# MyMeteoCal: DD 2.0

Design Document

**Authors:**
*Benedetto Vitale*
*Ettore Randazzo*
*Giacomo Scolari*
**Professor:**
*Raffaella Mirandola*
**Course:**
*Software Engineering 2*

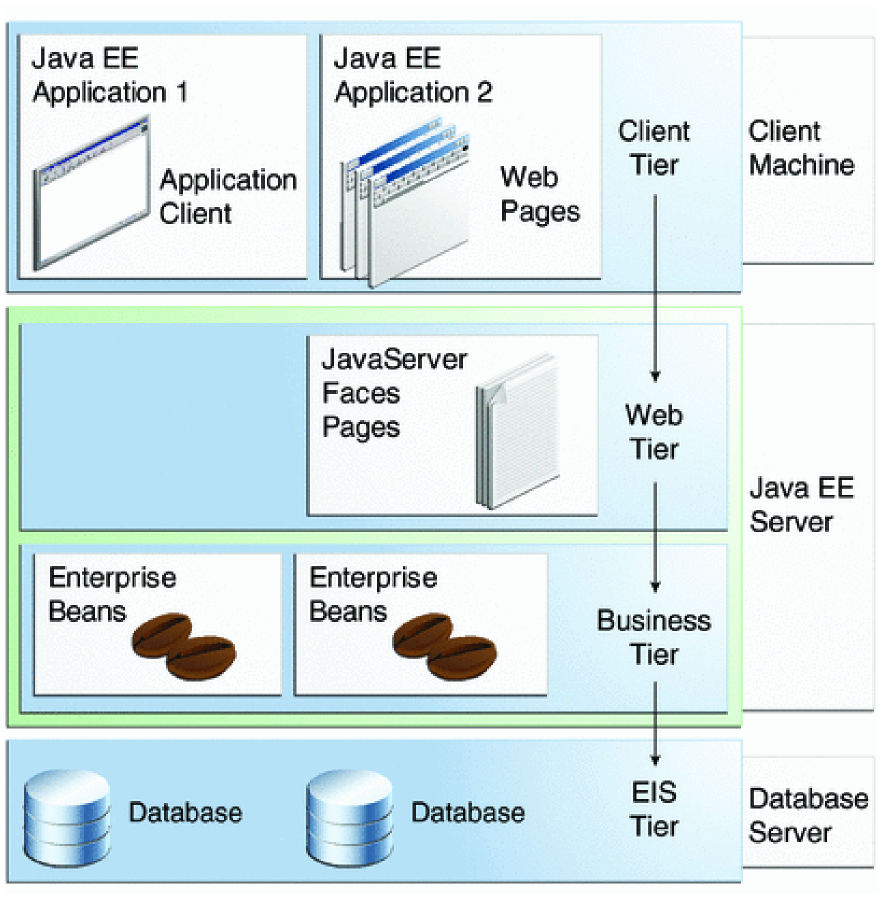
January 24, 2015

## Contents

## 1 Architecture Description

### 1.1 JEE Architecture Overview

Before starting to explain our applications architecture we want to focus on JEE Architecture:



JEE has a four tiered architecture divided as:

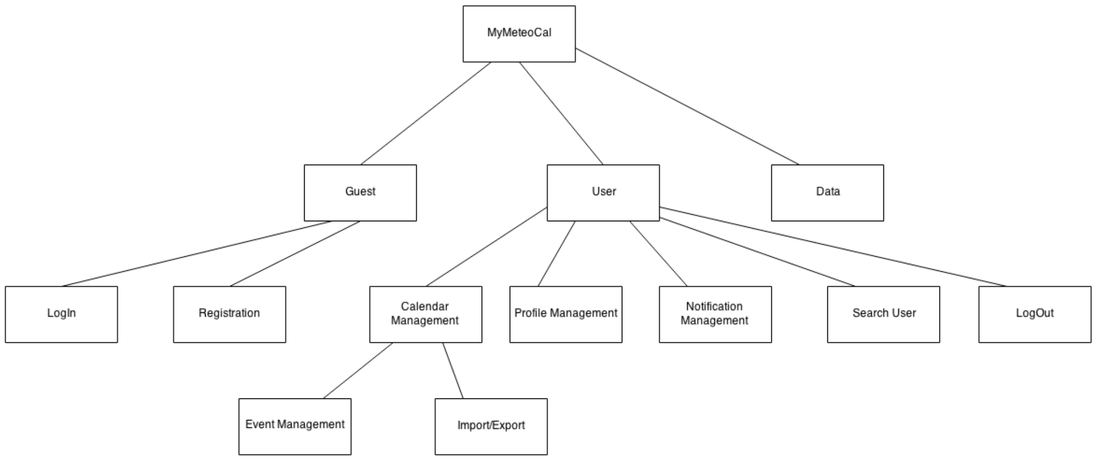
- **Client Tier:** it contains Application Clients and Web Browsers and it is the layer that interacts directly with the actors. As our project will be a web application the client will use a web browser to access pages;
- **Web Tier:** it contains the Servlets and Dynamic Web Pages that needs to be elaborated.This tier receives the requests from the client tier and forwards the pieces of data collected to the business tier waiting for processed data to be sent to the client tier, eventually formatted;

- **Business Tier:** it contains the Java Beans, that contain the business logic of the application, and Java Persistence Entities.
- **EIS Tier:** it contains the data source. In our case it is the database allowed to store all the relevant data and to retrieve them.

