Design Document
SmartCityAdvisor

Navid Heidari (798726) Hamidreza Hanafi (841408)

Thursday 14th July, 2016
version 1.0

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Purpose

Here we present the Design Document of the application *SmartCityAdvisor*. The aim of this document is to show our design choices and the rationale behind them. We will focus mainly on architectural choices such as the subdivision into modules of our software, the deployment of this modules and the communication between them.

1.2 Scope

This document is intended to be a description of the architecture of the software system *SmartCityAdvisor*, commissioned by the government of a big city which wants to improve and automatize the some city facilities for citizens and control the special situations in city.

In particular we will present different *views* of the system representing different levels of abstraction: the user's point of view, an internal sight of the subsystems and their high level interaction and the communication interfaces that they use to interact. We will describe the chosen architectural styles and pattern, some fundamental algorithms and how our choices are mapped on the requirements elicited in the RASD (Requirement Analysis and Specification Document).

1.3 Definitions

For the definitions and the glossary we refer to the Requirement Analysis and Specification Document.

1.4 Referenced documents

- *Requirements Analysis and Specification Document - SmartCityAdvisor*
- *Project document*

1.5 Document structure

The following document is structured in three important sections:

2. The architectural description (chapter 2)

3. The algorithm description (chapter 3)

4. The user interface design (chapter 4)

5. The mapping between this design and the requirements (chapter 5)

Chapter 2

Architectural Design

2.1 Overview

Our first sight of the system is from the highest point of view. Here we describe the architecture of the system from a *layer* view.

2.1.1 An overview of our system

In the specific case of *SmartCityAdvisor* we mapped the previous architecture in this way:

- **Presentation tier**:
  - *Citizen*:
    - Dedicated Mobile application
    - Browser
  - *Control Center*:
    - Browser
- **Application tier**: main server(s) located in some office of the government of the city or (better), a virtual server hosted by some expert company and accessed through the Internet.
- **Data tier**: main database(s) located in some office of the government of the city or (better), a virtual database hosted by some expert company and accessed through the Internet.

v

2.2 High level components and their interaction

We can sketch our system like in figure 2.1

Here we can see the different components, both hardware and software, and how they are interconnected from a very high point of view.

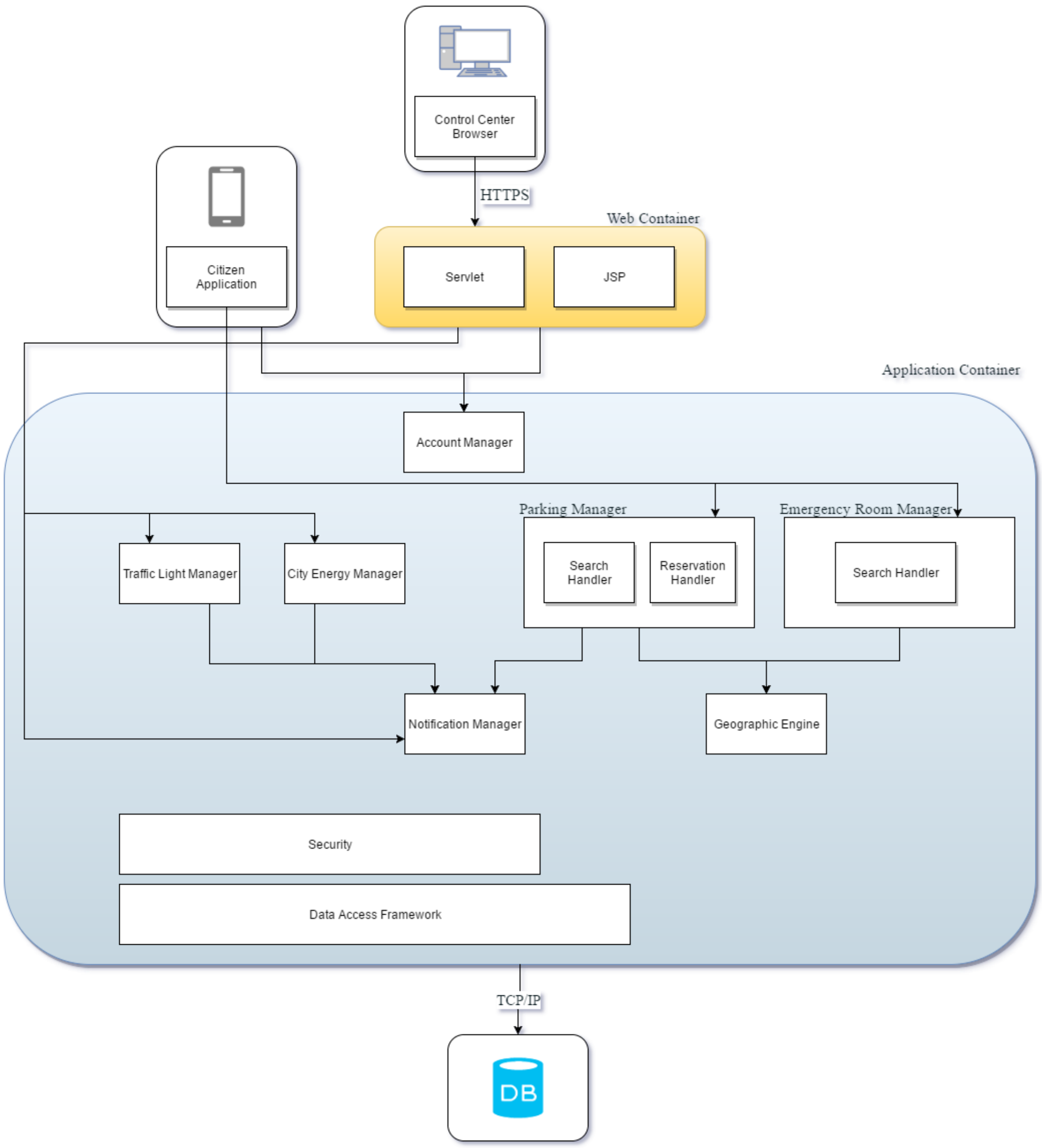


Figure 2.1: Architecture Overview

In particular the web browser will interface the business logic through a web server, which implements services like Servlets and JSP. In this case, so, the chosen protocol will be HTTPS. As far as mobile applications are concerned they will interface the system through remote calls.

The application server will be divided in software components differentiated by their role in the implementation of the business logic. For data access and security will be used ready-made components provided by the JEE infrastructure.

The application server and the data layer will communicate through a TCP/IP connection.

2.2.1 Business logic components

In figure 2.2 we find an overview of the business logic components of our application.

