Politecnico di Milano
AY 2017/2018

# Travlendar+

Design Document

Mirko Salaris, Piervincenzo Ventrella, Pietro Cassarino

November 26, 2017

| | |
|---|---|
| **Deliverable:** | DD |
| **Title:** | Design Document |
| **Authors:** | Mirko Salaris, Piervincenzo Ventrella, Pietro Cassarino |
| **Version:** | 1.0 |
| **Date:** | 26-November-2017 |
| **Download page:** | https://github.com/mirkosalaris/CassarinoSalarisVentrella/ |
| **Copyright:** |  |

# Table of Contents



# List of Figures

# 1 Introduction

## 1.1 Purpose

## 1.2 Scope

## 1.3 Definitions, Acronyms, Abbreviations

### 1.3.1 Definitions

Here we provide a list of definitions of words and expression used in the documents. Every time such words or expressions will be used they will be preceded by the symbol "(↑)" that will be a link to this section.

– ***Incoming Appointment***: the next scheduled appointment for which the S2B send a reminder to the user. The reminder is sent a certain amount of time before the appointment starts, to allow the user to get on time in the location.

– ***Future Notification***: a notification that is generated but it will be eventually sent only in a future time. The time has to be specified during the creation.

– ***Dynamic Configuration***: with this term we mean a reconfiguration that can be done without powering off the server or any physical component.

### 1.3.2 Acronyms

### 1.3.3 Abbreviations

## 1.4 Revision history

## 1.5 Document Structure

# 2 Architectural Design

## 2.1 Overview

Here we provide a high level representation of client-server interaction and of the submodule of the server. The orchestrator is needed only during the creation of the communication: his role is to dispatch the request of the client to the appropriate component. After that, the component can communicate directly with the client and vice versa.
In the server, the orchestrator and all the submodules are stateless. We thought to use the elastic component architectural pattern for the components. Moreover, the orchestrator can eventually be duplicated using a fixed dimension pool whose size is configurable by the system administrator.
Further details on the submodule and the overall interaction will be provided in the next sections.

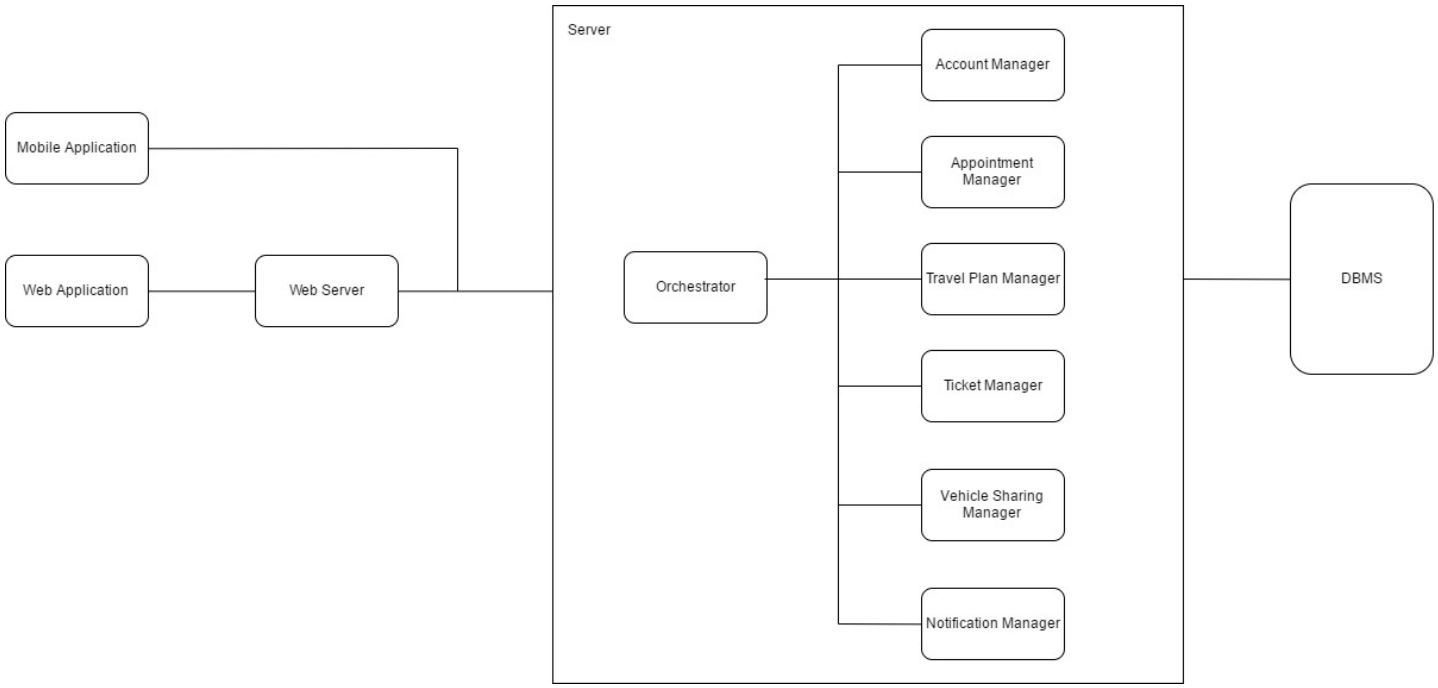


Figure 1: High level Architecture Diagram - Client-Server interaction

## 2.2 Component view

This section starts with the presentation of the Entity Relationship diagram. In the other diagram following ER model, we include a fictitious component, App, that will be highlighted in green and serves the purpose of representing both the mobile app and the web app (through the web server), without adding complexity to the diagrams.

### 2.2.1 ER diagram

The following ER diagram represents the conceptual schema of the system database. In Ticket entity we don't specify all the attributes because they depend from ticket Typology and from Transportation Company they belong. Ticket entity can be split into two categories of tickets, ordinary and pass tickets. Below these two, we have another hierarchy in which we include some examples.

