SOFTWARE ENGINEERING 2: TRAVLENDAR+

DESIGN DOCUMENT

Agostini Andrea, Ciampiconi Lorenzo, Es-skidri Rachid

NOVEMBER 20, 2017

CONTENTS

INTRODUCTION

PURPOSE

This is the Design Document for Travlendar plus system. Its aim is to provide a functional description of the main architectural components, their interfaces and their interactions, together with the algorithms to implement and the User Interface design. Using UML standards, this document will show the structure of the system and the relationships between the modules. It describes also information about the Test Plan.

This document is written for project managers, developers, testers and Quality Assurance. It can be used for a structural overview to help maintenance and further development.

SCOPE

Analysis of the world phenomena

Nowadays, time management is one of the most important things in the context of today's society, especially in big cities, where the variety of means of transport is so great to allow a better optimization of a person's events organization. Travlendar+ borns as a digital calendar that allows the user not only to display his events, but also to provide the user the best way to reach the events in the best possible way, according to the criteria chosen by him.

Analysis of the shared phenomena

There are two different kinds of shared phenomena, the first one includes phenomena that are controlled by the world and observed by the machine, such as the GPS position of the user, the traffic, the weather and for example, something quite abstract such as the time schedule of a bus or a tube. The second one contains all those phenomena controlled by the machine and observed by the world (in according to the domain) such as : the user follows Travlendar+ indications, the user inserts the true duration time events, the user uses the system inside the correct geographic area etc.

DEFINITIONS

- **User**: any individual subscribed to the service.
- **Visitor**: an individual not subscribed to the service.
- **Event**: an appointment that could be registered in the calendar.
- **Free Time Interval**: with this term we refer to the interval of time that user could registered in the calendar to indicate where the flexible break must be spent.
- **Overlapping events**: when two events A and B overlaps it means that they share a time interval. More formally, when A starts before the start of B and A ends after the end of B, A overlaps with B.
- **System**: the whole software system to be developed, comprehensive of all its parts and modules.

ACRONYMS

- **RASD**: Requirements Analysis and Specification Document (this document).
- **API**: Application Programming Interface.
- **UI**: User Interface.
- **DB**: Data Base.
- **DBMS**: Data Base Management System.
- **DD**: Design Document.
- **MVC**: Model View Controller.

ABBREVIATIONS

- **[Gn]**: nth goal.

REFERENCE DOCUMENTS

- This document refers to the specification document: Mandatory Project Assignments.pdf - Assignments AA 2017-2018

- This document refers to the RASD – the previous deliverable.

.

DOCUMENT STRUCTURE

This document is structured in six parts:

Chapter 1: *Introduction*. It provides an overall description of the system scope and purpose, together with some information on this document.

Chapter 2: *Architectural Design*. This section shows the main components of the systems with their sub-components and their relationships, along with their static and dynamic design. This section will also focus on design choices, styles, patterns and paradigms.

Chapter 3: *Algorithm Design*. This section will present and discuss in detail the algorithms designed for the system functionalities, independently from their concrete implementation.

Chapter 4: *User Interface Design.* This section shows how the user interface will look like and behave, by means of concept graphics and UX modeling.

Chapter 5: *Requirements Traceability.* This section shows how the requirements in the RASD are satisfied by the design choices of the DD.

Chapter 6: *Implementation, Integration and Test plan.* In this section we identify the order in wich our plan to implement the subcomponents of the system and the order in which we plan to integrate such subcomponents and test the integration.

ARCHITECTURAL DESIGN

OVERVIEW

High level components and their interaction:

Database :
the data layer is responsible for the data storage and retrieval.
It does not implement any application logic. This layer must guarantee ACID properties.

Application server :
this layer contains all the application logic of the system.
All the policies, the algorithms and the computation are performed
here. This layer offers a service-oriented interfaces.

Client :
This layer consists in an Android Mobile application. It's both presentation layer and logic/client layer, it communicates directly with the application server and it represents the user's interface.

These high-level components are structured into three layers, shown in *figure 1*. This choice give us the possibility to compute all the business logic in the Application Server layer and make the client application thin and efficient so it could provide a comfortable user experience. Furthermore this design allows to extend the system, inserting a Web Server layer to consult Travlendar in every device that has a generic browser. In this first release we focus on the Android Mobile client in order to implement the system in the expected times.

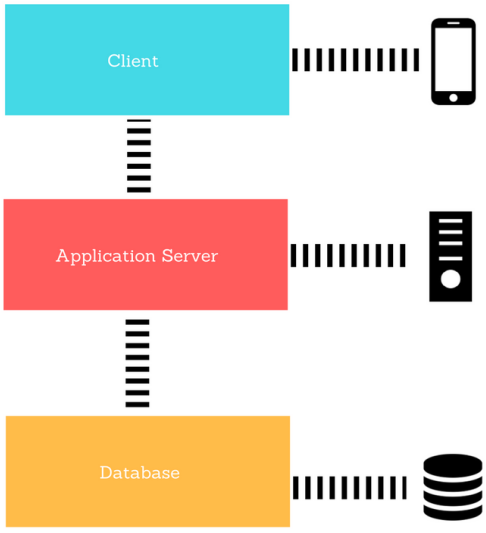


Figure 1, Layers of the system.

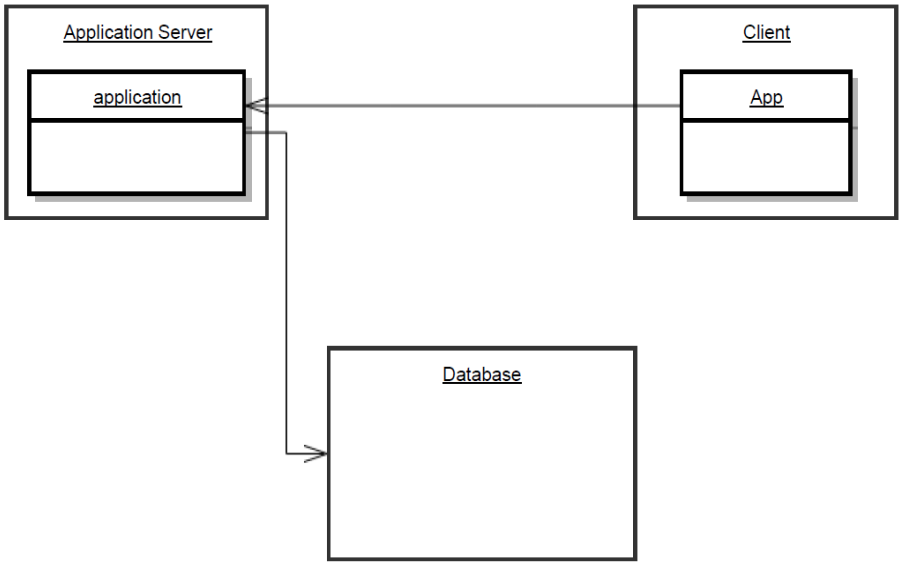


Figure 2, High level components of the system.

COMPONENT VIEW

The following diagrams describe the main component of the system and the interfaces through which they interact. The *client side* is identified with the mobile application. Is a very thin client that allows the user to interact with the application server which contains the main part of the business logic of the system.

Server Side Database

The database tier runs in an external database service that allow us to store data more safely than in an internal Db. We use InnoDB as the database engine: the DBMS has to support transactions and ensure ACID properties. Access to the data must be granted only to authorized users possessing the right credentials. Every software component that needs to access the DBMS.
It' important to remark that the user entity in the following schema is useful for the business tier(for example, we can store here the custom user policies). The user credentials are handled by an external backend service, for high security reasons.

