Department of

Computer Science and Engineering

MeteoCal

Project development for the 2014 Software Engineering 2 Course

Supervisor: **Prof. Di Nitto**

Authors

Andrea Bignoli
Matr. 837493
andrea.bignoli@gmail.com

Leonardo Cella
Matr. 838074
leonardocella@gmail.com

Design

Document

Digest

A software design document (SDD) is a written description of a software product, that a software designer writes in order to give a software development team overall guidance to the architecture of the software project. An SDD usually accompanies an architecture diagram with pointers to detailed feature specifications of smaller pieces of the design. Practically, a design document is required to coordinate a large team under a single vision. A design document needs to be a stable reference, outlining all parts of the software and how they will work. The document is commanded to give a fairly complete description, while maintaining a high-level view of the software.

Problem description

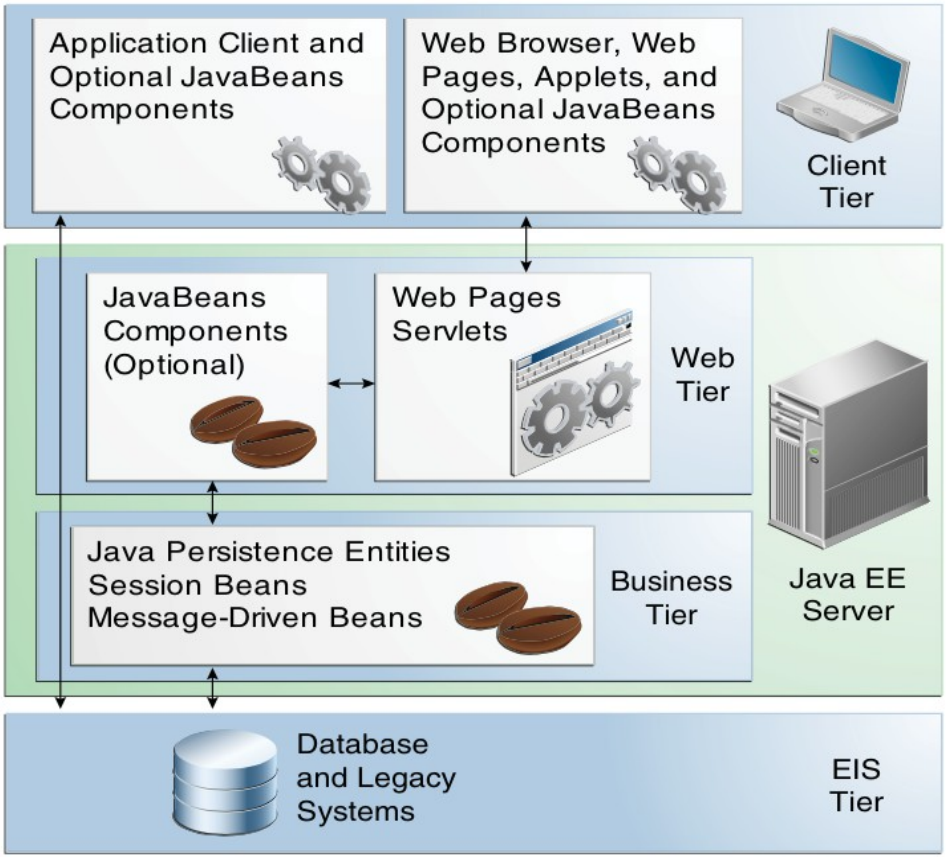
The X company wants to offer a new weather based online calendar for helping people scheduling their personal events avoiding bad weather conditions in case of outdoor activities. Users, once registered, should be able to create, delete and update events. An event should contain information about when and where the event will take place, whether the event will be indoor or outdoor. During event creation, any number of registered users can be invited. Only the organizer will be able to update or delete the event. Invited users can only accept or decline the invitation. Whenever an event is saved, the system should enrich the event with weather forecast information (if available). Moreover, it should notify all event participants one day before the event in case of bad weather conditions for outdoor events. Notifications are received by the users when they log into the system.

Table of Contents

1. Architecture Description

1.1. JEE Architecture Overview

First of all, since our application will be based on JEE we give here an overview over its architecture:



taken from From *Jendrock et al. The Java EE 7 Tutorial*

As showed in the figure above, JEE consists of four levels and for this reason is a multi-tier model. Starting from this model we define the MeteoCal architecture:

Client Tier:

this is the level in which the actors interact. It runs on the client machine, that in our case is a web browser.

Web Tier:

runs on the EE server and we choose to use JSF (JavaServer Faces) as web component. It consists of the web page that needs to be elaborated before being showed to the end users.

Business Tier:

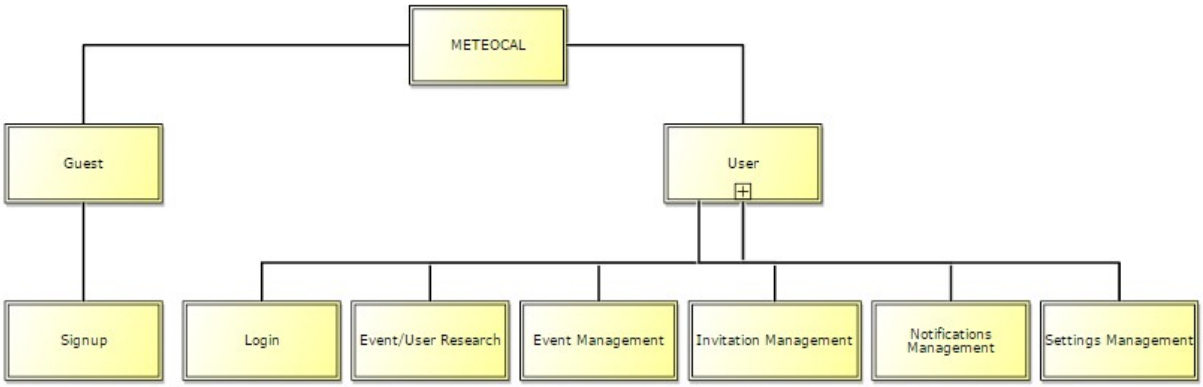
contains the Java Beans, that implements the business logic of the application, and Java Persistence Entities.

EIS Tier:

contains the relational database described later in this document that will store the data needed to offer our services.

1.2 Identifying Sub-Systems

To better define the domains involved in our system, we decided to report here a top-down approach that will describe at an high-level the sub-systems our application is based on an their relation with the actors interacting with the system.



The high-level structure of our system is the following:

- MeteoCal System
  - Guest Sub-System
    - Sign up Sub-System
  - User Sub-System
    - Login Sub-System
    - Event/User Research Sub-System
    - Event Management Sub-System
    - Invitation Management Sub-System
    - Notification Management Sub-System
    - Settings Management Sub-System

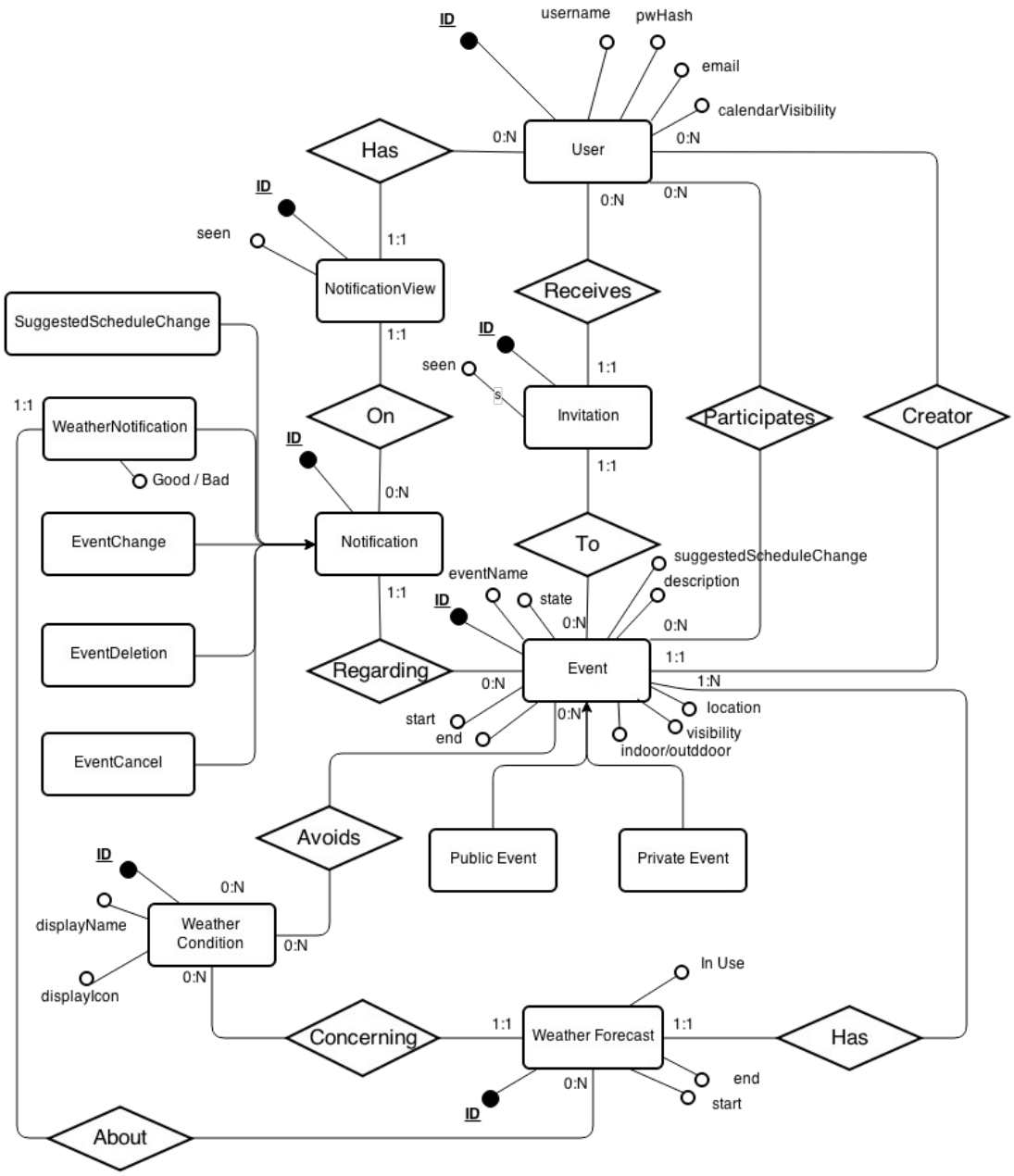
2. Persistent Data Management

The data needed to provide our services will be stored in a relational database. The following section will represent and describe its structure, starting from a conceptual level to get to the final database logic architecture.

2.1 Conceptual Design

This section will provide the reader with an overview on the design process that lead to our system architecture. Moreover, it will clarify some of the higher level specifications mentioned in the RASD.

A brief description of its components will follow the Entity Relationship diagram below:



A person registered in the system is represented by the *User* entity. Each user record includes a unique username, the hash of the user's password, an optional email and a boolean flag telling if his calendar is public. The system will identify each user with an ID.

Each event is represented by a *Event* entity, which can be a *Private Event* or a *Public Event*. The record includes all the event details and its privacy level using the *visibility* boolean flag.

As the reader will notice each event has an associated creator, which is a *User*, and an undefined number of participants and invitations. These facts are respectively modelled via the *Creator*, *Participates* and *To* relationships.

Each *Invitation* is associated to an invited *User* via the *Receives* relationship. The seen flag tells if the user already saw the invitation.

Each event has a set of associated notifications. There are five different types of notifications:

- *SuggestedScheduleChange*:
  to suggest schedule changes in case of bad weather forecast three days before the event, sent only to the creator.
- *WeatherNotification*:
  to warn the participants and the creator of bad weather conditions one day before the event.
- *EventChange*:
  to warn the participants about changes in the event details made by the creator, including also the event cancelling.
- *EventDeletion*:
  to warn the participants about the deletion of an event.
- *EventCancel*:
  to warn the participants about an event being cancelled.

To allow the system to actually know if a particular user already saw a notification the system needs to record this piece of information as a parameter of a relationship linking the user and the notification, or in a new entity linking *Notification* and *User* indirectly. In the above diagram, for readability purposes, we chose the latter option with the creation of the *NotificationView* entity. The actual way in which this will be translated in our database is described in the next section.

Each event has a set of weather conditions to avoid. This allows the system to warn the participants in case the weather forecast is adverse. In this case a *WeatherNotification* is sent. This is represented by the *Avoids* relationship.

An event has also a set of associated weather forecasts provided by the external weather forecast service we are going to use. Those forecasts form a sequence covering the whole duration of the event. If a forecast isn't available in a given interval of time, it will be represented in the database with a “Not available” weather condition.

When the weather forecasts for an event are updated, the attribute “In Use” in the previous ones will be set to false. Those forecasts will instead be deleted when no *Weather Notification* has been produced because of them.

Since each *WeatherNotification* is sent because of a weather forecast, the notification is going to include in its record also the forecast reference.

As mentioned, the system will offer schedule change suggestions to the creator in some conditions. The event record includes the *suggestedScheduleChange* attribute for this purpose.

The event state can be:

- *Planned*:
  the regular state for an event.
- *Cancelled*:
  the event enters this state if an event gets cancelled by its creator.
- *Concluded*:
  the event enters this state automatically when the end date is reached.

2.2 Logical Design

Having defined the conceptual schema of our system, we now can refine our previous conclusions to derive the actual logical structure of our database. This operation will consist, for the most part, in translating relationships and entities in data tables that will grant the functionalities needed to provide our service in an efficient way.

