POLITECNICO
MILANO 1863

TRAVLENDAR+

DD

Design Document

Kostandin Caushi 898749

Marcello Bertolini 827436

Raffaele Bongo 900090

Date 26/11/2017

Version 1

Table of Contents

List of Figures

1 Introduction

1.1 Purpose

This document has the aim of entering into the details of Travlendar+ system. We will show the software architecture that we have designed for our system with different levels of abstraction, analyzing deeply the main components. Additionally, we will exhibit a runtime view of the system, showing as the various components will interact with each other, specifying architectural and design pattern used. Finally, we will present the critical algorithm implemented, a requirement - software components correspondence and a high level plan about the implementation and integration of the various components.

1.2 Scope

Travlendar+ is a calendar-based application that has the aim of managing the many meetings, events and appointments that a user has to deal with every day.

The system will let the user create events in his personal calendar, checking if he is able to reach them on time and supporting his choices about the way of reaching the location.

Travlendar+ will also give the user the possibility to buy tickets of a town's public and private means of transport and it will also allow him to manage his travels to reach other cities, creating specific travel events in the calendar section. The system will offer other additional features :

- The possibility to register the season ticket for the public transport. Travlendar+ will notify the user when the expiry date is near.
- The possibility to set the starting time, ending time and the preferred duration of every day lunch. The system will guarantee to reserve at least 30 minutes for this purpose.
- In case of outdoor trips, the user will be able to insert the period he will spend out of town and the system will suggest him the most convenient transport tickets available, keeping in mind the information given.
- The possibility of setting the anticipation time for reaching the various events. The system will warn the user when he needs to leave in order to arrive on time.

1.3 Definitions

- *Overlap :* if there's an event previously planned in that specific moment.
- *Reachability :* if an event is reachable from the previous one.
- *Tier :* a physical structuring mechanism for the system infrastructure.
- *Server :* a computer program or a device that provides functionalities for other programs or devices, called clients.
- *Interface :* is a shared boundary across which two or more separate components of a computer system exchange information.
- *Design Pattern :* it's a general reusable solution to a common problem that occurs in a given context in software design.
- *Sub-Milestone :* a checkpoint of the implementation process.
- *Milestone :* a set of Sub-Milestones. The various teams have to reach a determined result for the milestone day.
- *Front-End :* it's a part of the system with which the user interact directly.
- *Back-end :* it's a part of the system that contains all the system logic and interacts with the Data Bases and the external software services, but it's hidden to the user.

1.4 Acronyms

- *RASD :* Requirement Analysis and Specification Document
- *API :* Application Programming Interface
- *SOA :* Service Oriented Architecture
- *MVC :* Model View Controller
- *HTTP :* HyperText Transfer Protocol
- *HTML :* HyperText Markup Language
- *DB :* Database
- *REST :* REpresentational State Transfer

1.5 Abbreviations

- [Gn] : Goal n
- [Rn] : Requirement n

1.6 Reference Documents

- Mandatory Project Assignments.pdf
- Travlendar+ RASD
- Project Management part 1.pdf
- Project Management part 2.pdf
- Verification and Validation.pdf
- The Component Diagram - IBM developerWorks

1.7 Document Structure

This paper is divided in 6 chapter:

1. The first chapter is composed by an introduction of the system, its application domain, its goals and a glossary containing the most common expression used in order to give to the reader a basic knowledge of the system and to make him understand better the subsequent parts.
2. In the second chapter it is defined the system architecture. Starting from a High level elements description, it goes deeper step by step inside the various system components and their interactions. The architecture styles and the design patterns that we have used in designing our system are also shown in the last subsections.
3. The third paragraph is focused on the algorithm we have designed to perform all the system check operations. The problem we faced and its solution are precisely described in this paragraph and a Java code example is also provided.
4. In the fourth paragraph some mockups, explaining the user interactions with the system, are shown and described.
5. In the fifth chapter is shown how the requirements we have defined in the RASD map to the design elements that we have defined in this document.
6. In the sixth chapter a startegy to address the Travlendar+ implementation is provided. Furthermore, we have defined a precise team structure describing the interactions between the various team parts. Finally, we have defined a table of the possible risks that may occur during the development process and a list of the possible solutions.

2 Architectural Design

2.1 Overview

In this chapter we will analyze the proposed architecture and components of Travlendar+ system.

The proposed architecture is composed by three tiers :

- *Presentation Tier :* it's represented by the Browser and the Mobile App, the View part of our system.
- *Web and Business Tier :* it's represented by the Web Server, which responds to the user's HTTP requests, and the Application Server, which contains all the Business Logic.
- *Database Tier :* it's represented by the DB Server, that contains and manages persistent data in an efficient way.

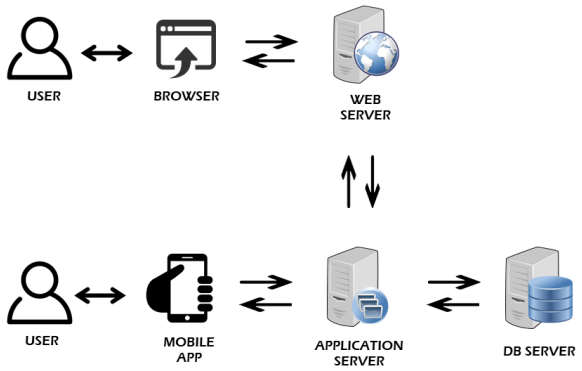

Figure 1: Proposed Architecture

2.2 High Level Components and Their Interactions

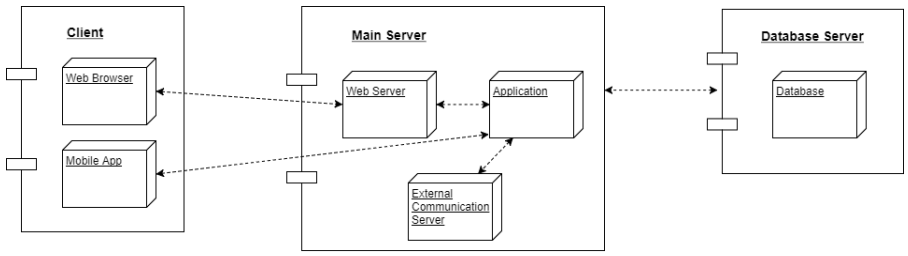

Figure 2: Components High Level

The system is divided in three main layers: Presentation Layer, Business Layer and Data Layer. The presentation layer is both a web-based application and a mobile application. For the web application we will have a very thin client with the only aim of performing requests, through a web

server, to the Business layer and receiving the HTML pages with the demanded information. On the other hand, the mobile application will not require an interface with the web server because it will communicate directly with the server.

The main server is composed by three parts : the Web Server, mentioned here above, the Application and the External Communication Server. The application part of the main server contains all the system's logic: it receives requestes and input data from the client side and performs all the necessary operations. The External Communication Server part is in charge to handle all the comunication with the external services to retrieve data of interests from other systems in order to achieve the system goals. Our system, being a calendar based application, will have the main issue of managing a richly structured body of information. In our case, this information will be persistently stored in multiple Data Bases, each one containing a precise schema with a precise kind of data, and they always have to reflect the true state of affairs. For this reason, every client request, implicating a changing data operation is preceded by a set of controls performed by the Business logic in order to maintain the database's consistency.

2.3 Component View

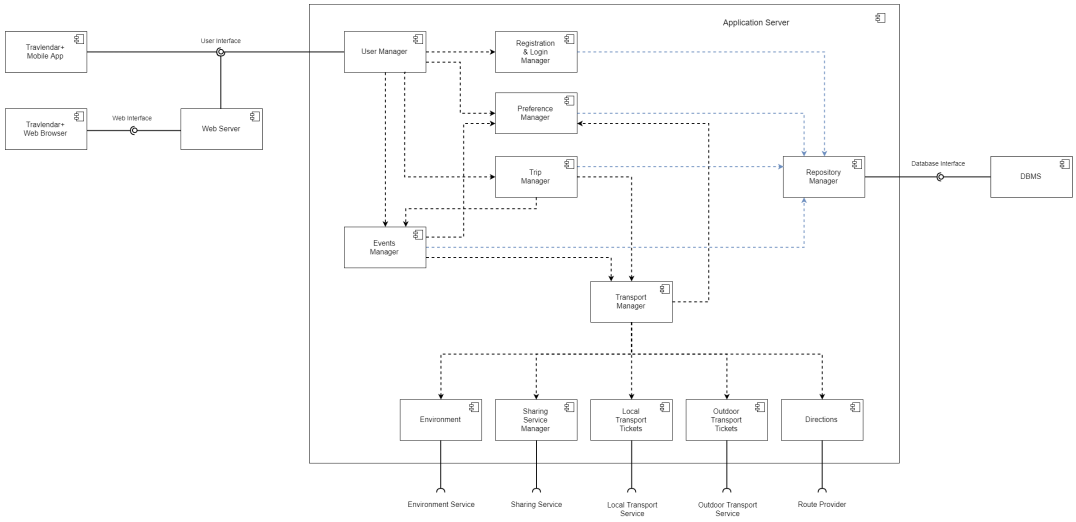

Figure 3: Component View

2.3.1 Travlendar+ Mobile App

This is the component responsible for the communication between user devices, i.e. smartphone, and the rest of the system. It will exchange REST messages with the *userInterface* and rendering the UI based on the server replies.

In order to communicate with the server the component will use specific frameworks and libraries depending on the specific device used by the user (i.e. Android SDK Platform for Android smartphones).

This component represents the View part of our MVC pattern.

2.3.2 Travlendar+ Web Browser

This is the component responsible for the communication between the user browser and the rest of the system.

It sends HTTP requests to the *webInterface* in order to get back from the server HTML pages and static resources like CSS and JavaScript files.

2.3.3 Web Server

This component is responsible for the HTTP responses of the *webInterface*. Its main function is to elaborate pages, generate contents dynamically and send them back to the browser. These processes are performed by the Web Server instead of the user Browser to manage efficiently the user CPU load.

An example could be the Apache HTTP Server. It's a cross-platform server, highly scalable, that using a *gzip* module reduces web pages size (weight).

2.3.4 User Manager

This is one of the most important server component that handles the communication with the user and takes care of all possible breakouts and errors that may occur at runtime. It also manages all user requests redirecting them to other server specific components.

2.3.5 Registration & Login

It allows the user to register and log into the system, memorizing and verifying user credentials.

2.3.6 Preference Manager

It allows the user to set all his preferences, such as carbon footprint, TAGs, residence location, etc... with the purpose to memorize these data in the DB through the Repository Manager and retrieve them whenever it's needed.

2.3.7 Trip Manager

This component handles all user trips, allowing him to add, delete or edit them. It also allows the user to buy tickets to reach the trip location (through the Outdoor Transport Tickets component) and the ones for local transport services in the destination city (through the Local Transport Tickets).

2.3.8 Events Manager

This component takes care of user calendar, managing the events that he wants to add, delete, edit or visualize.
Thus, anytime he can :

- *insert or edit an event :* the component will check through the Repository Manager if in the DB there is an overlap and if the event is reachable from the previous one (using also Transport Manager to verify route time).
- *delete an event :* the component will simply delete it in the DB.
- *visualize an event :* the component will check through the Transport Manager the available means of transport, the time needed and suggest the route to reach the event.

