Lessons Learned

Having mostly front-end JavaScript experience, this project held many opportunities to learn. Though it was challenging, I feel like I am walking away with a much deeper appreciation for higher level design applied to software.

I felt really good about the first 3 phases of the plan and only on phase 4 - implementation - did I start to see the problems with my documents. As noted in lecture and the textbook, this is to be expected. My next iteration of design and development will already be much different with all that I learned from this project. I will definitely have a better ‘gut’ feeling for why things might want to act a certain way. For example, the information expert pattern was hard for me to think about in terms of modelling and often became more clear that a method belonged elsewhere when actually coding. That is why my CLI class became so huge, and did too much itself instead of delegating to classes that “knew” what to do.

The patterns I was most drawn to were the facade pattern and high cohesion with low coupling. Those made a lot of sense to me, though I’ll readily admit that implementing them correctly was a challenge.

The feedback for phase 4 noted that there was a lot of really large methods, repetition, and low cohesion with high coupling. After starting to refactor just those specific methods I grew even more unsure of my design overall. Just abstracting the repeated code was not enough. I started to see some patterns in the way the different Domain classes in the system interacted and thought about trying to abstract some of those similarities into a design pattern. It was a bit drastic (more drastic than any production team would want to endure), but I decided to go through with a large refactor of how I was setting up the system. I ended up settling on something resembling MVC, though not exactly. I felt like this offered a lot better results,

particularly in reducing the coupling between the different Domains. The storage of data was moved away from the GymSystem class, leaving that class very lightweight and open for higher level changes and management should that time come.

I am still not entirely happy with this approach, and the class diagram shows some messiness around the way relationships are kept between different classes. I did however, feel very comfortable with the idea of having a separation between the Domain class and the Controller class. This also let me do the same pattern for most Controller classes in the system, allowing me to use interfaces to ensure some consistency. I feel like this has many benefits, including helping new users onboard easily, or adding another Domain/Controller to the system. There is a pattern to follow now, and interfaces to ensure consistency.

I did not end up following the refactoring patterns very well, as I moved and changed so much from my initial design. I needed to change, rewrite, and add a lot of tests, so the typical TDD process didn't work. However, the first few attempts at refactoring were not as drastic and I saw the full benefit of TDD. I was able to change code, and even move some methods around and check my tests to make sure nothing broke. Having Travis CI available to see (and be reminded of) tests not passing was also a great benefit. I will definitely use CI with TDD for projects in the future.

I still had a tough time thinking through all of this and there was a lot of time spent thinking about modelling, instead of actually modelling. Again, I know that this will increase with time, which will be a huge addition to my personal software development toolkit. I'm excited to get the point where modelling itself isn't the challenge, and I can use that skillset to tackle the real challenge of building a system that is well architected. I'm also curious to explore some of these ideas in terms of higher-level architecture like the business-wide collaboration between

different services at work. There is a lot of crossover that could benefit the way we handle the interaction between different team's APIs.