Project part IV – Cab Sharing Final Report

Mohammad Hashemi -CSCI 5448
Praveen Kumar Devaraj -CSCI 5448
Nachiket Bhagwat -CSCI 5448

1. **What features were implemented?**

We have implemented the following use cases:

- UR-001: Passenger can request for a cab immediately and at a specific time using the client application on android.
- UR-002: Passenger can specify the details used to group them while sharing cab on the client application on android.
- UR-003: Passenger can cancel an active trip on the client side.
- UR-004: Feedback on driver module is implemented on the Server end.
- UR-005: Passenger can select the number of passengers accompanying him/her.
- UR-006: Driver can select a ride.
- UR-007: Driver can start and end trip on the client.
- UR-008: User can signup, login and logout from the system.
- UR-009: User can see all active bookings.
- UR-010: Driver can view all new group bookings
- FR-002: Location access is enabled on the client application.
- NF-001: The User Interface is intuitive
- FR-003: System can find user availability in close locality.
- FR-004: System groups passengers travelling towards the same destination from close pickup locations.

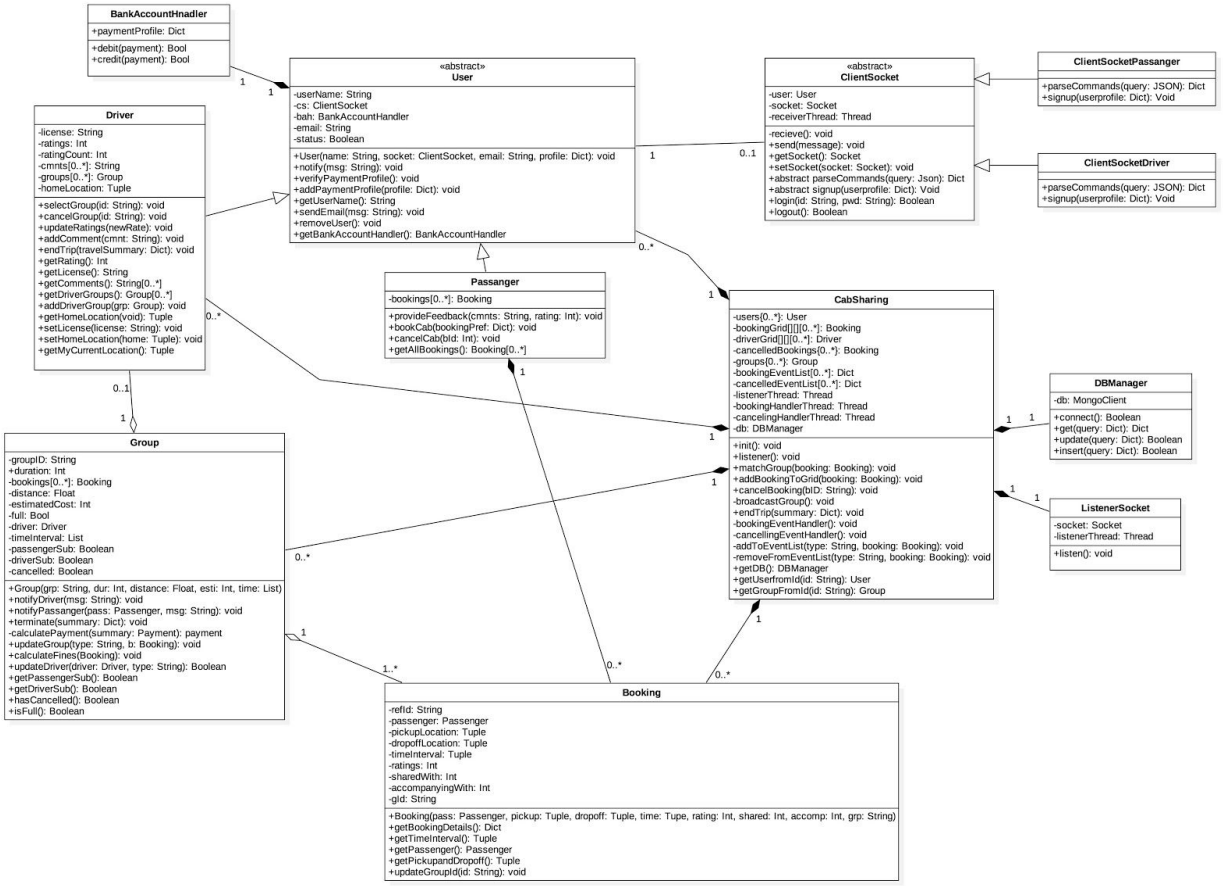
2. Which features were not implemented from Part 2?