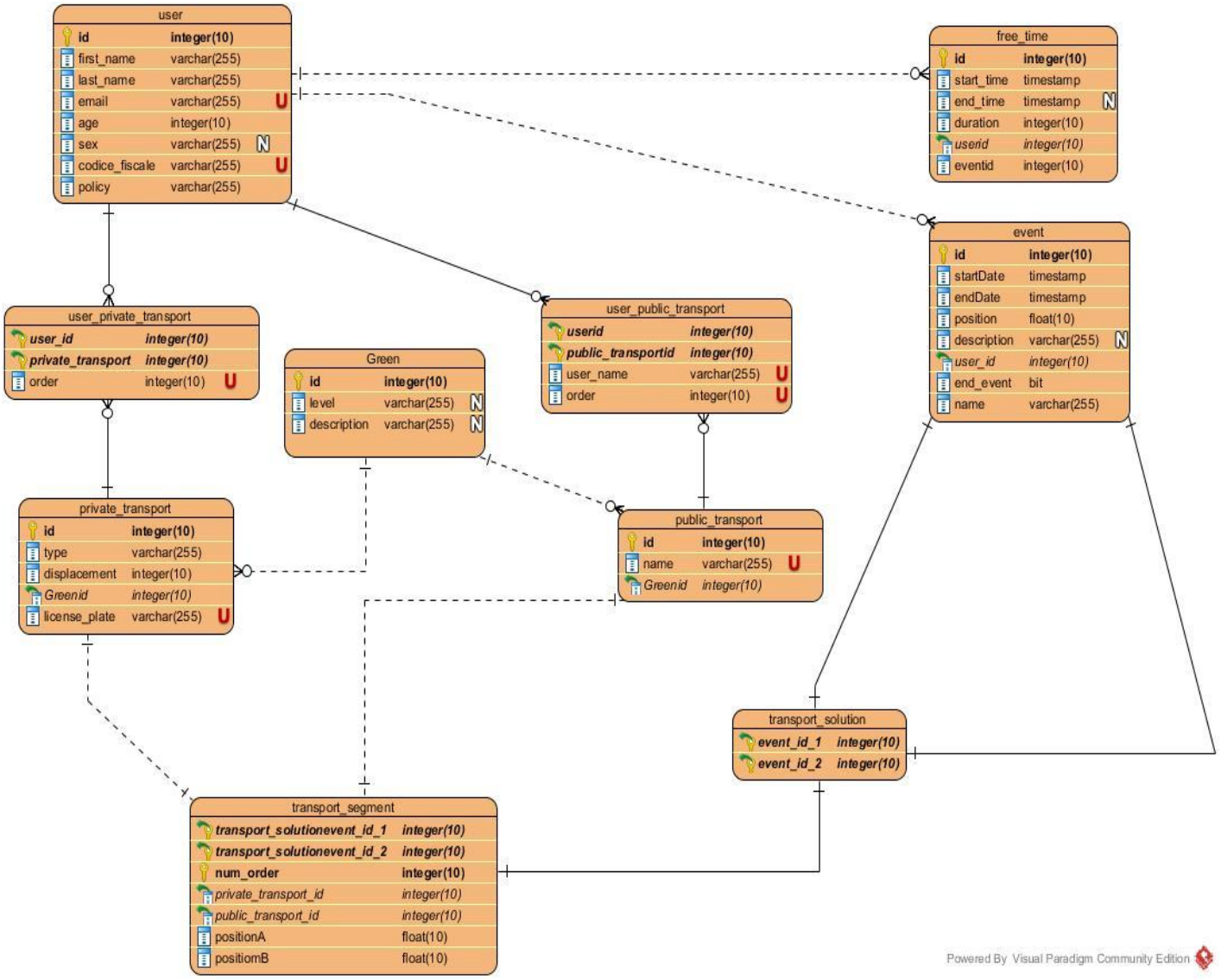


Figure 3, The Entity-Relationship diagram of the database schema.

Client Side Database

The Client Db schema is composed by a synchronized light copy of the Server side Db. We chose SQLite DBMS because is a best practice implement that on an Android application. The interaction with that is handled by an Android ORM (Object Relational Mapping), in particular we use greeDAO ORM service, like in the *Figure 4* below.

Figure 4, greenDao module connection

Controller projection

Controller subsystem is composed by three main components: *DataManager, AccountManager* and *RouteManager*. The first one concerns all the business logic about the management of the data that will be putted/getted into/from the database. The RouteManager provides the logic about find the best solutions to link the events of the use with se Google Maps Api and Sharing services Api support . The accountManager has the goal to identify the user with the support of Firebase Api.

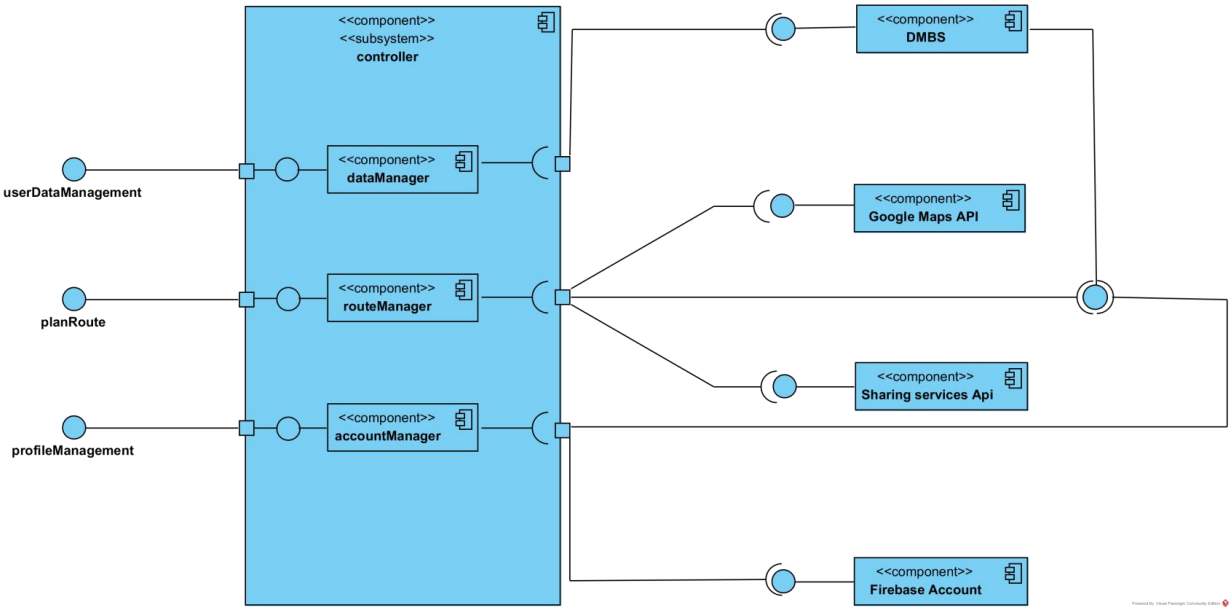


Figure 5, Controller Projection Schema

DEPLOYMENT VIEW

The system architecture is divided in three tiers and it is based.

- The client tier is composed by the mobile application that communicate with the business tier
- The business tier is based on Spring framework because it represent the best practice for these types of system. The details of this framework are discussed in the next page.
- The Database tier is mainly composed by the External Database Server. The communication with the business tier is performed via JDBC connector.

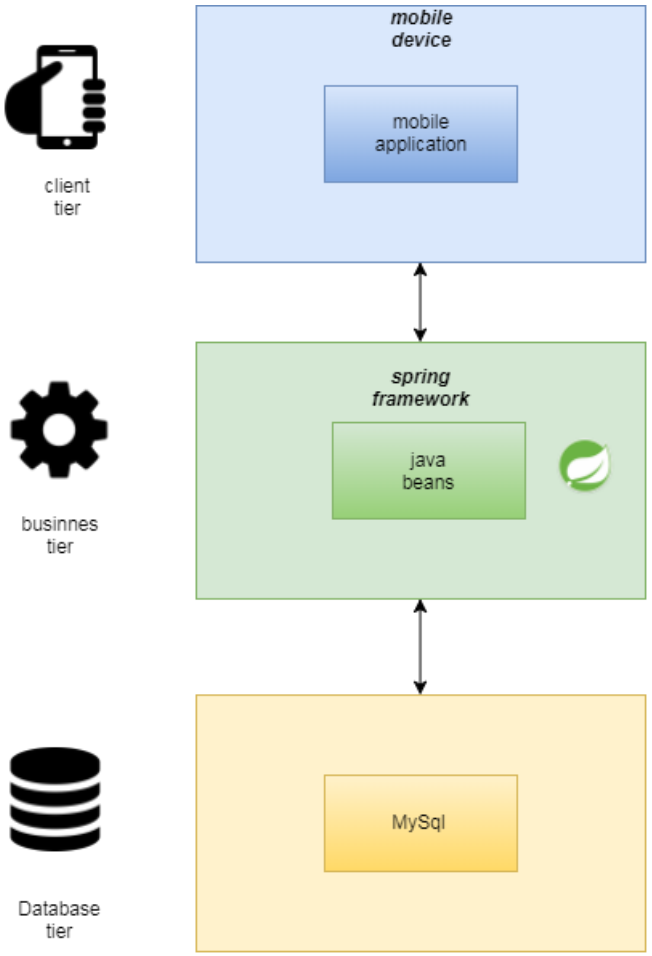


Figure 6, The detailed description of tiers

Why Spring Framework ?

We chose to use Spring because is a powerful framework based on Java Enterprise Edition, that simply the devolpment of the server side providing specific functionalities. It's a framework based on MVC paradigm (Architectural Pattern used in this system as specified in the dedicated section).
Spring comes with some of the existing technologies like ORM framework, logging framework, J2EE and JDK Timers and more, hence we don't need to integrate explicitly those technologies.
Spring can eliminate the creation of the singleton classes. Spring framework is both complete and modular, because it has a layered architecture. This framework includes support for managing business objects and exposing their services to the presentation tier components, so that the web and desktop applications can access the same objects. It has taken the best practice that have been proven over the years in several applications and formalized as design patterns. Spring application can be used for the development of different kind of applications, like standalone applications, standalone GUI applications, Web applications and applets as well. *Figure 7* shows clearly how the Spring modules are integrated with our designed system.