1. **Account Manager**: It manages the accounts, both of Citizens and Control Center. It also handles the login and logout phases of both users and the registration for the Citizen and the relative account deleting. It exposes the following interfaces:
   - `Citizen`: for the interaction with the Citizen
   - `Control Center`: for the interaction with the Control Center
2. **Emergency Room Manager**: this component is delegated to the management of the search system for emergency room with respect to user location and problem It exposes the following interfaces:
   - `EmergencyRoomHandler`: which can be used by the Citizen mobile application and the web browser (web server)
3. **Parking Manager**:this component is delegated to the management of the life cycle of the incoming parking searches and reservations from the citizens. From this point of view it can be divided in two main subcomponents:
   - `SearchHandler`
   - `ReservationHandler`

   It exposes the following interfaces:
   - `ParkingHanlder`: which can be used by the Citizen mobile application and the web browser (web server)
4. **Traffic Light Manager**: manages the status of traffic lights so we can limit the traffic of the city center.
   It exposes the following interfaces:
   - `LightManagement`: for the interaction with the Control Center
5. **City Energy Manager**: manages the city energy consumption so we can reduce the CO^2 emission by reducing this consumption.
   It exposes the following interfaces:
   - `EnergeyManagement`: for the interaction with the Control Center

6. **Geographic Engine**: the roles of this component is to find shortest path and public transport to the desired location.
   It exposes the following interfaces:

   - `ATMCalculator`: which provides client API call from ATM to get best public transportation route to hospital
   - `TimeCalculator`: which provides approximative traveling time calculation methods

7. **Notification Manager**: the roles of this component is to notify the Citizens about various conditions in the city as well as reminder of their reservations.
   It exposes the following interfaces:

   - `NotificationHandler`: to interact with various modules

8. **Security**: a ready-made software component to ensure the safety of tcp/ip connections from the mobile devices and from the web browser

9. **Data Access**: framework that abstracts the underlying database logic and enables to map the business logic entities into tables.

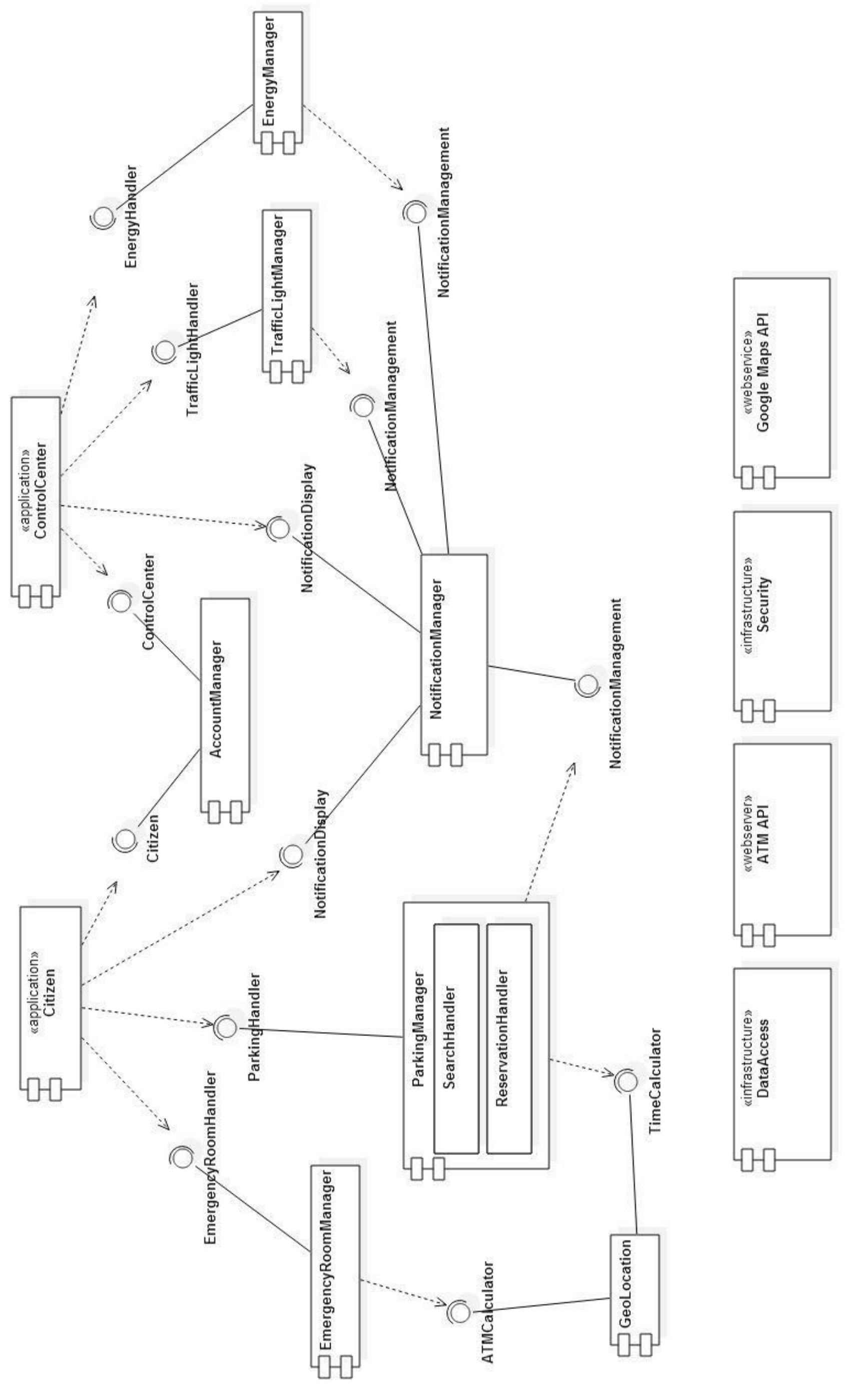


Figure 2.2: High level components of the business logic

2.3 Component view

Now we refine the description of the components of the system.

2.3.1 Account Manager

Internal Components

The Account Manager is responsible for all the actions related to users and their account, such as login, logout registration and deletion of an account. Internally it is composed of the following components each with a precise job:

- *AccountController*: it exposes the interfaces for external calls and manages them. It has access to the Data Access interface in order to add or remove accounts, and to check the validity of credentials during the login phase.
- *SessionManager*: during the login phase, if the credentials has been recognized by the AccountController, the session manager, through the Security layer, provides a valid session to return to the citizen application for future interactions.

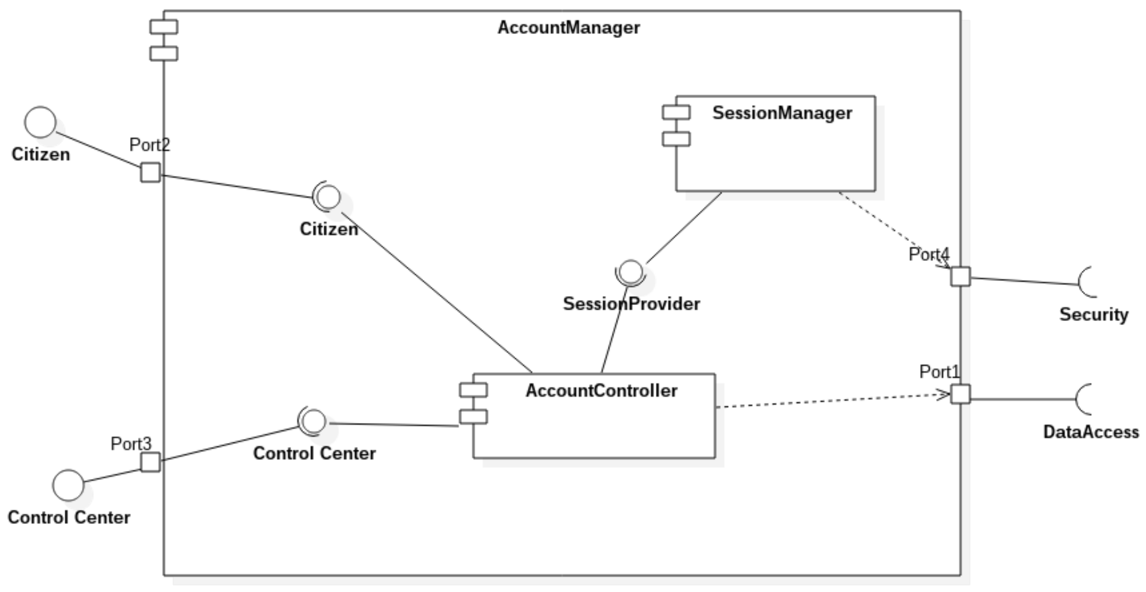


Figure 2.3: Account Manager

Provided interfaces

Provided Interface	**Dedicated user**	**Description**
Citizen	Citizen's application and relative web application	Login, logout, registration and deletion of the account
Control Center	Control Center's application	Login, logout

