POLITECNICO
MILANO 1863

Travlendar+

D E S I G N D O C U M E N T

Alessio CANTINA	*895395*
Simone CRIPPA	*898304*
Nicola CUCCHI	*893748*

Version 1.0 – November 25th

1. INTRODUCTION

A. Purpose

The aim of this document is to provide a functional description of Travlendar+ project, following the presentation outlined in the *Requirement Analysis and specification document*, briefly called RASD. We try from now on to give a comprehensive look on the architecture of the system, seen as all the components which rely on it. As a consequence, this document addresses professionals, developers in primary, who will use it as a solid guide for their implementation work that will follow.
Another section is also dedicated to designers, who will work on user interfaces of the application on the devices.

B. Scope

Since we will follow the system description outlined in the RASD, it is appropriate to refer that document as a general description of it, which can be taken for granted as familiar from now on.
In this document we will detail the overall design of the system and its architecture, showing how components interact one with each other to accomplish a specific task.
An analysis of the principal algorithms that govern the procedures of Travlendar+ will be provided to help developers in their work. Finally, a section will be dedicated to an overview of the user interfaces design, in order to complete the main specification needed.

Our aim is to provide a document which is the natural follow-up of the analysis started in the RASD, trying to achieve a solid cross-referencing between the presented documents, for a better coherence and consistency.

C. Definitions, Acronyms, Abbreviations

Definitions, Acronyms and Abbreviations already included in the RASD document won't be repeated here, only the new terms provided in this document will be described in this section:

- *Software Architecture*: The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.
- *Component*: independent software element of the system which encapsulates a set of related functions that respects the constraints imposed by a component model.
- *Component model*: combination of standard governing like how to build individual components, organize them to build an application and how components can communicate and interact among each other
- *Inspection*: a formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems
- *Java Servlet:* objects in Java operating in a server or application web, enabling the creation of Web Application.

And we will use also these acronyms through this document:

- *EJB*: Enterprise Java Beans
- *JSP:* Java Server Pages
- *JSF*: Java Server Faces
- *Java EE:* Java Enterprise Edition
- *JPA*: Java Persistence API
- *EIS*: Executive Information System
- *API*: Application programming interface

D. Revision History

No revision had been done up to now.

E. Reference Documents

Here there is the list of documents from where some data, terms, etc is recovered from:

- Bass, Clements, and Kazman, Software Architecture in Practice, SEI Series in Software Engineering. Addison-Wesley, 2003.
- https://it.wikipedia.org/wiki/Design_Patterns
- Requirements Analysis and Specification Document v1.1

F. Document Structure

This document develops as follows:

- Chapter 2 analyses the architecture design of the system, at different levels of abstractions and from various points of view, where some UML diagrams will be given as a support.
- Chapter 3 defines the most relevant algorithms on which the system relies using the "Pseudo-code" language or a programming language, trying to abstract from the hard-coded details of the programming language that will be used later on this project.
- Chapter 4 will offer an overview on how the user interfaces of the system will look like, increasing the level of detail of the provided mock-ups, trying to be coherent with the sketches already inserted in the RASD but leaving all more specific details to the designers.
- Chapter 5 will focus on how the requirements we have defined in the RASD map to the design elements that will be defined in this document.
- Chapter 6 is an explanation of the order in which we plan to implement the subcomponents of our system and the order in which we plan to integrate them into components and test the integration.