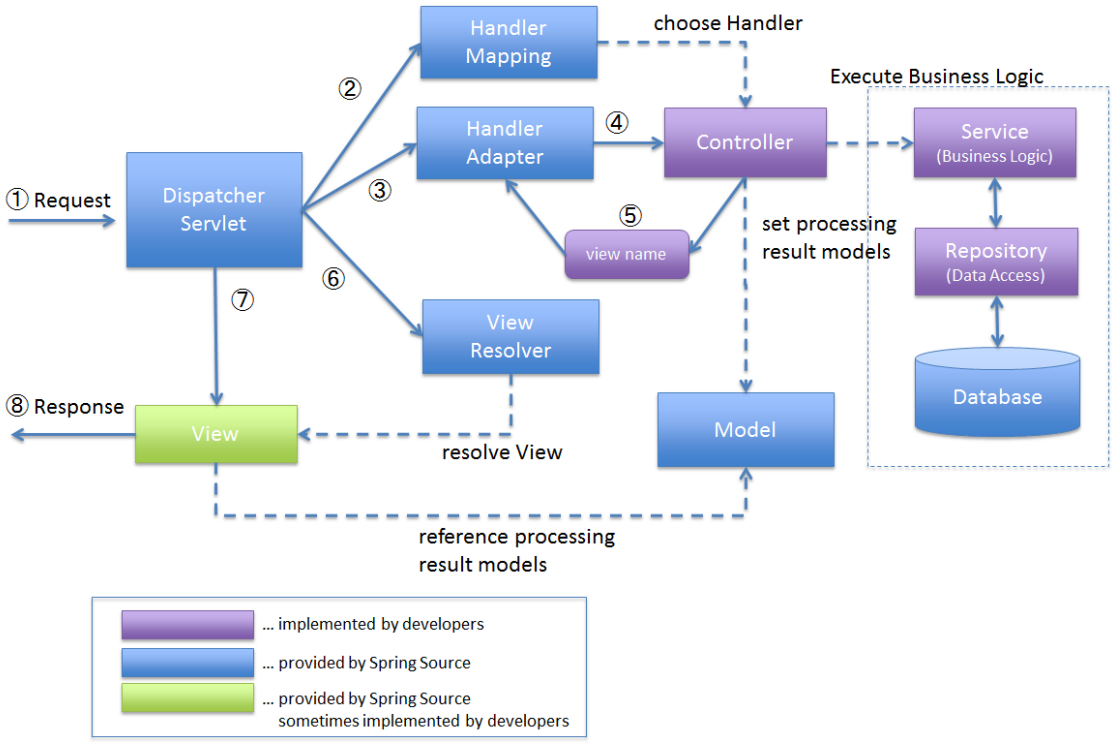


Figure 7: Spring Framework Modules Integration

RUNTIME VIEW

In this section we will describe the dynamic behavior of the system. In particular, it will be shown how the components previously defined interact one with another, using sequence diagrams. Beware that this is still a high-level description of the actual interactions that are going to take place, so functions and their names may be added or modified during the development process.