Table 2.2: Account Manager: provided interfaces

Required interfaces

Required Interface	**Description and usage**
DataAccess	Access to the data layer in order to retrieve the account information of the Citizens and the Control Center
Security	It asks the security layer to provide for a session during a log in request

Table 2.4: Account Manager: required interfaces

2.3.2 Emergency Room Manager

Internal Components

The Emergency Room Manager is responsible for finding best emergency room for Citizens. Internally it is composed of the following components each with a precise job:

- *Search Handler*: it searches in hospital emergency rooms and find the one which is near and has the specific user's problem.

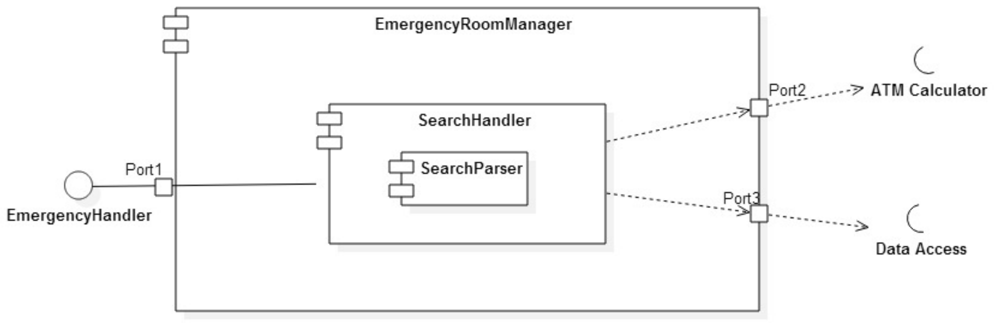


Figure 2.4: Emergency Room Manager

Provided interfaces

Provided Interface	**Dedicated user**	**Description**
EmergencyHandler	The Citizen's application and the relative web application	search of a emergency room

Table 2.6: Emergency Handler: provided interfaces

Required interfaces

Required Interface	**Description and usage**
DataAccess	Access to the data layer in order to • Store data of the search • Retrieve emergency rooms near with specific specialization
ATM Calculator	Check the ATM Api for best public transportation in the path provided

Table 2.8: Ride Handler: required interfaces

2.3.3 Parking Manager

Internal Components

The Parking Manager is responsible for the for the management of the life cycle of parking searches and reservations. Internally it is composed of the following components each with a precise job:

- *ParkingDispatcher*: it exposes the interfaces for external calls and dispatches them to the right subcomponent depending on the nature of the call.
- *SearchHandler*: it searches in parking spots in city center with respect to user's location and find the one which is near and the time is needed for citizen to reach there.
- *ReservationHandler*: for every reservation it receives, it first check for the validity of the location, then if this is valid it stores the reservation through the Data Access. The reservation should be one hour before the reserve time.

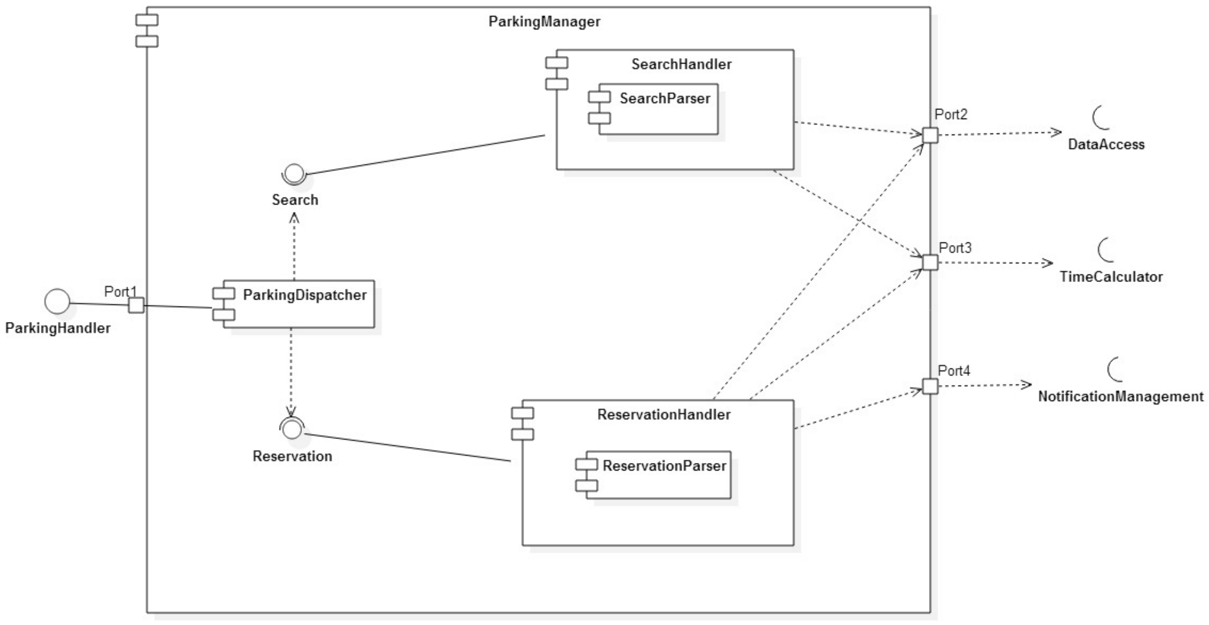


Figure 2.5: Parking Manager

Provided interfaces

Provided Interface	**Dedicated user**	**Description**
ParkingHandler	The Citizen's application and the relative web application	Search and reserve parking spots

Table 2.10: Parking Handler: provided interfaces

Required interfaces

Required Interface	**Description and usage**
DataAccess	Access to the data layer in order to • Store data of the search • Store data of the reservation • Retrieve reservations
Time Calculator	Calculate the approximative reach time
NotificationManagement	Sends to the citizen the info for the parking spot information which he reserved.

Table 2.12: Parking Handler: required interfaces

2.3.4 TrafficLight Manager

Internal Components

The Traffic Manager is responsible for the limiting the traffic in the city center through managing traffic lights. It also inform citizens about the changes :

- *LightManager*: it exposes the interfaces for external call of control center to limit the traffic or to abort the mission.

