Task Three Write-Up

By Group 7:

Members: Dan Daskivich and Michael Hanna

1. **Provide a README section for creating and running the project. Provide complete specifications for building your project. Specify all libraries, software, etc. needed to run the application. Specify expected installation directories. If you use a specific technology for the project, the technology’s download page must be listed.**

README:

The MySQL script file “daskivichhannafinalproject.sql” and the following.java files (contained within the “src” folder within the compressed “daskivichhannafinalproject.zip” folder) will be necessary to create the “project” database and launch the AutomonusRideSharing application.

AbstractPanel.java

BookOneWayPanel.java

BookOneWayTableModel.java

Constants.java

CreditCardsPanel.java

CreditCardsTableModel.java

PersonalInfoPanel.java

PersonalInfoTableModel.java

ProjectRenderer.java

RegistrationsPanel.java

RegistrationsTableModel.java

TripHistoryPanel.java

TripHistoryTableModel.java

VehicleHistoryPanel.java

VehicleHistoryTableModel.java

VehiclesPanel.java

VehiclesTableModel.java

View.java

Run the full “daskivichhannafinalproject.sql l” MySQL script to create the “project” database. This script creates all tables from the schema as outlined in the EER diagram included in this report. The “daskivichhannafinalproject.sql” script also creates starter tuples for the Person, Admin, Users, Owners, Customer, Make, Model, Type\_Of, Years, Model\_Year, Vehicle, Registration, and Credit\_Card tables. However, the “daskivichhannafinalproject.sql” script does not create any tuples for the Trip, Service, One\_Way, Personal, or Public tables; tuples for these tables must be created through the application user interface. Inserts for these tables rely heavily on inputs calculated by procedural code, so we decided to leave these out of initial tuple creation.

The View class, defined in View.java, contains the main() method that launches the AutomonousRideSharing application. Before compiling the View class (or before creating a .jar file), you will have to edit a few lines of code within the View.java file to provide this class with the URL, username, and password of the MySQL server you’ll use for the database. Specifically, on line 16, update the value of “DB\_URL” if the URL of the MySQL server you’re using is not “127.0.0.1:3306”. On line 17, update the value of DB\_USER if the username you’re using to access your MySQL server is not “root”. On line 18, update the value of DB\_PASSWORD if the password you’re using to access your MySQL server is not “password”.

**16 final private static** String ***DB\_URL*** = **"jdbc:mysql://127.0.0.1:3306/project"**;

**17 final private static** String ***DB\_USER*** = **"root"**;  
**18 final private static** String ***DB\_PASSWORD*** = **"password"**;

Finally, before creating the .jar file or running the View class from within an IDE, you’ll have to add “mysql-connector-java-5.1.42-bin.jar” to the classpath.

Once these steps have been completed, you should be able to create and launch a .jar to open the AutomonousRideSharing application (or just run the View class from within an IDE).