## 1.2 Identifying Subsystems

We decided to adopt a mostly top-down approach, meaning that it is definitely going to be thought as a top-down procedure, but we reserve the possibility to use a bottom-up approach for specific tasks in order to improve its re usability. Following this line of thought we are going to fragment our system into some blocks of functionalities, which we will call sub-systems, in order to have a better understanding of our goals and to facilitate a proper distribution of work among our team. The main sub-systems are:

- Guest
  - Log In
  - Registration
- User
  - Calendar Management
    - Event Management
    - Import/Export Calendar
  - Profile Management
  - Notification Management
  - Search User
  - Log Out
- Data (Weather Notification)

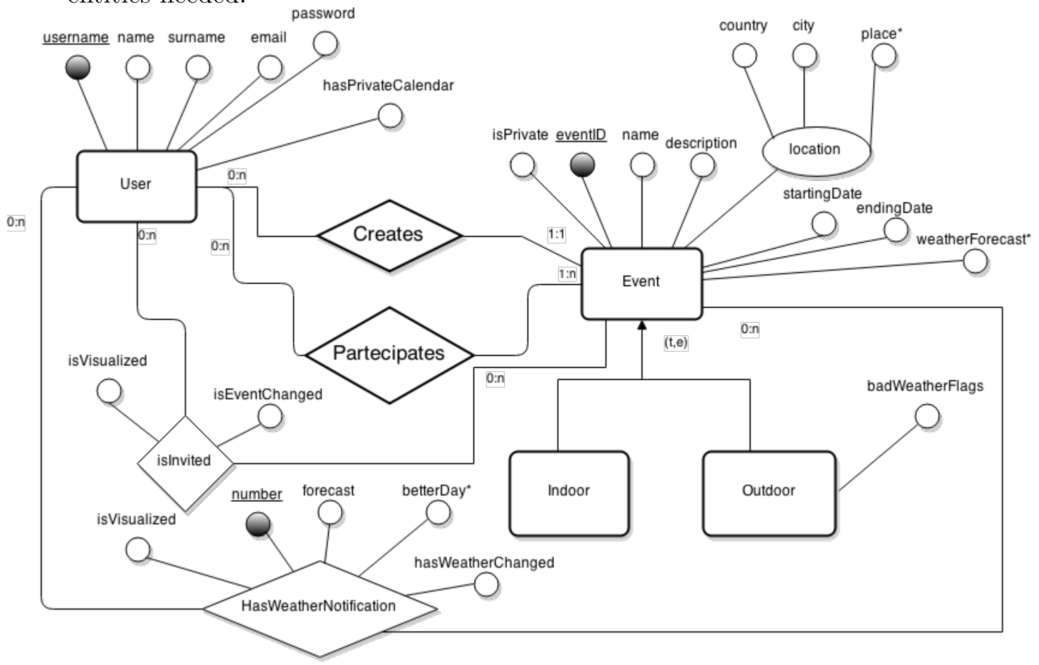


## 2 Persistent Data Management

As we are going to store our data into a relational database, it's important to define it with an Entity-Relationship diagram in order to derive an appropriate database.

### 2.1 Conceptual Design

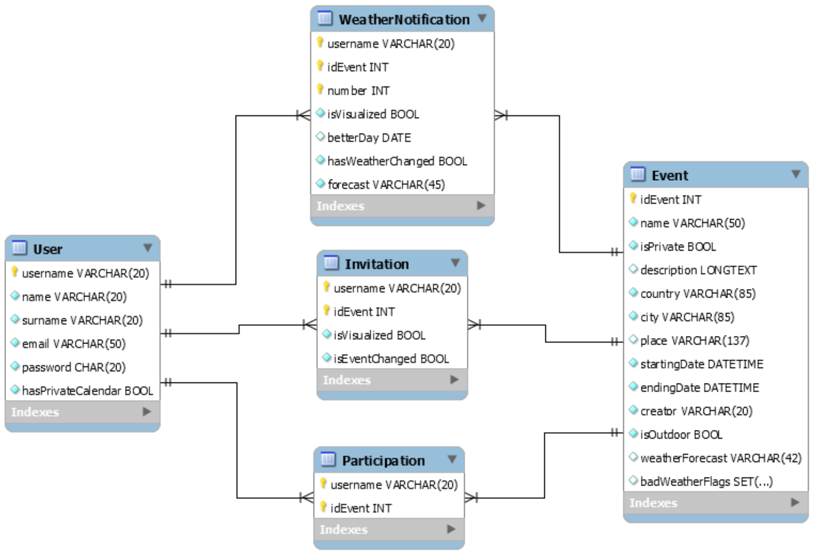
Starting with the ER diagram, we tried to express the relationship between the entities needed.



This diagram is self explaining, however we would like to clarify the reasons behind some decisions. In order to have a Calendar concerning one single user, we only need to know the events and the user, so as you can see no Calendar Entity appears. Regarding the notifications, we felt like we needed to know somehow whether one notification had already been visualized at a given time, in order to better handle later the notifications' logic. So we decided to add a flag attribute on both notification types to define their states. Also, even if it is an anticipation of the actual database mapping, we needed to add a number in the weather notifications, because we want to ensure the possibility to have multiple WeatherNotifications regarding one couple (Event,User), because we realized it might be troublesome or counter-intuitive to just leave the most recent notification regarding a Weather change, when there might be multiple changes and one might not connect for various days (And so, what? Imagine if User A checks its event and it's said that it is going to be sunny, then 12 hours later a Bad Weather is predicted and again after 12 hours sun is predicted again. It might be confusing without holding the history of the Weather changes to realize what that "It's going to be sunny"-like notification's intentions were!). Of course, such problem doesn't exist for Invitations, where we want to limit at most one invitation for each couple (Event,User).

## 2.2 Logical Design

Moving forward, the mapping between this ER diagram and the database structure is straight-forward and pretty much standard. The only things worth mentioning are the (obvious and again standard) fact that the attribute location, made by 3 sub-attributes, has been split into just the 3 attributes, and that the event entity, which had two children, different only for a Weather forecast regarding the Outdoor child, has been generalized adding a boolean in order to differentiate the two fragments (The weatherForecast was optional to begin with, so it remains optional, of course). Also, in our real implementation we are going to use single badWeather flags for each kind of weather, but it is irrelevant from a conceptual point of view.