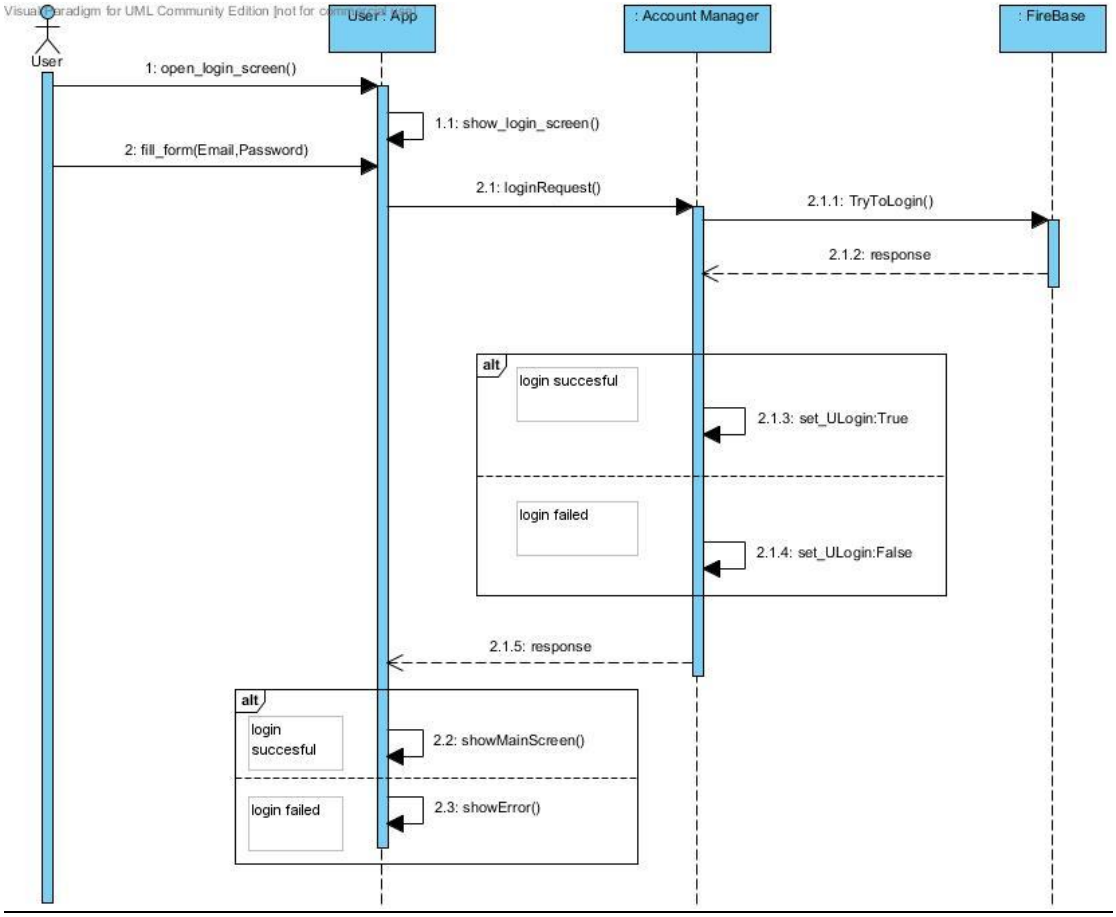


Figure 8, User Login

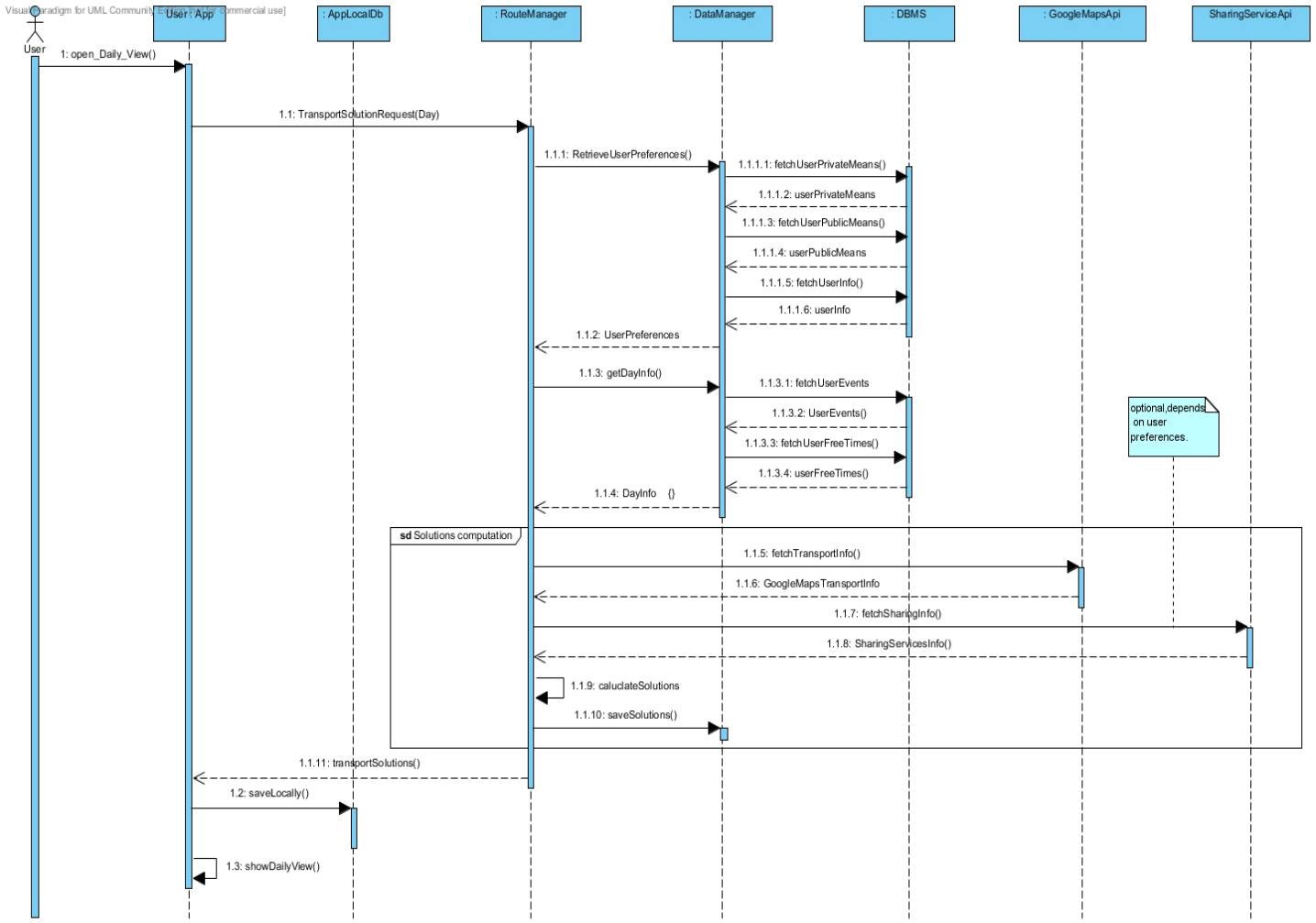


Figure 9, Transport Solution Calculation

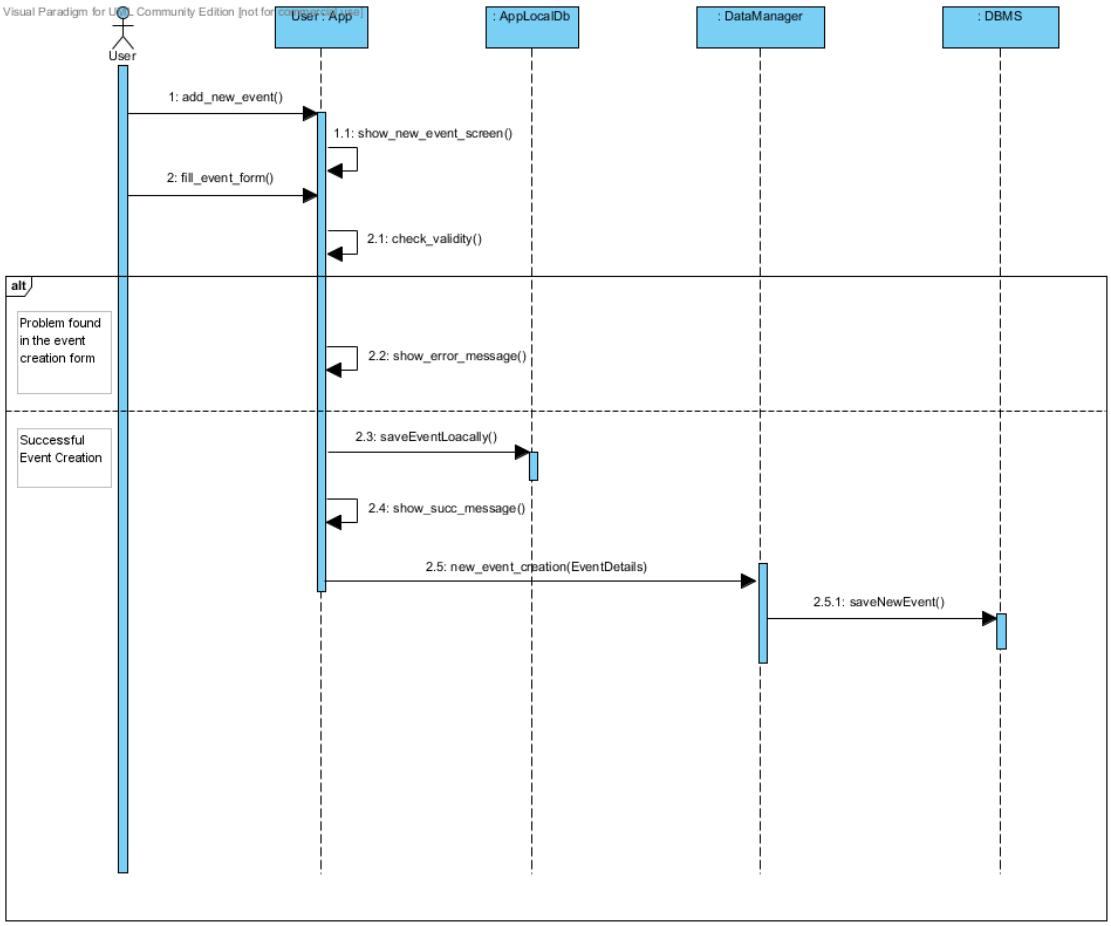


Figure 10, Event Creation

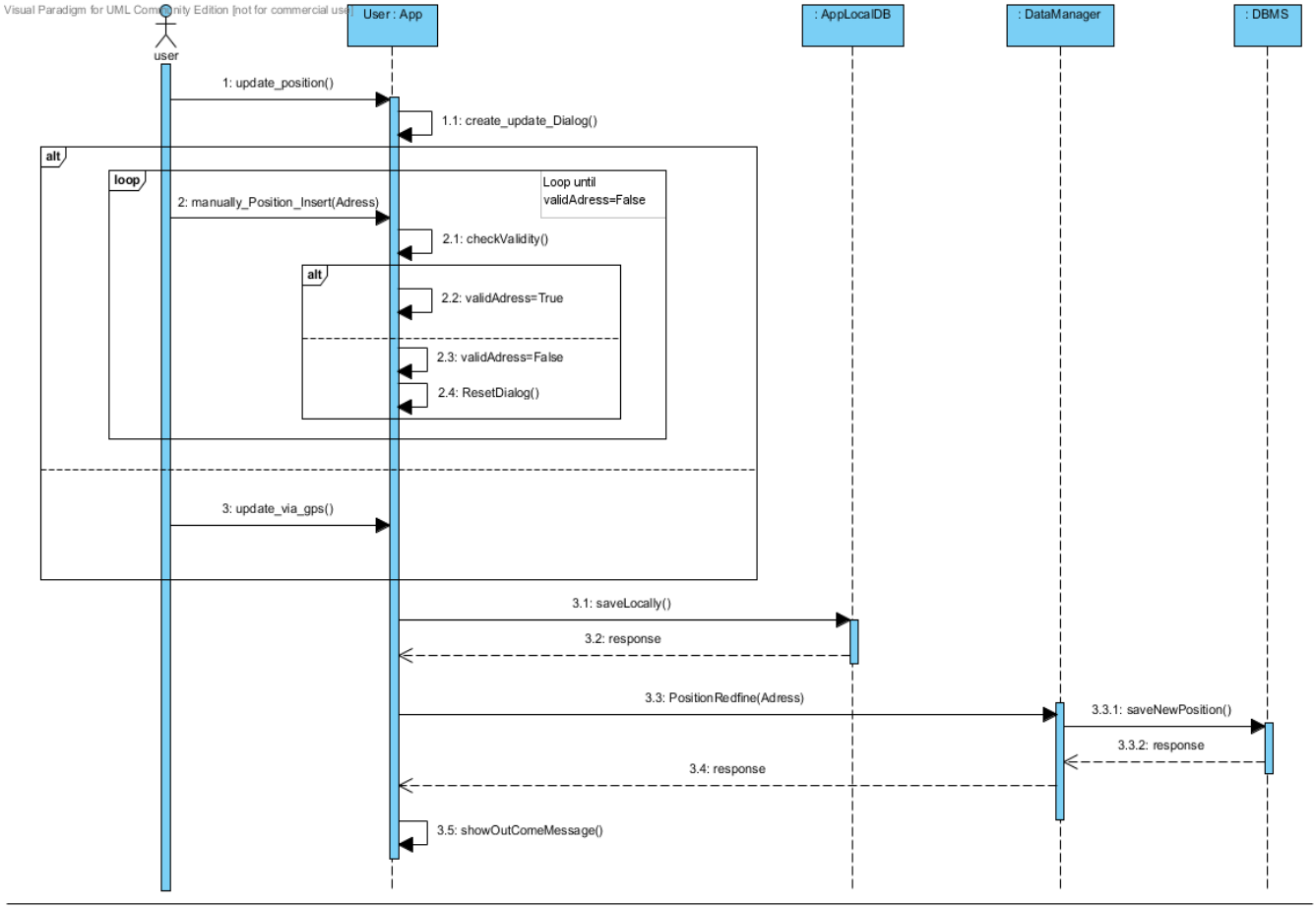


Figure 11, Update Position

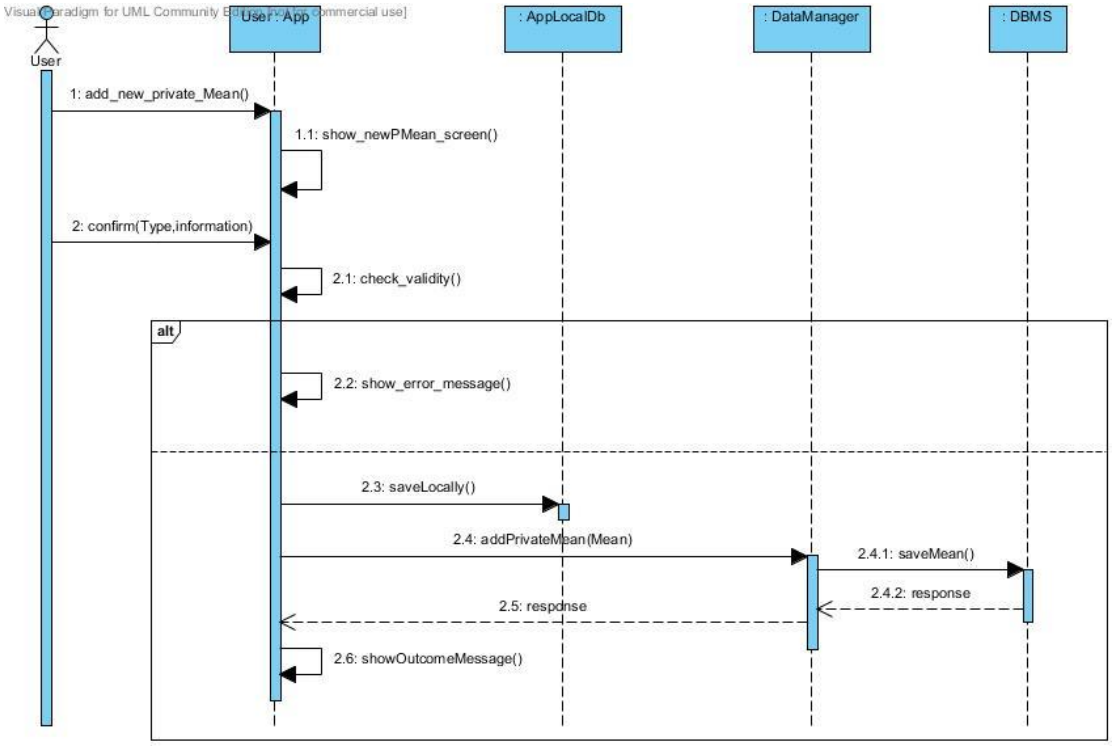


Figure 12, Add new private mean

COMPONENT INTERFACES

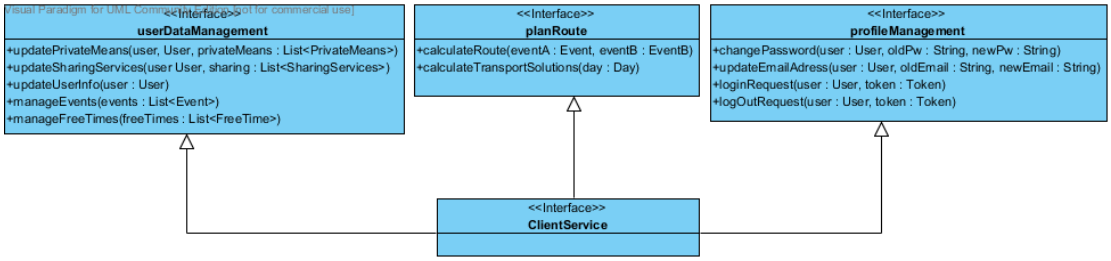


Figure 13, ClientService Interfaces

SELECTED ARCHITECTURAL STYLES AND PATTERNS

Architectural Patterns

MVC *(Model-View-Controller)* pattern has been widely in our application. There are a lot of benefits on this choice, first of all the separation of the three components allows the re-use of the business logic across application and multiple user interfaces can be developed without concerning the codebase. Another crucial point is that MVC facilitates the developing process, for example The developers of UI can focus exclusively on the UI screens without bogged down with business logic and the developers of Model / business can focus exclusively on the business logic implementations, modifications without concerning the look and feel of UI.

Our Application server will use the SPRING framework, which is an MVC framework.

Client Server Model

The application is strongly based on a Client-Server communication model. The clients being the mobile application in the first release. The clients are thin, thus to let the application run with low consumption of resources. All the computational processes relative at the transport solutions is powered server side. By the way the application is not only a UI layer, for example it has own local database, this is very important to maintain the persistence of information in limit conditions (for example an user is able to insert an event in case of broken internet connection, then the mobile application will send the new information on server when it will possible). This approach is very important also because it improvers the maintainability of the system.

Structural Design Patterns

- *Façade Pattern* hides the complexities of the system and provides an interface to the client using which the client can access the system. (Used in some components of the component diagrams)

- *Bridge Pattern* decouples an abstraction from its implementation so that the two can vary independently. This pattern decouples implementation class and abstract class by providing a bridge structure between them. (Used in the interfaces of the component diagrams)

Creational Design Patterns

- *Factory Pattern* allows to create object without exposing the creation logic to the client and refer to newly created object using a common interface. It is particularly useful if applied in combination with the MVC pattern. We use this pattern in the implementation in several cases, for example with the JPA Entities like in the code snippet below.

```
@Entity
@Table(name = "Users")
public class User {

    private static final String PERSISTENCE_UNIT_NAME = "UserDatabase";
    private static final EntityManagerFactory factory = Persistence.
            createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
```

Behavioral design pattern

- *Strategy Pattern* allows the changing of a class behavior or its algorithm at run time. This pattern, if applied correctly, guarantees a high level of polymorphism.

ALGORITHM DESIGN

In this paragraph we will discuss the algorithm design with focus on the interpretation of the calendar on the events inserted by the user and the calculation of the desired transport solutions between two subsequent event by the user preferences. To achieve our goal the algorithm that we will design must be divided on two main algorithms:

The *first algorithm* will receive the calendar, with the events state of a particular instant, it will recognize the first event and the possible connection between two events (it will recognize a connection between two events in case there's no events between them, we will focus later on more particular case of overlapping and the end event case), after a connection is recognized the algorithm will call the second one passing to it the two events of which the transport solutions must be calculated. This operation will be repeated until all the events and their connection have been handled.

