POLITECNICO DI MILANO
Computer Science and Engineering
Project of Software Engineering 2

MeteoCal

Design Document

Authors: Alessandra Decaneto 836131
Alberto Marchesi 835919

Reference Professor: Mirandola Raffaela

SUMMARY

1. INTRODUCTION 3

1.1 DESCRIPTION OF THE DOCUMENT 3
1.2 ARCHITECTURE DESCRIPTION 3
1.3 IDENTIFYING SUBSYSTEMS 4

2. PERSISTENT DATA MANAGEMENT 6

2.1 CONCEPTUAL DESIGN 6
2.2 LOGICAL DESIGN 9
2.2.1 ER Restructuration 9
2.2.2 Entities and Relations translation 10

3. USER EXPERIENCE 14

3.1 HOME 15
3.2 NOTIFICATION 17
3.3 EVENT LIST 19
3.4 CALENDAR 21
3.5 SEARCH 23

4. BCE DIAGRAMS 25

4.1 ENTITY OVERVIEW 25
4.2 SIGN UP AND LOG IN 27
4.3 SEARCH 29
4.4 CALENDAR MANAGING 31
4.5 NOTIFICATION MANAGING 33
4.6 INVITATION MANAGING 35
4.7 PROFILE MANAGEMENT 37
4.8 WEATHER MANAGING 39

5. SEQUENCE DIAGRAMS 41

5.1 LOG IN SEQUENCE DIAGRAM 42
5.2 SIGN UP SEQUENCE DIAGRAM 43
5.3 SEARCH SEQUENCE DIAGRAM 44
5.4 CREATE EVENT SEQUENCE DIAGRAM 45
5.5 THREE DAYS BEFORE SEQUENCE DIAGRAM 46
5.6 ACCEPT INVITATIONS SEQUENCE DIAGRAM 47

1.Introduction

1.1 Description of the document

In this document we focused on the Data design and on how the interaction between our system and the users should be.
The scope of this document is to provide the design of the application, describing and justifying the reason of our choices, for example the Data Base schema, the User Experience that our application should provide to our users and the main classes that we will make on the Implementation phase.
The Data Base design is essential to build a consistent Data Base that supports in an efficient way all the queries and the manipulations on the data.
It must be specified that the details of the implementation will be precisely defined on the implementation phase, so what we give is a general idea of our software.

1.2 Architecture Description

Before starting to explain our application's architecture we want to focus on JEE Architecture.
The infrastructure of our software is based on Java Enterprise Edition, with Glassfish as Application Server and MySQL as DBMS.
JEE has a four tiered architecture divided as:

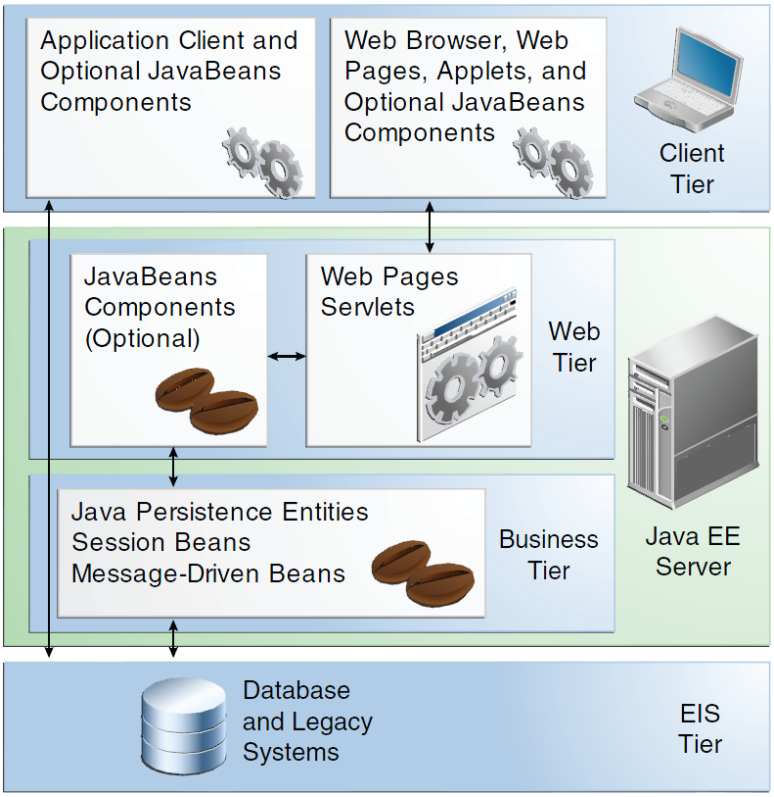
- Client
- Web
- Business logic
- EIS

The Client Tier manages the application User Interface via Web Browser using the support to managed bean given by J2EE7 and JSF pages, it is the layer that directly interacts with users.

The Web Tier manages the communication between the Client tier and the Business tier, it contains the Servlets and Dynamic Web Pages that needs to be elaborate.

The Business Tier manages all the business logic of the application, it contains the Enterprise Java Beans, which contain the business logic, and Java Persistence Entities.

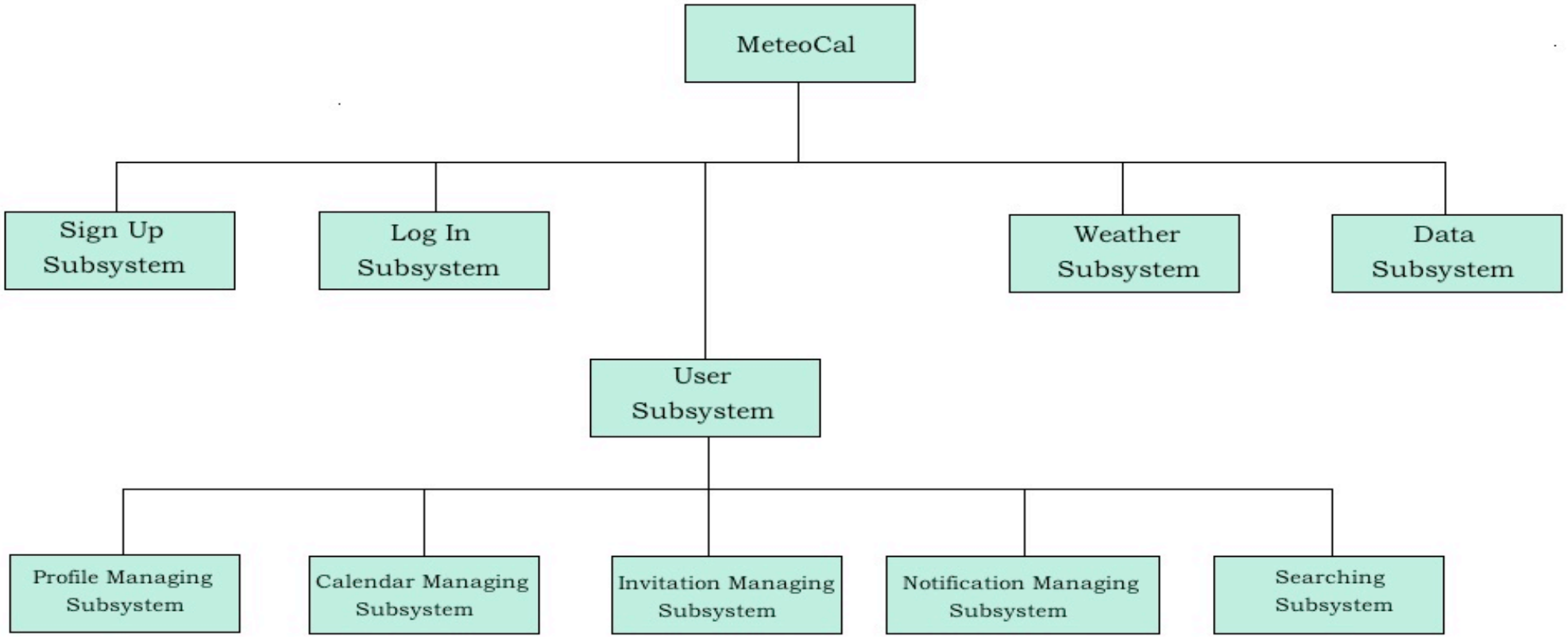
The EIS tier contains the data source and manages the operations of recovering data from the database and writing data on it.



1.3 Identifying Subsystems

We decided to decompose our systems in other sub-systems to figure the functionalities of our system out and to make them clearer.
We separate our systems into these sub-systems:

- Sign Up Sub-System
- Log In Sub-System
- User Sub-System
  - Profile Managing Sub-System
  - Calendar Managing Sub-System
  - Invitation Sub-System
  - Notification Sub-System
  - Searching Sub-System
- Weather Forecast Retrieval
- Data Sub-System


MeteoCal
Sign Up Subsystem
Log In Subsystem
User Subsystem
Weather Subsystem
Data Subsystem
Profile Managing Subsystem
Calendar Managing Subsystem
Invitation Managing Subsystem
Notification Managing Subsystem
Searching Subsystem

2. Persistent Data Management

We decided to store application's data in a relational database, because it has to manage persistent information that must be memorized durably. This paragraph explains our choices in the conceptual design of the database, using an ER diagram; then it describes the process we've gone through in order to design the relational model of the database.

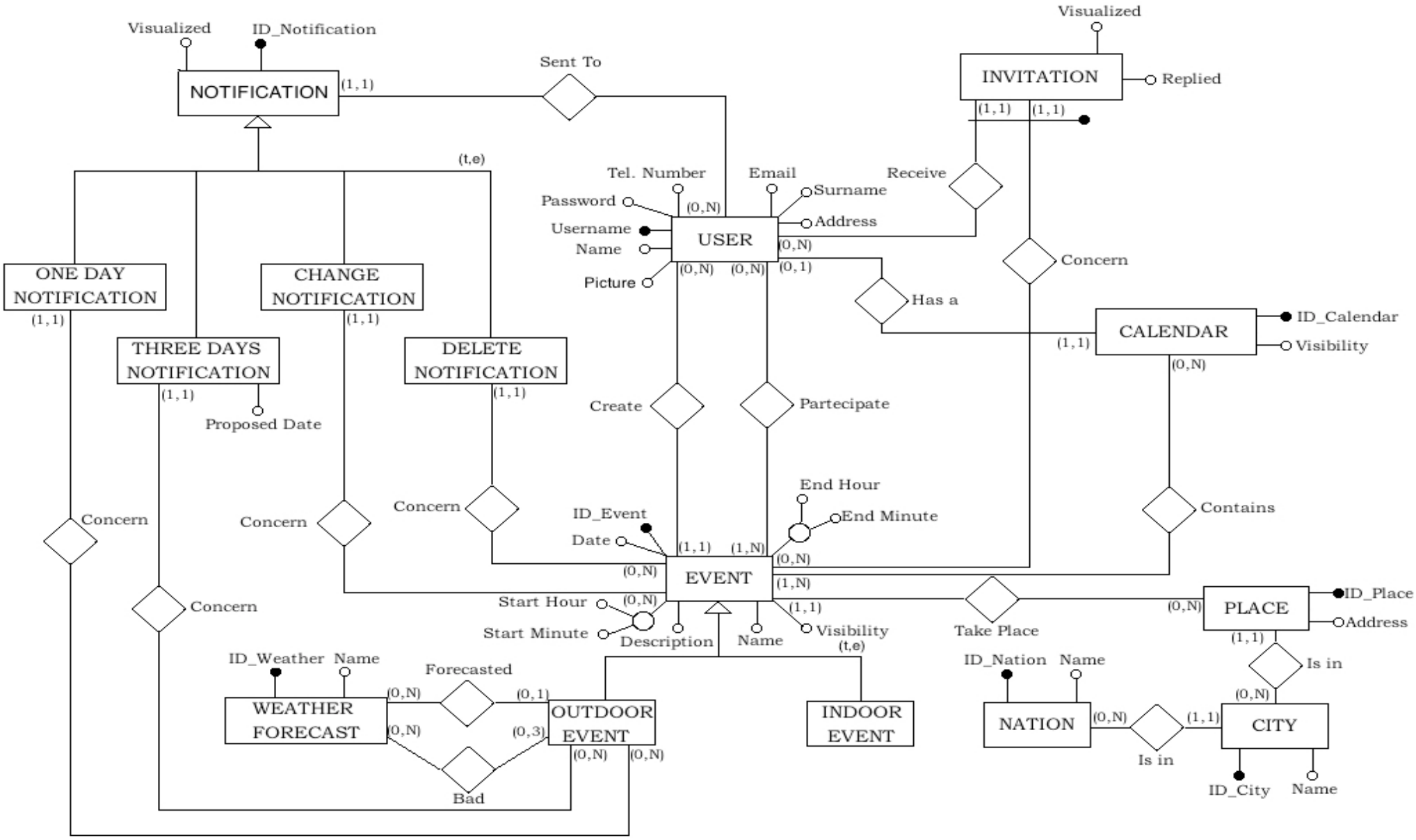
2.1 Conceptual Design

In this first phase we've designed the conceptual model of the database, a representation of the main entities and relations that are involved in the application data model.
This section explains in a clearer way how the ER diagram is constructed, of course we're not going to describe all the diagram in details but only the main aspects and those that are difficult to understand from the diagram itself. At the end we've reported the complete ER diagram.