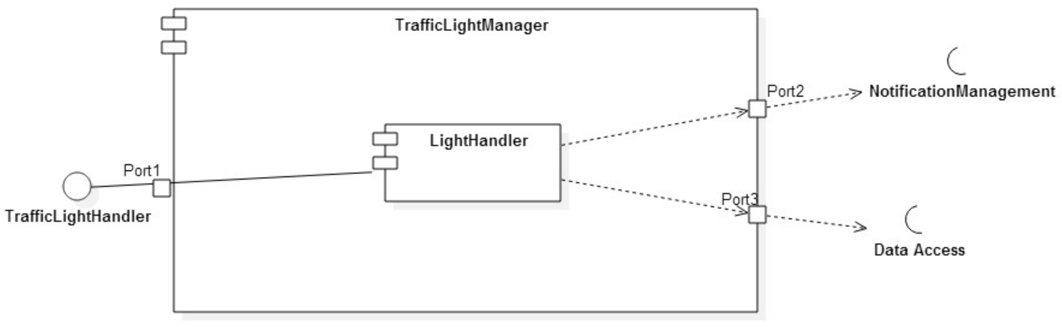


Figure 2.6: TrafficLight manager

Provided interfaces

Provided Interface	**Dedicated user**	**Description**
TrafficLight Handler	The Citizen's application and the relative web application	Change traffic lights patterns

Table 2.14: TrafficLight manager: provided interfaces

Required interfaces

Required Interface	**Description and usage**
DataAccess	Access to the data layer in order to • Retrieve traffic lights status • Store data of new lights pattern
NotificationManagement	Sends to the citizen the info about changes in traffic.

Table 2.16: Traffic Manager: required interfaces

2.3.5 Energy Manager

Internal Components

The Energy Manager is responsible for the reducing the CO_2 in the city center through managing city lights and etc. It also inform citizens about the changes :

- *EnergyHandler*: it exposes the interfaces for external call of control center to reduce the energy or to abort the mission.

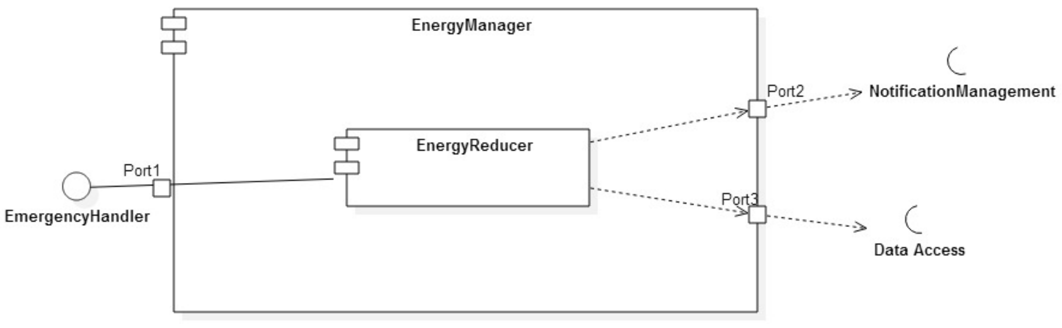


Figure 2.7: Energy manager

Provided interfaces

Provided Interface	**Dedicated user**	**Description**
Energy Handler	The Citizen's application and the relative web application	Reduce energy by city lights and etc

Table 2.18: Energy manager: provided interfaces

Required interfaces

Required Interface	**Description and usage**
DataAccess	Access to the data layer in order to • Retrieve CO_2 status
NotificationManagement	Sends to the citizen the info about changes in city energy use.

Table 2.20: Energy Manager: required interfaces

2.3.6 Notification Manager

Internal Components

The Notification Manager is responsible for the communication with citizens and control center. Internally it is composed of the following components each with a precise job:

- *NotificationParser*: it exposes the interfaces for external calls and dispatches them to the right target citizes.

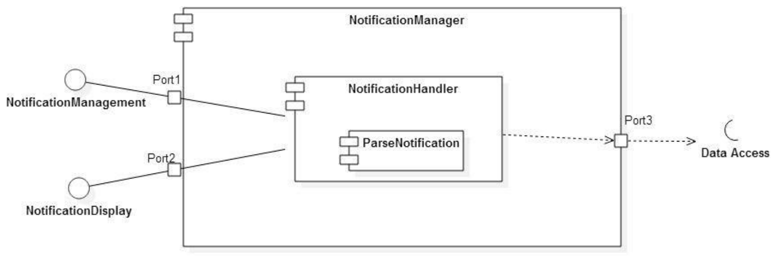


Figure 2.8: Notification Manager

Provided interfaces

Provided Interface	**Dedicated user**	**Description**
Notification Management	The Citizen's application and the relative web application and ControlCenter web application	Send, SendToAll
Notification Display	The Citizen's application and the relative web application and ControlCenter web application	Show

Table 2.22: NotificationManager: provided interfaces

Required interfaces

Required Interface	**Description and usage**
DataAccess	Access to the data layer in order to • Store data of the notification • Retrieve the notification

Table 2.24: NotificationManager: required interfaces

2.3.7 GeoLocation Engine

The Geographic Engine is responsible for calculating the approximative waiting time for a citizen, and for computing the series of ATM public transportation to the destination. Internally it is composed of the following components each with a precise job:

- *TimeCalculator*: when it receives a call, it forwards the source and the destination to a Google Maps Web Service which computes the approximative traveling time.
- *ATMCalculator*: it is responsible for finding a series of public transportation to destination using ATM API.

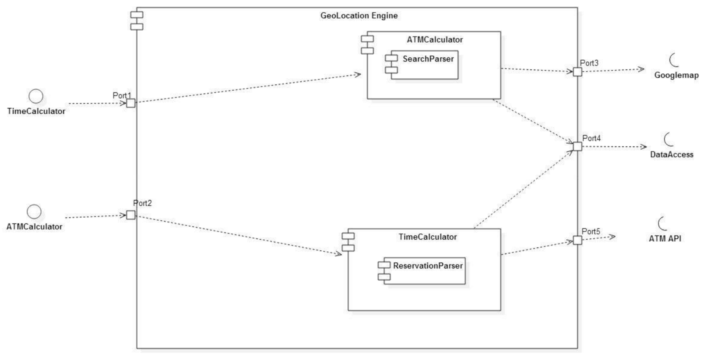


Figure 2.9: GeoLocation Engine internal structure

Provided interfaces

Provided Interface	**Dedicated user**	**Description**
TimeCalculator	ParkingManager + EmergencyRoomManager component	Given a source location and a destination, it calculates an approximative waiting time
ATMCalculator	EmergencyRoomManager component	It find the best public transportation

