Project Two: Summary and Reflections Report

When it came to building out each piece of the software and creating unit tests, the number one priority was ensuring that all of my work was aligned with the software requirements. When creating a contact service, I had to ensure that the contact service could add contacts with unique ID’s, that the contact could be deleted using the ID, and that the contact could update all fields except for it’s ID (which should remain static to ensure its uniqueness). All of the fields of a contact had strict limitations on what data types they could be and the string lengths that would be accepted. The fields also must contain some amount of data and cannot be left empty. I wrote my unit tests to specifically test that error catching worked when the fields contained either bad data, null data, or unacceptable string lengths. Once the tests were built and executed, we then checked the coverage percentage to ensure it was meeting a satisfactory standard (in our case, 80% or better).

Code being technically sound can mean a couple of things, but I will assume we are specifically talking about code that is clean (easy to read and follow) and (mostly) error free. The former is not always easy to do correctly, as something that seems easy to follow for you may not, in fact, be easy for a colleague to grasp without guidance. I do my best with commenting (although I’m not the best at this yet) and try to use more verbose naming conventions to ensure that a single read-through is sufficient for a code review. For the latter, efficiency can be broken down into clock cycles and resources invested. Most of the code bases we worked with were relatively cheap on resources (no giant arrays taking up tons of memory) so we can just focus on the time to execute. When running tests, we can see the time taken to run each test which gives us an idea of the performance we could expect in a real-time scenario.

The techniques that I used in this project were covered in an earlier entry, so I will summarize that here. Unit testing has been the primary focus of this project, with some static testing done early on in the project. The static testing needs to be done early to ensure the software we set out to build is consistent with the requirements set forth by our client and to avoid errors costing our client a lot of money in the long run. Unit testing is done frequently at every stage of development to ensure that the software we build not only meets all requirements, but also that it sufficiently catches and handles errors. Unit tests are not perfect and will not catch everything, however, they help us ensure that our software is at a great jumping off point before being integrated into the larger code base. At this point in our project, we would likely move to integration testing and then systems testing. This would help ensure that our software behaves as it should, properly catching and handling any errors or bad data, while also handing off proper data between each piece of the software system.

I had to make several improvements to my mindset in order to succeed in this project. I had to be willing to take more time with each assignment I worked on. I needed to be willing to write out lengthy tests checking every single little thing that could go wrong. Often in my own software I will write all of my error handling, manually test once, and assume it’s good enough for now. When working in a professional setting, this is obviously bad practice and could even walk the line of being considered malicious. I don’t think I had to work much on bias, as the real challenge was committing myself to creating exhaustive tests. Once I’m already working on creating the tests, I’ve already pushed past the bias boundary by admitting that my code may need testing. These improvements I made to my mindset ultimately helped me make small steps towards being more disciplined. While I definitely lacked discipline in getting my assignments done on time, I utilized increased discipline in ensuring my code and unit tests met all requirements set forth. In a professional setting, this will be invaluable as any small mistake that is allowed to pass from development to live could result in irreparable damage to both the client and their users. I would love to say that I can avoid technical debt, but alone I don’t think that is possible. I think in order to truly avoid any technical debt, it’s important to not just create exhaustive tests and commit yourself to excellent quality, but to also ditch your ego and seek code reviews from your peers. It’s ok to be told your code sucks. It’s not ok to potentially bankrupt an entire company and leave others without a job so that your feelings don’t get hurt.