Politecnico di Milano
A.A. 2015-2016
Software Engineering 2: “MyTaxi”
Design Document

Manzi Giuseppe (mat. 854470) &
Nicolini Alessandro (mat. 858858)

CONTENTS

1. Introduction

1.1 Purpose

The purpose of this document is to provide the design and the architecture of the application, describing and justifying the reason of our choices. A design is a conceptualization of the system that embodies its essential characteristics, demonstrates a means to fulfil its requirements, serves as a basis for analysis and evaluation and can be used to guide its implementation.

This document also shows the architecture of the system we are implementing, so its fundamental concepts and properties in its environment embodied in its elements, relationships, and in the principles of its design and evolution.

The document is intended for both developers, who will implement the software, and manager, who need understand the high-level structure of the system.

1.2 Scope

The scope of the design model of the system that we will describe in this document is to fulfil the requirements described in the RASD. Critical points are:

- The system must be web-based
- The system must handle with many users at the same time.
- The software we will develop will be a quite large-scale application that can become even larger if future implementations will be realized.
- The application must provide efficient algorithms to manage queues.

1.3 Definitions, Acronyms, Abbreviation

1.3.2 Acronyms

RASD: *Requirements Analysis Specifications Document.*
DB: *Data Base.*
DBMS: *Data Base Management System.*
MVC: *Model View Controller.*
RQI: *Request Queue Interface.*
TQI: *Taxi Queue Interface.*
UI: *User Interface.*
PRI: *Past Request Interface.*
VI: *View Interface.*

1.4 Reference Documents

- *Software Engineering: Principles and Practice*
  - *Author: Hans Van Vliet – Wiley ,2007*
- *DD TOC pdf*
  - *Author: Raffaela Mirandola*
- *Java EE pdf*
  - *Author: Clement Quinton*

- *SOFTWARE ARCHITECTURES AND STYLES pdf*
  - *Author: Damian Andrew Tamburri*
- *Design I-II pdf*
  - *Author: Raffaela Mirandola*
- *IEEE Standard for Information Technology-System Design-Software Design Descriptions*
  - *Author: IEEE Computer Society*
- *IEEE Standard for System and software engineering-Architecture description*
  - *Author: IEEE Computer Society*
- *RASD.pdf*
  - *Author: Nicolini Alessandro, Manzi Giuseppe*
- *Web resources:*
  - http://www.oracle.com/technetwork/java/javaee/overview/index.html
  - https://docs.oracle.com/javaee/7/api/toc.htm

1.5 Document Structure

Our Design Document will contains the following information:

- The **architecture design:** uses information flowing characteristics, and maps them into the program structure. The transformation mapping method is applied to exhibit distinct boundaries between incoming and outgoing data. The data flow diagrams allocate control input, processing and output along three separate modules.
- The **interface design** describes internal and external program interfaces, as well as the design of human interface. Internal and external interface designs are based on the information obtained from the analysis model.
- The **algorithm design** describes in pseudo code, simile to Java, the most important algorithm implementation.
- The **requirement traceability** shows which component guarantee each requirement.