The *second algorithm*, that is in charge to calculate the transport solutions, will receive two events and will first of all consider the user preferences and means of transport that he has registered (such as private means, subscriptions to public services or car/bike sharing services). It will identify a solution from three possible policies: the fastest, the cheapest and the most ecological. For the solution the algorithm will identify a mean of transport by the user preferences (the policy is included in the preferences as well) and a starting and location for the usage of this mean of transport, then it will call the Google external API passing to it the mean of transport and the desired ending location. After this the algorithm will have a portion of the trip covered by the desired mean of transport, until there's no portion uncovered the algorithm will call itself recursively.

First Algorithm

The first algorithm is designed to have a correct behavior on different condition:

- The transport solution for all the event must be calculated:
    - The user changes his preferences;
    - The user register new means of transport;
    - Extraordinary situation: such as loss of data on transport solution;
- An event has been inserted or modified;
- An event has been deleted;

Formalization of the First Algorithm

Input: Ordered List of Events

Description

The event inserted in the calendar can be modeled in an oriented graph as vertex and the necessity of transport solution between events are edges.
So we have a *Graph* as data structure:
V as set of *Vertex*;
E as set of *Edges*;
The object "Event" bundled in a vertex have three attributes: Location, start time, ending time and a Boolean for "ending event".
We assume that the calendar is passed to the algorithm as an ordered list of events (by the starting time) and we use an ordered list as REP for the set V.
Set E is defined as: $E \subseteq (V \times V)$
So the object bundled in an edge is a simple pair of events.
We use an ordered list as well as REP for the set E, the order is determined by the event from which the edge is outgoing. If two edges share the same starting vertex the order is determined by the events in which the edges is ingoing. The REP must not contain duplicate elements.

First of all the algorithm fill the set V with all the events received as argument, then insert in the list after the events marked as "end event" a new vertex that have as location the default location.
The algorithm scans the list and add to E an edge between the subsequent events by the ordered lists skipping the events marked as "ending". Then from each vertex v1 the algorithm scans the edges that is outgoing from it and, (condition c1) if the starting time of the vertex v2 in which the edge is ingoing is before the ending time of v1, for each edge e1 that is outgoing from v2 and ingoing in v3 an edge e2 outgoing from v1 and ingoing in v3 is added. Until there's no more edges that satisfies condition c1 the operation is repeated. This step is important to grant connection between events also in condition of overlapping.
In the final step the second algorithm will be called for each edge which is contained in E.

This algorithm will be called with specific inputs for any of the situation listed above in **:**

- Whole calendar is passed to the algorithm;
- A "small calendar" including the event added or modified and all events "connected" to it as described in the algorithm logic on both sides (antecedent events and subsequent events)
- A "small calendar" including the first subsequent event and all subsequent events "connected" to it as described in the algorithm logic

Pseudocode

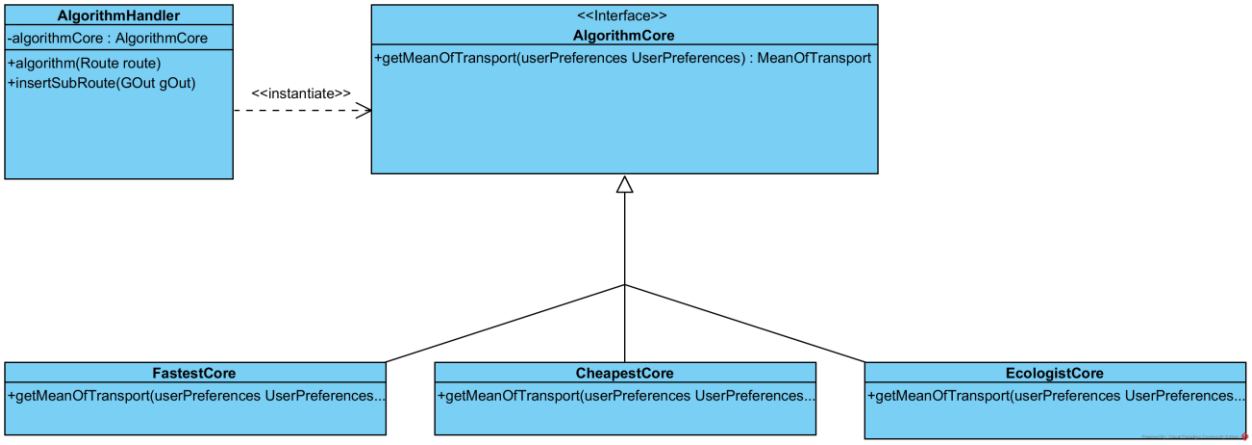
//Java pseudocode

```
//argument Calendar as List<Event> input
//Edge is a simple class that contains a pair of events and have two methods
getFirst() and //getSecond()
List<Event> V;
List<Edge> E;
V.addAll(input);
for( i = 0; i < V.size(); i++ ){
    if(V.get(i).isEndEvent()){
        V.add(new HomeEvent(), i);
        i++;
    }
    Else
    E.add(V.get(i), V.get(i+1));
}
connectionByOverlapping(E);
for(Edge edge: E)
    algorithm2(edge);

connectionbyOverlapping(List<Edge> E){
    for(Edge edge1: E){
        if(edge1.getFirst().getEnd() > edge1.getSecond().getStart){
                        for(Edge edge2: E){
                            if(edge2.getFirst().equals(edge1.getSecond())
                                addOrdered(new Edge(edge1.getFirst(),
                edge2.getSecond()));
                        }
        }
}
```

Second Algorithm

The second algorithm will receive first of all the two location (l0, l1) and the user preferences which contain the preference on the type of solution Fastest, Cheapest or Ecologist. By this preference the algorithm will choose, similar to a strategy pattern, how to compose his core.