The central entities in our application domain are the *User* and the *Event*. They are associated to each other by means of two main relations: *Participate*, which represents the participation of a user to some events, and *Create*, which represents the creator of an event.
Since there are two types of events, we've used a generalization hierarchy to distinguish between them: *Indoor Event* and *Outdoor Event*.
We've decided to represent a *Weather Forecast* using a dedicated entity; this choice will ease future expansions of the set of possible weather forecasts.
Of course, only the *Outdoor Event* is associated to a weather forecast through the relation *Forecasted*; whereas the relation *Bad* associates an *Outdoor Event* to at most three weather forecasts considered as bad for that event (since we've assumed that not all the possible weather conditions can be selected as bas for an event).
The *User* entity contains all the attributes that describes a user and it is identified via the *Username*. Although a calendar belongs to one user and a user has a unique calendar, we've decided to represent it as an entity, called *Calendar*, which contains the visibility attribute.
The main reason of that choice is that at this phase we want to concentrate on a conceptual level without taking into account any implementations concern.
The relation *Contains* associates a calendar to the set of events that forms it.
The *Invitation* entity represents an invitation, it is associated to an *Event* and to a *User* and it is identified through these relations in order to make clear that an invitation exists if and only if both the user and the event exist.
The entity has two Boolean attributes: *Visualized*, which indicates whether the invitation is received or not by the user, and *Replied* that indicates whether the user will participate or not.

Since there are four types of *Notification*, we've decided to introduce a generalization hierarchy, totally and exclusive. Children entities are: *One Day Notification*, *Three Days Notification*, *Delete Notification* and *Change Notification*.
The first two are associated to an *Outdoor Event*, which they concern with.
The others are associated to a generic *Event*, because they can concern both indoor and outdoor events.
The *Notification* entity is associated, through the relation *Sent To*, to a *User*.
Furthermore, *Three Days Notification* has an attribute, called *Proposed Date*, which contains the possible alternative date proposed by the system.
We've decided to store in our system all the information about the places in which the events take place in order to memorize them in the database only one time.
So we have introduced the entities *Place*, *City* and *Nation*. *City* represents all possible cities for which a weather forecast can be retrieve.

The foolowing is the complete ER Diagram:



2.2 Logical Design

The aim of this phase is to develop a relational schema of the database starting from the conceptual design. In this section we'll discuss the main project decisions we've taken during this step and the final result, which is the schema of the database including the specifications of all the tables (with their fields and fields' types) and the most significant constraints in order to make data consistent with respect to domain properties.

2.2.1 ER Restructuration

Hierarchies elimination

In our ER schema there are two hierarchies, since the relational model doesn't admit generalization structures we have to eliminate them.

- *Event* hierarchy is collapsed upward. As a consequence all the relations attached to the children entities are transferred to the *Event* entity and the entities *Outdoor Event* and *Indoor Event* are dropped from the schema. Moreover, in order to distinguish between Indoor and Outdoor events, we've added an attribute *type* to the entity *Event* that can assume only the values Outdoor or Indoor. There are two main reasons of that choice:

    - There are many relations attached to the *Event* entity, so a downward collapse would introduce a lot of duplicate relations.

    - There are some one-to-many relations going to the *Event* entity, in particular the *Concern* relations coming from *Delete Notification* and *Change Notification*. A downward collapse would be very difficult to manage because each of these relations should be split into two different relations, one pointing to *Indoor Event* the other pointing to *Outdoor Event*, which are mutually exclusive, in order to guarantee that each notification is associated exactly to one event.

- *Notification* hierarchy is collapsed downward. As a consequence the relation *Sent to* attached to the *Notification* entity is duplicated on each of the children entities and the *Notification* entity is dropped from the schema. The main reason of that choice is that there are many relations on children entities and some of those entities have attributes that differentiate them. Since there is only one relation attached to the *Notification* entity, this is the most efficient choice.

External identifications elimination

In our ER schema, there is the *Invitation* entity that is identified by two relations, which associate it to *User* and *Event*. In order to get a relational schema we have to eliminate these relations by substituting them with attributes attached to the *Invitation* entity that refer to the identifiers of *User* (*username*) and *Event* (*ID_Event*). We've decided to use the pair composed by these two attributes as identifier for the *Invitation* entity.

Attributes normalization

The *Event* entity has two composite attributes, we've decided to split them into single attribute, so the restructured *Event* entity has the following additional simple attributes: *Start Hour*, *Start Minute*, *End Hour*, *End Minute*. Moreover, all optional attributes are traduced in the relational schema as fields that allow a null value (we'll mark them with a * in the tables description).

2.2.2 Entities and Relations translation

Before translating all the entities and the relations, we've decided to collapse the entity *Calendar* in the entity *User*, because they're associated by means of one-to-one relation and we do not really need a specific table for representing users' calendars, we can implicitly represent them by means of the association between users and the events which they participate to. The consequence of that choice is that the *Contains* relation is moved on *User* and the *visibility* attribute too.
After that restructuration, we've noticed that the relations *Contains* and *Participate* represent the same concept, since we've assumed that user's calendar contains all and only the events, which he will participate to, so one of them is redundant. We've dropped the relation *Contains*.
Then we've translated all the entities of the reconstructed ER schema into tables.
In our ER schema the entity user is identified by means of the Username attribute, but in the logical schema we've decided to use an incremental integer identifier. Of course we force the uniqueness of the Username by declaring it as unique field.
The identifiers of the entities become the primary keys of the tables and the attributes of the entities become the fields of the tables.

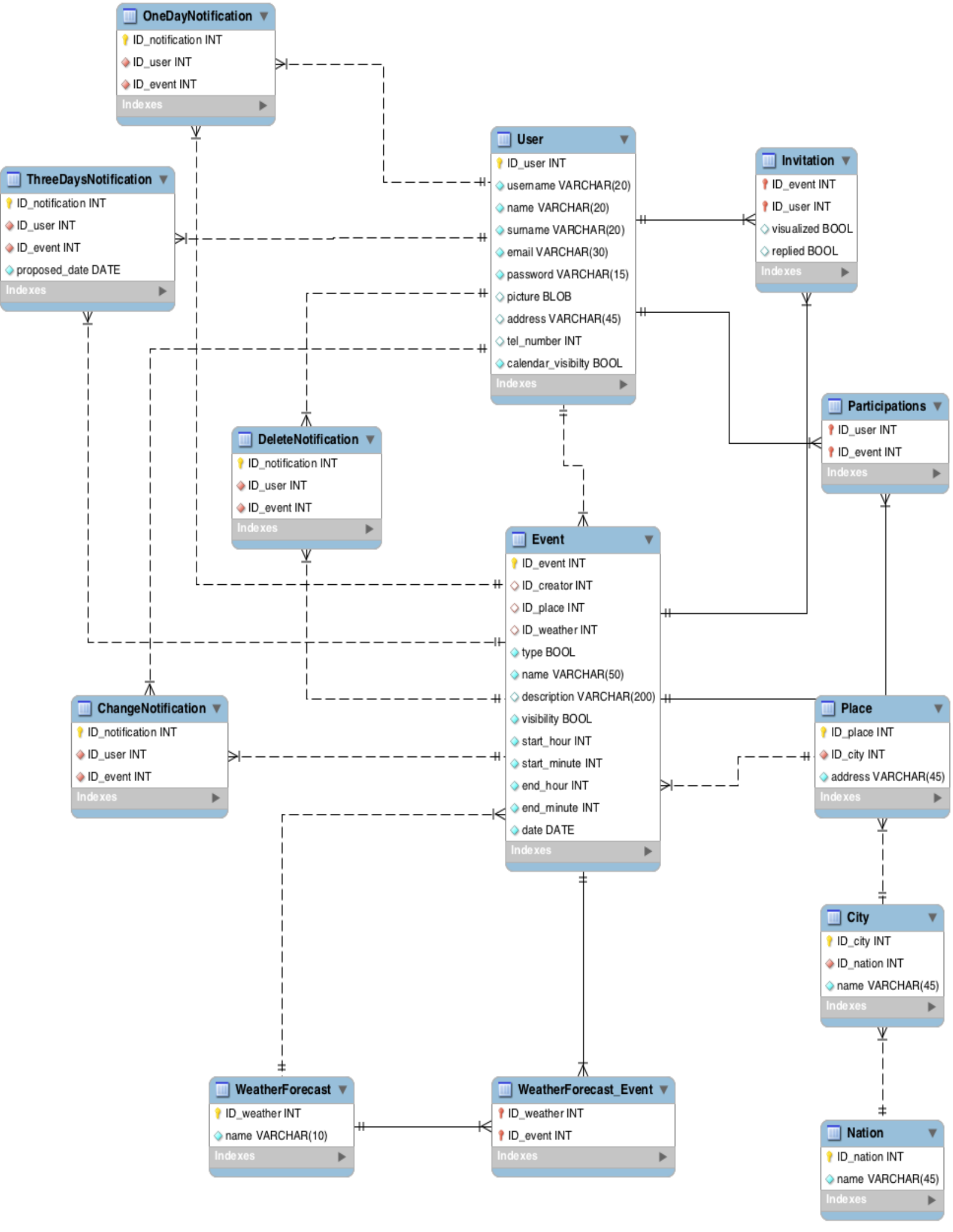
The next step is the translation of the relations; we've translated them as follows:

- The *Create* and *Take place* relations on *Event* are translated by adding two foreign keys to the table *Event*, named *ID_Creator* and *ID_Place*, referring respectively to the primary key of the table *User* and *Place.*
- The *Forecasted* relation on *Event* is traduced by adding a foreign key to the table *Event*, named *ID_Weather*, which allows null value and refers to the primary key of the table *Weather Forecast.*
- The *Receive* and *Concern* relations on *Invitation* are translated by declaring as foreign keys the attributes *ID_Event* and *Username*, referring respectively to the primary key of the table *Event* and *User*.
- All the relations named *Concern* that associate a notification entity (*One Day Notification*, *Three Days Notification*, *Delete Notification*, *Change Notification*) to *Event* are translated by adding two foreign keys to the notification table, named *ID_User* and *ID_Event*, referring respectively to the primary key of the table *User* and *Event*.
- The relation involving the entities *Place*, *City* and *Nation* are translated adding appropriate foreign keys in the corresponding tables as we've done so far for the other one-to-many relations.
- The *Bad* many-to-may relation between *Event* and *Weather Forecast* is translated by adding a new table to the schema, named *Weatherforecast_Event*, which contains two foreign key, named *ID_Weather* and *ID_Event*, referring respectively to the primary key of the table *Weather Forecast* and *Event*.
- The *Participate* many-to-many relation between *User* and *Event* is translated by adding a new table to the schema, named *Participations*, which contains two foreign keys, named *ID_User* and *ID_Event*, referring respectively to the primary key of the table *User* and *Event*.