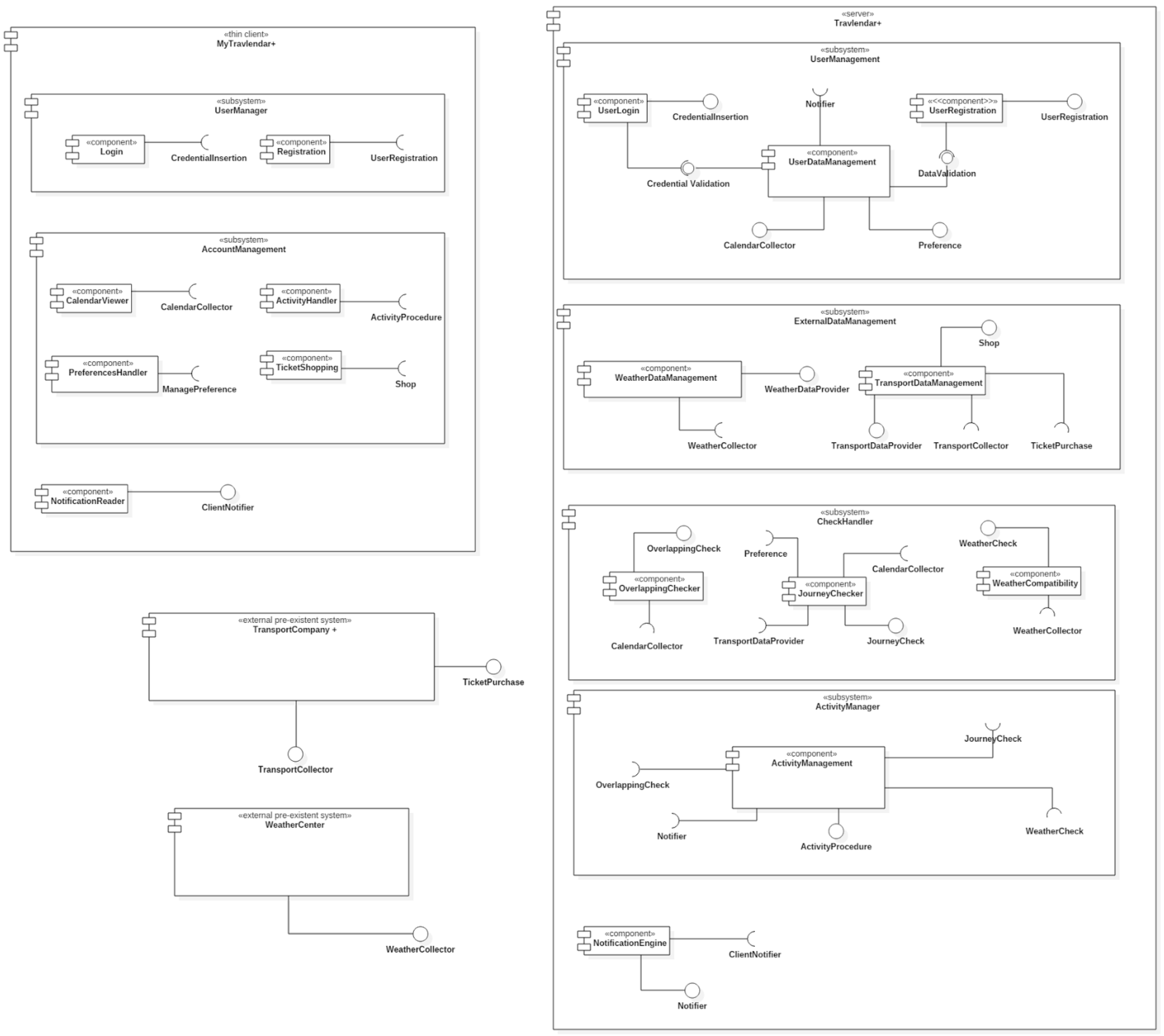
2. ARCHITECTURAL DESIGN

A. Overview

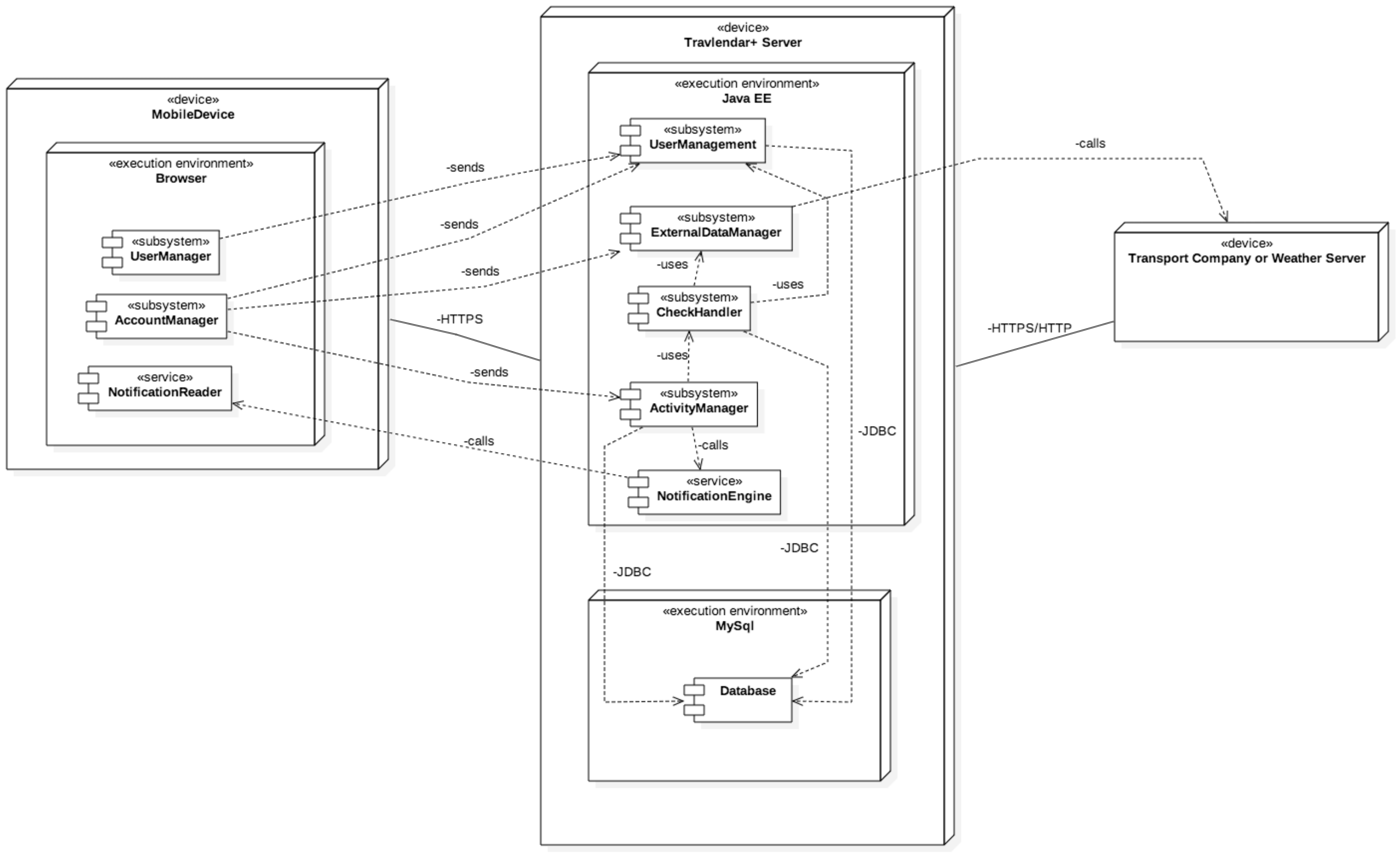
Chapter 2 is the most important of this document since all the different aspects of the architecture of Travlendar+ will be analysed at different level of detail and abstraction, from various view-points.
In section 2.B we give a high level description of the system structure, analyzing internally single components and their interaction with the other components in an abstract way.
In section 2.C we provide an analysis of the topology of a system's hardware, built as part of architectural specification with the aim of specifying the distribution of the components and identifying the most critical performance bottlenecks.
In section 2.D we outline a dynamic behavior, with a low level of details, focusing on the most important aspects such as modeling the flow of control, illustrating typical scenario and analysing the whole architecture.
In section 2.E we give a general idea of the relationships and connections between components, showing in more detail the interfaces that each component shows to the outside of its edge.
In section 2.F a list of architectural selected styles will be provided with a short analysis on particular architectural patterns used during the creation of this document.
Finally, in section 2.G other design decision will be described.

A very important issue of this section is to effectively analyse the possible solutions to be used in the implementation of our project, highlighting for each solution its pros and cons, achieving in a precise way the best option according to requirements, functionalities and goals of our system.

