DESIGN DOCUMENT
MeteoCal

Daniele Marangoni, Matteo Montalcini,
Daniele Moro

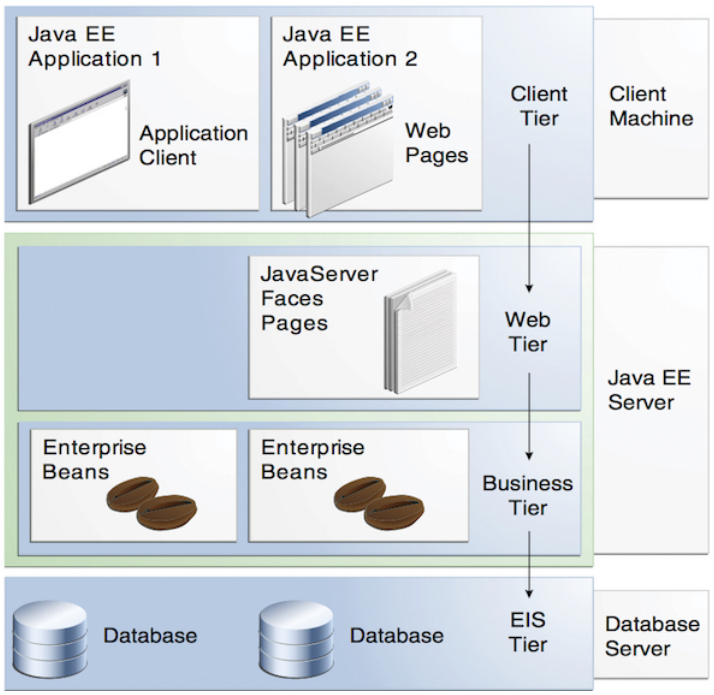
07/12/2014

Contents

1. Architecture Description

1.1 J2EE Architecture Overview

Thinking about the type of application we have to develop, we decide to implement it using J2EE Architecture. We start explaining its main features.



The Java EE platform has a multitier architectural model, divided into:

- **Client-Tier**, running on the client machine.
- **Web-Tier**, running on the Java EE Server.
- **Business-Tier**, running on the Java EE Server.
- **Enterprise Information System (EIS)**, running on the Database Server.

1.2 Tier Description

Client Tier: it contains a Web Browser consisting of Dynamic Web pages generated by components present in the Web Tier. The Client Tier doesn't do query on the Database, because they are designated at Server components. The Client Tier also has an Application Client, executed on a client machine. This is the layer of the final user, who can interface with the application through a web browser.

Web Tier: it contains the Servlets (Java classes that process dynamically requests and build responses). This tier receives requests from the Client Tier and, after a collection of data into the Business Tier, it forwards collected data to the Client Tier, eventually formatted.

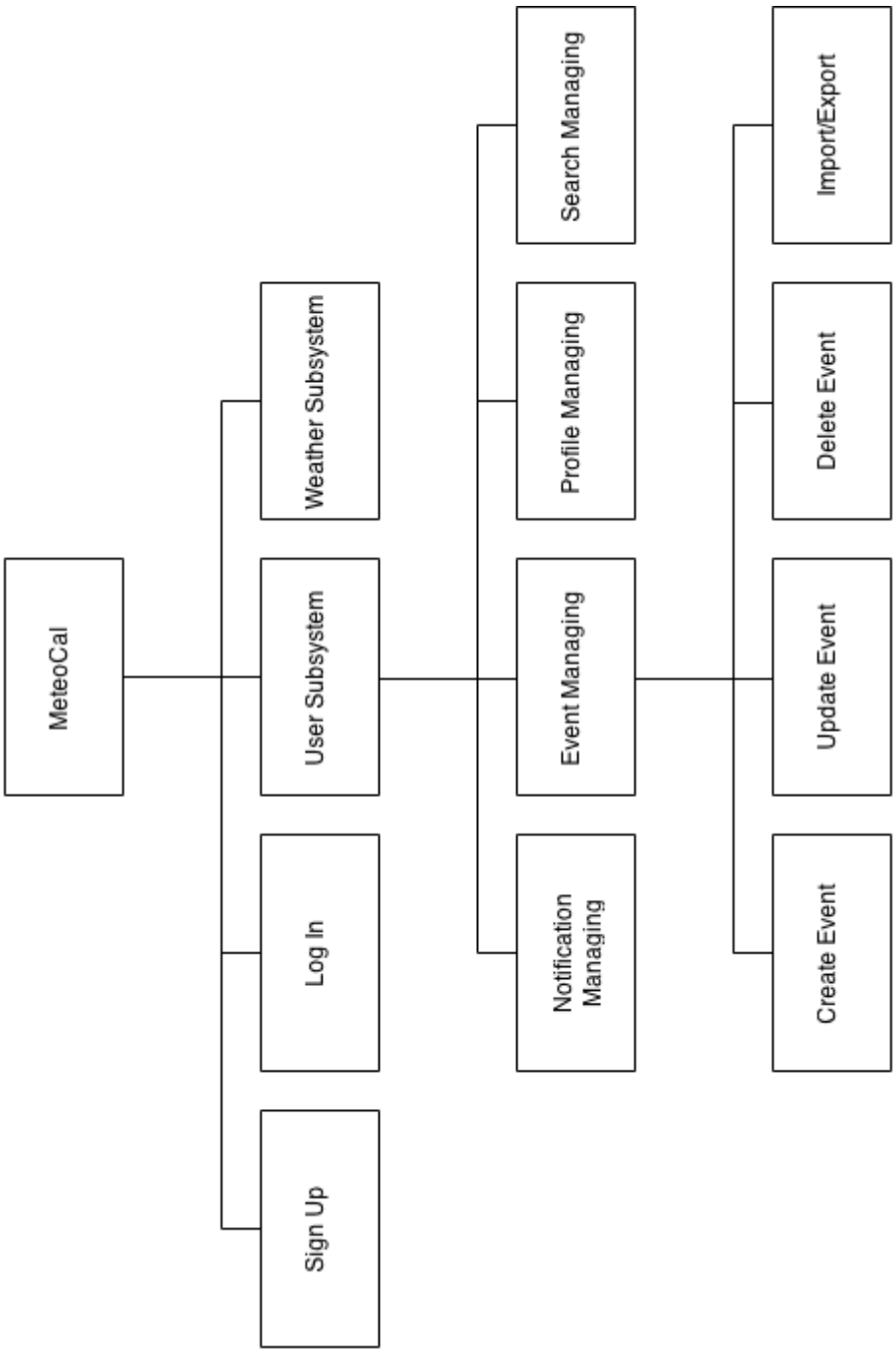
Business Tier: it contains the Java Beans, in which we have the business logic of the application.

EIS Tier: the Enterprise Information System that contains the data source. In our application the data source is a database which has the task of storing all the relevant data.

1.3 Identifying Sub-Systems

We decide to describe the structure that the system will assume, in order to clarify how our systems functionalities will be divided. To do this, we realize a schema that represents the composition of our main system. Starting from the top, we have the main system, that is splitted into many sub-systems which provide different functionalities.
We separate our system into these sub-systems:

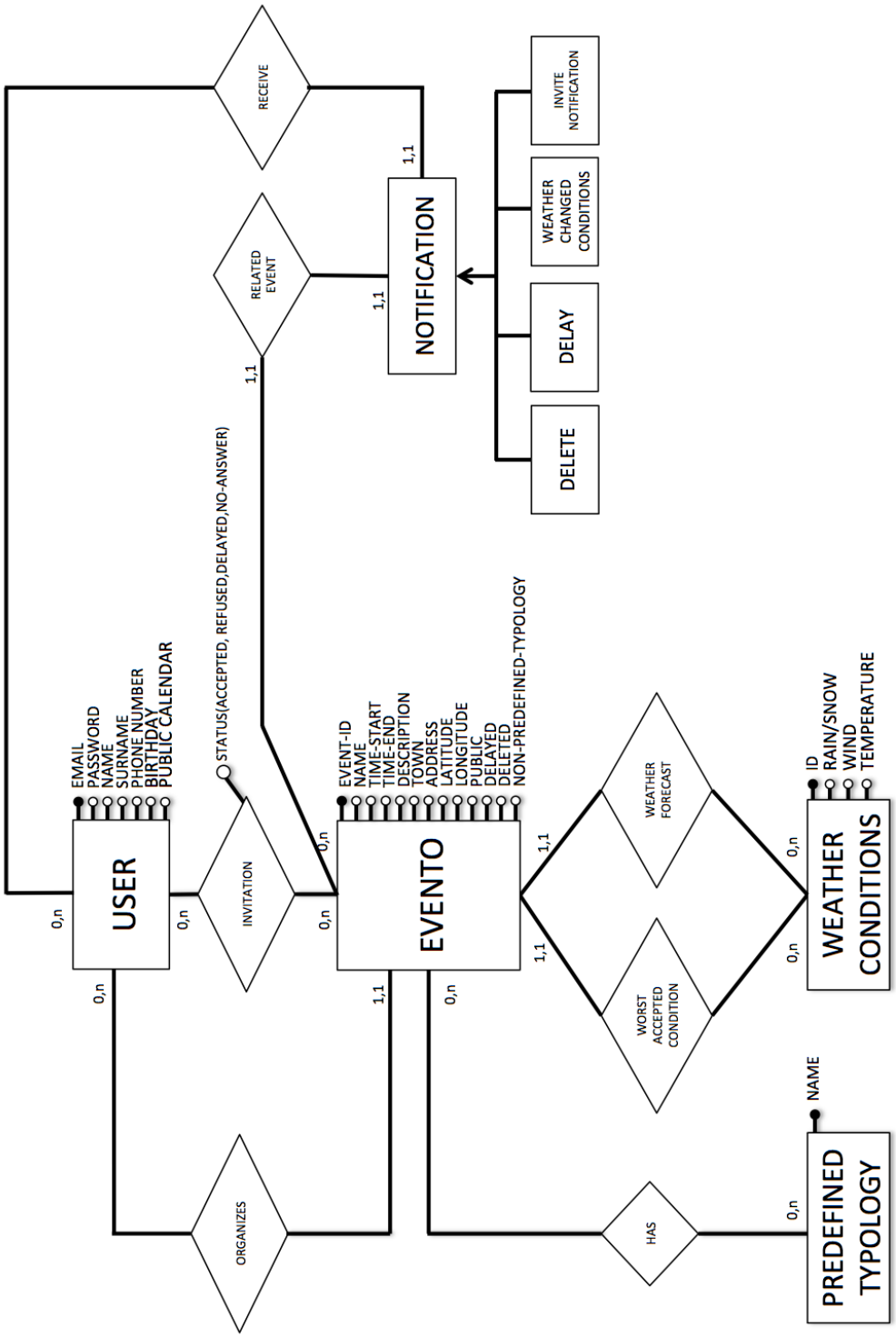
- Sign Up
- Log In
- User
  - Notification Managing
  - Event Managing
  - Profile Managing
  - Search Managing
- Weather


MeteoCal
User Subsystem
Weather Subsystem
Log In
Sign Up
Event Managing
Profile Managing
Search Managing
Notification Managing
Update Event
Delete Event
Import/Export
Create Event

2. Persistent Data Management

2.1 Conceptual Design

MeteoCal data are stored into a relational database.
First of all, we start thinking about the conceptual design: we have to decide which data we want to store and which relations characterized them. So we produce initially a representation of the design of our database using an Entity-Relationship Diagram, reported in the next page.


USER
EMAIL
PASSWORD
NAME
SURNAME
PHONE NUMBER
BIRTHDAY
PUBLIC CALENDAR
INVITATION
STATUS(ACCEPTED, REFUSED,DELAYED,NO-ANSWER)
EVENTO
EVENT-ID
NAME
TIME-START
TIME-END
DESCRIPTION
TOWN
ADDRESS
LATITUDE
LONGITUDE
PUBLIC
DELAYED
DELETED
NON-PREDEFINED-TYPOLOGY
RECEIVE
RELATED EVENT
NOTIFICATION
INVITE NOTIFICATION
WEATHER CHANGED CONDITIONS
DELAY
DELETE
WEATHER FORECAST
WORST ACCEPTED CONDITION
WEATHER CONDITIONS
ID
RAIN/SNOW
WIND
TEMPERATURE
ORGANIZES
HAS
PREDEFINED TYPOLOGY
NAME
0,n
1,1

In our system, we obviously have to store all the data about user: in fact, some of these data are fundamental in order to exploit MeteoCal functionalities (e.g., the email or the information about the visibility of the calendar), while the other ones are important because, through them, other users can know more information about another one. So we create an entity "User", with all the necessary attributes.

Another indispensable part is represented by the events: to store events's information, we add an entity "Event": some of its attributes refer to event information which has to be established by the organizer of the event, while other ones are strictly connected to the state of an event (which can be postponed or deleted).

Each event has exactly one organizer, and this fact is highlighted by the relation "Organizes", between the two entities "Event" and "User".