The following features were not implemented

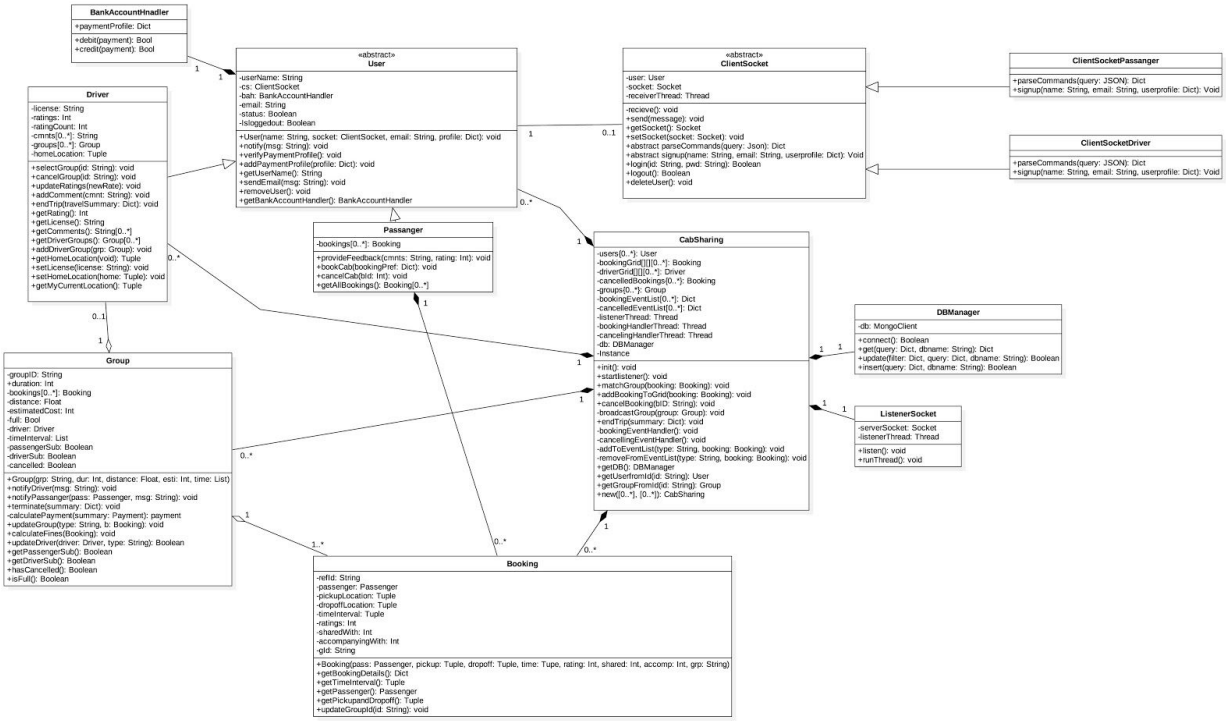
- UR-002: Passenger cannot specify filter criteria based on drivers ratings (like someone with rating greater than 4).
- UR-004: Feedback from the passengers is not implemented on the client
- UR-003: Cancellation of an active trip on the server side is not supported.
- UR-007: End trip on the server is not implemented.
- FR-001: Payment profile cannot be verified by external gateways.
- FR-005: Computation of Peak hour charges is not done.
- FR-006: Payment processing is not implemented with external payment systems.
- FR-009: Estimation of fare is not implemented during booking.