Plus, we are now able to define the logical schema of our database, relative to the E-R diagram presented before:

- **User**(<u>username</u>,name,surname,email,password,hasPrivateCalendar)
- **Event**(<u>eventID</u>,name,description,isPrivate,isOutdoor,country,city,place*,weatherForecast*, startingDate,endingDate,creator_username)
- **Participation**(<u>username</u>,<u>eventID</u>)

- **Invitation**(username,eventID,isVisualized,isWeatherChanged)
- **Weather_Notification** (username,eventID,number,isVisualized,hasWeatherChanged,forecast)

## 3 User Experience

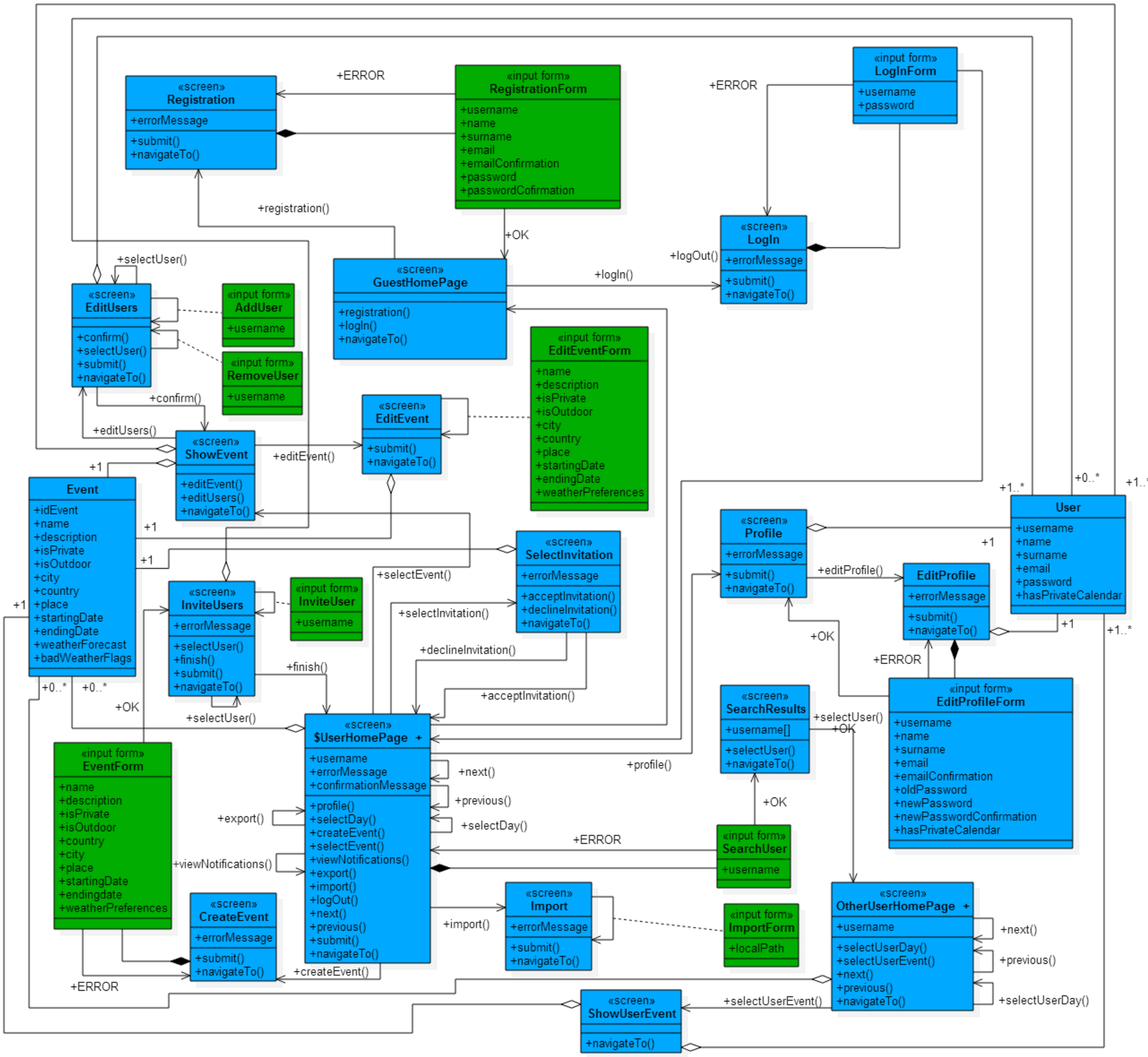
In this paragraph we are going to describe the User Experience given by our system to its users. We made a standard UX diagram, with the usual <<screen>> and <<input form>> pages. We decided not to use any <<screen compartment>> page, given the fact that no functionalities are really shared in different pages, also the Calendar, which is made in UserHomePage and in OtherUserHomePage, is done in different ways and offers different functionalities. As it is a huge diagram, it is fully shown on the next page, however we recommend to observe it now before continuing the explaining.

As it is self-explaining, and as most of if has already been explained in the RASD document, we believe it doesn't need many redundant clarifications, however we want to stress some few points.

- A guest, who we remember is meant to be somebody who hasn't logged in, can only visit three pages: GuestHomePage (the one which is most likely to be opened at first), LogIn and Registration. Any attempt done by a Guest in order to get to another page (by its URL), is going to be redirected (We still haven't decided whether we want a redirection to the GuestHomePage or to a specific page saying he needs to login, but we don't think it is a vital matter for an Alpha Version). We decided to split the login page and the registration page, it is just a stylish choice, and after logging in, the old-guest becomes a new-user (if you are familiar with DB triggers you should understand the "notation"), being redirected to the real home page.
- The UserHomePage is the real "home" (marked with a dollar), with this we mean that *after logging in*, from every user page (the ones not belonging to the three guest pages) users can click a certain button to return home.
- Every time another user has to be selected, be it either for looking at their calendar and inviting them to an event, there are two steps involving this functionality: The user writes a name and a list of users having that name (the "name" given is a substring of different unique usernames), then the user selects a single user from them and the regarding action is taken (we don't think it is going to be too hard to implement, if that is the case, we might change this part). Also, notice that when inviting users to an event, this operation has to be done several times.
- The calendar is dynamic. It shows only a month, so we need to have next() and previous() methods in order to switch months. Selecting a day the regarding events are visualized next to the calendar. Also, one

of the most important differences between the two calendars is that while visiting another user's calendar, it might be private and no events could be seen, likewise for its events.