2.3.9 Transport Manager

This component handles all the means of transport and tickets. Concerning transport, it will filter the means according to :

- the means selected in the event's TAG;
- the CO_2 preference;
- the means selected in the "Available Means" in Preferences;
- the available means to reach the event (suggested by the route provider);
- the weather forecast;
- eventual strikes.

It allows also to buy local transport tickets, if needed.

2.3.10 Directions

This component will handle the communication with the route provider in order to get maps, means and routes to reach a specific destination. To get these information it will use API requests (i.e. Google APIs). It will also parse and elaborate the API responses in specific structures, ready to be used by the server components when needed.

2.3.11 Environment Manager

This component will be used by our system to check and warn the user about the weather conditions and eventual transport strikes. It will use API requests to communicate with the external services.

2.3.12 Sharing Service Manager

This component will allow our server to communicate with the external sharing services and retrieve the nearest sharing mean of transport GPS position.

2.3.13 Local Transport Tickets

It will take care of the communication with the local transport services, in order to get information about the available tickets, suggesting the best deal based on his travel needs.

2.3.14 Outdoor Transport Tickets

It will take care of the communication with the outdoor transport services, in order to get information about the available tickets and means of transport.

2.4 Deployment View

Here's the Deployment Diagram of our system. We will use the Apache Server, for the Web Server, and MySQL, for the DBMS.

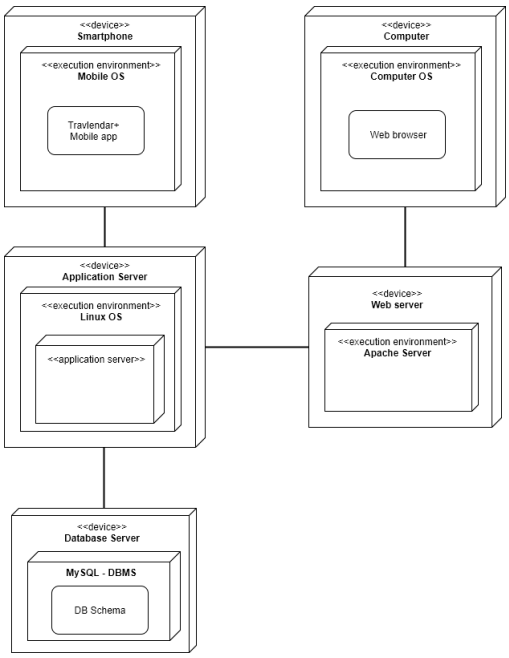

Figure 4: Deployment View

2.5 DB Structure

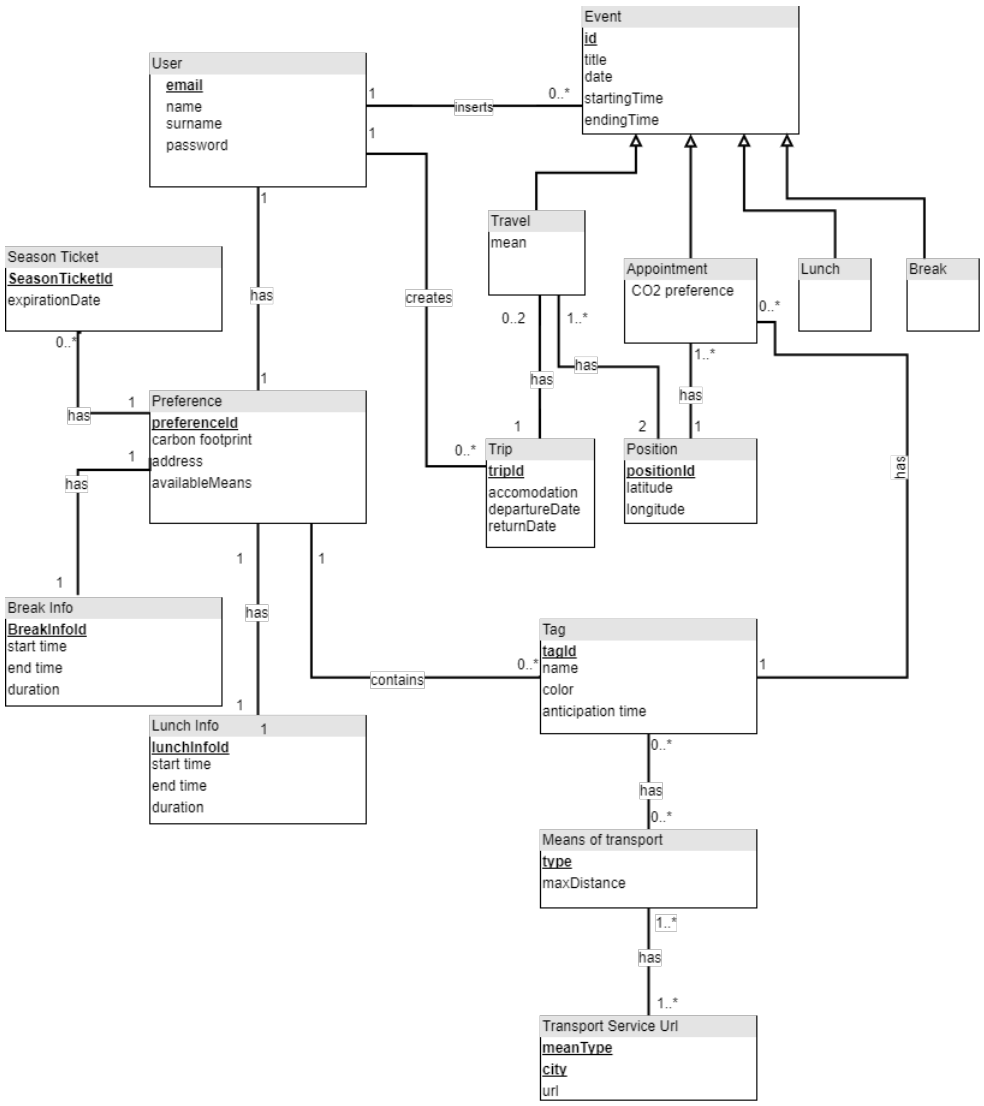

Figure 5: DB Structure

2.6 Runtime View

In this section we will see some Runtime Views in order to see how the components interact according to specific requests. For the client-side we have taken into consideration the Web Browser only, in order to evaluate the interaction between the Browser and the Web Server, the Web Server and the Application Server components.

2.6.1 Registration

This Runtime View shows a user Registration process.

First of all the Web Server sends to the user the registration_form to be fulfilled with his credentials. As soon as the information are inserted, the form is sent back to the Server.

At this point, it's important to check if the email is already associated to another account. In order to do that, the server asks the DBMS to search for it. If the get_result it's equal *Null*, it means that the email it's not used and the server can complete the registration saving in the DB the user credentials. Otherwise, the server will notify the user that the inserted email cannot be used.

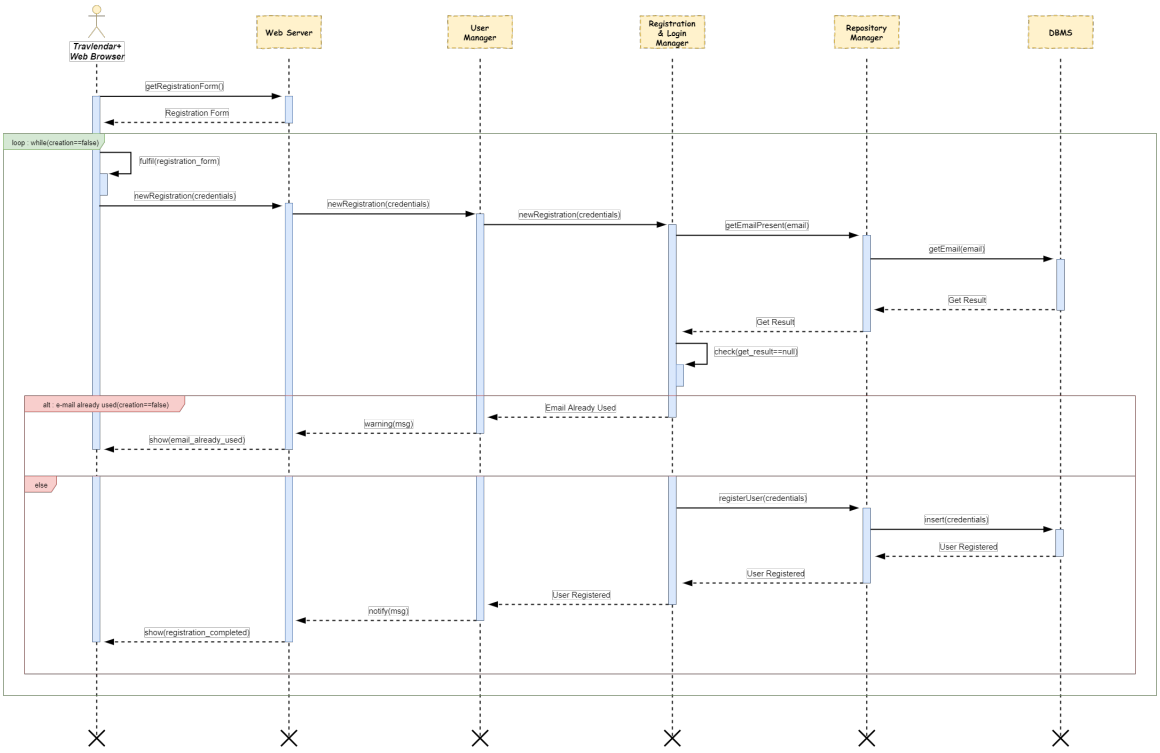

Figure 6: Registration Runtime View

2.6.2 Login

This Runtime View shows a user Login process.

As in the previous runtime, the Web Server sends to the user the login_form to be fulfilled with his credentials. As soon as the information are inserted, the form is sent back to the Server to be checked.

First of all it has to check if the user email is present in the DB and then if the user password is correct. In order to accomplish this task in an efficient way, the server asks the DBMS to retrieve him the password related to the inserted email. If the get_result is equal *Null* then there is no account registered with that email. Otherwise, the server, more specifically the Registration & Login Manager, has to check if the get_result is equal to the inserted password. If that check succeeds, then the user is logged and the server gets from the DB the events related to the current month and shows them to the client.