Furthermore, there are some constraints on the relational model that we have to directly state in order to maintain the consistency of data with respect to the domain properties.

- The field *ID_Event* in the tables *OneDayNotification* and *ThreeDayNotification* must contain an identifier associated to an *Event* with the field *type* set to *Outdoor*.
- The field *ID_Event* in the table *WeatherForecast_Event* must contain an identifier associated to an *Event* with the field *type* set to *Outdoor*.
- The field *ID_Weather* in the table *Event* must be null if the field *type* is set to *Indoor*.

We've obtained the following relational model:


OneDayNotification
ID_notification INT
ID_user INT
ID_event INT
Indexes
User
ID_user INT
username VARCHAR(20)
name VARCHAR(20)
surname VARCHAR(20)
email VARCHAR(30)
password VARCHAR(15)
picture BLOB
address VARCHAR(45)
tel_number INT
calendar_visibilty BOOL
Indexes
Invitation
ID_event INT
ID_user INT
visualized BOOL
replied BOOL
Indexes
ThreeDaysNotification
ID_notification INT
ID_user INT
ID_event INT
proposed_date DATE
Indexes
Participations
ID_user INT
ID_event INT
Indexes
DeleteNotification
ID_notification INT
ID_user INT
ID_event INT
Indexes
Event
ID_event INT
ID_creator INT
ID_place INT
ID_weather INT
type BOOL
name VARCHAR(50)
description VARCHAR(200)
visibility BOOL
start_hour INT
start_minute INT
end_hour INT
end_minute INT
date DATE
Indexes
ChangeNotification
ID_notification INT
ID_user INT
ID_event INT
Indexes
Place
ID_place INT
ID_city INT
address VARCHAR(45)
Indexes
City
ID_city INT
ID_nation INT
name VARCHAR(45)
Indexes
WeatherForecast
ID_weather INT
name VARCHAR(10)
Indexes
WeatherForecast_Event
ID_weather INT
ID_event INT
Indexes
Nation
ID_nation INT
name VARCHAR(45)
Indexes

The final model has the following physical structure:

Event (ID_event; ID_creator; ID_place; ID_weather; type; name; description; visibility; start_hour; start_minute; end_hour; end_minute; date)

User (ID_user; username; name; surname; email; password; picture; address; tel_number; calendar_visibility)

Invitation (ID_event; ID_user; visualized; replied)

OneDayNotification (ID_notification; ID_user; ID_event)

ThreeDaysNotification (ID_notification; ID_user; ID_event; proposed_date)

DeleteNotification (ID_notification; ID_user; ID_event)

ChengeNotifiaction (ID_notification; ID_user; ID_event)

WeatherForecast (ID_weather; name)

WeatherForecast_Event (ID_weather; ID_event)

Place (ID_place; ID_city; address;)

City (ID_city; ID_nation; name)

Nation (ID_nation; name)

Participations (ID_user; ID_event)

3.User Experience

In this paragraph we describe the user experience given by our system to the users by means of a UX (User Experience) diagram.
We've represented that diagram by extending the Class Diagram notation by means of appropriate stereotypes:

- <<screen>>: represents a web page of our application.
- <<screen compartment>>: represents a graphical component inside a page.
- <<box>>: represents a message box that is popped out on the screen.
- <<input form>>: represents a set of input fields or other input components inside a page.

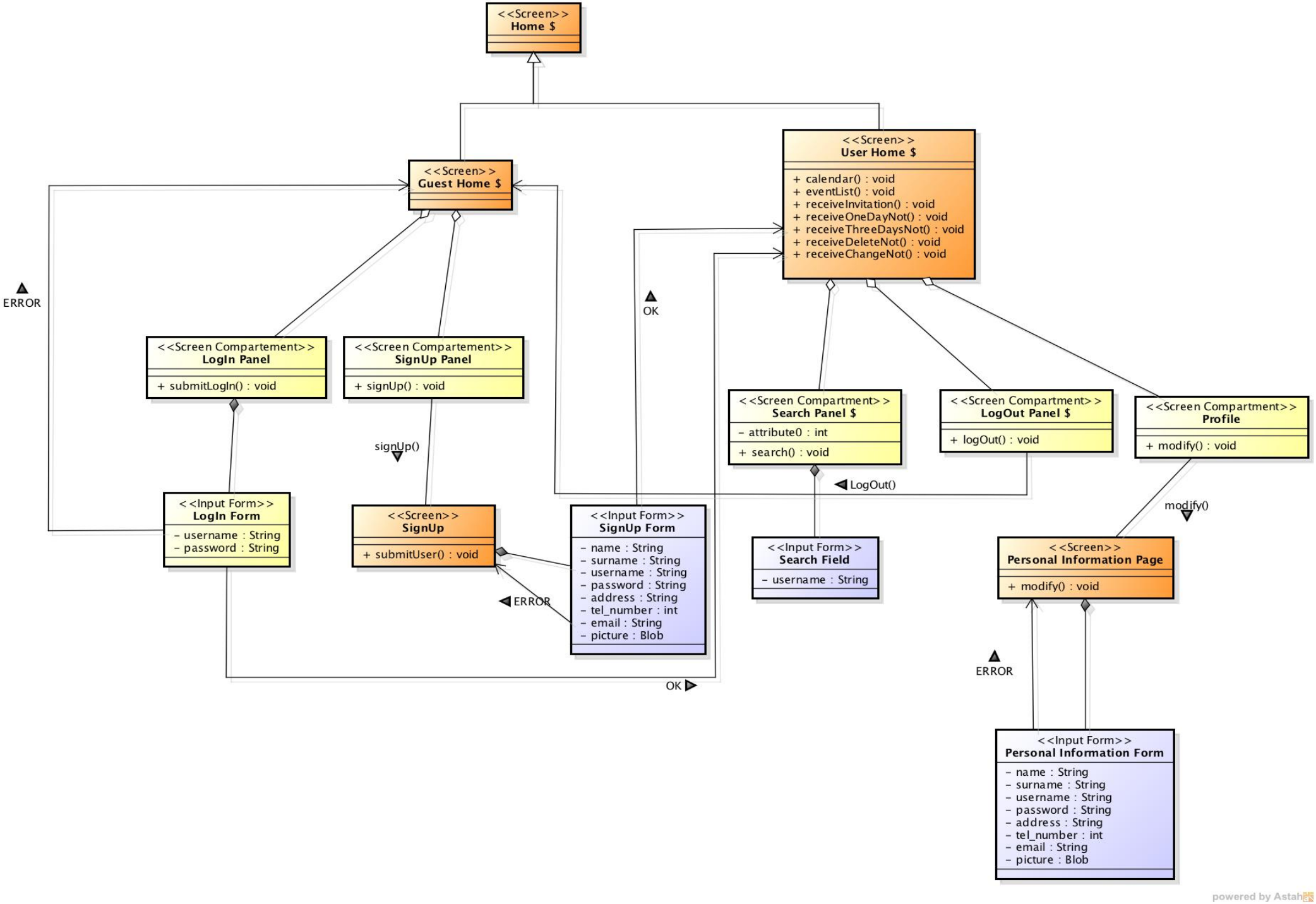
The arrows represent the flow followed by the user in the application by performing a specific operation or by triggering a particular behavior.
In some case we've labeled the <<screen>> classes with two symbols: $, which indicates that the page is reachable from any other page in the application, and +, which denotes a scrollable component that has particular methods (like previous () and next ()) which allow to modify the content visualized inside it.
Furthermore, we've used composition and aggregation relations in order to represent the relationship among the components of a page.

We've decided to split our diagram into many parts in order to make it more readable and to focus on each function of our application.
We've identified the following parts:

- Home
- Notification
- Event List
- Calendar
- Search

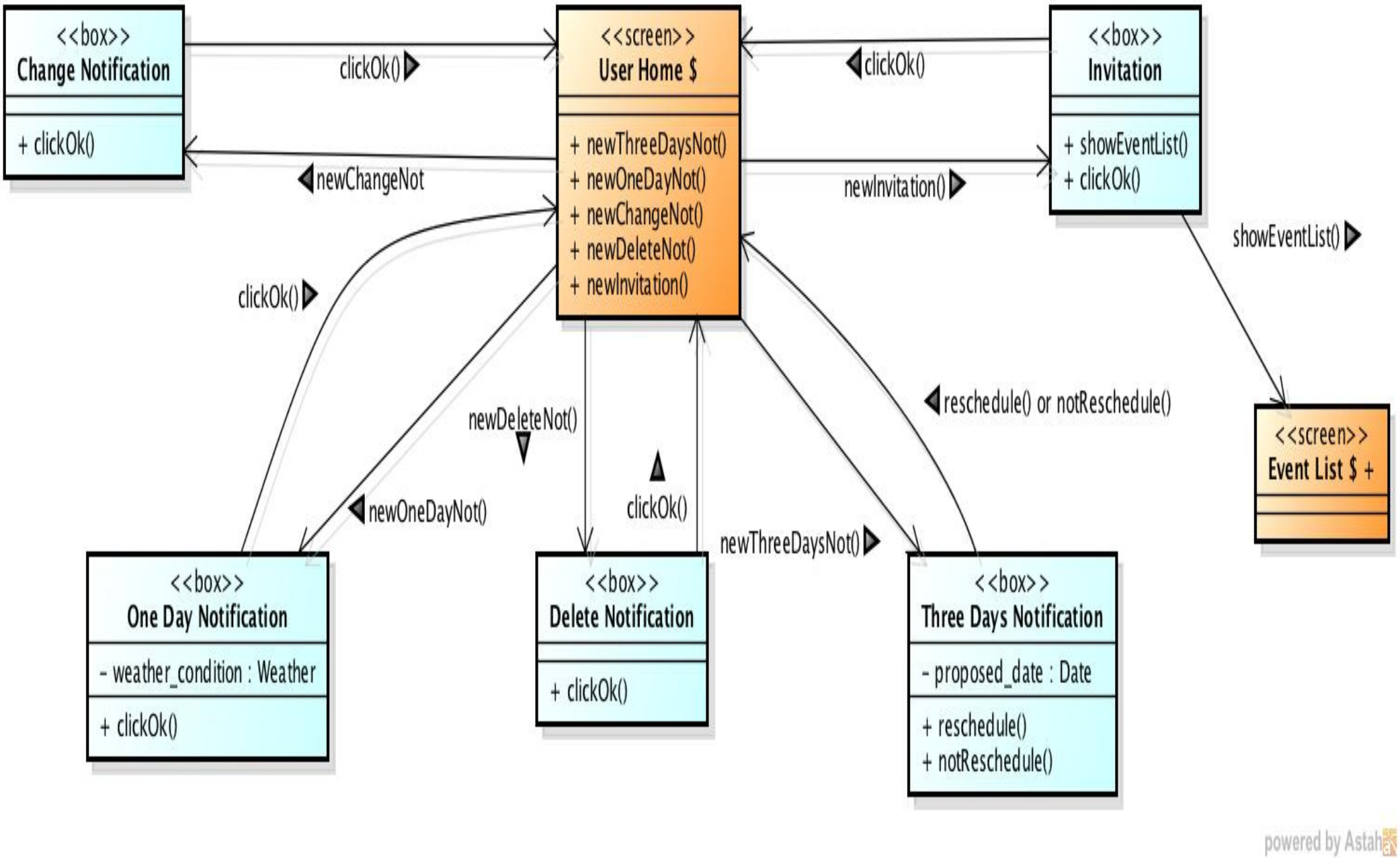
3.1 Home

This part of the UX Diagram describes the *Home* , which can be shown in two different ways if the user is authenticated or if the user is just a guest.
If the user is not already logged in he sees the *Guest Home* where he can either sign up or log in, if the user doesn't have an account he can click on the sign up button and he has to fill in the *Sign Up form*, then if every field of the form is correct and the username isn't already used form someone else the system shows him the *User Home.* If the user has an account already he has to click on the log in button, to fill in the *Log In Form* and if his data are correct the system shows him the *User Home*.
From the *User Home* a user can have access to the *Event List*, to his *Calendar*, he can fill the *Search Form* and search for another user, he can log out and see his own *Profile*.
When he clicks on his *Profile* another page is shown to the user where he can click on modify to modify his personal data or just see his *Personal Information F*