2. Architectural design

2.1 Overview

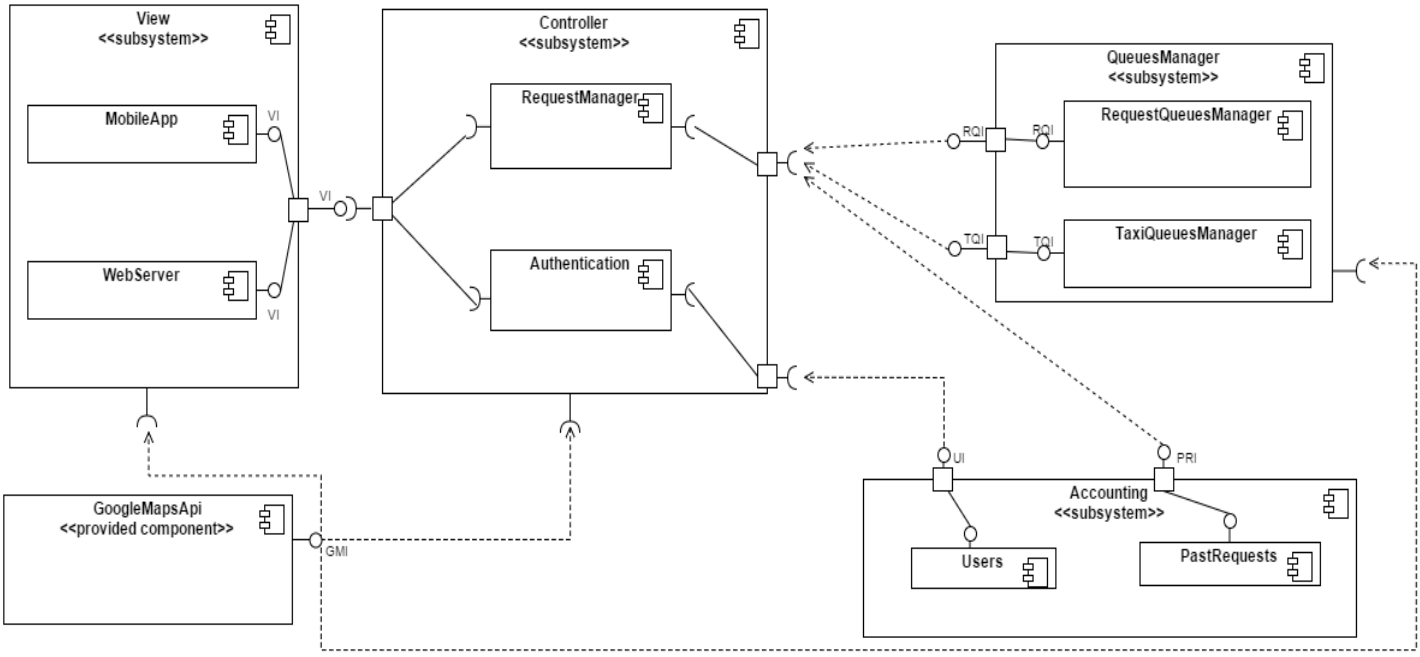
For our application we chose the MVC Pattern because we need a way to handle the triggering of synchronization between screen state and session state inside the system directly. MVC does it by making updates on the model and then relying of the observer relationship to update the views that are observing that model.
We choose the 3-Tier style because of the need to separate the Client from the Server and to store and protect sensible data in a DBMS and for the high system performances that we want to reach.
We will provide an objected oriented design, which is based on entities and on their interaction, consistent with the past object oriented analysis of the RASD.
We use also a simplified event based architecture that does not make use of the event dispatcher.

2.2 High level components and their interaction

- **WebServer** is the component in which creates pages for the browser interface and receives requests and data submitted form the browser.
- **MobileApplication** shows the graphic interface to users who access the system through a mobile device. This component notifies requests and data inserted by the users. It also detects the GPS position of taxi drivers and send it to RequestManager that call the QueueManager if a taxi moves from a zone to another.
- **Authentication** is the component that is called by the WebServer or the MobileApp when username and password are provided by the client, checks the user's category through the Accounting, if all the field are correct, gives the new page and the authorization to the client.
- **RequestsManager** creates instances of the Request class (that generalize GuestRequest, RegisteredRequest and Reservation) when MobileApp or WebServer notify the related event and the RequestQueueManager to enqueue the Request. It also makes the association between Requests and Taxis, thanks to the connection with the TaxiQueuesManager.
  It calls with PagesCreator and MobileApp to notify users about the association.
- **TaxiQueuesManager** creates a queue for each zone, in which all the taxis that are in that zone are enqueues. If a taxi exceeds the limit of the zone, his id is moved in the new zone and eliminated from the past one.
- **RequestsQueuesManager** creates a queue for each ZONE, in which all the requests for that zone are enqueues.
- **Accounting** stores in the database all the taxis and users personal data and looks for them when requested.
- **PastRequests** stores in the database all the past requests of the registered costumers and looks for them when requested.