- Whenever an user wants to see their notifications, he clicks a button on his home page. When he does so, a list of notifications appears, if he selects a weather notification, a popup is shown, explaining what is that notification about, if he selects an invitation, a new page is shown, where he can look at the info of the event and accept or decline the invitation.


«screen» Registration
+errorMessage
+submit()
+navigateTo()
«input form» RegistrationForm
+username
+name
+surname
+email
+emailConfirmation
+password
+passwordCofirmation
+ERROR
+registration()
+OK
«input form» LogInForm
+username
+password
+ERROR
«screen» LogIn
+errorMessage
+submit()
+navigateTo()
+logOut()
+logIn()
«screen» GuestHomePage
+registration()
+logIn()
+navigateTo()
+selectUser()
«screen» EditUsers
+confirm()
+selectUser()
+submit()
+navigateTo()
«input form» AddUser
+username
«input form» RemoveUser
+username
+confirm()
+editUsers()
«screen» ShowEvent
+editEvent()
+editUsers()
+navigateTo()
+editEvent()
«screen» EditEvent
+submit()
+navigateTo()
«input form» EditEventForm
+name
+description
+isPrivate
+isOutdoor
+city
+country
+place
+startingDate
+endingDate
+weatherPreferences
Event
+idEvent
+name
+description
+isPrivate
+isOutdoor
+city
+country
+place
+startingDate
+endingDate
+weatherForecast
+badWeatherFlags
+1
+1
+1
+1
+0..*
+0..*
«screen» SelectInvitation
+errorMessage
+acceptInvitation()
+declineInvitation()
+navigateTo()
+selectEvent()
+selectInvitation()
+declineInvitation()
+acceptInvitation()
«screen» InviteUsers
+errorMessage
+selectUser()
+finish()
+submit()
+navigateTo()
«input form» InviteUser
+username
+finish()
+selectUser()
+OK
«screen» $UserHomePage +
+username
+errorMessage
+confirmationMessage
+profile()
+selectDay()
+createEvent()
+selectEvent()
+viewNotifications()
+export()
+import()
+logOut()
+next()
+previous()
+submit()
+navigateTo()
+next()
+previous()
+selectDay()
+export()
+viewNotifications()
+profile()
«input form» EventForm
+name
+description
+isPrivate
+isOutdoor
+country
+city
+place
+startingDate
+endingdate
+weatherPreferences
«screen» CreateEvent
+errorMessage
+submit()
+navigateTo()
+createEvent()
+ERROR
«screen» Import
+errorMessage
+submit()
+navigateTo()
+import()
«input form» ImportForm
+localPath
«screen» Profile
+errorMessage
+submit()
+navigateTo()
+editProfile()
«screen» EditProfile
+errorMessage
+submit()
+navigateTo()
+OK
+ERROR
User
+username
+name
+surname
+email
+password
+hasPrivateCalendar
+1..*
+0..*
+1..*
+1
+1
+1..*
«input form» EditProfileForm
+username
+name
+surname
+email
+emailConfirmation
+oldPassword
+newPassword
+newPasswordConfirmation
+hasPrivateCalendar
«screen» SearchResults
+username[]
+selectUser()
+navigateTo()
+selectUser()
+OK
+OK
«input form» SearchUser
+username
+ERROR
«screen» OtherUserHomePage +
+username
+selectUserDay()
+selectUserEvent()
+next()
+previous()
+navigateTo()
+next()
+previous()
+selectUserDay()
+selectUserEvent()
«screen» ShowUserEvent
+navigateTo()

