Reflective Analyses of Mobile Application Development: A focus on Software Testing

Nicholas Kreuziger 12/6/2023

# Summary

The development of the customer’s mobile application was heavily guided by the software requirements. In extensive consultations with the customers, we gathered requirements surrounding the system needs to enable functionality between Customer, Task and Appointment objects. Utilizing these requirements we formulated the system design requirements for the Customer, CustomerService, Task, TaskService, Appointment and AppointmentService objects. After completing the initial code to specification we also wrote Unit Tests that verified the functionality was in alignment with those requirements. This process was done for every Object before moving onto development of the next object. This ensured a clean approach that was easier to manage. The JUnit tests were all in alignment with the requirements, and using JaCoCo we determined 89% code execution in these tests (Figure 1). The majority of uncovered code was in the ContactService Class because there were additional GET methods in that class that were never utilized in the software functionality.

*Figure 1.*

A screenshot of a computer

Description automatically generated

I took great care to ensure the code was technically sound and efficient. To verify that code was technically sound, the JUnit tests were made in alignment with the requirements provided, each test listed the requirements of the class with specific care to test each requirement provided (Figure 2). A unit test was created for each Class of the code that covered every requirement, executing 89% of the total code base. Code efficiency is a test of how swiftly code will execute, each test case is an example of the time investment of real-world functionality. My tests indicated that testing the full functionality of each class took less than 0.01s. This suggests that this software would run efficiently at scale generating 100s of records a day in a real business environment (Figure 3).

*Figure 2.* *A screenshot of a computer

Description automatically generated*

*Figure 3.*

A screen shot of a computer

Description automatically generated

# Reflection

This Project employed Unit Testing at each implementation of the classes Contact, ContactService, Task, TaskService, Appointment and AppointmentService. The characteristics the tests were based off were directly from the requirements. This involved verifying that each class object had the proper tolerance for nulls, character lengths, object deletion, object updating, and duplication. An example of this would be Milestone 3, which involved writing the Contact and ContactService Classes in addition to the ContactTest and ContactServiceTest classes to verify the functionality of the Class.

This project did not utilize additional testing methodologies such as Integration Testing, System Testing or User Acceptance Testing.

* Integration testing would be testing the interactions between the units of the Software system (in this case, the interactions between Contact, ContactService, Appointment, AppointmentService, Task and TaskService). This functionality would help to understand if additional functionality is required for the objects to work as intended in the Mobile Application. The requirements may have missed critical details about functionality.
* System Testing would be testing the functionality of the entire system. At this point comprehensive user stories would be put to the test with a functioning version of the software. Examples of features tested would be creation of 20,000 Tasks in a day and Updating of 500 of them, 1,000 Appointment creations in a day with cancellation of 230 and updating of 500 of them.
* User Acceptance Testing would be testing if the system aligns with the Expectations of the Stakeholders. The stakeholders would be provided with a beta version of the software to interact with. This type of testing is useful to determine if expectations exist that weren’t explicitly outlined in the requirements, an unanticipated functionality exists, or bugs exist in the software.

This project taught me extensively about the mindset of a Software Tester and why it is prudent to exercise caution. I did not fully anticipate how thorough one must be while testing a software project. The development of this project was not a degree of separation from the code, as a result it was more intuitive to develop tests with an understanding of the code base. If I were a software tester working on a new project with a foreign code base, I would take a considerable amount of time to fully appreciate the ins and outs of the code before writing tests. Caution has to be exercised because you could be writing tests that don’t fully test the functionality or evaluate false positives. A specific example of this was testing the Appointment Service Class, without knowledge of the constructors for the class, or the datetime format of an Appointment Object I could mistakenly test the wrong date format and receive a false positive.

Bias is very easy to introduce into the software testing process, especially if you took part in writing the code. I took steps to limit bias by leaning very heavily into the requirements documentation. I separated my tasks into development and testing and changed my hats as needed. The Developer was busy ensuring the software was functioning in alignment to requirements, while the Tester was trying to best determine how any possible bugs could be found. During the Appointment and Appointment Service class milestone I took the time to write utility classes to generate unique IDs for the Task, Contact and Appointment Classes. The process of writing those classes and changing how IDs and the Constructors were implemented resulted in most of my tests having to be rewritten. It forced me to critically evaluate the inner workings of the code to verify I was properly addressing the test conditions. Bias is very much a concern, and in a perfect world someone else is testing your code. A developer is very much partial to assuming base-level assumptions that aren’t guaranteed in the initial launch of the code.

Discipline in the field of Software Engineering is a continuing challenge. Scope Creep, improper project management, poor product management and bad project pacing are all real-world examples of pressures a Software Engineer will face. In this field is it important to push back on unrealistic expectations and insist on the importance of being thorough and testing a product to completion. Cutting corners can result in bugs that destroy the customer experience, create faulty data, deleting data entirely, being used in an unethical way, and in the most extreme cases causing fatal accidents or inciting war. The responsibility on the shoulders of every generation of Software Engineers is very high. I intend to be a responsible engineer of Software and push back when needed, insist on ethical use of my software, and seek other opportunities if I see no other choice.

Technical Debt is also a very taxing reality of the corporate landscape. Technical debt can be avoided by exercising a few best practices. Restricting scope of a project to what was originally intended. Writing code that is documented for inheritance and maintenance by others. In my professional experience I’ve run into significant Technical Debt. When I was creating the Data Pipeline for our Workday Data Warehouse we had a number of inconsistencies in the reporting and bugs crop up. Not enough effort was made by our small development team to make the project inheritable and seek additional resources to reduce the internal load. Scope creeped from the initial dataset to include supplementary data points. Additionally, we did not comment and document the metadata of the code immediately so the Organization took more interest in the dataset. Looking back I learned many valuable lessons about Tech Debt from that project. Data Engineering is a dovetail to Software Engineering, and incremental improvements with realistic project pacing and time spent on QA and documentation is one of the best antidotes to experiencing high tech debt.