PowerEnjoy - Design Document

December 11, 2016

A.Y. 2016/2017
Bolshakova Liubov, matr. 876911
Gao Xiao, matr. 876265
Kang Shuwen, matr. 876245

Contents

1 INTRODUCTION

1.1 Purpose

Our team is focus on specifying the accomplishment of the service *PowerEnJoy*, this is a system which supply the electric cars to facilitate the transportation of citizens. In particular, the *PowerEnJoy* will offer the accessible electric cars in the reachable areas in the city and assign them rationally to users; the system also set rider clauses to standardize the driving behaviours of users by provide discounts and add compensations.

This document aims to :

- Describe the software and hardware architecture of the system we build in the design part.
- Make some reasonable adjust based on the design part, improve the system model into a more realizable direction.
- Focus on the technical approach on the system realization; analyze the deep level about the algorithm.
- Draw a clearer blueprint not only on the interface aspect but also on the use interaction.

This document is intended to be a guide of the developers who will implement our system to explain and clarify their work.

1.2 Problem Description

In particular, the system wants to:

- Provide an easy and efficient approach to users.
- Guarantee an energetic and accessible car service.

Anyone can register to be a user by supplying the valid driving license information and credit card information.

After the log in, users are able to search the available cars around the location they are or around the certain position they input. System response the search request by enumerates the available cars around them in an available queue as well as the basic information. The basic information of cars including: the accurate location; the distance to start position; dump energy, and passenger capacity.

Users are able to reserve at most one car every time; they can cancel the reservation in free within a certain time after the reservation request. The user cannot pick the car in the one hour time span should pay one euro, and the reservation will be released as well.

The system will move away the car from the available queue as soon as the car is reserved, it will move back if user cancels the reservation or the car does not be picked up within one hour.

If the user accesses the car with time limit, he /she can send request to pick up the car, system will unlock the door and let user enter. The system will start charging the car as soon as user ignites the engine, and the charging information will be notified through the screen in the car.

The user is able to choose the *money saving option* after picking up the car, and the system will response the optimal terminal station to park the car.

Users are able to terminate the ride in the safe area, the charging will stop and the car will be locked when all passengers leave the car, no matter how engine works.

If the user parks the car in the unsafe area and leave, it is necessary to lock the car for the reason of security; the system will consider that as a complete driving process with unsafe parking.

Users can recharge the car in the power gird station by plugging the car into power grid.

System estimates the driving behaviours in this ride according to the rider clauses as follow to reset payment:

- If the user took more than 2 passengers, apply 10% discount.
- If the user parked the car in safe area with more than 50% dump energy, apply 20% discount
- If the user recharged the car after parking, apply 30% discount.
- If the place user parked car is in a longer distance than 3KM to the nearest power grid station, apply 30% compensation.
- If the user parked the car with less than 20% dump energy, apply 30% discount.
- If the user parks the car in an unsafe area, apply 30% compensation.

1.3 Glossary

- Car: the cars that supplied for the car-sharing service in the *PowerEnJoy* system.
- Car information: the basic information that helps guests and users to make decisions, include the dump energy, location information, distance to the setting location, the passenger capacity.
- Starting position: the current position of user or the positions user input to start a ride.

- Available car: the car has dump energy more than 50
- Available queue: a queue that maintains available cars
- Sensors: the GPS and power plug sensor, weight sensor, display screen, battery sensor, door state sensor, locks of door in the car, and the sensor on the power grid.
- System: the whole system, which include the electric devices and the *PowerEnJoy* system background.
- Ride: in this system, the ride process is started with ignite the engine and ended with all passengers leave the car.

1.4 Document Structure

2 ARCHITECTURAL DESIGN

2.1 Overview

Taking into account the achievement of requirement analysis, we decide to use the *top-down* design strategy for our *PowerEnJoy* system. By using this strategy, we will first give a high-level component view of the overall system, in order to describe the basic logic component and the interaction between them. The high-level component view is shown in Figure 1, and the corresponding details are explained in the following paragraphs.

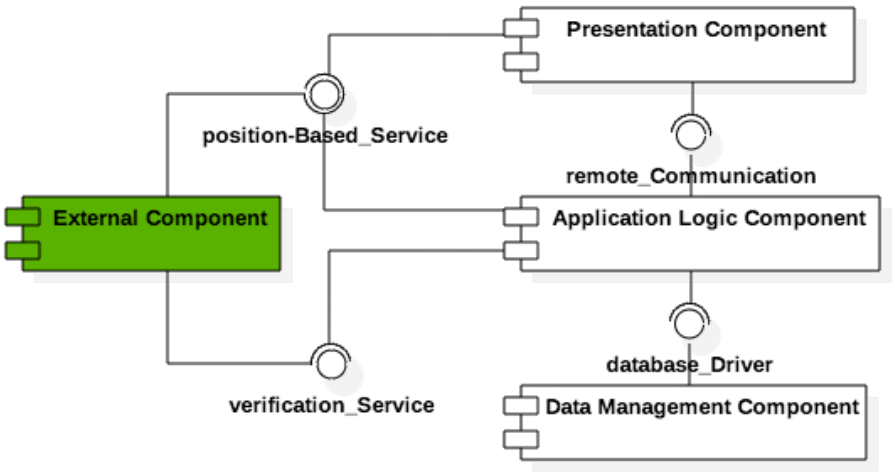

Figure 1: High Level Component Diagram

- **External Component**

  Thanks to the requirement analysis process, we have figured out two basic expected functionalities provided by external system, which are position-based service and document verification service. So in the high-level component view, these two services are provided by an high-level component, *External Component*, with specific interface.

- **Presentation Component**

  As we have decided in the RASD document, our system will offer two GUI to the user for visualizing the interaction. One of them is the mobile phone application and the other is the on-board application. These two perform a similar role in the overall system , so we combine them as one high-level component, *Presentation Component*.

  This component uses the position-based service provided by the *External Component*, in order to visualize the position of user and car

on a geographic map. Meanwhile, this component depends on the interface provided by the *Application Logic Component*, for the aim of interacting with the server.

- **Application Logic Component**

  This high-level component essentially represents all the business logic offered by our system.

  It provides logic response to *Presentation Component* with the help of *remote_Communication* interface. And it accesses to database by using the interface offered by the *Data Management Component*.

- **Data Management Component**

  This high-level component handles all the data related manipulation, and offers a driver interface for other components to access data.

2.2 Component view

After an overall looking of our system, now we are going to provide more details for the architecture with a low-level component view, which is shown in the Figure 2. The components here can be seen as the subcomponent of those in high-level view, and are described in detail as follows:

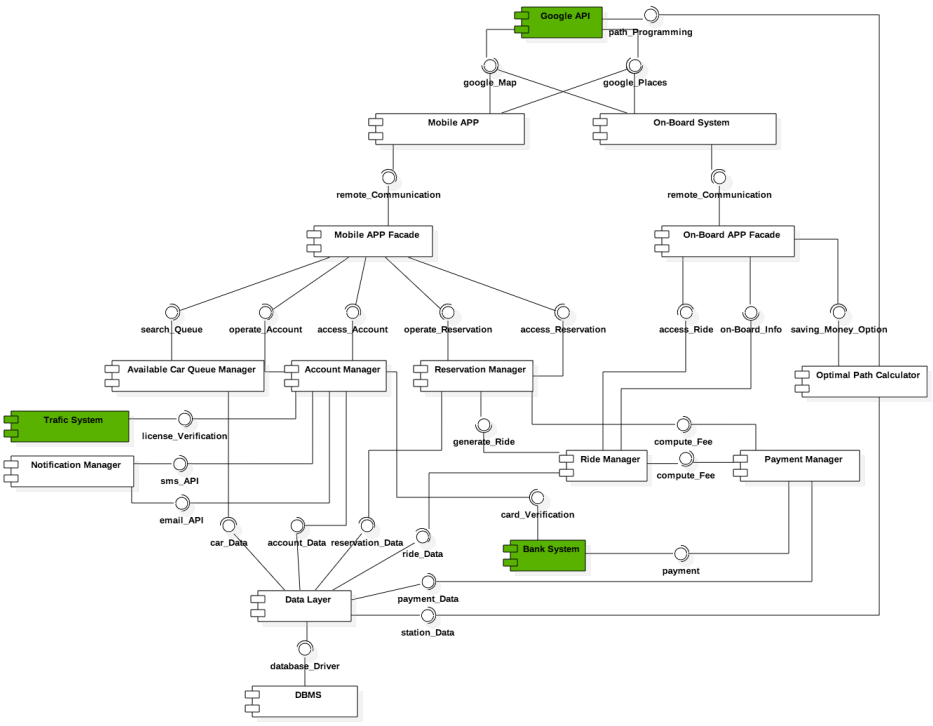

Figure 2: Low Level Component Diagram

- **Google API**

  It is an external Google component that provides the *Mobile APP* and *On-Board system* some interfaces, makes the *PowerEnJoy* system able to get the data about information from the same database of *Google Maps*, also the geographical location of points from the same database of *Google+Local*.

- **Traffic System**

  It is an external component that provides the *Account Manager* an interface, in order to make the *PowerEnJoy* system verify drive license by using data from the *Traffic system* database.

- **Bank System**

  It is an external component that provides the *Account Manager* some interfaces, makes *PowerEnJoy* system be able to verify the credit card by using data from the database of banks.

- **Mobile APP**

  It is the component which embeds on the mobile devices of users, it offers user a GUI that:

  - Allows the users to access the functionality the system provide him.
  - Displays the data obtained by the interaction with *Mobile APP Facade*.

  And it charges not only the presentation behaviours but also the communication duty in order to deliver the tasks user required as well as the message, and the response from *Mobile App Facade*.

- **On-Board System**

  It is the component which is embedded inside the panel of the car, it offers a GUI that:

  - Allows the users to access the functionality the system provide him to the car.
  - Displays the data obtained by the interaction with *On-Board System Facade*.

  And this component can also handle all communications with the *On-Board System Facade* to accomplish the tasks. All messages from *On-Board System* to the *PowerEnJoy* servers are also from this component.

- **Mobile APP Facade**

  This is a component implementing a *Facade Pattern*: this component offers the interfaces to accomplish the functionality that should be manipulated correspondingly by different managers. Only this facade has an interface which can interact with the *Mobile App*.

- **On-Board System Facade**

  This is also a component implementing a *Facade Pattern*: this component offers the interfaces to accomplish the functionality that should be manipulated correspondingly by different managers. Only this facade has an interface which can interact with the *On-Board System*.

- **Available Car Queue Manager**

  This component is able to access to the *Data Layer* component to get the car state information, and the manager updates the available cars as soon as the car state in *Data Layer* changes.

- **Account Manager**

  It is the component which support the functionalities about registration and login. In order to do this, the Account Manager is able to:

  - Access the external *Traffic System* component to verify the data.
  - Access the external *Bank System* to verify the data.
  - Access the *Notification Manager* to exchange messages.
  - Access the *Data Layer* to store user data as well as check the history record.

- **Reservation Manager**

  This component handles reservation by:

  - Access the *Data Layer* component in order to check or change the state of reservation.
  - Make a countdown as well as get an reservation.
  - Access the *Payment Manager* only if the reservation fails.
  - Access the *Data Layer* component in order to change the car state as well as store reservation information.
  - Access the *Ride Manager* only if the user picks up the car successfully.

- **Ride Manager**

  This component handles the car which is picked up successfully, it accesses to the *Reservation Manger*, commence the operation by the

trigger of the message from Reservation Manager,and the *On-Board System* is triggered at the same time. The *Ride Manager* access the *On-Board System* and supply the functionality the user can get on aboard.

The *Ride Manager* accesses to the *Data Layer* at the end of the ride in order to update the car state as well as store the ride information,also accesses to the payment manager to transmit the payment information.

- **Payment Manager**

  It can receive the payment information by accessing the *Reservation Manager* and *Ride Manager*, after calculate the accurate money, the *Payment Manager* accesses to *Bank System* to get the money, and also accesses to the *Data Layer* to store the payment details.

- **Notification Manager**

  It handles the text message and email which sent by the *PowerEnJoy* system to the users.

- **Optimal Path Calculator**

  This component accesses to the *On-Board System* in order to get the address information from uses, and also able to access both the *Google API* and *Data Layer* to calculate the optimal path from start point to destination.

- **Data Layer**

  This is the component that allows *Reservation Manager*, *Account Manager*, *Available Car Queue Manager* ,*Ride Manager*,*Optimal Path Manager* and *Payment Manager* to access the data which stored in database.

- **DBMS** It is a external component that is able to interact with the database where all the system data are stored via query.

2.3 Component interfaces

In this part, we are going to give a detail description for the interfaces offered by components presented in the Low Level Component Diagram. Also we will describe how these interfaces help components to interacted with each other.

2.3.1 Google API

- **google_Map**

  This interface provides three basic functions:

- a graphical representation of geographical maps
- the service of searching a particular address
- estimate the distance between two specified location

It is used by two other components listed as follows:

- *Mobile APP*:
  The interface provides a graphical map for the mobile phone application, in order to visualize the position detected by the GPS, or alternatively the position with a specified address inserted by the user.
- *On-Board System*: Similar to the component above, the interface provides a graphical map for the on-board application to display the current position of the car detected by GPS. Meanwhile, it enable the on-board application to search a particular address inserted by the user.

- **google_Places**

  This interface provides the service of searching marked position located around a given address.

  It is used by two other components listed as follows:

  - *Mobile APP*: Together with the available car position data, this interface enable the mobile phone application to present position of available cars close to the user's location or an address specified by the user.
  - *On-Board System*: Together with the power station position data, this interface enable the on-board application to get position info of stations closed to the user's input destination.

- **google_path_Programming**

  This interface provides the service of programming a path between two specified position. It is used by the *Optimal Path Calculator* component, for the aim of programming an optimal path for those users whose select the "saving money option".

2.3.2 Traffic System

- **license_Verification**

  This interface provides the service of verifying a driving license, with the driving license ID offered by the user. Thanks to this, the assumption that all the users have a valid driving license is satisfied.

  It is used by the *Account Manager* when a new user tries to create an account.

2.3.3 Bank System

- **card_Verification**

  This interface provides the service of verifying a credit card, with the card number offered by the user. With the help of this, the assumption that all the users have have a credit card which can be used for the payment is satisfied.

  It is used by the *Account Manager* when a new user tries to create an account, or whenever the user is willing to modify the card for payment

- **payment**

2.3.4 Mobile APP Facade

- **remote_Communnication**

  This interface provides access to all the functionalities, the mobile phone application offered to user, via proprietary protocols.

  It is used by *Mobile APP* component to transfer the user's action to the logical layer, as well as transfer the logical result back to the user.

2.3.5 On-Board System Facade

- **remote_Communication**

  This interface provides access to all the functionalities, the on-board application offered to user, via proprietary protocols.

  It is used by *On-Board System* component to transfer the user's action to the the logical layer, as well as transfer the logical result back to the user.