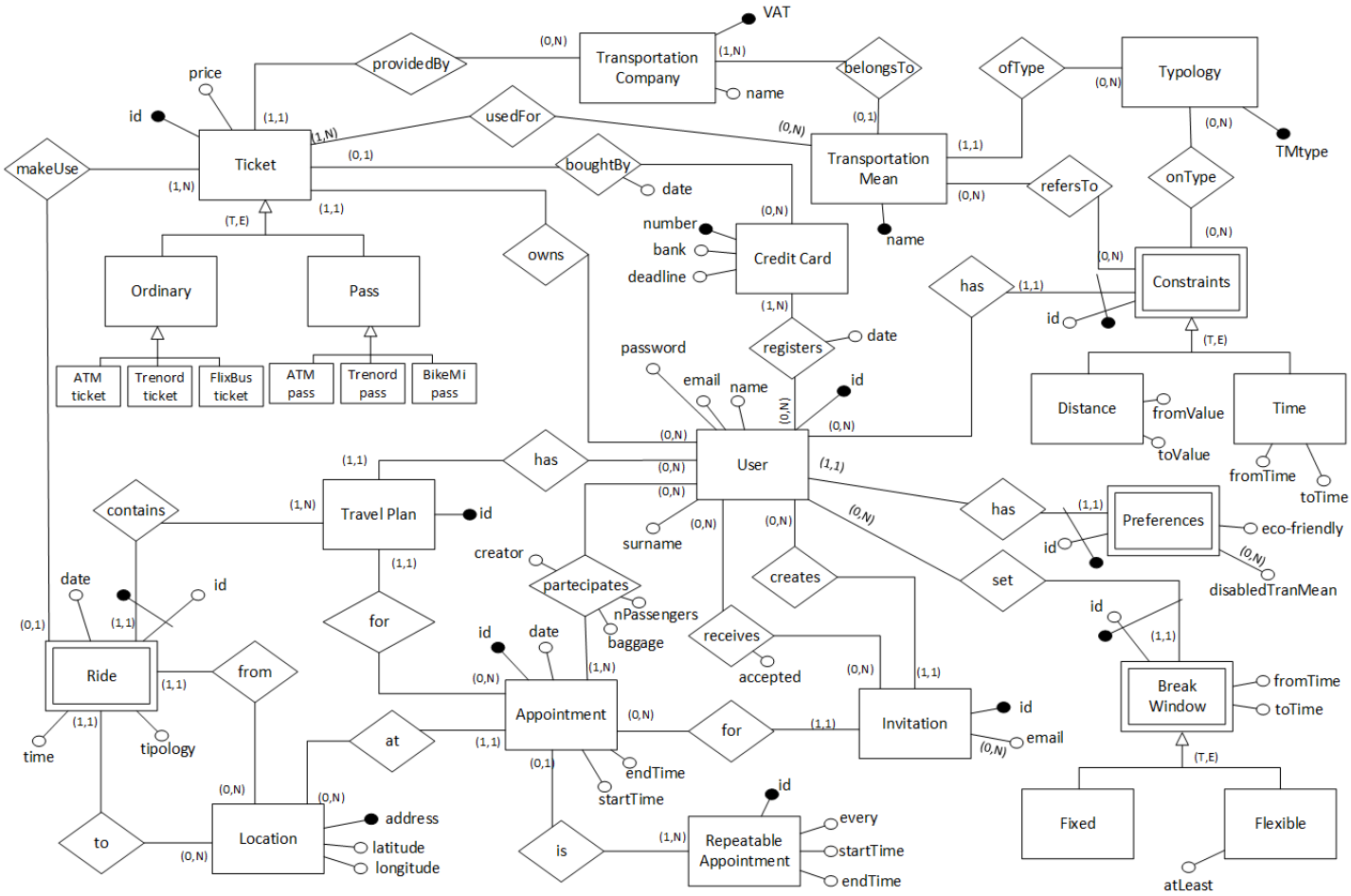


Figure 2: ER diagram

### 2.2.2 Appointment Manager and Notification Manager

Here we define the Appointment Manager and the Notification Manager. The Notification Manager will appear in others diagrams, but only here is detailed with its subcomponents.

Appointment Manager has three subcomponents:

– *Appointment Handler*: it is the only subcomponents communicating with the database: it manages all the read and write operations related to appointments. It is responsible for the final consistency check of the appointment before writing it to database. It exports an interface to let external components read appointments details and an internal interface to let subcomponents of Appointment Manager read and write appointments' details.

– *Appointment Editor*: this provides an interface to easily change appointments' details and store it to database through the interface of Appointment Handler. Every time an appointment is created or details of an appointment changes, this send to Incoming Appointment Scheduler the appointment. This module interacts with an external module "Solution Calculator" to compute travel solutions. In this diagram it is displayed in dashed line because it will be described in another diagram.

– *Incoming Appointment Scheduler*: for each appointment (provided by Appointment Editor) this module executes an algorithm to decide when that specific appointment will become an (↑)*incoming appointment*. An appointment becomes incoming according to a combination of the time of the appointment and the travel plan chosen, to let the User receive a notification at an appropriate time. This is done generating a (↑)*future notification* through the interface provided by Notification Manager.

Notification Manager provide an interface for the creation of new notification and it is composed by two module:

– *Notification Scheduler*: this is the module whose interface is actually exported and it manages the creation of new notifications, stores them in the database and schedules their dispatch. When it is time to send a notification this module uses the interface provided by Notifier and delegate to it the dispatching procedure.

– *Notifier*: it is in charge of the actual dispatch of the notification.

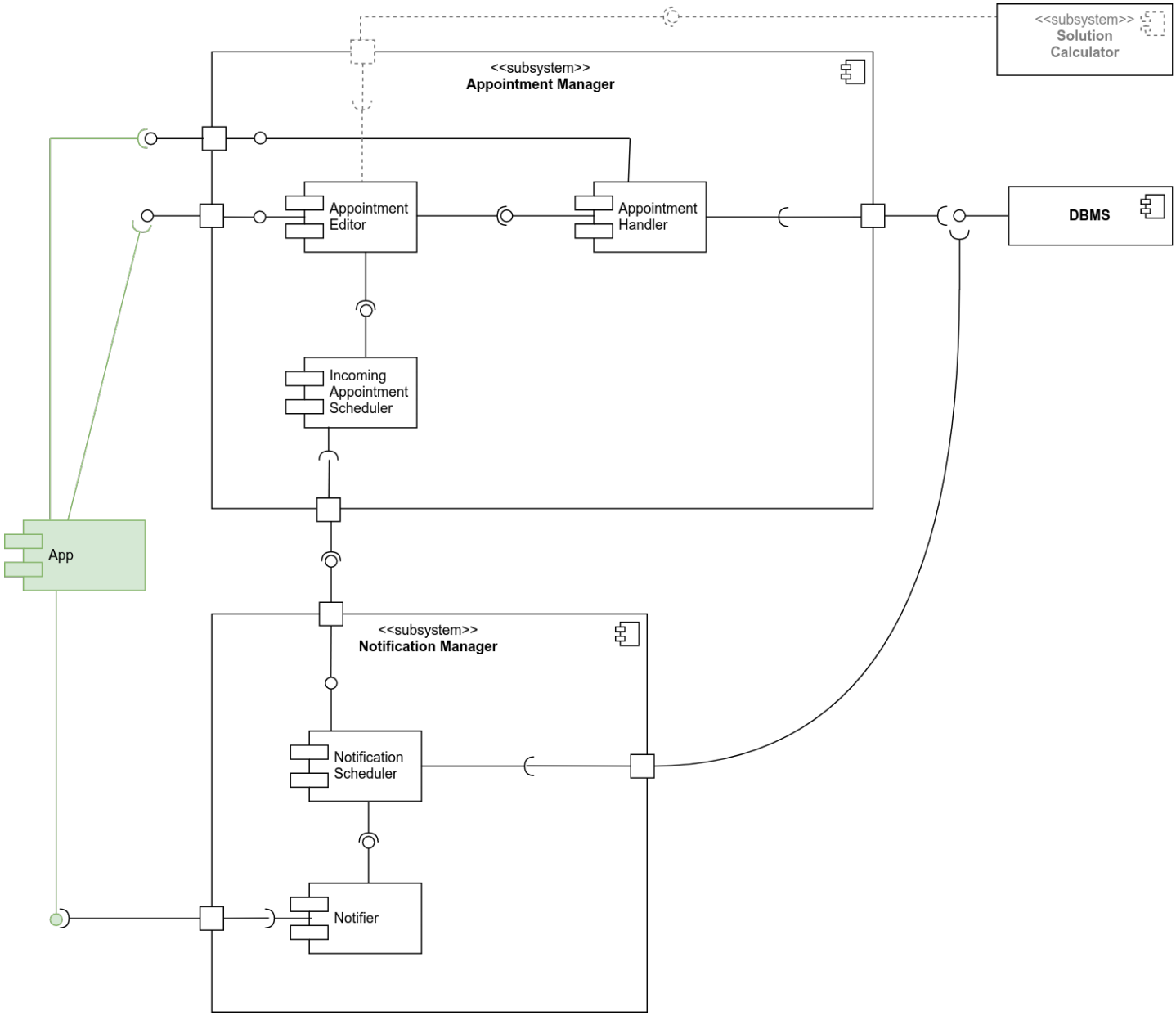


Figure 3: Component Diagram - Appointment Manager & Notification Manager

### 2.2.3 Invitation Manager

The Invitation Manager component is the module that manages all the aspects related to invitations, from their creation to the acceptance/refusal of the invited Users/Persons. It is divided into two sub-components. A sequence diagram is provided in section 2.4.