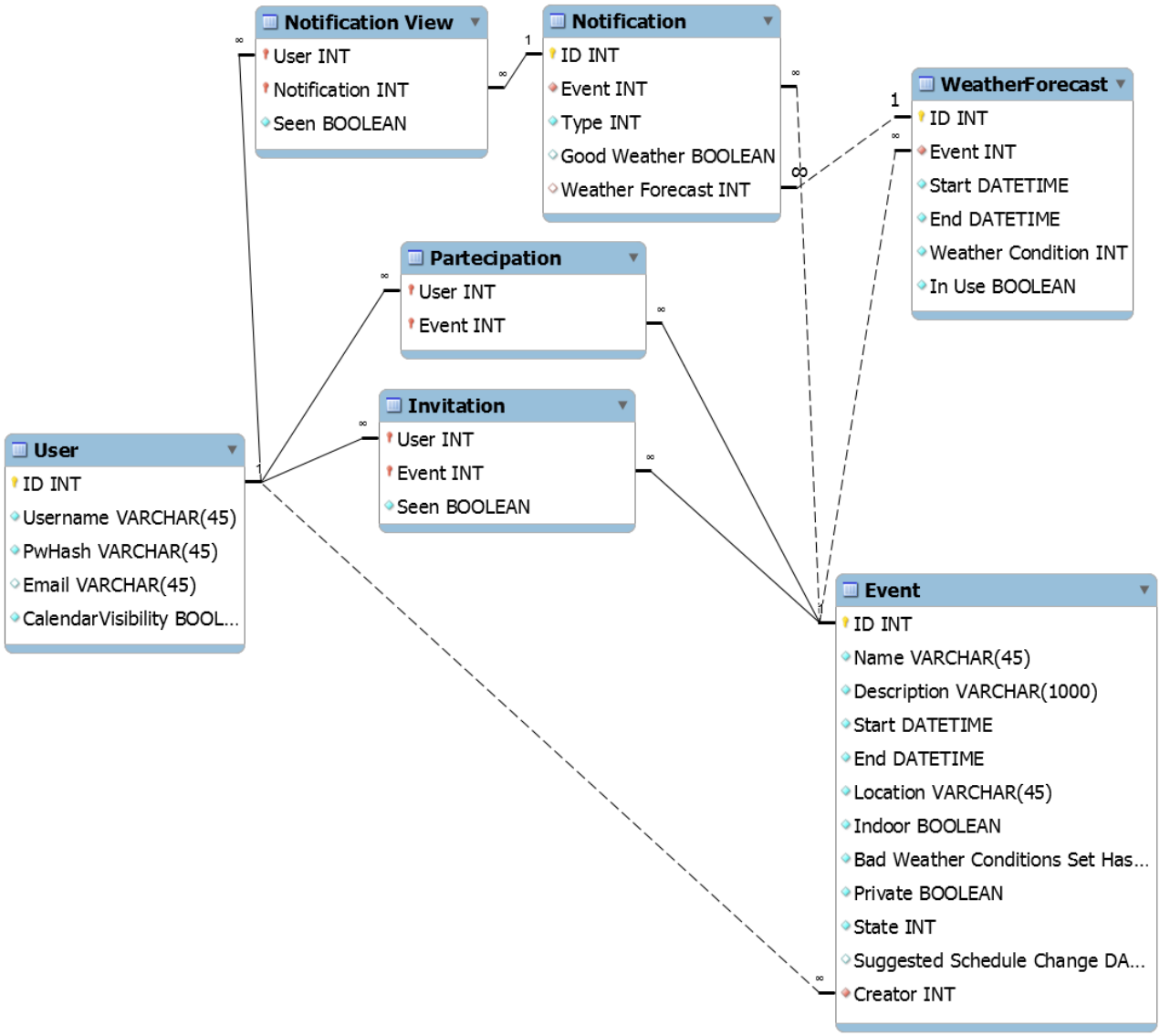
2.2.1 ER Restructuring

- *Private Event* and *Public Event* have no particular additional attributes. For this reason those entities will be both described by the Event entity, with the addition of a boolean flag to state if an event is private or not.
- The five kinds of notifications will be also described by the more general *Notification* entity, with the addition of an integer value referencing a code that will allow to distinguish between the different types of notifications. The additional attributes of *WeatherNotification* will be added to *Notification*, becoming optional.
- The *WeatherCondition* entity will be actually coded in our system, and there's no need to explicitly state the existence of a table for it in our relational database. The weather conditions will be reference via an integer code to be specified (e.g. “Not available”: 0, “Sun”: 1).

2.2.2 Translation to Logical Data Model

- Relationships *Receives* and *To* are translated into two foreign keys in *Invitation* table, that can be used as primary key since the couple is unique. The Invitation ID is redundant and won't be included in the table.
- Relationships *Has* and *On* are translated into two foreign keys in *NotificationView* table, that can be used as primary key since the couple is unique. The NotificationView ID is redundant and won't be included in the table.
- Relationship *Regarding* is translated into a foreign key to *Event* in *Notification* table.
- Relationship *Creator* is translated into a foreign key to *User* in *Event* table.
- Relationship *Has* is translated into a foreign key to *Event* in *WeatherForecast* table.
- Relationship *Participation* is translated into the creation of a new table with two foreign keys linking a *User ID* to an *Event ID*.
- Relationship *About* is translated into a foreign key to *WeatherForecast* in the *Notification* table, linking to the weather forecast that caused the Notification.
- Relationship *Concerning* is translated into an integer attribute added to *WeatherForecast* table, representing the code associated to the weather condition to represent.
- Relationship *Avoids* linking *Event* to *WeatherCondition* could be translated into a table with two attributes: Event ID and Weather Condition Code. Nevertheless, this would be extremely wasteful. A way more efficient solution consists in the use of an hash function to express the set of weather conditions to avoid with one integer. The simplest solution, that we are going to adopt, is given by a bitmask. Since the amount of possible weather conditions is limited, the solution is feasible. The possible configurations will be 2^n, where n represents the number of possible weather conditions.

2.2.3 Logical Data Model



As represented, the final model has the following structure:

User (ID, Username, PwHash, Email, Calendar Visibility)

Event (ID, Name, Description, Start, End, Location, Indoor, Bad Weather Conditions Set Hash, Private, State)

Weather Forecast (ID, Event, Start, End, Weather Condition, In Use)

Invitation (*User, Event*, Seen)

Participation (*User, Event*)

Notification (ID, *Event*, Type, Good Weather, Weather Forecast)

Notification View (*User, Notification*, Seen)

3. User Experience

In this section we are going to describe the user interface level of our application. As specified in the RASD, our service will be accessible using a modern browser.

Here we clarify the notation used:

<< screen >>
: Describes an actual web page, dynamically built by our server.

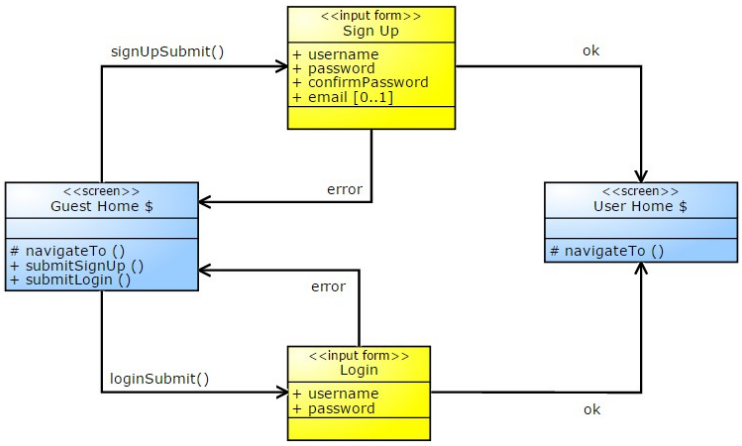
<< screen compartment >>
: Describes a component that will be shared among different pages. It can also be component of another screen compartment.

<< input form >>
: Describes a form where the end user should enter data required to provide different functionalities.

Landmark $ will be used to denote screens accessible by every other page. The + symbol will instead denote pages where there are one or more scrollable collections.

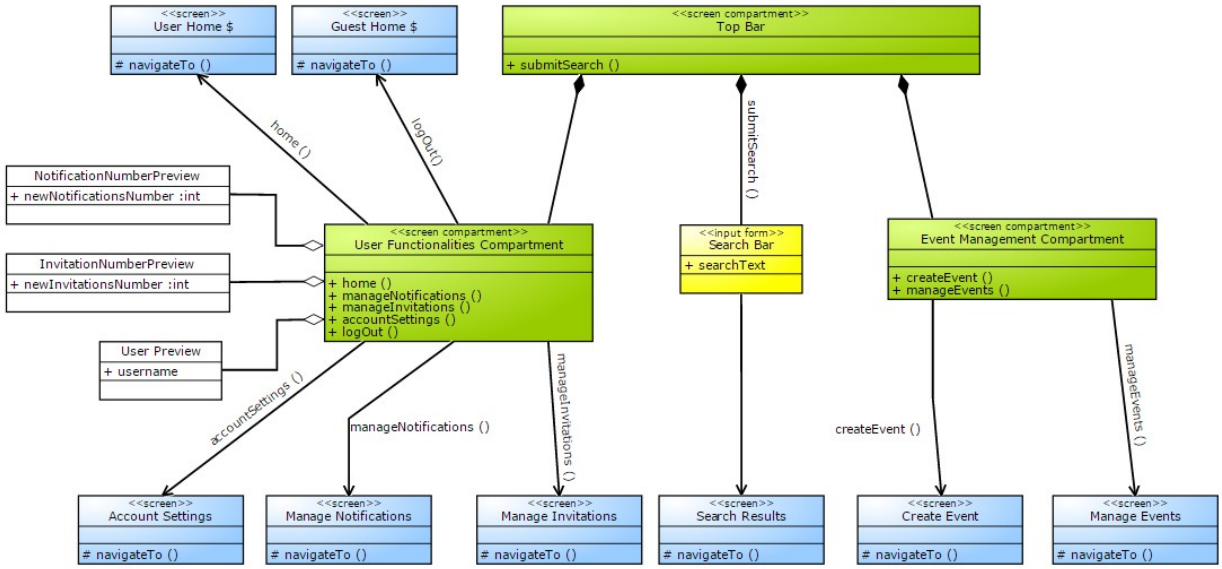
3.1 User Experience Diagrams

3.1.1 Access



This diagram shows the access functionality offered by our system to a guest. The guest home, accessible by every other page using the Log Out functionality, has two input forms, one for the login, and one for the registration of a new user.

3.1.2 Top Bar



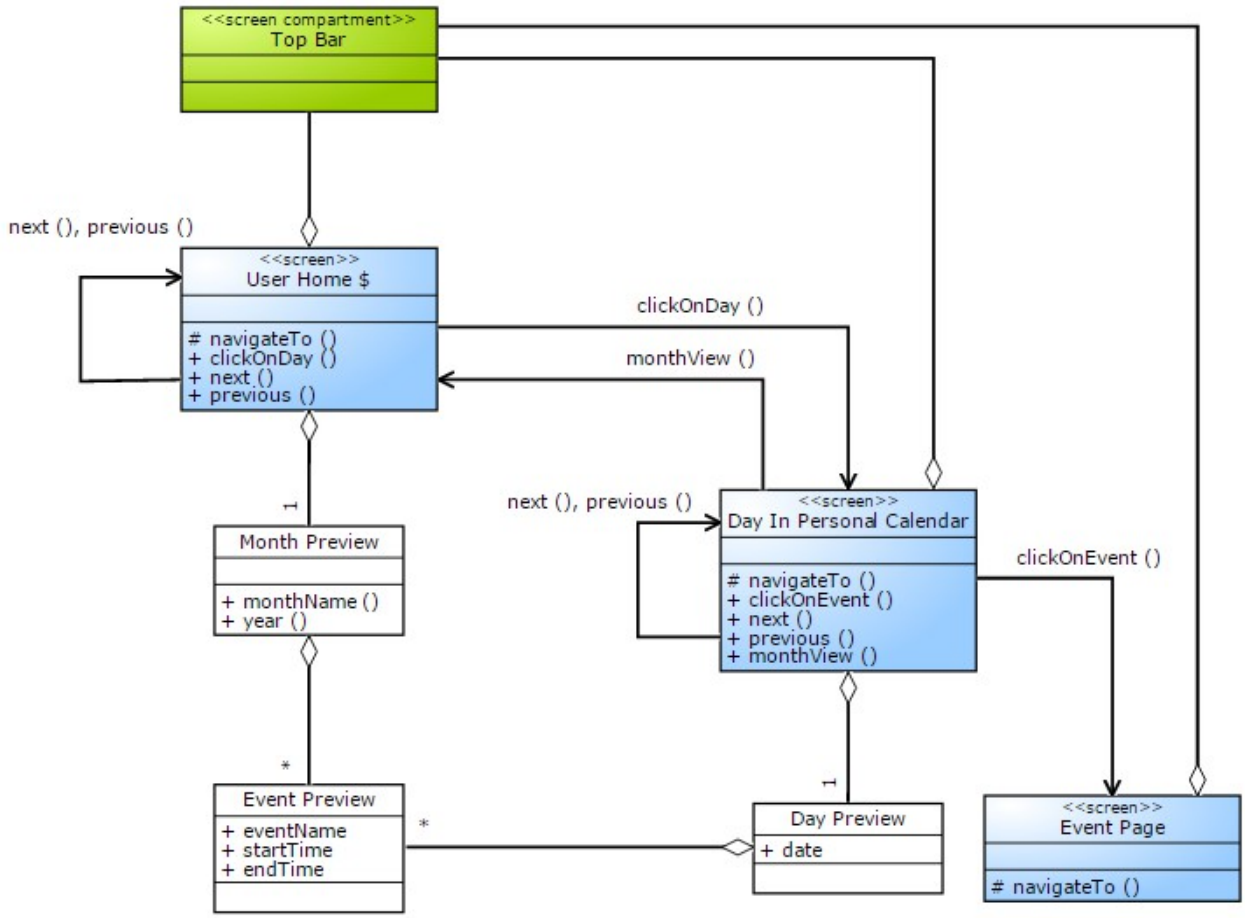
The Top Bar is a component that will be included in each page of the system, except for the Guest Home, where none of its functionalities can be offered, since the user isn't signed in.

This component is functionally a collection of links to:

- User Home
- Guest Home
- Account Settings
- Manage Notifications
- Manage Invitations
- Search Results
- Create Event
- Manage Events

The top bar will also include the number of unread notifications and invitations, and the username of the user. Moreover it will give direct access to a search functionality in which the user submitting content in a textbox, will be able to search for events and other users at the same time.

3.1.3 User Home



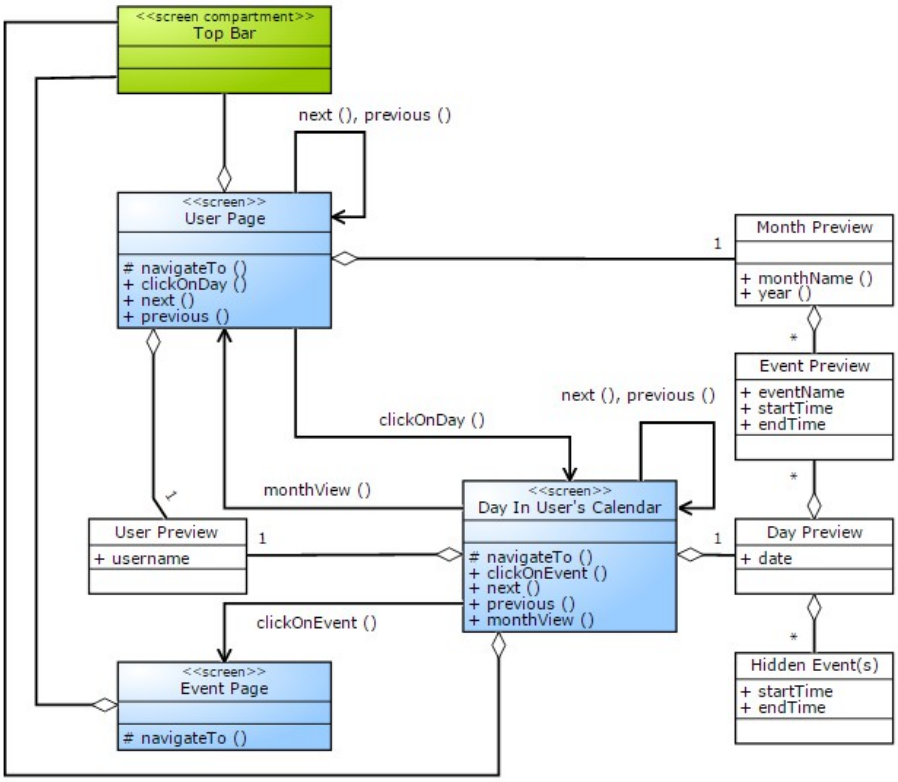
The user's home will show the current month in user's calendar. The user will be able to tell if there's an event in a particular day. Clicking on any day in the calendar, will lead the user to a different screen showing the preview of that particular day, and any event, if present, specifying its name, start and end time. Clicking on any event will lead the user to the event page. Both the month preview and the day preview have next and previous functions with the expected purpose.