- **on-Board_Info**

  This interface provides access to car state data which is detected by the sensors on the car and sent from the *On-Board System* component.

  It is used by the *Ride Manager* to:

  - count number of passengers
  - modify ride state according to engine state
  - detect battery level
  - detect charging state
  - detect parking position

2.3.6 Available Car Queue Manager

- **search_Queue**

  This interface provides access to data of available cars.

  It is used by the *Mobile APP Facade* component, when:

  - the user starts searching available cars;
  - the user selects one of the available to check detail information.

2.3.7 Account Manager

- **operate_Account**

  This interface provides the service of creating new account and modifying account data of specified user.

  It is used by the *Mobile APP Facade* component when a new user tries to register in the system, or when the user tries to modify personal account data, including:

  - changing password
  - changing payment method
  - changing contact information
  - modifying personal data
  - deleting previous ride records.

- **access_Account**

  This interface provides access to personal account data.

  It is used by the *Mobile APP Facade* component when the user tries to login an account and check personal account data, including:

  - checking current payment method
  - checking contact information setting
  - checking personal data
  - checking previous ride records.

2.3.8 Reservation Manager

- **operate_Reservation**

  This interface provides the service of creating new reservation and modifying current reservation state.

  It is used by the *Mobile APP Facade* component when the user tries to:

  - make reservation, which means create a new reservation and set status to ACTIVE
  - cancel reservation, which means change current reservation status to CANCELLED
  - pick up the car, which implies change current reservation status to COMPLETE.

- **access_Reservation**

  This interface provides access to current reservation data.

  It is used by the *Mobile APP Facade* component when the user tries to check current reservation data, including:

  - reservation count down
  - position of reserved car.

- **generate_Ride**

  This interface provides the service of generating a new ride once a user complete a reservation by picking up the reserved car.

  It transfers the user data and car data of a complete reservation to the *Ride Manager* component.

- **compute_Fee**

  This interface provides the service of computing demurrage and cancelled fine according to reservation status.

  It transfers the amount of fee generated by the reservation operation to the *Payment Manager* for the aim of completing the fee payment.

2.3.9 Ride Manager

- **access_Ride**

  This interface provides access to current ride data.

  It offers the real-time basic consumption to the *On-Board APP Facade*.

- **compute_Fee**

  This interface provides the service of computing final ride consumption by integrating the:

  - basic consumption regraded to riding duration
  - discount
  - compensation

  It transfers the amount of fee generated by the ride to the *Payment Manager* for the aim of completing the fee payment.

2.3.10 Notification Manager

- **sms_API**

  This interface provides the service of sending text message to user.

  It is used by the *Account Manager* in order to send the login password, which is generated by the application server, back to the user via text message.

- **email_API**

  This interface provides the service of sending email to user.

  It is used by the *Account Manager* in order to send the login password, which is generated by the application server, back to the user via email.

2.3.11 Optimal Path Calculator

- **saving_Money_Option**

  This interface provides the service of finding the optimal position of final parking power station, according to the destination specified by the user, as well as programming the optimal path to get there for the aim of saving money.

  It transfers the optimal solution to the *On-Board Facade* component.

2.3.12 Data Layer

- **car_Data**

  This interface provides the search operation related to car status data.

  It is used by the *Available Car Queue Manager*.

- **account_Data**

  This interface provides the database operations related to account data, including

  - search
  - insert
  - modify
  - delete (only for previous reservation and ride records)

  It is used by the *Account Manager*.

- **reservation_Data**

  This interface provides the database operations related to reservation data, including

- search
- insert
- modify

It is used by the *Reservation Manager*.

- **ride_Data**

  This interface provides the database operations related to ride data, including

  - search
  - insert
  - modify

  It is used by the *Ride Manager*.

- **payment_Data**

  This interface provides the insert operations related to payment data.

  It is used by the *Payment Manager*.

- **station_Data**

  This interface provides the search operations related to power station data.

  It is used by the *Optimal Path Calculator*.

2.3.13 DBMS

- **database_Driver**

  This interface enable the interaction with the database.

  It is used by the *Data Layer*.

2.4 Deployment view

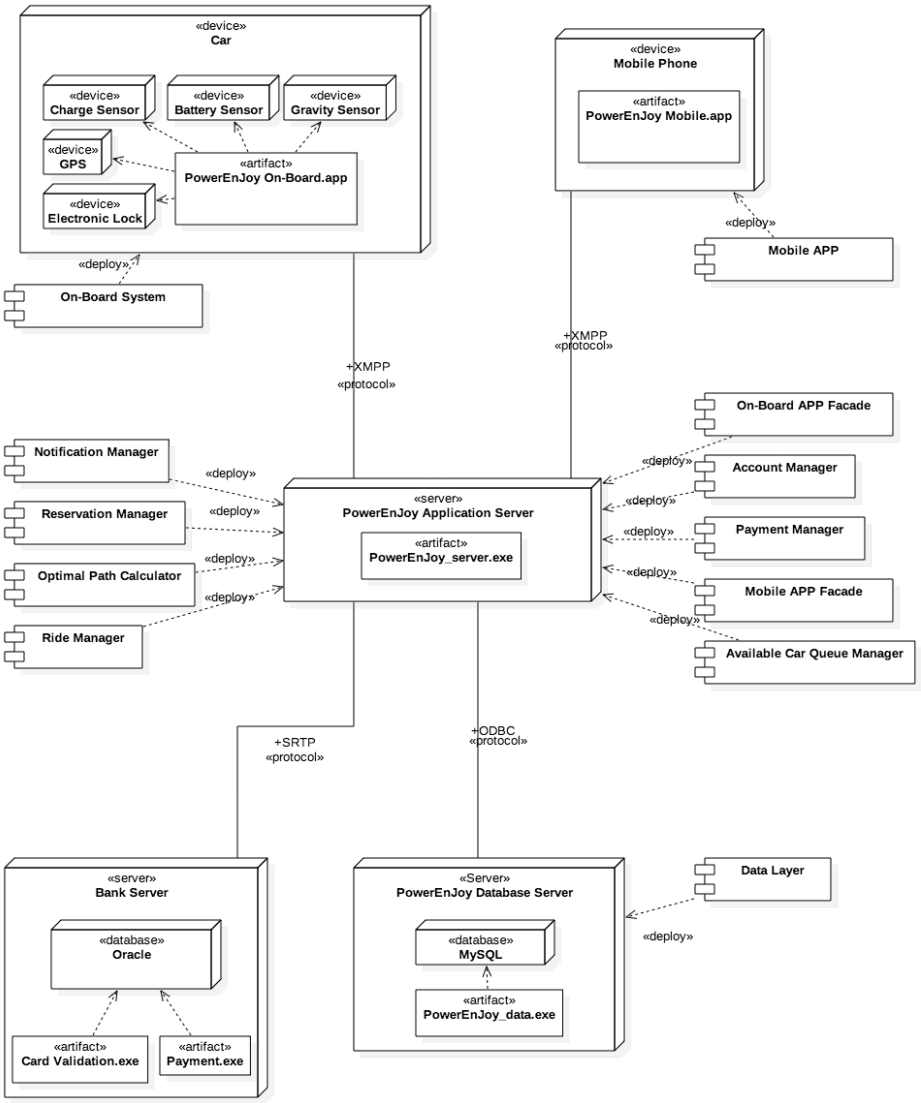

Figure 3: Deployment Diagram

In the Figure 3, we present the hardware of the system, the sub-device deployed on the hardware, as well as the software that is installed on the hardware. In order to show the connection between component view and deployment view of our *PowerEnJoy* system, we also describe the deploy relation between each hardware and the logic components previously referred.