<<Screen>>
Home $
<<Screen>>
Guest Home $
<<Screen>>
User Home $
+ calendar() : void
+ eventList() : void
+ receiveInvitation() : void
+ receiveOneDayNot() : void
+ receiveThreeDaysNot() : void
+ receiveDeleteNot() : void
+ receiveChangeNot() : void
ERROR
OK
<<Screen Compartement>>
LogIn Panel
+ submitLogIn() : void
<<Screen Compartement>>
SignUp Panel
+ signUp() : void
signUp()
<<Screen Compartment>>
Search Panel $
- attribute0 : int
+ search() : void
<<Screen Compartment>>
LogOut Panel $
+ logOut() : void
LogOut()
<<Screen Compartment>>
Profile
+ modify() : void
modify()
<<Input Form>>
LogIn Form
- username : String
- password : String
<<Screen>>
SignUp
+ submitUser() : void
<<Input Form>>
SignUp Form
- name : String
- surname : String
- username : String
- password : String
- address : String
- tel_number : int
- email : String
- picture : Blob
ERROR
<<Input Form>>
Search Field
- username : String
<<Screen>>
Personal Information Page
+ modify() : void
ERROR
OK
<<Input Form>>
Personal Information Form
- name : String
- surname : String
- username : String
- password : String
- address : String
- tel_number : int
- email : String
- picture : Blob
powered by Astah

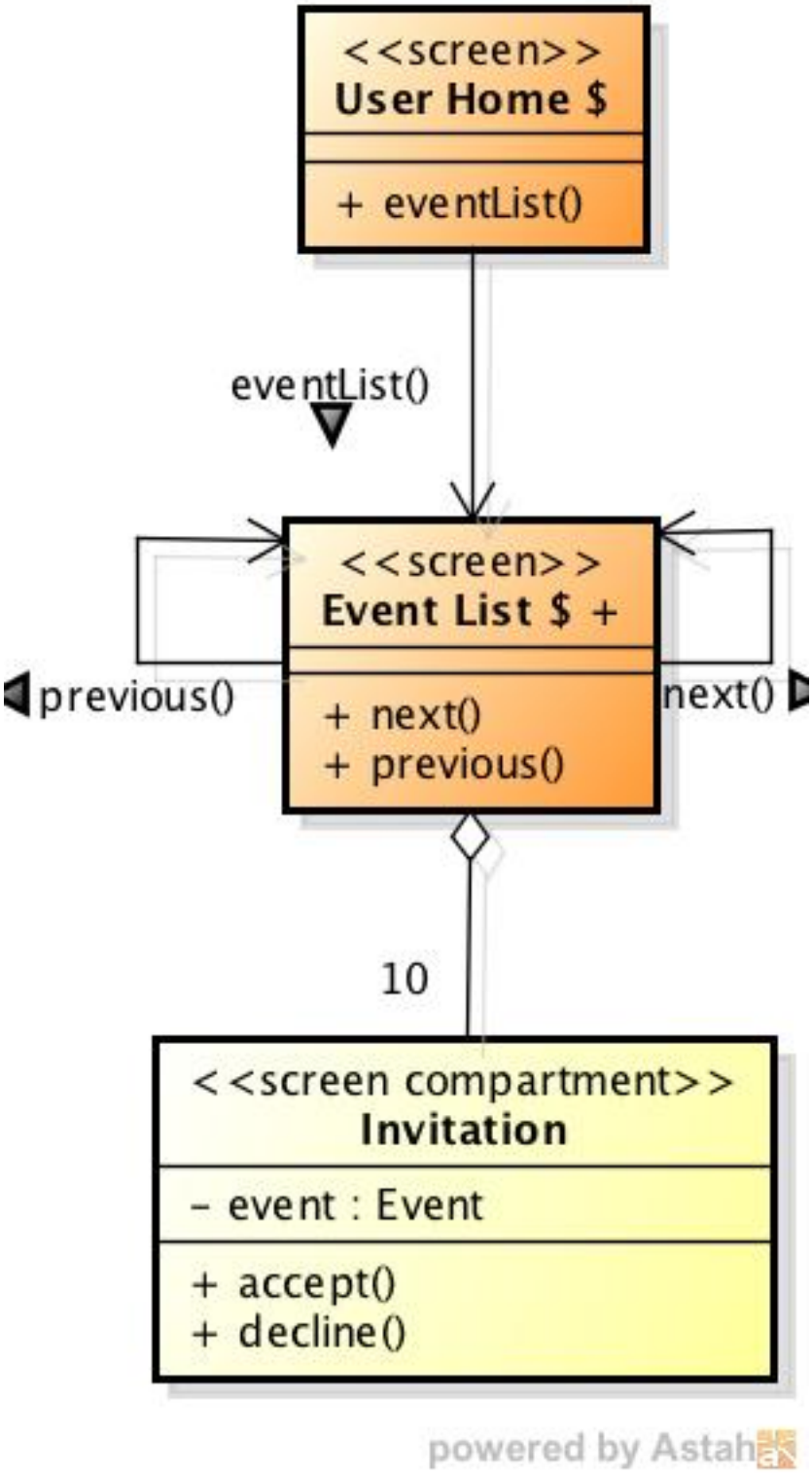
3.2 Notification

This part of the UX diagram describes how the notifications are visualized to the user.
When a user is in his personal page or whenever he logs in, the application will notify the user about new notifications. All of them are visualized by means of a box that appears on the screen and contains all the information about the notification.
One Day Before Notification boxes contain the name of the event, which they concern with, and the weather forecast for that event; the user can close them by clicking the button Ok.
Three Days Before Notification boxes contain the name of the event, the weather forecast for that event and a proposed date for the event to be rescheduled; the user can accept the proposed date by clicking on *Reschedule* or ignore it by clicking on *Not Reschedule*, in both cases the notification is closed.
Change Notification and *Delete Notification* boxes contain information about the operation and the event it is applied to; the user can close them by clicking on the button Ok.



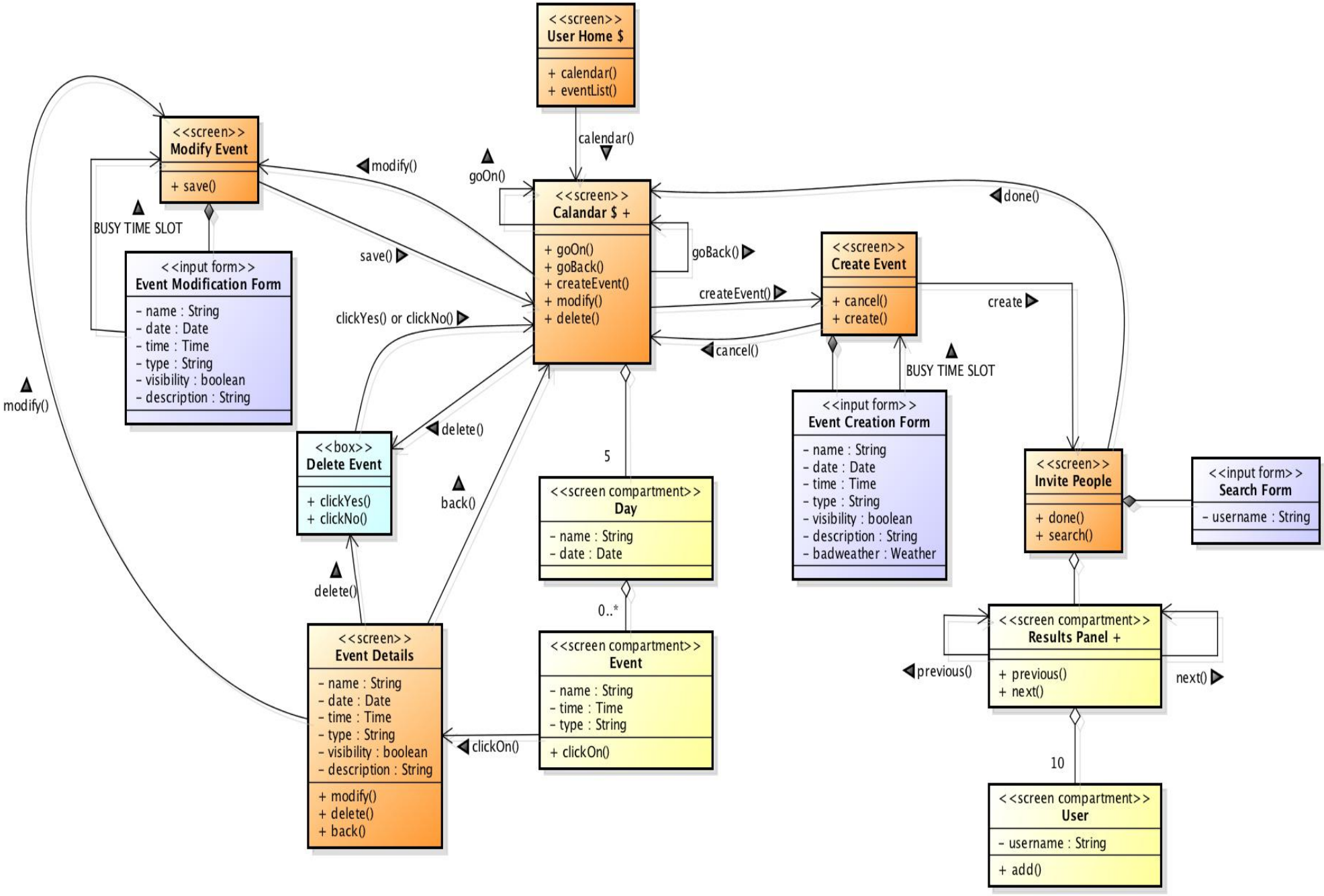
3.3 Event List

This part of the UX diagram describes the *Event List* page, which allows the user to mange his invitations accepting or declining them.
The *Event List* page is reachable from any other page by clicking on the dedicated link in the menu in the top of the page. It contains the list of the invitations received by the user; they are represented by means of the event name and the key information about the event.
The user can accept or decline an invitation clicking on the corresponding buttons next to it, then the buttons will be disabled by the system and the event will remain in the list only if the user has accepted the invitation.