Table 2.26: GeoLocation Engine: provided interfaces

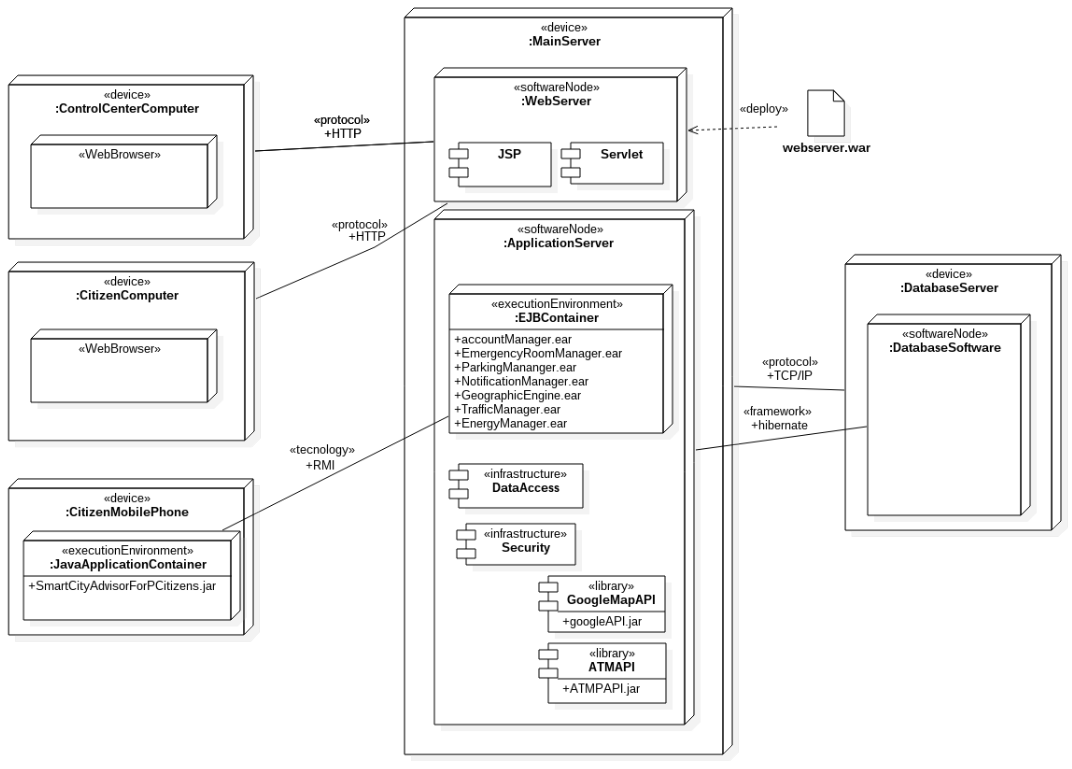
Required interfaces

Required Interface	**Description and usage**
Google Maps API	Used for the activity of geocoding and for calculating the approximative travel time
ATM API	Used for the activity of finding a series of ATM public transportation
DataAccess	Retrieve the information Hospitals and etc

Table 2.28: GeoLocation Engine: required interfaces

2.4 Deployment view

In this section we show how our components are really deployed on hardware devices.



The UML diagram is self-explicative. We organize our deployed files like this:

- *Client side* we show the devices that can interact with our system: a mobile application and a browser for the Citizen, and a browser for the Control Center.
  - The mobile application interact with the system through RMI calls
  - The browser interact with the Web Server through HTTP protocol
- *Server side* both the web server and application server are deployed possibly in a cloud service, this will ensure the system to be reliable 24/24 7/7. The Application Server will keep all the components in a EJB pool so that it can handle the increased work load during specific time of the day, by deploying multiple instances of the stateless beans. It will also use the already specified infrastructural libraries for data access, security, ATM API and the Google Maps API.
- The data are stored in multiple external *databases* accessed by the Main Server through the Hibernate framework. There will be more than one instance of the database in order to make the whole system more reliable. Hibernate will abstract the fact that more than one database is been used, so the system is not aware of this.

2.5 Runtime view

The components of the system interact with each other, in order to carry out the various activity that the system must has to accomplish. The activity are explained from different perspective as necessary.

2.5.1 Citizen emergency search

Components involved and their role:

- **Citizen (Application)**: the activity starts when the citizen, from the mobile application, submits the search request through the relative form. When the button is pressed, the client calls the method newSearch() of the EmergencyRoom Manager component passing as parameters all the data of the form. He then expects as response either an affirmative message with the emergency room info, or a negative message because there are no available emergency room for the location provided.
- **EmergencyRoom Manager**: once the EmergencyRoom Manager has been called by the citizen, it firstly check whether the emergency room is available for his problem, then check the path with ATM API and return the info to citizen.
- **Geographic Engine**: it checks for the public transportation path with ATM API.

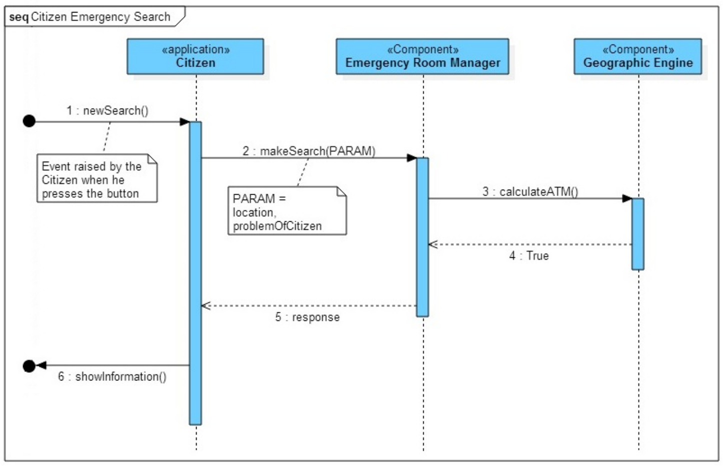

seq Citizen Emergency Search
«application» Citizen
«Component» Emergency Room Manager
«Component» Geographic Engine
1 : newSearch()
Event raised by the Citizen when he presses the button
2 : makeSearch(PARAM)
PARAM = location, problemOfCitizen
3 : calculateATM()
4 : True
5 : response
6 : showInformation()

Figure 2.10: Sequence Diagram of the search

2.5.2 Citizen makes a parking Reservation

Components involved and their role:

- **Citizen (Application)**: the activity starts when the citizen, from the mobile application, submits the reservation through the relative form. When the button is pressed, the client calls the method makeReservation of the Parking Manager component passing as parameters all the data of the form. He expects as a response the outcome of the submission. If there is a parking spot available it show him the list and when he choose the parking will be reserved for citizen.

- **Parking Manager**: once the Parking Manager as been called by the citizen, it sends to the citizen the outcome of the reservation.
- **Geographic Engine**: It checks the distance to the reservation area and calculate the time to reach there.
- **Notification Management**: It sends a notification to user 1 hour before reservation as a reminder.

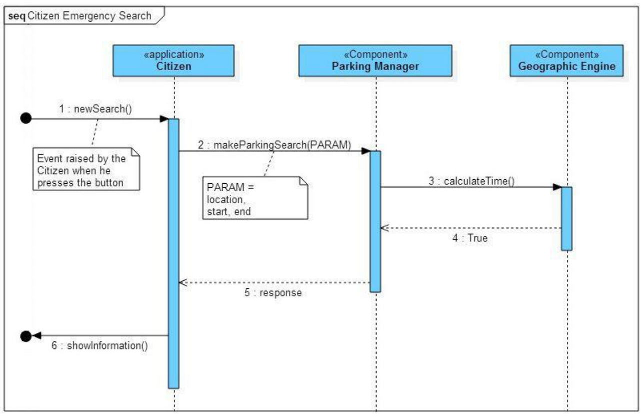

seq Citizen Emergency Search
«application» Citizen
«Component» Parking Manager
«Component» Geographic Engine
1 : newSearch()
Event raised by the Citizen when he presses the button
2 : makeParkingSearch(PARAM)
PARAM = location, start, end
3 : calculateTime()
4 : True
5 : response
6 : showInformation()

Figure 2.11: Sequence Diagram of the reservation

2.6 Component interfaces

Here we present the interfaces of our components: in particular which operations they offer, their meaning and their input/output parameters.