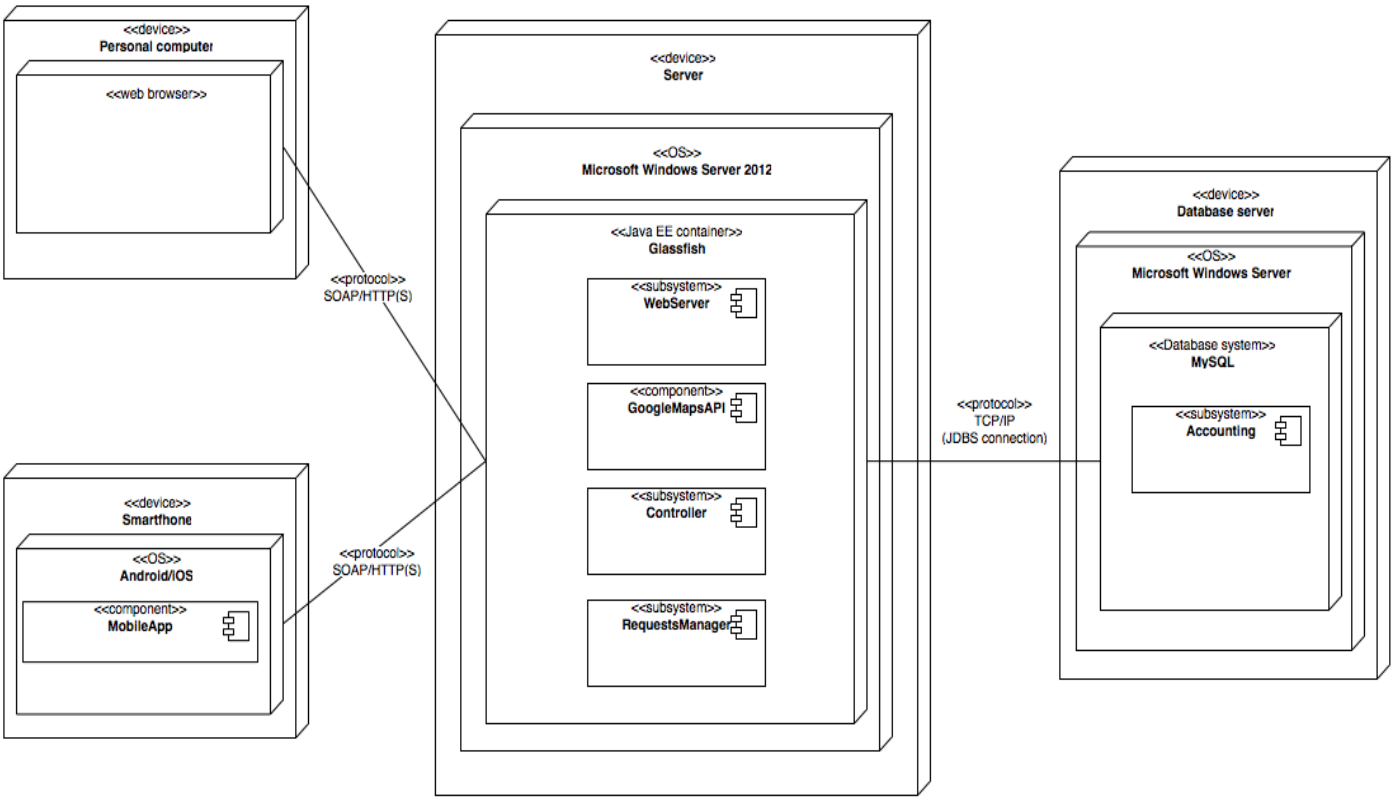
2.3 Component view

The following diagram shows how components interact. We grouped components into four subsystems according to their role in the MVC patter.
QueuesManager and Accounting can be considered as two different kind of Model, that differ for the fact that the data modelled by Accounting are stored permanently while the ones modelled by QueuesManager are temporary data.