<<screen>>
User Home $
+ eventList()
eventList()
<<screen>>
Event List $ +
previous()
+ next()
+ previous()
next()
10
<<screen compartment>>
Invitation
- event : Event
+ accept()
+ decline()
powered by Astah

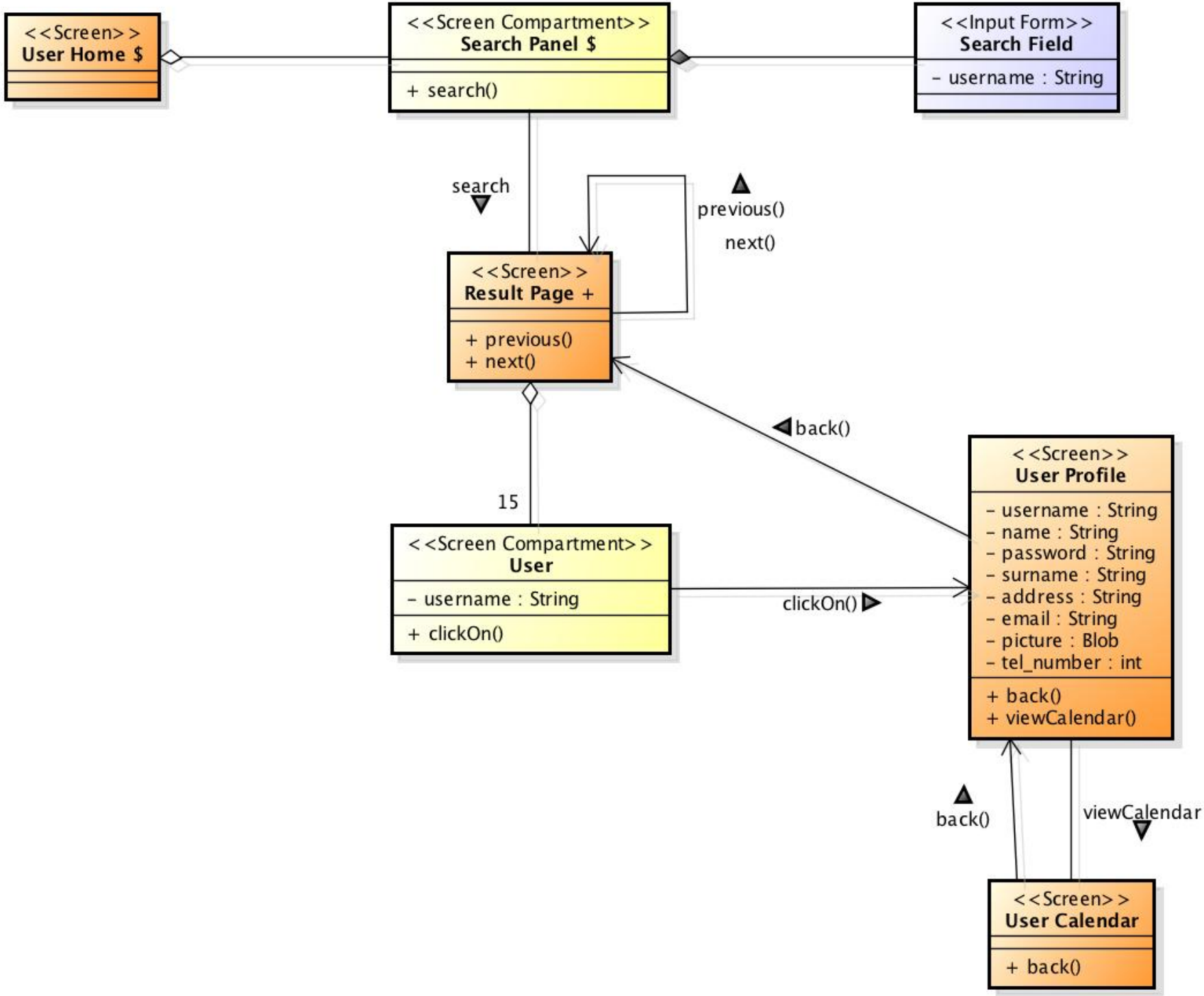
3.4 Calendar

This part of the UX diagram describes the management of the calendar.
The *Calendar* page is reachable from any other page by clicking on the dedicated link in the menu in the top of the page. In this page there is a representation of the events that are scheduled in the next five days; they are ordered by day (on column) and by hours (on rows).
By clicking on an event a new page is open, called *Event Details*, which contains all the information about the event. From it the event can be modified or deleted by clicking the corresponding button. Another possibility for modifying or deleting an event is that of selecting the event from the calendar and then clicking on modify or delete button.
From the *Calendar* page is possible to create a new event by clicking on the button *Create*, then a new page is open, called *Create Event*, which contains an input form that should be filled in with the information about the event. Of course if some information are wrong or the time slot selected is busy, the form will signal an error.
After the event is created by clicking on create, the *Invite People* page is open; it allows to search other users and invite them to the event just by clicking on their names in the list of the results that is visualized on the screen.
Whenever a user tries to delete an event, the application shows him a box that asks the confirmation of the operation.
The modification of an event is done in the page *Modify Event*, which allows changing all the information about the event by means of an input form similar to the one visualized in the creation phase.



3.5 Search

This part of the UX diagram describes the search for another User.
In the *User Home* there is the *Search Panel*, where he can click on search and he has to fill in the *Search Field* form where he has to write the username of the person he's looking for.
So when he clicks on search the *Result Page* is shown to him and by clicking on next and previous he can see more results or go back to the previous results.
The *Result Page* is made of many *User* screen compartments, which are clickable from the user and if he clicks on them he can access the User Profile of the one he as selected.
From the User Profile he can also access the User Calendar of the user he has selected.



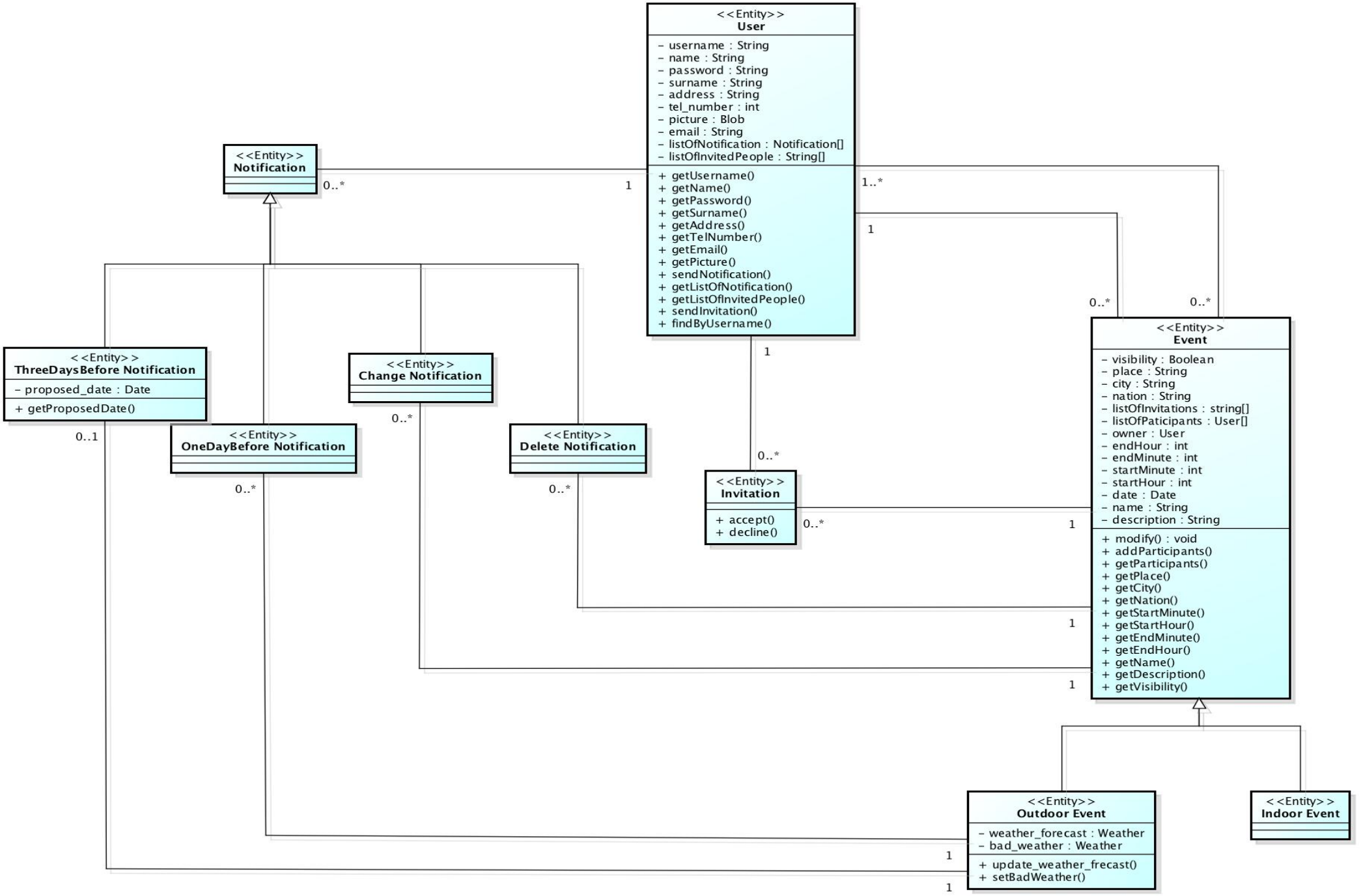
4. BCE Diagrams

We decided to go into the details of the structure of our application using a class diagram that represents the its components and the relationship among them. In particular we used the Boundary-Entity-Control (BCE) pattern, which is a variation of the MVC.
Boundaries define the view, so the interaction between the users and the application; they can be associated to the screens defined in UX diagrams, but they may gather more than one screen.
Controllers manage all the business logic of the system and act as a connection between the boundaries and the entities.
Finally the entities represent the data model of the application; we point out that the entities are slightly different from the ones in the Data Base, because we've decided to give a representation suitable for a class diagram. For example we've added the entity Calendar that is not present in the Data Base for efficiency reason.
The BCE diagram is divided into the following sub-systems in order to make it more readable:

- Log In and Sign Up
- Search
- Calendar Managing
- Notification Managing
- Invitation Managing
- Profile Management

4.1 Entity Overview

We now present a general schema of all the entities that you will find in the BCE Diagrams, in order to make them more readable and understandable for our readers.