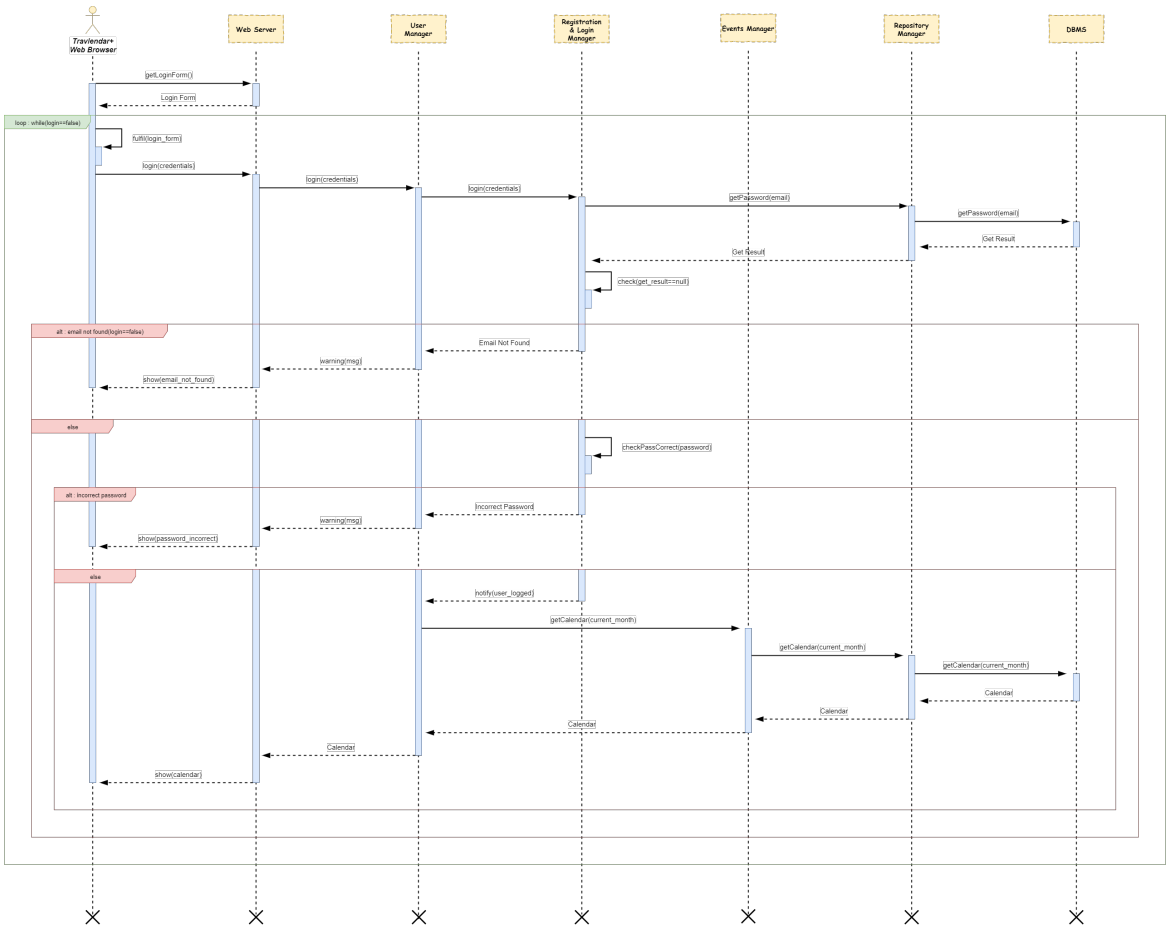

Figure 7: Login Runtime View

2.6.3 Add Event

This Runtime View shows the Add Event process.
In order to reduce the diagram size we have made some premises :

- There is a Previous and a Next Event, before and after the event that the user is adding. In this way we don't have to show runtime the check that verifies if the getPreviousEvent() or the getNextEvent() is equal to *Null*.
- The user hasn't already configured the lunch preference, so there is no lunch event in the calendar. In this way we don't have to run the check for verifying that the inserted event overlaps with a lunch event. In fact, if the overlapped event was a Lunch, the system would have to check if it could be moved or reduced respecting the lunch contraints.

Even in this runtime, firstly the user has to complete the event_form and send it to the server. Once Events Manager receives the data, it has to check :

1. *If the event Overlaps :* Events Manager asks the Repository Manager to search in the DB if there's a planned event at the same time of the inserted one. If the get_result is equal to *Null* then there's no overlap.
2. *If the event is reachable from the previous one :* Events Manager asks the Repository Manager to get the information of the previous event, especially its location. Once received, it sends both the locations to the Directions Component in order to get the route time. Finally, it checks if the event is reachable or not.
3. *If the next event is reachable from the inserted one :* it's like the previous check but it gets from the DB the information of the next event, instead of the previous one. If all the checks succeed, the server will add the event in the DB and show it to the user.

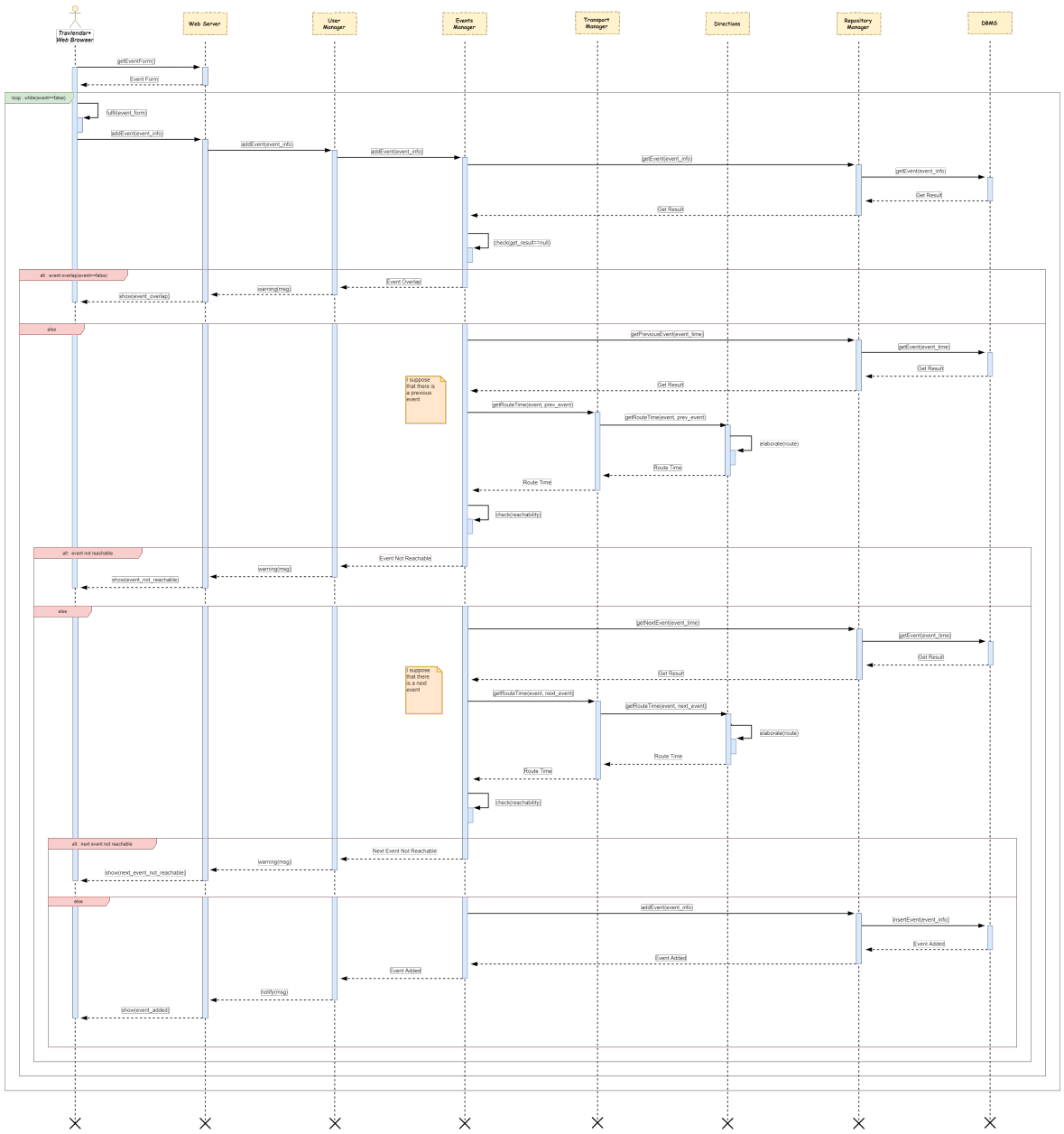

Figure 8: Add Event Runtime View

2.6.4 Visualize Event

This Runtime View shows a user's Visualize Event process.

In order to reduce the diagram size we make a premise : the system has already checked that there exists an event before the one that the user wants to visualize.

In this diagram the user asks the server to show him an event, so this mean that the server has to get the event info and the means of transport available to reach it. In order to do that the Events Manager, first of all, gets from the DB the event info and the previous event info. Next asks the Directions component to compute the route between this two locations and to send back for each mean of transport the time needed to cover the travel.

Before sending back the result, Transport Manager filters all the means based on preferences, weather and strikes.

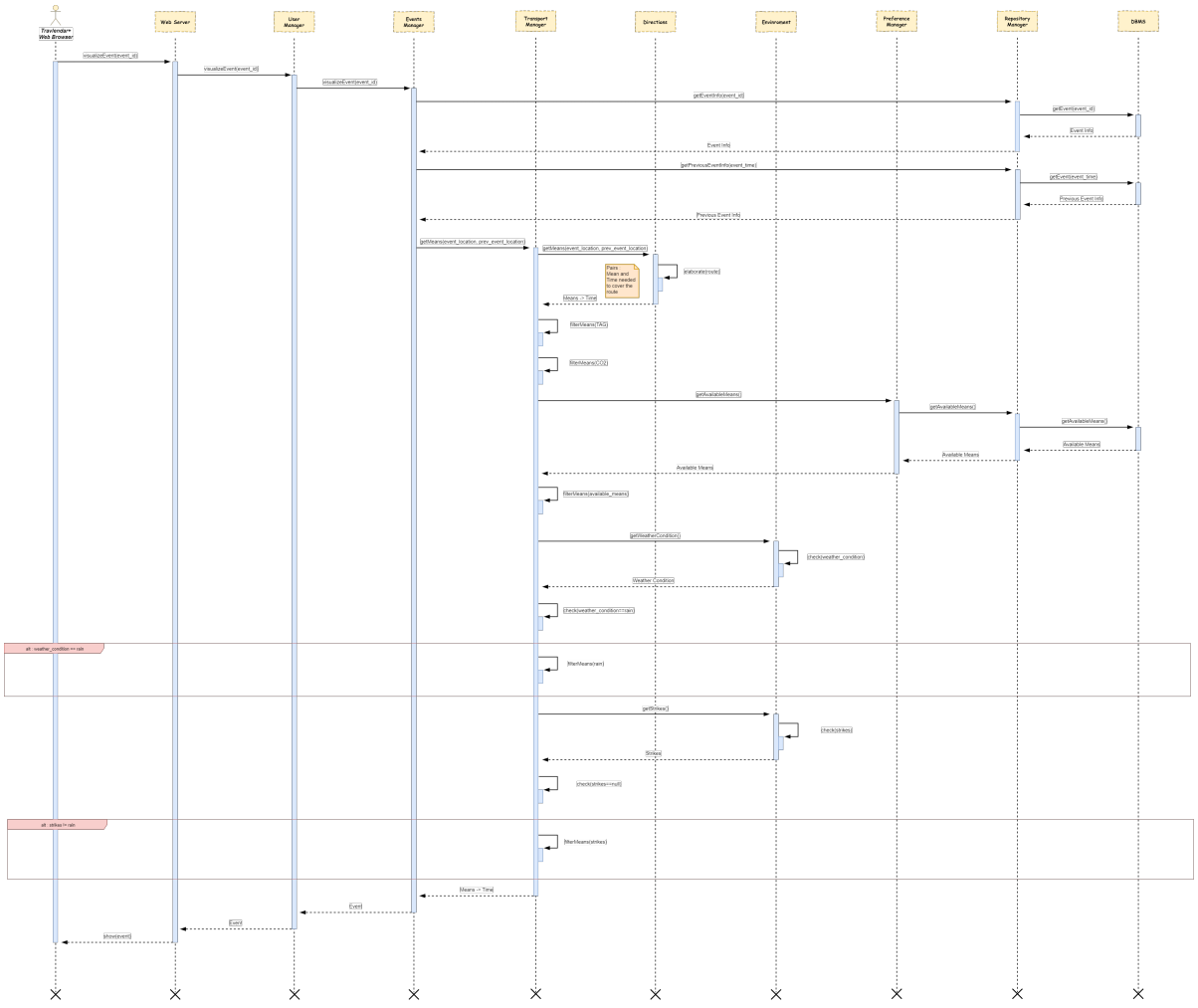

Figure 9: Visualize Event Runtime View

2.6.5 Visualize Directions

This Runtime View shows a user's Visualize Directions process.

