**Design-II**

**Paul Blackburn, Angela Mattamana, Charlie Nguyen, Michael Richter**

**UI Design**

**Requirements without front end component**

**Use Case 4: Driver is speeding**

This does not have a UI component as this is something that is dealt in the backend. If the driver is speeding at any point, the backend will automatically set a flag on the driver’s profile which may lead to a suspension.

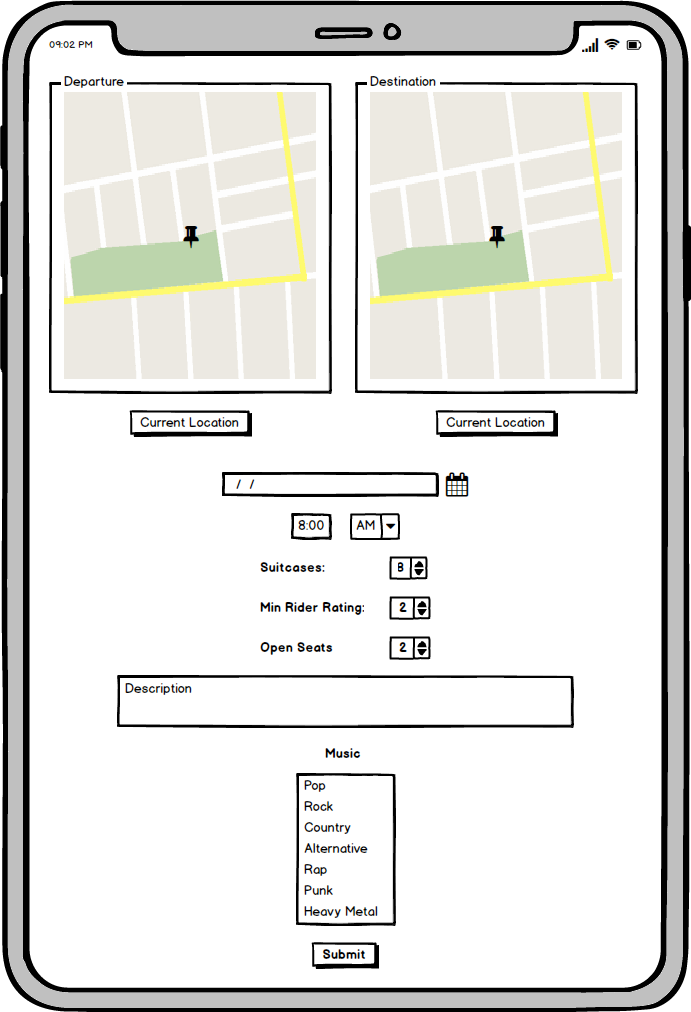
**Use Case 12: Remove accounts of riders and drivers who have graduated or engage in unsafe behavior**

This does not have a UI component as this is also dealt with on the backend. The backend does a nightly job where it checks all the accounts of users and suspends or flags any account that has a low rating, has graduated, or has an incident reported against them.