2.6.1 Account Manager

Citizen

- `Citizen login(String username, String password)`
  Given valid credentials, allows the citizen to receive a session and to login.
- `void logout(Citizen citizen)`
  Given a valid session, it deletes it.
- `boolean register(String username, String password)`
  It creates a new account for a citizen with the credentials provided.
- `boolean deleteAccount(String username, String password)`
  Given valid credentials, it deletes the account associated.

ControlCenter

- `ControlCenter login(String username, String password)`
  Given valid credentials, allows the control center to receive a session and to login.

- `void logout(ControlCenter controlCenter)`
  Given a valid session, it deletes it.

2.6.2 EmergencyRoom Manager

EmergencyRoomHandler

- `Response makeSearch(Location location, Problem problem`
  `Citizen citizen)`
  It starts the activity of making a search. It returns either informations about the emergency room and data of public transportation, or an invitation to try later.

2.6.3 Parking Manager

ParkingHandler

- `Response makeSearch(Location location,`
  `Citizen citizen)`
  It starts the activity of making a search. It returns either informations about the parking spot, or an invitation to try later.

- `Response makeReservation(Location location,`
  `, Date date, Time time, Citizen citizen)`
  It starts the activity of making a reservation.It check to best parking near location provided. If the time of the reservation is at least 1 hours later, it return a positive message. If there is no parking available or time is not 1 hour before it return negative message.

- `List<Reservation> getReservations(Citizen citizen)`
  It returns all the reservations of a given citizen.

- `boolean deleteReservation(Citizen citizen, Reservation reservation)`
  It deletes a reservation of a citizen.

2.6.4 TrafficLight Manager

TrafficLightHandler

- `Status getStatus(TrafficLight light)`
  It gets the status of a traffic light.

- `void setStatus(Status status, TrafficLight light)`
  It sets the status of a traffic light.

2.6.5 Energy Manager

EnergyHandler

- `EnergyConsumption getEnegryConsumption()`
  It gets the energy consumption of the city.

- `void setConsumption(EnergyConsumption consumption)`
  It sets the energy consumption of the city.

2.6.6 Geographic Engine

ATMCalculator

- `Zone getPublicTransportation(Location source, Location destination)`
  It computes the best public transportation from source to destination.

TimeCalculator

- `Time getTimeEstimation(Location source, Location destination)`
  Given valid source and valid location, it returns an approximative traveling time.

2.6.7 Notification Manager

NotificationManagement

- `void setNotification(Message message, Subject subject, Boolean sendToAll, Citizen citizen)`
  It set a new a notification to display.

NotificationDisplay

- `void showNotification(Notification notification)`
  It shows the notification to target user.

2.7 Architectural Styles and Patterns

2.7.1 Client - Server

The application main architecture uses the *client-server* style.

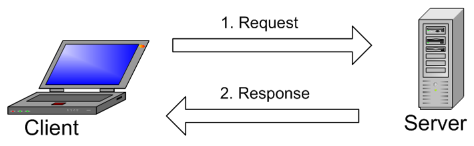


Figure 2.12: Interactions in a client-server architecture

The reasons for this choice are:

- The *centrality of the server* and the *sparsity of the clients*: we have different users that need some type of service provided by a particular organization
- The *absence of business logic client-side*: our user devices (web browser and mobile applications) must not need to know the back-end logic of the services they are invoking. This ensure scalability and the possibility to add new services or modify the already present with a minimum effort in the update of the applications client side.

The interaction of a client-server system has the client as initiator of the communication, which makes a *request* to the server. The server, received the request, elaborate a *response* using its internal business logic, and finally send it to the client.

We made the design choice to have *thin* clients, which leads to the following advantages:

- The client applications (and of course, the web pages) are easily modifiable
- The client applications are power consumption optimized
- Easy installation of the app and increased download speed of it (both mobile application and web pages).

2.7.2 The problem of the notification to the clients

A great design issue raised by the requirements of the application is *how the server can contact the clients* (both citizens and control center). A citizen is contacted by the server when a reservation of him is ready. A control center is notified by the server when the CO_2 is high or when there is special conditions. We notice immediately that this interaction is basically asynchronous.

This model of interaction is in conflict with the concept of *client - server* style.
Anyway this conflict is only at the modeling level because we can *simulate* this interaction with a low level *polling* policy which uses the client-server paradigm with some optimization in order to minimize the number of requests carried out by the client to the server.

Anyway the high level architectural style for this kind of interaction is basically a variant of the *publish-subscribe* (event-based).

From the point of view of implementation there are lots of *messaging frameworks* that can be used: JMS maybe is the most natural choice if we would choose the JEE infrastructure.

These design choices have been chosen in order to *reduce* the number of request sent by the clients to the server, which are the main drawback of a client-server approach.

2.7.3 Three-tier-architecture

We decided to use a *three-tier-architecture* for our system.
This is simply a specialization of the client-server architecture in which we specify the different layers and components of our system.
A schematic representation of this responsibility distribution is at figure 2.13

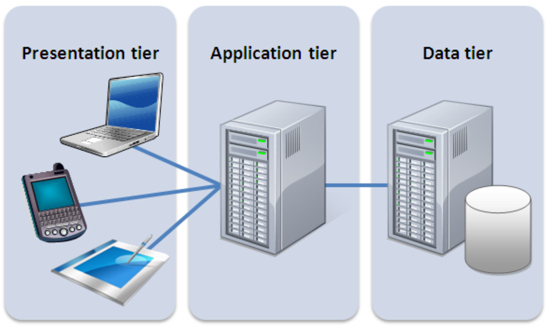


Figure 2.13: A representation of a three-tier-architecture

So we subdivide our system in three extremely separated layers (or *tiers*):

- A *presentation* layer for the graphic rendering of the data and events generated by the system, and from which the end user can interact with the system
- An *application* layer entitled to manage all the business logic of the system
- A *Data* layer responsible of the storage of informations to be used by the application and presentation layer

The three levels are completely independent and can be replaced easily. In particular, the presentation layer cannot communicate directly with the data tier, but it must forward its requests to the application one, which will use some data access framework to access the database.

The reasons of this architectural choice are:

- The possibility of use *well-defined interfaces* between the different layers. Each layer is dependent uniquely on the interfaces with the other elements of the system. This ensure scalability and, from the data layer side, the possibility of replication and clouding of the resources.
- This subdivision of roles makes the defining of the required *programmatic interface* easier. Each layer and each subcomponent will expose an interface which can be part of an API to be used by the future developers in order to improve the services and create new ones.

2.8 Other design decisions

Subdivision into components

From the previous sections is clear that all our architectural design is highly oriented to the subdivision of the entire system into submodules, having each of them a different role in the achieving of the requirements of our application. We can call this approach *component-oriented.* It is basically an application of the *divide and conquer* design principle.