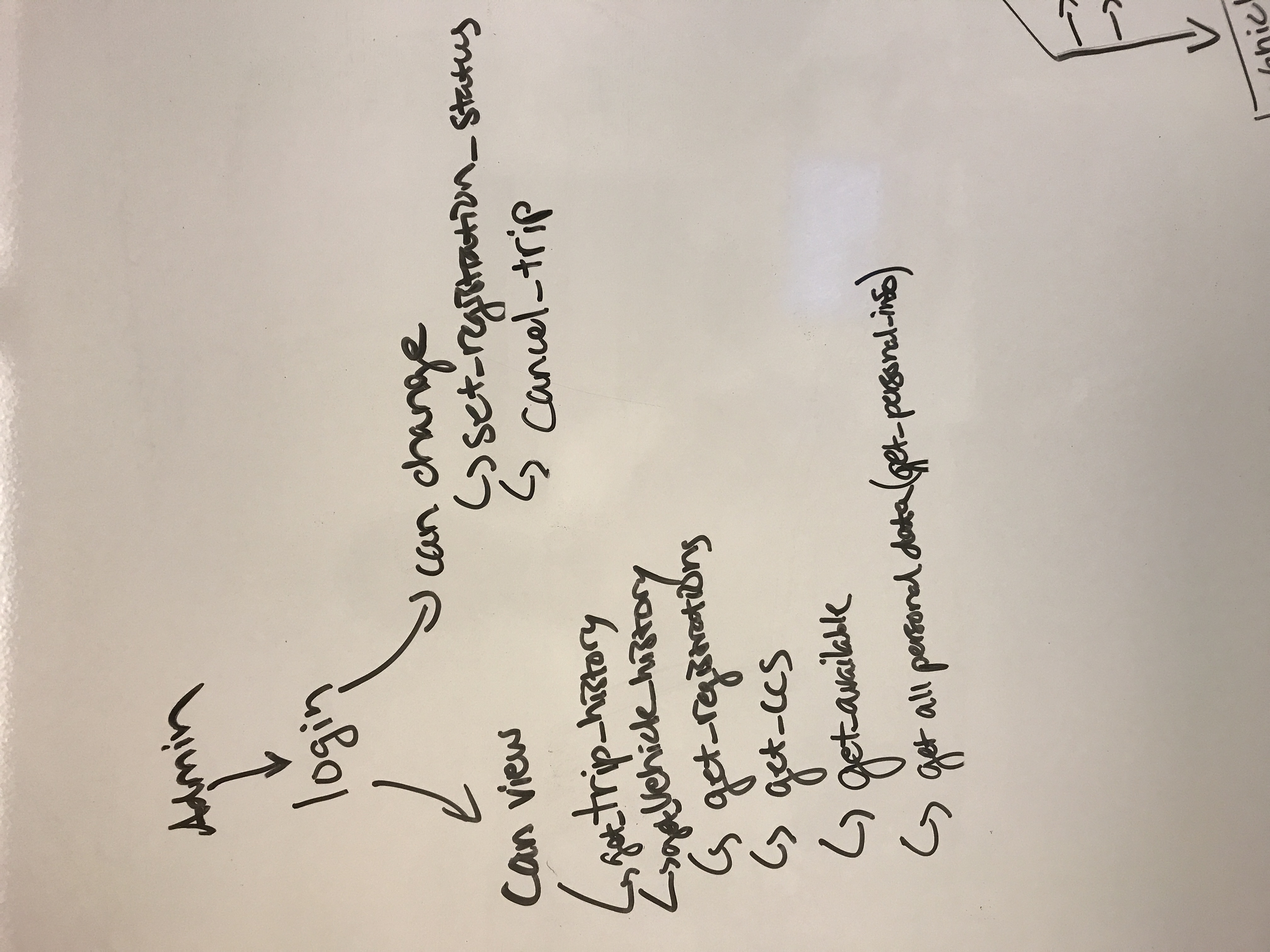
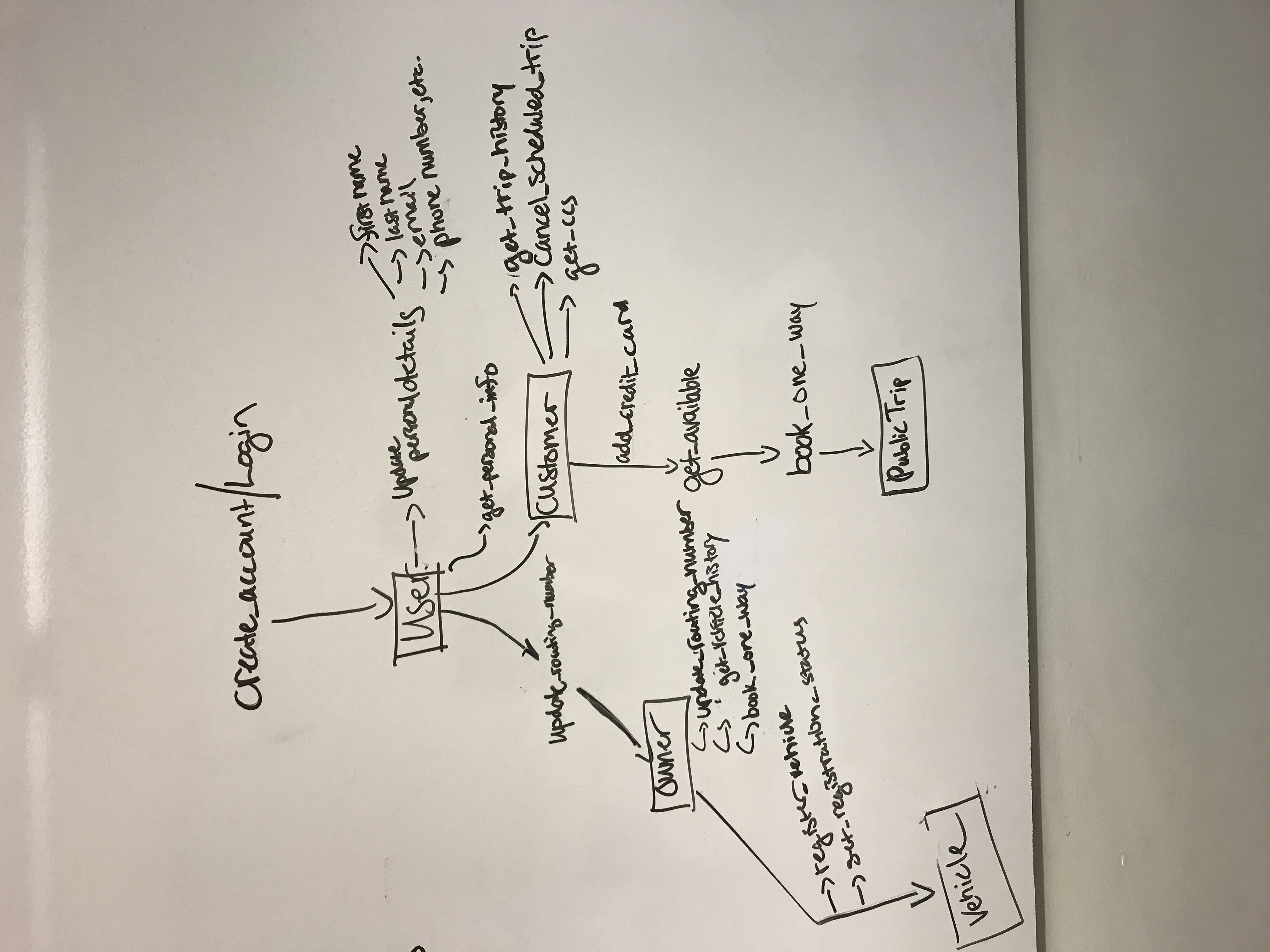
1. **Provide the Technical Specifications (as defined in the progress report) for the project.**