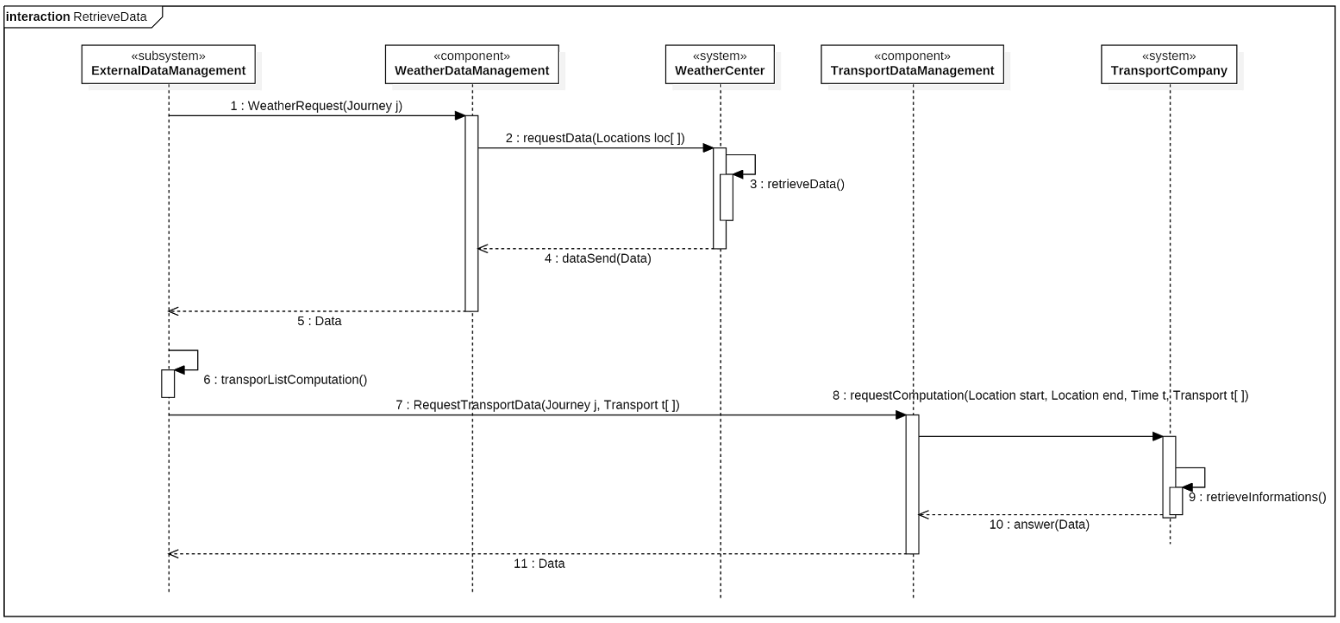
B. Component view



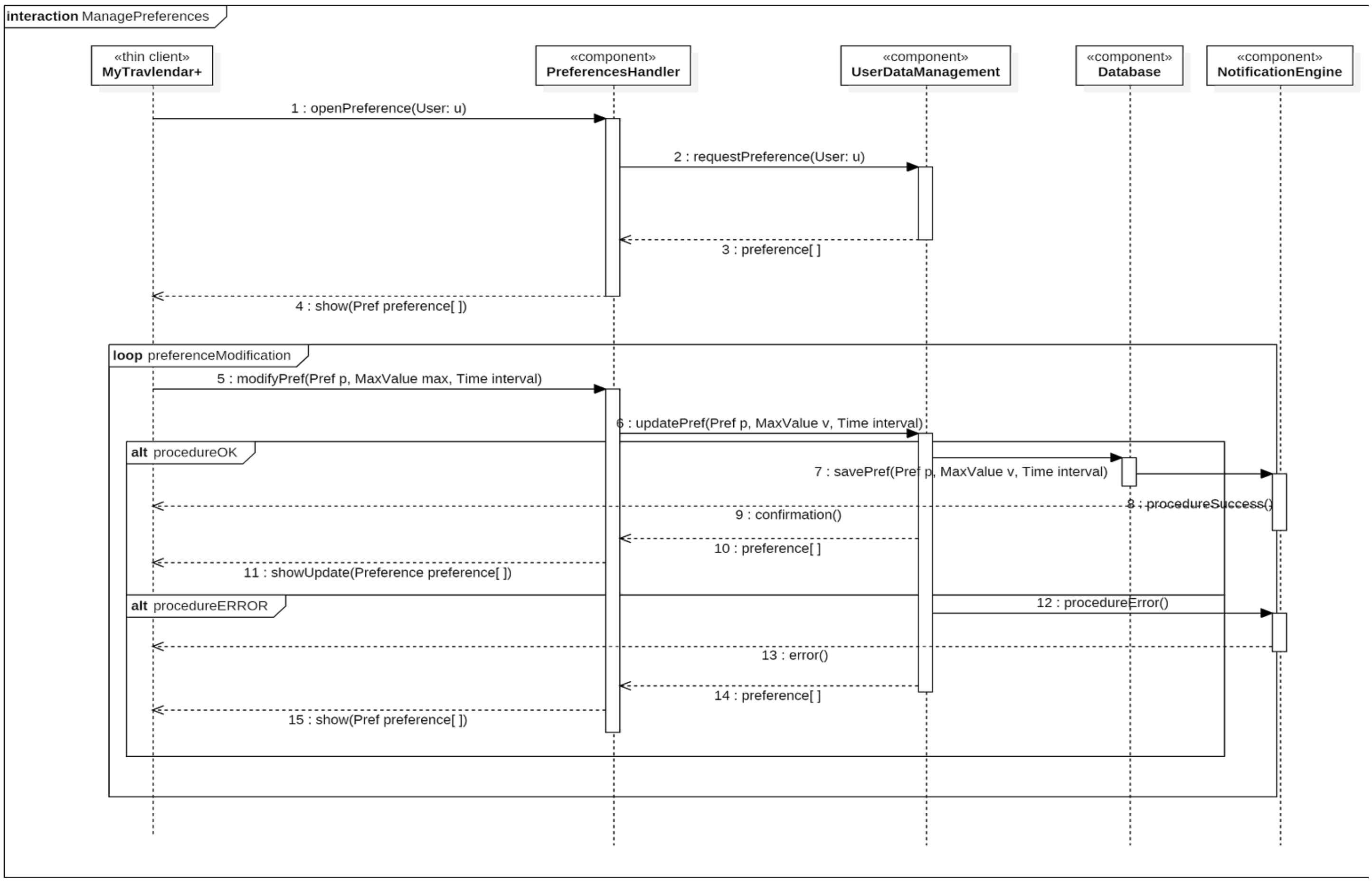
C. Deployment view



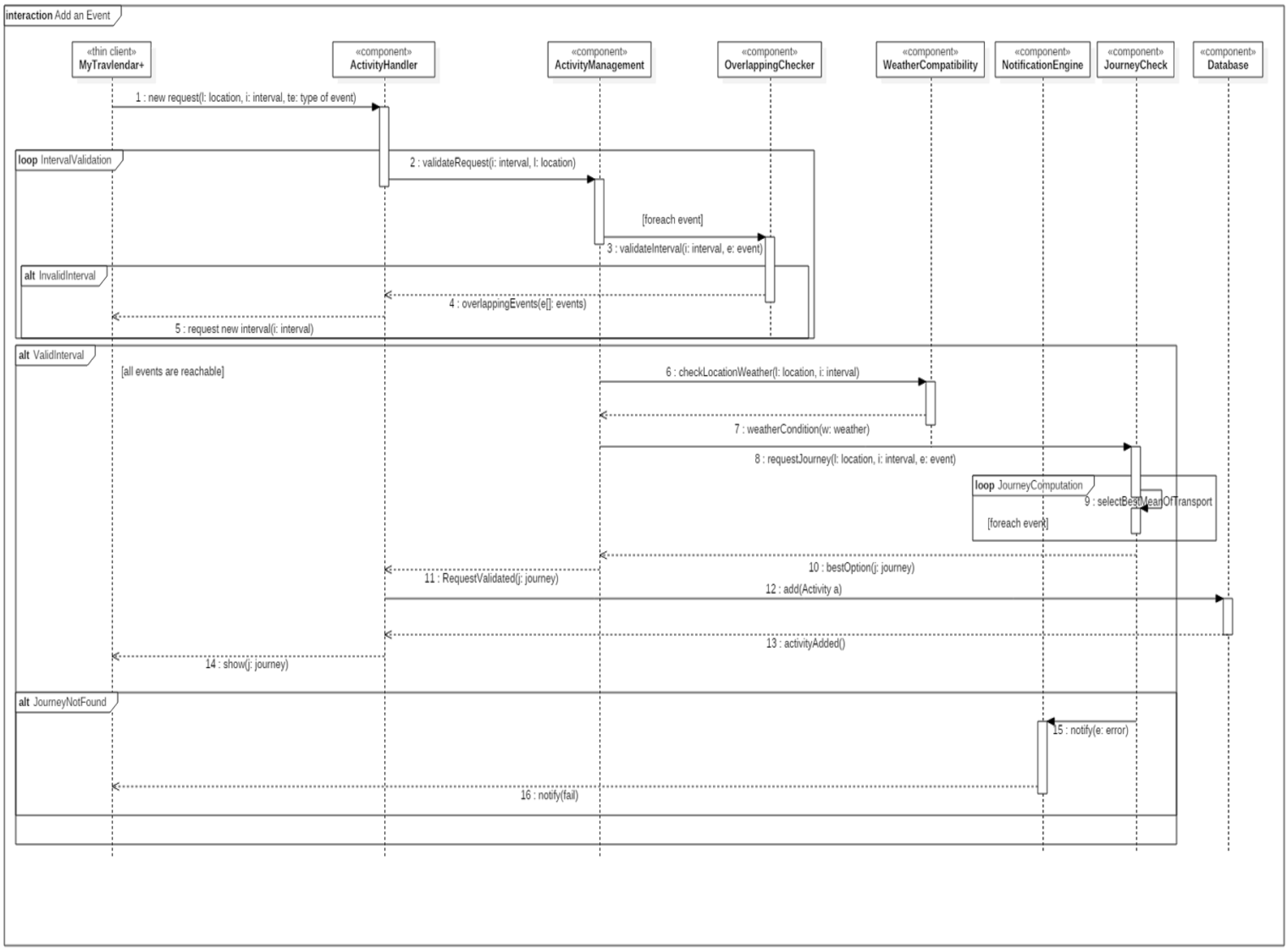
D. Runtime view



interaction RetrieveData
«subsystem» ExternalDataManagement
«component» WeatherDataManagement
«system» WeatherCenter
«component» TransportDataManagement
«system» TransportCompany
1 : WeatherRequest(Journey j)
2 : requestData(Locations loc[ ])
3 : retrieveData()
4 : dataSend(Data)
5 : Data
6 : transporListComputation()
7 : RequestTransportData(Journey j, Transport t[ ])
8 : requestComputation(Location start, Location end, Time t, Transport t[ ])
9 : retrieveInformations()
10 : answer(Data)
11 : Data



interaction ManagePreferences
«thin client» MyTravlendar+
«component» PreferencesHandler
«component» UserDataManagement
«component» Database
«component» NotificationEngine
1 : openPreference(User: u)
2 : requestPreference(User: u)
3 : preference[ ]
4 : show(Pref preference[ ])
loop preferenceModification
5 : modifyPref(Pref p, MaxValue max, Time interval)
6 : updatePref(Pref p, MaxValue v, Time interval)
alt procedureOK
7 : savePref(Pref p, MaxValue v, Time interval)
8 : procedureSuccess()
9 : confirmation()
10 : preference[ ]
11 : showUpdate(Preference preference[ ])
alt procedureERROR
12 : procedureError()
13 : error()
14 : preference[ ]
15 : show(Pref preference[ ])


interaction Add an Event
«thin client» MyTravlendar+
«component» ActivityHandler
«component» ActivityManagement
«component» OverlappingChecker
«component» WeatherCompatibility
«component» NotificationEngine
«component» JourneyCheck
«component» Database
1 : new request(l: location, i: interval, te: type of event)
loop IntervalValidation
2 : validateRequest(i: interval, l: location)
[foreach event]
3 : validateInterval(i: interval, e: event)
alt InvalidInterval
4 : overlappingEvents(e[]: events)
5 : request new interval(i: interval)
alt ValidInterval
[all events are reachable]
6 : checkLocationWeather(l: location, i: interval)
7 : weatherCondition(w: weather)
8 : requestJourney(l: location, i: interval, e: event)
loop JourneyComputation
9 : selectBestMeanOfTransport
[foreach event]
10 : bestOption(j: journey)
11 : RequestValidated(j: journey)
12 : add(Activity a)
13 : activityAdded()
14 : show(j: journey)
alt JourneyNotFound
15 : notify(e: error)
16 : notify(fail)

E. Component interfaces

In the following table we analyse the most significant interfaces between components, referring to section 2.B, where components are presented along with the interface they offer with a brief description of its aims.

UserManagement		
UserDataManagement	*CredentialValidation*	Provides method for checking if the credentials given by the user are valid or not
	CalendarCollector	Provides methods to access the events data of a user calendar
	Preference	Provides methods to access a user preferences and edit them
	DataValidation	Provides methods to check if the information a user has provided to register are valid or not
UserLogin	*CredentialInsertion*	Offers the methods to catch the user credentials from the GUI