Dependency inversion principle

We have taken care of always provide interfaces for the subcomponents applying so the *dependency inversion principle.* The various components of the system depend only on the interfaces of the others and never on the internal representation.

Deployment choices

As far as the deployment is concerned, we have already seen that our choice is to deploy both the business logic (application server) and the web management (web server) on the same machine.
We must say that our RASD did not specify any constraint about deployment and so we opted for the easiest and cheaper solution for our clients.
A future enhancement may be to delegate the data logic to a clouding infrastructure with a remote server. This possible choice will bring the following advantages:

- *Division of responsibility*: the data storage is delegated to a specialize provider which can manage it with advanced control systems. This feature brings to the following advantages.

- *Security*: to have the data stored far away from the Main Server enhance security
- *Easy scalability* of the database

Programmatic interface

For the satisfiability of the requirement to have a programmatic interface that can make the developers able to build new functionalities on the top of the already provided, we decided to expose all the component interfaces that we have described (figure 2.14).

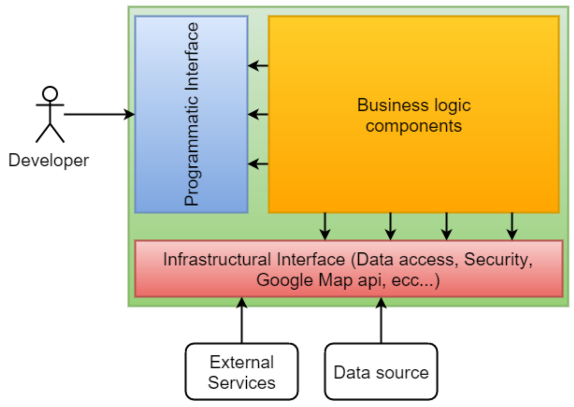

Developer
Programmatic Interface
Business logic components
Infrastructural Interface (Data access, Security, Google Map api, ecc...)
External Services
Data source

Figure 2.14: Schematic representation of the role of the programmatic interface in the system

Chapter 3

Algorithm Design

In this section we clarify some fundamental algorithm used by our system in order to achieve its objectives. We will deal basically with high level interactions between the components described in the previous sections.
All the algorithms are presented without any reference to a particular programming language but we are assuming an object-oriented paradigm for it.

Since all the architecture is designed for a client - server interaction an obvious choice of framework would be the *Java Enterprise Edition* framework and so the usage of the Java programming language.

3.1 Password Hash and Salt

This algorithm is a part of the AccountManager component and will be used for generating the hash of a password. The password must be combined with a randomly generated salt. In the database must be stored both the salt and the generated hash.

Algorithm 1 Hash for user password

```
import java.security.MessageDigest;
public byte[] hashPW(String pw, byte[] salt) throws NoSuchAlgorithmException {

   // Use sha1 with multiple iterations for generating the final hash
   MessageDigest digest = MessageDigest.getInstance("SHA-1");
   digest.reset();
   digest.update(salt);
   byte[] input = digest.digest(pw.getBytes("UTF-8"));
   // 1000 iteration will slow down an attacker
   for (int i = 0; i < 1000; i++) {
      digest.reset();
      input = digest.digest(input);
   }
   return input;
}
```

3.2 Reservation Handler

The nature of the problem of managing the reservation waiting time before the sending the notification to user requires a *multi-threaded* solution.
We design the following algorithm to be the conjunction of behaviors of 3 different threads:

- *Reservation Receiver*
- *Waiter*
- *Reservation Forwarder*

3.2.1 Reservation Receiver thread

Algorithm 2 *Reservation Receiver*: Adding of a reservation to the data structure

```
Require: Input:
    • reservation (data structure containing: citizen data, time and date, location, duration)
  Already present data structure:
    • reservationList: ordered list containing all the reservations waiting to be notified to the
      citizen. The order is done by the time specified.
Ensure: The reservation is added in the correct order in the reservationList and the timer of
  the Waiter is set correctly
  $meetingTime = reservation.meetingTime$
  $reservationList.PUT(reservation)$ {the reservation is put in order by meeting time}
  $index = reservationList.GETINDEX(reservation)$
  if index is the first then
    Sends a signal to the Waiter in order to stop is current timer (signal)
    $waitingTime = meetingTime - CURRENT\_TIME$
    Communicate to the Waiter thread the waitingTime
  end if
```

3.2.2 Waiter thread

Algorithm 3 *Waiter*: setting of the timer and timer termination

Require: Already present data structure:

- *timer*: is a counter that can be set as ON/OFF and which can raise an signal when the set time expires.

Ensure: The correct setting of the timer

```
$timer.count = 0$ {First initialization of the timer}
while true do
  if a signal is received then
    $waitingTime$ = the parameter communicated by Reservation Receiver
    $timer.SETTIME(waitingTime)$
    $timer.START()$
  end if
  if timer expires then
    Send a communication to the Reservation Forwarder
    $timer.STOP()$
  end if
end while
```

3.2.3 Reservation Forwarder thread

Algorithm 4 *Reservation Forwarder*: Forwarding of a reservation to the RequestQueueManager

Require: Already present data structure:

- *reservationList*: ordered list containing all the reservations waiting to be notified to the citizen. The order is done by the time specified.

```
while true do
  Wait for a signal by the Waiter thread
  A signal is received
  $first$ = the first index of $reservationList$
  $firstReservation = reservationList.GET(first)$ {Next reservation is got from the list}
  $reservationList.REMOVE(firstReservation)$ {Next reservation is removed from it}
  $nextReservation = reservationList.GET(first)$
  $waitingTime = nextReservation.meetingTime - CURRENT\_TIME$
  Communicate to the Waiter thread the $waitingTime$
  Forwards the $firstReservation$ to the RequestQueueManager
end while
```

Here we presented a pseudo-code of the behavior of this three threads. Each language, in particular Java, has its way to implement threads and waiting conditions. In this document we don't care about implementation.

Chapter 4

User interface design

We have already dealt with the interface design in the Requirements Analysis and Specification Document, where we have shown some mock-ups of the screens of our applications.
In this section we will refine the user interface basically from the point of view of *interaction* with the end-user (mapping between a sequence of actions and a screen flow).

We will use the class diagram for the specification of screen flows. This approach need some clarification about the used notation:

Symbol	**Meaning**
Directed arrow with *function()* over it	It defines a transition from a particular user interface element to another. This transition is triggered by the invoking of a particular method *function()* in the source interface element.
Class symbol	It determines a specific user interface element from the set {*Screen*,*Form*,*Element*,*Pop-up*} or a specific event (stereotype *event*). In both cases the type of element is specified in the stereotype of the class symbol.
Composition symbol	Means that a specific user interface element is contained in another. Typically is used when a form or a list of elements is contained in a particular screen.
Directed arrow with the stereotype *event* over it and a *text*	Means that it is not a user induced function to trigger the transition, but an event from the software, in particular an event sent by the server to the client. The *text* describe the event.