After this first fundamental step the algorithm identifies a mean of transport by the user preferences.
If this mean of transport is contained in his private mean of transport and is available for the specific situation the Google API will be directly called. If the output of the google API cover all the trip between the two location with the desired mean the algorithm has ended his task, in case not the algorithm call itself with the starting location, as where the Google API indicates that the desired mean of transport will not be used, and with ending location the same ending location which was initially passed to the algorithm (l1). Then he consider as the google solution only for the subroute where the desired mean is used.
In case the mean of transport desired to use is a public or a sharing one the specific External API will be called and a location is returned (of a bus stop or of the nearest car to be reserved) and used as a location l2.
First of all the algorithm recall itself passing to it two location (l0, l2). Then call the Google API with the mean (in this case a sharing/public one, but at this level is not important) and follow the same behavior after the Google API call as described above.
In every call of the algorithm at the end the solution returned from the Google API is inserted (entirely or partially depending on the case described above) by a specific function to compose the solution between two events.

Input: A route (Route that is a pair of location)

Pseudocode

```
mean = chooseMean();              //call the core as described in the UML
if(!isPrivateAvailable(mean)){
    l2 = callExternalAPI();       //obtain the location from a specific external API
    thisAlgo(new Route(route.getStart(), l2));
    route = new Route(l2, route.getEnd());
}
gOut = callGoogleAPI(mean, route);
if(!isCoverTotallyByTheDesiredMean(route, mean, gOut)){
    thisAlgo(new Route(whereStopUsingMean(mean, gOut)), route.getEnd());
    gOut = cutTrip(gOut);
}
AlgorithmHandler.InsertSubTrip(gOut);
```

USER INTERFACE DESIGN

UX DIAGRAM

The UX diagram in *Figure 15* shows the different screens of the User Interface of the client Application and the interaction between them. In According to the RASD the user after logging in is able to manage is events, free time, preferences and means options in a very simple way. Obviously he can also view the Daily schedule, consult the transport solutions and update his position if he needs to do it.

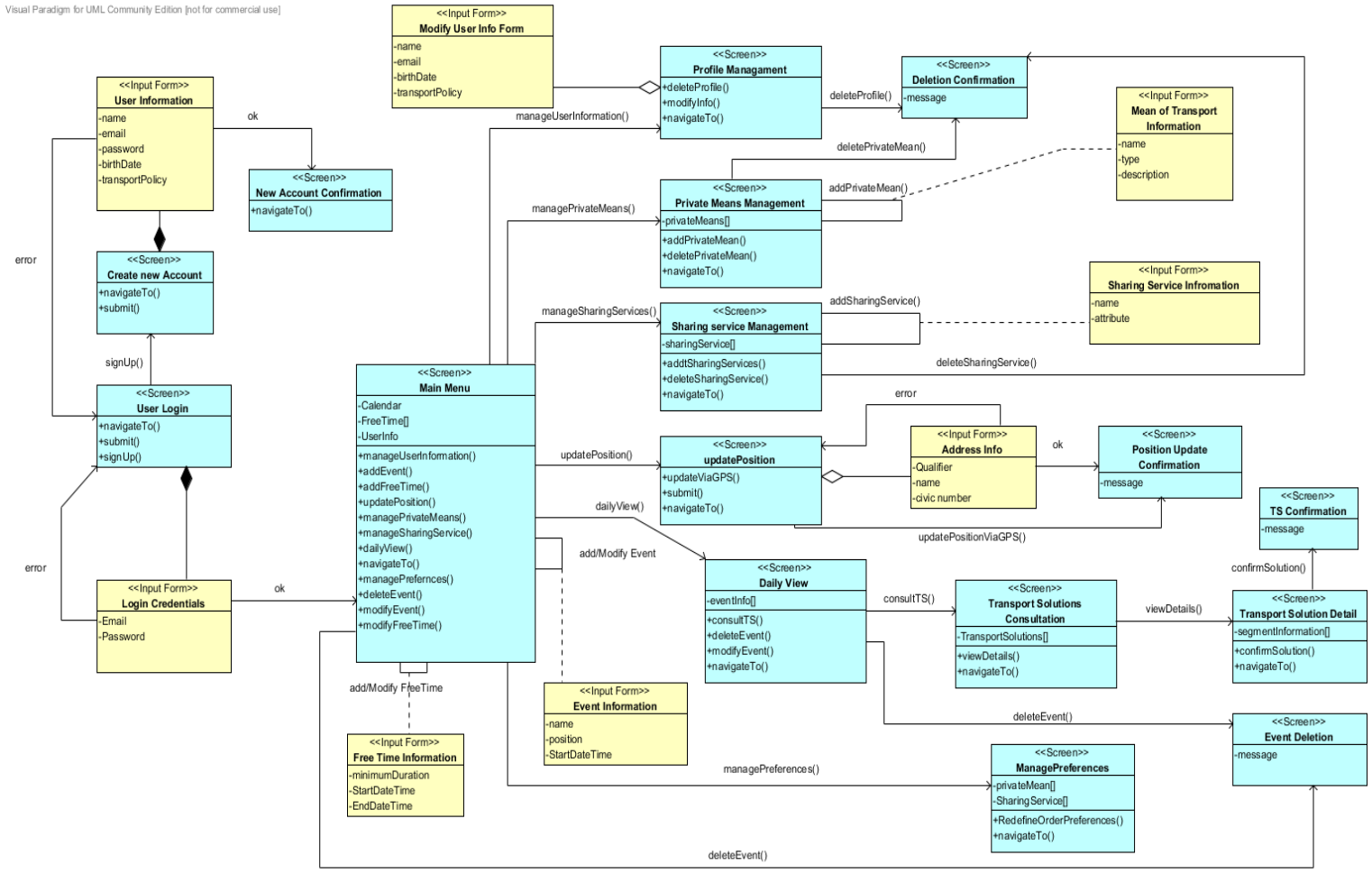


Figure 15, Ux diagram

MOCKUPS

In the RASD document we introduced a first prototype of mockups, after the definition of particular details and functionalities we propose in the following images the mockups of the Android Application.

In *Figure 16* is shown the login form with the Email and password fields, it is also offered the possibility to switch in the sign-up activity (*Figure 17).*

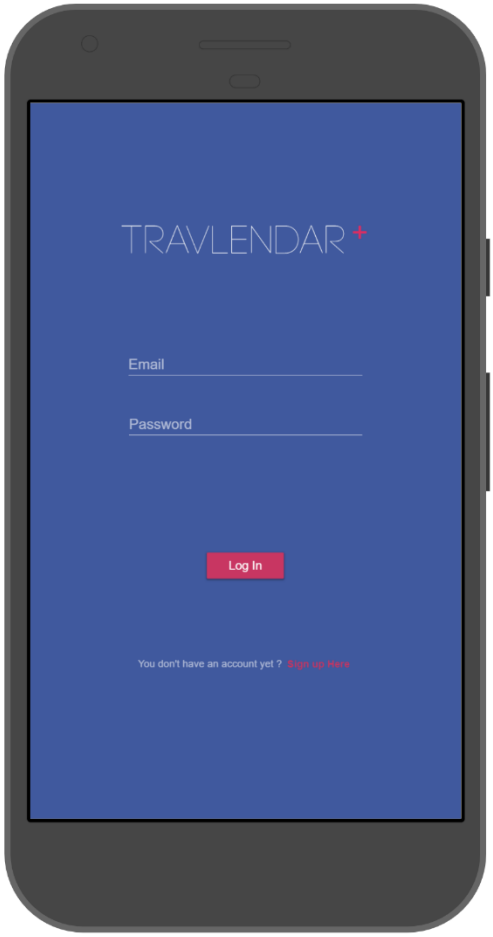


Figure 16, Travlendar Login

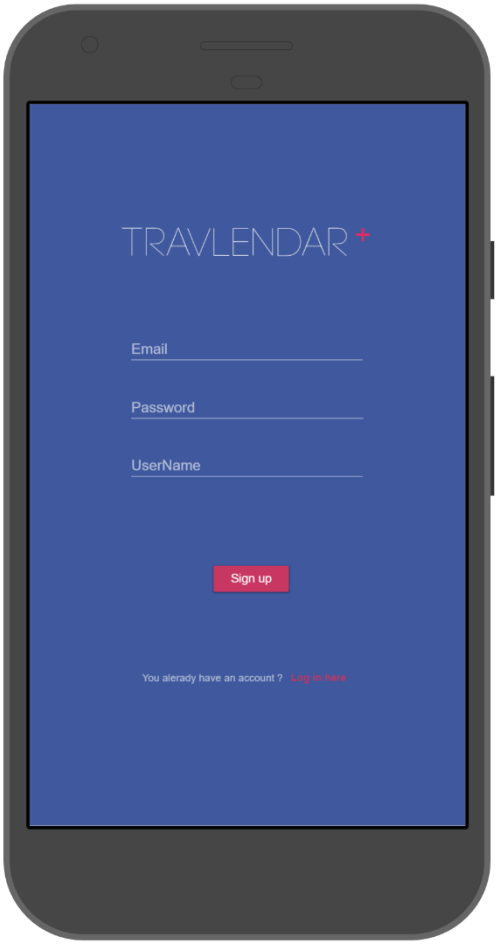


Figure 17, Travlendar Sign-Up

In *Figure 18* is represented the screen responsible to adding private means of transport, the user must select the type of mean pressing the correct radio button, Insert the name and the engine displacement.

