This document is used as an outline of what to talk about in the design video presentation. It will largely comprise of things we touched on in lecture and how we approached things.

The issues with the design and the good things (Using his principles from lecture)

- We want to explain how our design centers around the main software class and that we use the attendants and members as kind of external actors to this system that interact with them (hence why they are separate classes)
- We realize now after watching the design lectures how this was a poor choice, we should have tried to separate our classes to reduce the high amount of coupling we have acquired
- The coupling comes from having the software class containing a large amount of the functionality and all other classes being dependent on it (can be seen from the diagram as it is central)
- This is like design approach #1 you showed us in lecture, where the selectProductController class was heavily depended on by multiple other classes in the system and was central
- However, we do split our software class into lots of methods (this can be seen from the larger diagram with everything included). This helped us separate the different use cases and allowed us to find bugs very quickly, despite all being contained in one larger class. So we used the divide and conquer approach for our methods, but not as much for our class structure.
- In hindsight we should have split this more into different objects as we think this would have been easier to manage and better for testing as stubbing would have been far more useful to have stubs to test certain parts. As opposed to having to use the software class and sifting through the method where the bug occurred. (Not terrible due to our good method design)
- We should have used the principle of information hiding more to help us organize what parts needed to know what about each other. This was something that would have also helped with our coupling problem, because if we had split things up and hidden information that could have served as a good check and balance for our design/implementation.
- In terms of our simplicity, I think our design is relatively simple in that we map out the external actors (members, attendants) and we keep things in a software class that acts as the interface for the user. It seems simple in that it is relatively easy to explain to someone new in the group or a customer as there aren't any complex relations. However, from a maintenance and implementation point of view it could have been more simple we think.
- **Direct calls aren't always simpler (he mentions this in lecture)**
- For anticipating problems we think we did a better job, as we knew the GUI would be coming up and made sure to keep the functionality very separate and this made the GUI implementation far more simple than having to modify our system to handle the new interface required.

Some reasons why we might have done this

- One of the reasons as to why we chose this form of design was due to the changing of groups that we experienced every iteration, which although we really liked as it gave us a chance to work with lots of different people.
- However, it provided a few challenges in having a unified design and so this made it a little easier to use the design we did with lots of methods for each use case. This avoided team members having to wait if other members had done their part before doing more extensive testing. This was due to students having different schedules (literal time differences).
- We also found it hard to get a concrete design down as at each iteration we had different teams with different visions so we had to get each new team up to speed with the chosen code base/design.
- As well we did not know the full extent of project requirements before starting each iteration and the specific cases we needed to handle so this made it a little more difficult to foresee what needed to be placed where in which portion of the design.

Concluding type comments

- Although we should have been able to build a design foreseeable for future implementations, a fundamental redesign at the 2nd iteration would have been a lot of work (possibly worth it) and at the 3rd (it seemed not to outweigh the pros) so we had to make a decision and we choose to stick with procedural type approach as this was currently working for us.
- It helped that this was a relatively smaller piece of software as you allude to in lecture. However, we do understand that if this was to grow it would likely grow out of control from a maintenance point of view.
- This might be a time when we consider a redesign since it would be larger and we would hopefully have more time.

- Overall we think this design certainly has issues specifically for dependencies and coupling of classes. However for our internal design (state and sequence diagrams) we believe we separated our methods for the different use cases and GUI fairly well using divide and conquer type ideas.