3.1.4 User Page

A user will get to another user's page using the search functionality, or clicking on his username on any screen where it's displayed.

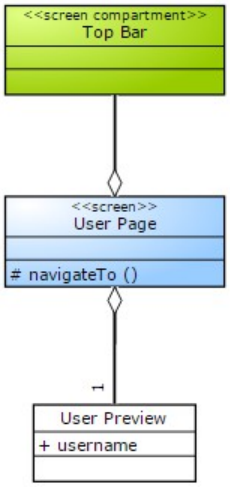
The user's calendar will be visible only if the owner set the visibility of his calendar as public.

If this is the case:

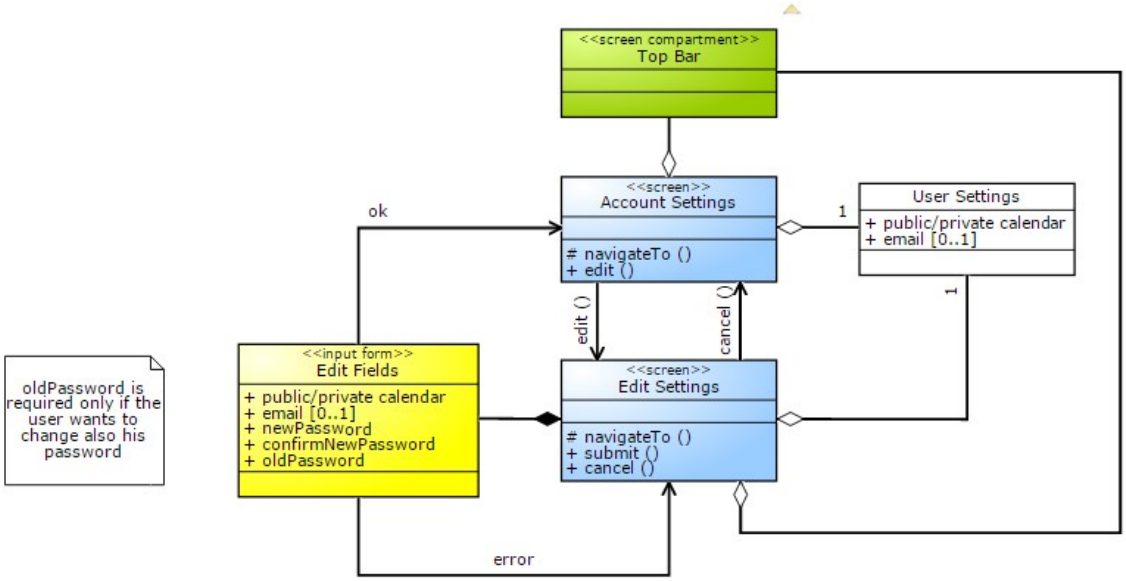


The screen looks really similar to the User Home screen. There are nevertheless noticeable differences. In particular the event's details will be shown if and only if the event itself is public. If the event is private the system will show that the owner of the calendar is busy in that time interval, but nothing more, and clicking on that won't lead the user anywhere. Clicking on a visible event will instead bring the user to the event's page.

If the calendar is private no particular information will be shown:



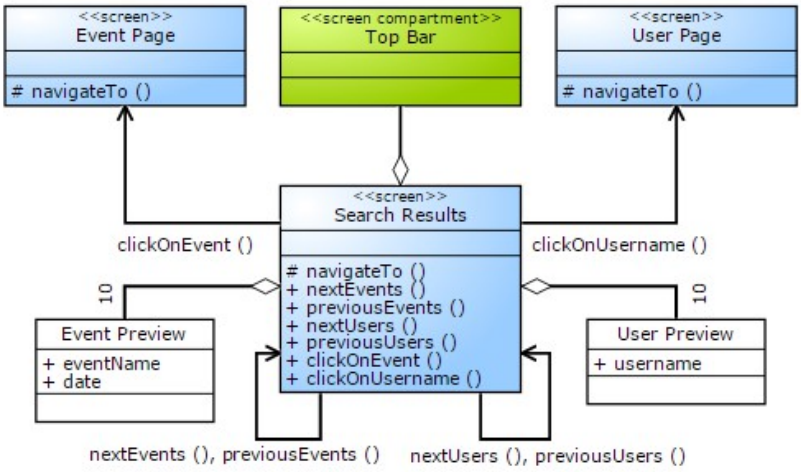
3.1.5 Account Settings



The account settings will allow the user to modify:

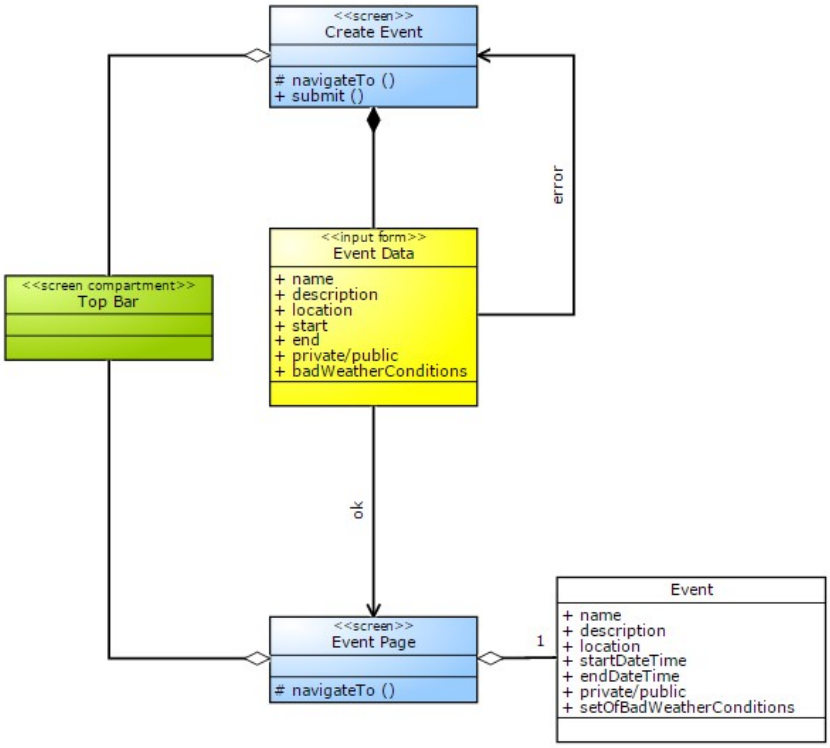
- the visibility of his calendar
- his email
- his password

3.1.6 Search Functionality



Using the search bar in the Top Bar component will bring the user on a results page that is going to include two scrollable separate collections that will include the results in, respectively, the user and the event domain. Clicking on any of the two, will bring the user to its page.

3.1.7 Create Event



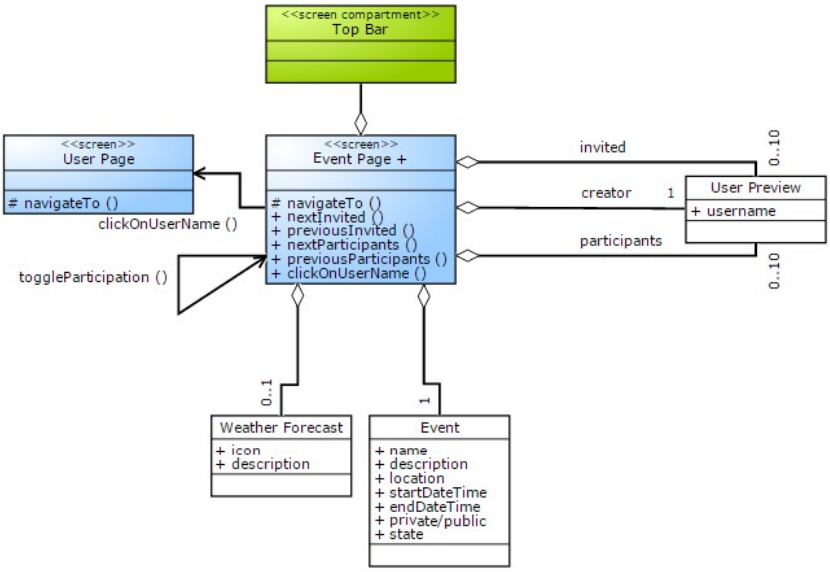
This page will allow the user to create en event filling the required fields. The page won't allow the user to set any possible weather condition as adverse, since this would be meaningless and wasteful for our system (since it will produce useless notifications in any case). The end must be posterior to the start. These and eventual other constraints will be shown to the user by proper error messages. When a successful submit is produced the creator will be lead to the new event's page where he will be able to invite other users to participate.

3.1.8 Event Page

This page will show different functionalities to the generic user and the creator.

If an event is private a user will have visibility over the event only if he has been invited to it. Otherwise he will land on a page warning that he has landed on the page of a private event he can't access.

This, instead, is view of the generic user with visibility of an event:



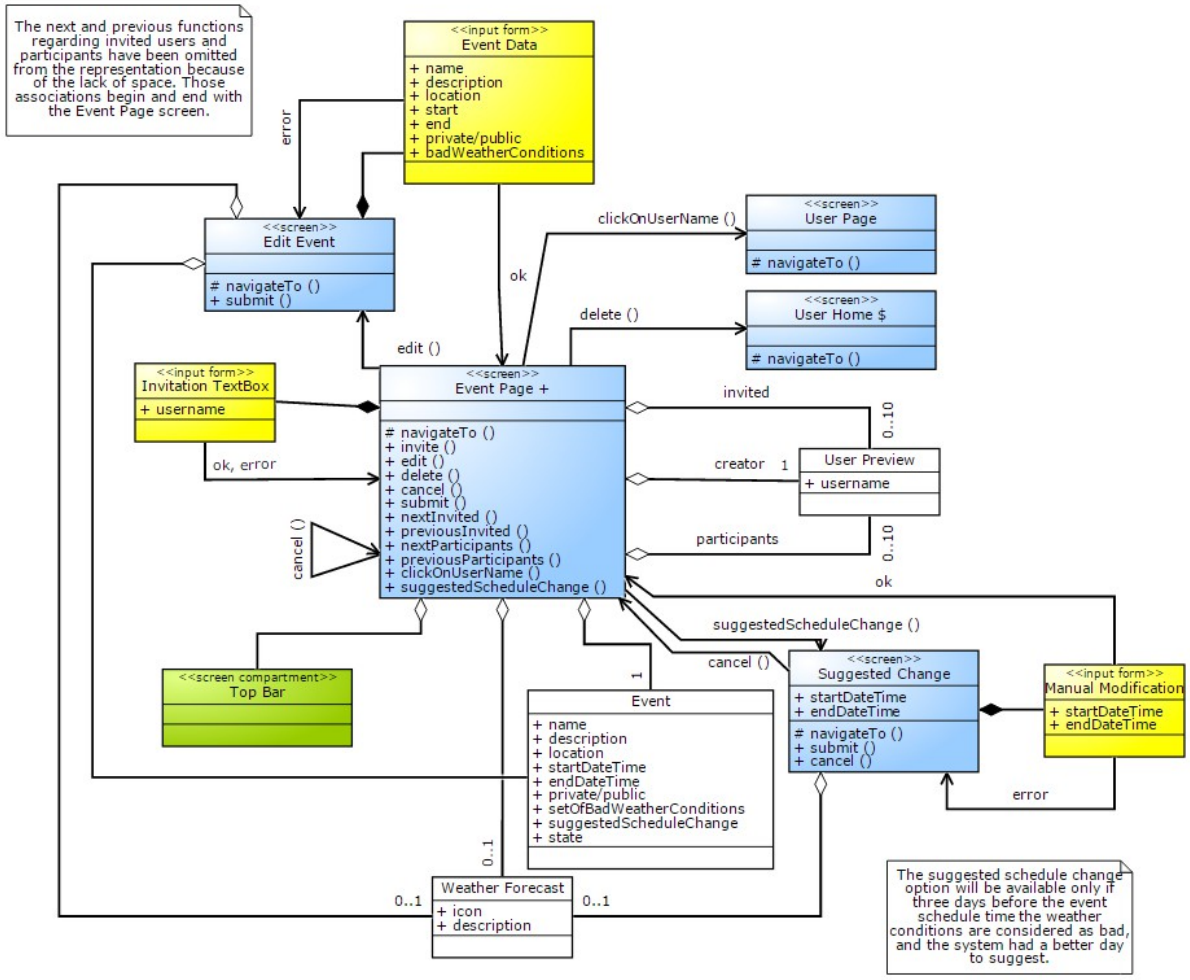
The page shows any meaningful event detail including:

- name
- description
- location
- start and end
- privacy level
- state
- creator
- invited users
- participants
- the forecast weather condition

Clicking on a username will open that user's page.

The page includes a button to toggle the participation to that event.