2.5 Runtime view

2.5.1 Register and Login

The following Figure 4 sequence diagram shows us how the component interact with each other when the *Mobile APP* get register and login request from the interface.

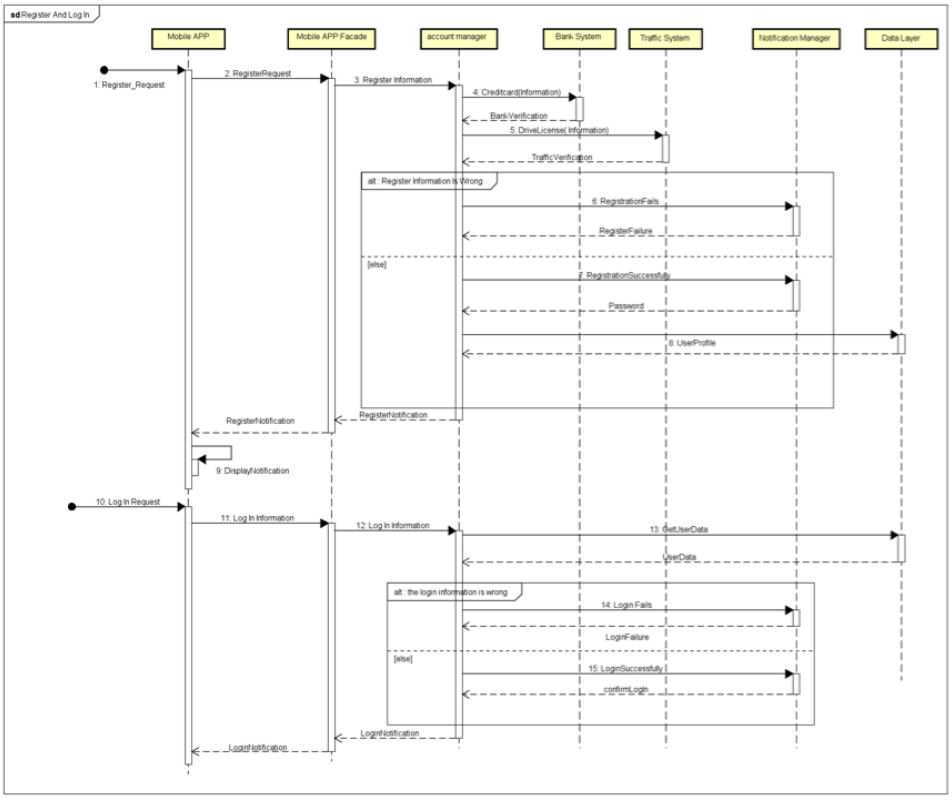

Figure 4: Register and Login

Once guest accesses the *Mobile APP* and send a register request, the *Mobile APP* terminal gets the register information from the interface, then deliver it by remote server. The information flow through *Mobile APP Facade* and *Account Manager*, and be dispatched to the diverse external interfaces: *Traffic System* and *Bank System* in order to verify the availability. These two external interface both response a verification to *Account Manager*, the *Notification Manager* responses the different message corresponding to the verification result .If the guest register successfully, the register information will be stored in the database. Note that the profile stored also include the password system generated, and the guest can decide by himself if he should launch a new round of register under the condition of registration failed.

When the *Mobile APP* terminal gets the login request as well as the lo-

gin information, it deliver the information to *Account Manager* through the *Mobile APP Faade* .The *Account Manager* checks the correctness of login information, then the *Notification Manager* responses messages corresponding to the check result. Note that the user can decide if he should launch a new round of login request under the situation of login failed.

2.5.2 Make a Reservation

The sequence diagram Figure 5 shows how components interact with each other to make a reservation from the *Mobile APP* side.

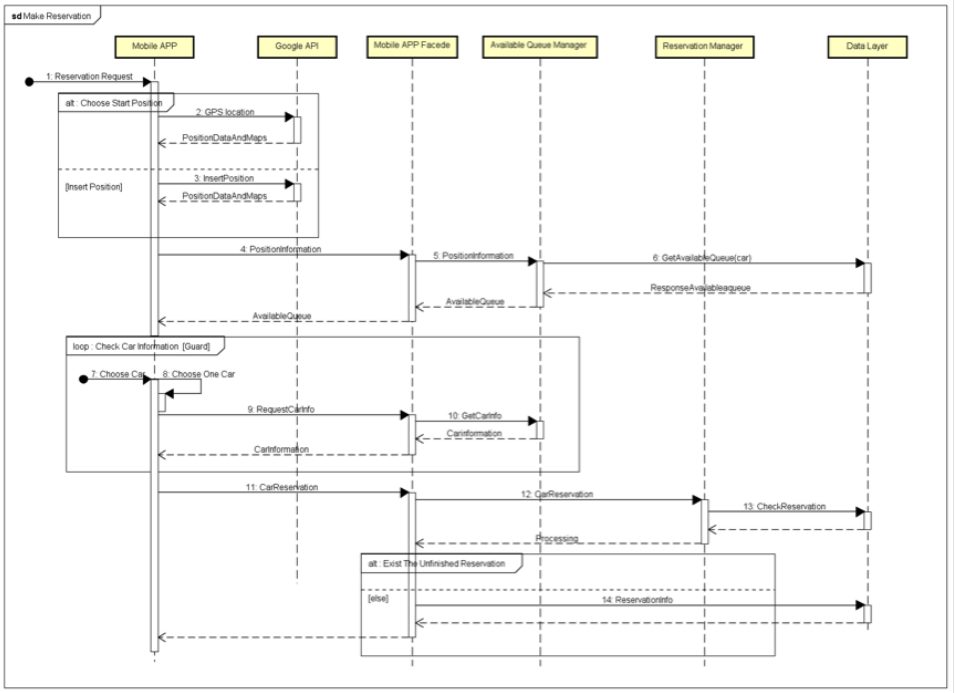

Figure 5: Make a Reservation

User offers a start position, once the position be found, the *Google API* send back the position data and map around the start position to *Mobile APP* terminal, and also deliver the position information to *Mobile APP Faade* ,which trigger the execution of searching an available queue of car.

The *Available Car Queue Manager* then responses a queue to the *Mobile App* terminal, so that user can make reservation. The car will be removed from the available queue as long as it is reserved. The reservation data will be created in the database as well.

2.5.3 Cancel Reservation Automatically

In this sequence diagram Figure 6, we can see how the reservation be cancelled automatically by system.

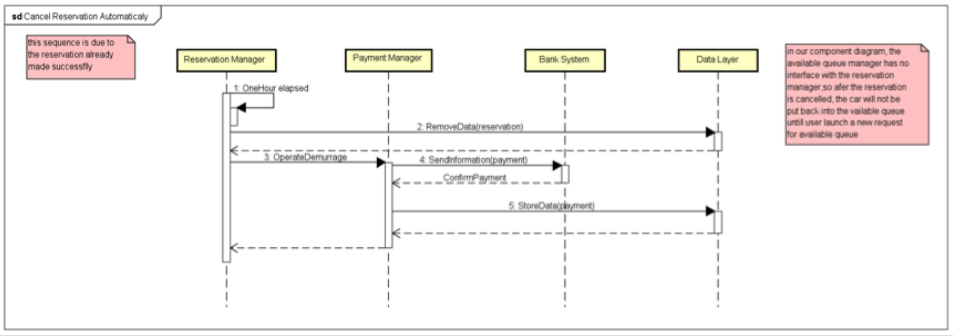

Figure 6: Cancel Reservation Automatically

An hour elapse can trigger the Reservation Manager to cancel the current reservation; the *Payment Manager* should also interact with *Bank System* consequently to deduce the money as the compensation fee.