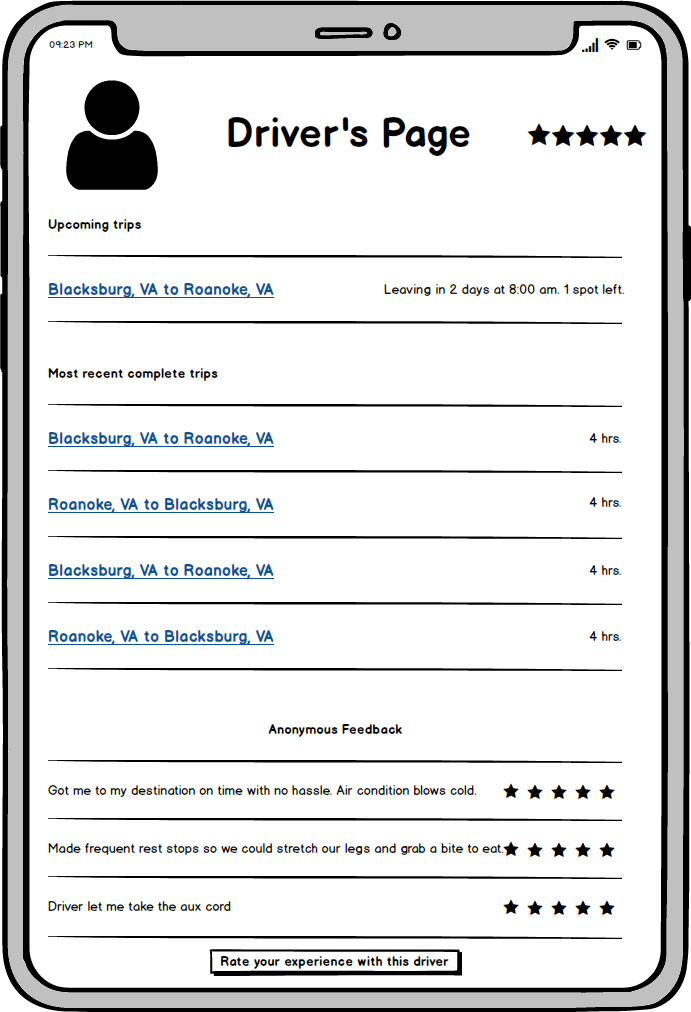
**Use Case 15: Validate payments**

This does not have a UI component as the validation of payment is done through their respective systems.

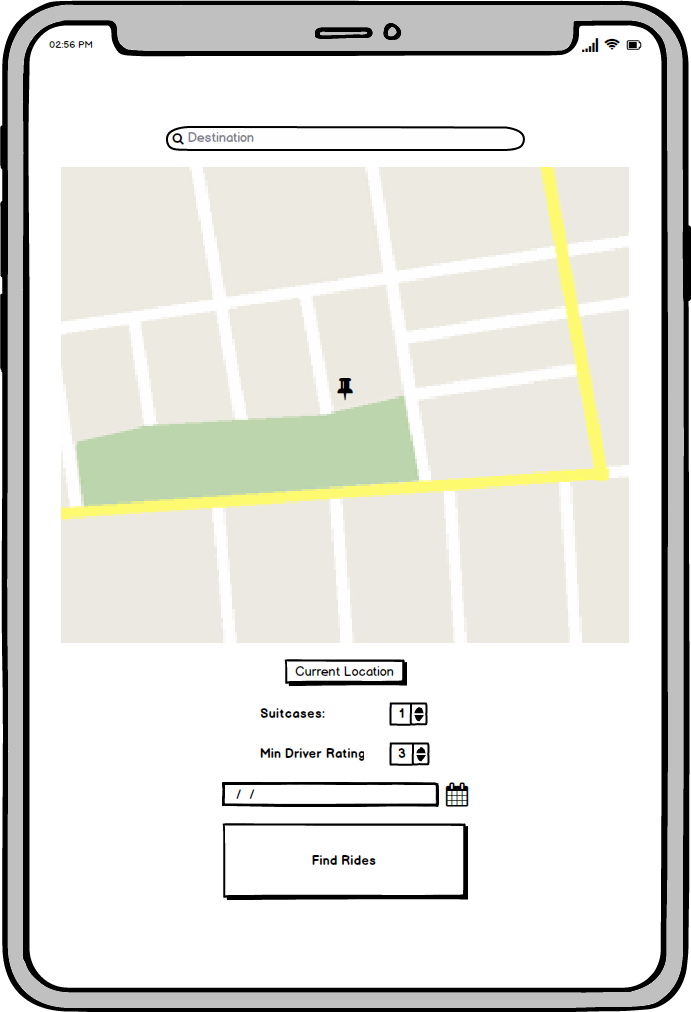
**Drivers can register a trip that they are planning into the database**