– *Invitation Generator*: it manages the creation of a new invitations

– *Invitation Handler*: it is concerned with the acceptance/refusal of the invitations sent.

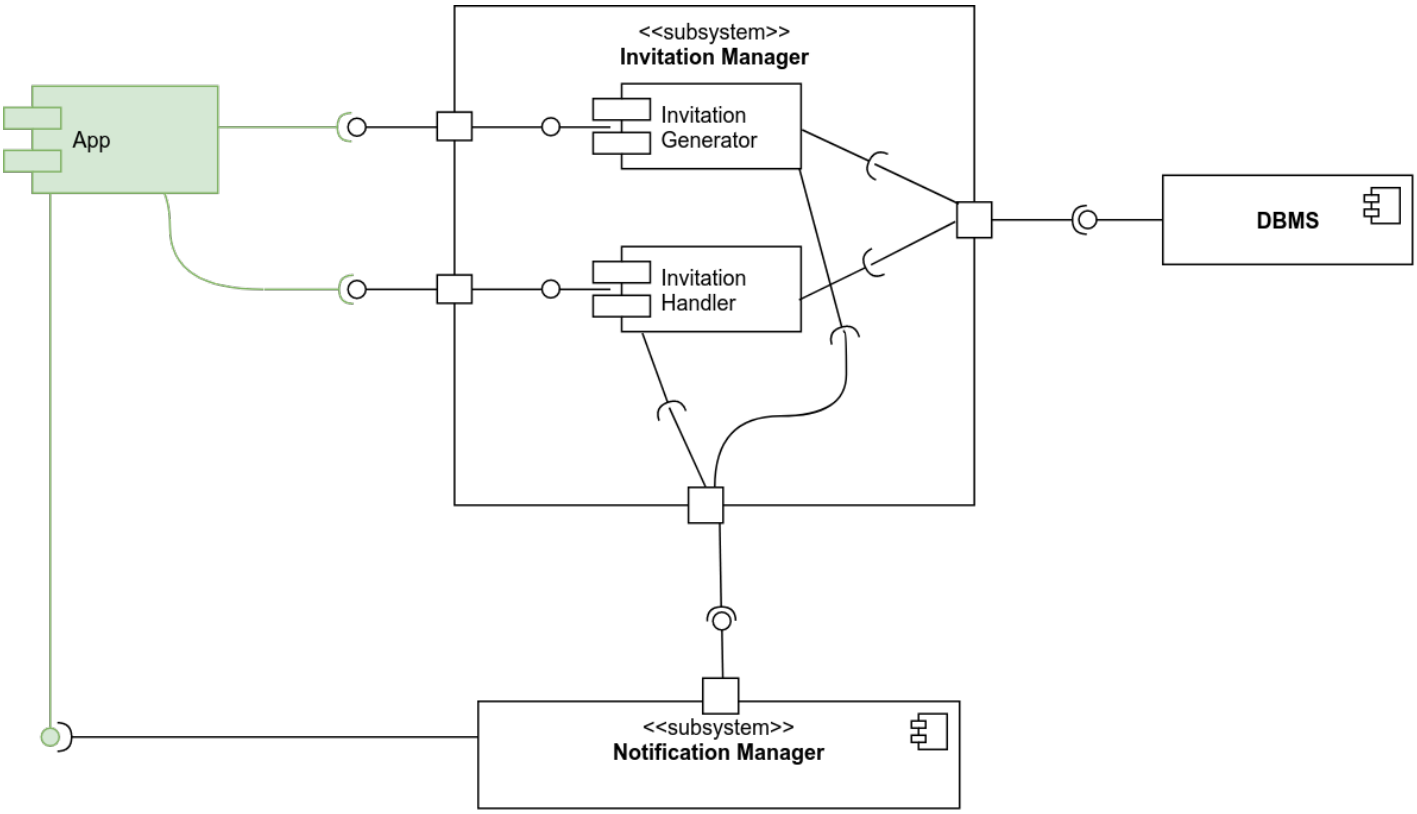


Figure 4: Component Diagram - Invitation Manager

### 2.2.4 Weather and Traffic Modules

In the diagram, we provide the representation of the Weather Module. The traffic module has exactly the same architecture and interactions. You can just derive the traffic diagram substituting *"Weather"* with *"Traffic"*. Both modules are needed to notify the User of changes in traffic/weather that can influence his/her daily travels and they are used by the component Solution Calculator to avoid bad solutions like, for instance, taking the bike when it rains. With the same convention as before, we represent here Solution Calculator but it will be detailed in another diagram.
The Weather *(traffic)* module has a quite articulated architecture and it will be explained here and in section 2.4 with an object diagram.
It is composed by:

– *Address Solver*: this component has a really basic functionality and it is exploited by Weather Manager to "understand" *(see below)* addresses. Provided an address as input, it decompose it in its components and return a hierarchy.
Example: "Italy, Milan, via Pacini 32" -> transformed to:
"State:Italy"→"City:Milan"→"CityZone:NorthEast"→"Street:Giovanni Pacini"→"Number:32"

– *Weather Manager*: this serves as a registry to support the publish-subscribe architecture between Weather Querier and Weather Notifier. Moreover, this component provides an external interface to let other components have information about weather. When it receives a request of weather information for a specific address, it asks Address Solver to interpret the address and then it asks the appropriate Weather Querier for the information.

– *Weather Querier*: it is the module responsible for retrieving the weather information and send them to the Weather Notifier.

– *Weather Notifier*: this is the component that decides if and when a User has to be notified. It subscribes to a specific Weather Querier and receives the information from it.

Both the Querier and the Notifier are instantiated by zone. The zone of the Querier are meant to be very wide, like Italy, Spain, France and the zone of the Notifier are meant to be more specific, for instance: West Milan, East Milan, Venice, Turin.
The Notifier has to execute an algorithm for each appointment to decide if and when to notify the User, so its load will depend on the number of appointments in that specific zone and (especially in the Traffic Module) on the day of the week and the time of the day. For this reason, a load balancing mechanism is required. When a 'node' is under load it can be splitt into two or more parts: "Milan"→"WestMilan" and "EastMilan".
The Querier has a more stable load and, for this reason, **no** load balancing mechanism has to be implemented. By the way, the zones of the Querier can be configured by the System Administrator, even dynamically.
The load balancing mechanism of the Notifier and the (↑)*dynamically configurable* Querier has to be supported by the Manager: it is not a simple registry but it registers all the active Queriers and Notifiers and it manages the subscriptions when they are modified.
The process is described in more details in section 3.