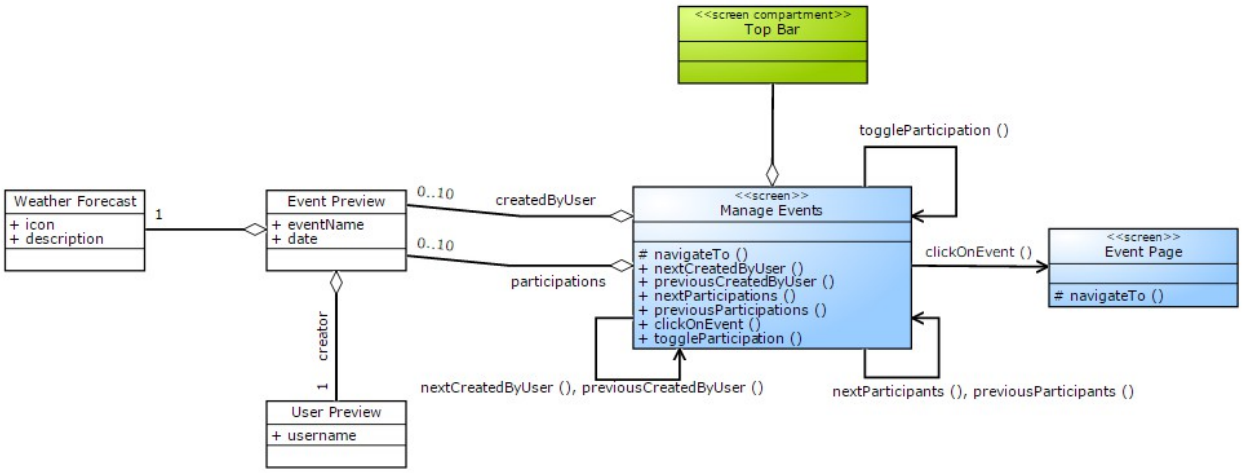
The page shown to the creator won't include the last mentioned button but will give access to functionalities reserved to the creator as the diagram shows:



The additional functionalities represented are:

- invite (using a textbox, where a username can be inserted)
- edit
- use suggested change (displayed only if available)
- cancel
- delete

3.1.9 Event Management



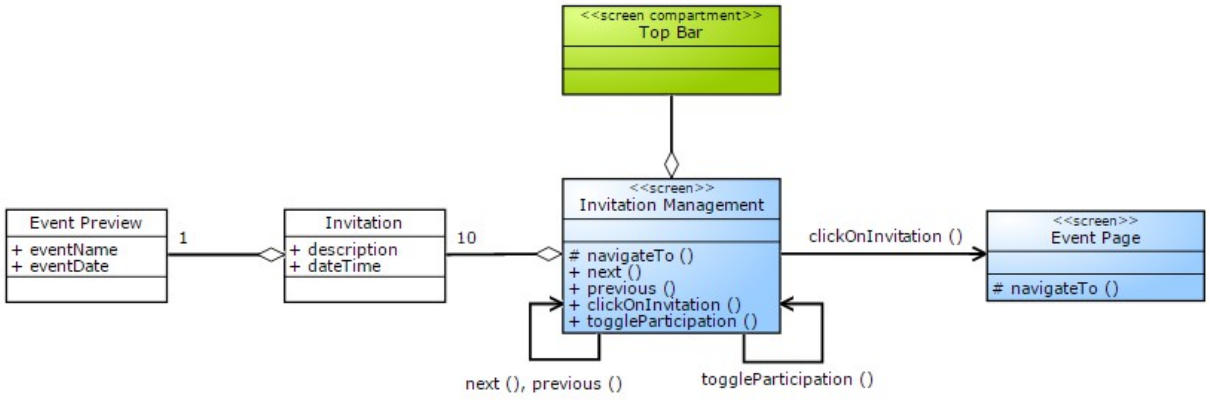
This page will show the user, in two different collections:

- the events he has created
- the events he's going to participate in

Clicking on an event brings the user to the corresponding event's page.

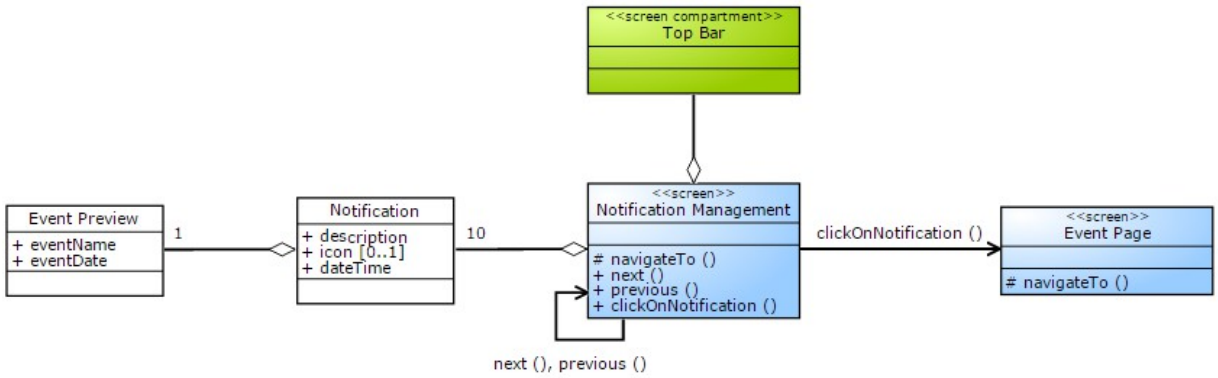
The functionalities of toggling the participation to the events the user he's participating in will also be given here.

3.1.10 Invitation Management



This page will display the user he has received and will allow him to both toggle the participation to those events and reach the page of any of the events he has been invited to.

3.1.11 Notification Management



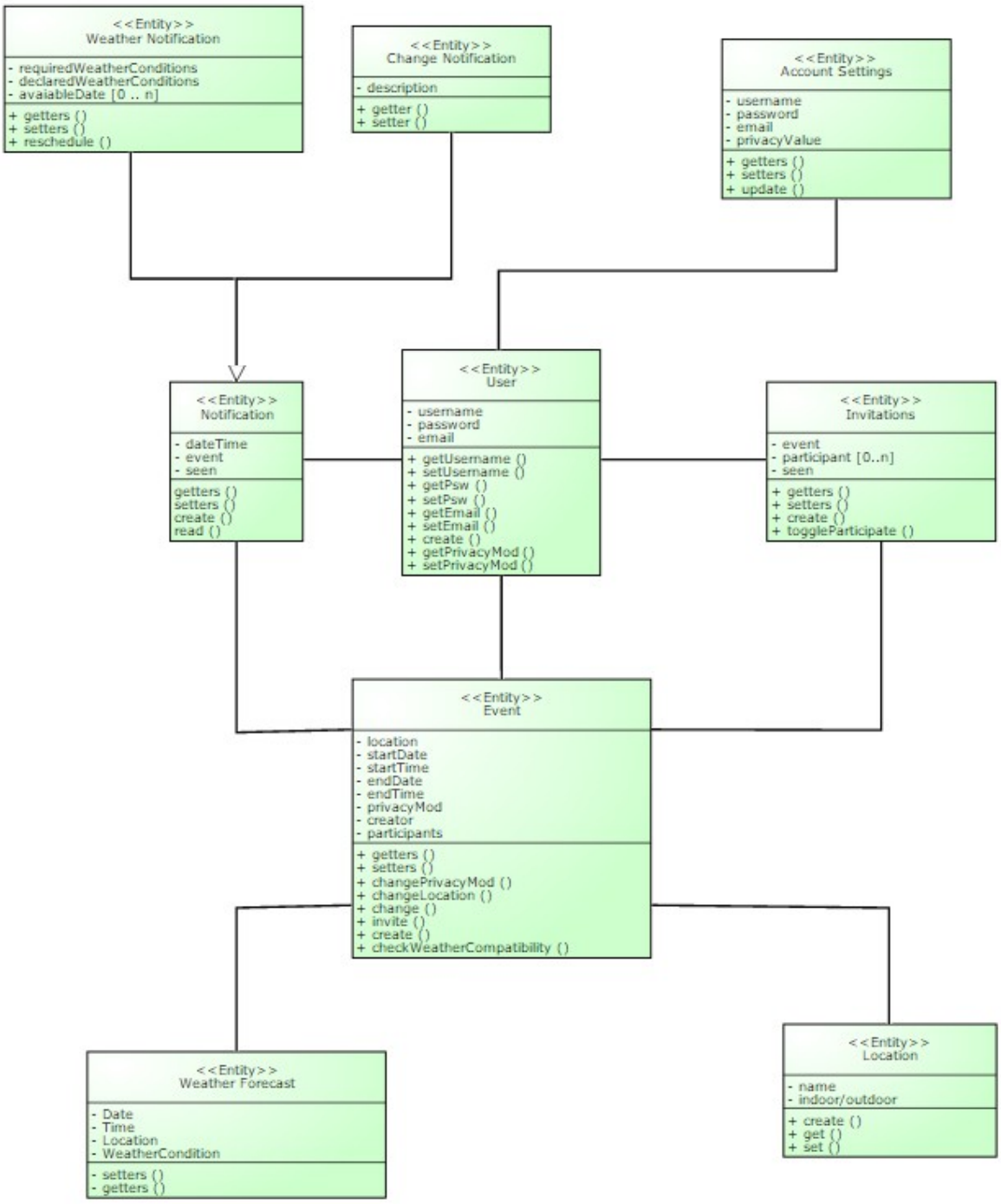
This page will display the user all the notifications he has received. Clicking on an event will bring the user to the event's page.

4. BCE Diagrams

We decided to give an additional design schema of MeteoCal in order to clarify all the possible doubts and analyse the application from different perspectives.
In particular by using the Boundary-Control-Entity schema we give a representation that is close to the Model-View-Controller design pattern. In fact the Model may be mapped to the Entity , the Control to the Controller and the View to the Boundary.
The boundaries are derived both from the Classes that were used in the Alloy specification, and according to the logical model diagram defined in this same document .
All the schemas are build starting from the UX diagrams for the screen, with the support of the Sequence Diagrams for the methods.
Then, to conclude we follow this path in order to avoid incoherences from the different schemas.

4.1 Entity Overview

Since we made schemas related to different functionalities,first we decided to give a representation of all the entities in order to make it clearer.



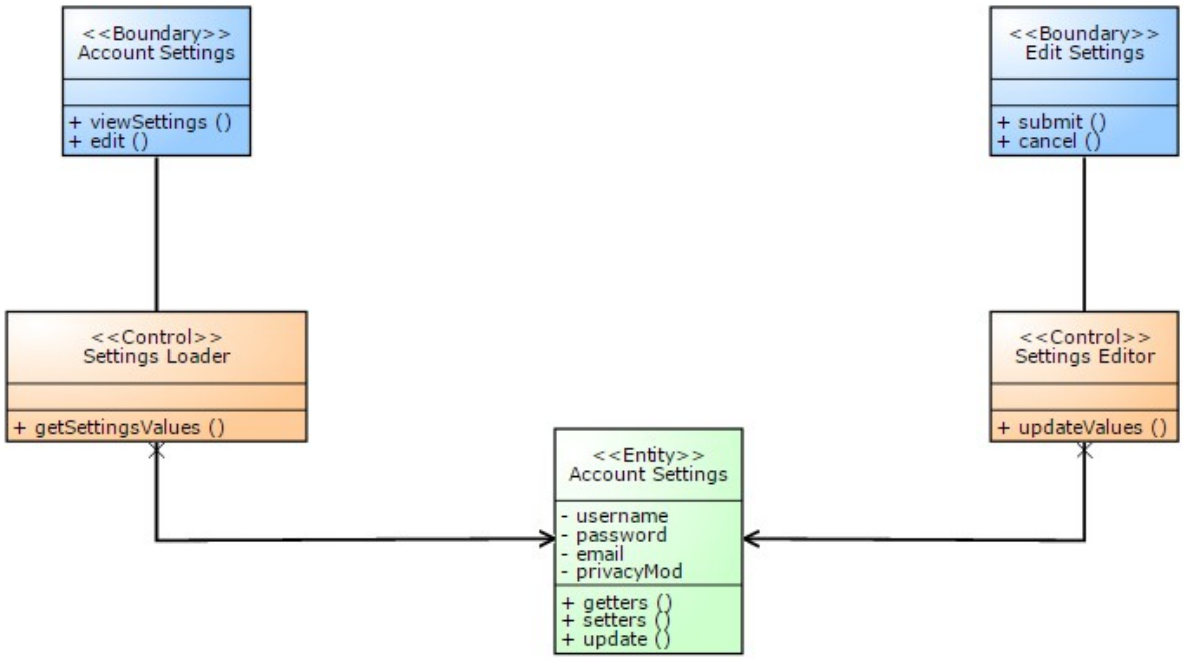
4.2 Account Settings

In this diagram the *Account-Settings boundary* represents the page in where the user may see his settings information, instead the *SettingsEdit boundary* represents the page that contains the edit form in where the user can puts the new settings values (password , mail, privacy mod).

Controllers:

Setting Loader: this controller has the duty of getting the current setting values from the database, and build the setting page with them .

Setting Editor: this controller has to get the new values submitted into the form by the user, checks their compatibility and then decide: if synchronize the database with this new vales, or, redirect the user to the edit-settings page in case of any error occurs.



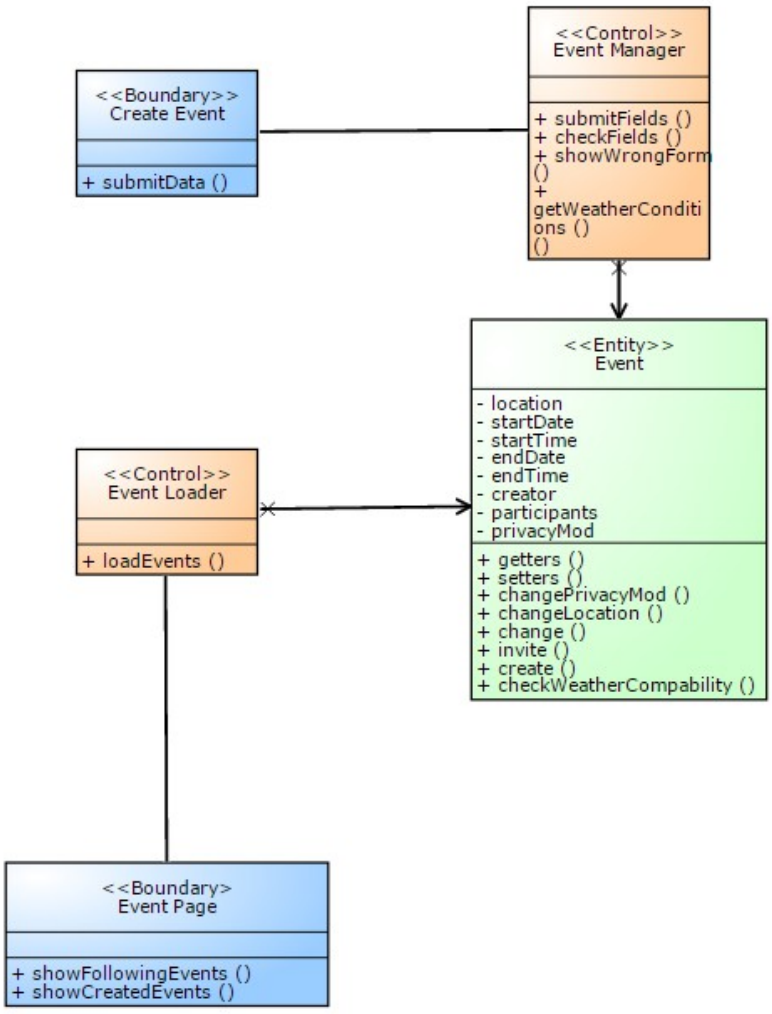
4.3 Event Creation

In this diagram the *EventCreate Boundary* represents the page in which there is the form that consists of the required fields needed for creating the event.
According to the rules defined by the control for the fields, if the event is realizable then the control redirect you to the *EventsPage Boundary.*
As regards as the controller, we have:

Controllers:

Event Manager: this controller manages the data submitted form the form by the user, and check if they define a realizable event, (for example the user must put a data value in the StartingData field). According to this policy, the control may redirect the user to the form page in case of not valid fields, or it might redirect the user to the created event in case of successful creation.