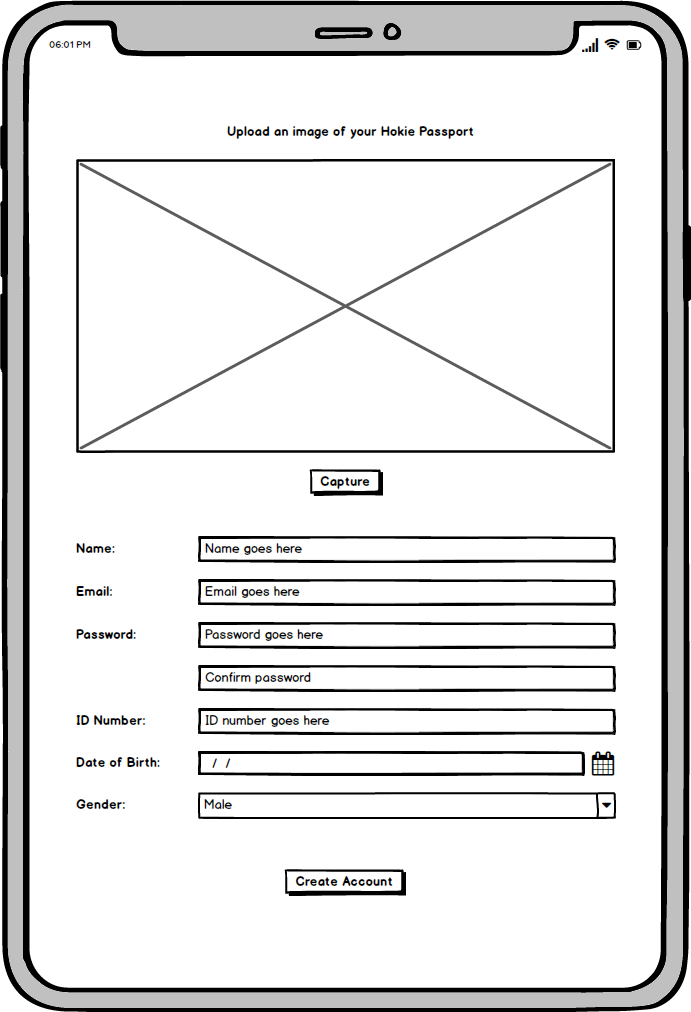
**Driver can choose riders that are interested for a trip that they want to take on their trip**



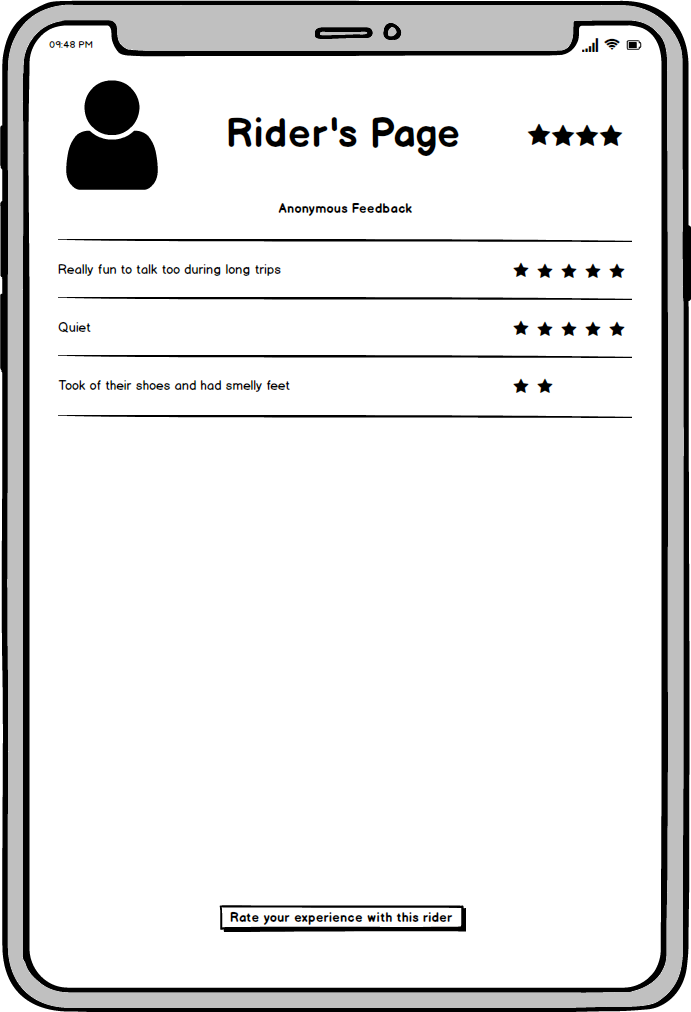
**Riders can search for trips based on proximity to final destination and also purchase luggage space in the ride**



