Lab 4: Class Diagrams

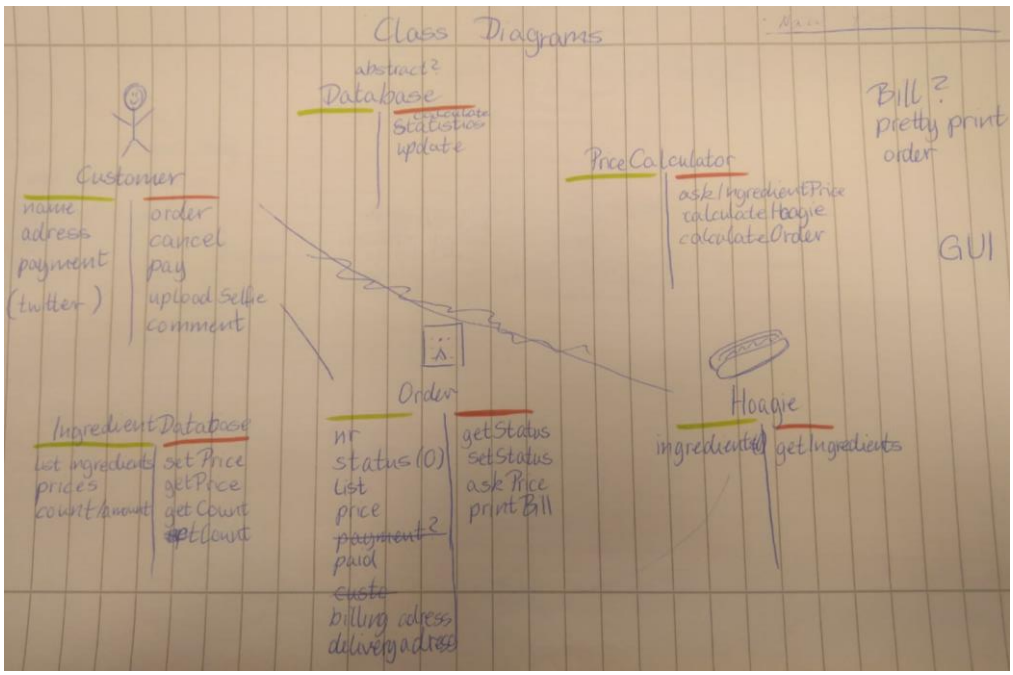
2018-11-19	*Info 3* *Lab Report*	*Johanna Spaude, Sao Chi Pham,* *Florian Murzov-Pirkl*

Review

In this lab we started with getting our Hoagie Shop back from Bernhard Hofmann, Felix Lausch and Jan Schelhaas.
We read through their work on our Hoagie Shop and saw that they already thought in class diagrams the week before. They had been a little bit confused because our use cases were more like sequence diagrams. It was hard for the other group to see the startpoint in our use cases. Of course, there should not be a start point in a use case, but we had some because we had thought of our use cases as sequence diagrams.
The other group also used many different names for the same actors. That confused us sometimes and we had to look twice on some diagrams.
Additionally, they added some actors. For some reason there was a kitchen in their diagrams. We decided to use some aspects of their work, but not all of them like removing the kitchen.

Class Diagram

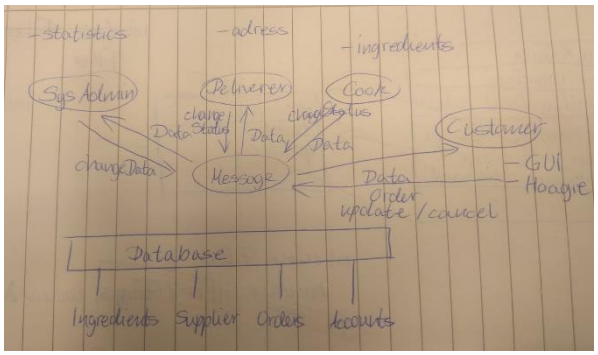
Now we wanted to identify the class candidates, method candidates and attribute candidates in our Hoagie Shop. This step costed us much time, because we were unsure about the architecture we wanted to use later on. And we already thought in the direction of the architecture.
We were brainstorming and came up with the potential class candidates based on our use case “order” (see picture).