Event Loader: this controller has the job of loading the events page in case of successful creation. Since the user, in this particular case, is the creator, the page will be enriched with all the functionalities that the system offers to the creators of the events (e.g. edit, invite).



4.4 Event Update

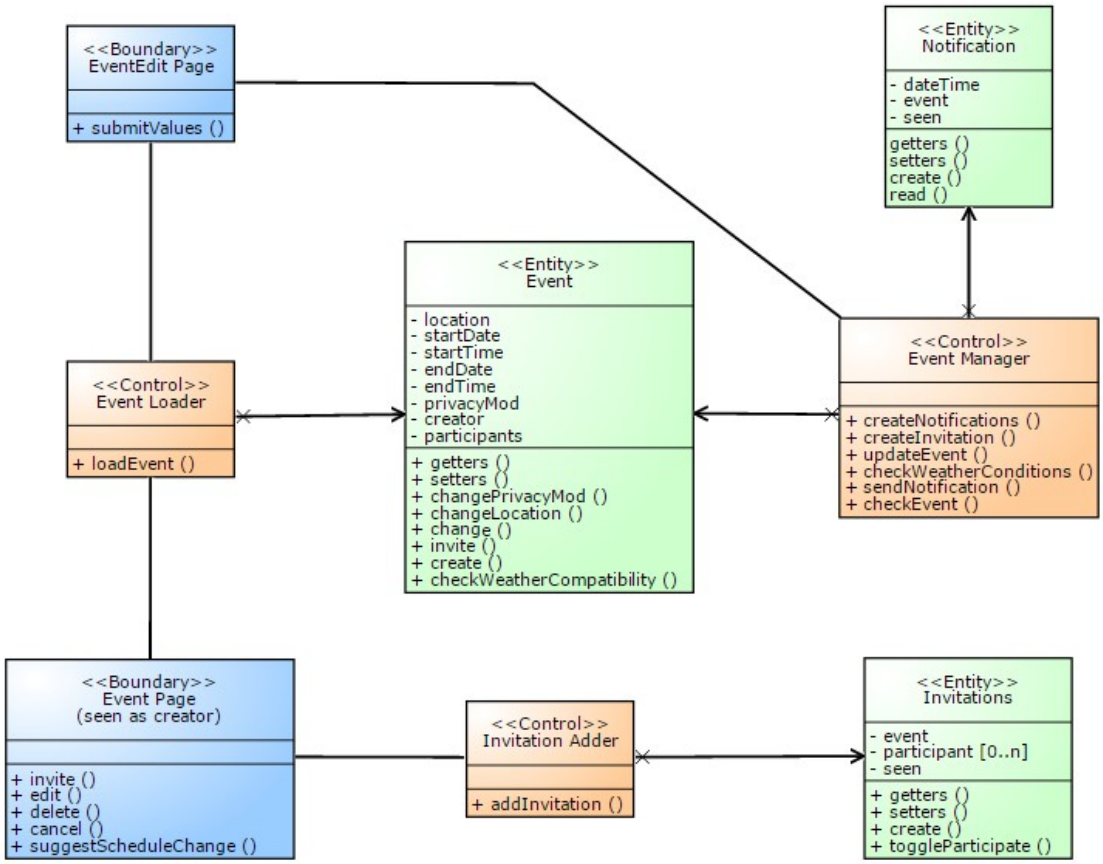
In this diagram the *EventEdit Boundary* represents the page in which there is the form that consists of the required fields needed for editing the event.
According to the rules defined by the control for the fields, if the event is realizable then the control redirect you to the *EventPage Boundary* .

Controllers:

Event Manager: this controller manages the data submitted form the form by the user, and check if they define a realizable event, (for example the user must put a data value in the StartingData field). According to this policy, the control may redirect the user to the form page in case of not valid fields, or it might redirect the user to the created event in case of successful editing.

Event Loader: this controller has the job of loading an existing event to build the event page.

Invitation Adder: this controller manages the invitation for the current event, in fact from the related page the user can add more participants to the event. Once he adds a new participant this controller get his data and send him an invitation, and synchronize the event's participants list in the database.



4.5 EventUser Research

In this diagram the *ResultsPage Boundary* represents the page in which there are all the results of the previous research. This means that there are both the events and the users, that can be found using the keywords typed in the research bar.

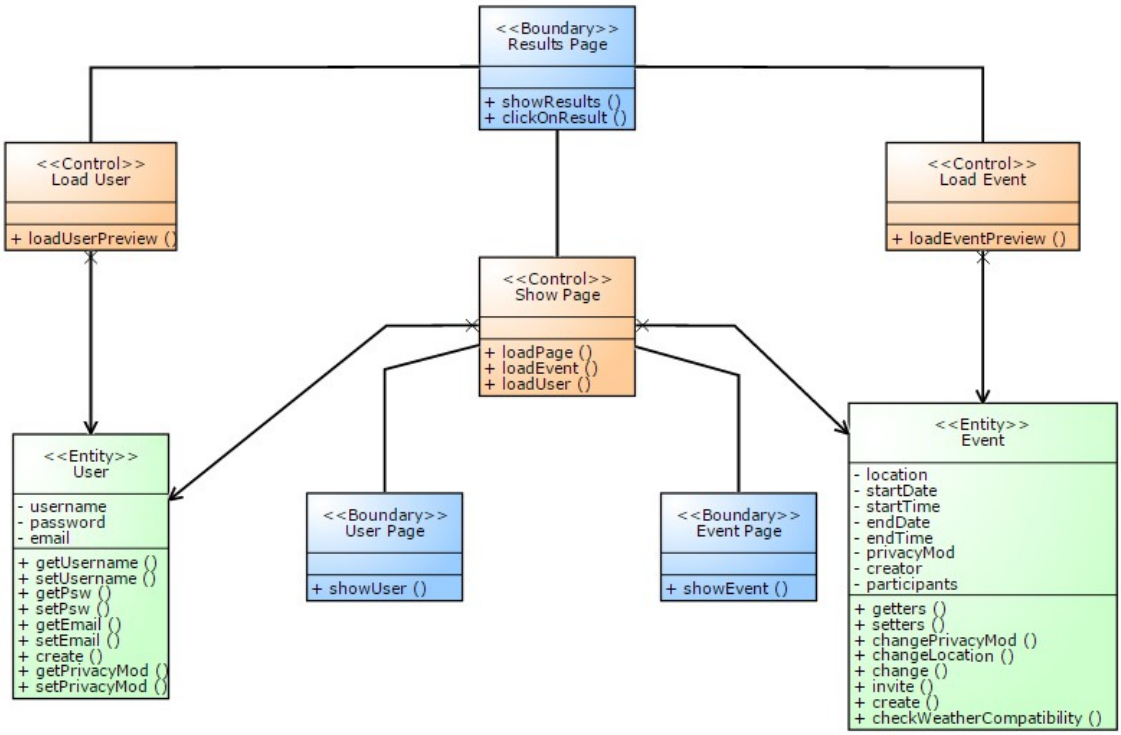
Once this page is loaded, the user may click on one item, then he's redirected to the item's page that could be either an event or an user page, which they respectively correspond to the *EventPage Boundary* and to the *UserPage Boundary* .

Controllers:

ShowPage: this controller has the job of loading the results page based on the typed string, that means it has to check in the database among the users and the events, all the compatible records and then it builds the results page with them.

UserPage Loader: this controller is called once the user clicks on an user from the results page. It has only to redirect the user to the UserPage, in particular it has to check the privacy settings of the User and show only what it's wished from the searched user.

EventPage Loader: this controller is called once the user clicks on an event from the results page. It has only to redirect the user to the EventPage, in particular it has to look the privacy field of the searched event.



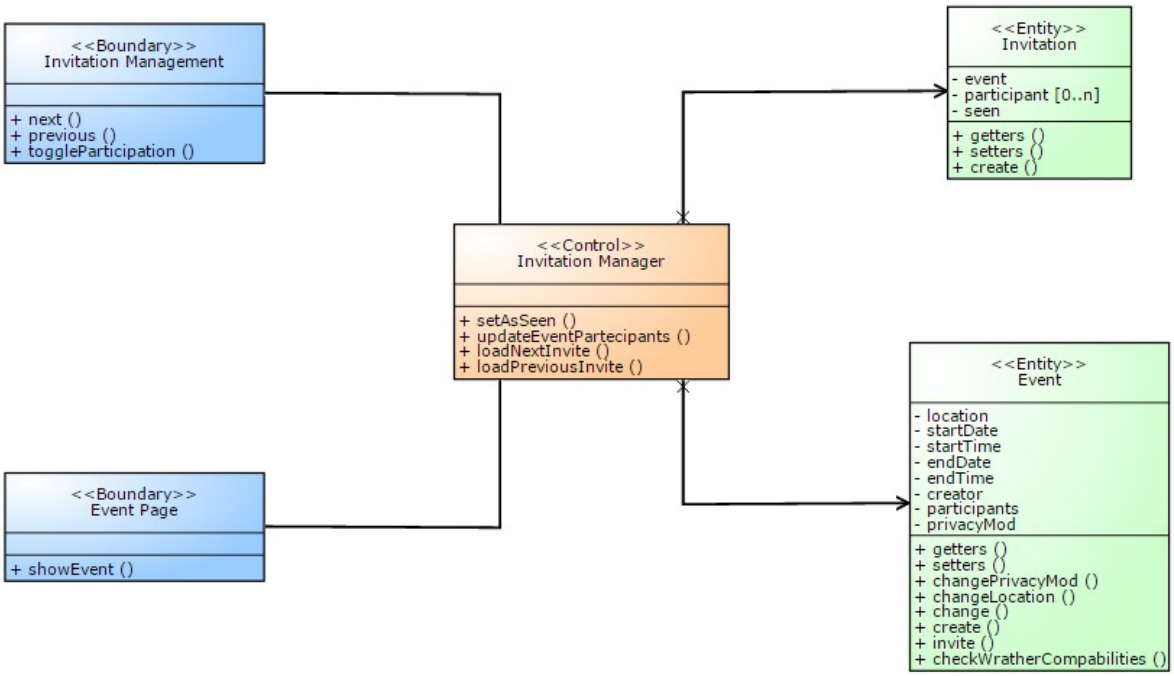
4.6 Invitation Management

In this diagram the *InvitationManagement Boundary* represents the page in which there are available all the invitations for the current user.
Once this page is loaded, the user may toggle his participation in the events he has been invited to. All the displayed invitations are also automatically set as seen.

Controllers:

Invitation Manager: Firstly this controller has the purpose of set as seen all the displayed invitations in the database. Secondly has to offered the possibility of seen the previous and the next list of invitations if the invitations are too much to be listed within one page.

It has also the job of update the database as soon as the user answers to an invitation, this operation will be performed in Ajax in order to not have to reload the page for each invitation's answer and have a more light and updated page.



4.7 Notification Management

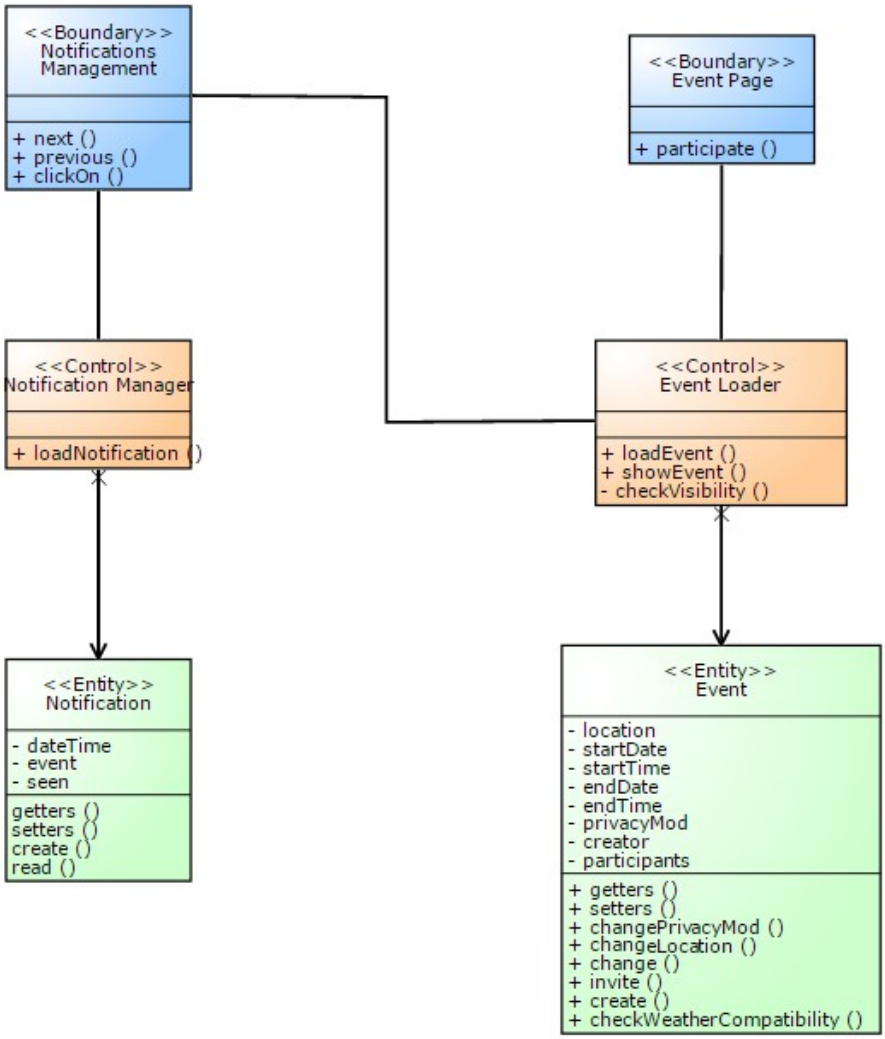
In this diagram the *Notifications Management Boundary* represents the page in which there are available all the notifications for the current user.

Once this page is loaded, the user may click on the notification and then he'll be redirected to the corresponding event page. This last page is mapped through the *EventPage Boundary*. All the displayed notifications are also automatically set as seen.

Controllers:

Notification Manager: this controller has the simple job of loading the notification page. This task consists of: build the notification page starting from all the unread notifications records in the database, that are related to the current user .

Event Loader this controller is called once the user clicks on a notification from his notifications list shown in the NotificationsPage. It has only to redirect the user to the EventPage, in particular it has to look the privacy field of the event.