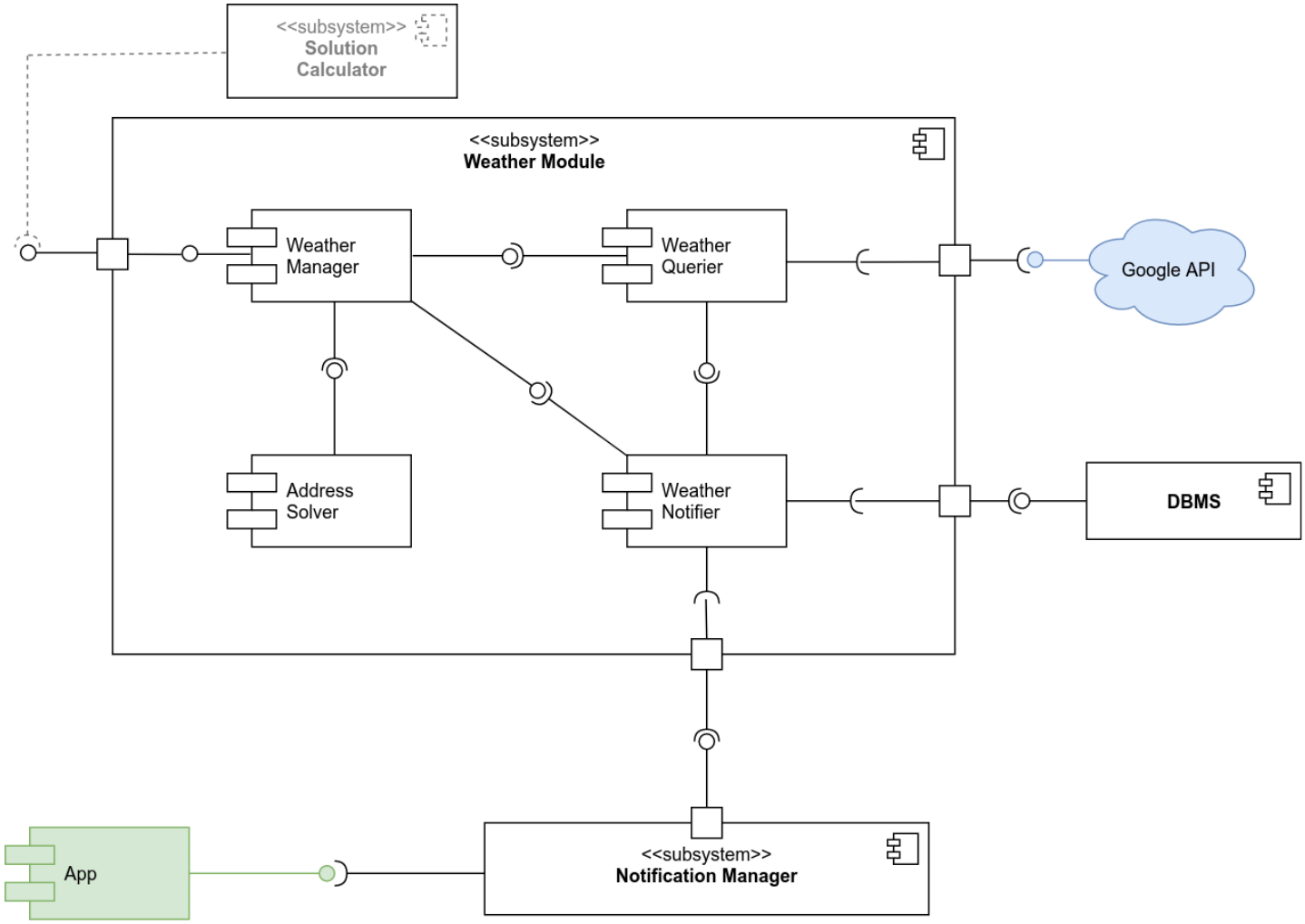


Figure 5: Component Diagram - Weather Module

## 2.3 Deployment view

## 2.4 Runtime view

### 2.4.1 Sequence Diagram - Sign Up

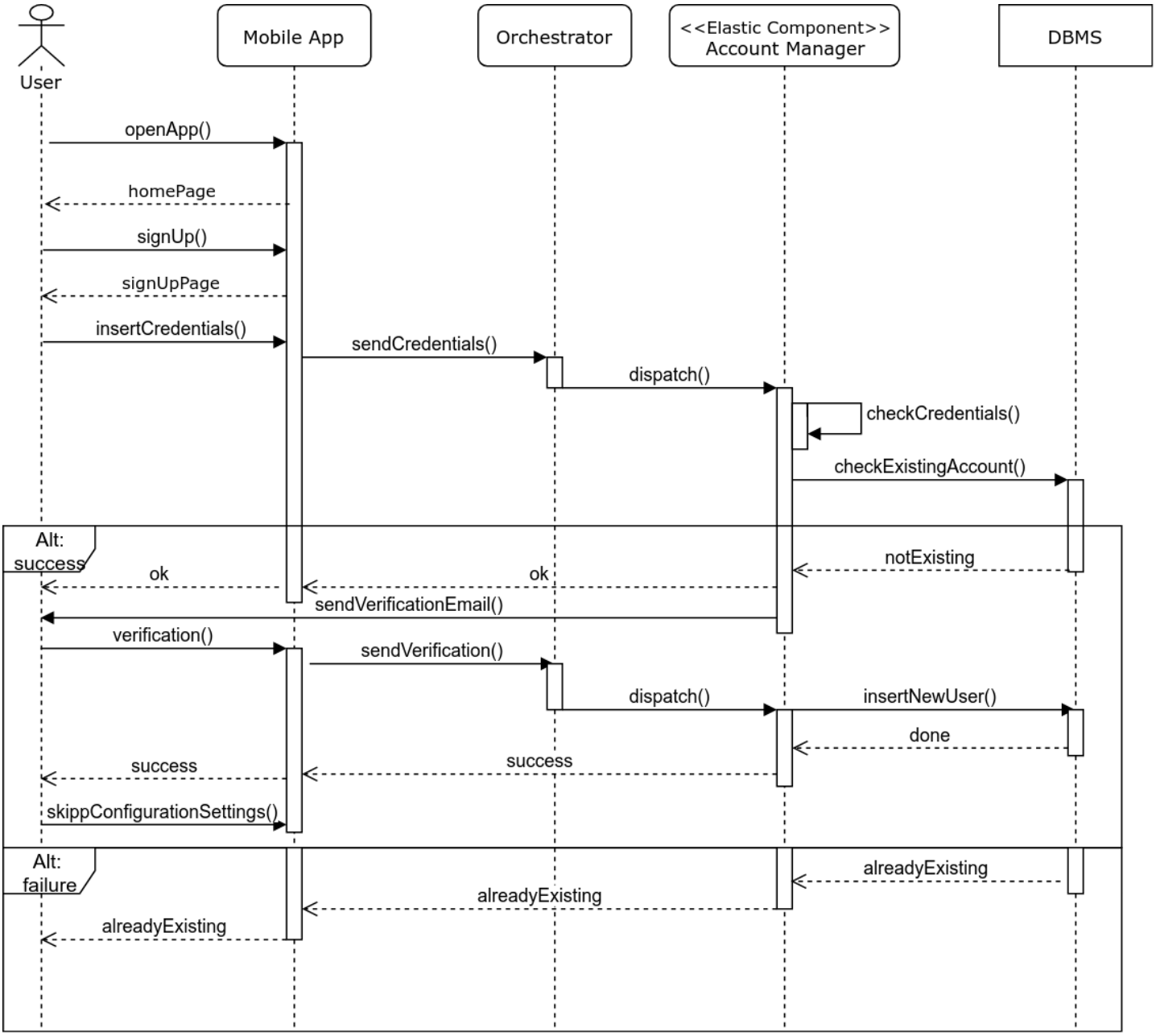


Figure 6: Sequence Diagram - SignUp

### 2.4.2 Sequence Diagram - Modify Settings

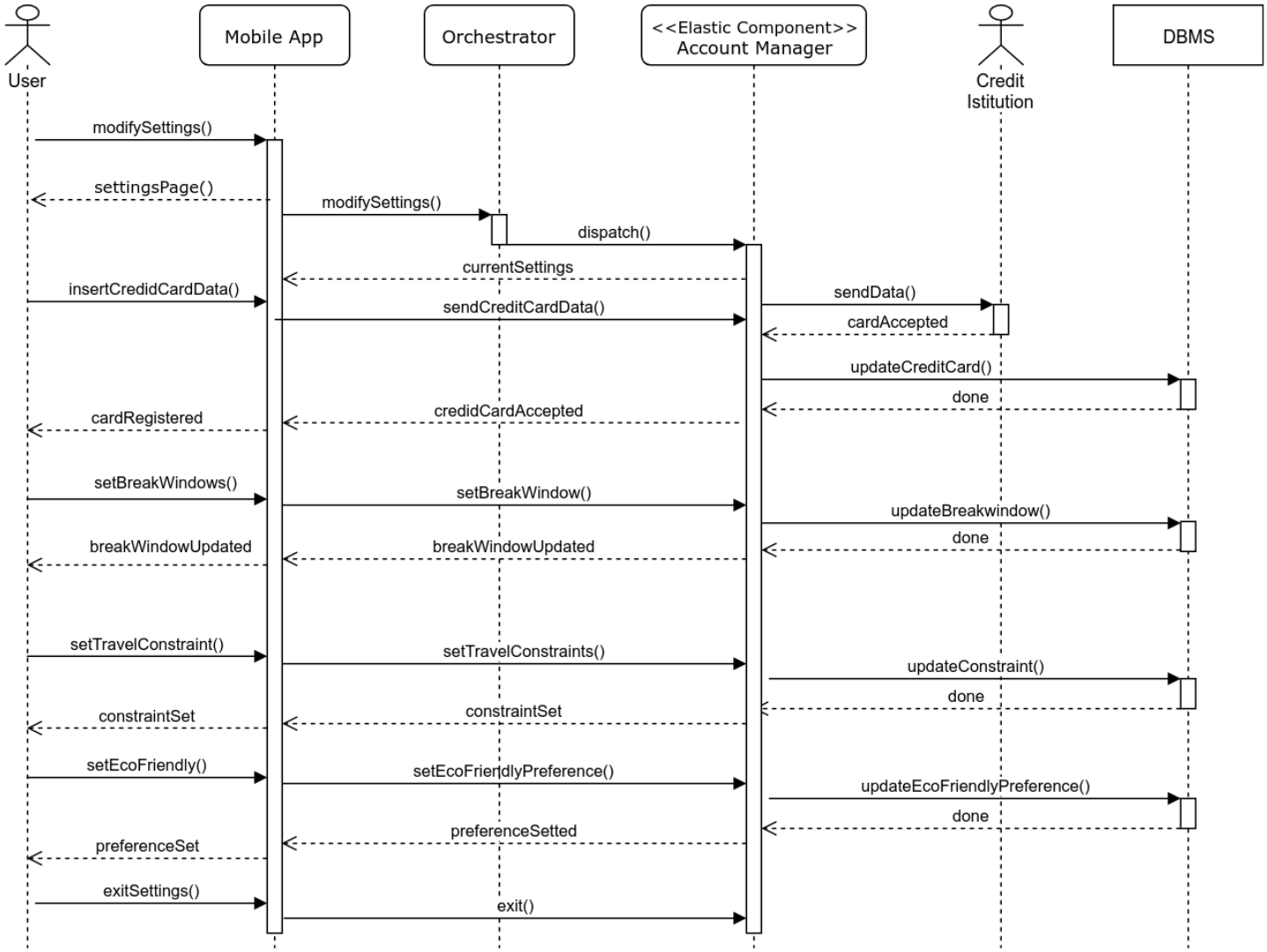


Figure 7: Sequence Diagram - Modify Settings

### 2.4.3 Sequence Diagram - Invitation Creation

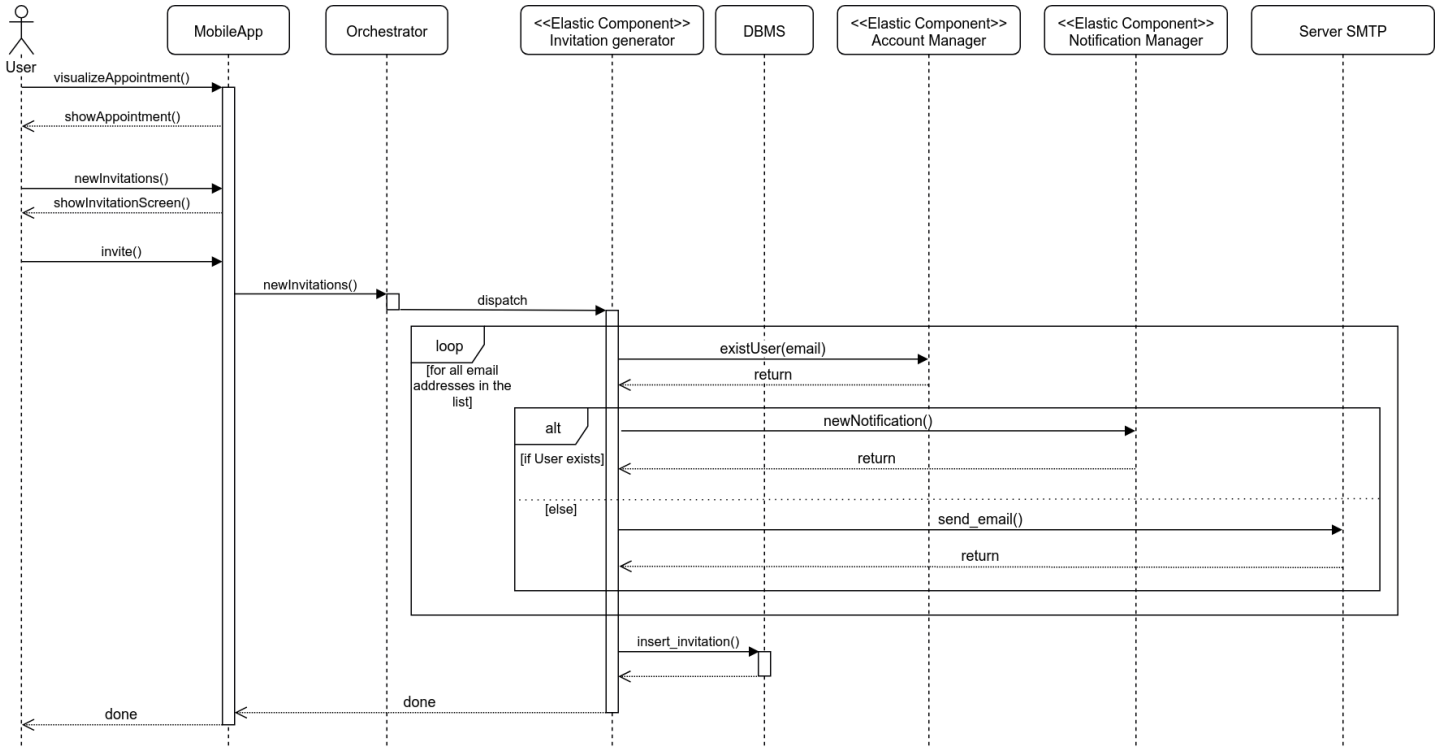


Figure 8: Sequence Diagram - Invitation Creation