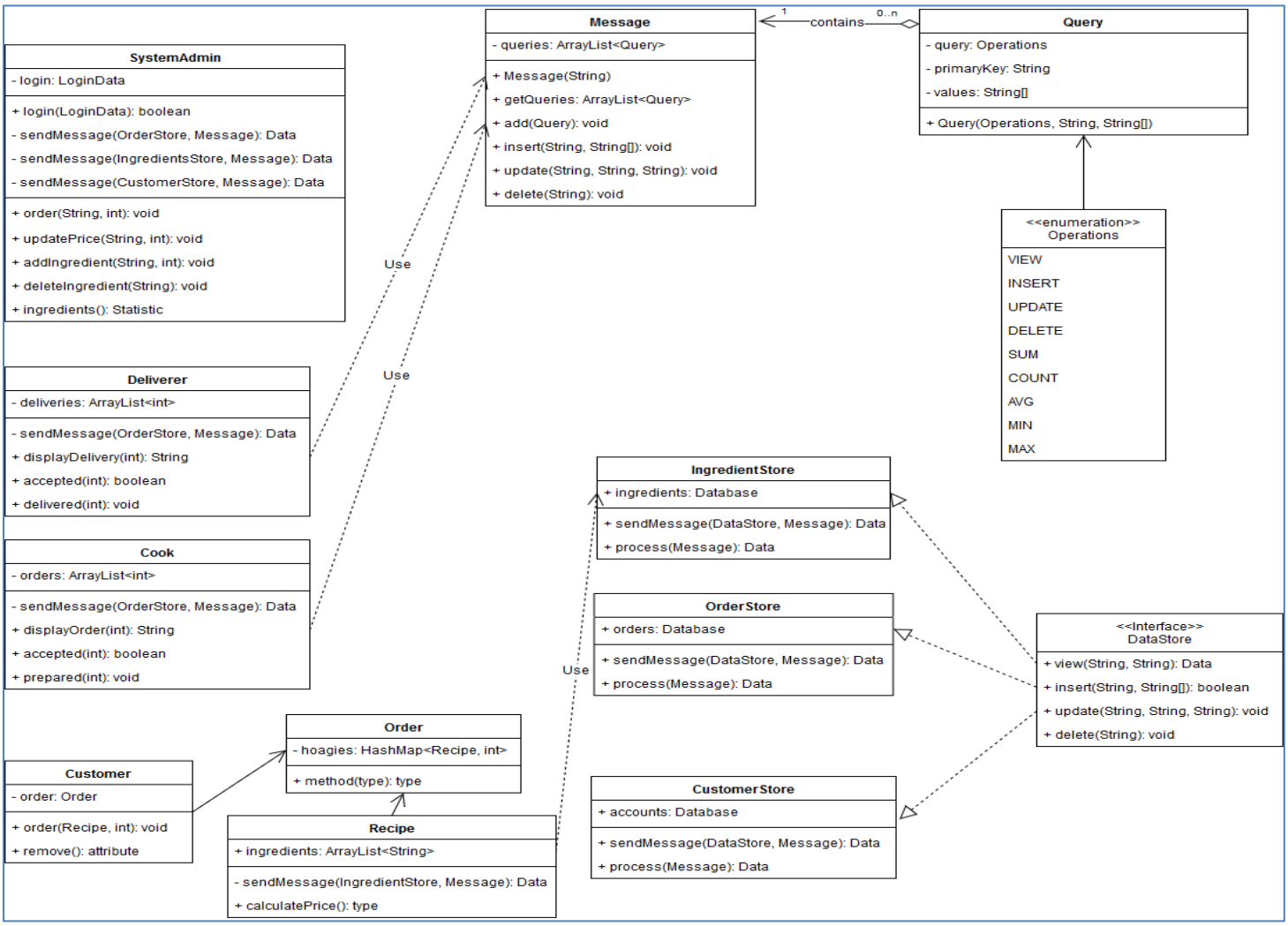
This was pretty much the base of our work the last weeks. But we came to the point when we had to specify where we want to put our data. We always spoke about a System Admin who knew all the things and were able to change and update all the data even on the website. Error handling would also be easy, as we just had to deal with one actor who oversees all things. But how could a Customer interact with the System Admin? Could we place a GUI in between them? Who would do the calculation and organization of the ingredients? Also, the