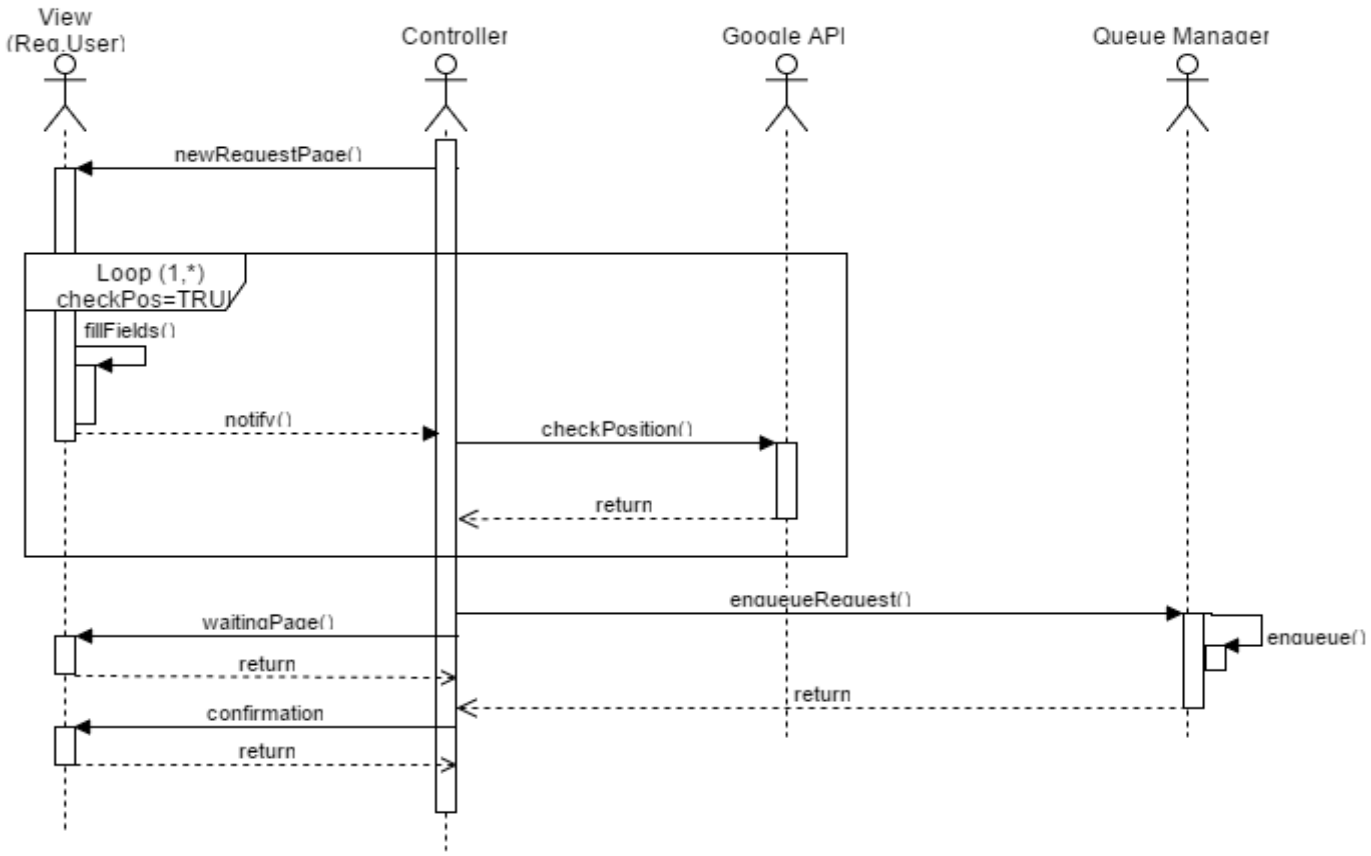
2.4 Deployment view

The following diagram shows how components are deployed on physical devices.
This mapping is developed basing of the MVC pattern and the 3-tier architectural style.