Note that the user can get the notification of reservation cancelled from the *Mobile APP* terminal.

2.5.4 Cancel Reservation Manually

This following sequence diagram Figure 7 tell us how the component work together to let user cancel the reservation from *Mobile APP* terminal.

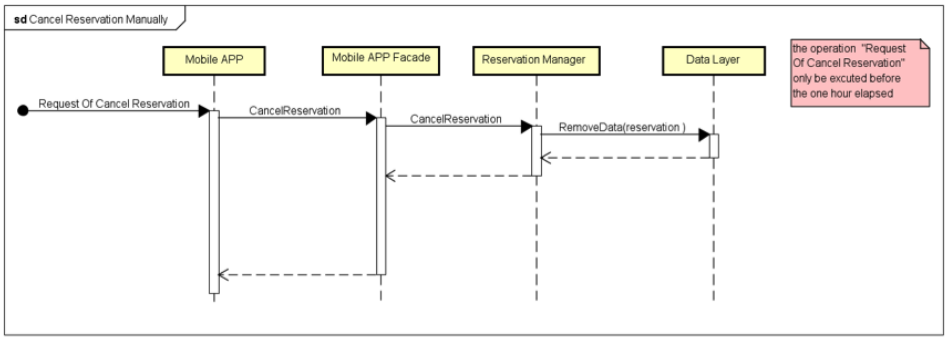

Figure 7: Cancel Reservation Manually

The user launch a request to cancel the current reservation, the request is passed from remote server to the *Reservation Manager*, which removes the corresponding data from database and response the confirmation back.

Note that this event can only exists when the reservation lives less than one hour, and the available queue will be updated when user launched a research of available queue.

2.5.5 Complete Ride on Mobile APP Side

We can know from the following sequence diagram Figure 8 about how the components cooperate together to accomplish a complete ride on the Mobile APP side.

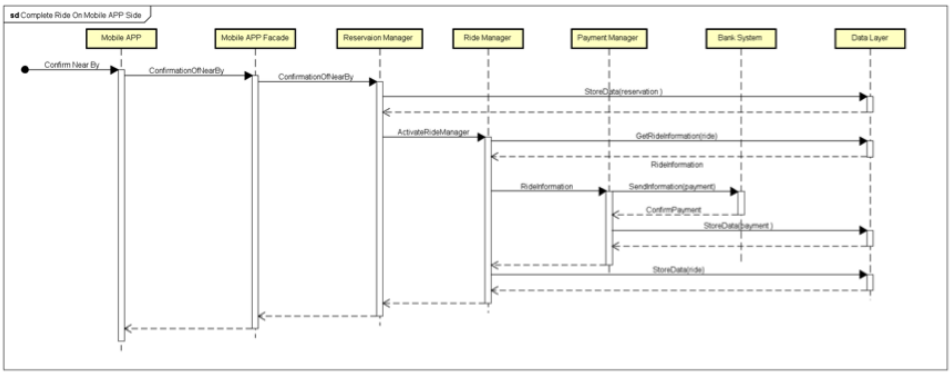

Figure 8: Complete Ride on Mobile APP Side

This operation is mainly focuses on the messages exchange and data exchange between components, the step is:

- The user triggers the ride operation by a confirmation, the *Mobile APP* sends the confirmation by remote server, and activates the *Reservation Manager*. The *Reservation Manager* ends current reservation a successful finished reservation; store the information into the database.
- The *Ride Manager* is ignited, the *Data Layer* responses the ride information back as well as the complete ride finished, the *Payment Manager* will calculate the accurate money that cost in the current ride.
- The *Payment Manager* sends the payment information to *Bank System*, which responses the confirmation of payment which signs the complement of payment. Then *Payment Manager* stores the payment data.
- The *Ride Manager* stores the data, and the whole system response step-wisely.

Note that the ride operation is executed in the same time at both the *Mobile APP* side and the *On-Board System side*, the ride is completely

finished only if the executions finished in both side. These two sides have data communication, which is not showed in this sequence diagram.

2.5.6 Complete Ride on On-Board System

In the following sequence diagram Figure 9 ,we can see the ride operation details on the *On- Board System* side.

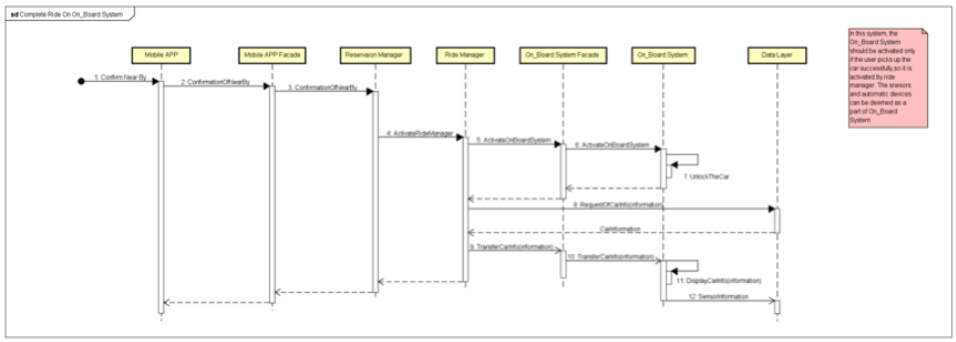

Figure 9: Complete Ride on On-Board System

The *On-Board System Facade* is activated by the Ride Manger, then the *On-Board System* is activated consequently. The *On-Board System* terminal unlocks the car, and it signify the start of a ride.

The *Ride Manager* get the car information, then the *On-Board System* terminal displays the information on the screen. the *On-Board System* also send the sensor information to the *Data Layer* asynchronously.

Note that the *On-Board System* include all the sensor embedded into the car, the display screen, the GPS, also the autonomic devices like the lock in the door. The *On-Board System* sends the car information real-timely to the *Data Layer*. The accomplishment of a complete ride is also gained from the sensors information.

2.5.7 Saving Money Option

In the last sequence diagram Figure 10, we can know exactly how the component work together, supplies the way to users to access the saving money option.

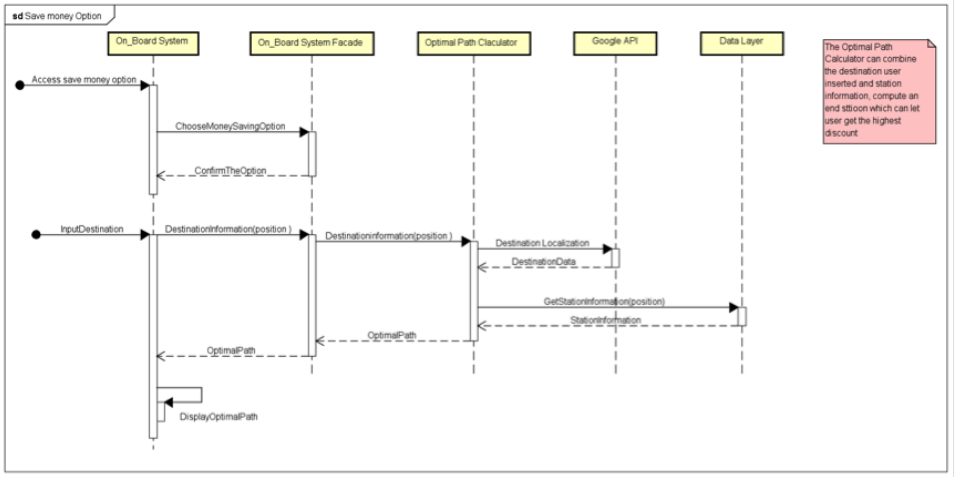

Figure 10: Saving Money Option

The request is received from the interface of *On-Board System*. When the *On-Board System* confirms the request successfully, the *On-Board System* will have the interface to get the destination information from user.