### 2.4.4 Sequence Diagram - Locate Nearest Vehicle

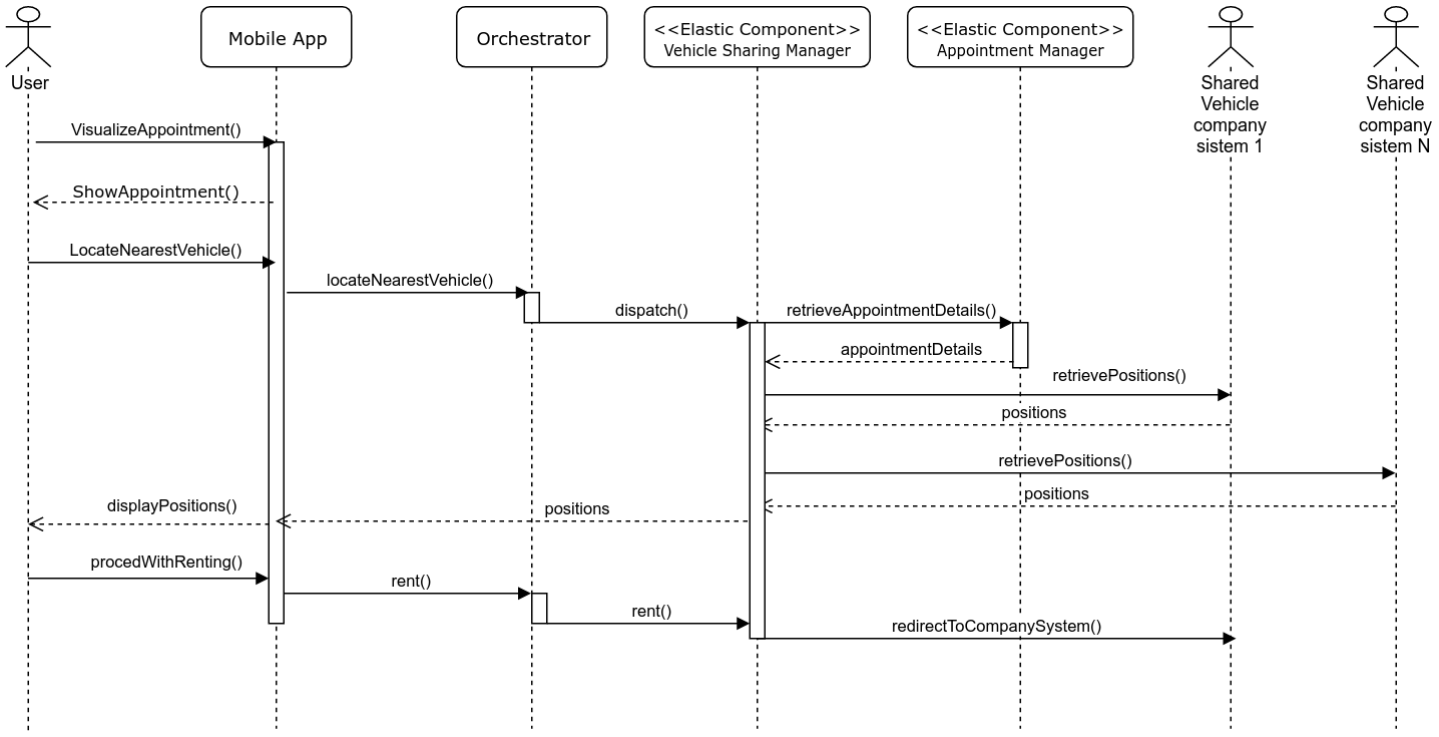


Figure 9: Sequence Diagram - Locate Nearest Vehicle

### 2.4.5 Object Diagram - Weather & Traffic Modules

As said in the component diagram description, this architecture is exactly the same used in the Traffic Module and so we provide here only the object diagram of one of them, the Weather Module, for consistency with the component diagram.
The two diagrams illustrate an evolution in the instances caused by a load balancing operation and an unexpected crash. Both diagrams are not complete of all the instances and we used dashed lines to represent the fact that some instances have been cut, to have simpler and easier to understand diagrams.
In the first diagram, we can see two Querier, one for Italy and one for France. This partition of the region have been chosen by the System Administrator (another option could have been Europe). The Querier are linked to the related Notifiers and to the Manager. The Manager keeps track of the Queriers, the active Notifiers and the Notifiers in the standby list.