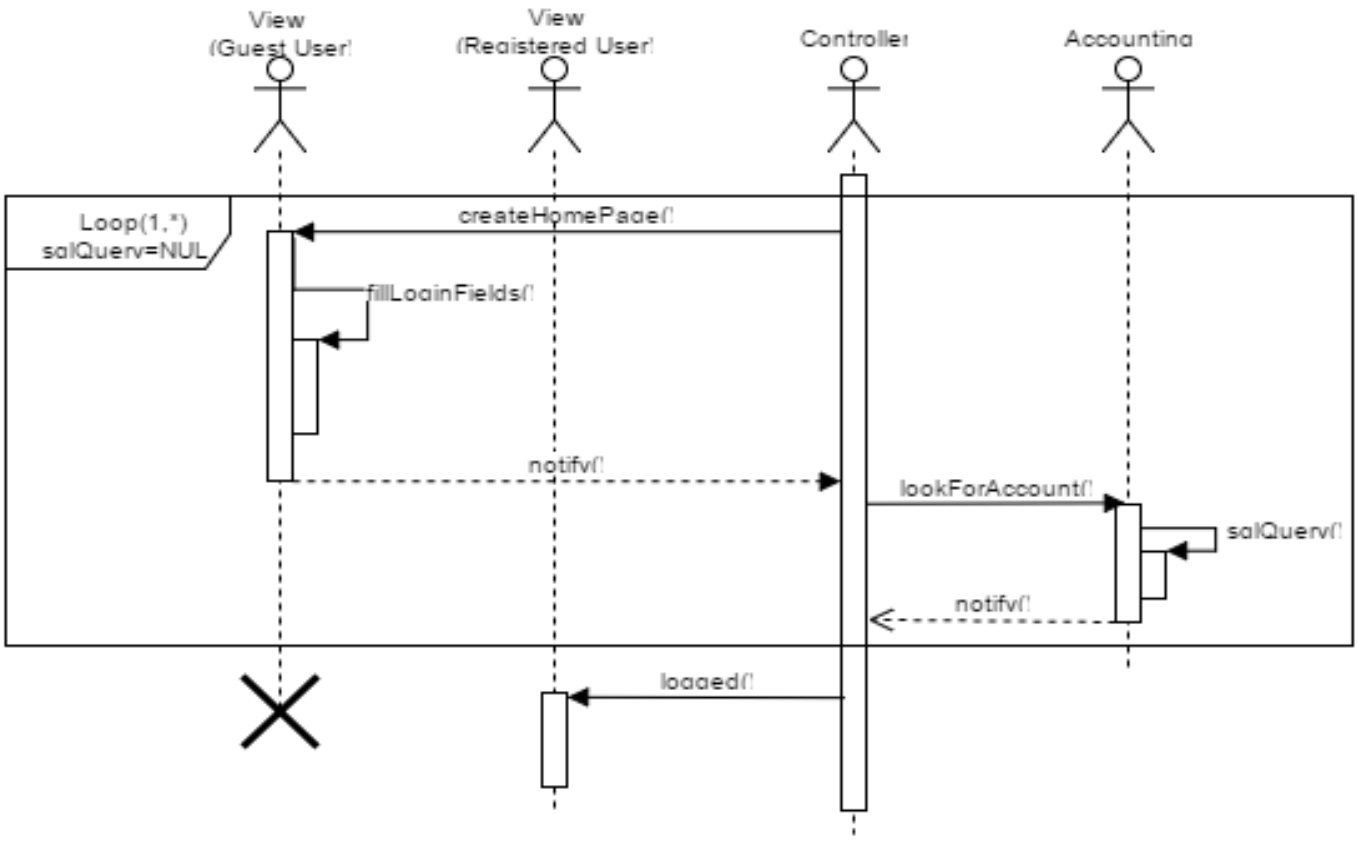


2.5 Runtime view

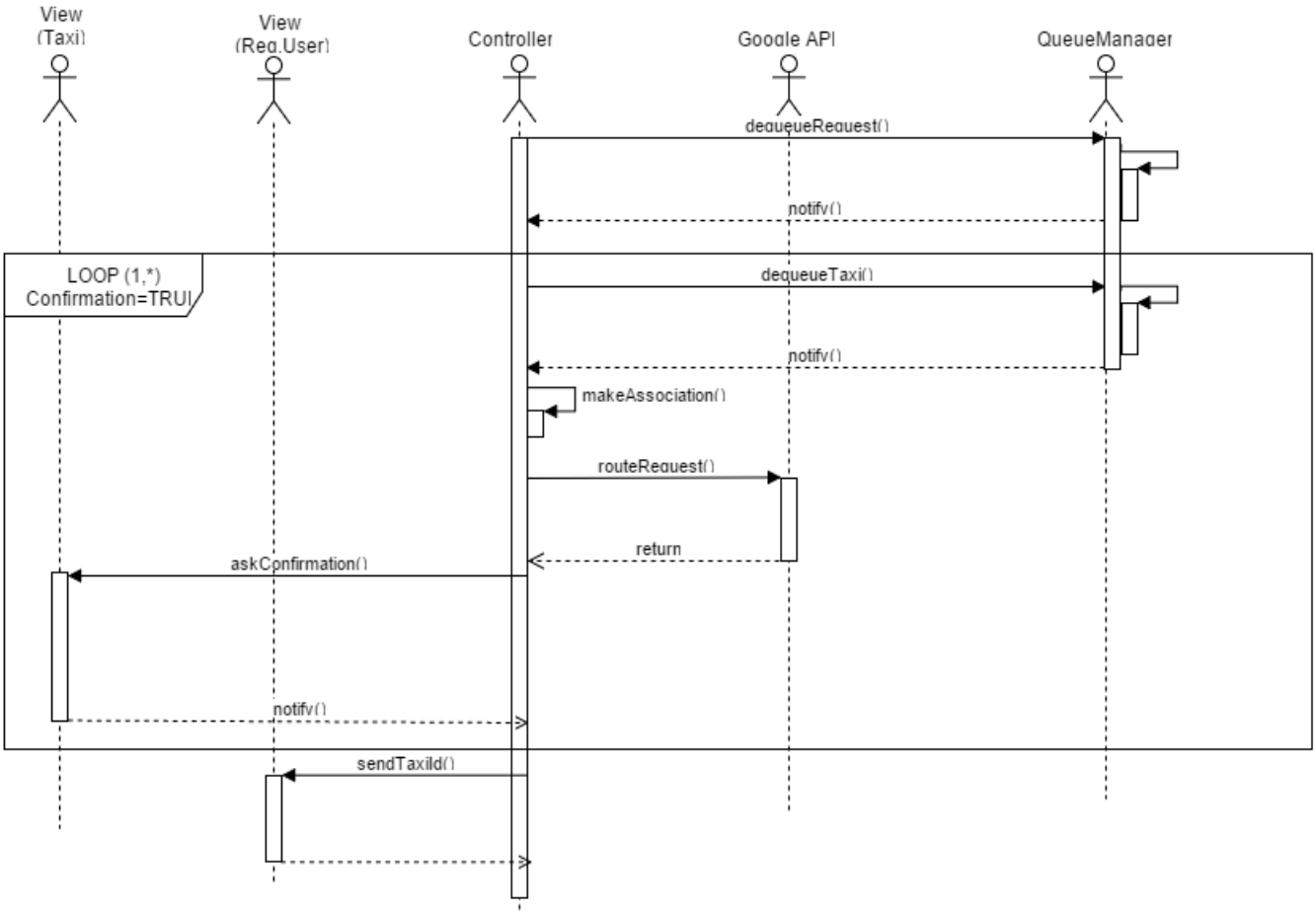
2.5.1 Add Request Sequence Diagram



2.5.2 Login Sequence Diagram



2.5.1 Request Association Sequence Diagram



2.6 Component interfaces

In this section we will describe all the interfaces showed in the Component view.
We will provide both a high-level description and a list of possible java-like methods (to be considered as a generic suggestion for implementation)

2.6.1 RequestsQueueManagerInterface

2.6.1.1 High level description

RequestsQueueManagerInterface (called RQI in component diagram) is used by RequestsManager. It provides functionalities that have the following scope:

- enqueue a request: it checks if the request is a normal request or a reservation and add it in the back of the queue in the first case or between the last reservation and the first request in the second one.
- dequeue the request: it extract and delete the frontmost element of the queue and notify it to the RequestManager

2.6.1.2 Possible methods

- void enqueueRequest(Request req)
- void dequeueRequest()

2.6.2 TaxiQueueManagerInterface

2.6.2.1 High level description

TaxiQueueManagerInterface (TQI in component diagram) is used by RequestsManager.
It provides functionalities that have the following scope:

- enqueue a taxi: it inserts the taxi in the queue as backmost element.
- insert a taxi as first item of the queue: it inserts the taxi in the queue as frontmost element; it is used to reinsert the taxi in the queue when the driver communicates that he didn't find the costumer at the meeting address.
- dequeue the taxi: it extract and delete the frontmost element of the queue and notify it to the RequestManager.

2.6.2.2 Possible methods

- void enqueueTaxi(TaxiDriver t)