An user could manage the Sharing Services with the interface proposed in *Figure 19,* it will be populated with the available sharing services integrated with our systems.

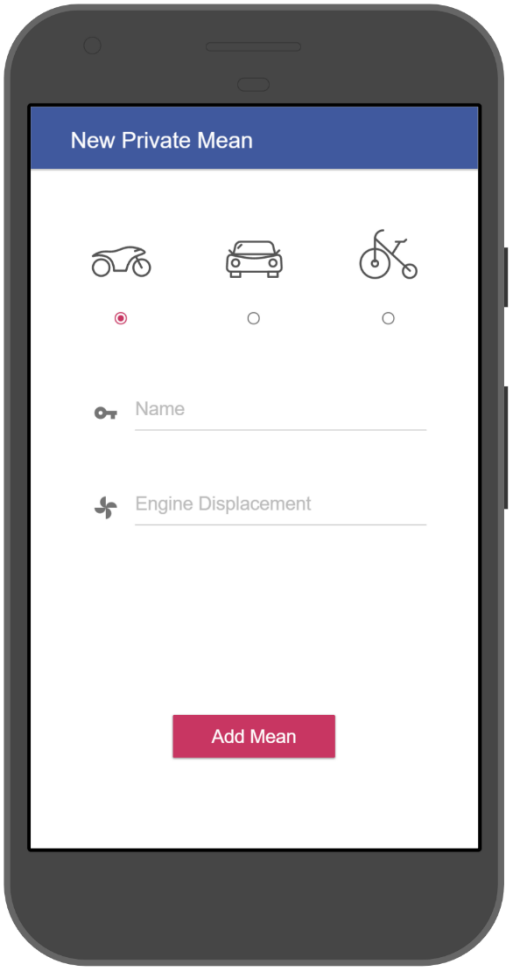


Figure 18, Adding private means

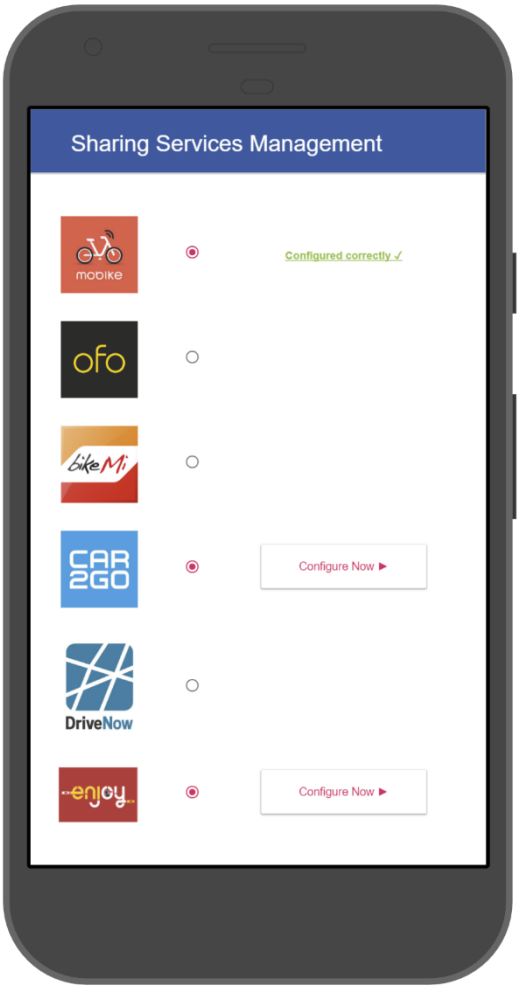


Figure 19, Managing Sharing Services

The Screen in *Figure 20* represents the home activity of the application. With The button in the left side the user could open the personal menu (SideBar) with all the links to the other screens.

Pressing the Button "Add Event" the user is redirected in the screen represented in *Figure 21* where he is able to specifying a new event and add it in the calendar.

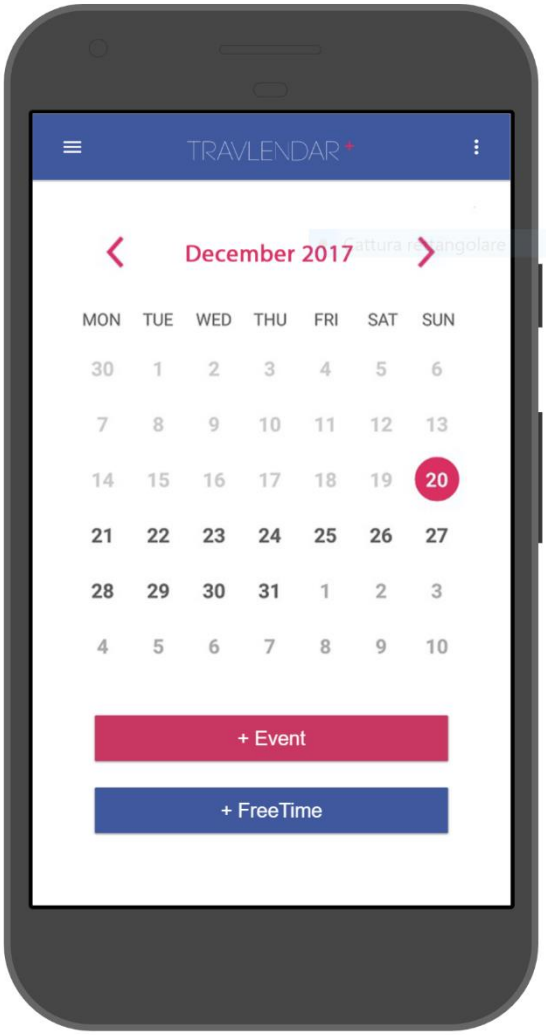


Figure 20, Travlendar Home

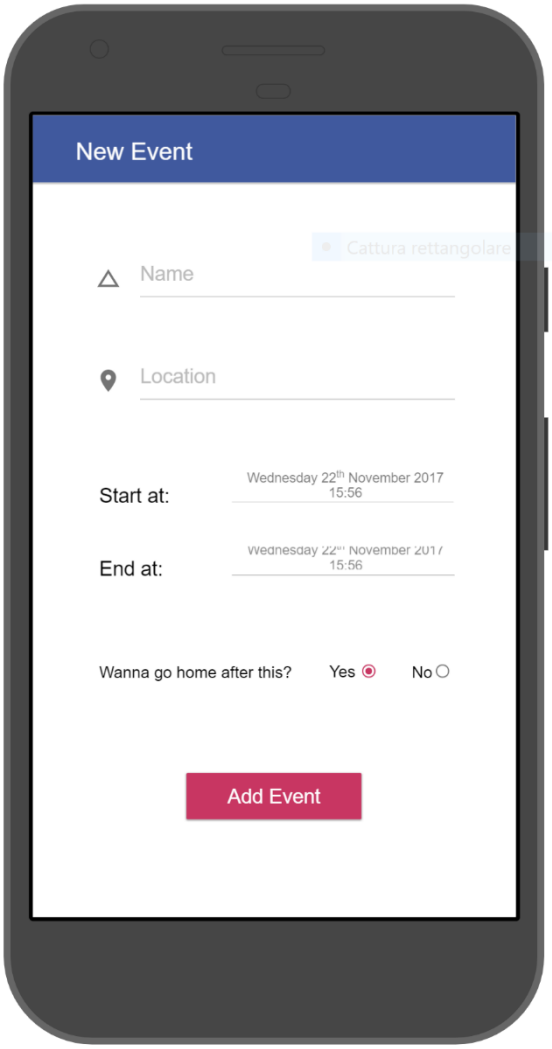


Figure 21, Adding a new event

The Screen in *Figure 22* represents the Daily View where there are represented the event scheduled during the day and the best mobility option calculated by the system on the basis of the user preference.

The user is able to change solutions pressing in the "More solutions" button or view the solutions details and be redirected in the screen in *Figure 23*. Here he could analyze the details of all the segment that compose the solution. He could Purse the solution, change it or redefine his position if it changes from the last event.