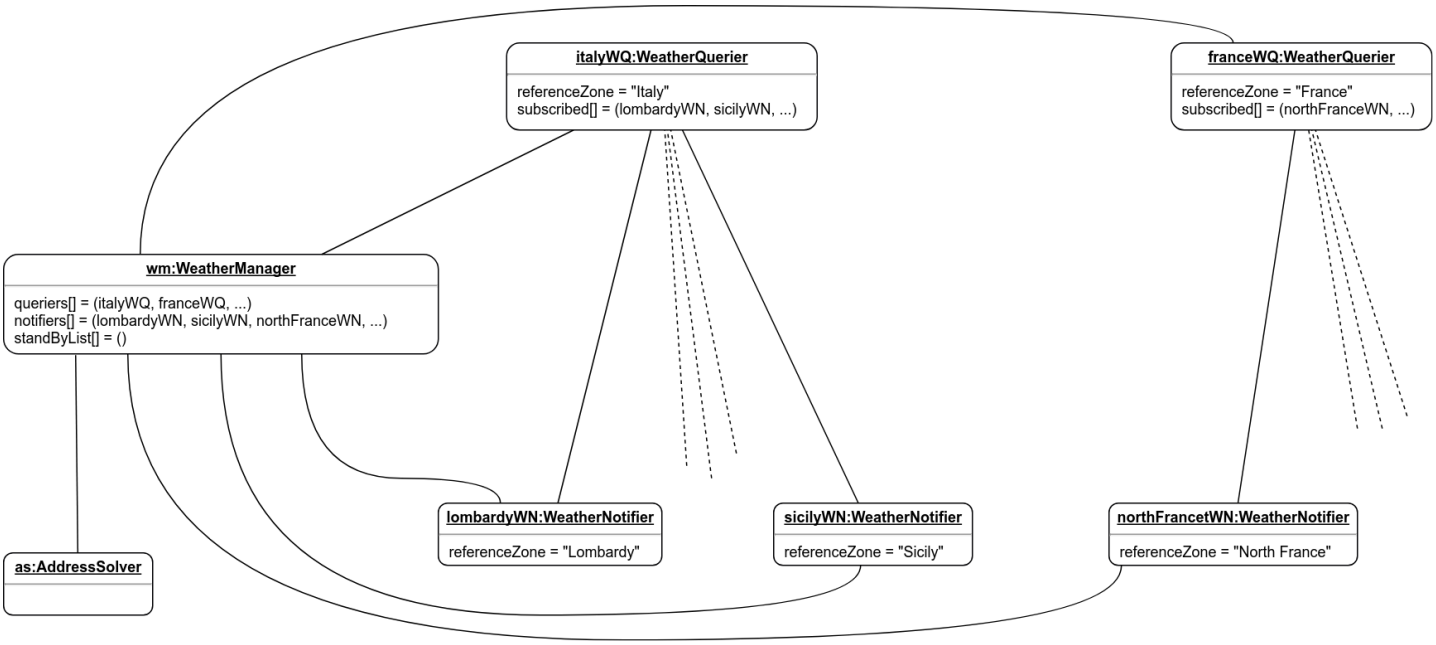


Figure 10: Object Diagram - Weather Module Original State

At some point, the Notifier related to Lombardy is under load and the load balancer split it into two Notifiers, respectively related to North Lombardy and South Lombardy. Their subscription to the Querier `italyWQ` is managed directly by the Manager as described in algorithm section (3.1).
The `franceWQ` Querier unexpectedly crashes and the Notifiers that were subscribed to it are put in the standby list by the Manager.
The next diagram represents the final situation.

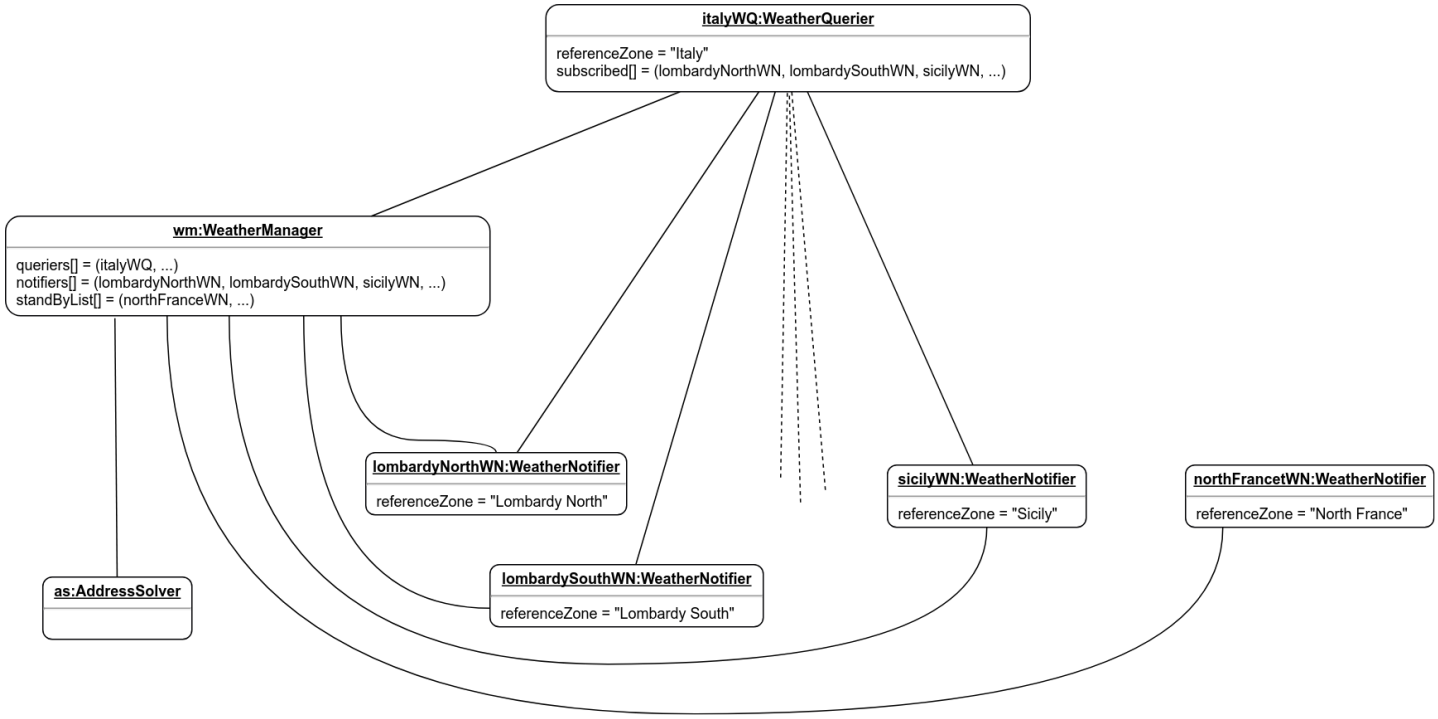


Figure 11: Object Diagram - Weather Module After Balancing and Reconfiguration

## 2.5 Component interfaces

## 2.6 Selected architectural styles and patterns

## 2.7 Other design decisions

# 3 Algorithm Design

## 3.1 Weather and Traffic modules - Dynamic configuration

As we previously described the Weather and Traffic modules are subject to load balancing and (↑)*dynamic configuration* by the System Administrator. This two mechanism causes 4 events to happen. How they are managed is described below:

- *a Notifier is deleted*: this is the only activity that is not managed by the Manager because there is no reason for doing it. If the Notifier is simply being closed, it detaches itself from the Querier. If the Notifier crashes or it is suddenly closed, the Querier will notice this when trying to notify it and it will detach the dead Notifier. No other actions are needed.
- *a Notifier is created*: the new Notifier communicates its zone to the Manager. The Manager will return to the Notifier a reference to the appropriate Querier using the Address Solver to interpret the zone of the Notifier. Thus, the Notifier can subscribe itself to the Querier.
- *a new Quierier is deleted*: all the Notifiers previously attached to the Querier have to be analyzed. If a less specific Querier exists *(see below)*, they are attached to it, otherwise they are put in a standby list (they will not receive any information about weather, or traffic)
- *a new Querier is instantiated*: the standby list is scanned searching for Notifier that can be attached to the Querier (matching the two zone through the Address Solver). If a less specific Querier exists *(see below)*, all the Notifier subscribed to it are analyzed and are eventually moved to the new Querier.