We used the relational database, MySQL (operated through MySQL workbench) for the backend, and Java (utilizing Swing) to create a functional frontend. The way we designed our schema allows for future expansion to incorporate Google Maps (an ambitious milestone we had set for ourselves if we had extra time).

1. **Provide the final UML/EER for the project or the submitted database schema (ReverseEngineer your final schema in the MySQL workbench).**

Macintosh HD:Users:hannam:Downloads:ProjectPDF.pdf

1. **Provide the final user flow of the system. List the commands or method the users performs to interact with the system.**



* **Procedure: get\_person\_id**
  + Given an email address and a password, if correctly matched within the system, returns their corresponding person\_id.
* **Procedure: is\_admin**
  + Given a person\_id, returns true if person is an admin, false otherwise.
* **Procedure: create\_account**
  + Creates an account for a new user (inserts data into the needed tables). If email address is present, a new user is not created, ‘user already exists’ message thrown.
* **Procedure: get\_personal\_info**
  + Returns the personal info, given a person\_id. If the person\_id belongs to an admin, all user data is returned, else just the specific person’s data is returned.
* **Procedure: update\_first\_name**
  + Allows one to update the first name of a given user.
* **Procedure: update\_last\_name**
  + Allows one to update the last name of a given user.
* **Procedure: update\_email**
  + Allows one to update the email of a given user.
* **Procedure: update\_password**
  + Allows one to update the password of a given user.
* **Procedure: update\_phone\_number**
  + Allows one to update the phone number of a given user.
* **Procedure: update\_routing\_number**
  + Allows an owner to update his/her routing\_number
* **Procedure: get\_makes**
  + Returns all the makes in the database
* **Procedure: get\_models**
  + Returns all the models in the database
* **Procedure: get\_years**
  + Returns all the years in the database
* **Procedure: add\_vehicle**
  + Adds a vehicle to the database; will only work if the owner who the vehicle belongs to, has a routing number in the system.
* **Procedure: get\_vehicles**
  + Returns all the vehicles that belong to a given owner (or all of the vehicles registered in the database if the admin calls this query).
* **Procedure: get\_registrations**
  + Returns all the registrations belonging to a specific owner (or if the user logged into the system is an admin, return all registrations).
* **Procedure: set\_registration\_status**
  + Allows an owner to change a registration status of a vehicle to ‘INACTIVE’ or ‘PENDING’, and an admin to change freely between ‘ACTIVE’, ‘INACTIVE’, and ‘PENDING’.
* **Procedure: add\_credit\_card**
  + Allows a user/customer to add a credit card to a system.
* **Procedure: get\_credit\_cards**
  + Get all credit\_cards associated with a given user (admin accesses all cards).
* **Procedure: get\_trip\_history**
  + Get all trips associated with a given person; admin gets access to all trips.
* **Procedure: get\_available**
  + Given potential trip details filled by a user, queries the database for all vehicles which have an ‘ACTIVE’ registration\_status in order for the user to be able to vehicle to book for a trip. Returns details about the trip, car, cost, pick-up and drop-off time, etc.
* **Function: baseRateCalc**
  + Calculates the baseRate for a given user. As of now it’s a simple procedure which only differentiates itself based on if the user is the owner of the vehicle (no cost) or not.
* **Function: costAlgorithm**
  + Calculates the cost of trip, using modifiers (year of the car, and type of car (Sedan, SUV, Luxury)).
* **Function: pickup\_time**
  + Calculates the time of pick up for a potential trip. This calculates regardless if the vehicle is in a trip or not (i.e. this is to say, for a vehicle in the midst of the trip, it will include finishing it’s current trip as part of it’s estimated pickup time).
* **Function: get\_trip\_end\_time**
  + Calculates how many hours, minutes. and seconds it takes to finish a trip, and adds this to the start time of the trip, ultimately estimating the arrival time of any trip.
* **Function: distance\_between\_points**
  + Calculates the Euclidian distance between two points. This is a simple placeholder to calculate distance but would ideally be replaced by Google Maps data.
* **Procedure: book\_one\_way**
  + Books a one way trip.
* **Trigger: email\_insert\_trigger**
  + Does not allow for insertion of emails to the database that do not include an ‘@’ sign.
* **Procedure get\_vehicle\_history** 
  + Gets the history of the vehicle’s trips for a given owner (if it’s an admin, gets the history of all vehicles)
* **Procedure: get\_ccs**
  + Gets the customer id, credit card type, and credit card number for a given customer (used to build drop down menu to select a credit card to be used as payment for a ride).
* **Procedure: cancel\_trip**
  + Allows a customer to cancel a trip for which he/she has not yet been picked up for (i.e. start\_time is in future).

**5.   Provide a “Lessons Learned” section that contains at least 1,4 of the following list:**

**1.   Technical expertise gained**

Regarding the backend, we both felt a great increase in familiarity with the design process. It was a good learning experience to start from an idea, and grow it from the bottom up (normalization to 3NF, superclass/subclass, etc.). In addition to the added familiarity with the design process, a great benefit was earnestly added practice utilizing the syntax of MySQL to execute procedures, functions, triggers, and handle errors. This in the context of a self-directed, real world topic, established confidence to the lessons we learned throughout the class. One particularly proud moment was when we created the ‘get\_available’ procedure. This procedure queries the database to gather all available cars for a user to book a ride. It was an amalgamation of the various topics we had talked about within course (including joins, calling functions within the procedure, utilizing variables, error handling, etc.), which was beyond the scope of what most of the homework assignments required of us to do.

On the front end, it gave us an opportunity to gain added familiarity with Java Swing. **(Dan can maybe insert more here),** *and gave us a chance to learn how to establish communication between Java and MySQL.*

**2.   Group work insights, time management insights, data domain insights etc.**

As both of us are participants in the ALIGN program, we have had limited exposure to coding compared to many of our peers. For this reason it was perhaps uniquely useful to have a final project where we worked in pairs to accomplish a task that incorporated both a frontend and backend. In the instances when we split to work on the separate ends of the project, we had to be in constant communication with one another about each ends’ respective needs. The better the communication, the smoother the integration between the two ends was. It’s easy to imagine how quickly the amount of work would add up in situations where communication was lacking (e.g. the backend user only gives access to ‘x’ amount of fields, when the frontend, needs ‘y’ amount or vice versa).