In order to reduce the diagram size we have made a premise : the system has already checked that exists an event before the one that the user wants to get directions.

The Server receives the user request to show him the directions to reach a specific event. In order to do that, Events Manager, first of all, gets from the DB the information about the choosen event and the previous one. Once it receives them, asks the Directions component to elaborate the route between the given locations and finally it shows the result to the user.

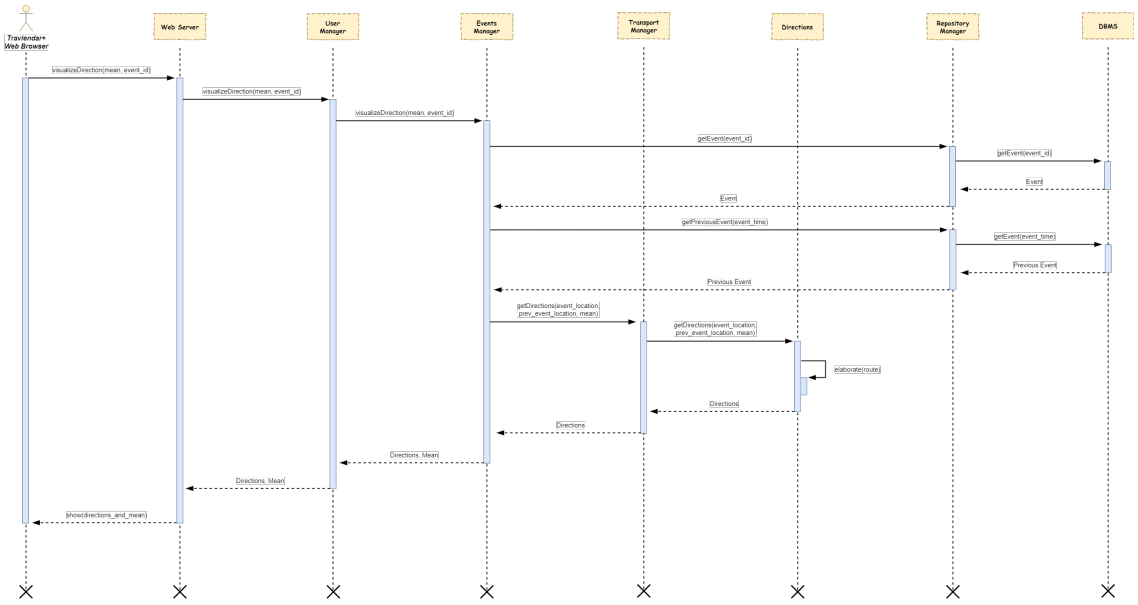

Figure 10: Visualize Directions Runtime View

2.6.6 Weather & Strike Check

This Runtime View shows the Weather and Strike Check process.

In order to implement this in an efficient way we decide to create two server threads that wakes up every 24h and checks the weather conditions and the presence of strikes for the next day.

As seen in Add Event diagram, in the first part the user adds an event for tomorrow, just to make sure that there's something to check for the next day.

At midnight the Weather thread, in the Environment component, wakes up and asks the DB for all the locations that it has to check. Once it receives them, it checks if tomorrow will rain and in that case will notify it to the user.

The same happens with the Strike thread.

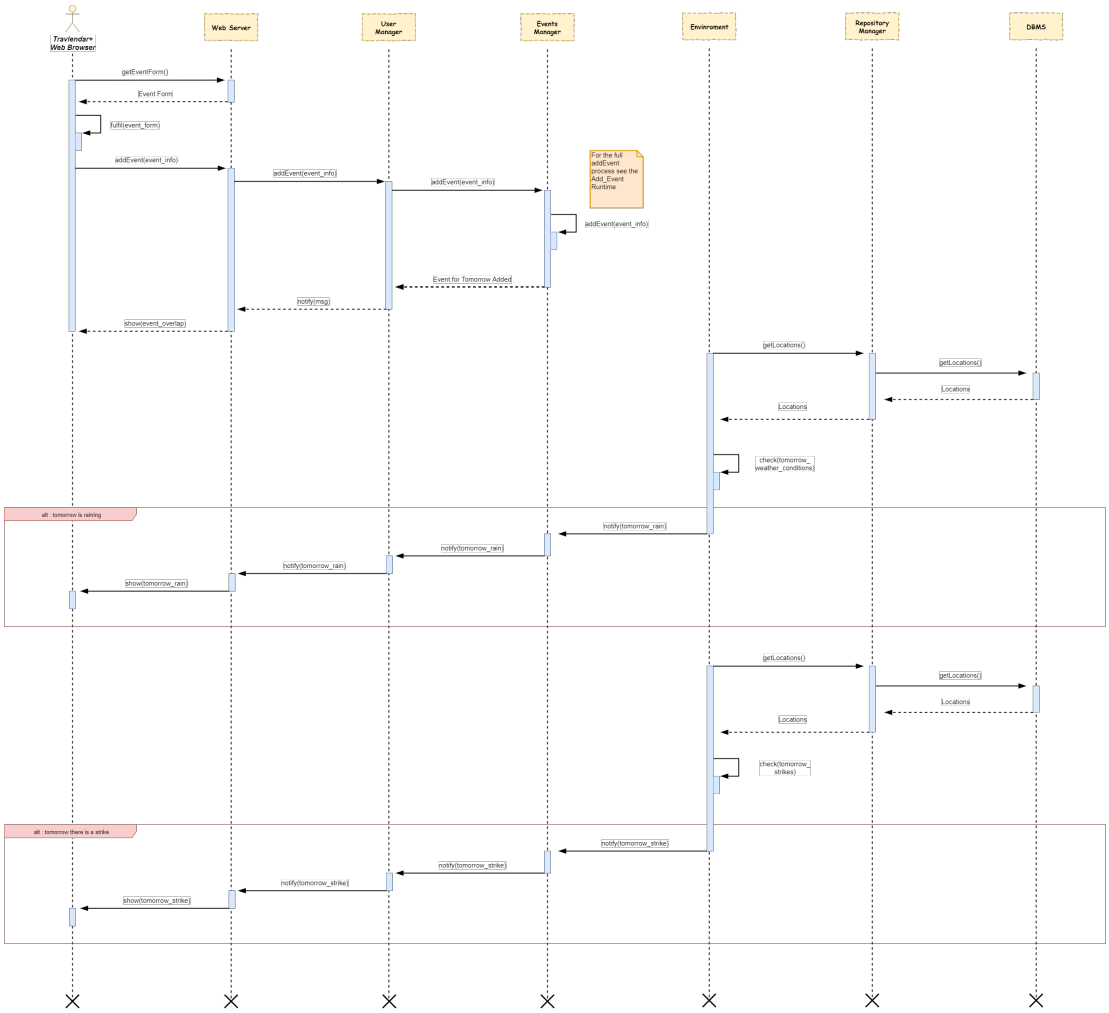

Figure 11: Weather & Strike Check Runtime View

2.6.7 Add Trip

This Runtime View shows a user's Add Trip process.
In order to reduce the diagram size we make some premises :

- The Overlap and Reachability checks aren't fully showed. For more details watch the Add Event Runtime View above.
- The user has inserted just a departure travel, so all the checks for the travel event are executed just one time, instead of two.

In this diagram, the user receives a trip_form to fulfil and to send back to the Web Server. Once the server receives the form, it sends that to the Trip Manager, which, first of all, checks if the trip is already present in the DB (get_result!=*Null*).

After that it has to check if the departure travel event can be added to the calendar. This mean that it has to ask the Events Manager to check the Overlap and the Reachability.

If all checks succeed Events Manager adds to the DB the travel event, while Trip Manager adds the Trip.

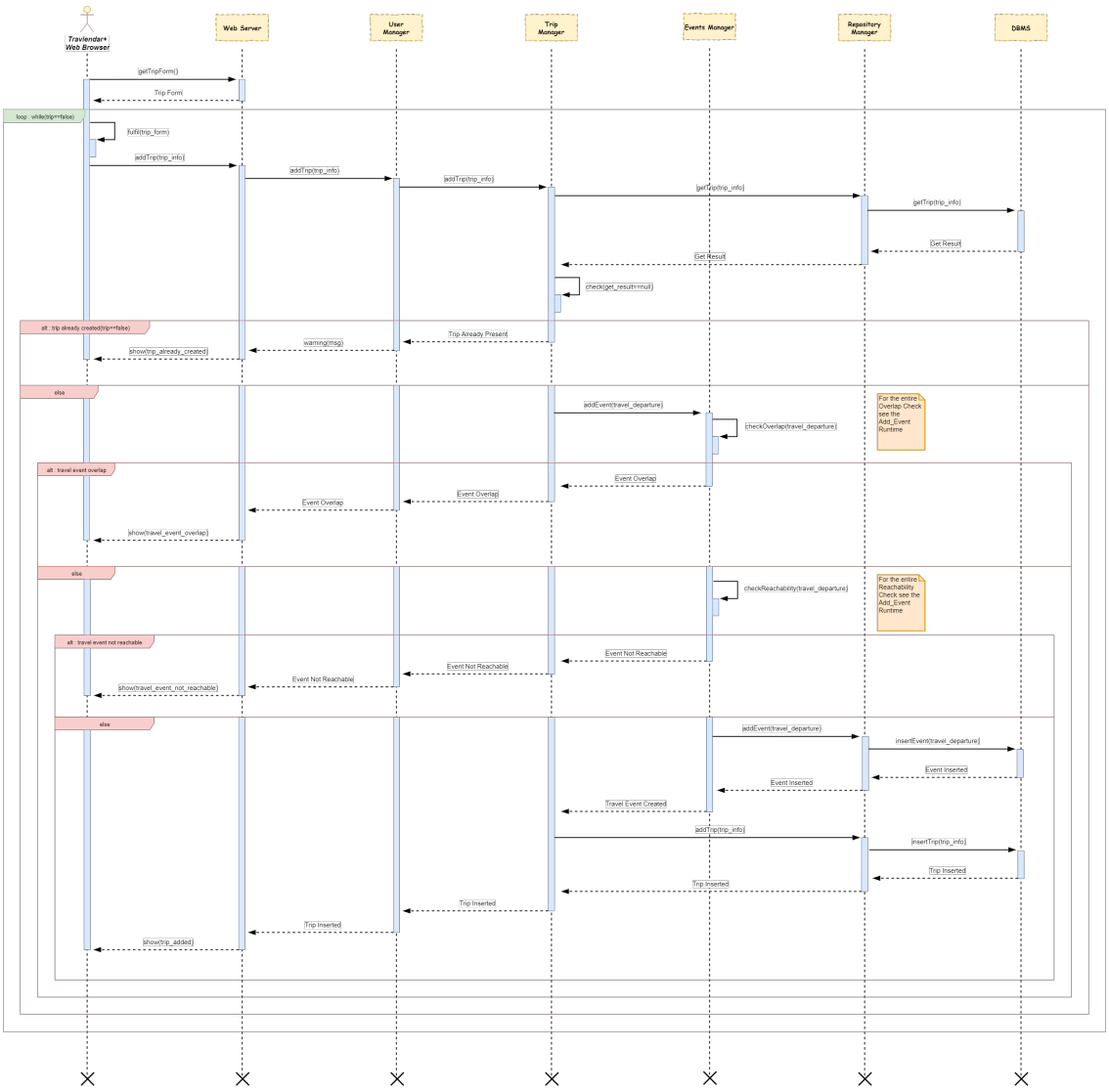

Figure 12: Add Trip Runtime View

2.7 Selected Architectural Styles and Patterns

Our system architecture proves to be a mix between three well known architectural styles, in particular :

- *Client/Server Architectural style*
- *Main program with subroutines architectural style*
- *Service oriented Architectural style*
- *Event Based Architectural style*

Travlendar+ is both a web-based application and a mobile application. For the web application we will have a very thin client with the only aim of performing requests to the Business Logic through a web server. On the other hand, the mobile application will not require an interface with the web server because it will communicate directly with the server.