UserRegistration	*UserRegistration*	Offers the methods to catch the user data from the GUI needed for the insertion of a new user in the database
ExternalDataManagement		
WeatherDataManagement	*WeatherDataProvider*	Provides the methods to access the latest weather forecasts
TransportDataManagement	*TransportDataProvider*	Provides the methods to access the latest transport data
	Shop	Provides methods useful to process an e-commerce procedure
CheckHandler		
JourneyChecker	*JourneyCheck*	Provides methods able to verify the effective feasibility of a journey, according to user preferences
OverlappingChecker	*OverlappingCheck*	Checks if an event overlaps with an already existing ones
WeatherCompatibility	*WeatherCheck*	Checks if the selected means of transport for a journey are suitable with the weather conditions

ActivityManager		
ActivityManagement	*ActivityProcedure*	Offers methods to compute the process of the addition of an activity
NotificationEngine	*Notifier*	Offers methods to compute messages of various kind
NotificationReader	*ClientNotifier*	Offers to the server an interface to provide warnings, messages etc

External Systems		
TransportCompany	*TicketPurchase*	Provides the methods to purchase public transport tickets
	TransportCollector	Provides the methods to access data like timetables and warnings of a transport company, to query the internal database of the transport company
Weather Center	*WeatherCollector*	If offers methods to make requests and queries on the weather center system, in order to provide weather data.

F. Selected architectural styles and patterns

1. Architectural Styles

Our project will focus on a **client-server architecture**, as we described in the Component View, this solution is the most used architecture and it perfectly fits what is required for the implementation of the various functionalities.
The **thin client** will only send requests to the application server, the most important subsystem of the entire project since it encapsules the main functionalities, computes solutions, responses or retrieves requested data and send them all back to the user.
Secondly, we focused on the type of client, where we were able to choose between application client and web client. In this case we found more appropriate to choose the **web client**, since we evaluated pros and cons of each solution:

TYPE OF CLIENT	**PROS**	**CONS**
Web Client	cross platform	more difficult to implement notifications alert
	easier to maintain (no need to updates and installations on every client)	all the computation must be done by the server (server more powerful required)
	lightweight	
	more precise client-server distinction (computationally speaking)	
	multiple device synchronization easier	
Application Client	computation also available on the devices	one implementation for each platform
	more powerful and complex implementation	no precise distinction between client and server tasks
		multiple device synchronization more complex
		application will be heavier