System Admin? At the end of the first lab, we were not sure about these questions and had to specify our processes and which actors we really needed.
The main problem was that the data would not be properly encapsulated with just adding getters and setters to classes containing data since even less powerful actors could make fatal changes. A big question mark was how we can connect our data with the actors at this point.

Chi came up with the brilliant idea of a database where we put all the data. The actors then would just interact with the database over messages and would not touch the data itself. You can see the first version of this idea on the right-hand side:



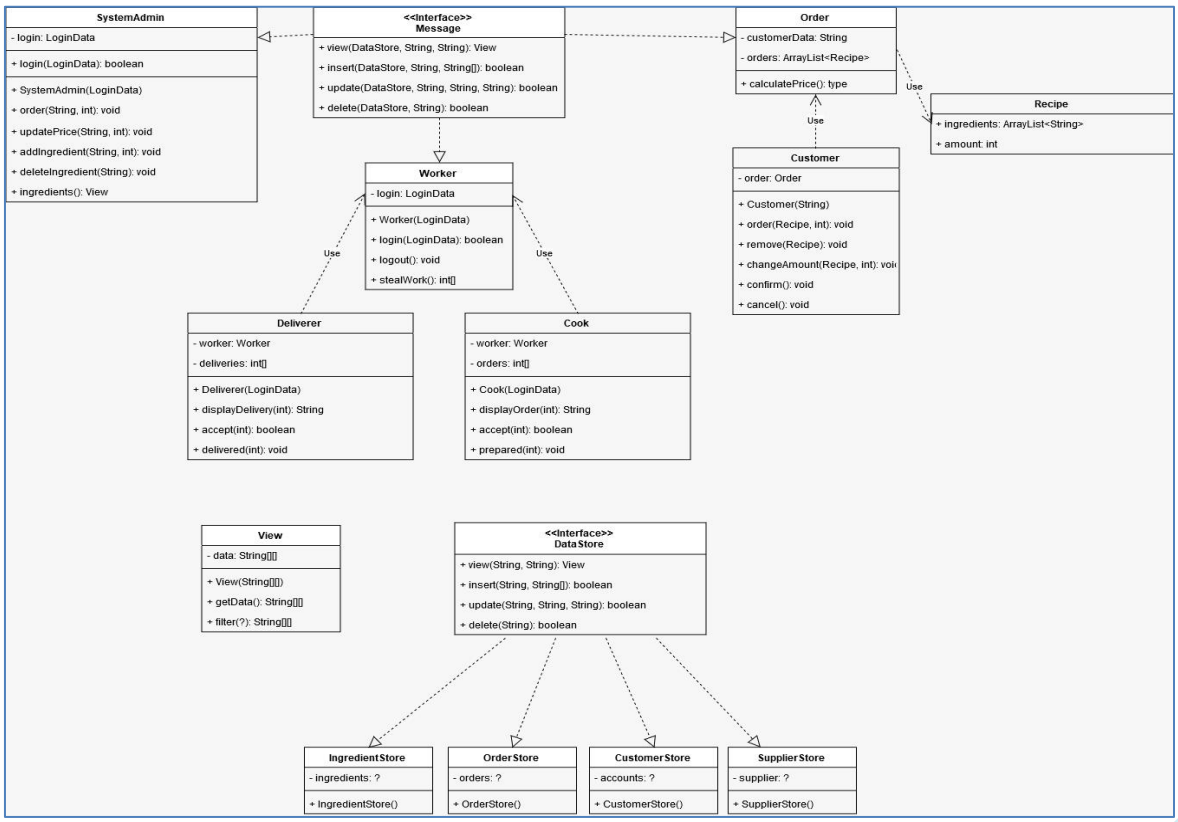
Chi had many ideas and drew different diagrams. This was the first one:



We threw out some actors like the PriceCalculator (just a method now) and the kitchen (from the group last week) and tried to show the connection of the database with the other actors.
In this iteration the actors were literally sending messages as objects. The kind of messages an actor could be send would be defined by us.
However, making messages was ambiguous because in one message could be several queries and how would the database actually process all queries. In a sense the Message class was so powerful but complex that it became kind of confusing and rather a cognitive burden for future development. Most importantly the problem was that we could not get easily data back with messages since those contain only queries and displaying would prove difficult.