The main server will contain all the logic of the system. The main component of the logic will be the User manager. This component will perform as the Main program in the *Main program with subroutines style*: it will receive the requests from the clients and call the right sub-components for accomplishing the goals related to them.

An essential of our system business, as said in the RASD, will be to provide directions related to the precise travel means, sharing means' position and tickets information about local and outdoor public transport. All this information will not be stored in our DBs, but they will be retrieved with API REST requests to an appropiate external services. For this goal we will use a *SOA style*. In our system we will have a component called Transport manager that will be able to distinguish the kind of the request and it will delegate to another precise component, devoted to a certain type of external services connection, the aim of submitting it to the right external service . Once that the information has been received, its manipulation will be performed internally to the business logic of the system. In this way we will have the possibility of adding new components for incoming kind of means of transport, also personalized, in a very simple and scalable way.

The last architectural style exploited is the *Event based Architectural style*, and it is used for the notification system that will be always waiting for a particular event that, if happens, triggers a notification that is immediately sent to the right clients subscribed to the topic.

2.8 Design patterns

2.8.1 MVC

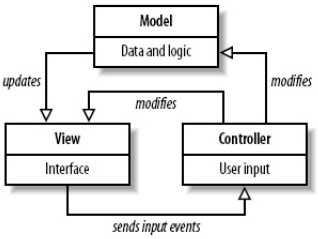

Figure 13: MVC Pattern

The Database server contains all the software's data and constitutes the model part. The presentation layer, composed by the web-based application and the mobile application, represent the view that is released to the user. Finally the main server, that is the business logic layer, is the control part.

2.8.2 Observer

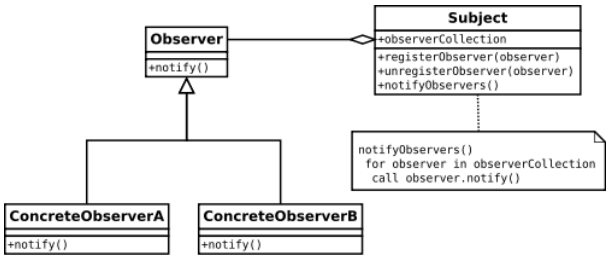

Figure 14: Observer Pattern

Our software has to be able to advise the user that he has to take a certain mean of transport for allowing him to arrive on time, that the registered season ticket is going to expire or also if a strike or bad meteorological conditions are forecasted. All these events have to be supported by a notification system. These needs will be implemented with the *observe pattern*. There will be a specific group of event listener, namely objects that extend a common abstract class. The system will have as many event listeners as is necessary and they can be added freely for future expansion. When during a process a state of interest for a listener is changed, some checks are performed and, under some particular conditions, a notification is created and sent to the user.

2.8.3 Strategy

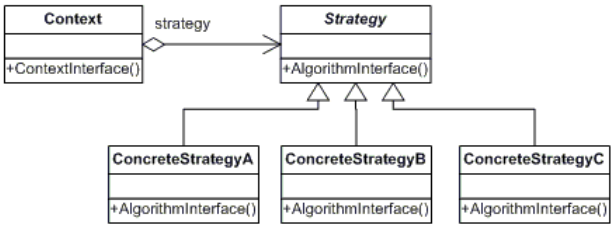

Figure 15: Strategy Pattern

The filters applied on the data to show particular results, based on the user's choices, for example to advise the best means of transport for a destination, will be implemented with a strategy pattern. The class that will order the results will have an instance of an abstract class that will be implemented by some concrete classes, representing different ordering strategy. In this way the system will be able to adapt its strategy runtime and it will be very simple to add new strategies creating new classes that will implement the abstract strategy class previous mentioned.

2.8.4 Composite

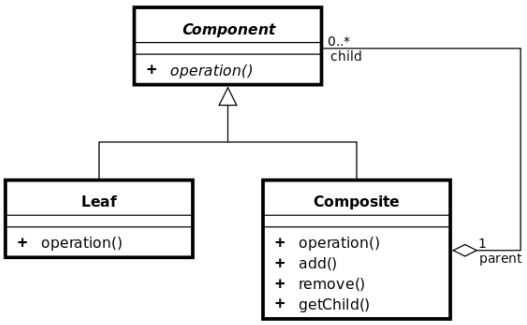

Figure 16: Composite Pattern

We will use this pattern in our system to perform, in a cleaned and elegant way, the various checks that will have to be applied on the user input. There will be a abstract class called Checker that will be extended by Composite checkers and Checkers. Each composite checker will contain one or more checkers and both implement a common interface using the *check* method. The class that will have to manage a specific user input will have one or more composite checkers. When a method that have to process the user input data, addresing them to an external service or to another component that will insert such a data in a DB, the check method of all the composite checker in such a class will be called

and each composite checkers will recursively call his checker's check method. In this way all the checks will be applied and if something is wrong, the subsequent operation will not be executed and all the possible problems linked with that will be avoided. Thanks to this pattern is possible to create new checkers/composite checkers very easily and to deal with complex and composite check that will be needed in future software expansions.

2.8.5 Façade

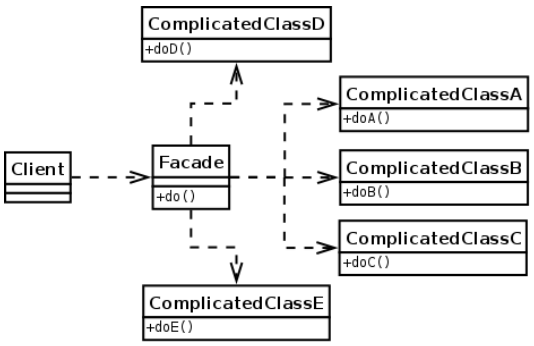

Figure 17: Facade Pattern

As guideline of our implementation, for all the complex operations that will require multiple methods calls from different classes we will use a façade pattern. In this way we will able to hide a complex logic operation within a single method call and to simplify the software maintenance for future changing needs.

3 Algorithm Design

3.1 Problem faced

A very delicate phase of our system is when the user input is checked. This is very important because the most user's operations in Travlendar+ terminates writing data on the Data Bases and if the system letted wrong input to be written in the persistent memory, big problems of inconsistency could be raised.

The operations that the user is allowed to do work all with similar data but the checks that have to be performed are different, sometimes very complex and also some of them require a check combination.

Moreover, giving a quick look on the rapid world evolution, it sounds quite possible that new kinds of events and operations could be implemented in the next future. Thus, the problem of finding a modular and scalable way to take on this possible software expansion assumes a primary importance.

3.2 Solution found

We will face this problem using the composite pattern, that has been described item by item in the design pattern paragraph. We give downward an example of how it will be implemented using Java.

3.3 Algorithm sample

3.3.1 EventsManager

```
/**
 * The EventManager is the class that manage all the operations concerning the
    events.
 * In this case is handled the add event operation.
 *
 * ActionParameter is the abstract class that all the check operation's
    parameters extend
 */
public class EventsManager {

    public String addEvent(ActionParameters eventInfo){
        CheckerInterface operationChecker = new AddEventChecker();
        RepositoryManager repositoryManager = new RepositoryManager();

        Message result = operationChecker.check(eventInfo);
        if (result.getTruthValue())
            return repositoryManager.addEvent(eventInfo);
        else
            return result.getMessage();
    }
}
```

Algorithm/mainServer/EventsManager.java

3.3.2 Checker Interface

```
/**
 * Inteface implemented by the composite check class and by the leafs
 */
public interface CheckerInterface{
     Message check(ActionParameters parameter);
}
```

Algorithm/checkers/CheckerInterface.java

3.3.3 AddEvent Checker

```
/**
 * Composite concrete class of the Composite pattern. It contains a list of the
   so called "leafs" of the composite pattern
 * and it calls the method check on all of them.
 */
public class AddEventChecker implements CheckerInterface {
    private ArrayList<CheckerInterface> myCheckers = new ArrayList<>();

    public AddEventChecker(){
        myCheckers.add(new CheckOverlapping());
        myCheckers.add(new CheckLunchGuaranteed());
        myCheckers.add(new CheckReachability());
    }

    @Override
    public Message check(ActionParameters eventInfo) {
        for ( CheckerInterface checker : myCheckers ) {
            Message result = checker.check(eventInfo);
            if (!result.getTruthValue())
                return result;
        }
        return new Message(true, "All ok");
    }
}
```

Algorithm/checkers/AddEventChecker.java

3.3.4 Check Overlapping

```
/**
 * Concrete class that implements the check interface. Its method check if the
   event in creation overlaps other events
 * already existing in the calendar
 */
public class CheckOverlapping implements CheckerInterface {

    private EventsManager manager = new EventsManager();

    @Override
    public Message check(ActionParameters eventInfo ) {
        EventInfo info = (EventInfo)eventInfo;
        if ( startingTimeInAnotherEvent(eventInfo)
                && manager.getStartingTimeNextEvent(eventInfo) <= info.
    getEndingTime()  )
            return new Message(false, "The event overlaps the next one!");

        return new Message(true, "all ok");
    }
}
```

Algorithm/checkers/subCheckers/CheckOverlapping.java

3.3.5 Check Lunch Guaranteed

```
/**
 * Concrete class that implements the check interface. Its method check if the
   Lunch is guaranteed
 */

public class CheckLunchGuaranteed implements CheckerInterface {

    private EventsManager manager = new EventsManager();

    @Override
    public Message check(ActionParameters eventInfo ) {
        EventInfo info = (EventInfo)eventInfo;

        if ( !checkLunchPreferenceRespected(info))
            return new Message(false, "The new event do not respect the lunch
    constraits!");

        return new Message(true, "all ok");
    }
}
```

Algorithm/checkers/subCheckers/CheckLunchGuaranteed.java

3.3.6 Check Reachability

```java
/**
 * Concrete class that implements the check interface. Its method check if the
   event is reachable from the previous
 * and the next one.
 */
public class CheckReachability implements CheckerInterface {

    @Override
    public Message check(ActionParameters eventInfo) {

        if (  isReachble(eventInfo) )
            return new Message(true, "check ok");

        return new Message(false, "Event not reachble on time!");
    }

}
```

Algorithm/checkers/subCheckers/CheckReachability.java

4 User Interface Design

In this section we will some mockups in order to describe Travlendar+ user navigation.