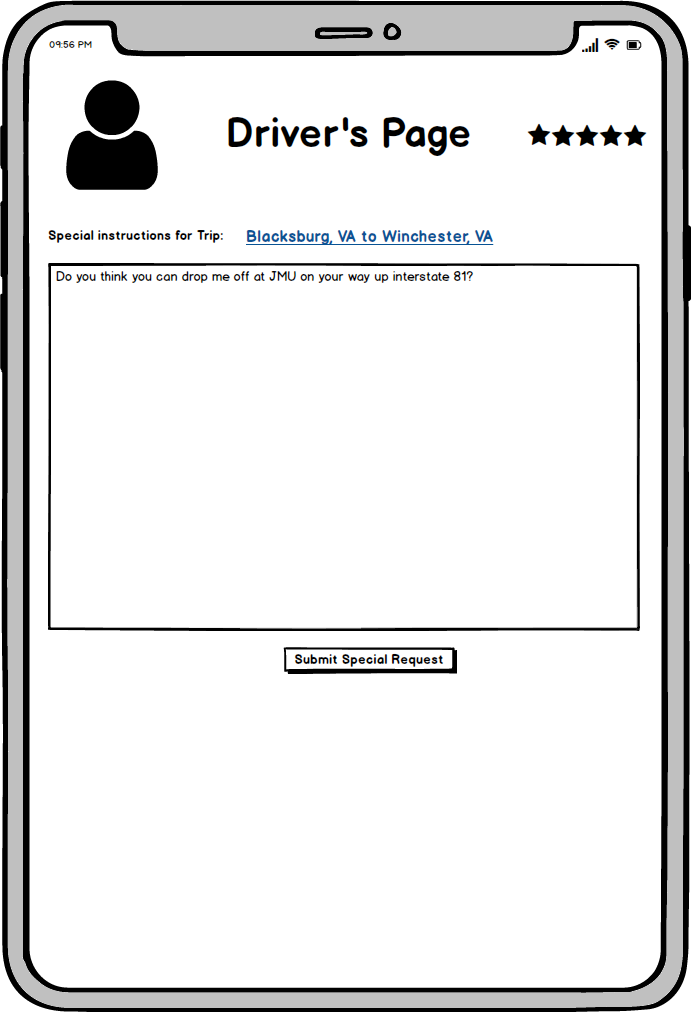
**Potential new riders can sign up using their Hokie Passport**



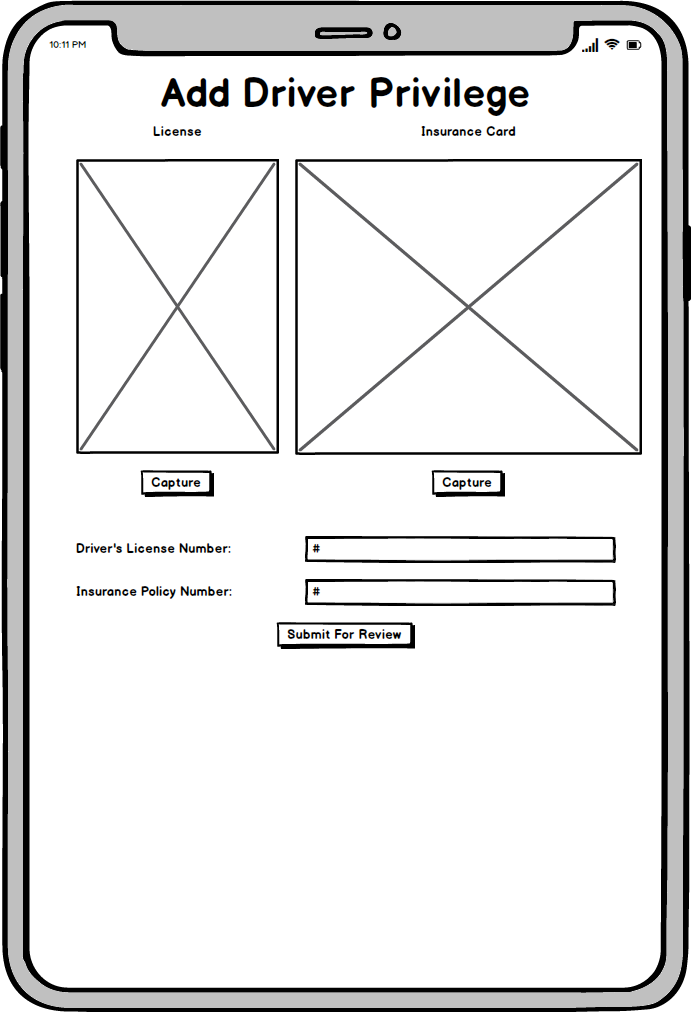
**Rider’s profile page with their feedback**