The *On-Board System Facade* sends the destination information to the *Optimal Path Calculator*; the *Optimal Path Calculator* gets the destination data from *Google API*, the station information from the *Data Layer*, and responses the optimal path to the fittest station.

The optimal path is returned to the *On-Board System* and be showed on the screen.

Note that the station information also includes the information of plugs sensor in the power station, so that the *Optimal Path Calculator* can also consider the free plugs in that power station.

2.6 Selected architectural styles and patterns

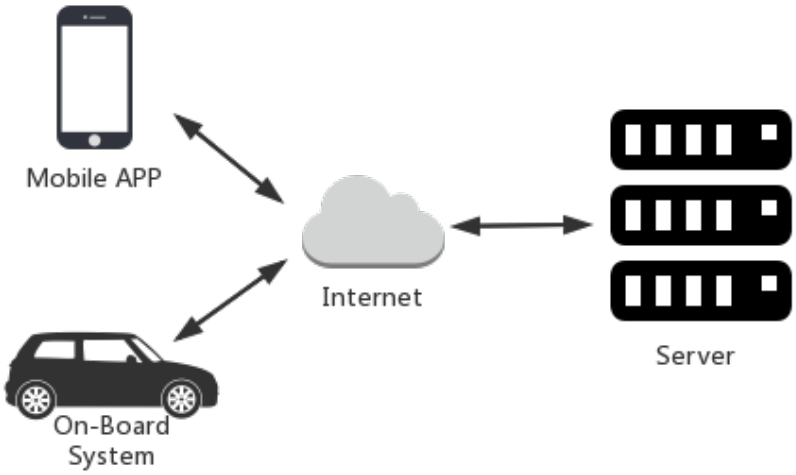

Figure 11: Client-Server Architecture Style

2.6.1 Architectural style

As it is shown in Figure 11, our system is basically based on *client-server* architecture style. And there are two types of clients connected to the server. According to the system feature, we choose to use 3-tier architecture to deploy our system. The physical architecture is presented in Figure 12. In the following two paragraphs we will explained the two different types of clients in detail.

- **Mobile Application Client**

  The Mobile Application is a *thin* client, because the business logic is implemented entirely one the application server. This client provides a presentation layer for the user to manipulate data related to personal account, current reservation, and previous records. Also it can deliver the response of application server to user by displaying them on the mobile phone screen. So basically this client represents a GUI.

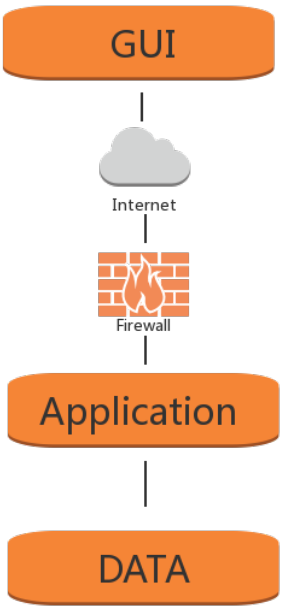

Figure 12: 3-Tier Architecture

- **On-Board System Client**

  The On-Board System is also *thin* client, because it transfers all the car status information to the application server, which actually executes the business logic. This client provides a GUI for user who has successfully picked up a car; allows the user to select *saving money option*; and displays the current basic consumption on the on-board screen.

2.6.2 Design patterns

- **MVC Pattern**

  For the entire software structure, we use MVC pattern, which stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns. In our project:

  - View lies on both two client side, to handle the interaction with users.
  - Model and Controller are deployed on the server side, for the aim of maintaining business logic and dealing with data manipulation.

- **Facade Pattern** For the structure of server side, we use Facade pattern to hide the complexities of the system and provide an interface

to the client using which the client can access the system. We implement this pattern by simply adding to facade components:*Mobile APP Facade* and *On-Board System Facade*.

2.7 Other design decisions

2.7.1 Database Design

In this paragraph we report an ER diagram that describes the structure of *PowerEnJoy* database.

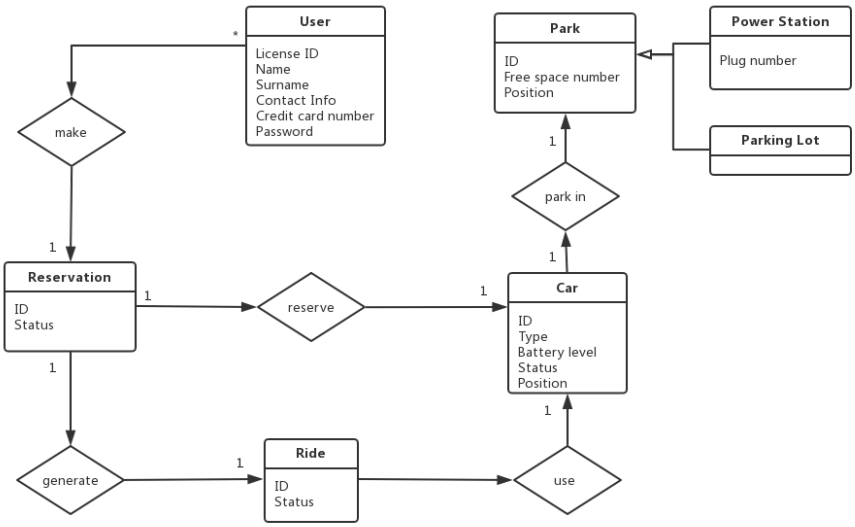

Figure 13: E-R Diagram

2.7.2 Programming Language

Considering the implementation of our *PowerEnJoy* system, we decide to use JAVA as our programming language. This is mainly because of Java's robustness, ease of use, cross-platform capabilities and security features. And our system shall be developed on the Java EE platform, which ensures us large-scale, multi-tiered, scalable, reliable, and secure properties.

3 ALGORITHM DESIGN

For a clearer description about our project, we clarify the algorithm about queue management, pay management and money saving option, which we think they are important aspects of development of application.

3.1 Queue Management

This algorithm needs the position information about all the available cars in the city, as well as the accurate start position information from the user. Every time user launches a request for the available cars,the *Available Queue Manger* gets the coordinates of available cars in the city, then it starts the research of an available queue of cars around the start position which insert by user or oriented by GPS.

Cars in the queue will be updated when a new request of searching is launched. It can be added with the cars which is released from a ride or a reservation, it can also be removed by reservation.

Cars in the available queue are enumerated in the increase order in distance to the start position and the decrease order both in dump energy of car. The available car which is far than 3 KM are be consider as unreachable cars, and will not be enumerated in the available queue.

The research is done in according to the algorithm bellow, written in pseudo code:

```
Function searchAvailableQueue
cars(array)
destination(GPS coordinate)
while Distance(car.location,start)<=3
      Sort(
                cars as car;
                by  Distance(car.location,start)
                *10000000+(100-car.battery)
           )
      return cars;
End function
```

3.2 Payment Management

This algorithm needs the sensor information about both the car and power station. When a ride finished, the system sends all the sensor information to the *Payment Manager* in order to compute the money user should pay in this ride.