It's also important to memorize all the invitations: we decide to represent them through a relationship among "User" and "Event", obviously called "Invitation". In this way, we can underline its most important characteristics: an invitation is sent from the organizer of the event (we can find this information exploiting "Event" and the relation "Organizer") to other users and it is related to an event. It has also a status, which highlights if the invitation has been accepted, refused, if it has no answer yet or if it is an invitation for an event which has been delayed.

For each event, the organizer, upon creation, specifies the worst weather conditions he would accept for his event: this information must be stored to check, day by day, if the weather forecasts for the event day are acceptable or not. In case of a negative response, a notification is sent to the organizer. So we decide to introduce an entity called "Weather Conditions", in which we memorize information about the acceptable rain/snow, wind, temperature. We underline the connection between the Event and the particular Weather Condition selected as the worst acceptable by introducing a relationship "Worst Accepted Condition" between the two entities.

The creator has to select also a typology for the event: he can choose among a predefined set of typology, that we store using the entity "Predefined Typology", or he can write another typology. Finally, it's important to memorize also the notifications: we introduce the entity "Notification", and four sub-entities, "Delete Notification", "Delay Notification", "Weather Changed Notification", "Invite Notification", which correspond to all kinds of notification a user can receive. We decide to introduce a hierarchy because every kind of notification has the same features of the others, except for the type. In fact, every notification is sent to a user and has a related event, feature made visible through the introduction of the relationships "Receives" (between User and Notification) and "Related Event" (between Notification and Event).

2.2 Logical Design

In order to represent better the database structure of our system, it's not sufficient to represent the design of our database as we have done above, but it is necessary to perform a translation from the ER Diagram to the Logical Model. But we can't do it directly, because we previously need to perform some transformations.

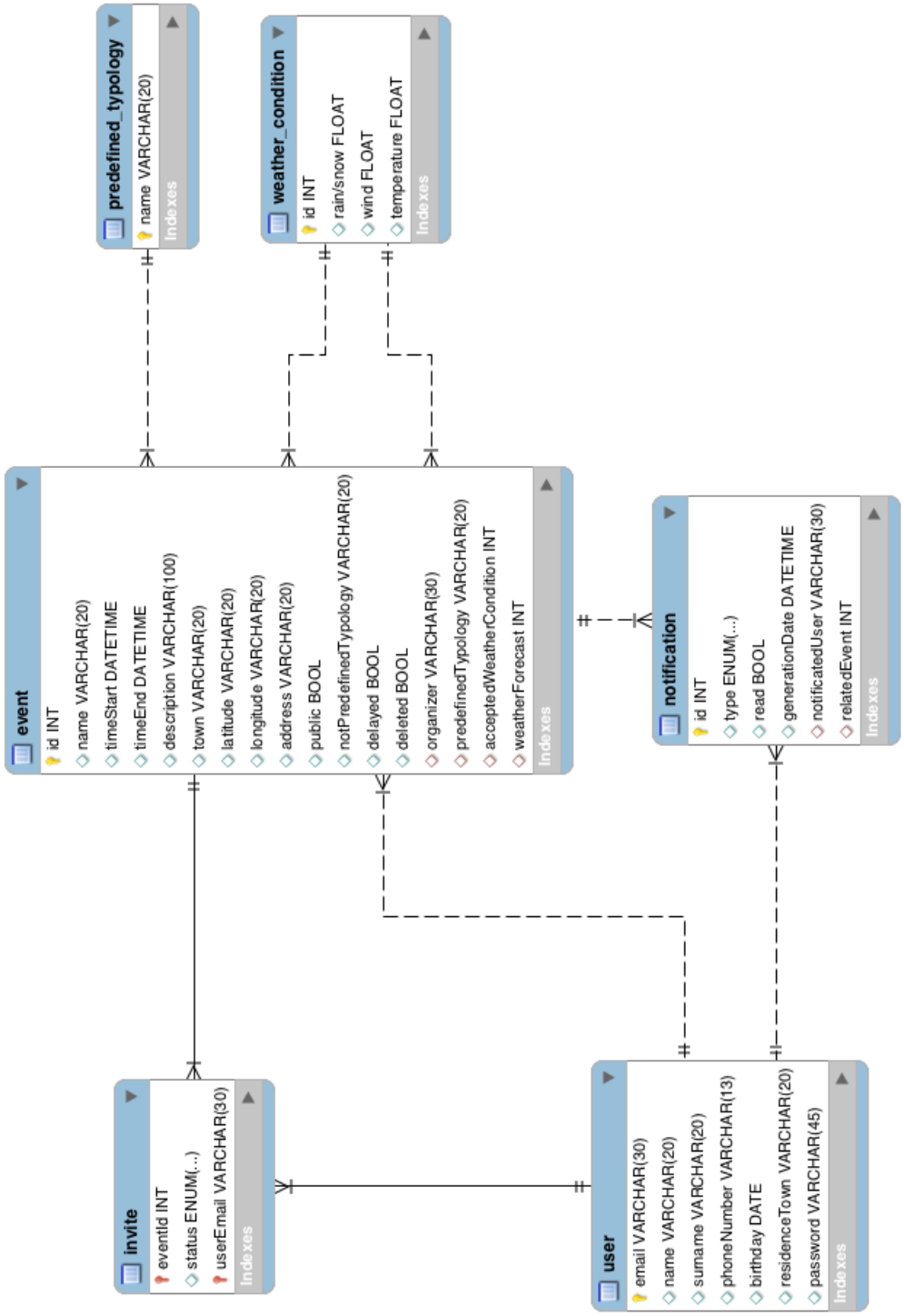
2.2.1 ER Restructuration

We have a hierarchy between Notification and its four sub-classes: as we have seen before, the five entities are very similar, so it's possible to merge Delete Notification, Delay Notification, Weather Changed Notification and Invite Notification into Notification. Doing this change, we have only one entity: we only have to introduce an attribute "type" to differentiate from a kind of notification to another.

Since the hierarchy was total and exclusive, the attribute "type" will assume one of the following value: "delete", "delay", "weather changed", "invite".

2.2.2 Translation to Logical Model

1. Relation "Receives" between Notification and User is translated inserting a foreign key called "notificatedUser" into the table Notification.
2. Relation "Related Event" between Notification and Event is translated inserting a foreign key named "relatedEvent" into the table Notification.
3. Relation "Organizes" between User and Event is translated inserting a foreign key called "organizer" into the table Event.
4. Relation "Invitation" has 0:N cardinalities on both edges, so we create a new table Invite with the foreign keys userEmail and eventId, adding also the relationship attribute status.
5. Relation "Has" between Event and Predefined Typology is translated inserting a foreign key, called "predefinedTypology", into the table Event.
6. Relation "Worst accepted condition" between Event and Weather Conditions is translated inserting a foreign key "acceptedWeatherCondition" into the table Event.
7. Relation "Weather Forecast" between Event and Weather Conditions is translated inserting an attribute "weatherForecast" into the table Event.


predefined_typology
name VARCHAR(20)
Indexes
weather_condition
id INT
rain/snow FLOAT
wind FLOAT
temperature FLOAT
Indexes
event
id INT
name VARCHAR(20)
timeStart DATETIME
timeEnd DATETIME
description VARCHAR(100)
town VARCHAR(20)
latitude VARCHAR(20)
longitude VARCHAR(20)
address VARCHAR(20)
public BOOL
notPredefinedTypology VARCHAR(20)
delayed BOOL
deleted BOOL
organizer VARCHAR(30)
predefinedTypology VARCHAR(20)
acceptedWeatherCondition INT
weatherForecast INT
Indexes
notification
id INT
type ENUM(...)
read BOOL
generationDate DATETIME
notificatedUser VARCHAR(30)
relatedEvent INT
Indexes
invite
eventId INT
status ENUM(...)
userEmail VARCHAR(30)
Indexes
user
email VARCHAR(30)
name VARCHAR(20)
surname VARCHAR(20)
phoneNumber VARCHAR(13)
birthday DATE
residenceTown VARCHAR(20)
password VARCHAR(45)
Indexes

The final model has the following physical structure:

USER (email, password, name, surname, phoneNumber, birthday, residenceTown)

EVENT (id, name, timeStart, timeEnd, description, town, address, latitude, longitude, public, delayed, deleted, notPredefinedTypology*, *predefinedTypology*, *organizer*, *acceptedWeatherCondition*, *weatherForecast*)

INVITE (*eventId*, *userEmail*, status(ENUM))

WEATHER_CONDITION (id, rain/snow, wind, temperature)

PREDEFINED_TYPOLOGY (name)

NOTIFICATION (id, type(ENUM), *notificatedUser*, *relatedEvent*, read, generationDate)

The primary keys of each table are underlined, while the foreign keys are written in Italic. It is important to point out that the symbol " * " associated to the attribute notPredefinedTypology of the table Event means that the value of that attribute could be present, but could also be NULL: in fact, it is always NULL, except in the case in which the user wants to add a new typology, which is not present in the predefined list, and he does it by selecting the predefined Typology "other" and inserting the new typology.

3. User Experience

In order to better explain how the system would be used, we decide to describe the User Experience given by our system to its users.

We used a class diagram with the following stereotypes:

- *screen* : it represents a page. In each screen we have inserted the operations that can be invoked by performing some actions on the page. A page can be composed of some sub-screens.
- *screen compartment* : it represents a sub-screen. In fact, sometimes it is possible to split a page in different parts and, through "screen compartment", we can highlight every single part of the page, by analyzing it separately from the rest. Then we put together all them to create the complete page, underlining which "screen compartment" are contained in the "screen" by the composition operator.
- *input form* : it stands for some input fields, that can be fulfilled by a user. It is necessary every time a user has to insert some data (sign up, log in, event creation, ...) to collect them. This information will be submitted to the system, through a click on a button.
- class with no stereotypes: it represents an entity with its own attributes.

3.1 Homes

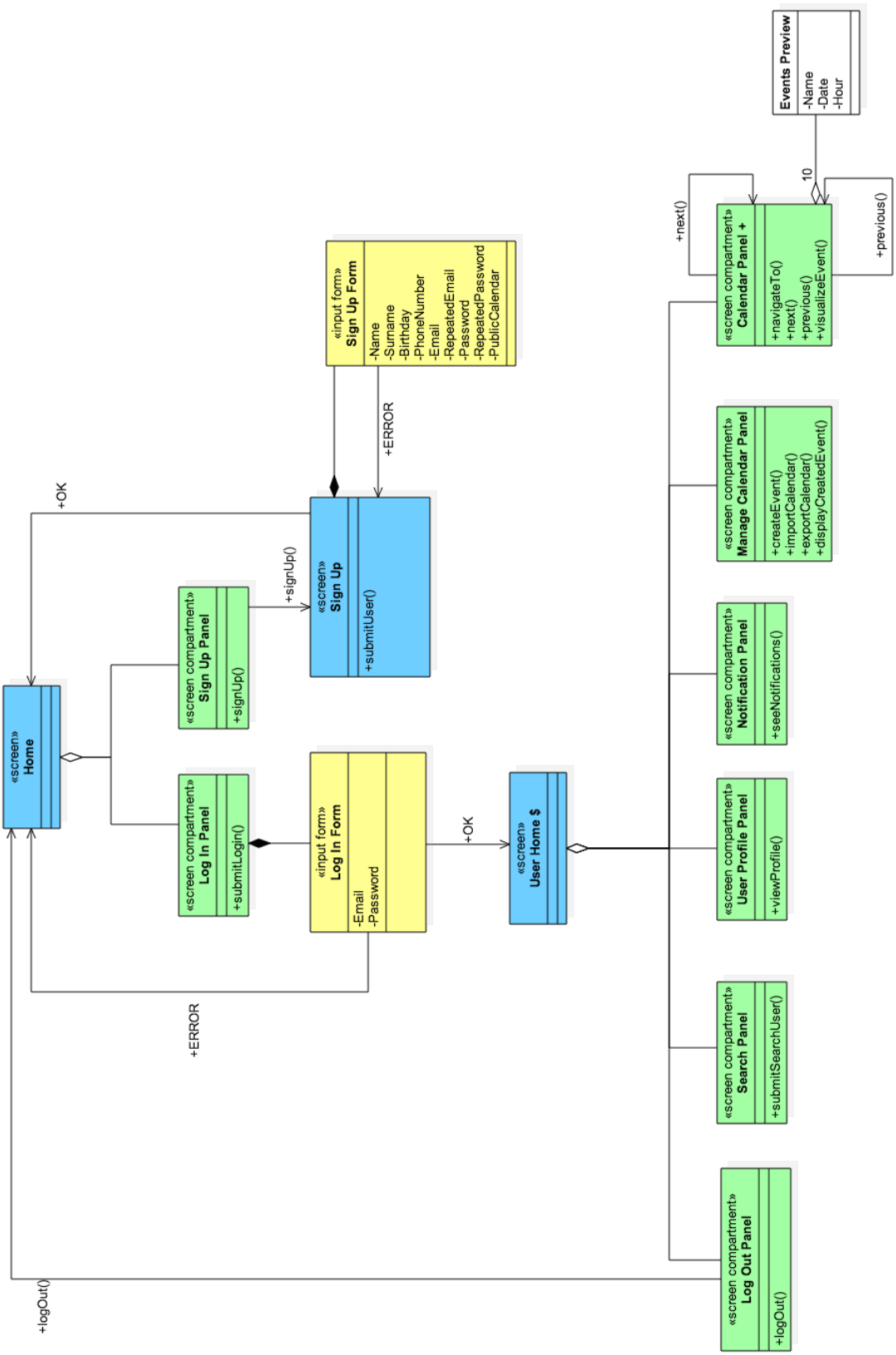
Below there's the UX-diagram that represents the general schema of our system. We now analyze how a user can interact with the system through the different pages.