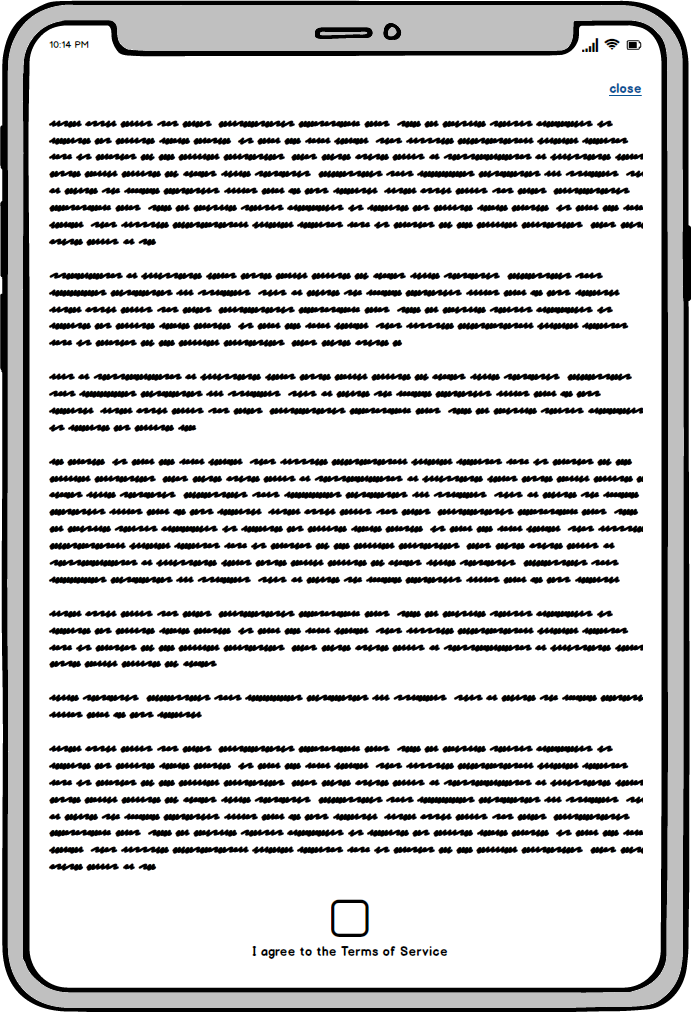
**Driver’s page displaying submitting a special request**