4.1 Login & User Profile

The login interface is composed by email and password fields. After the insertion of the credentials the user will be able to access to his calendar and personal page.

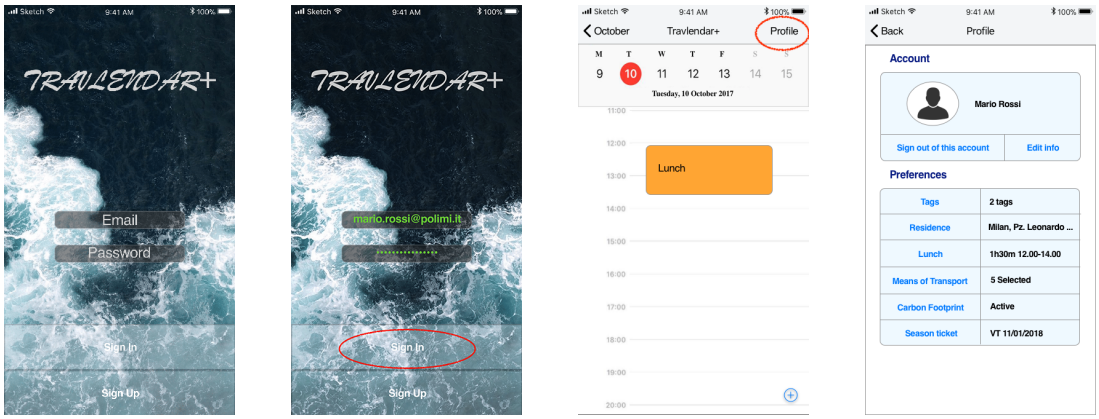

Figure 18: Login Sketch

4.2 Calendar

By tapping on the add button on the bottom-right side of the screen it'll be possible to plan a trip or add an event.

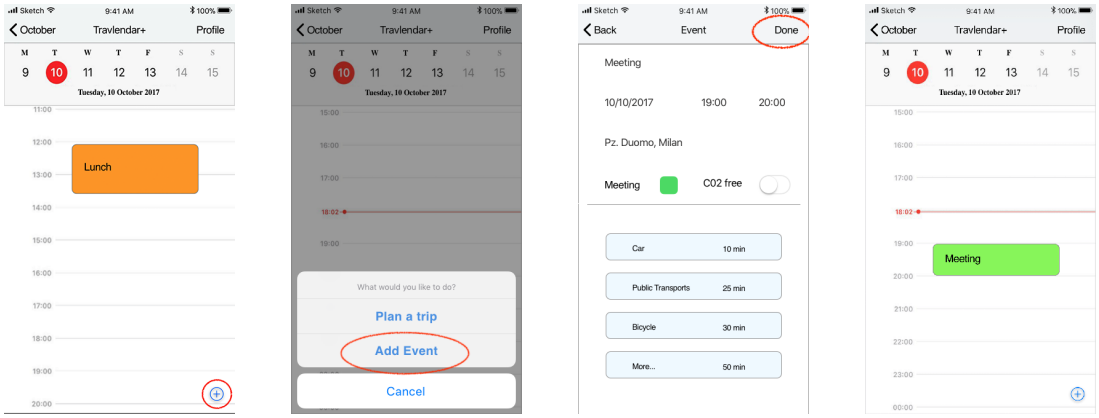

Figure 19: Calendar Sketch

4.3 Lunch

An everyday lunch event is created by default at the beginning of the time window indicated by the user. During the creation of an event, the system will move the lunch break and/or shrink its duration if needed.

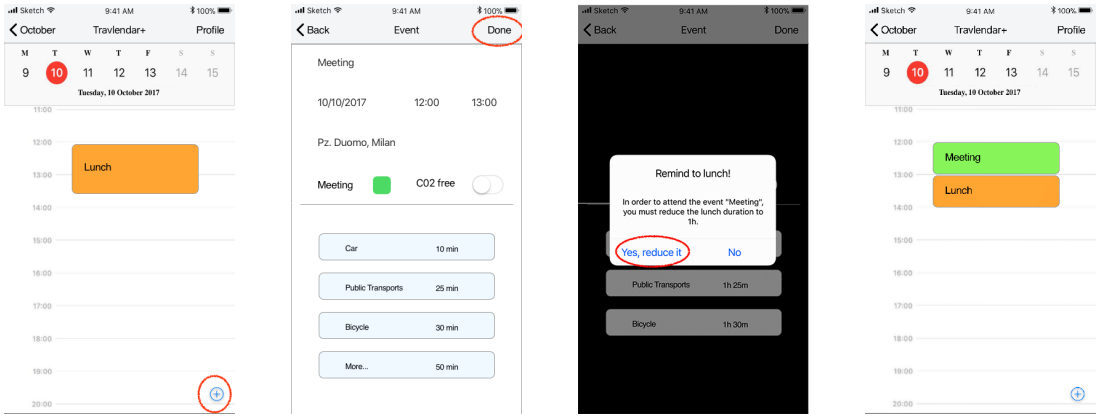

Figure 20: Lunch Sketch

4.4 CO_2 free

It can be enabled or disabled as show below.

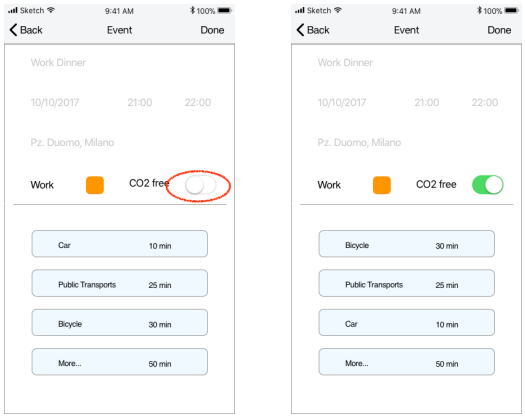

Figure 21: CO_2 Sketch

4.5 Weather Forecast

A blue arrow will connect two reachable events. If the arrow is yellow, it means that the weather forecast is not optimal.

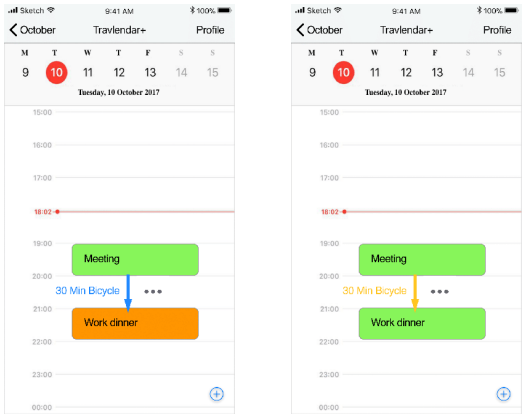

Figure 22: Weather Forecast Sketch

4.6 Directions

By tapping on the event its details will be shown. By tapping on one item of the means list (the bottom side of picture 2) reported below, the directions (picture 3) will be shown.

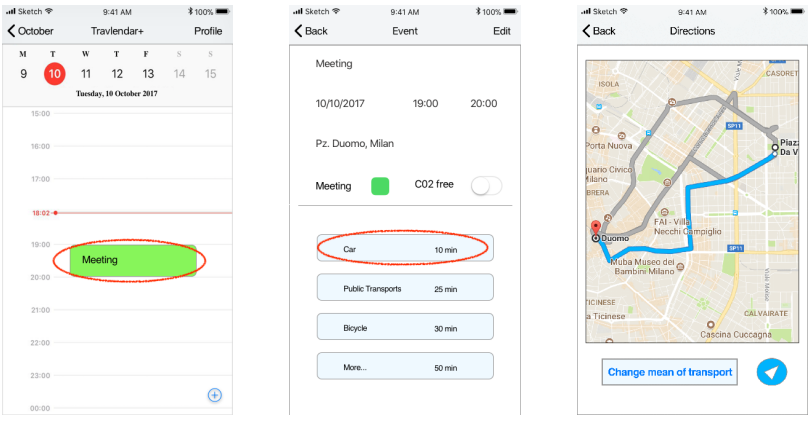

Figure 23: Directions Sketch

4.7 Tags

In this section the user can create tags by tapping on the add button set in the centre-bottom side of the UI. The user is asked to assign a colour and a name to the tag.

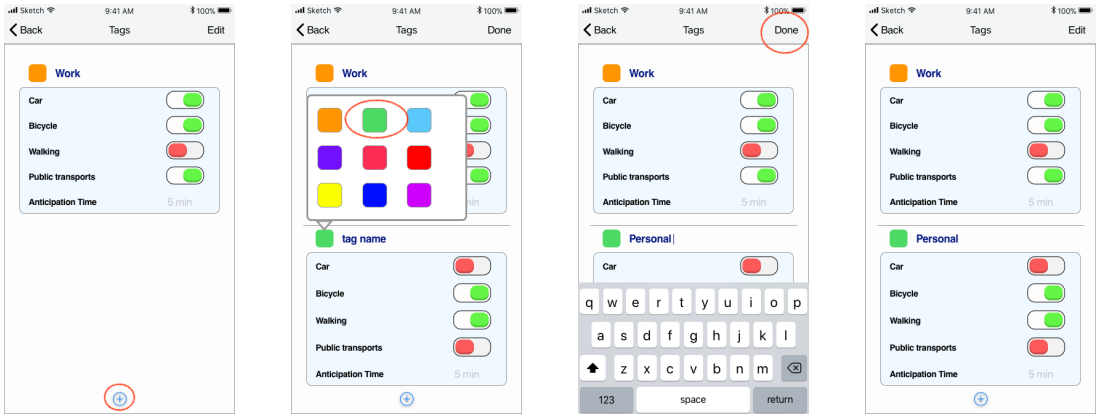

Figure 24: Add Tag Sketch

The user can delete one or more tag pressing on the "edit" and select the item he wants to delete.

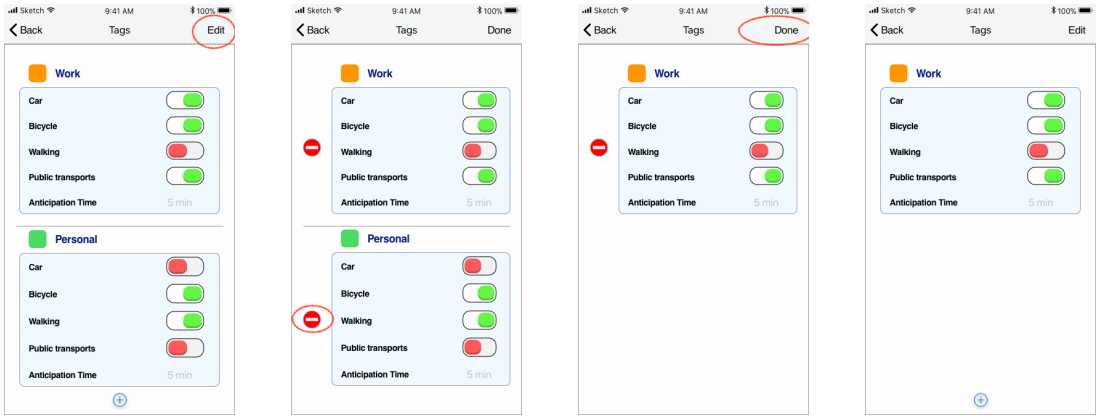

Figure 25: Delete Tag Sketch

4.8 Trips

Trips will be suggested if the event is in another town.