- void insertTaxiAsFirstItem(TaxiDriver t)
- void dequeueTaxi()

2.6.2 UserAccountingInterface

2.6.2.1 High level description

UserAccountingInterface (called UI in component diagram) is used by Authentication. It provides functionalities that have the following scope:

- look for an account in the DB: it looks for account having a specific username in database by asking the DBMS to make a SQL query.
- add an account to the DB.

2.6.2.2 Possible methods

- void lookForAccount(String username)
- void addAccount(String username, String password, String firstName, String lastName, String email)

2.6.2 PastRequestsInterface

2.6.2.1 High level description

PastRequestsInterface (called PRI in component diagram) is used by RequestsManager. It provides functionalities that have the following scope:

- store a normal request made by a registered costumer
- store a reservation
- look for a past request in the DB: it looks for the list of requests made by a registered costumer in database by asking the DBMS to make a SQL query.

2.6.2.2 Possible methods

- void storeRequest(RegisteredRequest r)
- void storeReservation(RegisteredReservation)
- void lookForListOfRequests(String username)

2.6.2 ViewInterface

2.6.2.1 High level description

ViewInterface (called VI in component diagram) is used by RequestsManager and Authentication and implemented by MobileApp and WebServer.
It provides functionalities that have the following scope:

- creation of pages for the GUI

2.6.2.2 Some possible methods

- void createHome()
- void createPersonalAreaHome(Class typeOfUser)
- void createNewRequestPage()
- void createNewReservationPage()
- void createWaitingPage()
- void createEnqueuedRequestPage()
- void askConfirmation(String taxiDriverUsername, Position pos)
- void notifyTaxiId(String id)
- void createSuccessfulPage ()