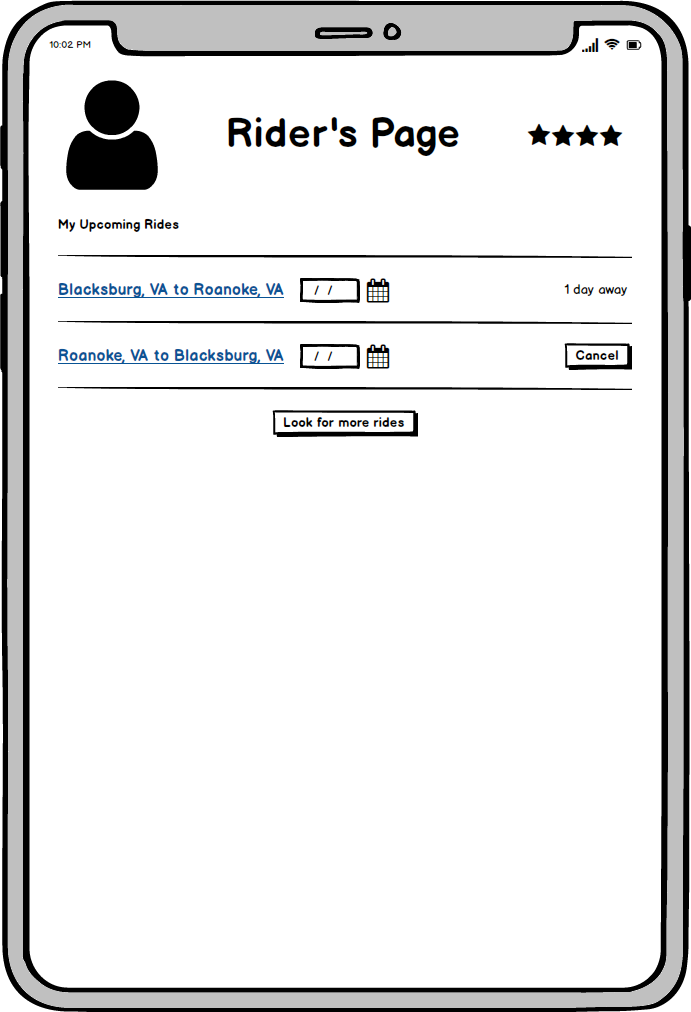
**Account registration screen for becoming a driver**



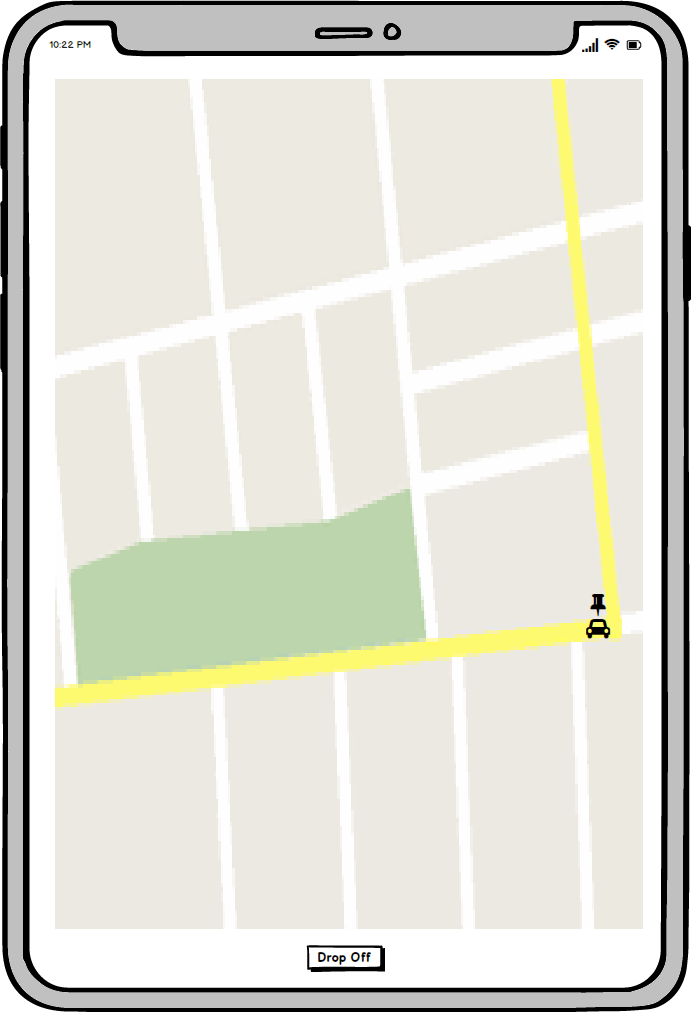
**Terms of Service**



**Rider’s page showcasing upcoming rides. They can cancel any rides pending that the ride is greater than 72 hours away. In that case the button is replaced with a timer showing how much time until the ride.**



**The Driver can click drop off and pending the drop off location is close to the scheduled drop off location. The payment is debited into the drivers account from the riders account.**



**Users can add a new payment method to their profile**



**Algorithm Design**

**Register Trip**

/\*\*

\* Submits the trip by validating data then writing it in the database.

