each java class. For example, the AppointmentServiceTest.java shows a 100% coverage on the AppointmentService.java class. This shows that there is complete coverage for the class the Junit test is designed for.

There are four main testing methodologies: Unit, integration, system and acceptance testing. There are also non-functional testing methodologies such as performance, security and stress testing. I did not perform any stress or performance testing. Building these four java files required the use of unit testing, integration testing and acceptance testing. Functional and non-functional testing plays an important role in the software development lifecycle. Each methodology within these two styles has very different rules and regulations. Non-functional testing is used for testing what goes on behind the scene. Security of the application, performance and compatibility of the application as well as the overall usability. The implications for not testing these four main categories before an application is launched can be catastrophic. The four functional methodologies listed previously are used to thoroughly test how the application will function on the users end. These tests are what the majority of people will see initially and if anything is off, the company may suffer customer loss due to a poor performing application.

This project was broken down into three sections, each containing two java class files and two Junit test files for those class files. Breaking the overall project into three sections helped me not be as overwhelmed when reading the final projects requirements. Taking this mindset one step further, I was able to break down each java class into individual components and code for those functions then move on to the next function. I was careful to make sure to not code any functions, such as setters and getters, that were not specifically called for. Also, I had to make variables and functions encapsulated to protect them from outside intrusion or modification. This was done by making specific variables private or final, as well as making certain functions private. If everything was public, this would increase the security vulnerabilities.

In order to limit a bias when reviewing my code, I used a Junit test to test each part of my code for vulnerabilities. Since I wrote my code and manually reviewed my code, if I missed anything during my review process, it would be very difficult to find the issue during subsequent review sessions. This is where a Junit test comes in. The test is an unbiased way to test your code for deficiencies and vulnerabilities. Without this test, my code would have had many security issues since it helped me reevaluate my code multiple times.

Finally, as a software developer, my commitment to quality coding practices must be highly regarded. If I were to lapse in my commitment and cut corners, or do lazy coding, I could do some serious harm to the company’s reputation as well as introduce security vulnerabilities into the application. Technical debt is when the developer writes software code that violates good coding practices resulting in flaws in the code. If these flaws are not caught or left unfixed, they put the business at serious risk. To avoid these pitfalls, constant diligence is needed to not be lazy, as well as putting in the required effort.