2. Design Patterns

The analysis produced in this section will be strongly related to what is reported in "Other design decisions" section that follows since we will refer to components related on different layers.
During the implementation of the project we suggest the adoption of these kinds of pattern:

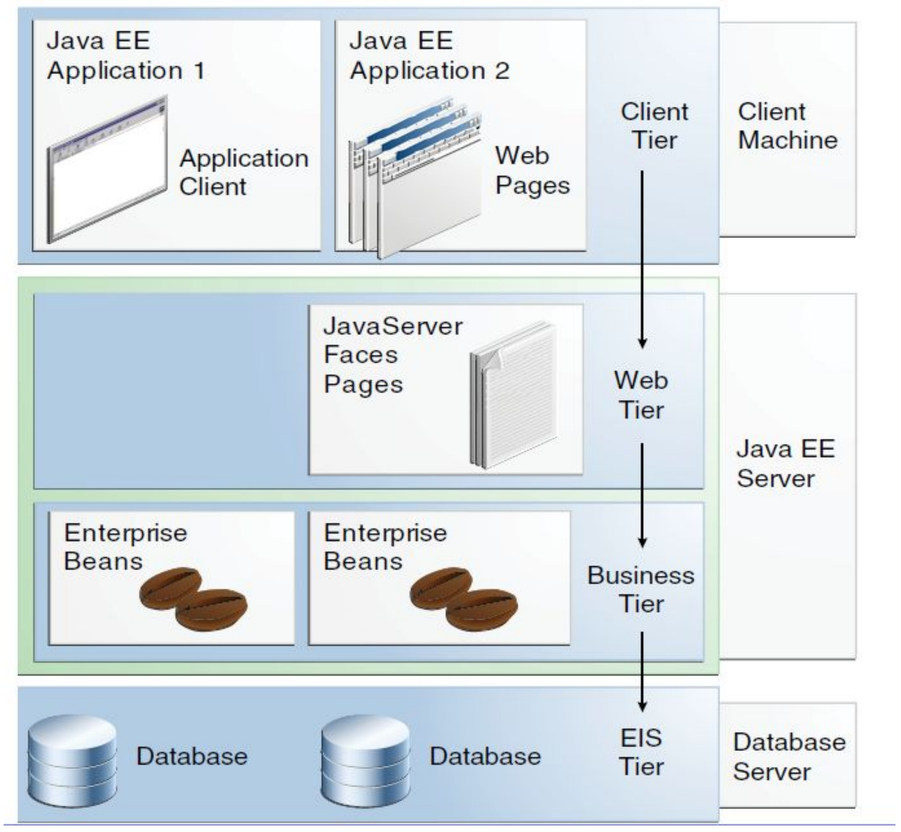
- **MVC Pattern**: concerning the whole structure of the system, trying to have a distinction between the logic, inserted in the Controller, the view through the browser and defined by the user, composing the View and finally the Model which consists of all the parts related to the database;
- **Adapter Pattern**: used for connecting our system to external providers interfaces such as Transport Companies and Weather Center;
- **Factory Pattern**: mainly used by EJBs and JPAs to handle connections with the database;
- **Dependency Injection Pattern**: used mostly in situations where we apply instance pooling of beans of the business part to respond to synchronous requests from different clients;

Notice that many other might be used during the implementation of the project.

G. Other design decisions

Our proposed implementation consists of different tiers, whose subcomponents might have already been declared in some sections of this document:

1. **Client tier(Web clients, application clients):** it contains all the components which run on the client machine (applications, web pages); in our case MyTravlendar+ system, which runs on a browser. This layer identifies all the thin clients, which means that they do not directly query the database, nor execute complex operations but only requests.
2. **Web tier(Java Servlet,JSP,JSF):** the components of this tier run on the Java EE server; this tier is intended to manage the data flow between clients and Java EE.
3. **Business tier (EJB e JPA Entities):** this tier, which runs on the Java EE server as well, contains the so called enterprise beans. Enterprise beans handle business code, which is the logic that governs Travlendar+ system. In order to achieve this, they are also able to ask for data from the database and send it back to the client program, through the web tier.
4. **EIS tier (Data Tier):** this tier, typically, handles EIS 2 software and includes enterprise infrastructure systems, mainframe transaction processing, database systems, and other legacy information systems. In our specific context, Java EE application components might need access to enterprise information systems for database connectivity.



Since our project will provide many functionalities we suggest a schedule of possible releases in order to satisfy all the functionalities mentioned in this document and the previous one.
We assume that everything that is required in order to have a running server and application will be given as a base for the releases described here where we provide only a suggested plan of how our project will be implemented once the main component of the whole system will be implemented:

- 1st release:
    - registration & login
    - calendar
    - basic travel computation
    - preferences (mean of transport selection) and lunch interval
    - weather
- 2nd release:
    - max distance preference introduction for each mean of transport
    - sharing systems introduction
    - public transport news alerts
    - deadline institution for updates on journeys
- 3rd release:
    - carbon footprint minimization preference
    - ticket shopping
    - preferred time for each mean of transport introduction

3. ALGORITHM DESIGN

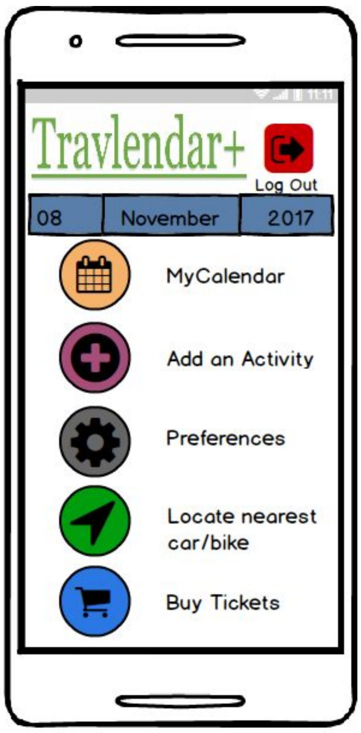
In this chapter we are going to outline the main algorithms that will govern our system.
To specify the algorithms in this chapter we decided to use Java pseudocode but our main intention is to focus on the structure of the algorithm instead of its real implementation.
overlappingChecker algorithm is a method that, when a user adds an event to his calendar, checks if it overlaps with other events already present and related to the same user.
This method will be used by OverlappingChecker component and it is necessary to complete the “Add an Event” function which can be considered as a main function of the system, that’s why we decided to analyse an algorithm like this one.