There's the Home page which we have decided to split into two sections:

- **Log In:** part of the Home page from which the user can perform the log-in into the system. It has an input form that has to be fulfilled by the user. If data are correct, then the user will be redirected to his Home page based on user's data, otherwise the user will be redirected another time to the Home page.
- **Sign up:** part of the Home page from which the user can sign up into the system. It has an input form that has to collect the user's data to save them into the database in order to guarantee the future log in of the user and the permanence of his data.

Once a user has logged in, he's redirected into the User Home page that is reachable from every page of the system after the log in and for this reason it has the '$' landmark. We decide to split the description of the User Home page in some sections that highlight the operations that can be performed by a user once he has logged in:

- **Log Out:** a part of the User Home that allows a user to log out.
- **Search:** a part of the User Home through which a user can search for other users.
- **User Profile:** a part of the User Home in which a user can see his own information and he can also modify them.
- **Notification:** a part of the User Home in which a user can see the notifications related to events.
- **Manage Calendar:** a part of the User Home in which a user use can create an event, import/export calendar and display created event in order to see or modify event information.
- **Calendar:** a part of the User Home in which a user can see information about the events he is going to participate to.


«input form»
Sign Up Form
-Name
-Surname
-Birthday
-PhoneNumber
-Email
-RepeatedEmail
-Password
-RepeatedPassword
-PublicCalendar
+ERROR
+OK
«screen»
Sign Up
+submitUser()
+signUp()
«screen compartment»
Sign Up Panel
«screen»
Home
«screen compartment»
Log In Panel
+submitLogin()
«input form»
Log In Form
-Email
-Password
+OK
«screen»
User Home $
+ERROR
+logOut()
«screen compartment»
Log Out Panel
+logOut()
«screen compartment»
Search Panel
+submitSearchUser()
«screen compartment»
User Profile Panel
+viewProfile()
«screen compartment»
Notification Panel
+seeNotifications()
«screen compartment»
Manage Calendar Panel
+createEvent()
+importCalendar()
+exportCalendar()
+displayCreatedEvent()
«screen compartment»
Calendar Panel +
+navigateTo()
+next()
+previous()
+visualizeEvent()
+next()
+previous()
10
Events Preview
-Name
-Date
-Hour

3.2 Profile and Search Managing

This part of the UX Diagram permits to understand how the user can visualize and manage/update his Profile Information.

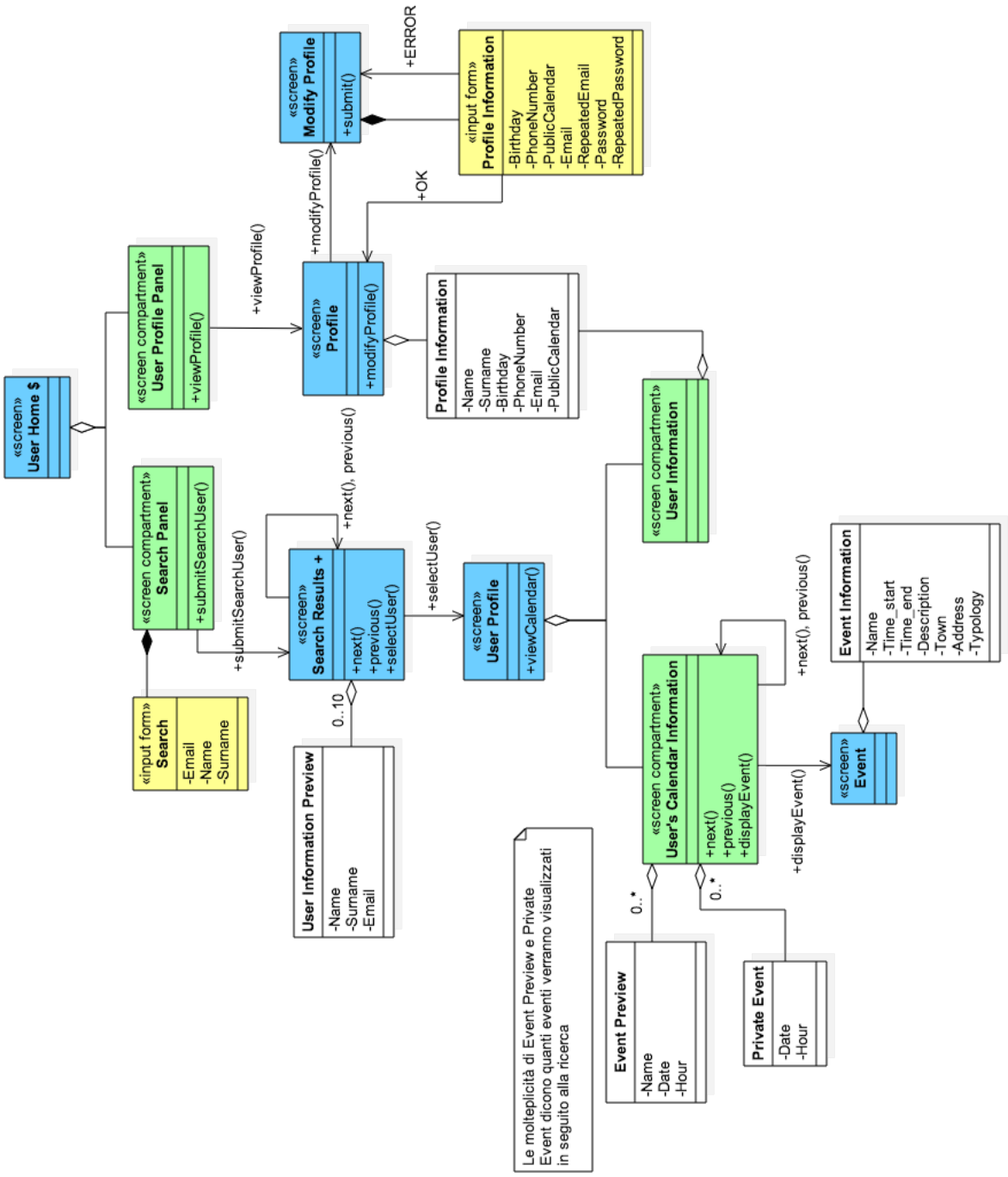
Once the user is logged in the system he's redirected to the User Home page. In this context, we focus our attention on two of the User Home page sections: User Profile and Search.

From the User Profile section, the user can access to his profile and then view or modify his own information. All the information that a user can modify are specified in the input form.

From the Search section, the user can search for other users through Name, Surname or email. After clicking the search button he is redirected into another page that contains the results (if there are some users whose information matches with the inserted one).

In the Search Result page, the user can see the main information (Name, Surname and email) about ten users whose data match with the inserted ones. He can load also other previous or next results respect to the ten that are visualized and then he can click on one of this to see the user's profile in a new page (e.g. User Profile page). This new page contains two sections:

- one for user's information, that shows all the public information of the searched user;
- one for the user's calendar information, that shows information about events at which he takes part. It appears only if the user's calendar is public. It is important to point out that for events classified as "public" name, data and hour range are visualized while for events classified as "private", only the date and the hour range of the event are visible.
  In order to see more information about a public event, it is also possible to click on it to see all the related data in the Event page.

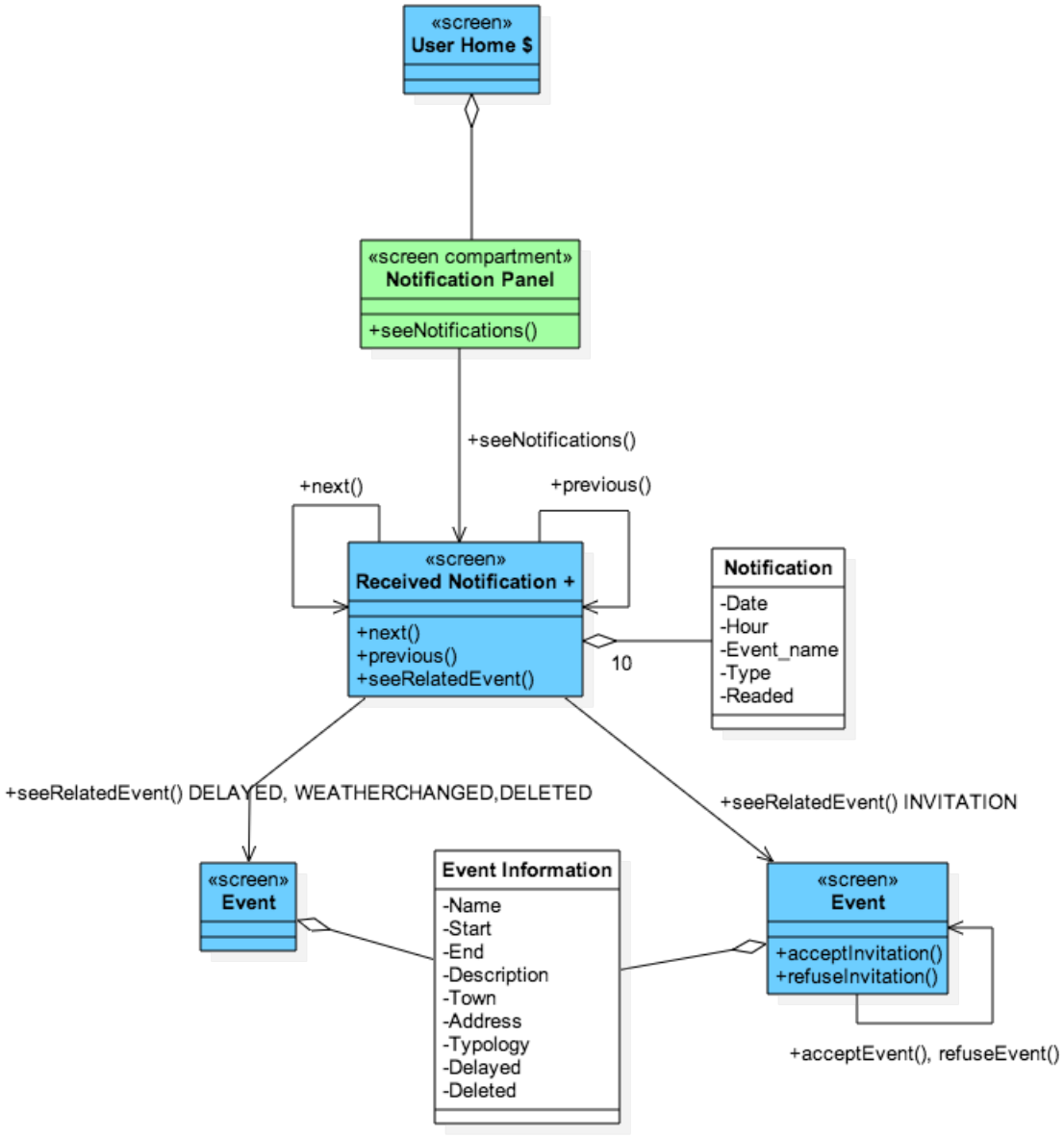

«screen» User Home $
«screen compartment» User Profile Panel
+viewProfile()
+viewProfile()
«screen» Modify Profile
+submit()
+ERROR
«input form» Profile Information
-Birthday
-PhoneNumber
-PublicCalendar
-Email
-RepeatedEmail
-Password
-RepeatedPassword
+modifyProfile()
+OK
«screen» Profile
+modifyProfile()
Profile Information
-Name
-Surname
-Birthday
-PhoneNumber
-Email
-PublicCalendar
«screen compartment» User Information
«screen compartment» Search Panel
+submitSearchUser()
+submitSearchUser()
+next(), previous()
«screen» Search Results +
+next()
+previous()
+selectUser()
+selectUser()
«screen» User Profile
+viewCalendar()
«input form» Search
-Email
-Name
-Surname
0..10
User Information Preview
-Name
-Surname
-Email
«screen compartment» User's Calendar Information
+next()
+previous()
+displayEvent()
+displayEvent()
+next(), previous()
Event Information
-Name
-Time_start
-Time_end
-Description
-Town
-Address
-Typology
«screen» Event
0..*
Event Preview
-Name
-Date
-Hour
0..*
Private Event
-Date
-Hour
Le molteplicità di Event Preview e Private
Event dicono quanti eventi verranno visualizzati
in seguito alla ricerca

3.3 Notification Managing

This part of the UX Diagram permits to understand how the user can visualize notifications about events depending on the type of the notification.