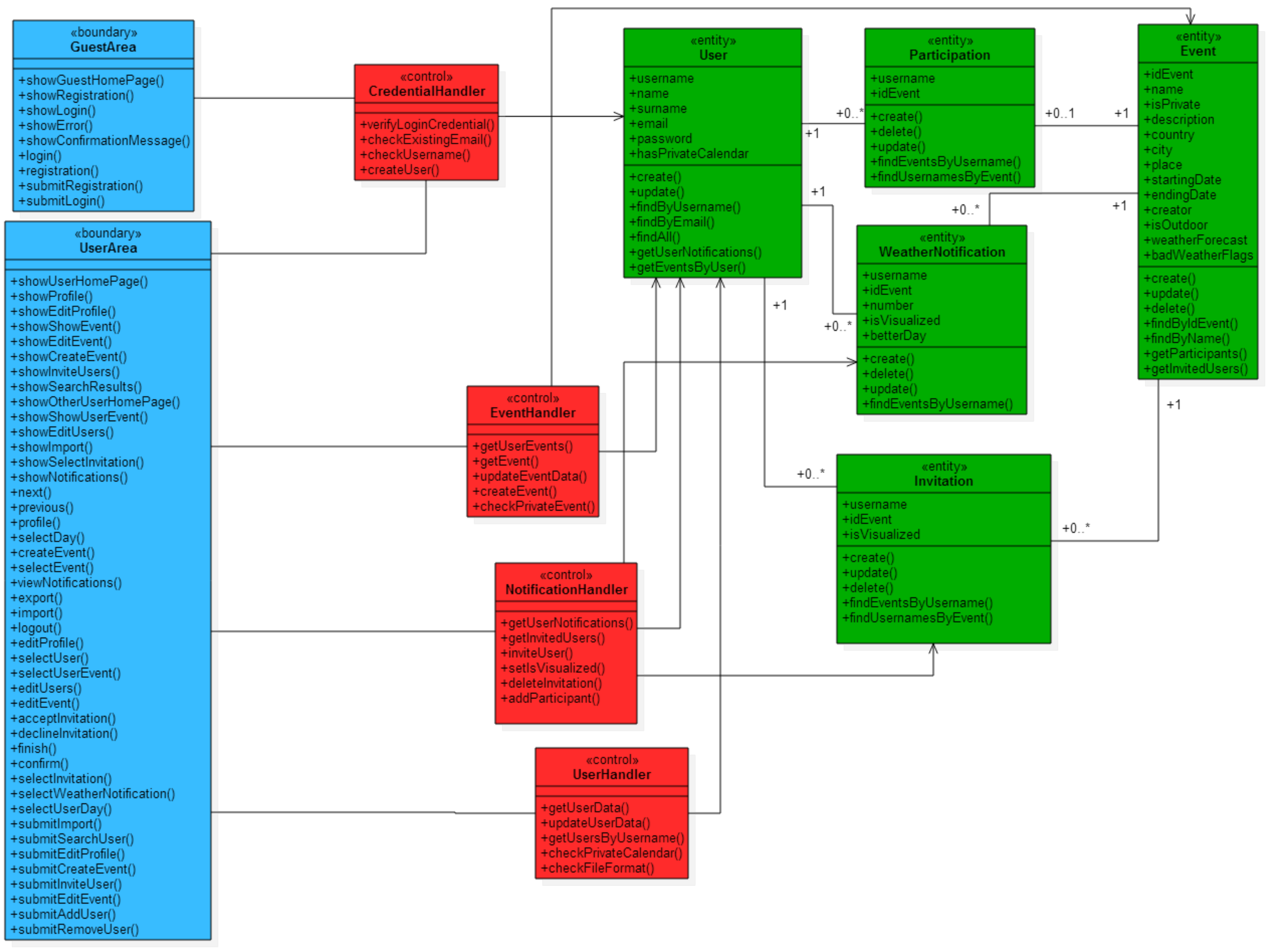
## 4 BCE Diagrams

Now we are going to explain the relation between the front-end and the back-end of our system using a Boundary-Control-Entity diagram. Notice that the boundary's role is to communicate with the person, as such for every screen in the UX diagram, a "show*thatScreen*" method has to exist, also, the boundary has to ensure that every use case can be done by invoking some methods (which makes things easier to explain, as you already know those use cases), the control's role is the core of our system, by which we ensure the actual execution of the functionalities, and the entities are, by our choice, the exact representation of our real database, as it was already small enough to not be confusing, adding some methods needed to retrieve information needed by the control. As for the UX diagram, we recommend to have a look at the BCE diagram before reading the clarifications.

Despite most of it should be clear enough, we still want to clarify some stylish decisions.

- There are only two boundaries. The GuestArea is reasonably small, so there shouldn't be any problem with that, the UserArea, however, might seem odd. We thought of splitting it into smaller parts, but none of them were convincing, firstly because most of them would be overlapping at least from the "show" point of view, secondly because we believe that all the things doable by a user are in a sense connected and related, so we decided to keep them altogether.
- Moving to the control, things are different. We really believed what we said about the UserArea, still nothing forces us not to split the control into different areas of interest.
- We are not explicitly saying that errors occur when users write incorrect values in their input form. That's just because we consider it as an exception thrown by the method interested in those values to be correct, it's not a method itself, exception made for the import functionality, where we really didn't know where to implicitly say it but to add a charming "checkFileFormat()" method. Also, consider we are going to use Ajax, so no incorrect values *should* be queried.

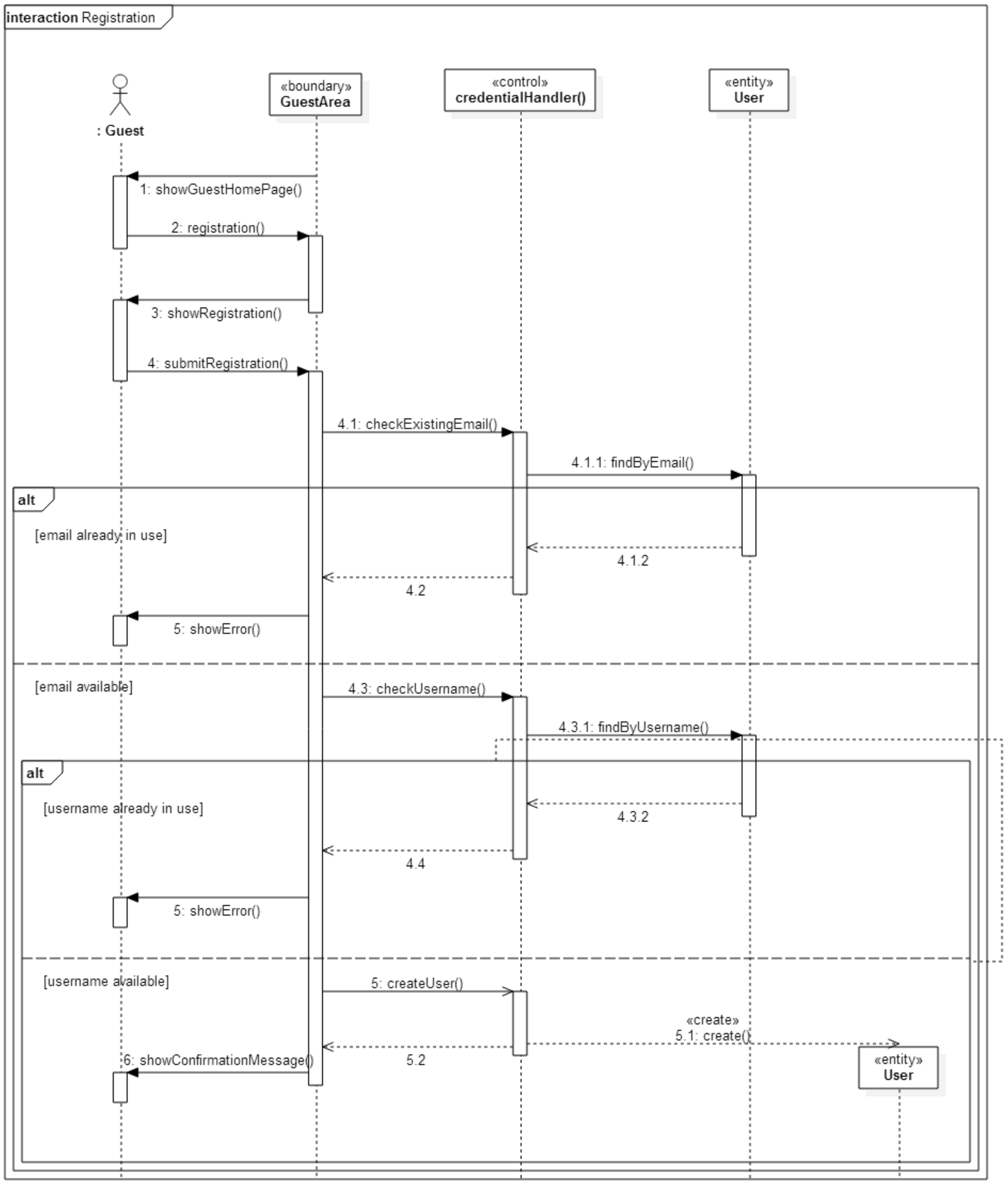
As a last comment, notice that everything regarding the weather forecasts is not part of the BCE, as the user doesn't need to interact with it. The observing of such requirements is going to be done by the actual implementation.