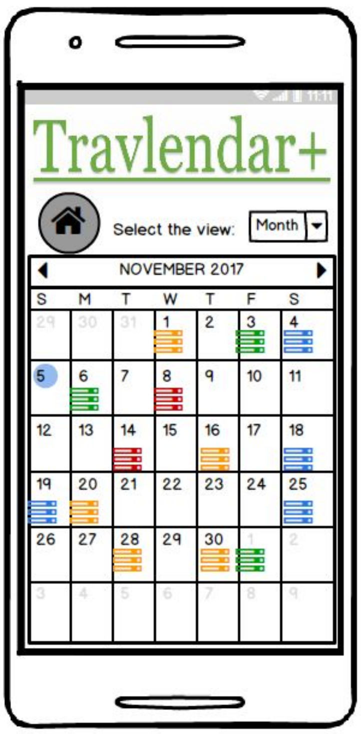
```java
ArrayList<Event> Calendar;  //events in the calendar are in chronological order

public Boolean overlappingChecker(Event e){ //return true if Event e doesn't overlap

    if(Calendar.get(Calendar.size()-1).endTime < e.startTime)
        //if we want to add an event after the last event in our calendar
        //there is no overlapping
        return true;

    if(Calendar.size() == 1){//there is only 1 event in the calendar
        if(Calendar.get(0).startTime > e.endTime)
            return true;
        else
            return false;
    }

    for(int i = 0; i < Calendar.size() - 1; i++){
        if(Calendar.get(i).endTime < e.startTime
            && Calendar.get(i+1).startTime > e.endTime)
            return true;
        if(Calendar.get(i).startTime > e.startTime)
            return false;
    }

    return false;
}
```

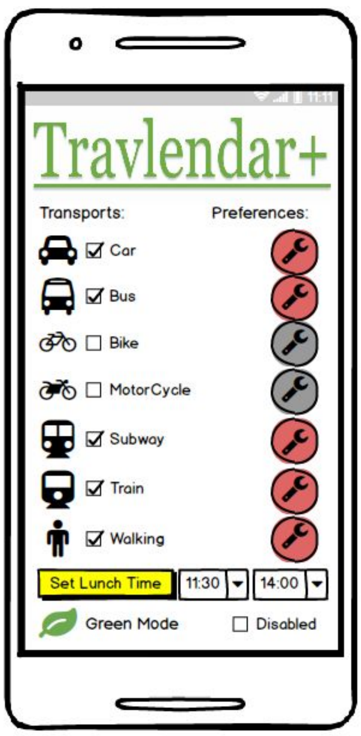
weatherChecker algorithm is a method necessary to check if the weather conditions are suitable with all the means of transport chosen for a journey.
This method is useful during the computation of the journey and also when an activity is added.

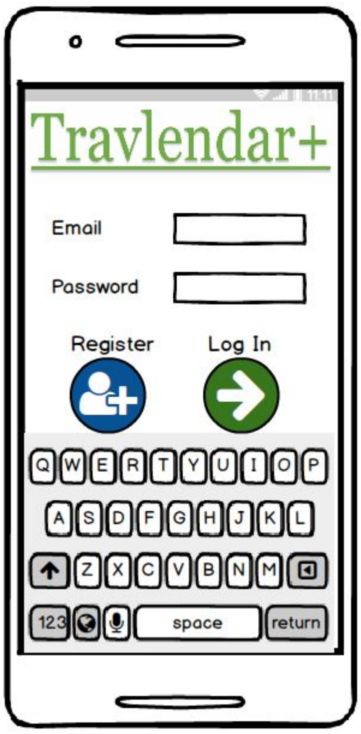
```java
//weatherChecker returns true if a journey means of transport are compatible with weather conditions
public Boolean weatherChecker(Journey j){

    //journey is divided in parts, every part contains a mean of transport, starting and ending location/time
    for(Part p: j.parts){
        //badWeatherCondition is a method that queries the weather company db and returns true if the weather
        //for a specific location is bad
        if(badWeatherCondition(p.location) && (p.meanOfTransport.equals("bicycle") || p.meanOfTransport.equals("walking"))){
            if(p.meanOfTransport.equals("bicycle"))
                return false;
            //distance return the distance of the part in meters
            else if(p.distance() > 300)
                return false;
        }
    }
    return true;
}
```