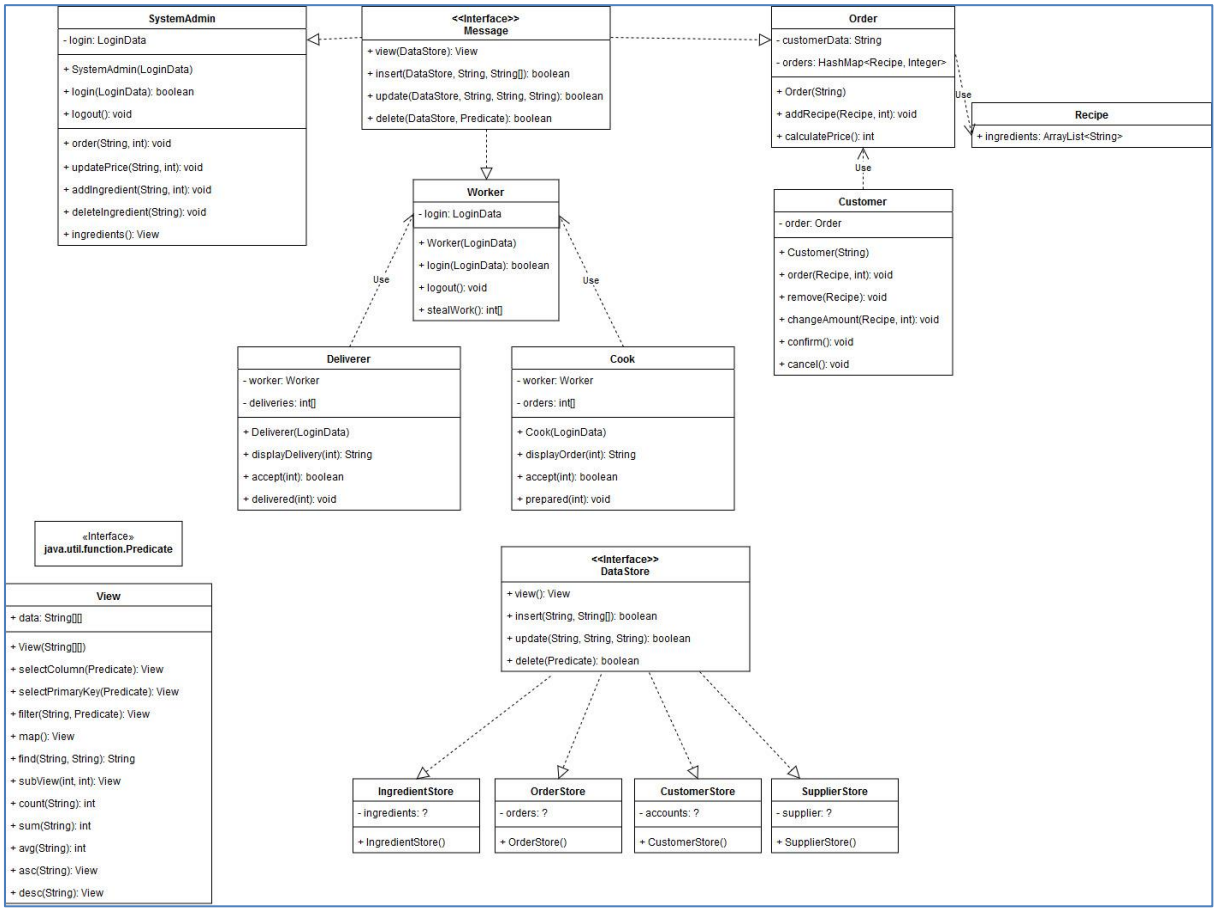
In the next lab, we thought about this idea and discussed the separation of data and actors. One proposed option was making an InputMessage and OutputMessage class, which would be more work, or use an interface.
We could use an interface as the view of the actors on the data. The data then would be stored in a big database. We were not sure about which architecture that would be, but we came to the conclusion that it mostly fits the Repository architecture, since we have shared data, which are held in a central database and is accessed by all sub-systems.
The Message interface would be used from the actors to ask the database for data. A very important point about this idea is, that the actors cannot change data itself but have to ask the "owner" of the data which is why the Message interface requires as one parameter a DataStore. Each actor would have prepared messages (e. g. "I am ready" or "I am waiting for work") defined as their public interface. This would guarantee safe and limited communication. So, we would not have one superclass that could kill everything.