Being able to collaborate with a partner also proved to be an effective means of getting through subsets of the project that were particularly challenging (e.g. a fresh pair of eyes for debugging, deconstructing complicated procedures, etc.) Additionally, this pair project provided an opportunity to learn a new approach to problem solving (through observation of how each partner approached the situation).

In regards to time management, working on this project reemphasized the importance of starting early on any given task within CS. It’s seems particularly evident (at least for this writer) that within the field of CS it is hard to accurately estimate the amount of work a certain task will take. The debugging process can take an indeterminate amount of time, as unintended interactions will inevitably surface (e.g. what if an admin tries to book a trip?). Many of these cases, are hard to realize until you’re actually interacting with the software.

**3.   Realized or contemplated alternative design / approaches to the project**

**Derivable**

We constructed our design to easily incorporate derived values. A few of our key fields (distance, cost, rate, estimated pickup and drop off times) all at least in part relied on derivable values. Rewards, had we gotten a chance to incorporate such, would have been derived as well (possibly from total miles taken or number of trips taken).

**Superclass/Subclass**

We chose to incorporate two instances of the superclass/subclass relationship (the person to user, user to owner, and customer, as well as the trip and its children). We chose to do this instead of keeping the table unified (i.e. having a larger ‘person’ entity with fields that specified role(s) amongst all the other unshared fields (e.g. customers don’t have routing numbers)). Despite the appeal of perhaps simplifying our query statements, this would have introduced a large amount of null statements in our code, and would have added additional work (and room for human error) to be done in terms of enforcing the constraints in an explicit manner. Choosing to use the superclass/subclass structure allowed us to have the structure of the database implicitly enforce constraints at the table level.

**Further Table Level Constraints**

As one can see in our UML diagram, we also chose to separate the potential attributes of vehicle into make, model, model-year (which shared attributes of the relationship), type, and year. We did this to A) further normalize our design, B) to allow for tables to naturally enforce limitations as opposed to having explicit constraints to not allow a user to submit an invalid vehicle (i.e. preventing incorrectly pairing a make with a model it does not make, e.g. Tesla Corolla), and C) to allow for effective pulling of data from the database (e.g. query, wishing to compare the miles driven by a Tesla Model S to a Toyota Camry).

**Trip Superclass/Subclass versus Log**

We had contemplated simplifying our trip entity and it’s corresponding children into one structure, and replace in its stead a separate log to simplify our design. We chose not to at this time due to wishing to be able to more easily accommodate new types of trips in the future that may easily have different needs (e.g. already, the planned feature of a ‘Service’ Trip is a type of trip with distinctly different needs than One-Way trips). Another benefit of choosing this structure allows us to more simply flesh out our cost algorithm. We could relatively easily add baseRate modifiers (as we did for type of car (Type\_of) and year of car (Years)) in each type of trip, to alter the costs of each trip.

**Error Handling**

We chose to handle the majority of our error handling related to the data in MySQL to better increase our familiarity with doing so, even in some instances where we knew it would be better practice (in theory) to do so from the UI (e.g. the trigger for the email address having an @ sign).

**4. Document any code not working in this section**

I believe all our included code is functioning as intended. We did not get a chance to include our book Service Trip and Rewards use cases, so those have been moved to future work.

1. **Provide a “Future work” section**

This project is far from market ready. In addition to our need to further develop this project to include the planned use cases of service trips and a reward system, the priority would be to integrate our project with Google Maps. We believe our design is structured to allow for integration with such as our our database incorporates the equivalent of the longitude/latitude coordinate system Google Maps has support for. The next step would likely then be to develop the UI to be accessible from a mobile platform, and to incorporate GPS functionality to allow users to more easily establish their location (and vehicles to be able to deviate from their last destination). Further use cases would be scheduling varying types of trips (e.g. upon collaboration with independent businesses, we could allow the car to do simple errands such as pickup groceries or food from restaurants), giving purpose for the ‘Preferred’ customer status, and allowing for both owners and users to book trips for future scheduled times (as of now it’s simply on-demand use).