\*/

public boolean registerTrip(Time startTime, Location startLocation, Location finalLocation, Integer seatsOpen, Payment payment, Music music)

{

boolean status = true

IF startTime is less than 0 or startTime is greater than 24.0 THEN

status = false

ENDIF

IF seatsOpen is less than or equal to 0 THEN

status = false

ENDIF

IF startLocation is not valid THEN

status = false

ENDIF

IF finalLocation is not valid THEN

status = false

ENDIF

IF payment is not validated THEN

status = false

ENDIF

IF music is not validated THEN

status = false

ENDIF

IF status is true THEN

enter trip into database

ENDIF

RETURN status //successful or any potential errors

}

**Choosing Riders**

/\*\*

\* Swipe through all the riders given a list of riders with swiped information.

\*/

public List<Rider> swipeRiders(List<Riders> riders)

{

ArrayList<Rider> chosen = new ArrayList<Rider>

FOR each rider in riders

IF rider.swipe is right THEN

chosen.add(rider)

ELSE IF rider.swipe is left THEN

status = false

ENDIF

ENDFOR

FOR each rider in chosen

Send notification to rider that they have been chosen

ENDFOR

RETURN chosen

}

**Driver is Speeding**

/\*\*

\* Given the location and the drivers speed determine what punishment to give.

\* The higher the value the worse the punishment.

\*/

public int analyzeSpeed(Location location, Speed speed)

{

int speedLimit = getSpeedLimitInLocation(location)

int speedDiff = abs(speedLimit - speed)

IF speedDiff is greater than or equal to 20 THEN

call the police //if reckless driving the police will be called

RETURN 3

ELSE IF speedDiff is greater than or equal to 10 THEN

RETURN 2

ELSE IF speedDiff is greater than or equal to 5 THEN

RETURN 1

ENDIF

RETURN 0

}

**Calculate Rider and Driver Scores using provided feedback**

/\*\*

\* This takes in feedback given and calculates the new rating for the user. It also sends

\* comments to hr if comments are negative

\*/

public float calculateScore (Feedback feedback, User user) {

int score = ((user.rating \* user.numberFeedbacks) + feedback.rating) / (user.numberFeedbacks + 1)

user.rating = score

user.numberFeedbacks += 1

IF feedback.comments is negative THEN

send comments to hr

ENDIF

RETURN user.rating

}

**Rider must be able to purchase luggage space**

/\*\*

\*Returns the options for luggage space to the rider

\*/

public List<LuggageSpace> findLuggageOptions(Ride ride) {

RETURN ride.luggageOptions

}

/\*\*

\* Rider chooses an option for luggage space that best suits their needs and then pays for it

\*/

public boolean purchaseLuggage (Rider rider, List<LuggageSpace> options) {

IF options.size is 0 THEN

RETURN false

ENDIF

rider chooses space from options

rider confirms space

IF confirmed THEN

payment of space

space.occupied = true

RETURN true

ELSE

RETURN FALSE

ENDIF

}

**Search for rides based on proximity to desired final destination**

/\*\*

\* Searches for all trips registered in database that best suits the final destination of rider. Options

\* are then presented to rider and then the rider chooses the options that they like.

\*/

public List<Ride> findRide (Rider rider, String destination) {

List<Ride> options = new ArrayList<Ride>()

List<Ride> chosenRides = new ArrayList<Ride>()

FOR each ride in allAvailableRides

IF ride.city is destination THEN

options.add(ride)

ENDIF

ENDFOR

IF options.size is 0 THEN

notify rider error

IF rider wants to add another destination THEN

FOR each ride in allAvailableRides

IF ride.city is destination THEN

options.add(ride)

ENDIF

ENDFOR

ELSE

Rider is prompted to try again later

ENDIF

ENDIF

FOR each ride in options

rider swipes left or right on ride

IF right THEN

chosenRides.add(ride)

ENDIF

ENDFOR

FOR each ride in chosenRides

send notification to driver of ride

ENDFOR

send notification to rider to wait for drivers to reach out

RETURN chosenRides

}

**Submit requests for detours**

/\*\*

\*Rider can submit a request for a detour and if the driver and other riders approve, then the

\* stop is added to the trip. Otherwise request is denied.