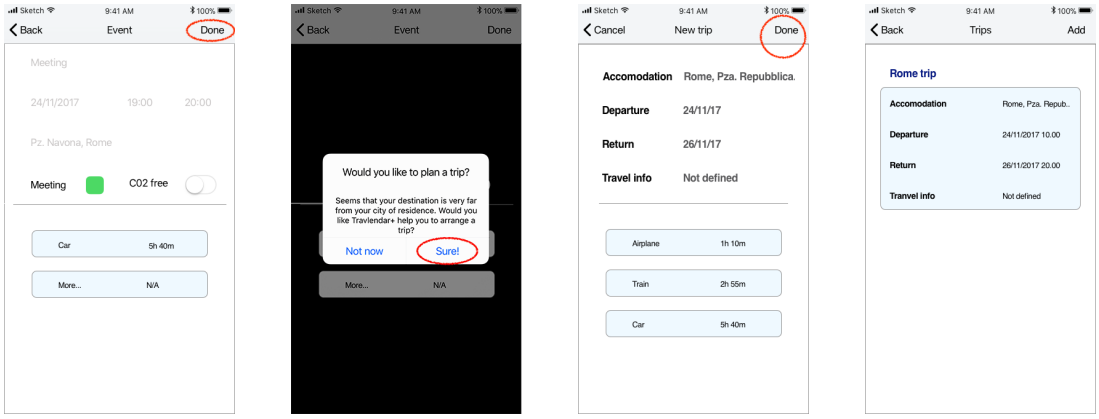

Figure 26: Add Trip Sketch

After the creation of a trip, it will be shown also in the profile section.

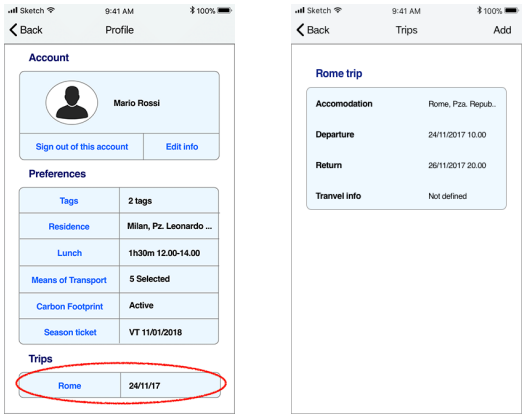

Figure 27: Visualize Trip Sketch

The user can insert ticket info by tapping one of the means suggested. By inserting the data, the system will provide some tickets available for the journey. The user will be able to add an existing ticket if he has already bought one (pressing "add").

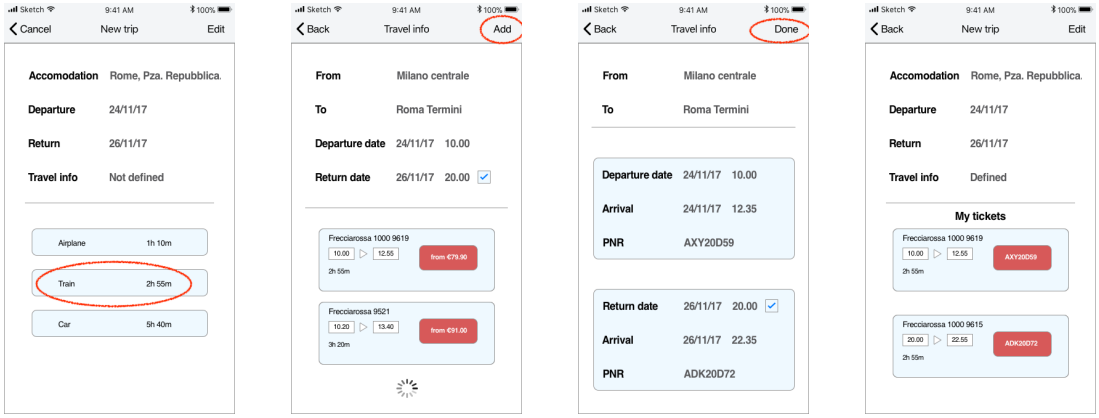

Figure 28: Add Own Tickets Sketch

It will be also possible to buy a suggested ticket after being redirected to the service website. If the service does not provide API a manual insertion of the ticket id is required (picture 4 and 5 below).

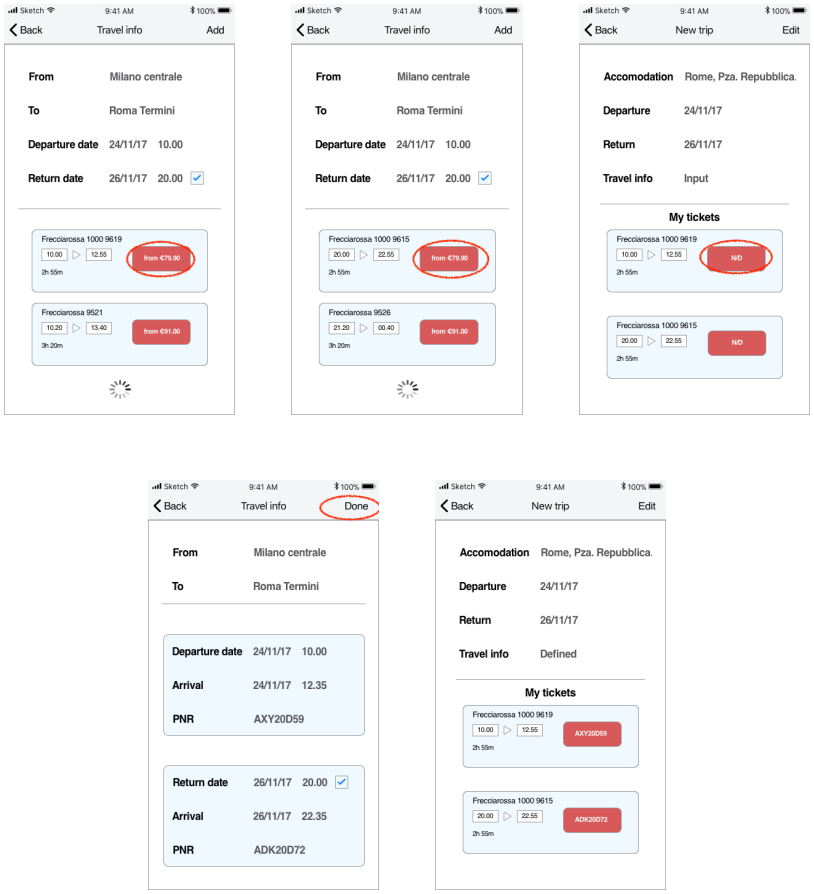

Figure 29: Buy Tickets Sketch

5 Requirement Traceability

Tis Section explain how the requirements we have defined in the RASD map to the design elements that we have defined in this document.

- *[G_1] : Allow the user to register and log into the system*

Requirement	Componets	Methods
[R_1] : The System has to check the credentials.	- Registration & Login Manager	.checkEmail() .checkPassword()
[R_2] : A registered user must be able to log in the system.	- User Manager - Registration & Login Manager	.login() .login()
[R_3] : The System has to handle the connection and user requests.	- User Manager	(all user manager methods)

- *[G_2] : Allow the user to add events in the calendar*

Requirement	Componets	Methods
[R_1] : The system mustn't allow events overlapping.	- Events Manager	.checkOverlap()
[R_2] : The system has to guarantee the minimum lunch duration.	- Events Manager	.checkMinLunchDuration()
[R_3] : The system has to give the possibility to the user of arriving on time to the new event.	- Transport Manager - Directions - Events Manager	.getRouteTime() .getRouteTime() .checkReachability()
[R_4] : The system has to be able to add the new event to the calendar.	- Events Manager	.addEvent()

- *[G_3] : Allow the user to receive the mobility options*

Requirement	Componets	Methods
[R_1] : The system has to check that the event has been created.	- Repository Manager - EventsManager	.getEvent() .checkEvent()
[R_2] : The system is able to communicate with the route provider.	- Directions	(all Directions methods)
[R_3] : The system sends data about the starting and ending location to the route provider.	- Directions	.getDirections(locations)
[R_4] : The system receives the path and the available transports from the external service.	- Directions	.getMeans() .getDirections()
[R_5] : The system has to use the information given in the event's TAG to filter the possible means of transport.	- Transport Manager	.filterMeans(TAG)
[R_6] : The system has to check the travel means enabled by the user.	- Preference Manager	.getAvailableMeans()
[R_7] : The system has to obey the constraints set by the user about the travel means.	- Preference Manager	.getMeansConstraint()
[R_8] : The system has to check if the Carbon footprint preference has been enabled.	- Repository Manager - Events Manager - Preference Manager	.getEventInfo() .getCO2() .getCO2()
[R_9] : The system has to retrieve the user's position.	- User Manager	.getUserLocation()
[R_{10}] : The system has to check the weather forecast.	- Environment	.checkWeather()

Requirement	Componets	Methods
[R_{11}] : The system has to retrieve the appointment location.	- Repository Manager	.getEventInfo()
[R_{12}] : The system has to know how to show the solution found.	- User Manager - Events Manager	.visualizeEvent() .visualizeEvent()

- *[G_4] : Support the user to avoid getting late on appointment*

Requirement	Componets	Methods
[R_1] : The system has to retrieve the user's GPS position.	- User Manager	.getUserLocation()
[R_2] : The system has to know how to send warnings to the user.	- User Manager	.warning()
[R_3] : The system has to check the destination position.	- Repository Manager	.getEventInfo()
[R_4] : The system has to check the possibility of reaching on time the next event with the slowest mean of transport in the list of suggested ones.	- Repository Manager - Directions - Transport Manager - Directions	.getNextEvent() .getMeans() .filterMeans() .getRouteTime()

- *[G_5] : Allow the user to have advices about the means of transport that can minimize his carbon footprint*

Requirement	Componets	Methods
[R_1] : The system must be able to check the carbon footprint preference.	- Repository Manager - Events Manager - Preference Manager	.getEventInfo() .getCO2() .getCO2()
[R_2] : The system must know how to filter the means of transport to minimize their carbon footprint.	- Transport Manager	.filterMeans(TAG)

- *[G_6] : Support the user to have at least 30 minutes of lunch every day*

Requirement	Componets	Methods
[R_1] : The system must be able to check the lunch preferences.	- Preference Manager	.getLunchPref()
[R_2] : The system must be able to create every day a lunch based on the user's preferences.	- Preference Manager - Events Manager	.getLunchPref() .addLunchEvent()
[R_3] : The system must be able to avoid the creation of those events that prevent the presence of a lunch with the minimum duration.	- Events Manager	.checkOverlap() .checkMinLunchDuration()