3. Show your Part 2 class diagram and your final class diagram. What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

Old Server Class Diagram



Final Server Class diagram



Changes in final class diagram:

- We added a public method in Listener Socket class known as runThread(), which is called from CabSharing Class to start a new listener thread. This was missing and unidentified in part 2.
- DBManager insert/update/query methods need dbname to distinguish between “Users Collection” and “Bookings Collection”. We have added it in method definition. We missed this in part 2 as we thought we could use only one collection in the database.
- ClientSocket has a new method called removeUser to remove a Blocked User. We wanted to consider the failure cases of payment and update the changes in the database. This was not considered in part 2.
- “signUp” method in ClientSocket class needs 2 more parameters - “user name” and “email id”
- User class has a flag “IsLogggedOut” as a guard clause.
- CabSharing class “Broadcastgroup()” method needs “Group” object. This is because we changed our notification model so that it can support Observer Pattern in future.
- CabSharing class has new attribute called “Instance” and new method called “new”. We needed this to implement Singleton pattern in python.

Designing upfront helped a lot because we had good understanding of whole system before implementing the classes. We could also parallely work on implementation of different modules as we didn’t face many conflicts. Some of the changes are because Python has certain way of implementing Object Oriented design. Some of the changes were internal changes where we didn’t understand the need of the changes before implementation.

4. Did you make use of any design patterns in the implementation of your final prototype? If so, how? If not, where could you make use of design patterns in your system?

We used 2 Design patterns while implementing our project: Singleton, Proxy.

- The Singleton design was used for implementing the CabSharing class. CabSharing groups the active passengers based on the grids which is a single object for the whole system.Thus Singleton design pattern ensures that there would be only one instance of the CabSharing, regardless how many users have registered.
- For accessing database, we use pymongo library to connect to the MongoDB. We use Proxy design pattern in DBManager class to do input validation before using pymongo library.

We can use 2 other design patterns.

- We can use Observer Design pattern to let the server to broadcast the full groups of passengers to the drivers in same locality as passengers and then drivers can select a group.
- We can use Strategy Design pattern to calculate the payment method for different countries as each country has its own taxes, currencies and fare computation methods.

Our system is based on a Client-Server architecture pattern where the client is on android and the server is built using python and Mongodb.

5. What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?

While working on lengthy projects, we should have concise idea of what system requirements are, who are the stakeholders and what are priorities and for every requirement. This is how we define our system. Requirement gathering is crucial step in designing new system because it is difficult to add functionalities seamlessly when design and implementation already started.

By creating UML diagrams, we were able to see problems in our use cases. Eg. From use cases diagram, we found out passenger and driver have many same functionalities and some differences. This helped in creating abstract classes and deciding where we can use polymorphism.

Class diagrams was the lengthiest part in our design. While creating class diagrams, we understood most functionalities in our system. We decided to have grid system for passengers and drivers. We also found that some classes had no definite functionalities and we could have moved some functionalities to other classes for strong cohesion and weak coupling. This changed our class definitions a lot.

Activity and sequence diagrams were important to see flow of our system. For synchronization, we found out that we need to have event lists for notifying actors which get affected in the group. This changed our class diagram again as we to add event list for passenger and driver. Also, from sequence diagrams, we found that we need to have constructors for some classes.

Class, activity and sequence diagrams helped us understand our system a lot. We found out that we needed to implement design patterns such as Singleton pattern for shared grids, Proxy pattern for database validation and Strategy pattern in many places where we had different strategies for calculating payments. We are using Publish-Subscriber model to reduce number of notifications from system. From diagrams, we could see that we are using Observer pattern.

Going through the process of design and analysis before starting implementation, we saved a lot of time as whole team had same vision for the project and in case of doubts, we could fall back to our diagrams. Still, while coding we found out there are some changes needed in classes for some functionalities. These changes were minor and it did not affect the system definition.