\*/

public boolean requestDetour (Rider rider, Ride ride, String stop, String reason) {

stop and reason are shared to other riders and driver

IF riders and driver approve detour THEN

add detour to ride

RETURN true

ELSE

detour request is denied

RETURN false

ENDIF

}

**Rider can cancel ride up to 72 hours before departure**

/\*\*

\*If a rider tries to cancel a trip up to 72 hours before departure, then cancellation is approved and

\* driver is notified and payment is refunded. Otherwise, cancellation is denied.

\*/

public boolean cancelRide (Rider rider, Ride ride) {

IF current time is 72 hours before ride.startingTime THEN

cancel rider.payment

ride.numberOfRiders--

notify ride.driver that rider cancelled

RETURN true

ELSE

notify rider that ride cannot be cancelled

RETURN false

ENDIF

}

**Drivers must sign up with valid license and insurance**

/\*\*

\* Driver uploads picture of license and insurance, and once both of those are validated, driver

\* is added to database.

\*/

public boolean driverSignUp (Driver driver, Picture license, Picture insurance) {

parse LicenseNumber and Sex from license

parse Insurance Policy Number from insurance

IF LicenseNumber is valid THEN

calculate driver.safetyScore from license

IF driver.safetyScore is low THEN

RETURN false

ENDIF

ELSE

RETURN false

ENDIF

IF insurance is valid THEN

add driver to allDrivers in system

add driver to allUsers in system

RETURN true

ELSE

RETURN false

ENDIF

}

**Riders sign up for app with Hokie Passport**

/\*\*

\* Rider creates an account and has information validated

\*/

public boolean riderSignUp(HokiePassport hokieP)

{

boolean status

IF hokieP is not equal to null THEN

parse name, birthday, ID from hokieP

ELSE

status = false

ENDIF

IF name is not valid THEN

status = false

ENDIF

IF birthday is not valid THEN

status = false

ENDIF

IF ID is not valid THEN

status = false

ENDIF

IF status is true THEN

add user to allUsers in system

status = true

ENDIF

RETURN status //successful or any potential errors

}

**Remove accounts of riders and drivers who have graduated or engage in unsafe behavior**

/\*\*

\* In a nightly job, this loops through all the users in the database and if a user has graduated, or

\* the rating for the user is less than or equal to 2, or an incident was reported about them, then

\* the account is either removed or suspended.

\*/

public void removeAccount () {

FOR each user in system

IF user has graduated THEN

remove user

ELSE IF user has user.rating less than or equal to 2 THEN

suspend account

ELSE IF user has incident reported against them THEN

suspend account

ENDIF

ENDFOR

}

**Support Hokie Passport, Venmo, and Zelle as valid payment methods**

/\*\*

\* checks if each payment method for a user is valid or not before adding it to the user’s

\* payment methods.

\*/

public List<Payment> supportPayment(PaymentType type){

List<Payment> types = new ArrayList<Payment>()

FOR each payment type the rider has

Check for valid payment method

IF payment is valid

types.add(type)

ELSE

Do not add payment method

ENDIF

ENDFOR

RETURN types

}

**Validate payments**

/\*\*

\* Checks if the user has required amount in balance before payment, otherwise notifies

\* user that there is not enough funds.

\*/

public boolean validatePayment(Payment payment){

IF rider has needed balance THEN

Approve payment.validated

Notify driver that payment is there

RETURN true

ELSE

Notify driver that rider has insufficient funds

RETURN false

ENDIF

}

**Driver receives payment upon completion of ride**

/\*\*

\* Once the rider’s destination is reached, the driver is paid from the rider’s payment

\* method.

\*/

public boolean paymentReceived(Payment payment){

Checks map to see destination

IF driver reaches riders destination THEN

payment is taken from rider’s chosen payment method and deposited to driver

RETURN true

ELSE

Driver hasn’t reached destination and does not receive payment

RETURN false

ENDIF

}