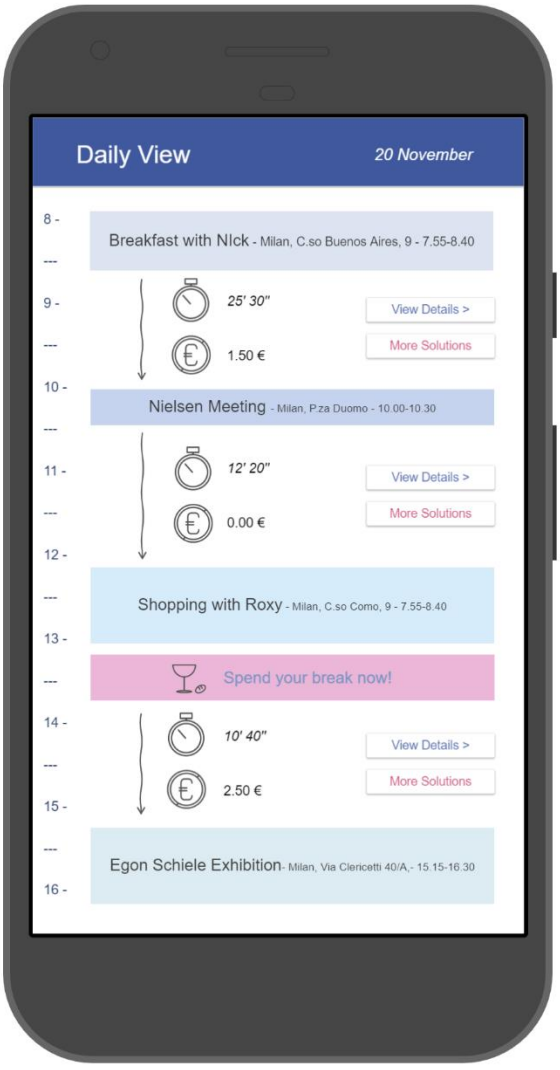


Figure 22, Daily View

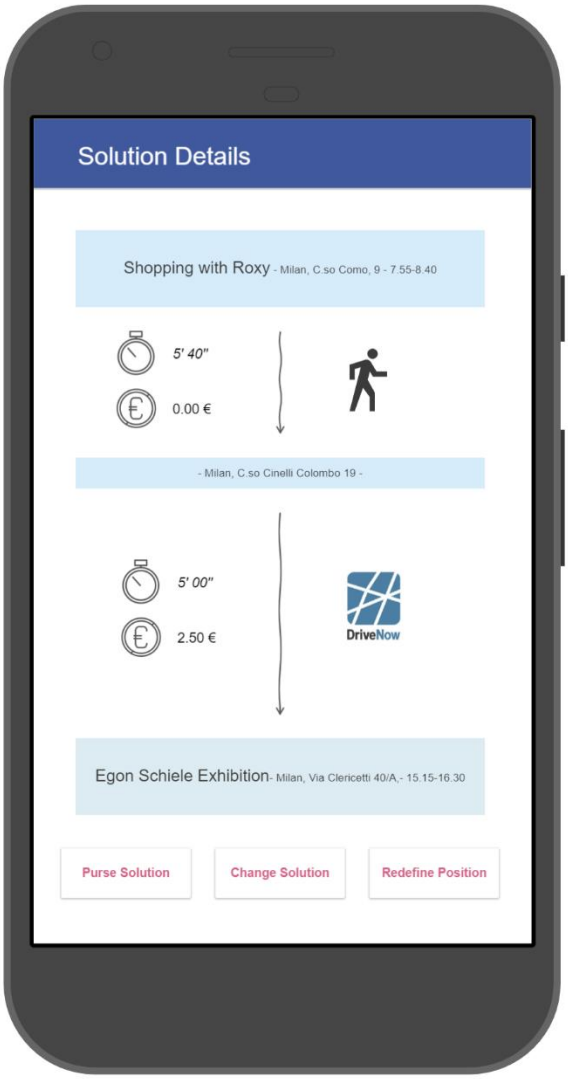


Figure 23, Solution Details

In *Figure 24* is shown the transport preferences screen. The user is able to choose his favorite policy and choose the preference's order of his private means and Sharing services previously added.

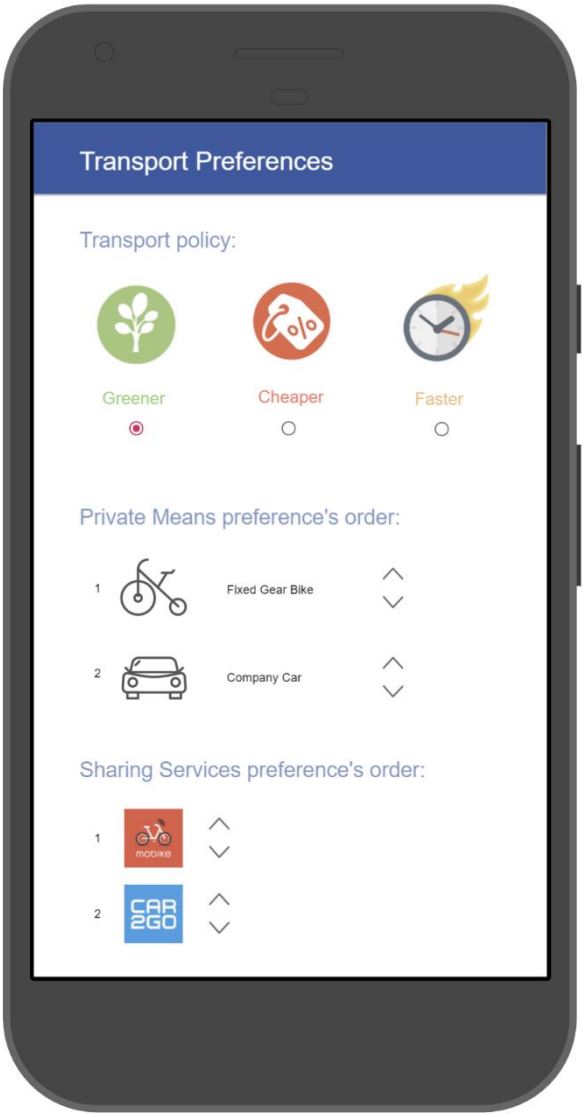


Figure 24, Transport Preferences

REQUIREMENTS TRACEABILITY

All the decisions in the DD have been taken following functional and nonfunctional requirements written in the RASD. The following list provides a mapping between goals and requirements defined in the RASD and system components illustrated in the DD.

[G1] Allow a visitor to became a registered user.
-Account Manager.
-Mobile App component.

[G2] Allow user to login to application.
-Account Manager.
-Mobile App component.

[G3] Allow user to create a new event in the calendar.
-Data Manager.
-Mobile App component.

[G4] Allow user to modify an existing event of his/her calendar.
-Data Manager.
-Mobile App component.

[G5] Allow user to delete an existing event of his/her calendar.
-Data Manager.
-Mobile App component.

[G6] Allow user to consult the transport solutions between events in the calendar proposed by the system.
-Data Manager
-Route Manager
-Mobile App component.

[G7] Allow user to re-define dynamically his instant position to re-plan the transport solution.
-Data Manager.
-Mobile App component.

[G8] Allow user to set free time (break) during the day schedule.
-Data Manager.
-Mobile App component.

[G9] Notifying events overlapping.
-Data Manager.
-Mobile App component.

[G10] Notifying the time-unreachable event.
-Data Manager.
-Mobile App component.

[G11] Allow user to configure transport preferences and external services to be used
-Data Manager.
-Mobile App component.

[G12] Allow user to set an event as ending event.
-Data Manager.
-Mobile App component.

[G13] Allow user to buy a ticket or to reserve a mean of transport of a suggested solution.
-Data Manager.
-Mobile App component.

IMPLEMENTATION, INTEGRATION AND TEST PLAN

The implementation of our system will follow two fundamental phases.
The first will see our team split on the parallel achievement of three goal:

- Creation of the server-side Database, all the tables, relations, triggers and everything is crucial for the desired behavior.
  Component to focus on:
  - DBMS
- Structure the application server and its modules in a way that basic function (for example the connection and interaction with the DB and the Client) is implemented and they are ready to integrate the algorithms for heavy calculation.
  Component to focus on:
  - DataManager (also integration with DBMS)
  - RouteManager
  - Account Manager

- Structure the client application in a way to have a ready and working ORM to hide the interaction and integration with the light DB and have ready connection with the server including the possibilities to call specific request and function on him.

The second will see our team split again on the parallel achievement of three goal:

- Front-end developing of the client-side application.
- Back-end developing of the client-side application.
- Algorithms here designed implementation on the application server and develop of the interface to integrate with the Google Maps API and Sharing Services API

At last we will connect Firebase to the Account Manager.

In case some of these goal are reached before on of the other is completed the team will consider to go on the next phase (if this happens on the first) or to carry out some pair programming.
The first test practice that our team will consider is to use test-driven development on the module that are prone to this kind of practice (controller modules in particular of the routeManager) so our policies is to anticipate more possible the test phases and to parallelize it with the develop in order to avoid more possible bugs that can grew with the growth of our systems and the various component. So is fundamental to do unit test (Junit for Java modules) before a module of a component is released and integrated with the other modules. The type of test (black box and white box) depends on the module that is tested: algorithm of the route manager requires black box and white box, the modules that interfaces two components or modules that parse data (as Json data files) requires in particular black box testing and so on.

EFFORT SPENT

Agostini Andrea:

- INTRODUCTION: 0.5h
- ARCHITECTURAL DESIGN: 8h
- ALGORITHM DESIGN: 0.5h
- USER INTERFACE DESIGN: 12h
- IMPLEMENTATION AND TEST PLAN: 1h

Ciampiconi Lorenzo:

- INTRODUCTION: 0.5h
- ARCHITECTURAL DESIGN: 5h
- ALGORITHM DESIGN: 10h
- USER INTERFACE DESIGN: 0.5h
- IMPLEMENTATION AND TEST PLAN: 4h

Es-skidri Rachid:

- INTRODUCTION: 0.5h
- ARCHITECTURAL DESIGN: 15h
- ALGORITHM DESIGN: 1h
- USER INTERFACE DESIGN: 0.5h
- IMPLEMENTATION AND TEST PLAN: 3h

REFERENCES

[1] Google, Android Developers - Design https://developer.android. com/design/index.html

[2] Software Engineering 2 Project, AA 2017/2018 - Project goal, schedule and rules

[3] Spring Framework - https://projects.spring.io/spring-framework/

[4] GOF Patterns - http://www.blackwasp.co.uk/gofpatterns.aspx

[5] Oracle Documentation JEE -https://docs.oracle.com/javaee/7/