POLITECNICO
MILANO 1863

Design Document

Lo Bianco Riccardo - Manzoni Mirco - Mascellaro Giuseppe

December 9, 2016
v1.0

Contents

1 Introduction

1.1 Purpose

This document represents the continuation of the RASD document for the PowerEnjoy application previosly presented. Through the discussion we will explain in depth the architecture of the the system to be deployed, then we will focus on components, algorithms developed to implement the described architecture and user interface.

1.2 Scope

The system must provide the following macro functionalities, which can be mapped from the goals presented in the RASD document

- **User functionalities** PowerEnjoy will make available for the users (both logged and unlogged) the full set of functionalities described in the RASD.
- **Operator functionalities** PowerEnjoy will make available for the operators the full set of functionalities described in the RASD.
- **Sensors management functionalities** PowerEnjoy will manage the full set of sensors installed on the cars, processing the signals and storing the data in the database.

1.3 Definitions, acronyms, abbreviations

1.3.1 Definitions

Eh qui c'è da inventarsi qualcosa nel caso, ci pensiamo man mano che scriviamo il documento

1.3.2 Acronyms

- **FR:** functional requirements
- **NFR:** non-functional requirements
- **G:** goal
- **JEE:** Java Enterprise Edition
- **AS:** application server
- **EJB:** Enterprise Java Bean
- **JB:** Java Bean

1.4 Reference documents

1. **Analysis document:**
2. **IEEE Standard for Information Technology - Systems Design - Software Design Descriptions:**

1.5 Document structure

2 Architectural design

2.1 Overview

2.2 Component view

2.3 Deployment view

2.4 Runtime view

2.5 Component interfaces

2.6 Architerctural styles and patterns

2.7 Other design decisions

3 Algorithm design

In the following paraagraph we reresent the implementation of the most significant algorithms to be used in PowerEnjoy. In particular, we chose to represent the algorithm for implementing the activation of the money saving option and the one used for the detection of bad distribution of the cars amongst the areas covered by the service. The two algorithms are expressed in an object-oriented pseudo-programming language, enriched and sometimes integrated with comments in natural language, which are useful to comprehend the logical process of the functionality to be developed without going too deep into technical details, expecially when they are related to the interaction between code and Google APIs.

3.1 Check car distribution algorithm

The car distribution amongsts different areas needs to be checked periodically inn order to avoid having some overcrowded areas and others that are almost empty at the same time. The main idea of the following algorithm is to give every area a weight that is variable in time to represent the attractivity of the area for possible customers. The perfect distribution of cars is achieved when the summation of the number of cars present in each area per area multiplied per the weight of the area is equal to a constant value represented with the name of OPTWEIGHT. If the value of the summation is higher than OPTWEIGHT+UPPERTHRESH or lower than OPTWEIGHT-LOWERTHRESH, where UPPERTHRESH and LOWERTHRESH are constants, the distribution is considered not acceptable and a notification containing the desired changes in the cars' distribution is produced.

/*Before the first iteration of the algorithm we load the data in temporary variables, so we do not change their real values while updating the variables used in the algorithm*/
for $zone$ in $zoneList$ **do**
  $carsNum = zone.getAvailableCars()$
  $zoneActWeight = zone.actWeight(date, time)$
  $zoneTotWeight = carsNum * zoneActWeight$

```
    totWeight+ = zoneTotWeight
    zoneWeightsRec.append(zoneTotWeight)
end for
maxZone = zoneWeights.getMaxIndex()
minZone = zoneWeights.getMinIndex()
while totWeight > OPTWEIGHT + UPPERTHRESH and
zoneWeightsRec[maxZone].getAvailableCars() > 2 and
zoneWeightsRec[minZone].getAvailableCars() < zoneWeightsRec[minZone].getMaxCarNum()
do
    maxZone.setAvailableCars(maxZone.getAvailableCars() − 1)
    minZone.setAvailableCars(minZone.getAvailableCars() + 1)
end while
/*We generate a message containing the right amount of cars to be moved
from maxZone to minZone*/
while totWeight < OPTWEIGHT − LOWERTHRESH and
zoneWeightsRec[minZone].getAvailableCars() > 2 and
zoneWeightsRec[maxZone].getAvailableCars() < zoneWeightsRec[maxZone].getMaxCarNum()
do
    minZone.setAvailableCars(minZone.getAvailableCars() − 1)
    maxZone.setAvailableCars(maxZone.getAvailableCars() + 1)
end while
/*We generate a message containing the right amount of cars to be moved
from minZone to maxZone*/
newTotWeight = sum(zone)forzoneinzoneWeightsRec
if newTotWeightnotinrange(OPTWEIGHT+UPPERTHRESH, OPTWEIGHT−
LOWERTHRESH) then
    /*Recursive call to the function*/
    chechCarDitribution()
else
    exit()
end if
```

3.2 Money saving option algorithm

The money saving options is meant to make expenses as low as possible for a given ride. To do so, the primary objectives are to minimize the time spent driving and to park the car in a special parking area. To achieve this, the algorithm exploits the Google APIs for navigation to produce a list of all the special parking areas in a range of K metres from the desired destination and sets the one with less cars parked as the new destination. If no SPA are found within K metres from the original destination, the algorithm is relaunched with a new K equal to the double of the original K. The process continues following this policy until K reaches a constant value indicated as MAXDIST. If this happens an exception is raised and the algorithm stops.

```
/*We refer to the APIs for navigation and geolocalization provided by Google*/
carPos = car.getCarPosition().Thefunctioniscalledpassinganargument K, representingthemaximumdi
destinationPos = getDestPosition(dest)
for spa in specParkAreaList do
```

/*The estimated cost of the ride is initially set to the maximum possible amount*/
$rideCost = MAXCOST$
if $spa.getPosition()$ in $range(destPos, destPos+K)$ and $spa.getCarsNum() < spa.getMaxCarsNum()$ **then**
$possibleSpecParkAreaList.append(spa)$
end if
end for
if $possibleSpecParkAreaList.lenght() == 0$ and $k == MAXDIST$ **then**
/*We launch an aexception signaling the impossibility of activate the money saving option and abort the operation*/
else if $possibleSpecParkAreaList.lenght() == 0$ **then**
$moneySavingOption(2K)$
else
for $pspa$ in $possibleSpecParkAreaList$ **do**
$cost = estimateRideCost(carPos, spas.getLocation(), car.getPassengersNum())$
if $cost < rideCost$ **then**
$rideCost = cost$
end if
end for
end if
/*We provide the user with the information regarding the chosen special parking area, the estimated cost of the ride and all the obtainable discounts*/
$exit()$

4 User interface design

The following paragraph is strongly connected to the information regarding user experience we already provided in the RASD. In particular, basic mockups useful to understand the interface to be provided to the user were already present in the previous document, so in the following pages we concentrated our efforts in describing the same concepts using UX diagrams. The first diagram represents the whole set of functionalities provided both to users and operators, while the others are focuses on the different aspects of the application interface.

4.1 Complete interface design