Table 4.2: Notation for the user interface flow diagram

4.1 Citizen App

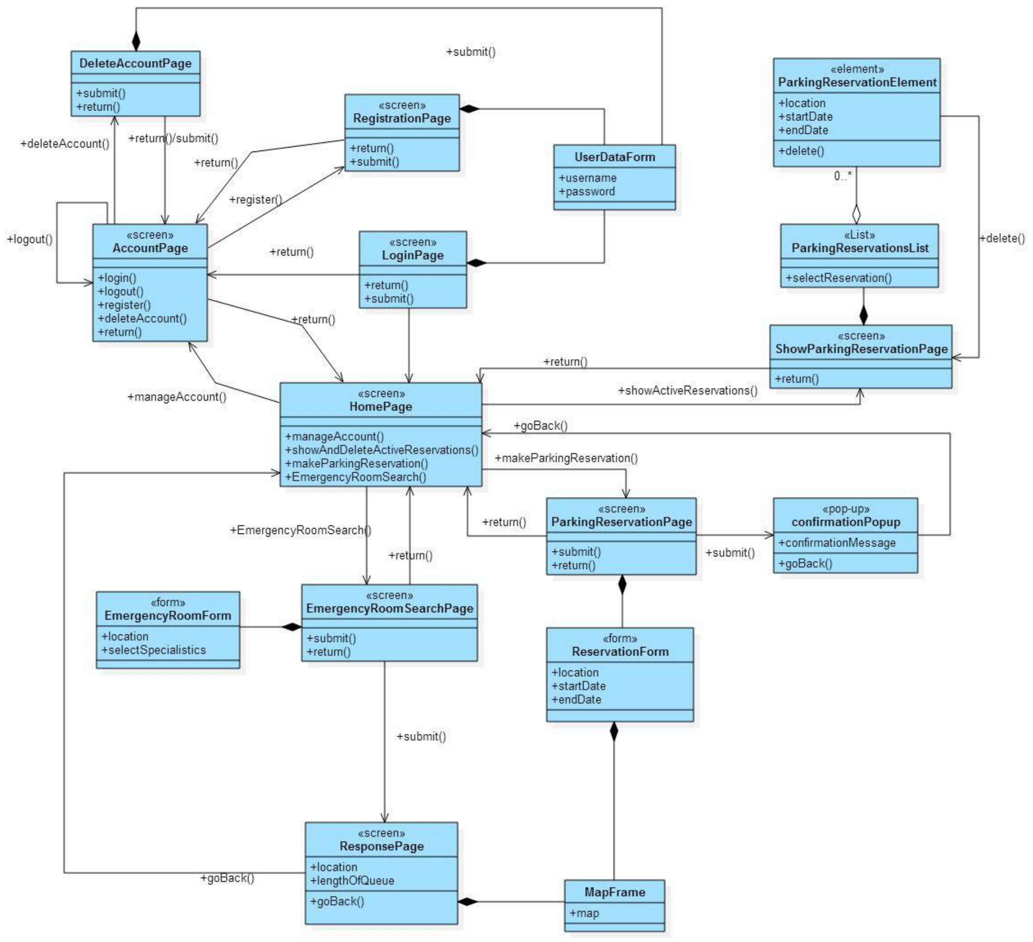


Figure 4.1: Screen flow of the Citizen mobile application

4.2 ControlCenter App

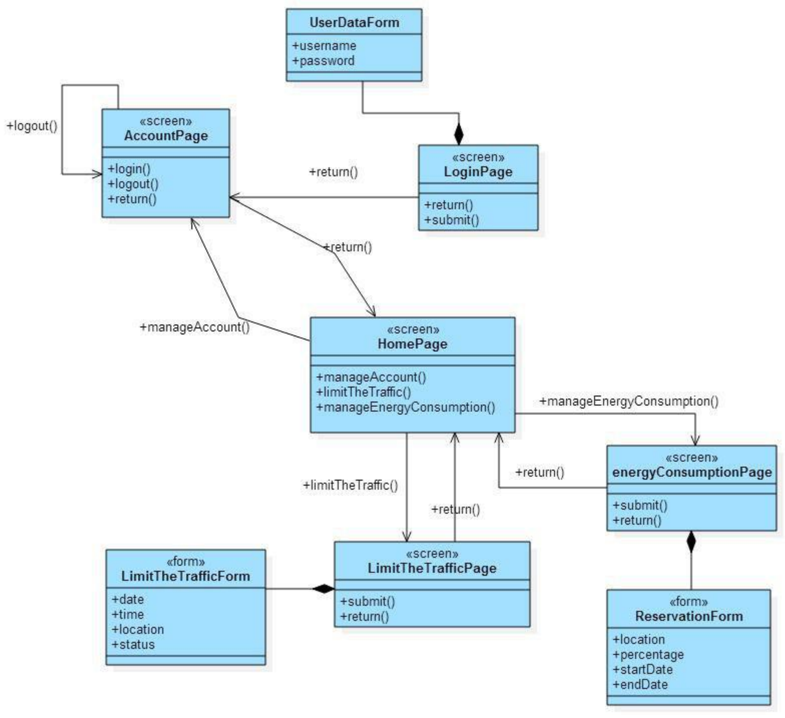


Figure 4.2: Screen flow of the control center web application

Chapter 5

Requirements Traceability

Requirement		Design Solution	
R.C.1	Citizens can't register twice with same username	CMP: Account Manager	The AccountManager manages the login and registration phase, avoiding this to happen
R.C.2	Username provided by citizen in registration phase must not be empty	CMP: Account Manager	It makes this check
R.C.3	The password provided in the registration phase must not be empty	CMP: Account Manager	It makes this check
R.C.4	The system must notify and abort the registration procedure in the previous 3 cases	CMP: Account Manager	The $register(username, password)$ method provided by the Account Manager will return an error in any of the cases and the citizen GUI will show an appropriate pop-up.
R.C.5	The system must provide for the Citizen with a way to abort the registration procedure	UX: Citizen App	Will be provided, in the user interface, a specific button for stopping the registration procedure
R.C.6	If a citizen makes a emergency room search and the hospital queues are not empty, then the system must sooner or later respond positively to the Citizen	SD: Citizen emergency search	The SearchHandler will return a hospital to who search for it if the hospital queue is empty, and an error message if not empty

R.C.7	A reservation must be refused if: • location is invalid • time(parking time) - time(reservation) $<$ 1 hours	CMP: ParkingManager SD: Reservation	The ReservationHandler inside the ParkingManager will use its parser and will return an error in case the the time of parking is less than 1 hour of reservation time or the location is invalid.
R.CC.1	A ControlCenter, must notify Citizen's when there is an event in the city	CMP: NotificationManager	It checked for the notifications and send them to users
R.CC.2	A ControlCenter, must be always available.	CMP: AccountManager	The account manager don't let a control center logout if it's the only one
R.CC.3	If CO_2 emission is high limiting or reducing must be active.	CMP: EnergyManager TrafficManager	One of these modules should be active in case

Table 5.1: Requirements traceability table

Chapter 6

Appendix

6.1 Working hours

- Navid Heidari: 30 hours
- Hamidreza Hanafi: 30 hours

References

- ISO/IEC/IEEE 42010:2011(E), Systems and software engineering - Architecture Description
- IEEE Std 1016-2009 (Revision of IEEE Std 1016-1998) IEEE Standard for Information Technology - Systems Design - Software Design Descriptions