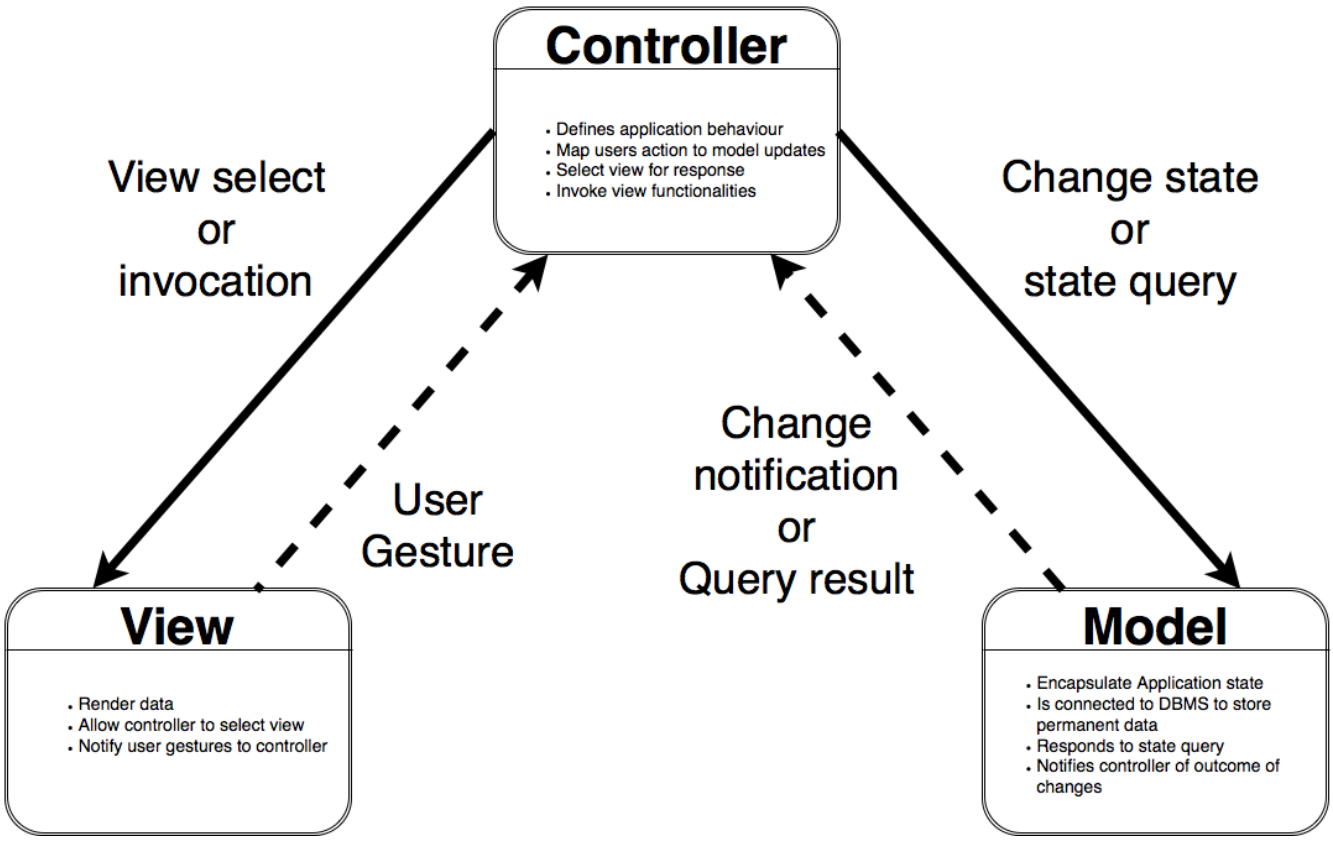
2.6 Selected architectural styles and patterns

We selected the **MVC** architectural pattern. It implies he division of the whole application in three subsystems each one with a specific scope:

- **Model**: it models the data used by the system and the state of the application. We split it into two parts, basing on the type of data:
  - **Accounting:** it stores data permanently and so it makes use of the database;
  - **Requests Queue Manager:** it stores data temporary.

- **Controller:** it is the part of the application that contains the application logic.
- **View:** it provides the functionalities that allows the users to interact with the application

The interaction between these component is managed with a **simplified version of the event-based** pattern, that does not use the event dispatcher as the classical one. The basic idea is that the components of the controller invokes functionalities provided by the view and the model. After a call, controller waits for the announcement of one of the events the invoked functionality can leads to. For instance, controller can invoke the functionality to create the home page, provided by the view; the view creates the home page and, as soon as the user submit some data, it announces the happening of the event; the controller, who had registered an interest for that event, is notified and can behave consequently.