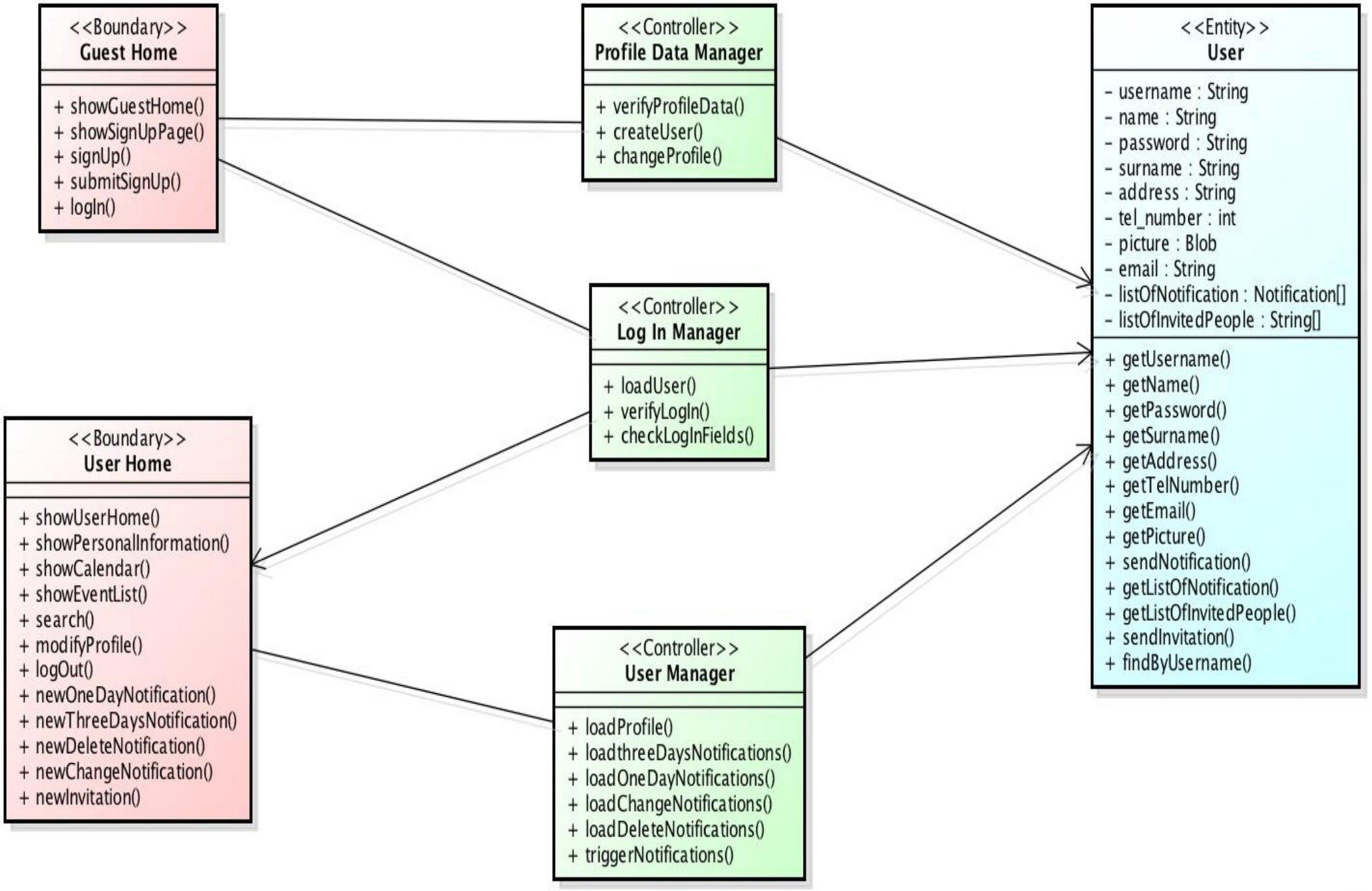


4.2 Sign Up and Log In

The two boundaries *Guest Home* and *User Home* represent the interfaces between the user and the system; the first one when the user is a guest, the second one when the user is authenticated. They expose to the user the main functionalities that can be activated when the user is the home page of the application and those that can be accessed from any page of the application (such as Search, Calendar, ..).
In this subsystem there are three main controllers:

- *Profile Data Manager*
  It controls the correctness of the information inserted by the user during the sign up page, it creates new user entities and it manages the modification of profile's information.
- *Log In Manager*
  It controls that the username and password inserted by the user in the log in fields are conform with their correct structure (by means of the method *checkLogInFields*) in order to avoid the access to the DB in the case that information inserted can't be valid. It verifies that the username and password inserted by the user correspond to an existing user, using the method *verifyLogIn*. It loads the information about the user by means of the method *loadUser*.
- *User Manager*
  It controls the interaction with the user when he is in the home page; essentially it manages the load of profile's information and the visualization of the notifications both when he logs in and when a new notification is received while he is on-line.

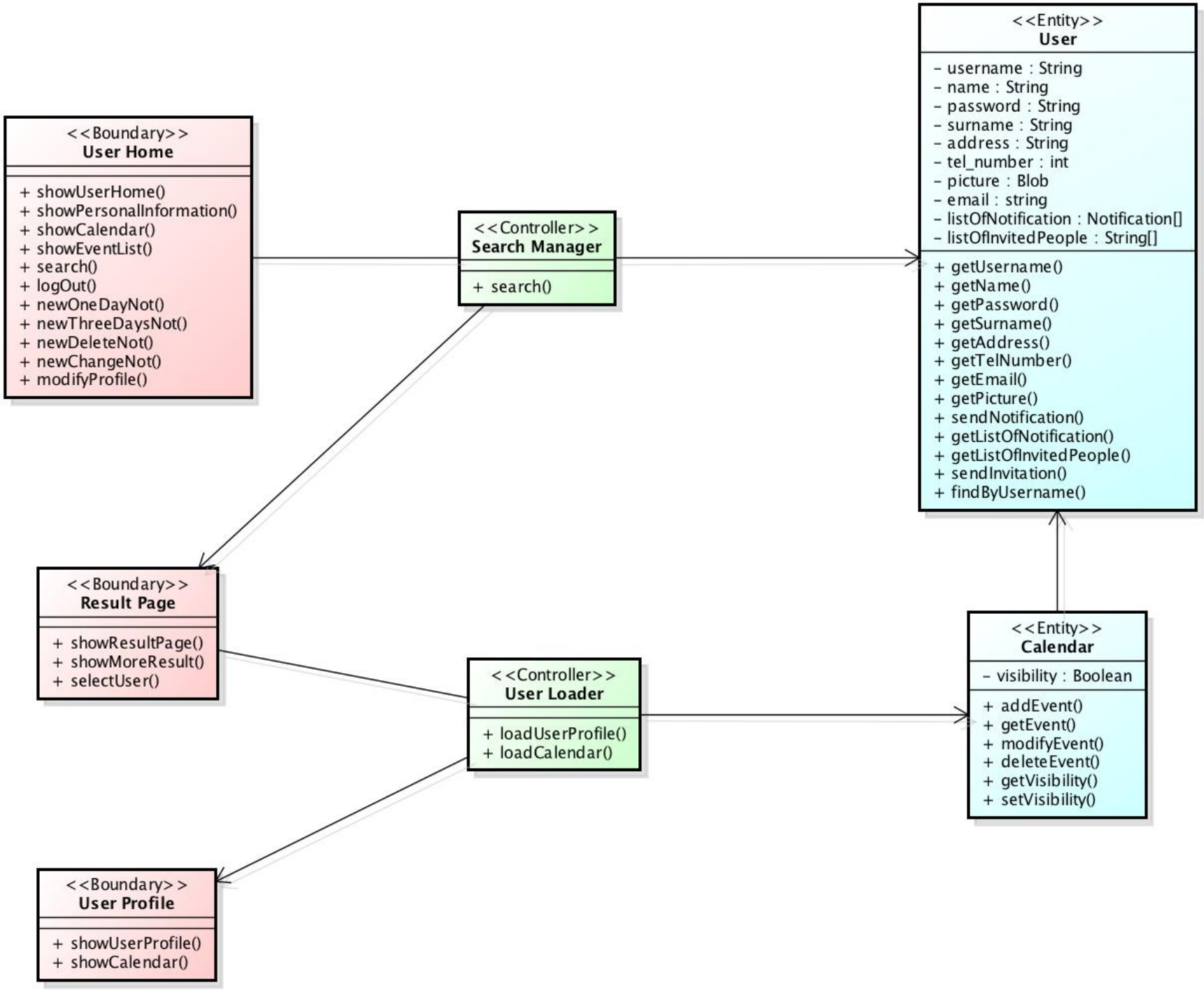
This controller is central in our application so it's related to other components, but since here we're concentrating in a specific functionality we've left out other dependencies that will appear in other sections.


<<Boundary>>
Guest Home
+ showGuestHome()
+ showSignUpPage()
+ signUp()
+ submitSignUp()
+ logIn()
<<Controller>>
Profile Data Manager
+ verifyProfileData()
+ createUser()
+ changeProfile()
<<Entity>>
User
- username : String
- name : String
- password : String
- surname : String
- address : String
- tel_number : int
- picture : Blob
- email : String
- listOfNotification : Notification[]
- listOfInvitedPeople : String[]
+ getUsername()
+ getName()
+ getPassword()
+ getSurname()
+ getAddress()
+ getTelNumber()
+ getEmail()
+ getPicture()
+ sendNotification()
+ getListOfNotification()
+ getListOfInvitedPeople()
+ sendInvitation()
+ findByUsername()
<<Controller>>
Log In Manager
+ loadUser()
+ verifyLogIn()
+ checkLogInFields()
<<Boundary>>
User Home
+ showUserHome()
+ showPersonalInformation()
+ showCalendar()
+ showEventList()
+ search()
+ modifyProfile()
+ logOut()
+ newOneDayNotification()
+ newThreeDaysNotification()
+ newDeleteNotification()
+ newChangeNotification()
+ newInvitation()
<<Controller>>
User Manager
+ loadProfile()
+ loadthreeDaysNotifications()
+ loadOneDayNotifications()
+ loadChangeNotifications()
+ loadDeleteNotifications()
+ triggerNotifications()
powered by Astah

4.3 Search

The three boundaries *User Home*, *Result Page* and *User Profile* represent the interfaces between the system and the user.
The first exposes to the user the main functionalities that can be activated when the user is the home page of the application and those that can be accessed from any page of the application (such as Search, Calendar, ..); the second shows the list of the result of the search for an user, and from this page you can see more results or the previous one; the third one shows the User Profile of the user that has been selected, and form here you can have the access to his profile and to his calendar.
In this sub-system there are two controllers:

- *Search Manager*
  It accesses the Data Base and elaborates the results that will be shown in the boundary, so it searches for the User you are looking for.
- *User Loader*
  Once the user has selected the result he is interested in, the User Loader can load the Profile of the user searched by the method *loadUserProfile* and can load his Calendar by the method *loadCalendar*.