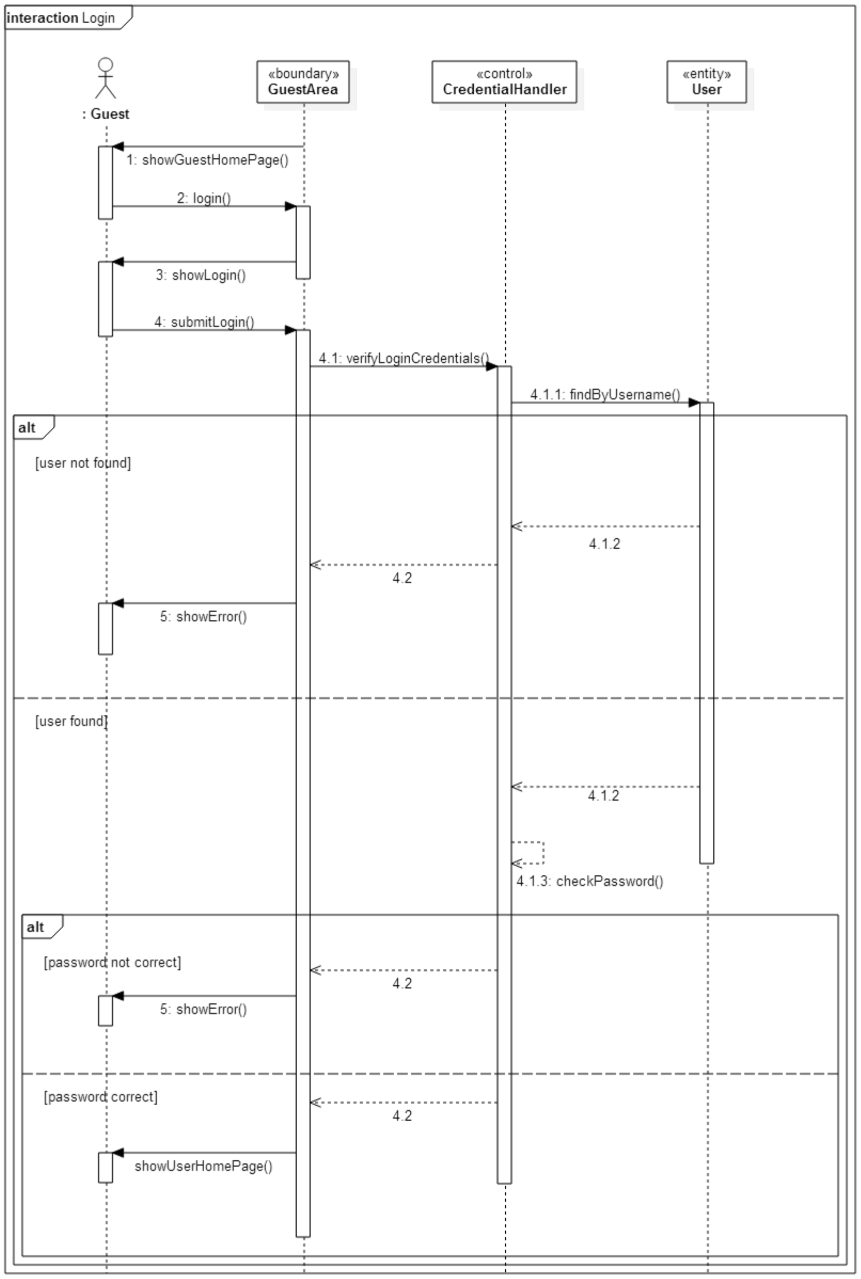
## 5 Sequence Diagrams

We now add a more detailed version of the most relevant sequence diagrams in order to better clarify the dynamics of our application at this specification level:

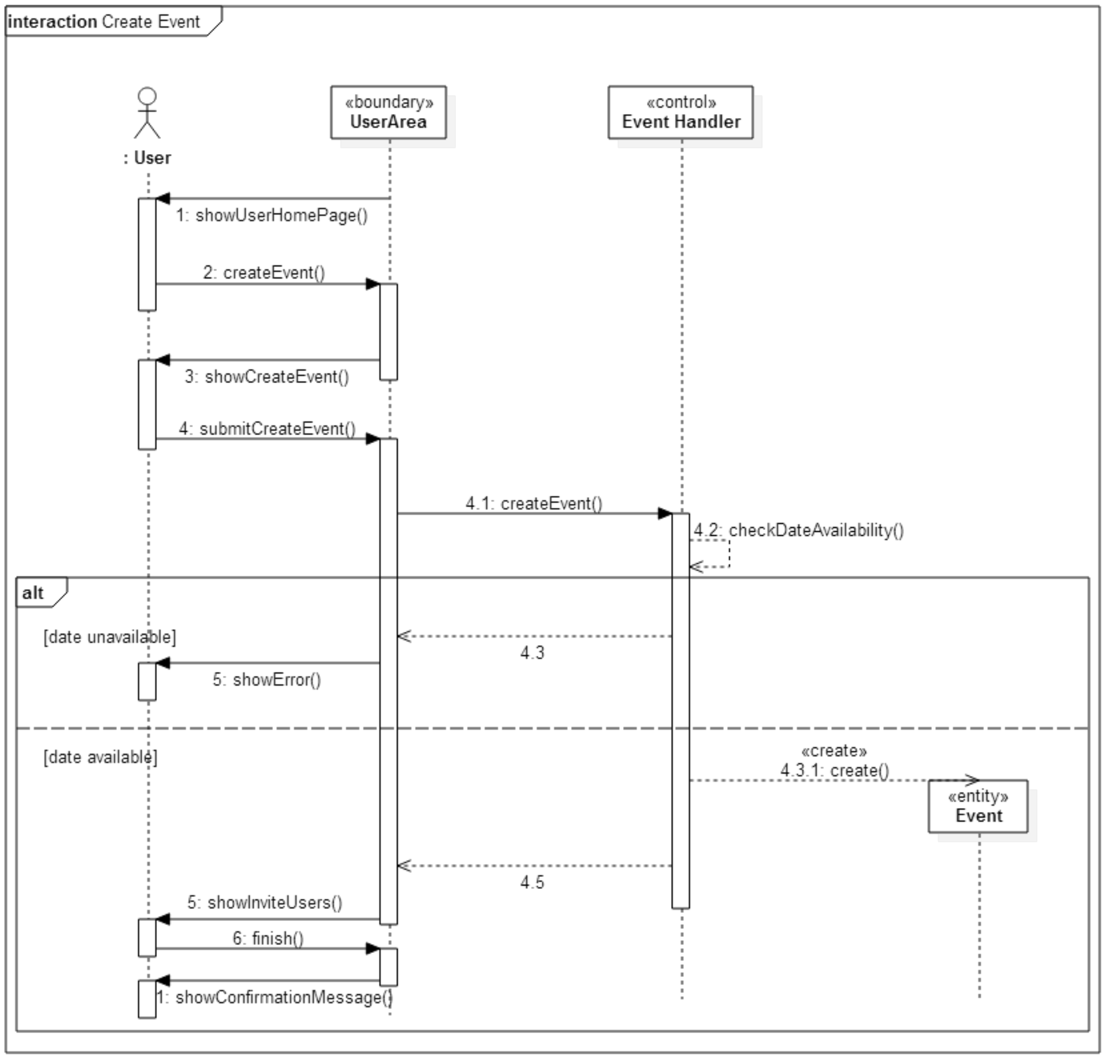
- **Registration**



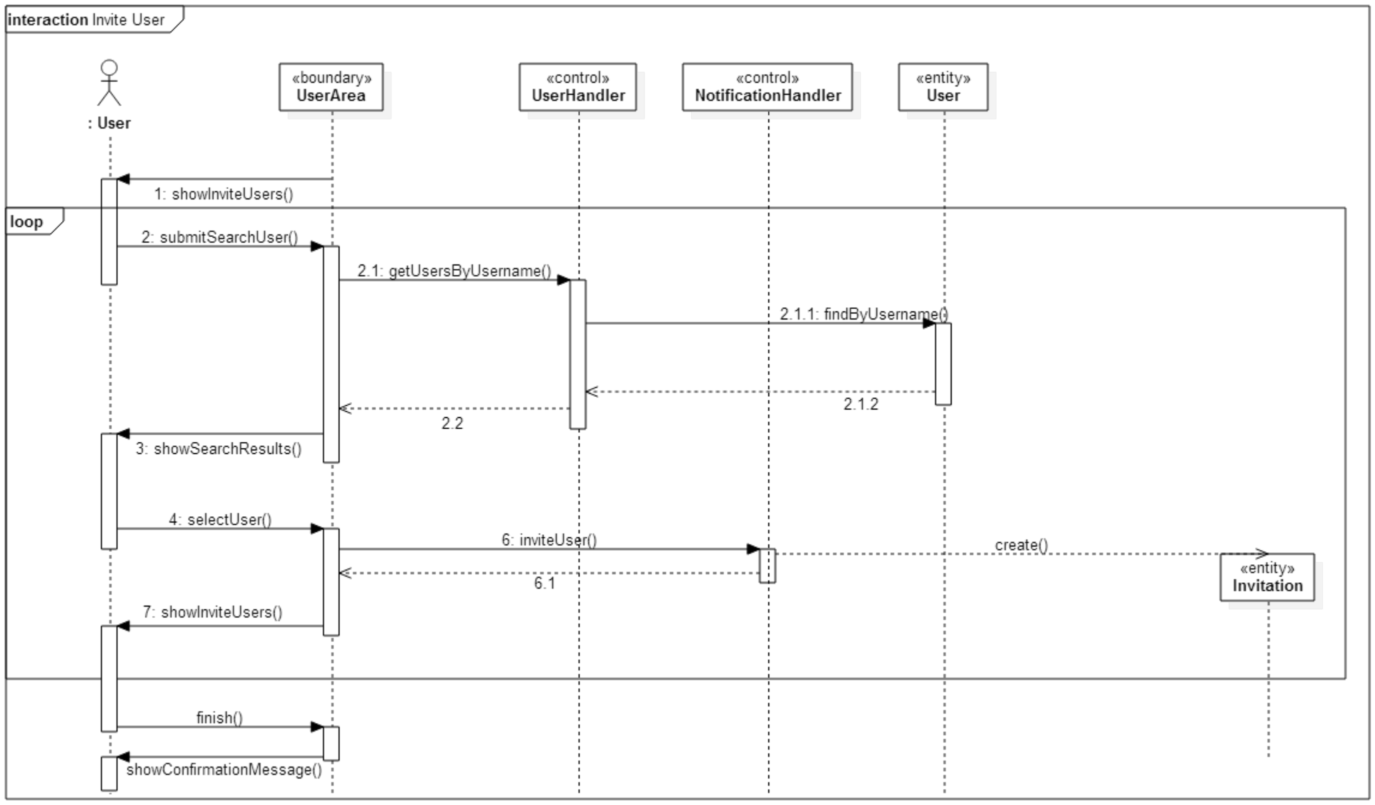
- **Login**


interaction Login
: Guest
«boundary» GuestArea
«control» CredentialHandler
«entity» User
1: showGuestHomePage()
2: login()
3: showLogin()
4: submitLogin()
4.1: verifyLoginCredentials()
4.1.1: findByUsername()
alt
[user not found]
4.1.2
4.2
5: showError()
[user found]
4.1.2
4.1.3: checkPassword()
alt
[password not correct]
4.2
5: showError()
[password correct]
4.2
showUserHomePage()

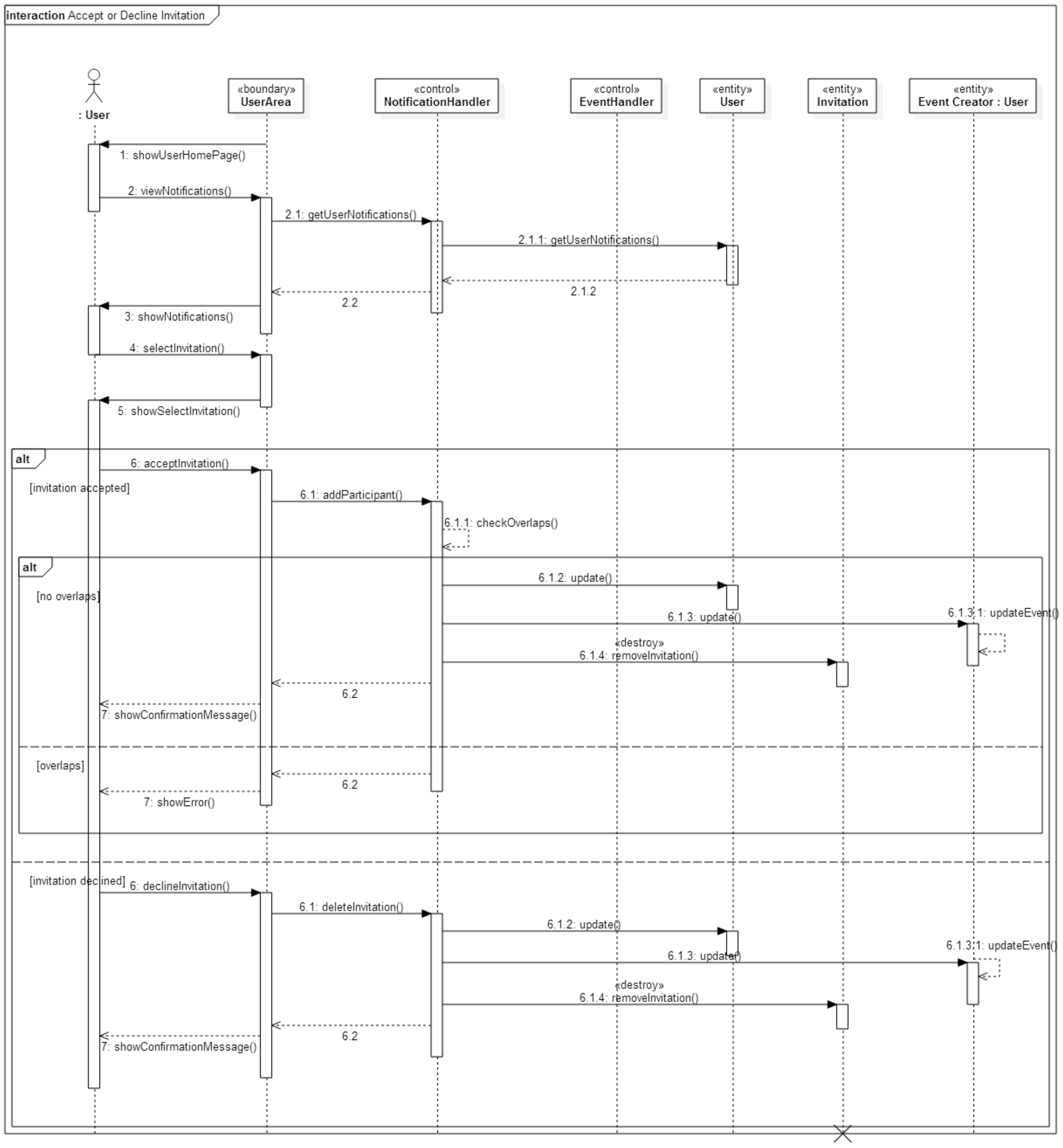
- **Create Event**



- **Invite User**


interaction Invite User
: User
«boundary» UserArea
«control» UserHandler
«control» NotificationHandler
«entity» User
1: showInviteUsers()
loop
2: submitSearchUser()
2.1: getUsersByUsername()
2.1.1: findByUsername()
2.1.2
2.2
3: showSearchResults()
4: selectUser()
6: inviteUser()
create()
«entity» Invitation
6.1
7: showInviteUsers()
finish()
showConfirmationMessage()

- **Accept/Decline Invitation**



## 6 Final Considerations

We decided not to draw any detailed diagram because we don't think that would add anything more than what our UX and BCE diagrams combined already said. Also, given our "ignorance" about Servlets and Java Beans, we are not sure about what the mapping between Server Pages and controls with them would be, which contributed to our decision of not doing any detailed diagram. Moreover, as we also don't know enough about the JEE architecture, we couldn't efficiently and knowingly write a correct Deployment View/ Run-Time View.