To deploy these component on physical machines, we used the **3-tier** style.
We chose this style because it provides:

- **Performance**: network utilization is minimized and the load is reduced on the Application and Data tiers.
- **Flexibility, maintainability** and **scalability**: since presentation, application and data layers are deployed on different tiers; it is quite easy to make changes to a layer without affecting the others.
- **Security**: because data are stored in the bottommost tier.
- **Manageability**: the project implementation can be divided into simpler projects, that could be assigned to different programmers or programming teams (divide et impera principle).
- **Reusability**: it is easy to share and reuse the components and services.

- **Decoupling**: dividing presentation, application and data layers according to MVC pattern reduces the possibility of interdependency between classes.

The approach to architectural design we provide is an **object-oriented** approach, in keeping with Object Oriented Analysis described in RASD (especially in chapter 3.4) with architectural details. The advantages of this pattern are:

- Making problems simpler by dividing them in sub-problems and approaching them independently (Divide et impera principle).
- Having a high level of abstraction.
- It is easy to reuse components and classes generated by this approach.

The platform that, according to us, better fits our decisions is Java EE ver. 7. The benefits of using this platform are:

- The system will be maintainable in terms of updating the application and the website.
- It is heavily object oriented (according to the chosen style).
- Good grade of performance compared to other possible solution.
- Many API are provided.
- It is good for large-scale application (our application is not much large, but neither so small).

3. Algorithm design

1.1 void enqueueRequest (Request req)

```
{
requests = list of key req.Position.getZone() in hashmap<Zone, List<Request>>;
if (req is not an instance of Reservation)
        add req as last item of requests;
else{
        int i;
        while(i-th item of the list is an instance of Reservation)
                i++;
        add req as i-th item of requests;
    }
}
```

1.2 void dequeueRequest (Zone z)

```
{
        requests = list of key z in hashmap<Zone, List<Request>>;
        req = first item of requests;
        delete the first item of requests;
        notify observers that the class has changed and that req has been dequeued;
}
```

1.3 void dispatchTaxis ()

```
{
        N = number of zones ;
        r, t = arrays of N integer;
        lhreq[] = number of request handled during the the last hour for each zone

        for (i:0->N){
                r[i] = lenght of the list that has zone of id i as key in hasmap<Zone, List<Request>>;
                t[i] = lenght of the list that has zone of id i as key in hasmap<Zone,
List<TaxiDriver>>;
        }

        tot_r = sum of r[*];
        tot_t = sum of t[*];

        for(i:0->N)
                if(t[i]<2 && tot_t >2*N)
                        adj = get the adjacents zones of i-th zone
                        possibleTaxis = set of taxies in the list of the ones having adj[*] as key in the
hashmap
                        taxi = random taxi in possibleTaxis
                        notify(taxi, i-th zone)
```

```
            if(t[i]<lhreq[i] && tot_t > sum of (lhreq[*]))
                adj = get the adjacents zones of i-th zone
                possibleTaxis = set of taxies in the list of the ones having adj[*] as key in the
hashmap
                taxis = set of lhreq[i]-t[i] random taxis in possibleTaxis
                for(taxi in taxis)
                    notify(taxi, i-th zone)
}
```

4. User interface design

See the RASD Mockup.

5. Requirement Traceability

Functional Requirement	Component	Comment
[R1.1] [R1.2][R1.4]	Authentication + Accounting	Allow Taxi to have a personal reserved area
[R1.3][R1.5][R1.6]	MobileApp + Authentication + Accounting	Allow taxi to Login through the Mobile application and see
[R1.5][R1.6][R1.8]	MobileApp + RequestManager	Allow taxi to manage the request that the system assigns
[R1.7]	MobileApp + GoogleMapsAPI	Allow taxi to see his position and the shortest way to the destination
[R2.1] [R2.1] [R2.1]	MobileApp/WebServer + RequestManager	Allow an unregistered User can request a taxi in a specific position and receives the Taxi ID with an SMS
[R3.1] [R3.2]	MobileApp/WebServer + Authentication + Accounting	Allow a User to Register and login with his credentials
[R3.3] [R3.4] [R3.5] [R3.6] [R3.7]	MobileApp/WebServer + RequestManager	Allow a Registered User to request or reserve a taxi and will receive the Taxi ID
[R3.8] [R3.9]	MobileApp/WebServer + Authentication + PastRequest + Accounting	Allow a Registered User to see, modify and delete his request (before 2 hours)
[R4.1] to [R4.12]	QueueManager + GoogleMapsAPI	Allow TaxiQueue and RequestQueue to be ordered and managed fairly as we see in the Functional Requirement
[R5.1] [R4.6] [R4.	MobileApp + QueueManager + GoogleMapsAPI	Allow TaxiQueue to be managed through the sending a notification to a taxi from one zone to another