4.8 Signup and Login

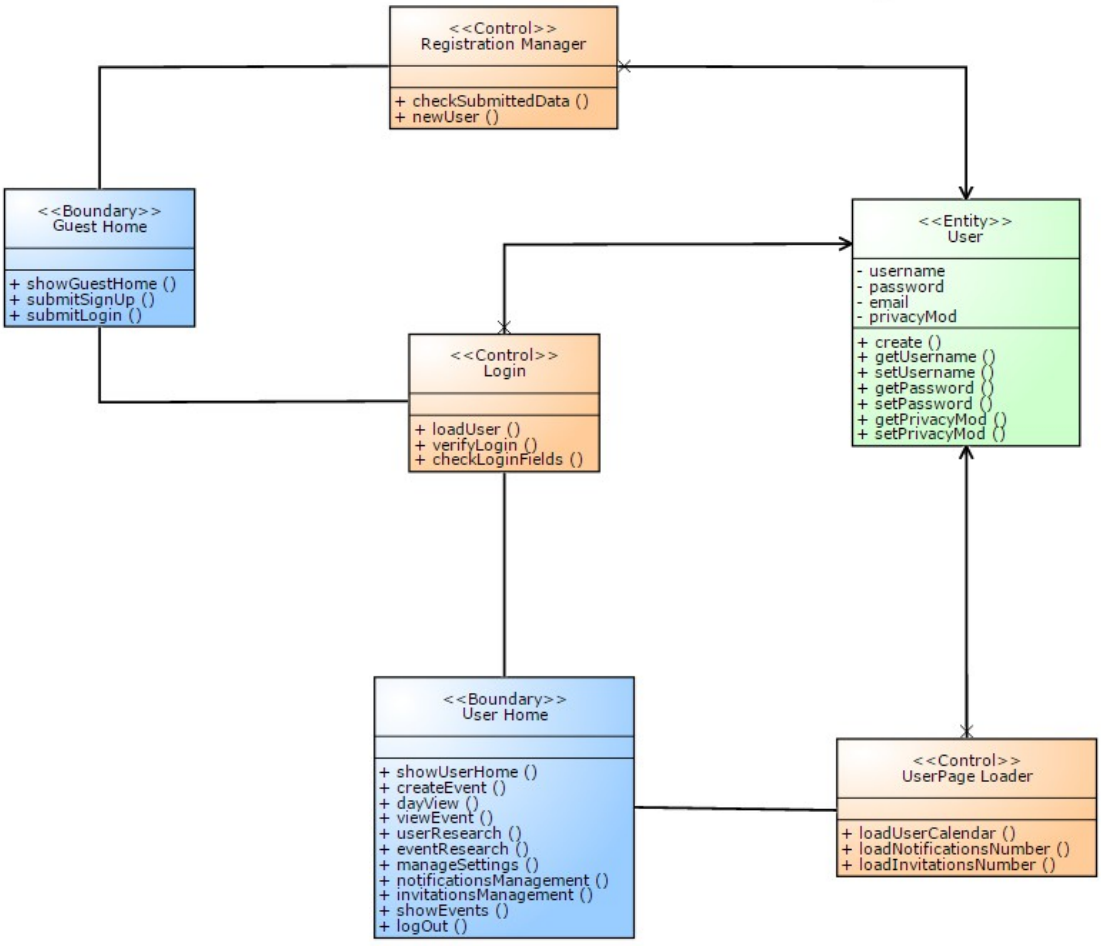
In this diagram the *GuestHome Boundary* represents the starting page of our application. From this page one may either sign up for an account or login. According to the success of this operations the guest may be redirected to his UserPage, that corresponds to the *UserHome Boundary* in the schema.

Controllers:

Registration Manager: this controller has the purpose of manage the registration task. It's has only two tasks the first one is to check the submitted fields values and evaluate if it's possible to create a new user with these data. The second one is performed only if the first task ends successfully, and has to create the corresponding record in the database.

Login :this controller is called once the user submitted his credentials in the login-form, and have to check the correctness of them. There are only two possibilities: the first one is that the insert values weren't correct, and, then the user is redirected to the same form where an error message is shown. The second case occurs when the login ends successfully and then the user is redirected to his own page.

UserPage Loader: this controller is called once the user: login rightly or his sign up task ends successfully. It has only to redirect the user to the UserPage.



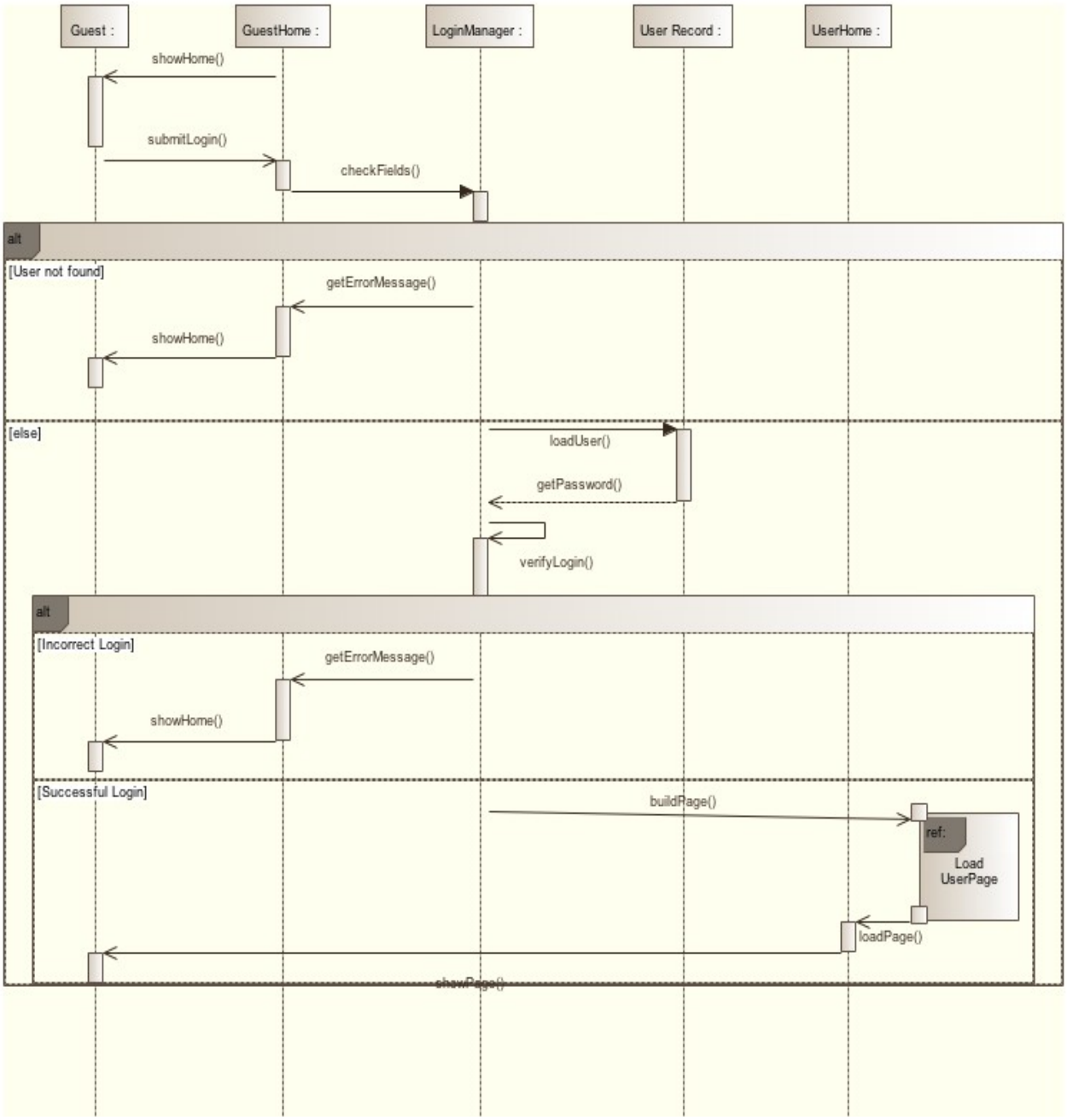
5. Sequence Diagrams

We provide some sequence diagram to let the reader better understand BCE diagrams described above.

5.1 Login

A generic user:

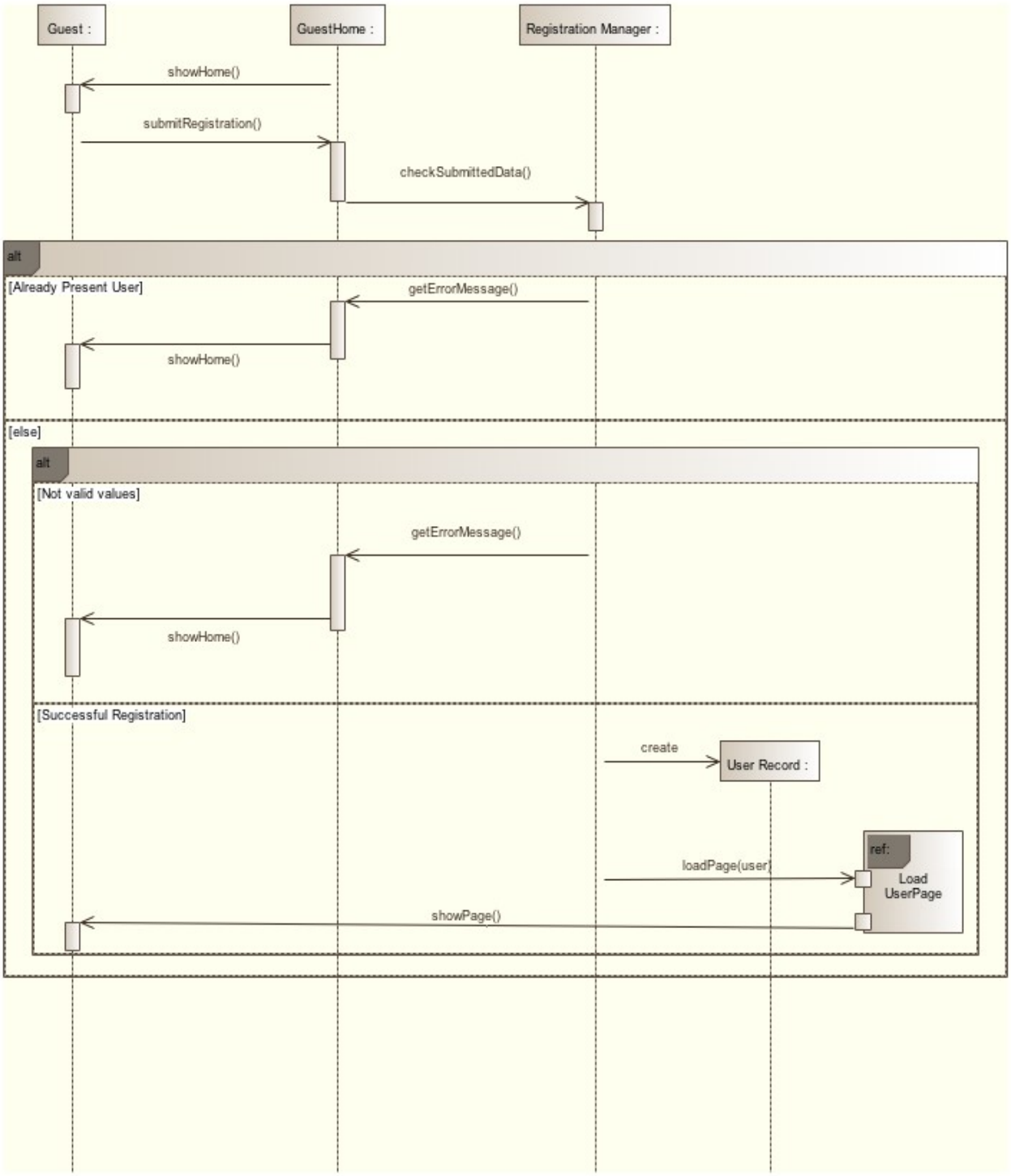
- logs in.



5.2 Registration

A guest:

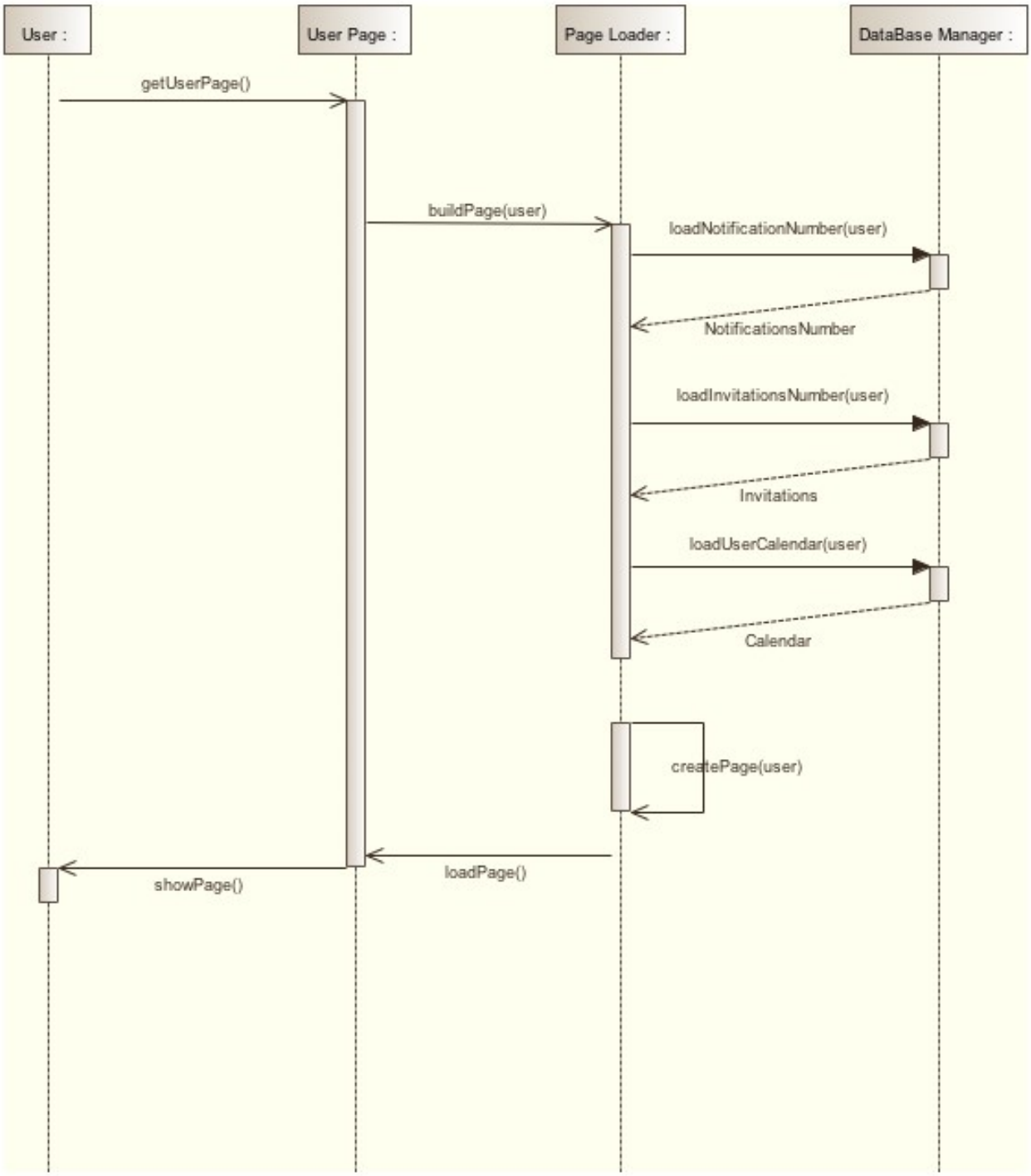
-signs up for an account.



