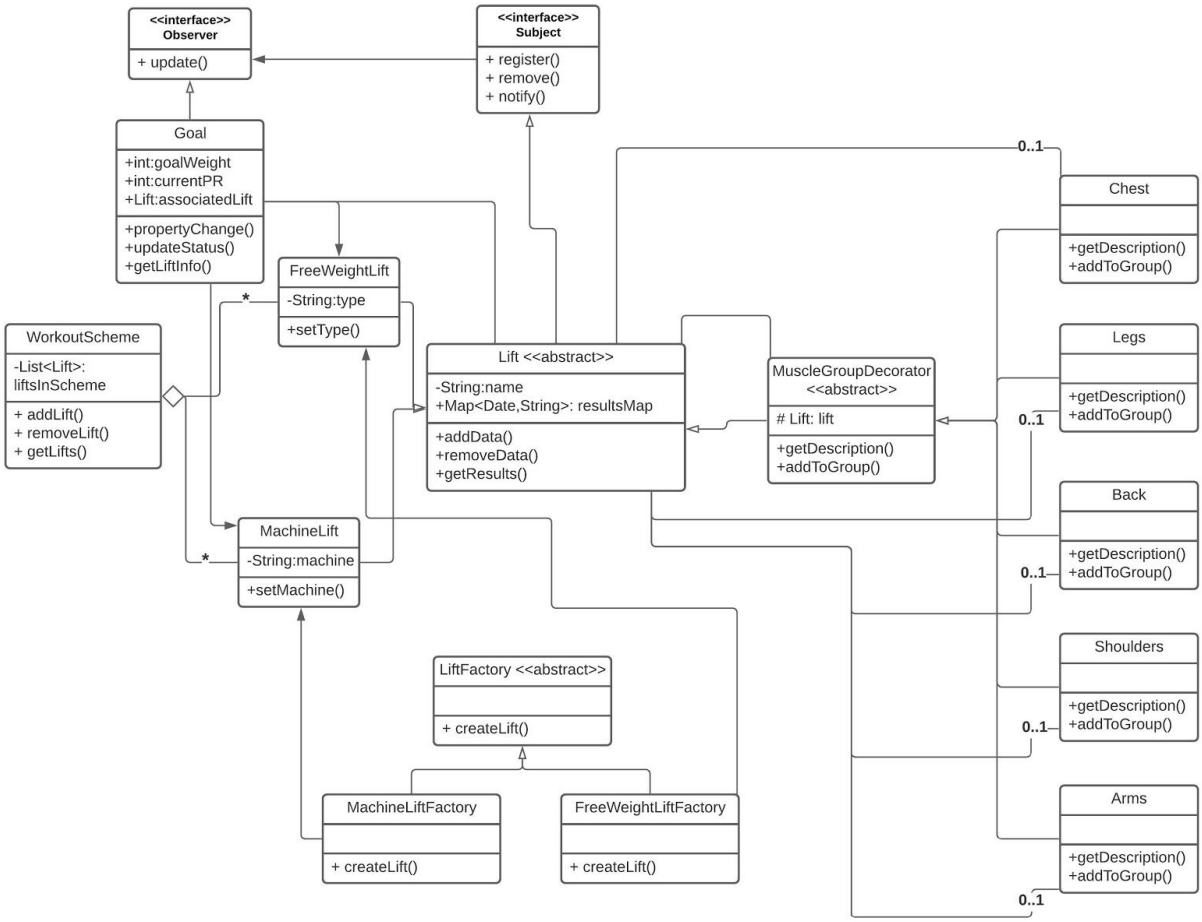
Project: Gains
Team Members:Chace Trevino, Joe Wilson

State of System:
Given our four week timetable, we had a good estimate of how much work we thought we could accomplish. Based on this we defined features that we felt we could include in the final project and we hit the nail right on the head. In project 4, we hoped to build a system that did the following: worked on IOS and android devices, worked offline, allowed users to create lifts, allowed users to create workouts, allowed users to input and track performance history, and support goal setting. In the final state of our system nothing was left out and every one of the features we hoped to include was implemented. Although we were able to implement all of our features, the three patterns we used were not the three we originally planned for. Instead of using factory, observer, and decorator, we ended up using factory, observer, and mediator patterns. Altogether we are pleased to see how our project has developed the past few weeks.

Class Diagram and Comparison:

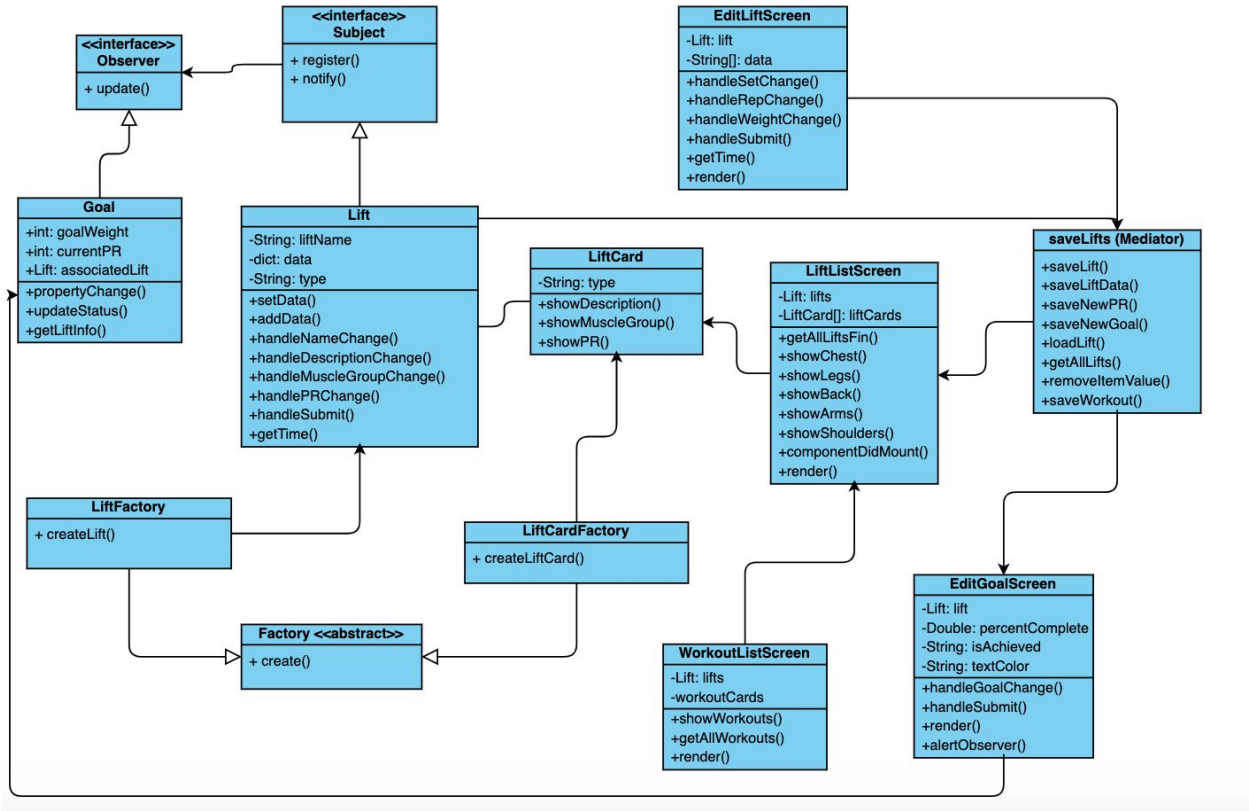
Project 4 UML:



Patterns:

In our final system design we used the following patterns: observer, factory, and mediator. When a user is inputting data about their most recent performances on lifts, we used the observer pattern to notify the user if their new pr completed their goal that they had set for that lift. The lift was the subject class and the goal was the subscriber class. Since many different lifts are being created in this app, we used the factory in order to write decoupled code when creating the lift objects and the lift cards that displayed their data. Lastly, we used the saveLift class as the mediator pattern. None of the classes know the other exists as they only need to communicate back and forth with the saveLift class.

Project 6 UML:



Since Projects 4, we took out the decorator pattern as it didn't fit with React Native's framework. Instead, we used a mediator pattern. We no longer needed to decorate lifts with different muscle groups, so the muscle group classes are not on the final class diagram. We also decided not to differentiate between machine weight lifts and free weight lifts as they were virtually the same class. Since Project 5, we have added in the mediator saveLifts class, which allows the edit lifts and edit goals features to communicate with each other through that saveLift class. We also added the WorkoutList functionality which allows us to compile liftCards into a workout.

Third Party Code vs Original:

Factory Pattern Classes: Although we knew how the factory pattern worked, implementing it in react native works a little differently. We referenced this tutorial on how to create a factory mapper and factories for specific classes.
https://medium.com/mop-developers/factory-pattern-in-react-native-without-using-switch-df99bca31a55

Save Lift Class: We chose AsyncStorage as our persistent storage solution but were unsure how to use it. We used this website to help learn how to interact with the AsyncStorage api that react native provides.
https://whatdidilearn.info/2018/11/25/local-data-persistence-in-react-native-using-asyncstorage.html

General React Native Components: React Native supplies many components that can be used in the html views of the ui. These are pre packaged components that can be included in the package.json and injected right into the project's html. Some of these components used in our projects are FlatList, Touchable Opacity, ScrollView, and Button. The link below lists all of React Native's built in components.
https://reactnative.dev/docs/components-and-apis

Besides what is noted above, all of the other javascript logic, classes, and html is our own group work.

OOAD Process Elements:

1. Use Cases: This design process made us think harder into how a user would interact with the system. We laid out how the user's interactions would lead to certain actions. By doing this, we were able to see classes form and the responsibilities each class would take. This was a very positive process because it kept our scope very clear and we knew exactly what we wanted the user to be able to do.

2. UML Class Diagram: We really enjoyed the aspect of planning out the program before coding using UML. This allowed us to carefully think how classes would interact and work together to create one cohesive program. This activity provides a high level of abstraction so that we can see the flow and connections of our program in a simple way. By doing this, we saved time at the beginning stages of implementation since we already had an idea of what we wanted to do and how patterns could be utilized. As seen from earlier, this changed a lot for us, so it is important to keep in mind that these plans are not set in stone.

3. UML Sequence Diagram: This was the most granular design process element that we did as it felt like a very abstract pseudocode. It was interesting to think about at what time objects would be created and what action that object would be called to do. We

probably could have done without this portion of the planning just because it is very hard to predict the sequence of how things will work without actually coding it. Overall, it was a good exercise but was not super helpful in the implementation phase.