Once the user is logged in the system he's redirected to the User Home page. A part of this User Home page is the Notification section. By clicking on the notification button, the user is redirected into the Received Notification page, that allows to see new and previous notifications. In this page, at most 10 notifications are preloaded with the related event main information: if the user wants to see the complete event page, he has to click on the notification. The user can decide to load other previous or next notifications respect to the ten that are visualized.

Depending on the type of the notification, the system provides to redirect the user on the related page. If the type of the notification is "DELAYED", "WEATHERCHANGED" or "DELETED", the user is redirected in a page that shows only the information about the related event. If the type of the notification is "INVITATION" the user is redirected in another page that allows, in addiction to the information about the related event, to accept or refuse the event invitation.


«screen»
User Home $
«screen compartment»
Notification Panel
+seeNotifications()
+seeNotifications()
+next()
+previous()
«screen»
Received Notification +
+next()
+previous()
+seeRelatedEvent()
10
Notification
-Date
-Hour
-Event_name
-Type
-Readed
+seeRelatedEvent() DELAYED, WEATHERCHANGED,DELETED
+seeRelatedEvent() INVITATION
«screen»
Event
Event Information
-Name
-Start
-End
-Description
-Town
-Address
-Typology
-Delayed
-Deleted
«screen»
Event
+acceptInvitation()
+refuseInvitation()
+acceptEvent(), refuseEvent()

3.4 Event Managing

This part of the UX Diagram permits to understand how the user can manage his calendar and which actions can be done on it and on the events.

Once the user is logged in the system he's redirected to the User Home page. Now we focus on two sections of this page: Manage Calendar and Calendar.

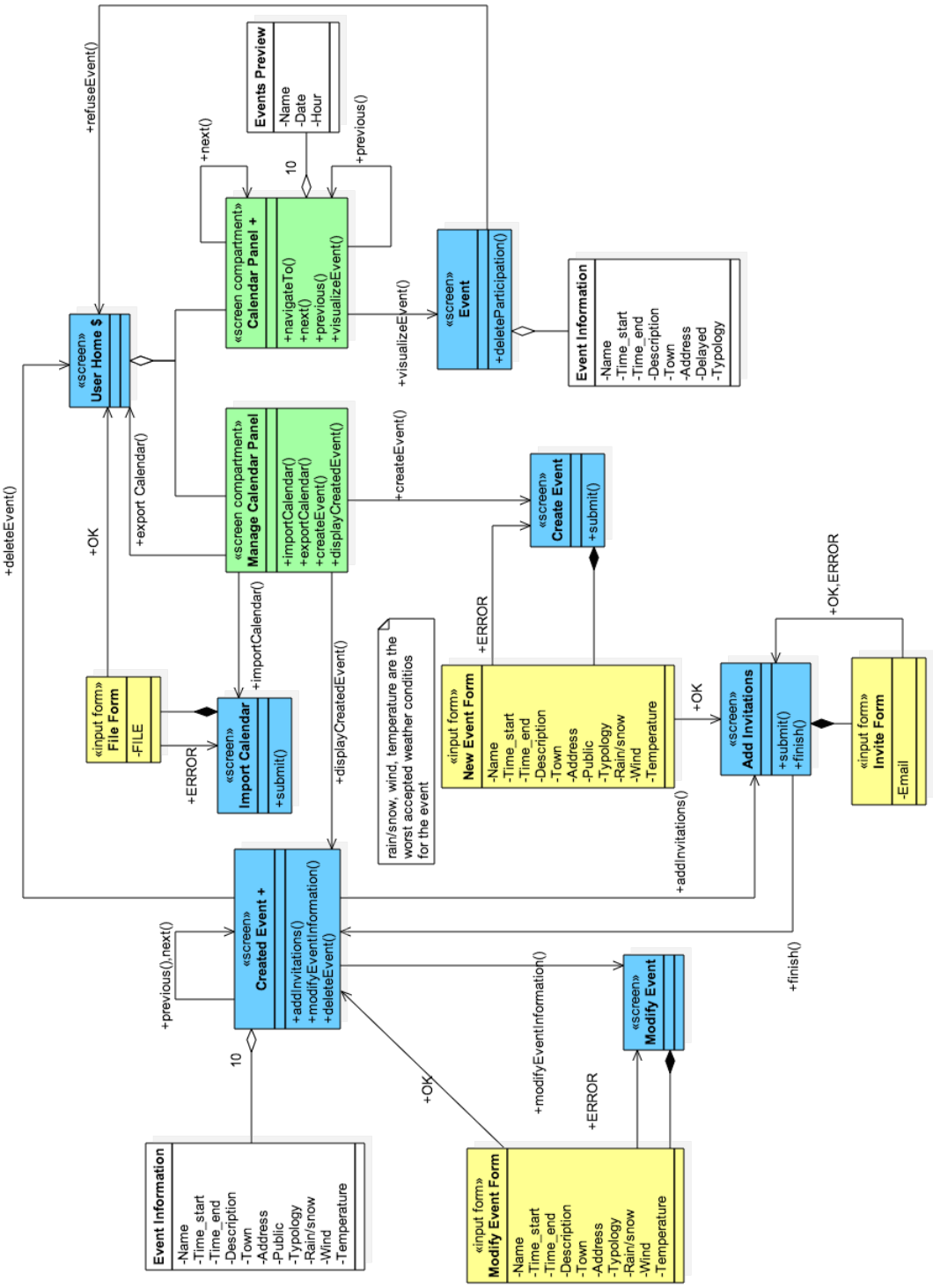
From the Calendar section, the user can see the main information about accepted events. In this section are preloaded 10 events (also in this case, the user can decide to load other previous or next events respect to the ten that are visualized). To see all the information of one of the listed events, the user has to click on it: after performing this action, he is redirected in the Event page in which he can also decide to delete his participation to the selected event.

From the Manage Calendar section, the user can perform many actions. He can export his own calendar: in this case the download of the file will start and the user is redirected to the User Home page. He can also import an existing calendar by uploading a file in a new Import page. If the file is valid, the import is performed and the user is redirected to the User Home page, otherwise he is redirected another time to the Import page and he has the possibility to upload another time the file, obviously after having modified it or his calendar on MeteoCal, in order to make them compatible for that operation. From Manage Calendar section the user can perform other two actions: create an event and see his created events. We decide to dedicate a section to keep track of the created event to facilitate user in managing his own events.

If the user decides to create an event, by clicking on "Create event" button, he's redirected into the Create Event page, in which he has to fulfill an input form with the information about the event. If all the inserted information are ok, by clicking on the "Confirm" button, the user is redirected into the Add Invitations page, in which he can invite other users to the event by inserting their email. By clicking on the "Finish" button, the user is redirected into the Created Event page, in which the user can see a list of his created events. By default, in this page are preloaded at most ten events, but the user can decide to load other previous or next events respect to the ten that are visualized. In this page it is possible to see event's information and also to add invitations to an event and delete or modify events.

If the user wants to add invitations for an event, he's redirected on the same page described before: Add Invitation page.

If the user wants to modify the information related to a created event, he's redirected to the Modify Event page, in which there's a fulfilled input form that the user can modify. If all is ok, the user is redirected into the Created Event page, otherwise he has to modify again the input form with valid information.


«screen»
User Home $
«screen compartment»
Calendar Panel +
+navigateTo()
+next()
+previous()
+visualizeEvent()
Events Preview
-Name
-Date
-Hour
10
+next()
+previous()
+refuseEvent()
«screen»
Event
+deletePartication()
+visualizeEvent()
Event Information
-Name
-Time_start
-Time_end
-Description
-Town
-Address
-Delayed
-Typology
«screen compartment»
Manage Calendar Panel
+importCalendar()
+exportCalendar()
+createEvent()
+displayCreatedEvent()
+createEvent()
+export Calendar()
+OK
+deleteEvent()
«screen»
Create Event
+submit()
«input form»
File Form
-FILE
«screen»
Import Calendar
+submit()
+importCalendar()
+ERROR
+displayCreatedEvent()
rain/snow, wind, temperature are the worst accepted weather conditios for the event
«input form»
New Event Form
-Name
-Time_start
-Time_end
-Description
-Town
-Address
-Public
-Typology
-Rain/snow
-Wind
-Temperature
+ERROR
+OK
+addInvitations()
«screen»
Add Invitations
+submit()
+finish()
+OK,ERROR
«input form»
Invite Form
-Email
«screen»
Created Event +
+addInvitations()
+modifyEventInformation()
+deleteEvent()
+previous(),next()
10
+OK
+modifyEventInformation()
+ERROR
+finish()
«screen»
Modify Event
Event Information
-Name
-Time_start
-Time_end
-Description
-Town
-Address
-Public
-Typology
-Rain/snow
-Wind
-Temperature
«input form»
Modify Event Form
-Name
-Time_start
-Time_end
-Description
-Town
-Address
-Typology
-Rain/snow
-Wind
-Temperature

4. BCE Diagrams

We decide to realize a further design schema of MeteoCal, using the Boundary-Control-Entity pattern: there is a precise motivation for this choice. In fact, there is a great similarity between Boundary-Control-Entity pattern and Model-View-Controller pattern, which is based on the work separation between different software components, which interpret three different roles (as suggested by the name of the pattern):

- Model: it deals with the data, in the sense that it gives the necessary methods to access data which are useful for the application;
- View: it visualizes data which are content in the model and it deals with the interaction with users and agents;
- Controller: it receives commands from the user (generally, through the View) and it applies them, modifying the states of the other two components (e.g., changing data stored in Model and data visualized by the View).

The similarity is due to the fact that there is a correspondence between views and boundaries, controllers and controls, models and entities.

We realize boundaries starting from the Use Cases Diagram provided in the RASD and the screens of the User Experience Diagrams. All the methods highlighted in the screens are inserted in the exact boundaries and, for each screen, we have also inserted a showScreenName() method (while there isn't a show method for the screen compartment: they are only parts of the screen, so it is not necessary to insert a specific show method for each of them, because they will never appear alone).

We decide to realize more than one BCE diagrams, because, by splitting the entire system in more subsystems, it is possible to analyze better the specific aspects and also the representations will be clearer.

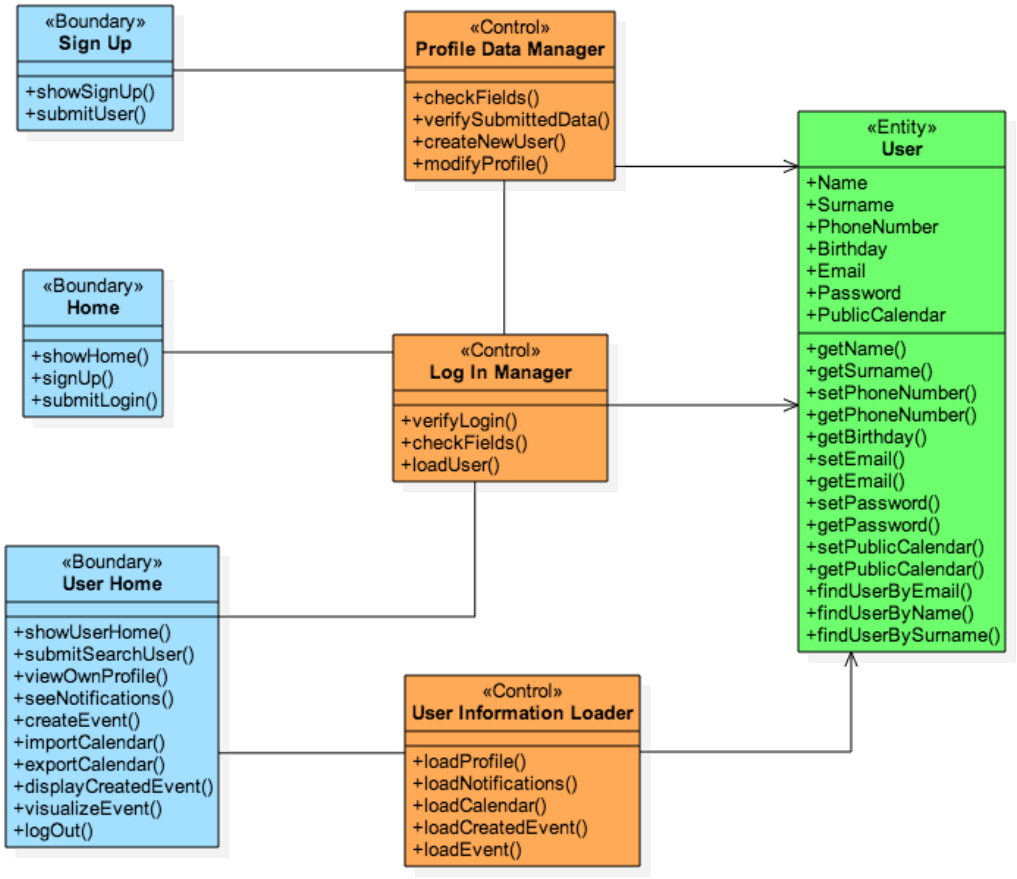
4.1 Sign Up and Log In

There are three different boundaries:

- **Home:** it is the first page a user can see and, from this page, he can insert his data to log in, or he can go to the Sign Up page.
- **Sign Up:** in this page, there are all the fields that a user has to fulfill in order to register himself. After the data insertion, he can submit the information to the system.
- **User Home:** after the log in, the user is redirected to this page. From here, the user can exploit, directly or indirectly, all the functionalities provided by MeteoCal (search for a user, view own profile, see notification, import/export calendar, create or manage events). Obviously, this boundary appears in all the BCE Diagrams, so, for the sake of conciseness, its description is reported only here.