**Meaning of specificity of a Querier**
Let's assume, for instance, that there are four Querier: `ItalyQ`, `MilanQ`, `LazioQ`, `ParisQ`, respectively related to zones: Italy, Milan, Lazio, Paris. `MilanQ` and `LazioQ` are more specific with respect to `ItalyQ` because Milan and Lazio are inside the region Italy. On the other hand, `ItalyQ` is less specific with respect to `MilanQ` and `LazioQ`. `ParisQ` has no relation of specificity with all the others.
Hence:

- `LazioQ` crashes → all the Notifiers subscribed to `LazioQ` are now moved to `ItalyQ`
- `RomeQ` Querier is created → `ItalyQ` is less specific than `RomeQ`, so `ItalyQ` is scanned searching for Notifiers associated to region Rome: if there are such Notifiers, they are moved to `RomeQ`. Notice: the Notifiers that were associated to `LazioQ` but are actually outside the region of Rome would remain associated to `ItalyQ`.
- `ParisQ` crashes → all the Notifiers associated to `ParisQ` are moved to the standby list, because no 'less specific' Queriers are available.

Notice that Queriers are meant to be wide regions and not single cities, this was just an example.

# 4 User Interface Design

## 4.1 UX Diagrams

We chose not to expand the methods preceded by **, considering them unattractive and not to weigh the diagram unnecessarily.

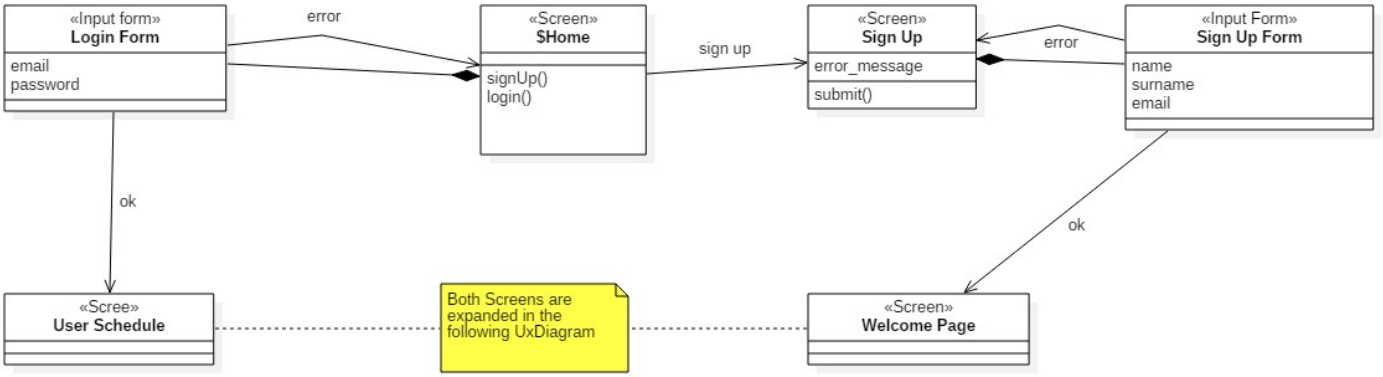


Figure 12: UX Diagram - Person UX

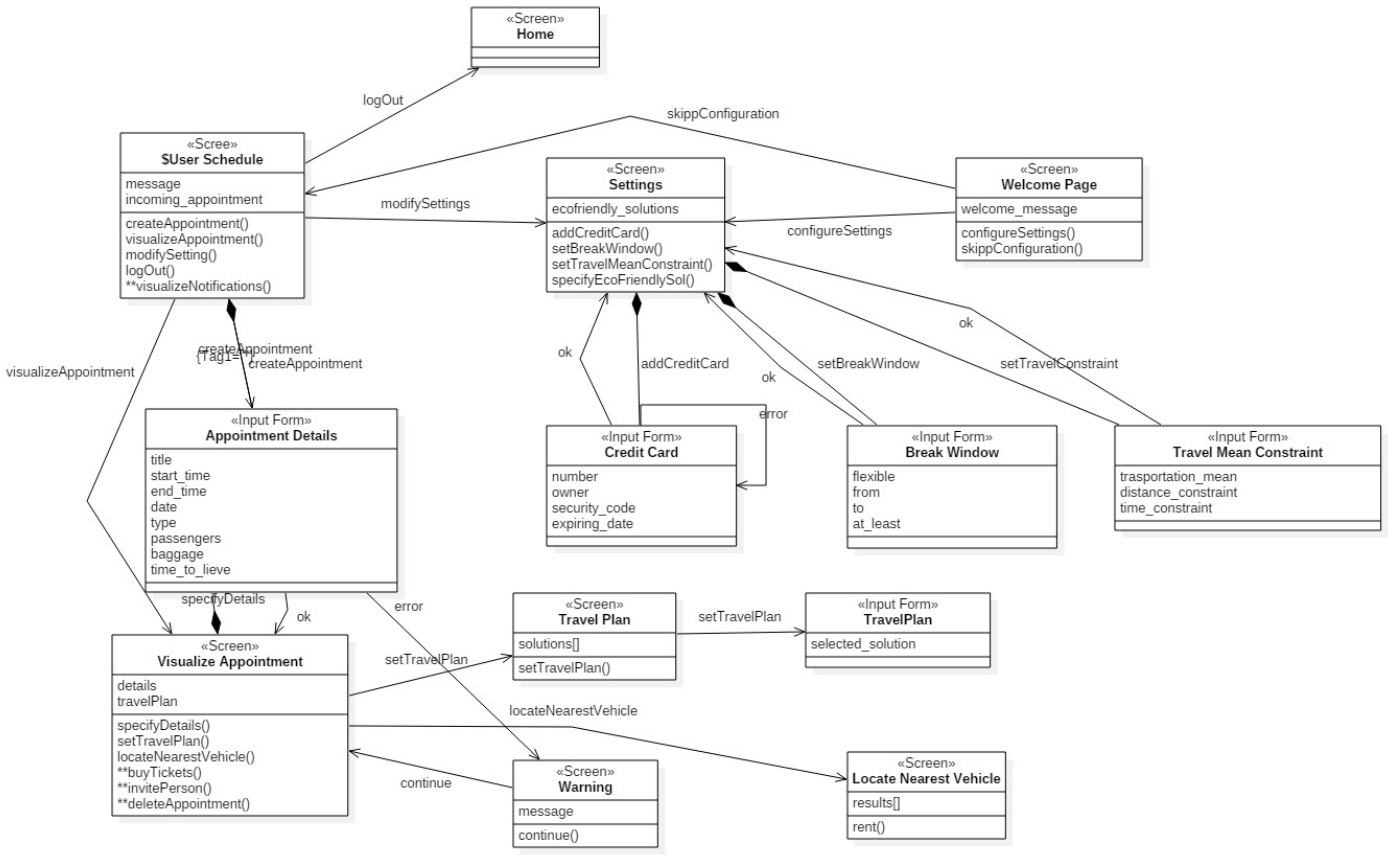


Figure 13: UX Diagram - User UX

# 5 Requirements Traceability

# 6 Implementation, Integration and Test Plan

# 7 Effort Spent and Team Work

*This section will be filled at the end of the work*

**Cassarino Pietro Total hours:** ****, **Hours working alone:** ****

**Salaris Mirko Total hours:** ****, **Hours working alone:** ****

**Ventrella Piervincenzo Total hours:** ****, **Hours working alone:** ****

# 8 References

## 8.1 Software and Tools

- LaTeX for typesetting document
- TeXstudio as LaTeX IDE
- GitHub for version control and team work
- StarUML for UML models

## 8.2 Reference Documents