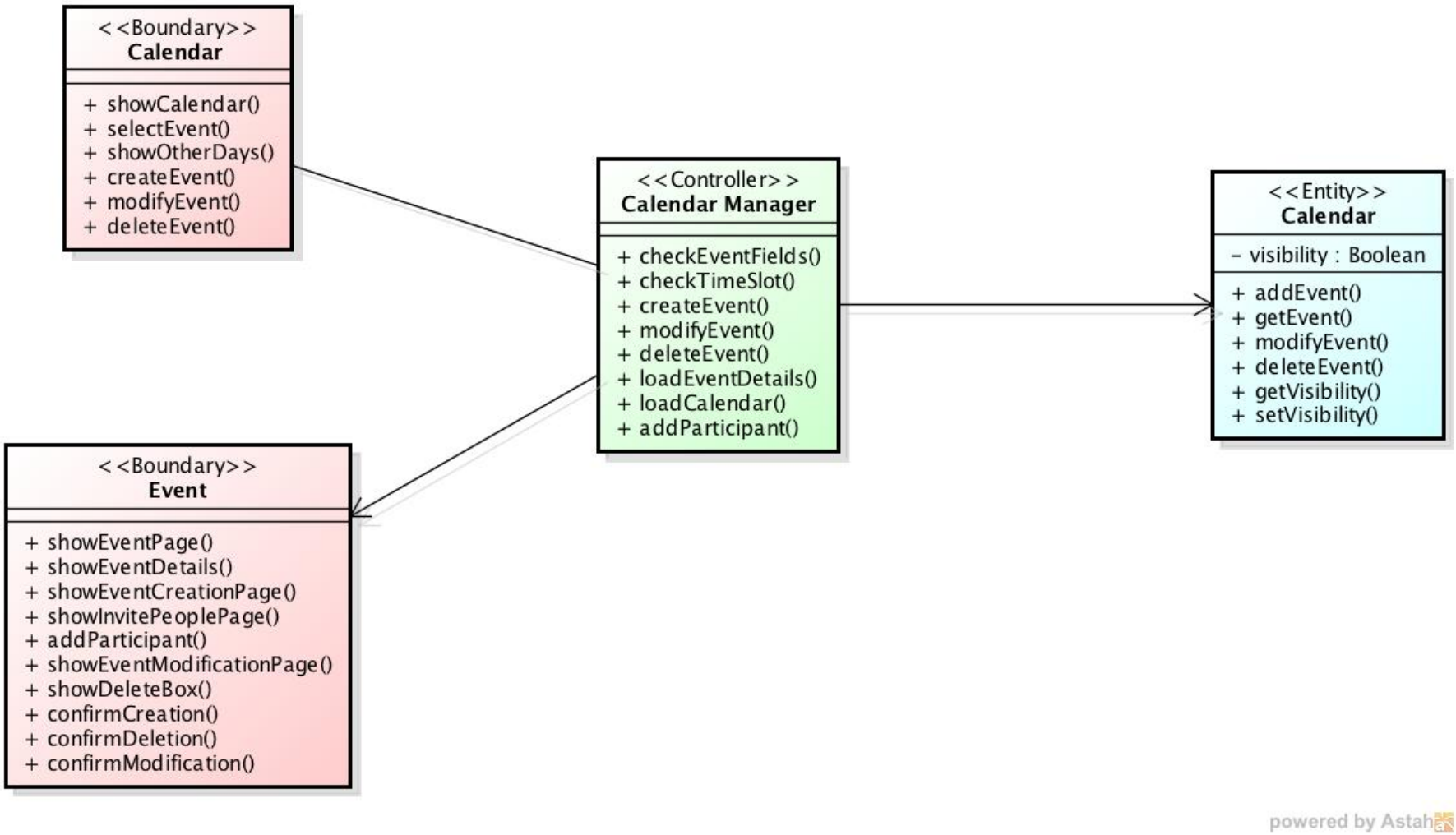


4.4 Calendar Managing

The boundaries of this subsystem are *Calendar* and *Event*. The first one manages the interaction with the user during the management of his calendar, essentially when he visualizes it. The second one manages the interaction with the user when he creates, modifies or deletes an event and when he visualizes the details of an event. There is only one controller called *Calendar Manager*, which controls the main tasks related to:

- The creation of an event, controlling the correctness of the information with respect to format constraints and checking the availability of the selected time.
- The modification and deletion of an event.
- The load of the event details and the calendar structure.

We also point out that the *Calendar* controller interacts only with the *Calendar* entity, this is due to the fact that the management of the events is done indirectly through the *Calendar* entity.
This sub-system also deals with the visualization of the calendar of another user, of course avoiding the functionalities related to the modifications of the calendar.

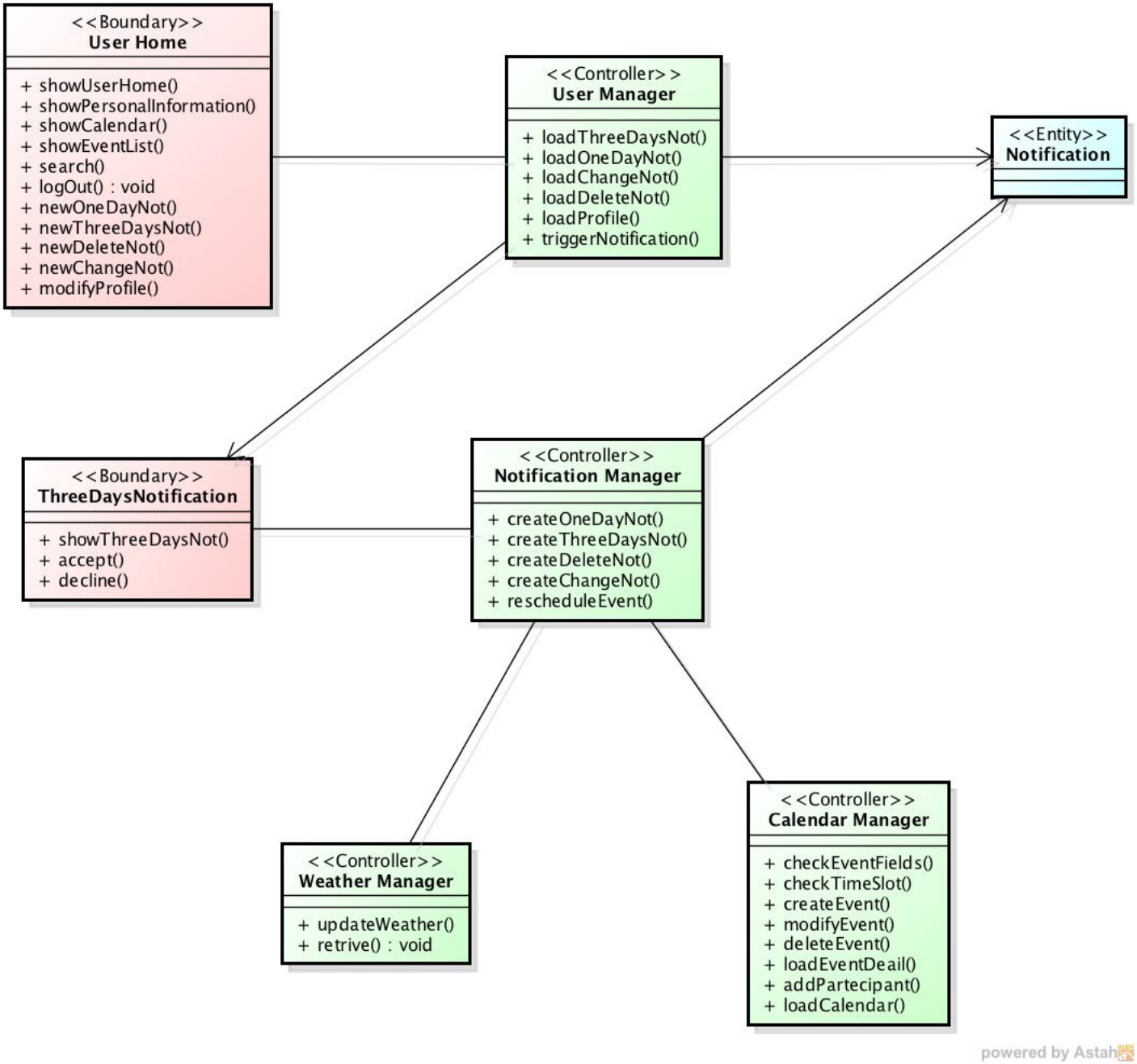

<<Boundary>>
Calendar
+ showCalendar()
+ selectEvent()
+ showOtherDays()
+ createEvent()
+ modifyEvent()
+ deleteEvent()
<<Controller>>
Calendar Manager
+ checkEventFields()
+ checkTimeSlot()
+ createEvent()
+ modifyEvent()
+ deleteEvent()
+ loadEventDetails()
+ loadCalendar()
+ addParticipant()
<<Entity>>
Calendar
- visibility : Boolean
+ addEvent()
+ getEvent()
+ modifyEvent()
+ deleteEvent()
+ getVisibility()
+ setVisibility()
<<Boundary>>
Event
+ showEventPage()
+ showEventDetails()
+ showEventCreationPage()
+ showInvitePeoplePage()
+ addParticipant()
+ showEventModificationPage()
+ showDeleteBox()
+ confirmCreation()
+ confirmDeletion()
+ confirmModification()
powered by Astah

4.5 Notification Managing

The two boundaries of this sub-system are *User Home* and *Three days Notification*.
The *Three Days Notification* boundary has the function of showing the new Three Days Notification, where the user can also accept or decline the date that the system proposes him.
There are two controllers:

- *User Manager*
  It has to show all the Notification of every type loading them with the *loadXNotification* methods. Moreover, it has a method called *triggerNotification* that is invoked when a new notification is created and allows the immediate visualization of the notification on the user page when he is online.

- *Notification Manager*
  It creates all the Notification of every type and in the case of the Three Days Before Notification it can also reschedule the event in case the user accepts the proposed date. Since the notifications are created when there is a modification or a deletion of an event or when new weather forecasts are retrieved, the controller has to interact with both the Weather Manager and the Calendar Manager.

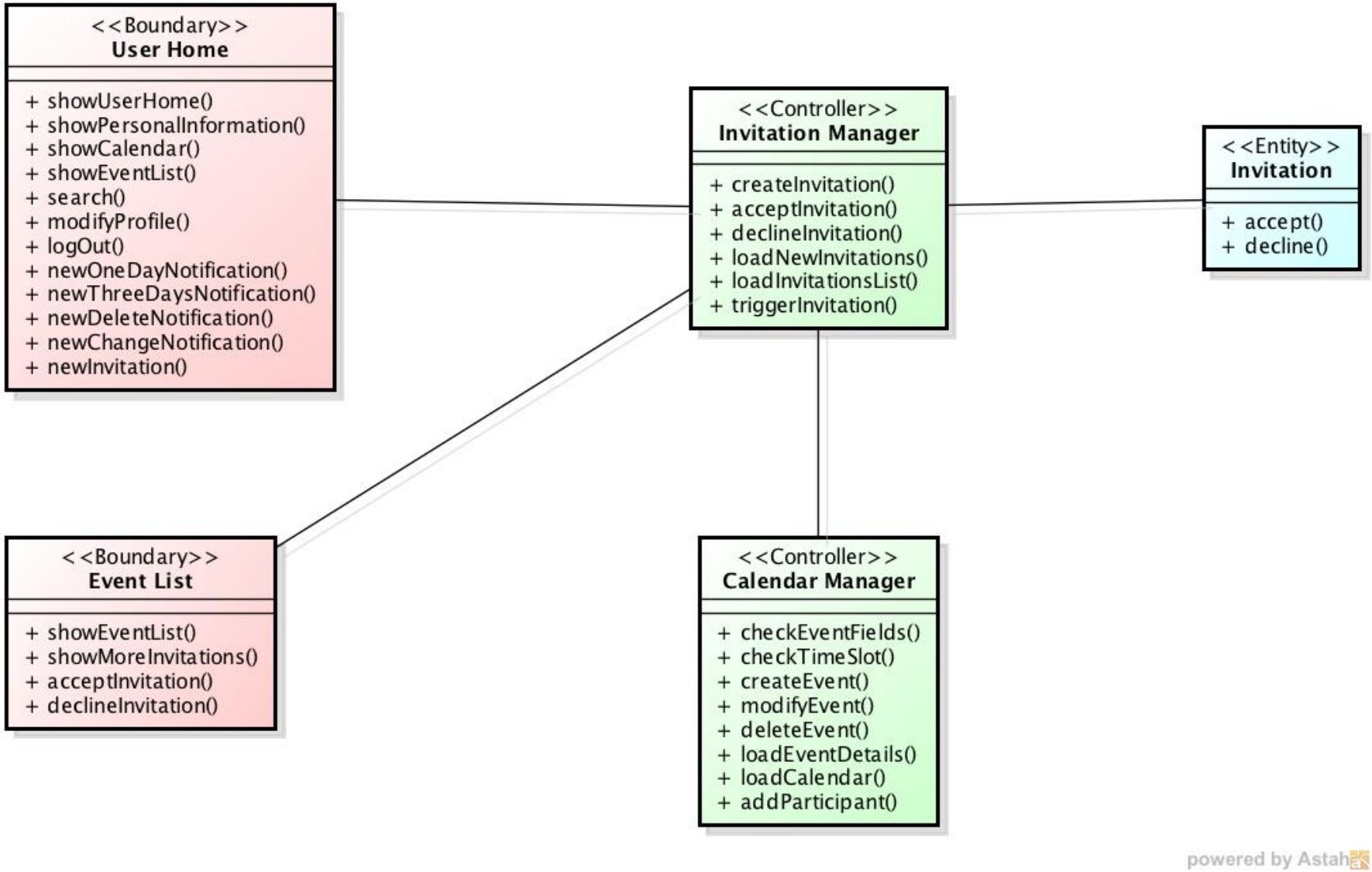

<<Boundary>>
User Home
+ showUserHome()
+ showPersonalInformation()
+ showCalendar()
+ showEventList()
+ search()
+ logOut() : void
+ newOneDayNot()
+ newThreeDaysNot()
+ newDeleteNot()
+ newChangeNot()
+ modifyProfile()
<<Controller>>
User Manager
+ loadThreeDaysNot()
+ loadOneDayNot()
+ loadChangeNot()
+ loadDeleteNot()
+ loadProfile()
+ triggerNotification()
<<Entity>>
Notification
<<Boundary>>
ThreeDaysNotification
+ showThreeDaysNot()
+ accept()
+ decline()
<<Controller>>
Notification Manager
+ createOneDayNot()
+ createThreeDaysNot()
+ createDeleteNot()
+ createChangeNot()
+ rescheduleEvent()
<<Controller>>
Weather Manager
+ updateWeather()
+ retrive() : void
<<Controller>>
Calendar Manager
+ checkEventFields()
+ checkTimeSlot()
+ createEvent()
+ modifyEvent()
+ deleteEvent()
+ loadEventDeail()
+ addPartecipant()
+ loadCalendar()
powered by Astah