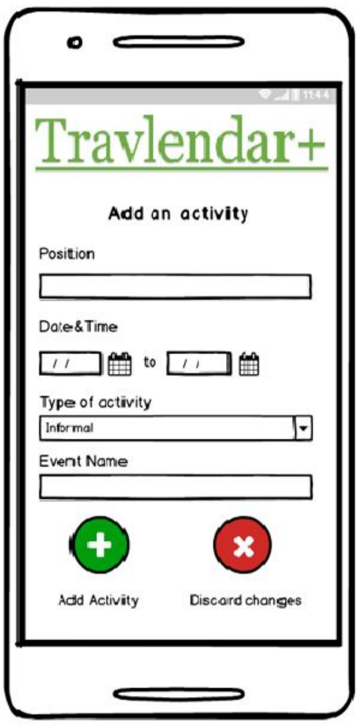
4. USER INTERFACE DESIGN

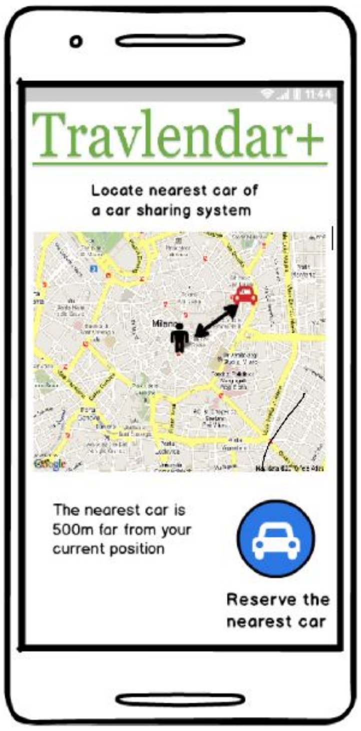




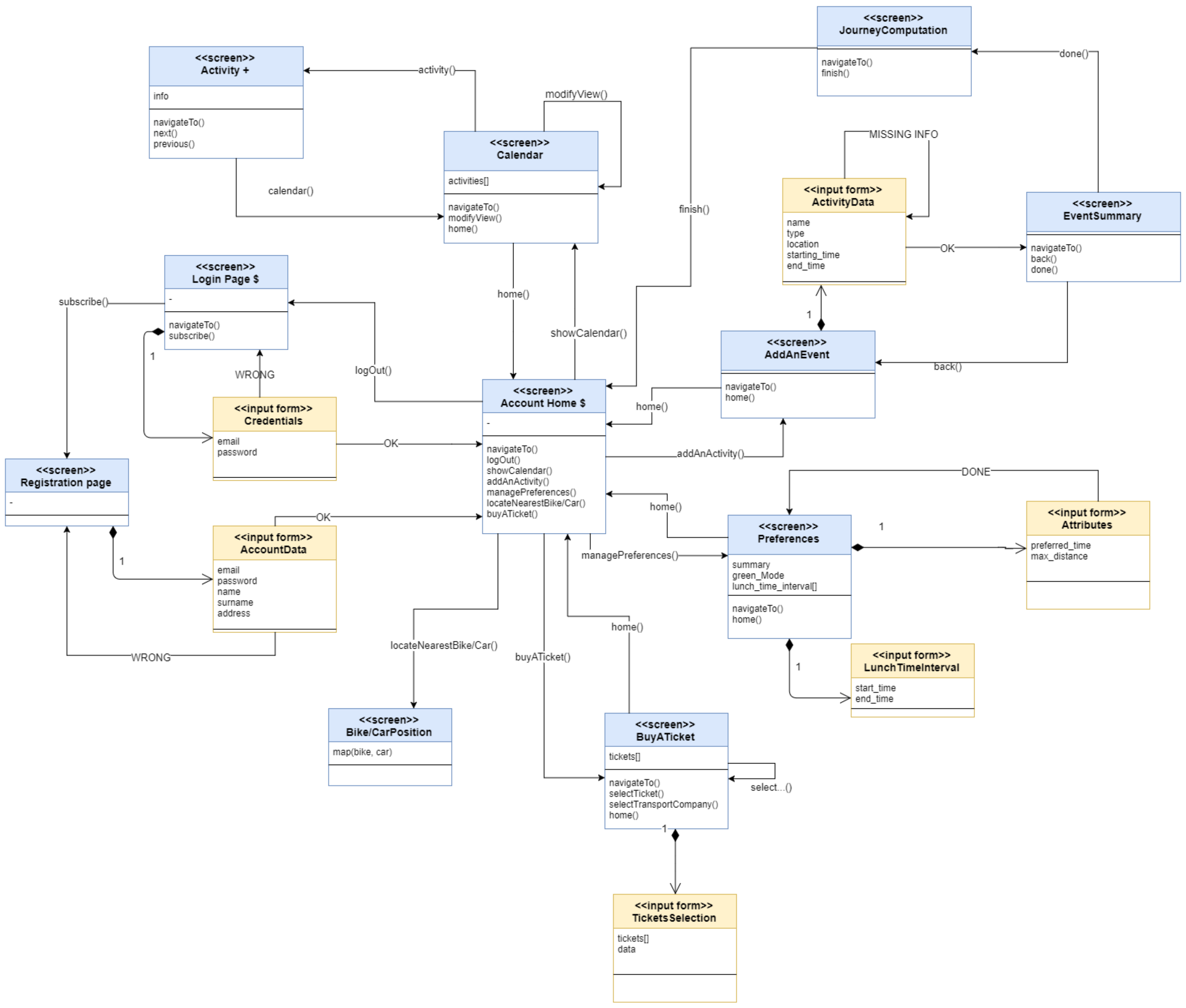







Travlendar+
Locate nearest car of a car sharing system
The nearest car is 500m far from your current position
Reserve the nearest car

In the RASD document few mockups of the user interfaces have been already introduced with a high level of abstraction, here some of them are reproposed in a more detailed way.
This is only a proposal and can be followed or discarded, it might depend on who takes care about the user interface implementation, generally we will leave more freedom to designers groups.
In order to better describe the user experience, a UX Diagram follows showing the possible sequences of interfaces that the user could see performing a specific task.



<<screen>> Activity +
info
navigateTo() next() previous()
<<screen>> Calendar
activities[]
navigateTo() modifyView() home()
<<screen>> JourneyComputation
navigateTo() finish()
<<input form>> ActivityData
name type location starting_time end_time
<<screen>> EventSummary
navigateTo() back() done()
<<screen>> Login Page $
navigateTo() subscribe()
<<input form>> Credentials
email password
<<screen>> Registration page
<<input form>> AccountData
email password name surname address
<<screen>> Account Home $
navigateTo() logOut() showCalendar() addAnActivity() managePreferences() locateNearestBike/Car() buyATicket()
<<screen>> AddAnEvent
navigateTo() home()
<<screen>> Preferences
summary green_Mode lunch_time_interval[]
navigateTo() home()
<<input form>> Attributes
preferred_time max_distance
<<input form>> LunchTimeInterval
start_time end_time
<<screen>> Bike/CarPosition
map(bike, car)
<<screen>> BuyATicket
tickets[]
navigateTo() selectTicket() selectTransportCompany() home()
<<input form>> TicketsSelection
tickets[] data
activity()
modifyView()
calendar()
finish()
done()
MISSING INFO
OK
back()
home()
showCalendar()
subscribe()
WRONG
logOut()
OK
addAnActivity()
DONE
managePreferences()
locateNearestBike/Car()
buyATicket()
select...()
1

5. REQUIREMENTS TRACEABILITY

In this section, we would like to highlight some significant correspondences between this document and the previous RASD. Indeed, functional and nonfunctional requirements expressed there were kept as a guide while writing this document.

First of all, our aim is to map the functional requirements listed in the RASD with the related components introduced *"Architectural Design"* section of this document.

- [G.1]: allow a Guest to create an account and log into it
    - [RE.1]: A visitor is able to begin the registration process. During the process the system will ask him/her to provide credentials.
    - [RE.2]:The system is able to check if credentials are correct or not and also if there's already an account linked to the inserted mail for registration procedure.
    - [RE.3]: The system is able to store the data inserted during the registration.
- [G.2]: allow the system to update stored info (timetables and weather forecasts)
    - [RE.4]: The system is able to retrieve data and info from Transport Companies and Weather Center.
    - [RE.5]: The system is able to retrieve data from external providers.
- [G.3]: allow the user to buy a specific public transport ticket
    - [RE.6]: The user is able to select tickets and the relative amount he wants to buy.
    - [RE.7]: The user receives a receipt of payment within the tickets he purchased.
    - [RE.8]: The user is able to buy online through most common payment methods such as PayPal and credit cards.
- [G.4]: allow the user to locate the nearest bike or car of a bike/car sharing system
    - [RE.9] The system is able to provide the list of available bikes within a certain area.
    - [RE.10]: The system is able to collect the location from the calendar regarding the previous registered activity.
- [G.5]: allow the user to add an activity to its calendar or modify an existing one
    - [RE.11]: The app must prevent data loss.
    - [RE.12]: The system is able to verify whether the additional event is reachable and the journey is feasible.
- [G.6]: allow the user to calculate the best route from an activity to the next one
    - [RE.13] The system is able to retrieve activities data from the calendar.
    - [RE.14] The system is able to retrieve weather and transport data.
    - [RE.15] The system takes into account user preferences during the journey computation.
- [G.7]: allow the user to be in time for his appointments
    - the same for [G.6]

- [G.8]: allow the user to select different kind of preferences
    - [RE.16]: The system is able to store account values regarding different preferences.
- [G.9]: allow the user to consult his calendar
    - [RE.17]: The system is able to provide to the user all the list of registered events.

REQUIREMENT	**SERVER COMPONENTS**	**THIN CLIENT COMPONENTS**
RE.1	UserRegistration, UserDataManagement	Registration
RE.2	UserLogin, UserDataManagement	Login
RE.3	UserDataManagement	/
RE.4	WeatherDataManagement, TransportDataManagement	/
RE.5	WeatherDataManagement, TransportDataManagement	/
RE.6	TransportDataManagement	TicketShopping
RE.7	NotificationEngine	TicketShopping, NotificationReader
RE.8	/	TicketShopping
RE.9	ActivityManagement	ActivityHandler
RE.10	UserDataManagement	/
RE.11	/	/
RE.12	OverlappingChecker, UserDataManagement, NotificationEngine	NotificationReader
RE.13	UserDataManagement	CalendarViewer
RE.14	TransportDataManagement	/
RE.15	UserDataManagement, JourneyChecker	PreferencesHandler
RE.16	UserDataManagement	/
RE.17	UserDataManagement	CalendarViewer

6. IMPLEMENTATION, INTEGRATION AND TEST PLAN

In this section we will provide a general plan for each process: implementation, testing and integration. We will also analyse the relationships between each process and the others since we think this will help the understanding of the whole process we want to describe.