The second one:



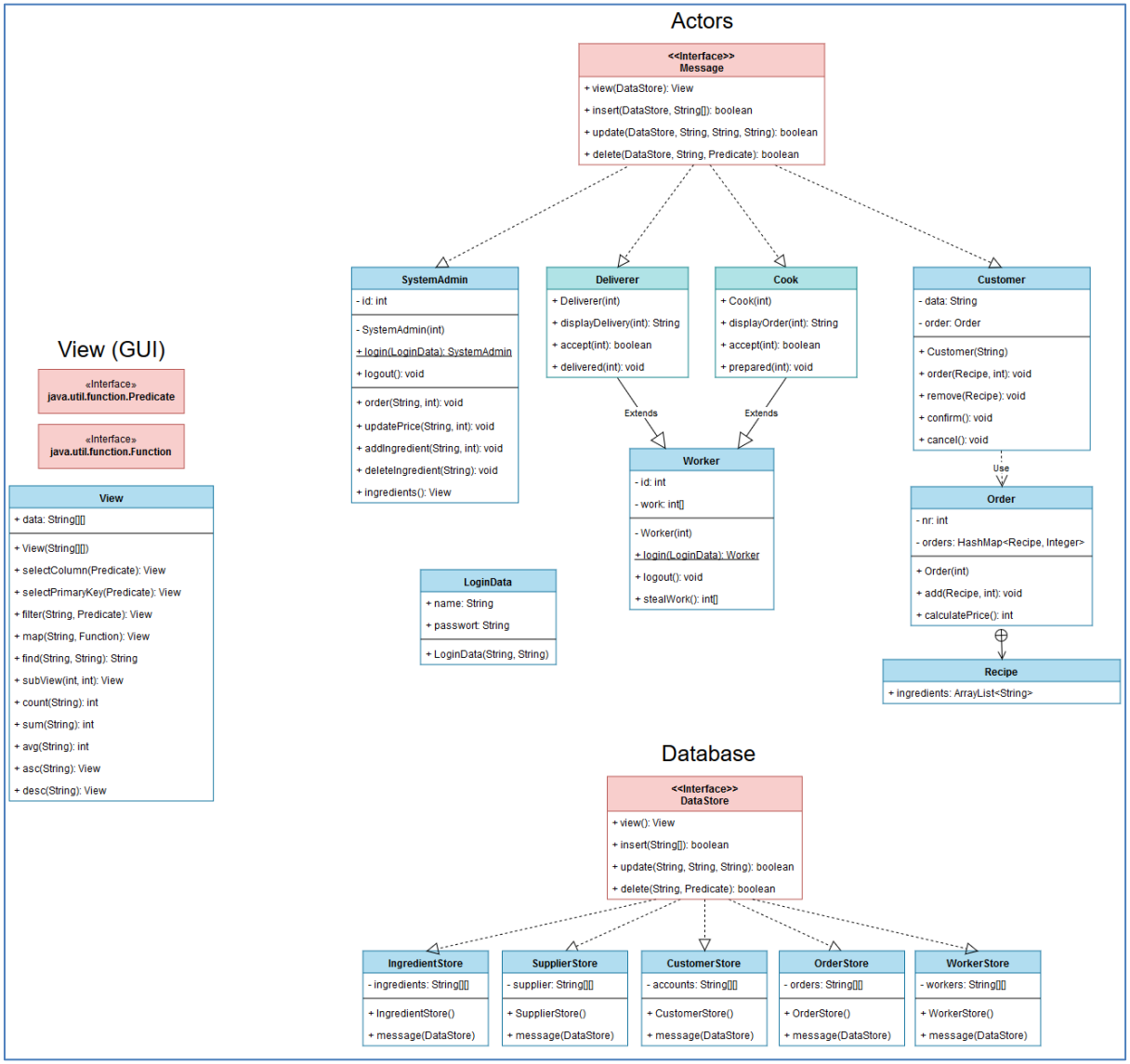
We noticed that the deliverer and the cook had nearly the same methods. So, they became later subclasses from Worker. A Worker can steal work by looking for orders that have not been accepted by another Worker, yet.
The actors interacted over messages and the DataStore was an interface on its own, so we can also limit how data could be manipulated and viewed in the first place. The classes implementing the DataStore interface can be thought of as tables and DataStore groups them to one database.