5.3 Load Use Page

A user:

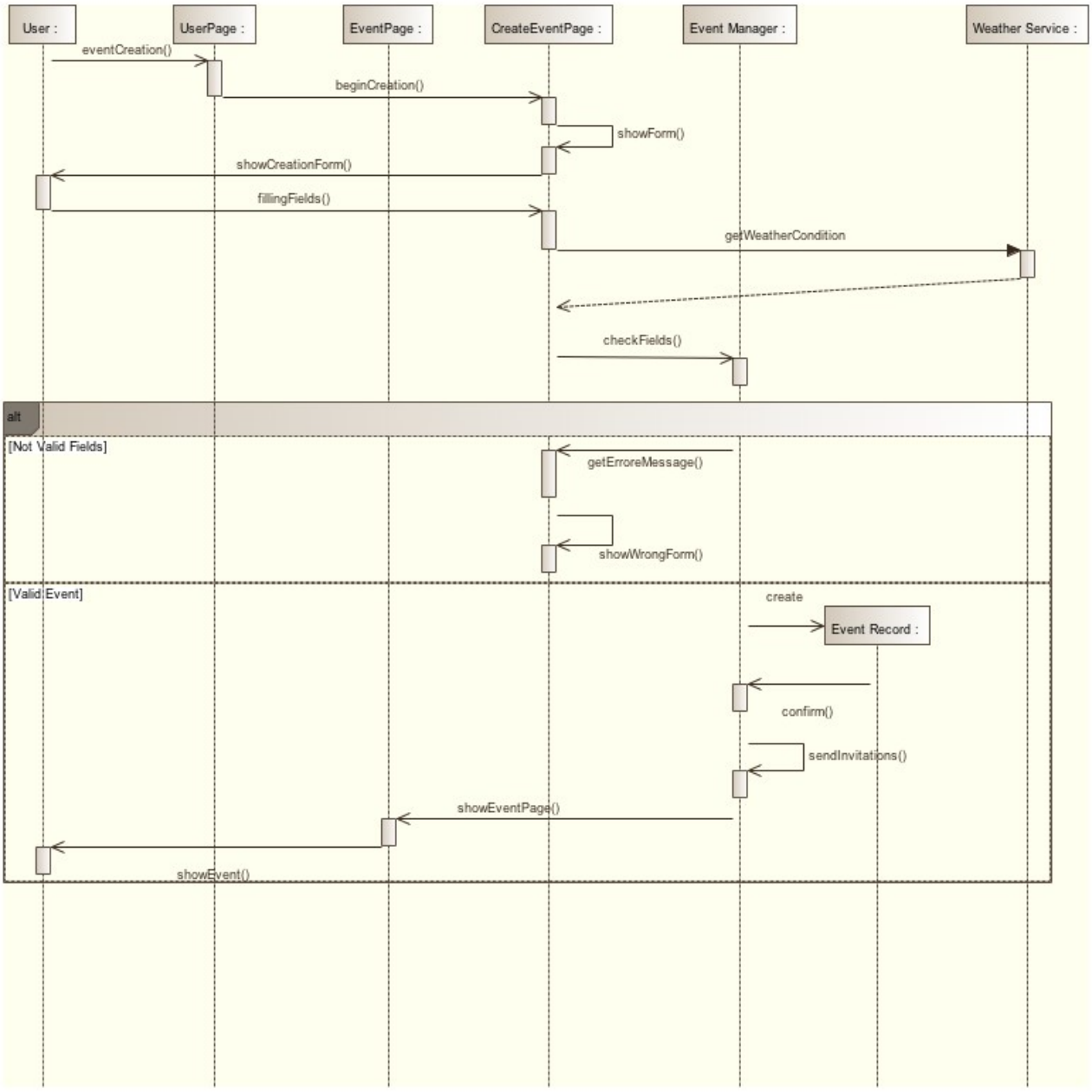
- connects to his personal page.



5.4 Event Creation

A generic user:

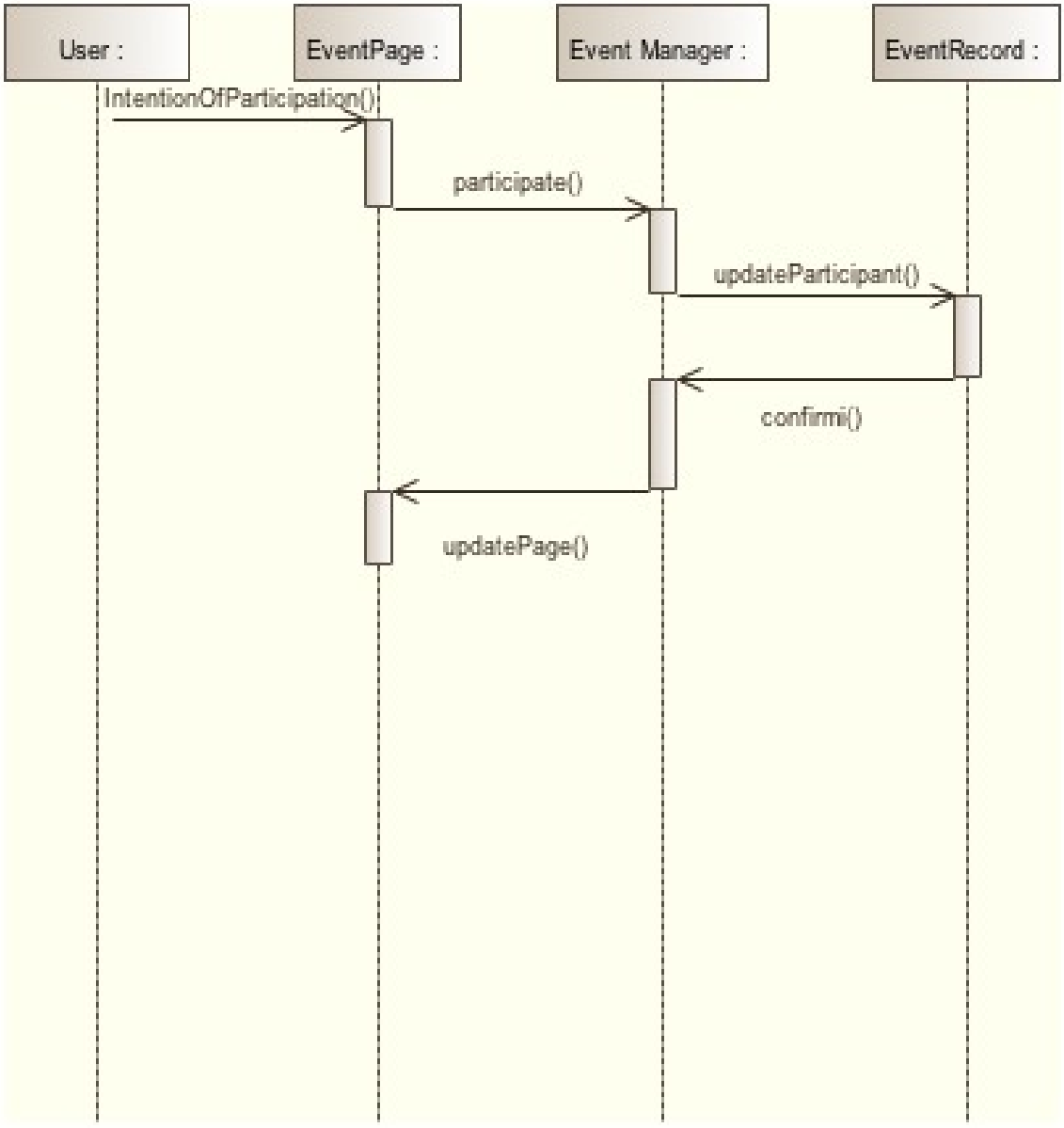
- creates a new event.



5.5 Event Participation

A generic user:

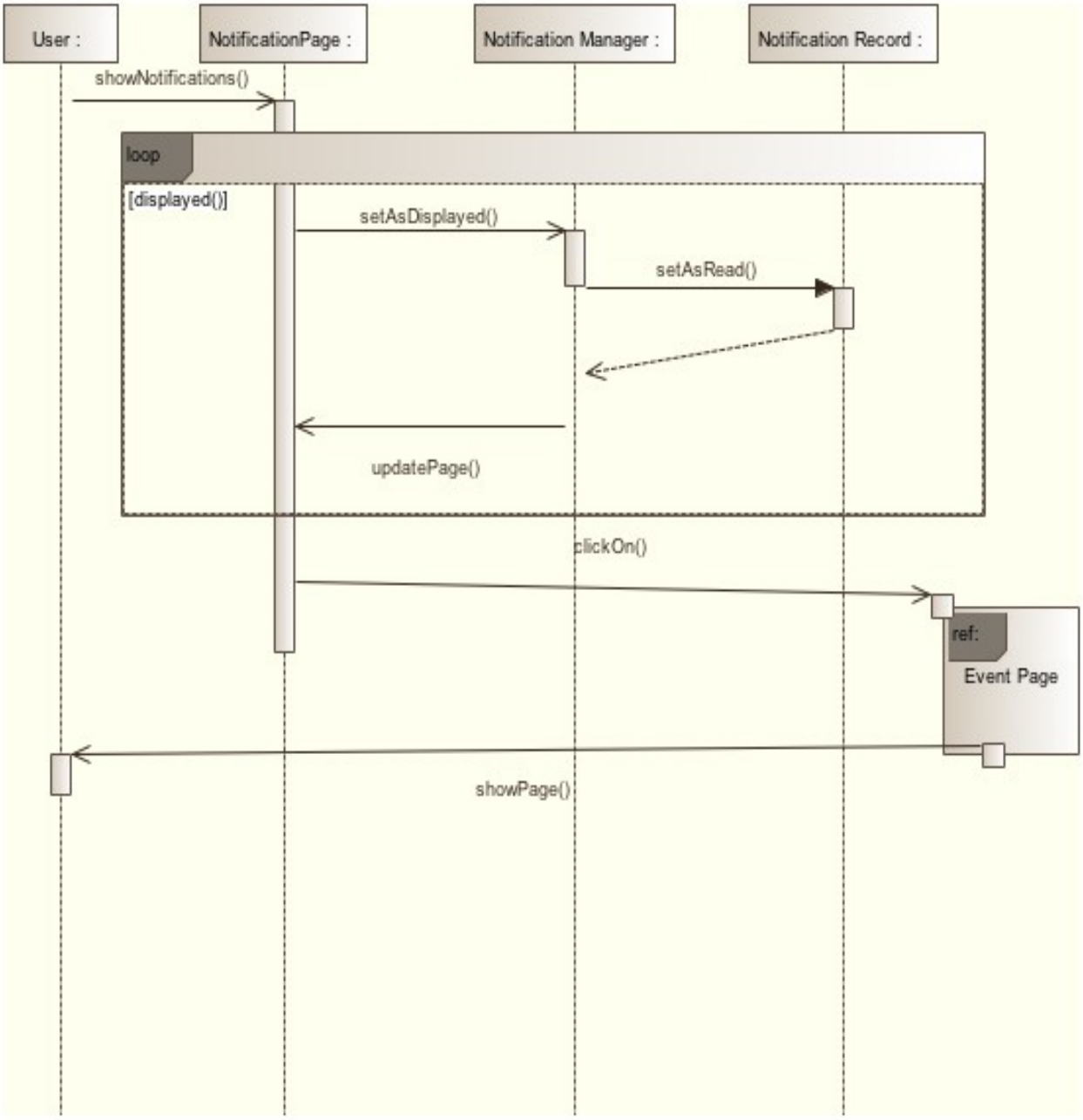
- participates to a public event.



5.6 NotificationsManagement

A generic user:

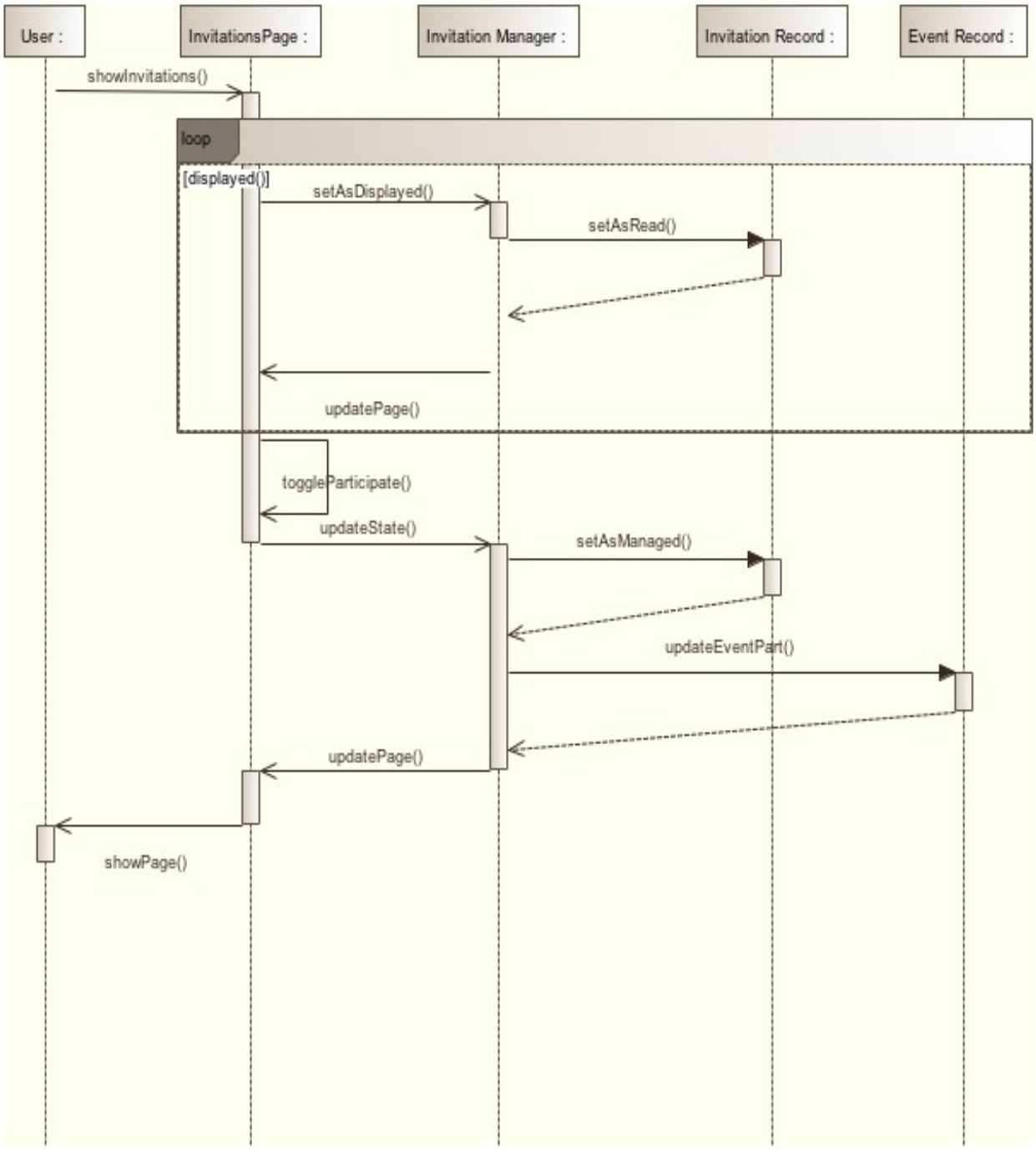
- manages his notifications.