We use different controllers:

- **Profile Data Manager:** this controller has the purpose of check the data inserted upon sign up, but also when the user decides to modify his profile. In fact all the mandatory fields has to be fulfilled and it is necessary to check all constraints about data (e.g., every user must have a different email).
- **Log In Manager:** this controller manages the log in phase: it checks if all the fields (email and password) are fulfilled and, in the positive case, has to verify if there is a user in the system whose email and password correspond to the inserted ones. If such a user exists, the Log In Manager can load the exact user, finding him through his email and password.
- **User Information Loader:** this controller must load all the information related to the user which has just performed the log in. In particular, he has to find in the system all the data concerning the user's profile, notifications, calendar. We have to point out that it loads the calendar by searching for only the events that the user has created (and to which, obviously, he is going to participate) and the events for which he has accepted the invitation. These two different sets of events are selected throw the two operations loadCreatedEvent() and loadEvent().


«Boundary»
Sign Up
+showSignUp()
+submitUser()
«Control»
Profile Data Manager
+checkFields()
+verifySubmittedData()
+createNewUser()
+modifyProfile()
«Entity»
User
+Name
+Surname
+PhoneNumber
+Birthday
+Email
+Password
+PublicCalendar
+getName()
+getSurname()
+setPhoneNumber()
+getPhoneNumber()
+getBirthday()
+setEmail()
+getEmail()
+setPassword()
+getPassword()
+setPublicCalendar()
+getPublicCalendar()
+findUserByEmail()
+findUserByName()
+findUserBySurname()
«Boundary»
Home
+showHome()
+signUp()
+submitLogin()
«Control»
Log In Manager
+verifyLogin()
+checkFields()
+loadUser()
«Boundary»
User Home
+showUserHome()
+submitSearchUser()
+viewOwnProfile()
+seeNotifications()
+createEvent()
+importCalendar()
+exportCalendar()
+displayCreatedEvent()
+visualizeEvent()
+logOut()
«Control»
User Information Loader
+loadProfile()
+loadNotifications()
+loadCalendar()
+loadCreatedEvent()
+loadEvent()

4.2 Search for a User

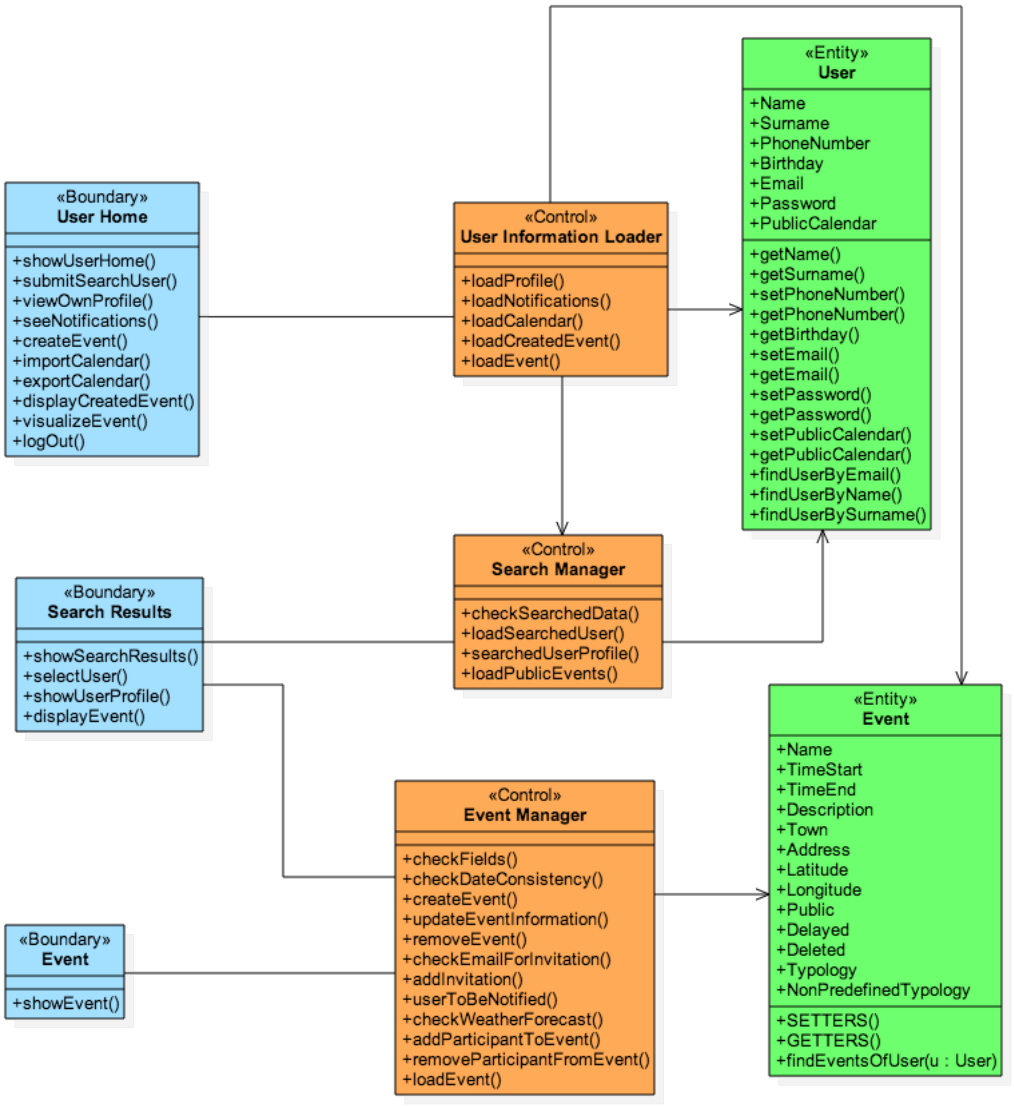
Here there are two new boundaries:

- **Search Results:** it allows to perform the desired searches;
- **Event:** it shows the features of a selected event.

The search functionality is managed by the **Search Manager**: it has to check the inserted data (e.g., since the user can be searched through their email, name or surname it is not possible to have many of the punctuation...) and then he has to verify if there are, in the system, users whose data correspond to the inserted ones. There can be more than one result (for example, if I search by name and not inserting also the surname), so it has to take into account all the possible matches, which will be displayed on the Search Results boundary.

The Search Manager has also to load the user's information, if he is selected from the results page, and in that case it also lists the public events present in his calendar.

There is also the introduction of another controller called **Event Manager**: it is connected to everything related to events and it will be described in detail later.


«Boundary»
User Home
+showUserHome()
+submitSearchUser()
+viewOwnProfile()
+seeNotifications()
+createEvent()
+importCalendar()
+exportCalendar()
+displayCreatedEvent()
+visualizeEvent()
+logOut()
«Control»
User Information Loader
+loadProfile()
+loadNotifications()
+loadCalendar()
+loadCreatedEvent()
+loadEvent()
«Entity»
User
+Name
+Surname
+PhoneNumber
+Birthday
+Email
+Password
+PublicCalendar
+getName()
+getSurname()
+setPhoneNumber()
+getPhoneNumber()
+getBirthday()
+setEmail()
+getEmail()
+setPassword()
+getPassword()
+setPublicCalendar()
+getPublicCalendar()
+findUserByEmail()
+findUserByName()
+findUserBySurname()
«Control»
Search Manager
+checkSearchedData()
+loadSearchedUser()
+searchedUserProfile()
+loadPublicEvents()
«Boundary»
Search Results
+showSearchResults()
+selectUser()
+showUserProfile()
+displayEvent()
«Entity»
Event
+Name
+TimeStart
+TimeEnd
+Description
+Town
+Address
+Latitude
+Longitude
+Public
+Delayed
+Deleted
+Typology
+NonPredefinedTypology
+SETTERS()
+GETTERS()
+findEventsOfUser(u : User)
«Control»
Event Manager
+checkFields()
+checkDateConsistency()
+createEvent()
+updateEventInformation()
+removeEvent()
+checkEmailForInvitation()
+addInvitation()
+userToBeNotified()
+checkWeatherForecast()
+addParticipantToEvent()
+removeParticipantFromEvent()
+loadEvent()
«Boundary»
Event
+showEvent()

4.3 Event Management

In this BCE diagram we have introduced new Boundaries and Controls. Let's start with the analysis of the boundaries:

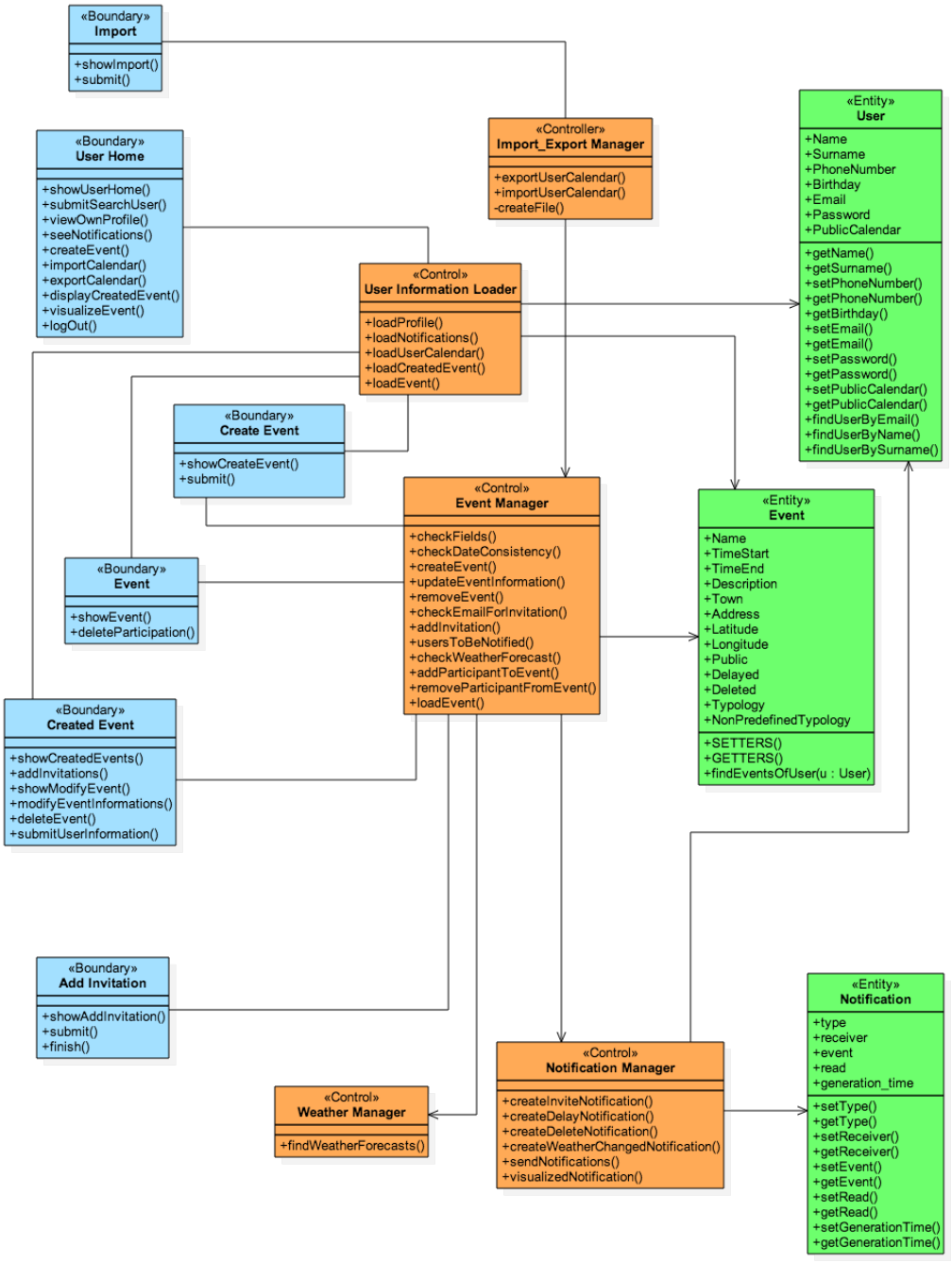
- **Create Event:** from this page a user can create a new event;
- **Created Event:** it visualizes events of which the user is the organizer. From this page, the user can modify an event (changing some information or adding new invitations), but also delete a created event.
- **Add Invitations:** here the user can send new invitations for the event to other users.
- **Event:** in this page the user can find the information about an event and he can also decide to delete his participation to it.