The third one:



The Cook and the Deliverer itself cannot do anything about the data (add or delete some) but can only use the Worker object. The class Worker implements the methods of Message which will not count as public methods for Deliverer and Cook, so those methods are hidden from the public interface.
View is very powerful in this class diagram since it can be created from the DataStore classes, which means a view on a DataStore object can differ depending on the actor and manipulated from actors.
The Predicate in a message would be a lambda function that takes a boolean and outputs a boolean something like (e.g. `s -> s == "Tomato"`). This can be used to filter out values/rows in a table.

The fourth and last one:



In this last class diagram, you can see clearly who is interacting with whom. The View class has been changed a bit to make it more flexible, so it can be used in our GUI somehow. Additionally, the workers have to actually login and cannot be created directly. They are only created when the login was successful.

Nevertheless, there are still some parts left that we are very unsure about. Should we have different classes for one table in a database or should it be one huge Database class? One benefit of our current implementation is, that it is always clear which message is for which data table but maybe that is achievable with one class.

We have many parameter types that look the same but are different. For example within the method update(DataStore, String, String, String). It would be very nice to see which type exactly we use here and where it comes from. Documentation is sufficient, but it would be better if we could create meaningful data types to make future changes easier. Maybe some type that can be either a String or a Predicate. Furthermore, a String for the data of the Customer is not good.

And last but not least, we have to think about the database implementation. Currently we imagine a table as two-dimensional array for the database. But we are not sure if this is good or even possible this way, we want to use it.

Evaluation

Johanna:
In this lab I was (once again) very happy to work with my partners. Chi had so many wonderful ideas. It was amazing how we came from the first messy solution of classes to the class diagram above. This class diagram forced me to review inheritance and interfaces. I am still amazed about the idea of the database and hope that we can program this in a good way.
Time: I needed the time in the labs plus around two hours to work on the diagrams, understand the diagrams and write the report. (300 min)

Flo:
It was nice to start by thinking about potential class candidates, which I remember well from the first lab last semester. It was a really interesting process so far and all the things we did in the labs before, were very helpful by doing this exercise. Also interesting was, how the other group was looking at our work before and how we used or discarded their ideas. Building up from all the documents before and continue working on the same project is way more enjoyable than I thought it would be. It makes me want to develop it further and further and I am very grateful to be surrounded by such a great team with all their ideas, perspectives and motivation!
Time: Additionally, to the time in the labs, I needed 2.5 hours for working and understanding the diagrams and 1.5 hours for working on the report.

Sao Chi:
I must say I put a lot of time in this lab but I enjoyed it. At the beginning I was quite unhappy how the first iteration turned out. We had good ideas and also had already thought of how things should work, especially that data should be treated carefully, but it really did not show on our first try. So, I tried different things and also incorporated the things I learned in the database classes. With each iteration and discussion it started to looked more like an actual plan. So far, the direction seems good and there are still many things to explore, so it should be easy to make changes.
Time: I needed the time in the lab and then 5 hours for concepting diagrams and writing the report.