Bell-Northern Research:

- Inspection cost: 1 hour per defect.
- Testing cost: 2-4 hours per defect.
- Post-release cost: 33 hours per defect

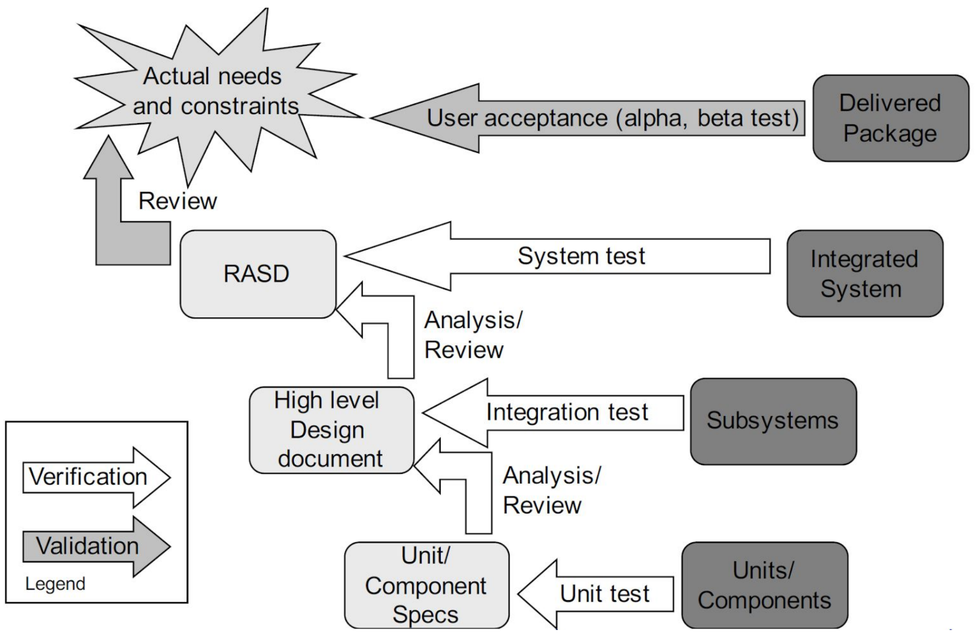
Since this increasing load of defecting time highlighted by Bell-Northern, we strongly suggest to follow our integration and testing plan, seeing it as a parallel task of the implementation part in order to minimize the whole defecting workload.

A. Implementation

Regarding the implementation, we have already discussed some details in the 2.G section. Other suggestions can be to look for and employ all the useful APIs that can be found and helps to achieve a given functionality. The implementation phase will firstly focus on the main aspect of the architecture we have already defined in section 2 and secondly on the functionalities that must be provided in order to fulfil the requirements described in the RASD. Among the entire phase of implementation, parallel to the testing as mentioned multiple times, developers must be focused on the implementation of the integration aspects of different components. Nevertheless we strongly recommend to use common sense in using or choosing external APIs, trying to analyse their goodness and reliability.

B. Integration and Test Plan

An Incremental Integration Test is suggested as soon as a first version of two components is released, they should be integrated and tested, then if test passes the procedure can continue with all the remaining modules.
A Structural Strategy is valid in our case considering that relatively small components and subsystems will be implemented. For example, JUnit could be used as a testing tool, its most important property is to provide a mixed testing process using both top-down and bottom-up hierarchical structure.
Modules will be developed starting from the riskiest ones, firstly focusing on the components that identify the fundamentals of the system. Two different types of testing will be used during the testing plan, *white-box* for **unit testing** while *black-box* for **integration testing, system testing and user acceptance testing**. The testing will be done entirely using a systematic way.



A well-known practice of project testing-implementation planning, also achievable through the already mentioned JUnit, is to write unit tests before their implementation, trying to achieve implementation and testing as parallel tasks. Finally we suggest to plan automatic test execution since can strongly speed up testing processes, avoiding the intervention of CLI or GUI.

Concerning the all **testing** phase, we would like to highlight that it is strictly related to the choices made during the implementation:

1. regarding the code internally produced we can perform the entire testing process, starting from white-box unit testing;
2. while the use of external APIs, seen as black boxes, will be focused mainly on the integration testing, trying to use them in the best possible way (in this part we are referring to all the external providers our system will interrogate and also other useful APIs that could be imported in the logic of our system);

The **acceptance test** for the first release will be focused on usability feedback from a subset of users and will check typical security problems (focusing on requirements and design validation), while the acceptance test for the following releases will include a check of all functionalities and reliability measures.

System Test Plan

Other different tests could be eventually performed to ensure the proper functioning of the whole Travlendar+ System.

- *Performance Tests:* to identify bottlenecks that may affect the responsiveness and the usability of the application, the aim is to identify any inefficient algorithm and query or to find hardware limitations.
- *Load Tests:* to expose bugs and identify component limits, the load on the server will be kept at the maximum level for a long period to check its stability.
- *Stress Tests:* to make sure that the System recovers after failure, the server will receive requests for double the users it is developed to support on parallel.

7. EFFORT SPENT

Engineer	Student ID	Effort Time
*Alessio **CANTINA***	*895395*	*15h 15m*
*Simone **CRIPPA***	*898304*	*13h 20m*
*Nicola **CUCCHI***	*893748*	*18h 05m*

Nicola Cucchi

-	1h 30m	Introduction part	03/11
-	3h 40m	Component View part	04/11
-	3h 20m	UX Diagram + Mockups + Component Interfaces	05/11
-	1h 30m	Runtime View + Component View	06/11
-	1h 55m	Requirements Traceability + Integration & Test Plan	11/11
-	40m	Other decision design	23/11
-	2h	1.C/2.D/2.F Sections additions	24/11
-	2h 30m	Runtime View + Implementation, Integration and Test Plan	24/11
-	1h	Final Review	25/11

Simone Crippa

-	2h 50m	Component View part + Mockups	04/11
-	2h 30m	Runtime view	05/11
-	2h 30m	Algorithm Design	05/11
-	2h 00m	Implementation & Test Plan	11/11
-	3h 30m	User interface update, Runtime View corrections, DD check	24/11

Alessio Cantina

-	3h 30m	Component View + Component Interfaces	04/11
-	5h 00m	Deployment View + Algorithm Design	05/11
-	2h 00m	Implementation, Integration and Test Plan	11/11
-	4h 00m	Deployment View fix, whole DD check, Implementation	24/11
-	45m	Final Review	25/11

8. REFERENCES

During the drawing up of the document the following tools have been used:

- Draw.io 7.5.6
- Google Docs
- StarUML version 2.8.0
- Balsamiq Mockups 3.5.15