We have four new controllers: we can start to analyze the **Event Manager**, which manages everything related to events.

Its control starts already upon event phase: it checks that all mandatory fields are fulfilled in a correct way and that there is no overlap with another event to which the user is going to participate. If all is right, it creates the event. Immediately after the event creation, the user has the possibility to send invitations for the event: in this situation, the Event Manager must check the inserted email for the invitation (not only in terms of correctness of the characters, but also verifying the presence of a user whose email matches with the inserted one) and, in case of positive response, provides to add a new invitation, exploiting the work of another controller, the **Notification Manager**. In fact, the Event Manager finds all the needed information (email of the invited user, event) and "sends" it to the Notification Manager, which can create a new invite notification and sends it to the exact user. The Event Manager controls also the modification of an event, checking the same previous constraints about fields and date, and update the event information. It is important to say that, for some modifies (delete or delay), it is necessary to send also notification to users: the Event Manager selects all the users to be notified and "communicates" this information to the Notification Manager, which creates and sends the appropriate notification to the correct users.

The Event Manager exploits also the work of another controller: the **Weather Manager**. It has the task of find the weather forecast connected to an event. By comparing the results coming from the Weather Manager with the organizer's desired conditions, the Event Manager can discover the necessity to inform the organizer (or all the users, it depends on the exact situation) about a change in weather conditions, and it does it by exploiting another time the work of the Notification Manager.
Moreover, the Event Manager is used by the Import_Export Manager: in

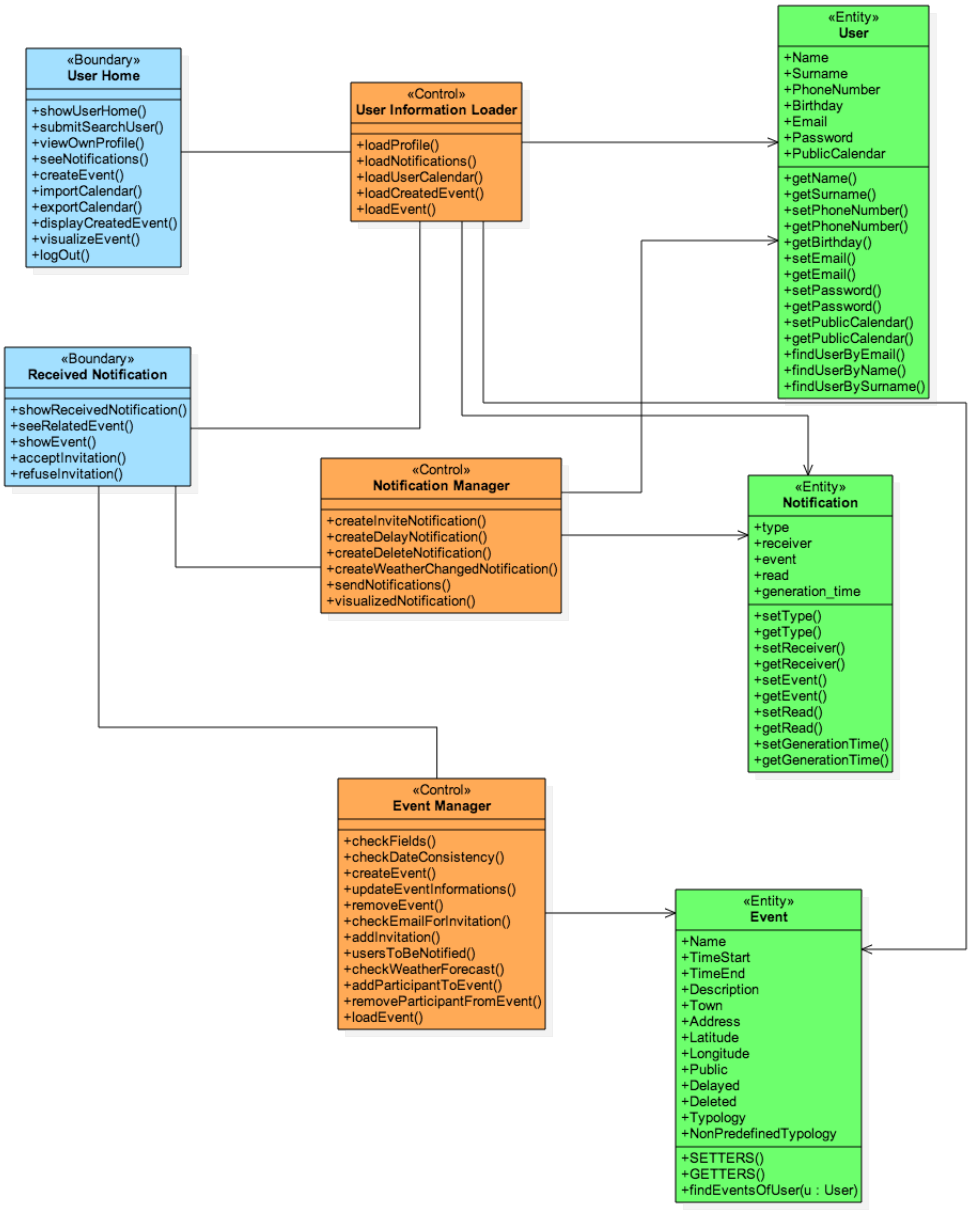
fact, when a user wants to import his calendar, he has to upload the file of the calendar and the Import_Export Manager has to invoke some methods of the Event Manager (e.g., checkDateConsistency(), createEvent()) to check if the import is possible or to highlight that it can't be performed because there are some overlapping events.
Finally, in case of a confirm/delete of a participation to/from an event, the Event Manager updates the information about the participants.
We have only to add a detail about the Notification Manager: it also provides to update the information about the "read state" of the notifications every time a user visualizes one of them.


«Boundary»
Import
+showImport()
+submit()
«Controller»
Import_Export Manager
+exportUserCalendar()
+importUserCalendar()
-createFile()
«Entity»
User
+Name
+Surname
+PhoneNumber
+Birthday
+Email
+Password
+PublicCalendar
+getName()
+getSurname()
+setPhoneNumber()
+getPhoneNumber()
+getBirthday()
+setEmail()
+getEmail()
+setPassword()
+getPassword()
+setPublicCalendar()
+getPublicCalendar()
+findUserByEmail()
+findUserByName()
+findUserBySurname()
«Boundary»
User Home
+showUserHome()
+submitSearchUser()
+viewOwnProfile()
+seeNotifications()
+createEvent()
+importCalendar()
+exportCalendar()
+displayCreatedEvent()
+visualizeEvent()
+logOut()
«Control»
User Information Loader
+loadProfile()
+loadNotifications()
+loadUserCalendar()
+loadCreatedEvent()
+loadEvent()
«Boundary»
Create Event
+showCreateEvent()
+submit()
«Control»
Event Manager
+checkFields()
+checkDateConsistency()
+createEvent()
+updateEventInformation()
+removeEvent()
+checkEmailForInvitation()
+addInvitation()
+usersToBeNotified()
+checkWeatherForecast()
+addParticipantToEvent()
+removeParticipantFromEvent()
+loadEvent()
«Entity»
Event
+Name
+TimeStart
+TimeEnd
+Description
+Town
+Address
+Latitude
+Longitude
+Public
+Delayed
+Deleted
+Typology
+NonPredefinedTypology
+SETTERS()
+GETTERS()
+findEventsOfUser(u : User)
«Boundary»
Event
+showEvent()
+deleteParticipation()
«Boundary»
Created Event
+showCreatedEvents()
+addInvitations()
+showModifyEvent()
+modifyEventInformations()
+deleteEvent()
+submitUserInformation()
«Boundary»
Add Invitation
+showAddInvitation()
+submit()
+finish()
«Control»
Weather Manager
+findWeatherForecasts()
«Control»
Notification Manager
+createInviteNotification()
+createDelayNotification()
+createDeleteNotification()
+createWeatherChangedNotification()
+sendNotifications()
+visualizedNotification()
«Entity»
Notification
+type
+receiver
+event
+read
+generation_time
+setType()
+getType()
+setReceiver()
+getReceiver()
+setEvent()
+getEvent()
+setRead()
+getRead()
+setGenerationTime()
+getGenerationTime()

4.4 Notification Management

The boundary **Received Notification** permits to visualize a received notification, to visualize information about the related event and, in case it is an invite notification, to accept/refuse it. This is the only way through which a user can accept or refuse an invitation.

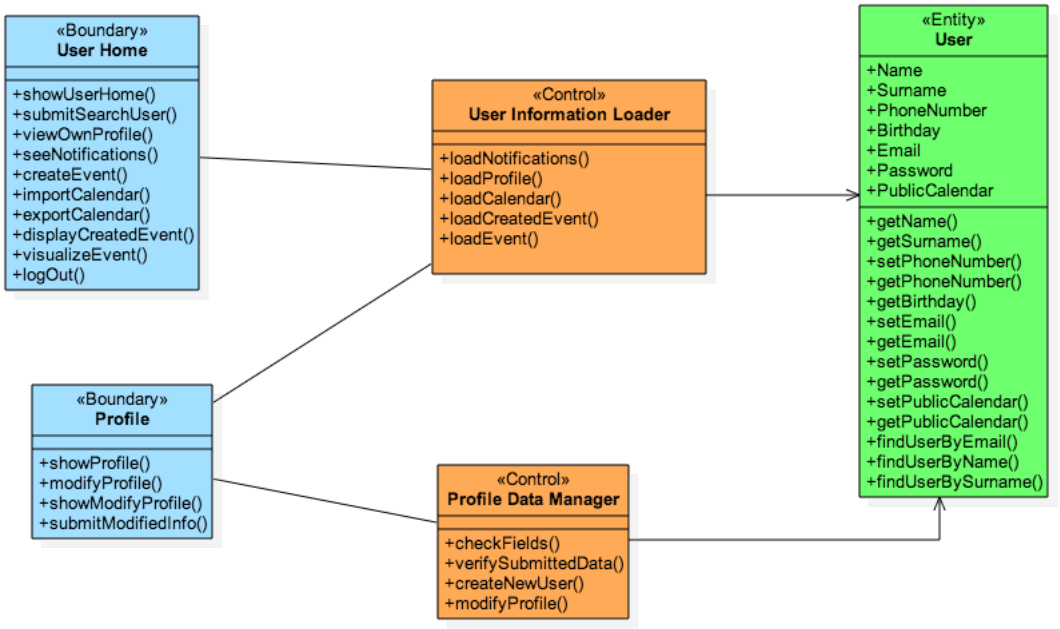
The two controllers involved are the Event Manager (it has to update the information about the participants of an event, but also to show data about the event) and obviously the Notification Manager, but we have already analyzed them.


«Boundary»
User Home
+showUserHome()
+submitSearchUser()
+viewOwnProfile()
+seeNotifications()
+createEvent()
+importCalendar()
+exportCalendar()
+displayCreatedEvent()
+visualizeEvent()
+logOut()
«Control»
User Information Loader
+loadProfile()
+loadNotifications()
+loadUserCalendar()
+loadCreatedEvent()
+loadEvent()
«Entity»
User
+Name
+Surname
+PhoneNumber
+Birthday
+Email
+Password
+PublicCalendar
+getName()
+getSurname()
+setPhoneNumber()
+getPhoneNumber()
+getBirthday()
+setEmail()
+getEmail()
+setPassword()
+getPassword()
+setPublicCalendar()
+getPublicCalendar()
+findUserByEmail()
+findUserByName()
+findUserBySurname()
«Boundary»
Received Notification
+showReceivedNotification()
+seeRelatedEvent()
+showEvent()
+acceptInvitation()
+refuseInvitation()
«Control»
Notification Manager
+createInviteNotification()
+createDelayNotification()
+createDeleteNotification()
+createWeatherChangedNotification()
+sendNotifications()
+visualizedNotification()
«Entity»
Notification
+type
+receiver
+event
+read
+generation_time
+setType()
+getType()
+setReceiver()
+getReceiver()
+setEvent()
+getEvent()
+setRead()
+getRead()
+setGenerationTime()
+getGenerationTime()
«Control»
Event Manager
+checkFields()
+checkDateConsistency()
+createEvent()
+updateEventInformations()
+removeEvent()
+checkEmailForInvitation()
+addInvitation()
+usersToBeNotified()
+checkWeatherForecast()
+addParticipantToEvent()
+removeParticipantFromEvent()
+loadEvent()
«Entity»
Event
+Name
+TimeStart
+TimeEnd
+Description
+Town
+Address
+Latitude
+Longitude
+Public
+Delayed
+Deleted
+Typology
+NonPredefinedTypology
+SETTERS()
+GETTERS()
+findEventsOfUser(u : User)

4.5 Profile Management

The boundary **Profile** displays all the user's information: from this page, the user can modify his profile. By performing this choice, he has to fulfill the fields, inserting the new information, and then he can submit the new data.