- *[G_7] : Allow the user to buy local transport ticket*

Requirement	Componets	Methods
[R_1] : The system has to check that all the necessary travel info has been inserted by the user.	- Events Manager	.checkEventInfo
[R_2] : The system has to address the user to the web site/application to complete the ticket purchase.	- Local Transport Tickets	.showWebSite()

- *[G_8] : Give advices about the best transportation ticket to buy*

Requirement	Componets	Methods
[R_1] : The system has to know how long the user will stay in town.	- Repository Manager	.getTripInfo()
[R_2] : The system has to be able to get information about tickets from the public transport service.	- Local Transport Tickets	.getTickets()
[R_3] : The system has to be able to interpret and elaborate the information received.	- Local Transport Tickets	.parseInfo()
[R_4] : The system has to find the most convenient ticket.	- Local Transport Tickets	.getBestTicket()
[R_5] : The system has to be able to show the results.	- User Manager	.visualizeLocalTickets()

- *[G_9] : Remind the user about the expiry date of his season ticket, if he has inserted one*

Requirement	Componets	Methods
[R_1] : The system has to check the season ticket expiry date.	- Preference Manager	.getSeasonTicket() .checkExpiryDate
[R_2] : The system has to be able to notify the user that his season ticket is expiring.	- User Manager	.visualizePreferences()

- *[G_{10}] : Allow the user to buy ticket for outdoor travels*

Requirement	Componets	Methods
[R_1] : The system must check the data inserted by the user to buy the ticket.	- Trip Manager	.checkTrip()
[R_2] : The system must be able to send the data to the external transport service.	- Outdoor Transport Tickets	.getTickets(trip)
[R_3] : The system has to be able to interpret and elaborate the information received.	- Outdoor Transport Tickets	.parseInfo()

- *[G_{11}] : Allow the user to use local sharing services*

Requirement	Componets	Methods
[R_1] : The system has to retrieve the sharing means location.	- Sharing Service Manager	.getMeansLocation()
[R_2] : The system has to check the user's location.	- User Manager	.getUserLocation()
[R_3] : The system has to be able to show the sharing mean position to the user.	- User Manager	.visualizeSharingMean()
[R_4] : The system has to be able to redirect the user to the external sharing service system for booking the mean.	- Sharing Service Manager	.bookMean()

- *[G_{12}] : Allow the user to set some preferences in the settings section*

Requirement	Componets	Methods
[R_1] : The system must be able to show the user the possible preferences.	- User Manager	.visualizePreferences()
[R_2] : The system has to register these preferences in its database.	- Preference Manager - Repository Manager	.registerPreferences() .registerPreferences()
[R_3] : The system has to have access to these preferences each time is needed.	- Preference Manager - Repository Manager	.getPreferences() .getPreferences()

- *[G_{13}] : Allow the user to handle his trips*

Requirement	Componets	Methods
[R_1] : The system has to be able to create travel events.	- Events Manager	.addTravelEvent()
[R_2] : The system has to be able to show the list of trips created.	- Trip Manager - Repository Manager	.getTrips() .getTrips()
[R_3] : The system has to be able to let the user modify the trips when needed.	- Trip Manager	.editTrip()
[R_4] : The system has to be able to check the trips data.	- Repository Manager - Trip Manager	.getTrip() .checkTrip()

6 Implementation, Integration and Test Plan

In this section we will show the plan we have projected for implementing Travlendar+. Firstly we will describe the strategy for the processes chosen for taking the project and the structure of our team, how it is divided and the tasks that each part has to accomplish. After we will state how each team's part has to interact with the others and the span of time of these interactions. Finally we will supply a list of possible risks, the probability of their presence, their impact and a possible strategy to deal with them.

6.1 Strategy adopted

To give a quality assurance of the project and to be sure that the final product will be as our stakeholders expect, we will follow an Agile planning process. It consists in a first initiating part in which it's given an overall plan of the system process. We have started to build the plan with the information given in the RASD and we will terminate it in this document: in fact in the previous sections we have described the software architecture and the design patterns that will be used and in this section we will give a schedule for all the various team tasks.

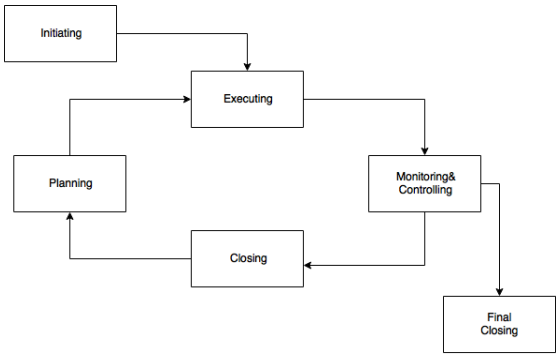

Figure 30: Agile Planning Strategy

After this part, there will be a cycle of phases called respectively: Executing, monitoring&Controlling, Closing and Planning. Every cycle round has as input a specific process to accomplish, that is divided between the various team's parts. After the execution of all the tasks in a precise given timeframe, the work is checked by all the parts, customers included. When an agreemnt is reached, it will follow another planning phase that will contain also the corrections that will result after the agreement. This cycle is repeated until the end of the project, when all the functionalities stated in the RASD will be covered.

6.2 Team structure

Our team will be composed by five main parts :

- *Developers :* people devoted to implement the software.
- *Testers :* people devoted to test the software functionalities.
- *Front-End Team :* developers appointed to implement the system's front end.
- *Back-End Team :* developers appointed to implement the system application logic.
- *Supervisors :* developers that belongs to the Back-End Team or to the Front-End Team and that have the main task of having any time the picture of all the their team work.

6.3 Implementation and testing plan

The work will be divided in sub-milestone from a minimum of one to a maximum of two weeks, and main-milestone every three or maximum four weeks. In these span of time all the teams have to complete the tasks that have been established in the previous planning phase. Every two days the different team supervisors and a Tester's team delegate will organize a meeting to discuss about their team progress and they will give directives to proced on basically at the same speed. In the meanwhile, the Tester team will work on the *white-box* units test for the functionalities that the developer's teams are implementing. When the sub-milestone day come, the Tester team will have from a minimum of one day to a maximum of three days for testing the functionalities and give the results to the development team that will have to fix the eventual bugs.

During the week before the the milestone day, the Tester team has to perform an *integration test* of the components and give the result to the development teams which have to release a beta version to restrict group of selected people for the milestone day. In this way, it will be performed also an incremental *User acceptance test* on the work that has been performed. Furthemore, it will take place a meeting with the customers in which their needs and doubts will be discussed and taken into account in the next planning phase.

The precise duration of the timeframe will be established dynamically taking into account the minimum and maximum span on time mentioned above considering that the project must be completed in no more of 16 weeks.

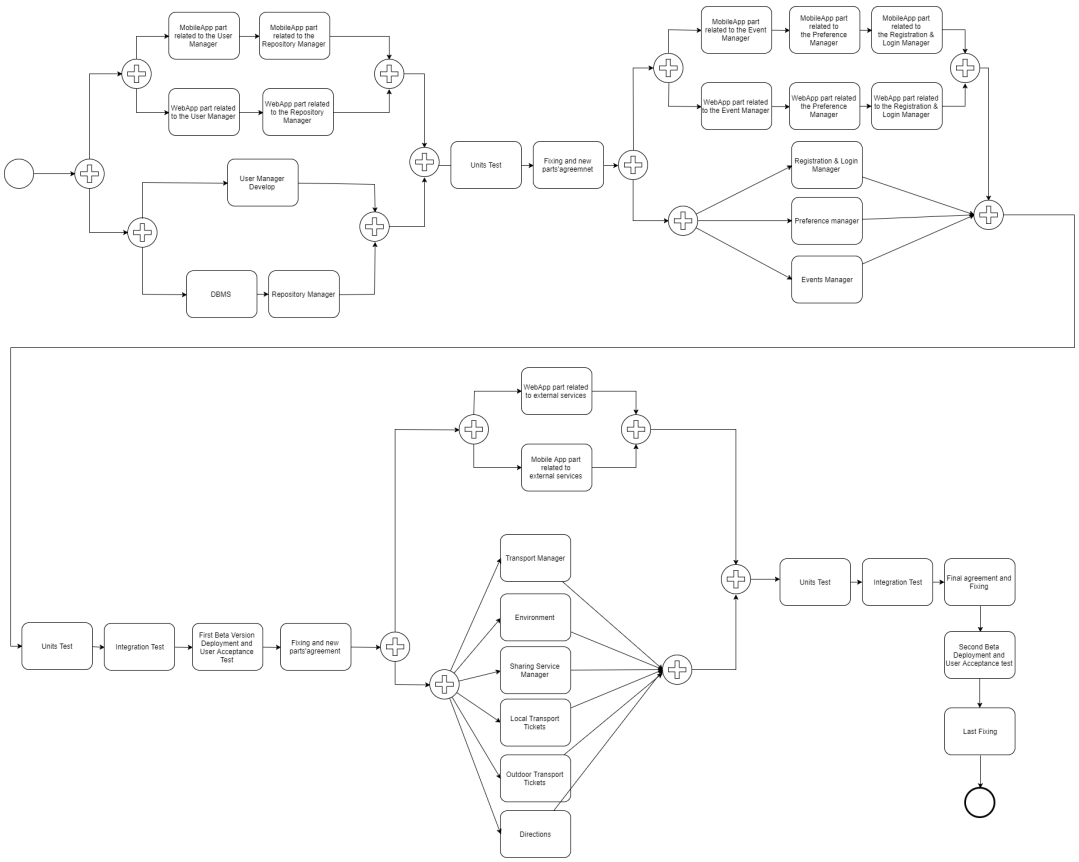

Figure 31: Implementation Flow

6.4 Possible risks

Risk	Probability	Effects
Budget Problems	Low	Catastrophic
Time for Implementing a Function Exceed	Medium	Serious
Team's Members not Available in a Critic Moments	Medium	Catastrophic
Impossibility to Recruit Staff with Required Skills	Medium	Serious

6.5 Possible solutions

- *Budget problems :* specify before the beginning all the functionalities, realize prevision of the costs and agree with the customers a margin of budget that could increase due to unpredicted situation during the development.
- *Time for implementing a function exceed :* agree with the stakeholder a span of time of delay that they can accept for unpredicted problems.
- *Team's member not available in critic moment :* being sure that at least one of the supervisors could substitute a team member in case of necessity
- *Impossibility to recruit staff with required skills :* advices the customers about possible delays and to be ready with a table of COTS to buy in case of needs with the relative costs.

7 Revision History

Version	Date	Description	Notes
1.0	26-11-2017	Final Draft	-

8 Effort Spent

8.1 Kostandin Caushi

Date	Hours
13.11	2h
14.11	2h
16.11	4h
18.11	5h30
19.11	4h
21.11	6h
22.11	5h
23.11	6h
24.11	7h30
25.11	5h30
26.11	8h

8.2 Marcello Bertolini

Date	Hours
13.11	2h
14.11	2h
16.11	2h
17.11	3h
18.11	5h
19.11	3h
20.11	3h
22.11	4h
23.11	3h
24.11	4h
25.11	3h

8.3 Raffaele Bongo

Date	Hours
13.11	2h
14.11	2h
16.11	2h
17.11	2h
18.11	5h
19.11	5h
20.11	7h
21.11	5h
22.11	4h
23.11	6h
24.11	2h
25.11	3h30
26.11	3h30