The computation is done in according to the algorithm bellow, written in pseudo code:

```
Function PaymentCalculate
Sensor(array)
Discount1:=0.8
Discount2:=0.7
Compensation:=1.3
ComputeFinalPrice(
                price:=Sensor.rideTime*moneyPerMinute
                numPassengers:=Sensor.passenger
                dumEnery:=Sensor.battery
               if( numPassengers>2)
                    price:=price*Discount1
               if( dumEnergy/fullBattery >0.5)
                    price:=price*Discount1
               if( dumEnergy/fullBattery <0.3)
                    price:=price*Compensation
               if( numPassengers>2)
                    price:=price*Discount
               if(Sensor.plug)
                    price:=price*Discount2
               if(( Distance(car.location,powerStation))>3)
                    price:=price*Compensation
)
return price
End
```

3.3 Money Saving Calculation

This algorithm is used for calculate the optimal station near the destination which user can get the height discount. It needs the destination from *On-Board System*, also the distribution of power station.The algorithm consider both the distance and the accessibility of power plugs, return user the optimal station not only reduce the cost of user but also contribute the cars in a reasonable way.

```
powerStation(array)
destination(GPS coordinate)
function moneySavingOption(powerStation,destination)
suggestedStation<-none
currentDistance<-infinite
for ps in powerStations
   distance<-distance(destination,ps.location)
    if ps.numPowerPlugs>0 and (
          distance<currentDistance or(
```

```
                distance=currentDistance
            and suggestedStation.numPowerPlugs
                  <ps.numPowerPlugs
                                      )
                                  )
    then
        suggestedStation<-ps
        currentDistance<-distance
return suggestedStation
```

4 USER INTERFACE DESIGN

In this section we discus about the design of two user interfaces with the help of mockups.

4.1 Mobile Application Interface

Figure 14: Welcome Page Figure 15: Register Page

- Figure 14 shows the welcome page of our mobile application, which is presented to users once they open the APP. This page provides a register choice for new users, and offers a login choice for those users who already have an account.
- Figure 15 simply gives the basic required information that user has to provide for the aim of registering in our system.

Figure 16: Self Positioning Figure 17: Search Position

- Figure 16 presents the self-positioning, which is one of the two approaches to specify a position in order to search for available cars.
- Figure 17 shows another method to search available car, which is to input a specific address.
- Figure 18 is the page provided to user for making a reservation. In this page, user can check the basic information of the car to reserve, including car type, parking position, and battery level.
- Figure 19 shows the page for picking up the reserved car. In this page user is notified by the countdown of current reservation. This page is alive only within the one hour's countdown, and it will dead as soon as the reservation time up.

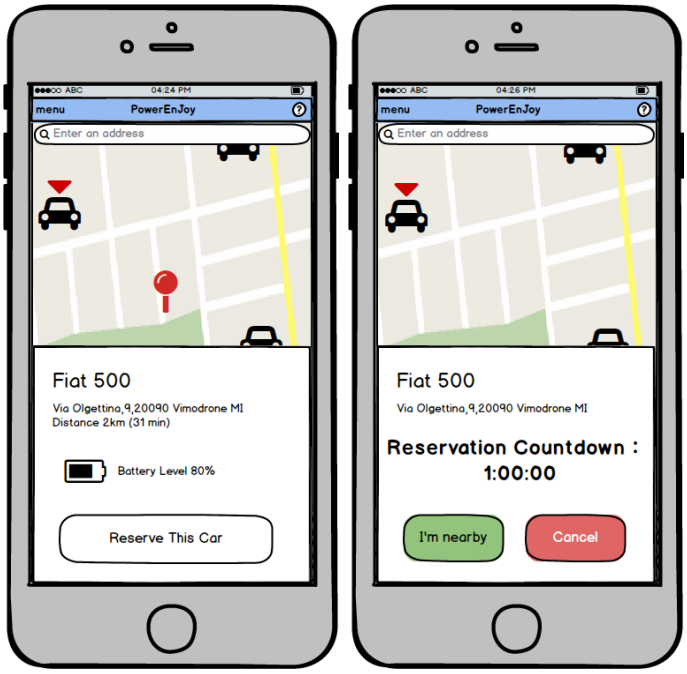

Figure 18: Make Reservation

Figure 19: Pick Up Car

4.2 On-Board Application Interface

- Figure 20 shows the desktop of our on-board application.It basically visualizes the essential information of a ride to user, including the current position of the car, real-time consumption, battery level, and ride duration.
- Figure 21 shows the page for user to input the destination. This page is triggered whenever the user press on the Navigation button or the Money Saving button.
- Figure 22 presents the general navigation page provided to user who has not activated the money saving option. Here we can see that the destination is exactly the one that user specified.
- Figure 23 shows the navigation page for user who has activated the money saving option. In this page, the final destination is calculated by our system according to the *Money Saving Calculation* algorithm.
- Figure 24 presents the service of searching the power station closed to user. This is an additional service which has not been referred before, but it is quite simple to realize by using part of the logic of existed service. We offer it here to simplify the parking place finding process of our user.