As we said before, the Profile Data Manager checks the consistency of the inserted data and, in case of positive response, modify the user's profile.

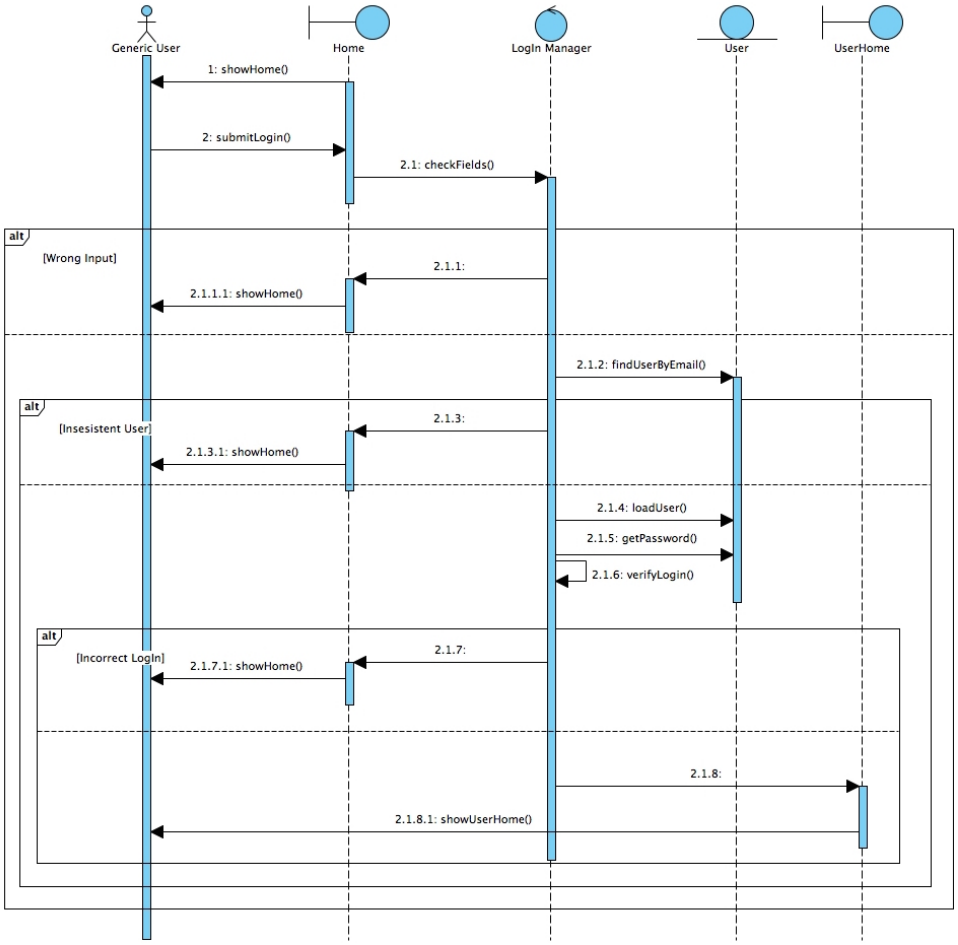


5. Sequence Diagrams

We provide some sequence diagrams to better understand BCE diagrams described above.

5.1 Log In

A generic user tries to log in.



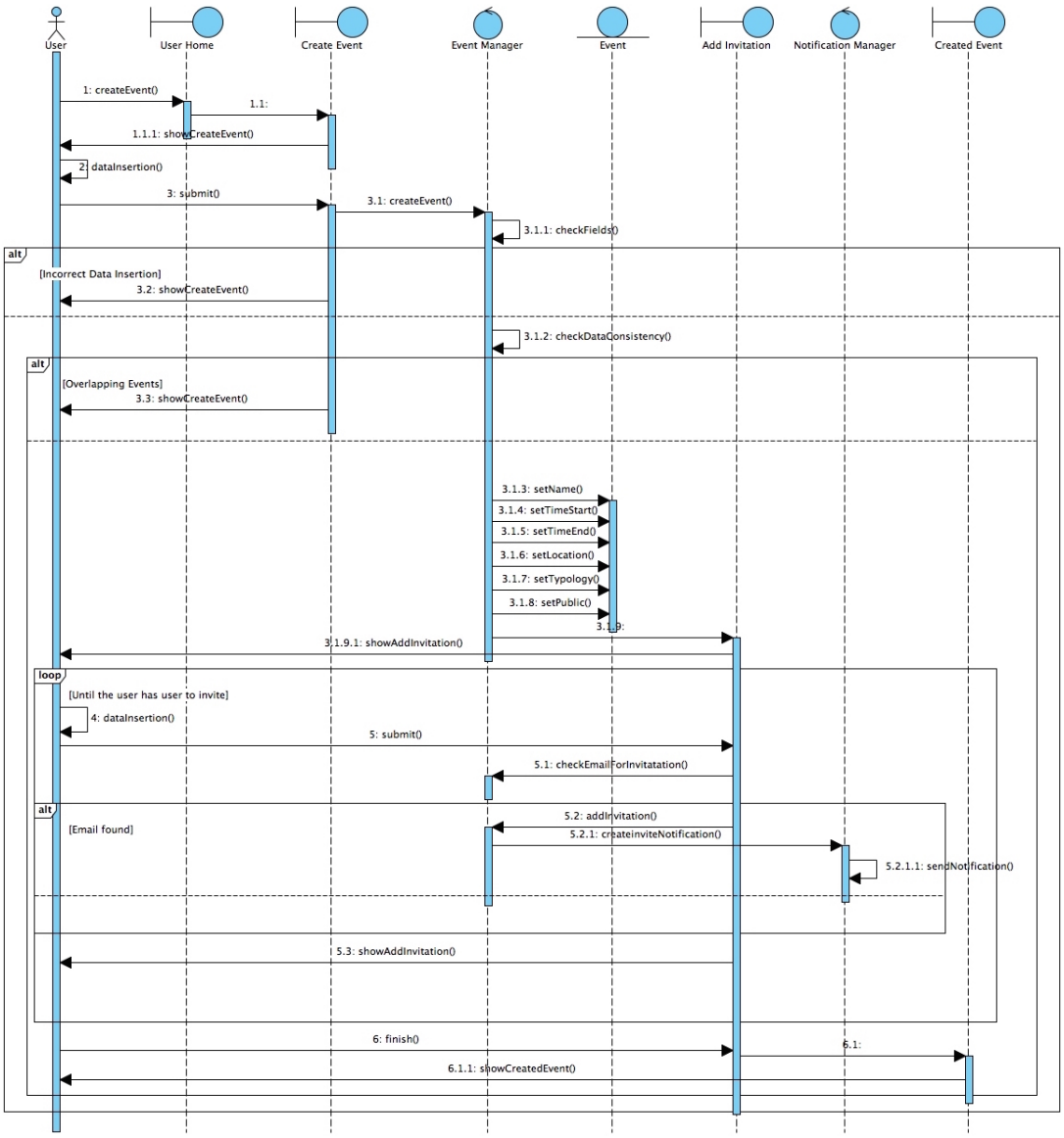
5.2 New Event

A user:

- Wants to create new event.
- Adds people to be invited to the new created event.

If there's any overlapping event into the creator's calendar, the system notifies the error and prevents the creation of the events.

In this diagram all the methods that the system has to call to create the event and to search a user to be invited are not specified.

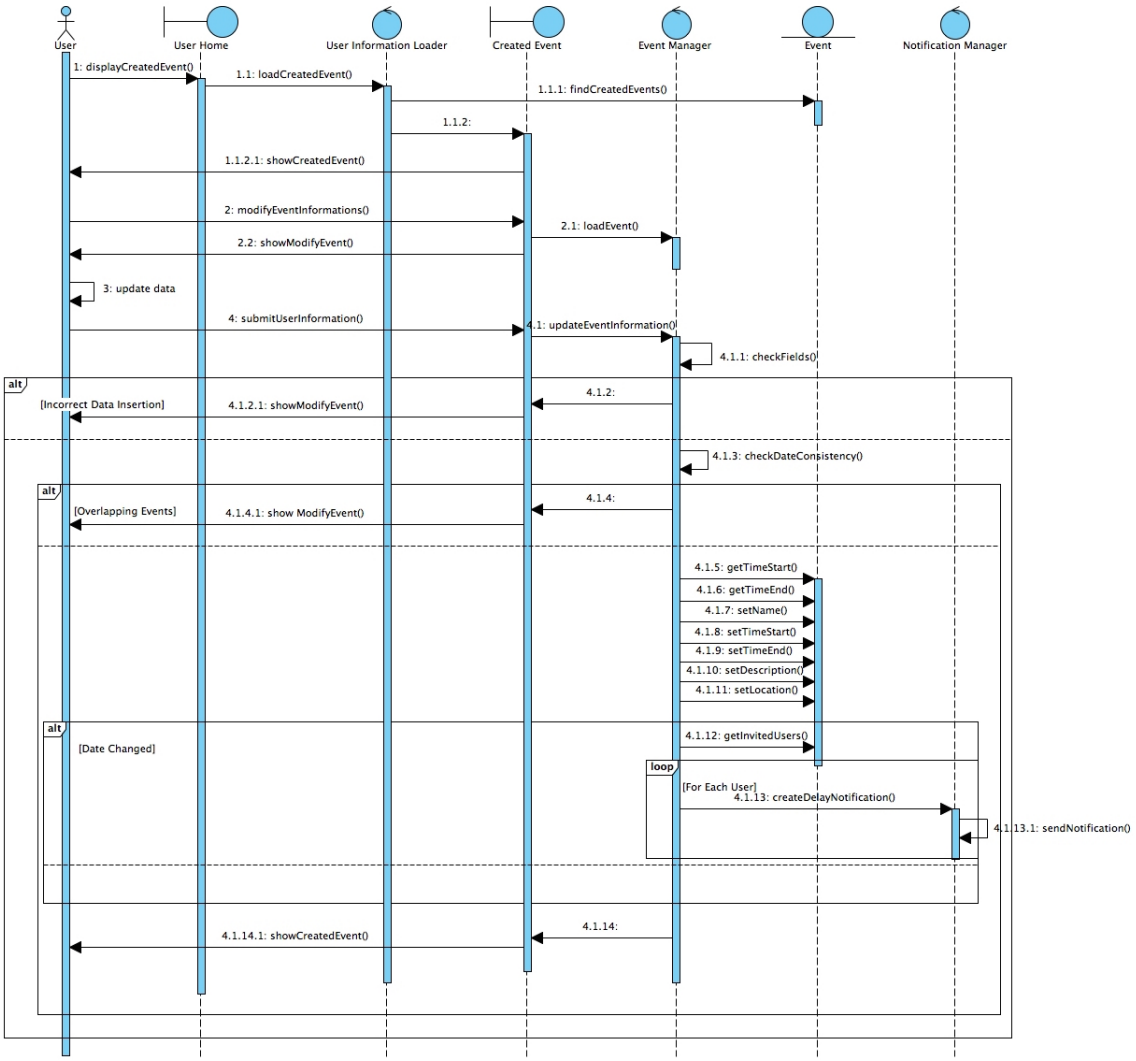

User
User Home
Create Event
Event Manager
Event
Add Invitation
Notification Manager
Created Event
1: createEvent()
1.1:
1.1.1: showCreateEvent()
2: dataInsertion()
3: submit()
3.1: createEvent()
3.1.1: checkFields()
alt
[Incorrect Data Insertion]
3.2: showCreateEvent()
3.1.2: checkDataConsistency()
alt
[Overlapping Events]
3.3: showCreateEvent()
3.1.3: setName()
3.1.4: setTimeStart()
3.1.5: setTimeEnd()
3.1.6: setLocation()
3.1.7: setTypology()
3.1.8: setPublic()
3.1.9:
3.1.9.1: showAddInvitation()
loop
[Until the user has user to invite]
4: dataInsertion()
5: submit()
5.1: checkEmailForInvitatation()
alt
[Email found]
5.2: addInvitation()
5.2.1: createinviteNotification()
5.2.1.1: sendNotification()
5.3: showAddInvitation()
6: finish()
6.1:
6.1.1: showCreatedEvent()

5.3 Update Event

A user:

- Wants to update the information about an event that he created in the past.
- Modifies also the date of the event.

In this diagram all the methods that the system has to call in order to load the information of the event are not specified .