5.7 Invitation Management

A generic user:

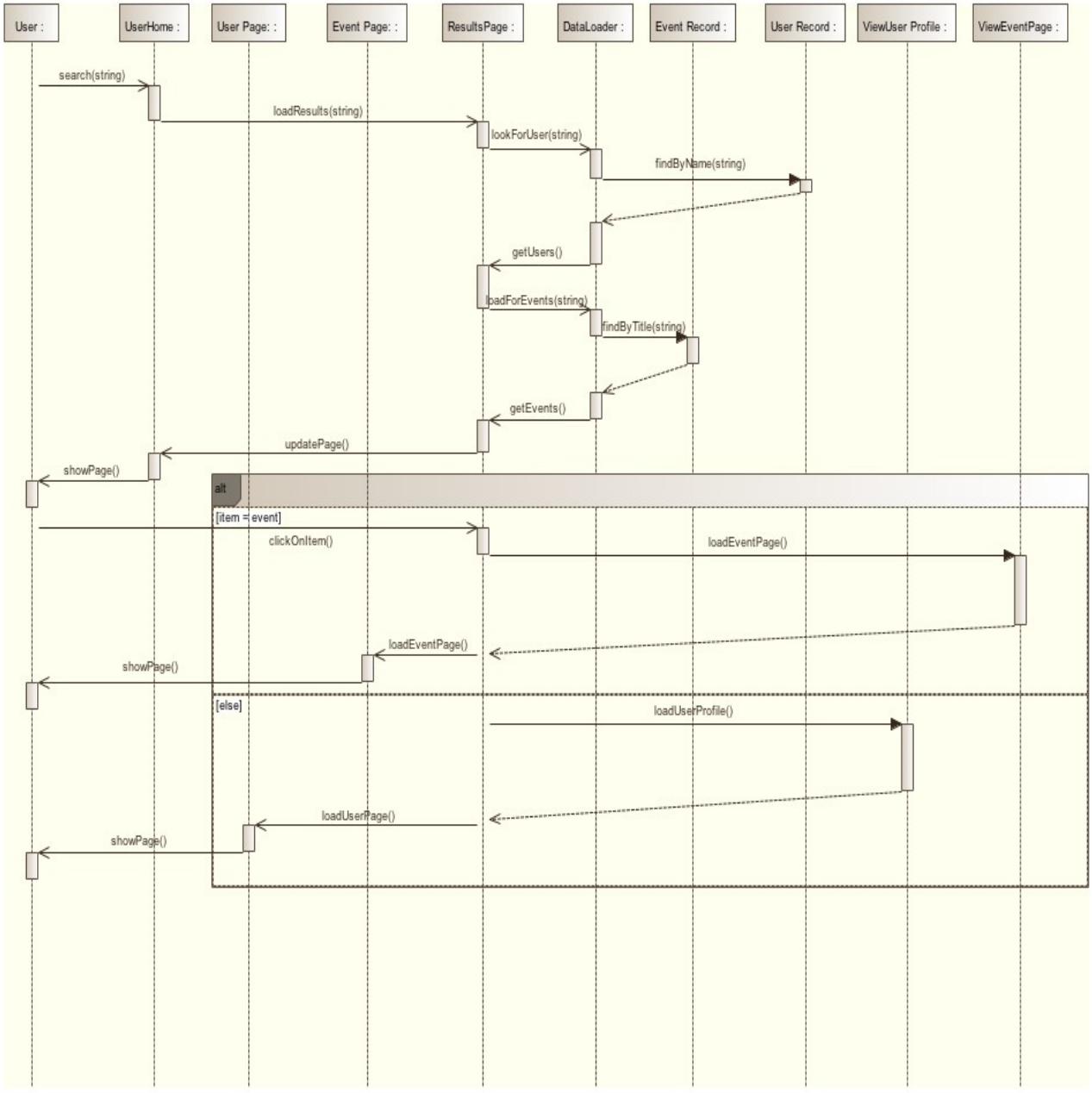
- manages his invitations requests .



5.8 Research

A generic user:

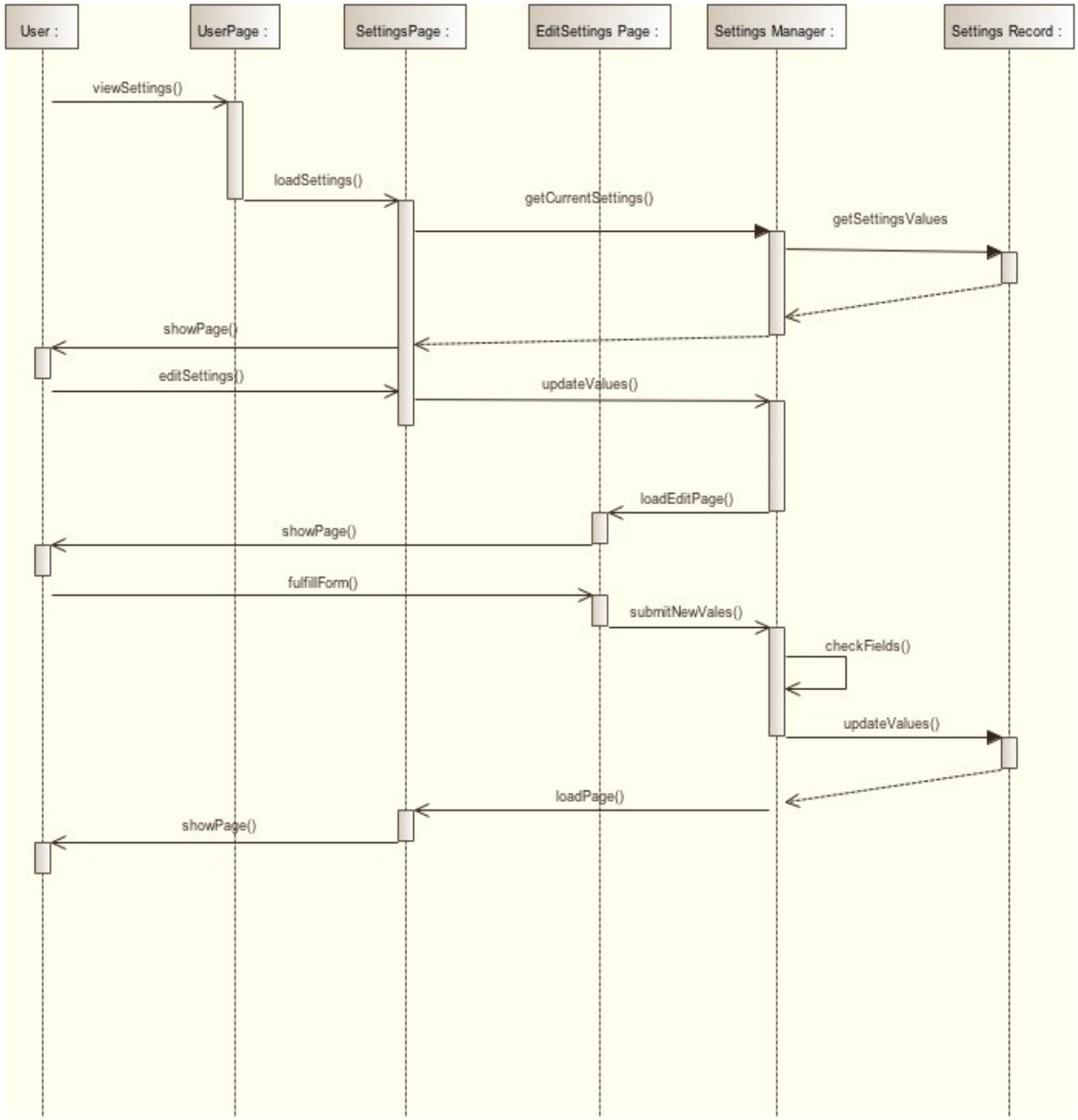
- searches for a user,
- searches for a event,
- selects a result and goes to the corresponding page.



5.9 Settings Management

A generic user:

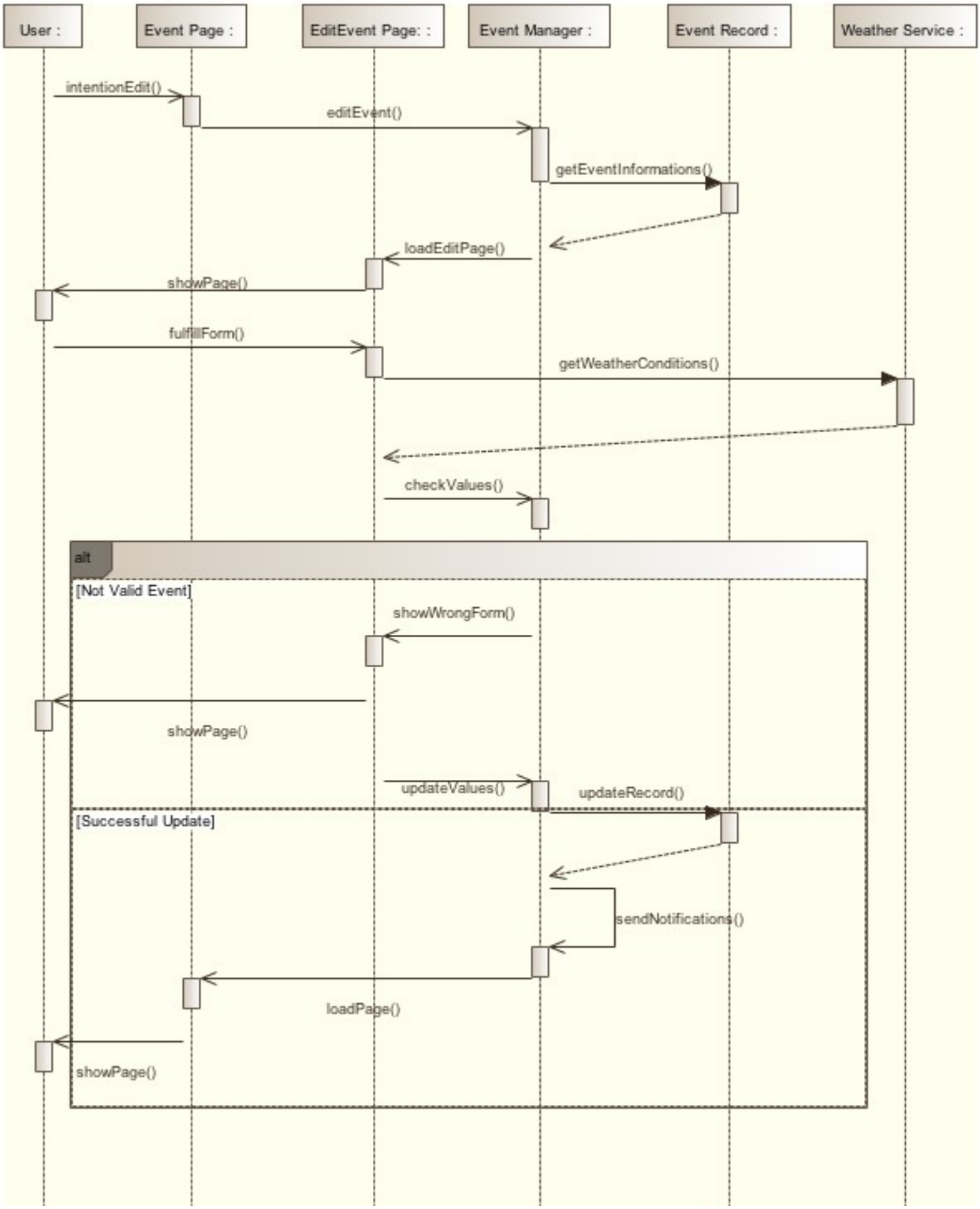
- checks his settings parameters,
- edits his settings parameters.



5.10 Update an Event

An event's creator:

- sees the current event's parameters.
- updates the event.6. Further additions



6. Further additions

6.1 Software and Tools

The tools used in the production of this document are listed below:

- *OpenOffice Writer:*

  to redact this document.

- *Modelio:*

  for the sequence diagrams;

- *Signavio Academic platform:*

  for the BCE and the UX;

- *Draw.io:*

  for the ER models;

- *Mysql Workbench:*

  for the Logical Data Model representation.

6.2 Production Report

The following report of project hours is provided to give an indication of the efficiency of this collaboration.

- *Andrea Bignoli:* 30 - 35 hours
- *Leonardo Cella:* 30 - 35 hours.

It should be noted that part of the time spent on this production was shared. We estimate the amount of shared work being around 4-5 hours.

7. Additions and corrections to RASD

The redaction of this document made us face some flaws in the previous RASD and introduced the need for some modifications to our previous specifications. It should be notice, nevertheless, that the additions listed below aren't in any way critical for the model of our system.

- Added the possibility to cancel an event and relative notifications. The event page will still be visible.
- Added the possibility to delete an event and relative notifications. The event page won't be visible anymore.
- Clarified suggested schedule changes:
  - Added a suggested change link for the creator on the event page if a suggested change is available,
  - Notifications will be sent to the creator if a suggested schedule change is available.
  - The system will compute the suggested schedule changes, if needed, from three days before the event scheduling.
- The forecasts relative to the events will be updated periodically. The time interval dividing the updates will be defined depending on the external forecast provider service.