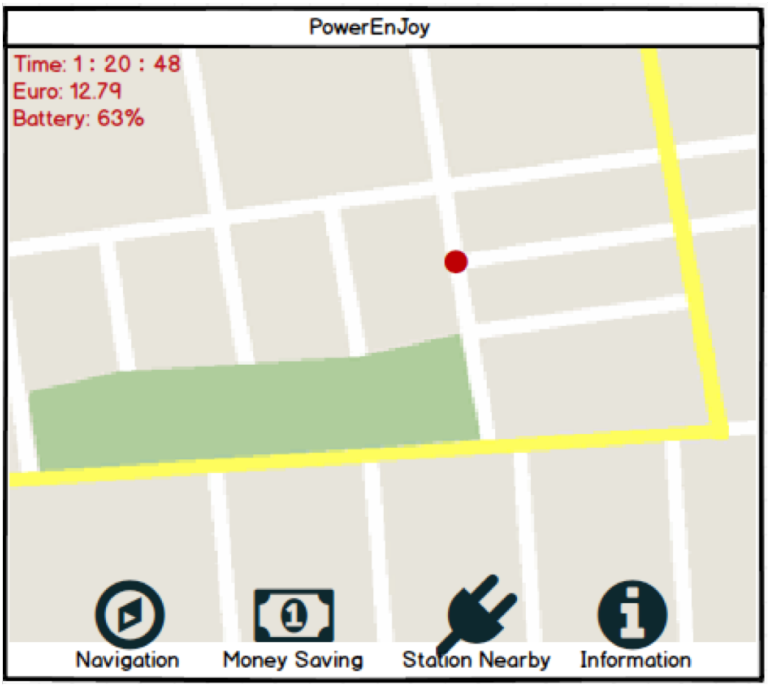

Figure 20: On-Board Application Desktop

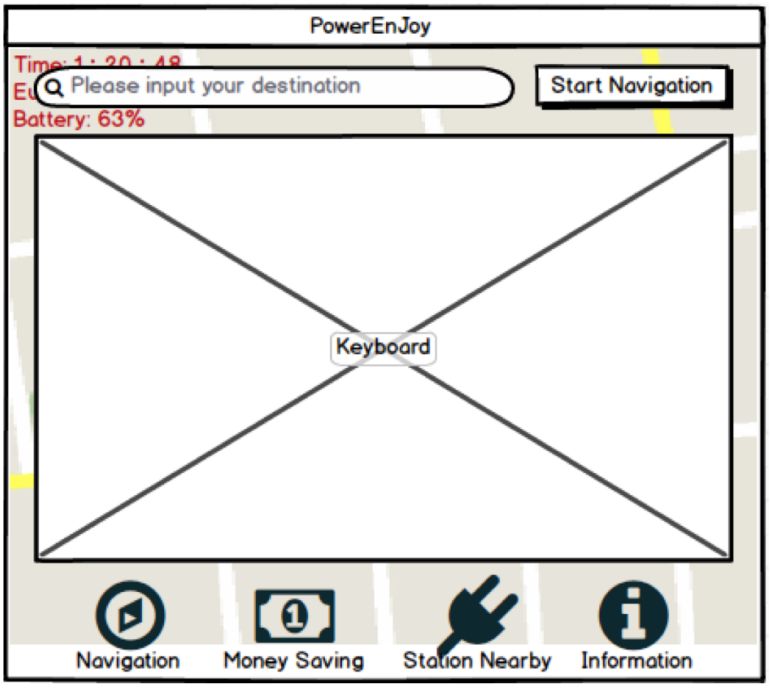

Figure 21: Input Destination

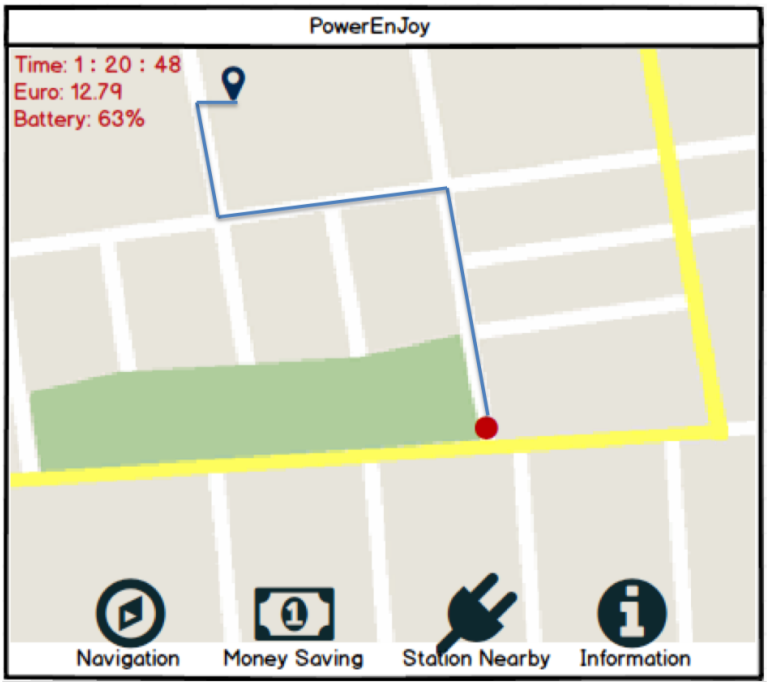

Figure 22: Navigation Path

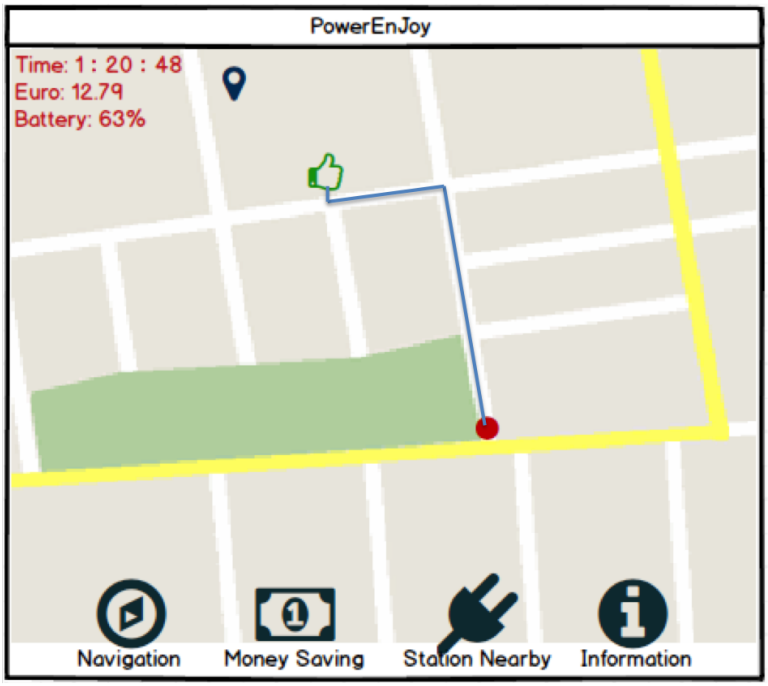

Figure 23: Money Saving Option

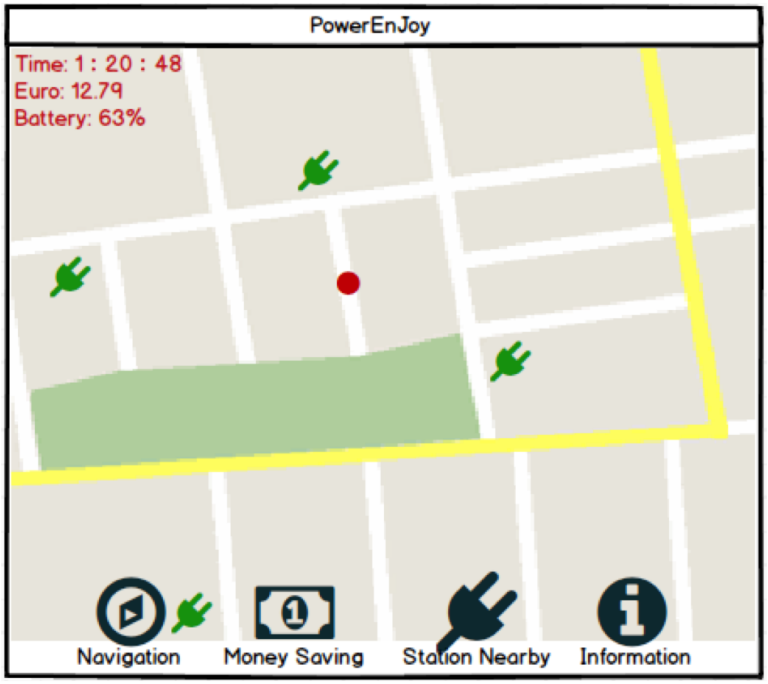

Figure 24: Search Power Station Nearby

5 REQUIREMENTS TRACEABILITY

In this section, we list the requirements specified in the previous RASD together with the components, which are designed in this document to assure its fulfillment.

Reference	Requirement	Component
[1]	Register to be a user.	Account Manager Bank System Traffic System
[2]	Get the notification about register.	Notification Manager
[3]	Login	Account Manager
[4]	Search for available cars.	Available Queue Manager Google API
[5]	Make a reservation.	Reservation Manager
[6]	Cancel reservation.	Reservation Manager
[7]	Check current reservation.	Reservation Manager
[8]	Pick up the car.	Mobile APP Reservation Manager
[9]	Access the saving money option.	Google API Optimal Path Calculator On-Board System
[10]	Get the discount and compensation.	Payment Manager

6 EFFORT SPENT

Gao Xiao	50 Hours
Kang Shuwen	50 Hours
Liubov Bolshakova	30 Hours

7 REFERENCES

7.1 Reference Documents

- Specification Document Assignments AA 2016-2017
- RASD

7.2 Used Tools

The tools used to creat this document are:

- UMLStar: for UML models
- Github: for version control
- Latex: for typesetting
- Balsamiq: for mockup design
- ProcessOn: for network diagram