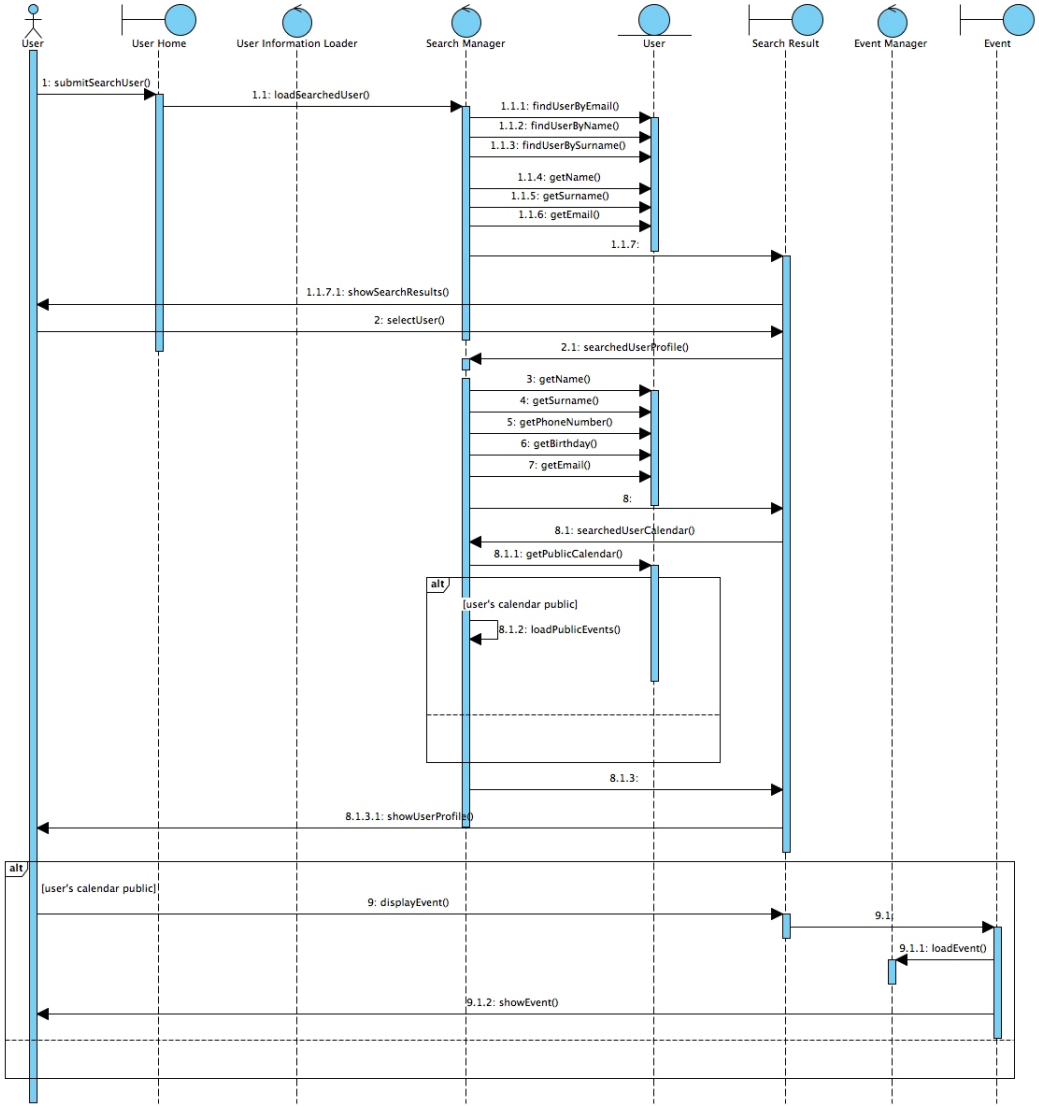


5.4 Search for a User

A user:

- Performs a search to find a user.
- Clicks on the user's name and views his profile and his calendar (if it is public).
- Clicks on an event of the user's calendar to see all the information.

In this diagram all the methods that the system has to call in order to load all the information of the events are not specified.

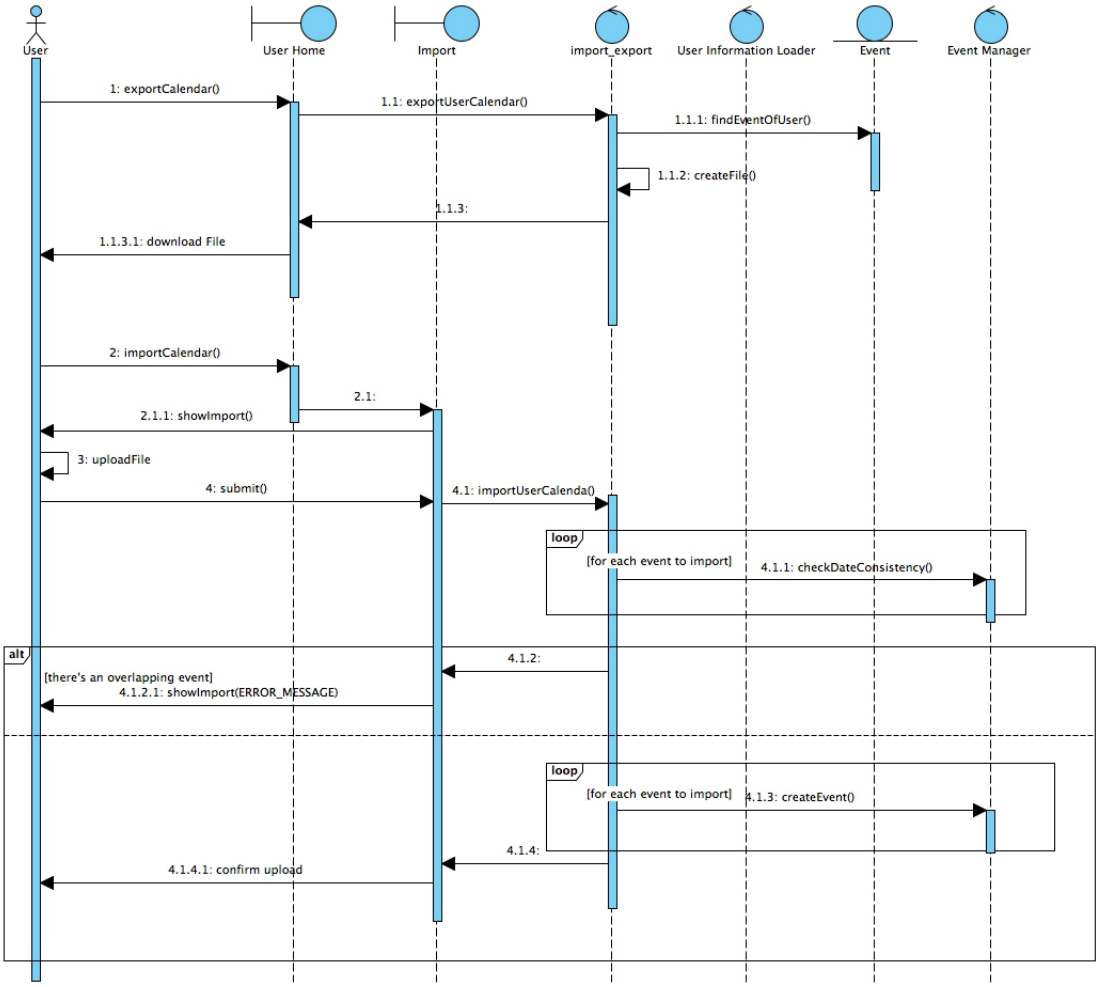


5.5 Import/Export

This diagram includes both the import and the export functionalities only for simplicity, it is not required that the user has to perform both the actions consecutively. A user:

- Wants to export his own calendar.
- Wants to import a calendar into the system.

In this diagram all the methods that the system has to call in order to load all the information of the events, and also all the methods that has to call in order to create the event are not specified.



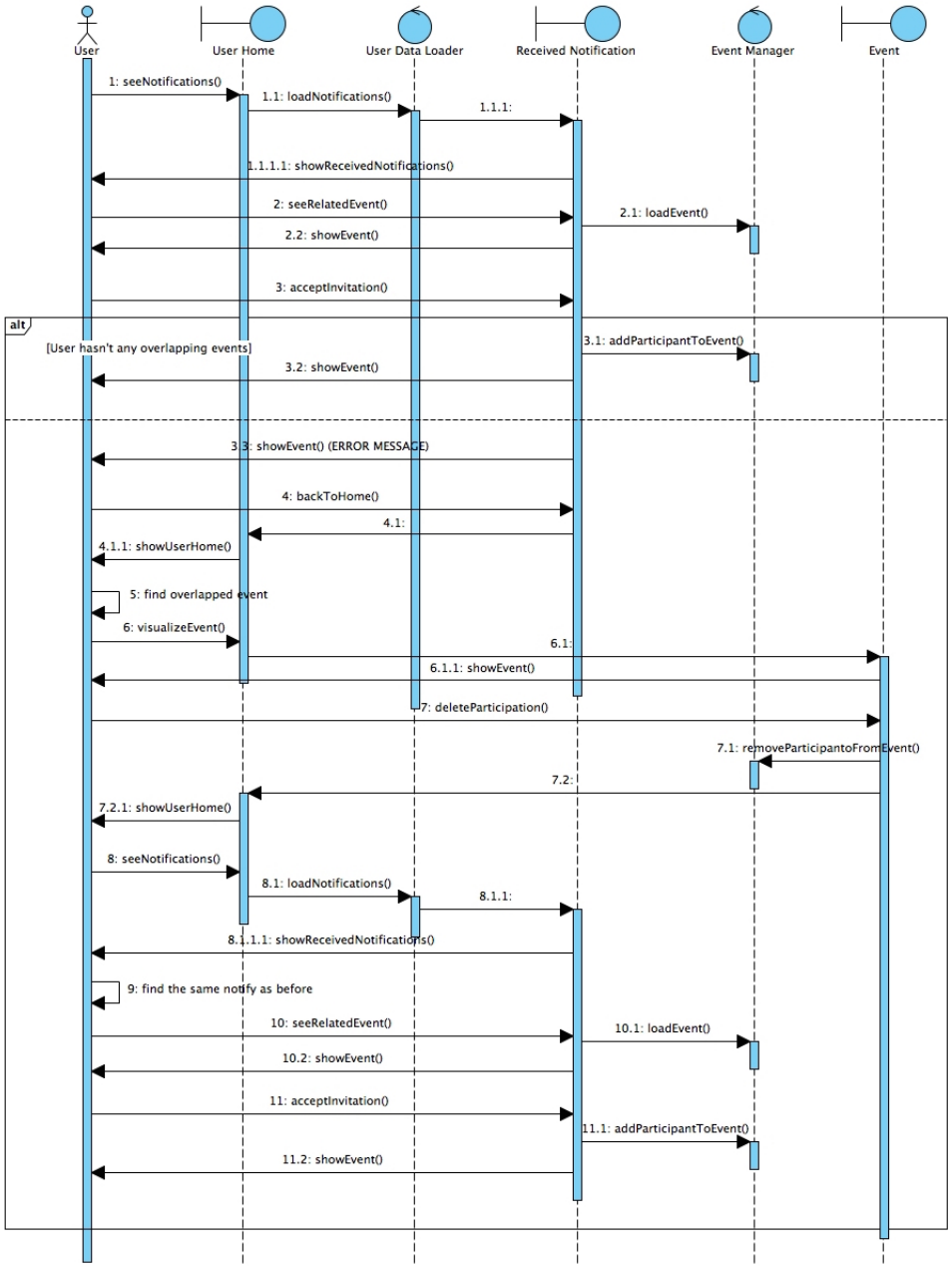
5.6 Accept Invitation

A user:

- Wants to accept an invitation to an event received.

In case of overlapping between the event that the user wants to accept and another event in his own calendar, the system has to notify it: in order to accept that invitation, the user has to delete his participation to the other event, after that he has to perform again the acceptance to the invitation.

In this diagram all the methods that the system has to call in order to load all the information about the event and the searching of overlapping events are not specified.


User
User Home
User Data Loader
Received Notification
Event Manager
Event
1: seeNotifications()
1.1: loadNotifications()
1.1.1:
1.1.1.1: showReceivedNotifications()
2: seeRelatedEvent()
2.1: loadEvent()
2.2: showEvent()
3: acceptInvitation()
alt
[User hasn't any overlapping events]
3.1: addParticipantToEvent()
3.2: showEvent()
3.3: showEvent() (ERROR MESSAGE)
4: backToHome()
4.1:
4.1.1: showUserHome()
5: find overlapped event
6: visualizeEvent()
6.1:
6.1.1: showEvent()
7: deleteParticipation()
7.1: removeParticipantoFromEvent()
7.2:
7.2.1: showUserHome()
8: seeNotifications()
8.1: loadNotifications()
8.1.1:
8.1.1.1: showReceivedNotifications()
9: find the same notify as before
10: seeRelatedEvent()
10.1: loadEvent()
10.2: showEvent()
11: acceptInvitation()
11.1: addParticipantToEvent()
11.2: showEvent()

6. Used Tools

The tools we used to create this RASD document are:

- TexStudio: to redact and format this document;
- VisualParadigm: to create Sequence Diagram;
- MySQLWorkbench: to create the DB Logical schema;
- StarUML: to create BCE diagrams;