4.6 Invitation Managing

The main boundary of that sub-system is *Event List*, which manages the interaction with the user when he is in the *Event List* page from which he can accept or decline invitations.
We've also reported the boundary *User Home* because the user can receive the invitations when he's in his personal page.
The controller is the *Invitation Manager* that manages the invitations by performing the following operations:

- Acceptance of an invitation.
- Decline of an invitation.
- Creation of new invitation.
- Load of the invitations sent to a user and.
- Visualization of new invitations by mean of the method called *triggerInvitation*.

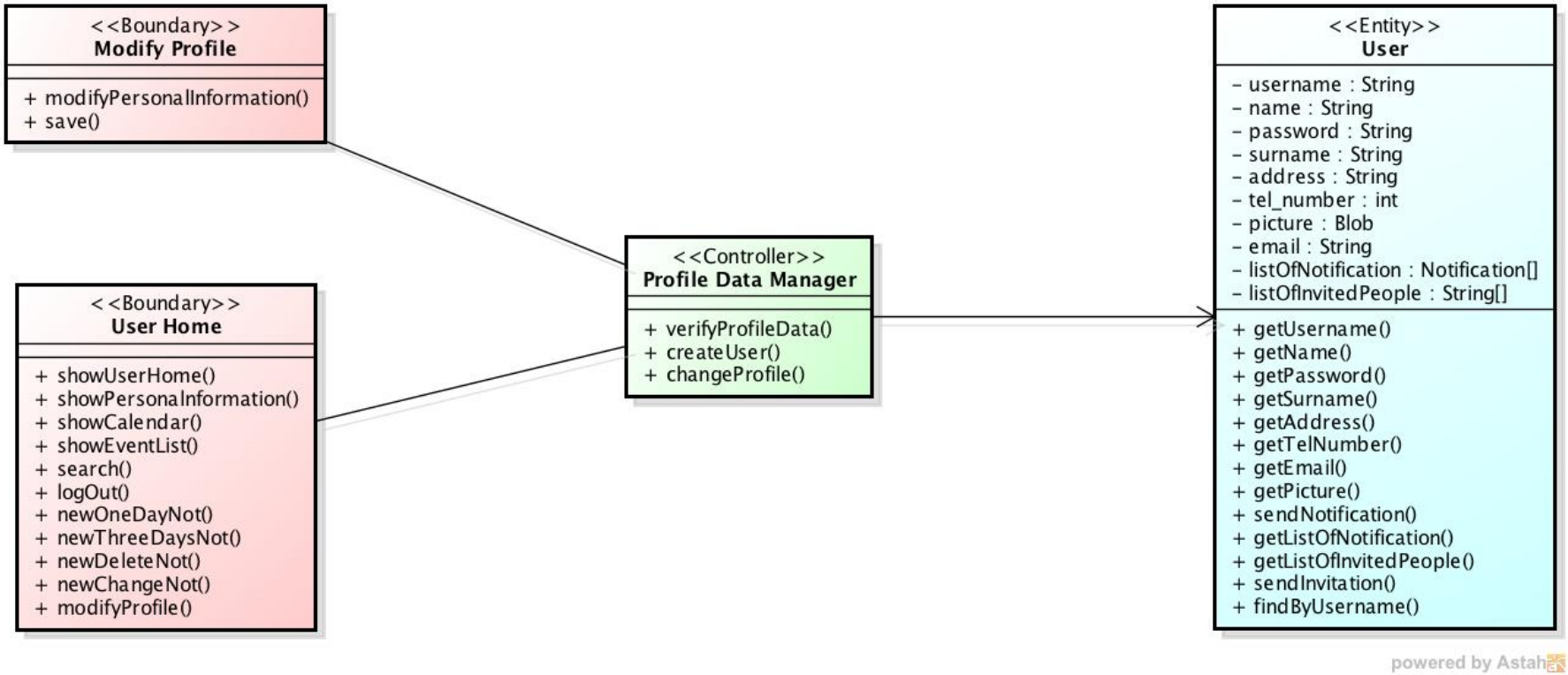
Since the *Invitation Manager* has to perform the creation of the invitations, which is strictly related to the creation of the event, it is associated to the *Calendar Manager*.


<<Boundary>>
User Home
+ showUserHome()
+ showPersonalInformation()
+ showCalendar()
+ showEventList()
+ search()
+ modifyProfile()
+ logOut()
+ newOneDayNotification()
+ newThreeDaysNotification()
+ newDeleteNotification()
+ newChangeNotification()
+ newInvitation()
<<Controller>>
Invitation Manager
+ createInvitation()
+ acceptInvitation()
+ declineInvitation()
+ loadNewInvitations()
+ loadInvitationsList()
+ triggerInvitation()
<<Entity>>
Invitation
+ accept()
+ decline()
<<Boundary>>
Event List
+ showEventList()
+ showMoreInvitations()
+ acceptInvitation()
+ declineInvitation()
<<Controller>>
Calendar Manager
+ checkEventFields()
+ checkTimeSlot()
+ createEvent()
+ modifyEvent()
+ deleteEvent()
+ loadEventDetails()
+ loadCalendar()
+ addParticipant()
powered by Astah

4.7 Profile Management

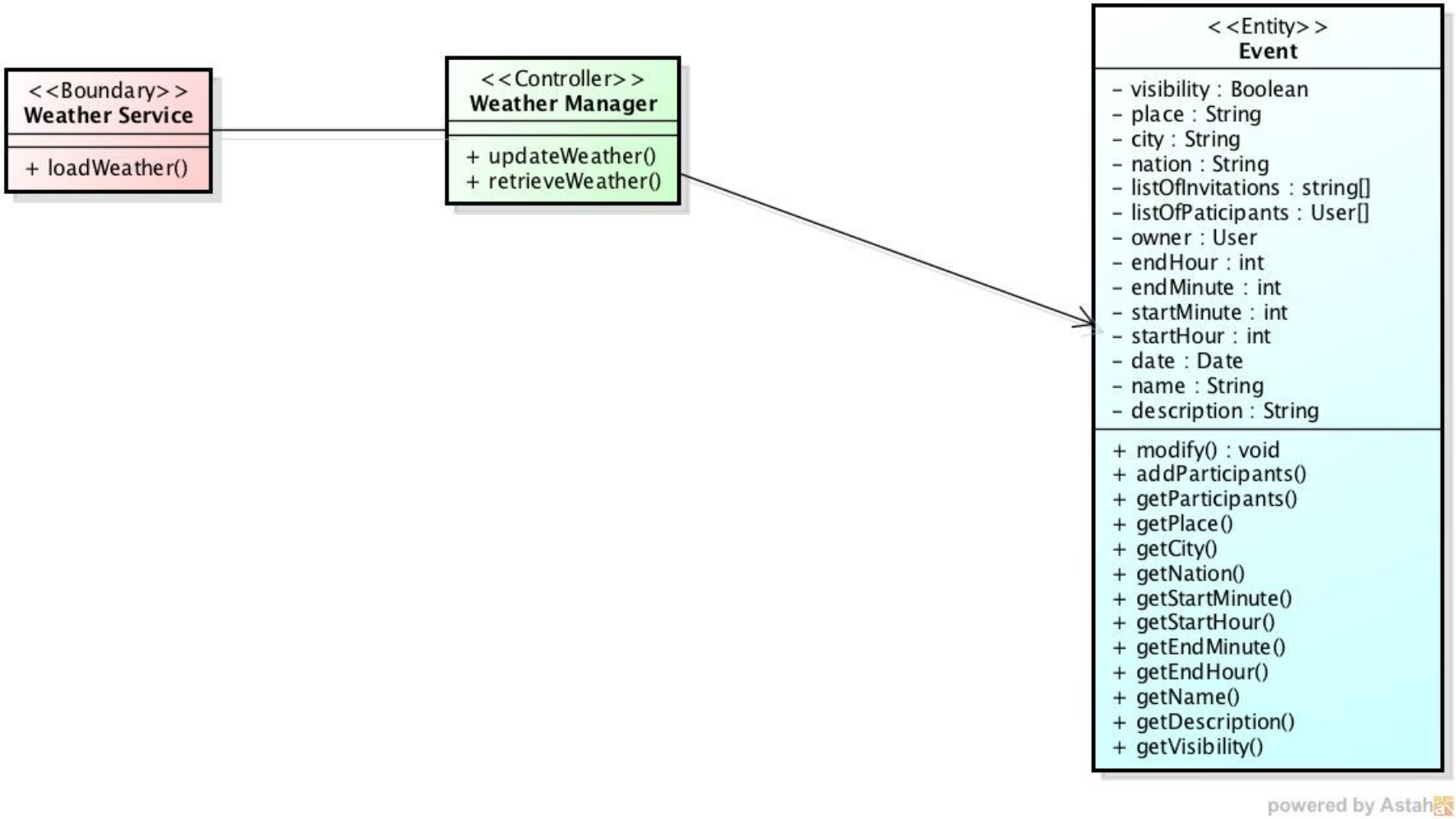
The boundaries of this subsystem are *Modify Profile* and *User Home*.
The first is the page where you can modify your personal information and save them by the two methods *modifyPersonalInformation* and *save*, the second is the same ad we have already explained before.
We have one controller:

- *Profile Data Manager* that has the function to verify the information the user has written in the form he has to compile when he wants to modify his personal information and then he has to *changeProfile* with the information that has been written.
  It has also to create a new User, by *createUser*, in fact the form the user has to fill in when he wants to modify his personal information is the same as the one he has to fill in when he wants to sign up.


<<Boundary>>
Modify Profile
+ modifyPersonalInformation()
+ save()
<<Boundary>>
User Home
+ showUserHome()
+ showPersonalInformation()
+ showCalendar()
+ showEventList()
+ search()
+ logOut()
+ newOneDayNot()
+ newThreeDaysNot()
+ newDeleteNot()
+ newChangeNot()
+ modifyProfile()
<<Controller>>
Profile Data Manager
+ verifyProfileData()
+ createUser()
+ changeProfile()
<<Entity>>
User
- username : String
- name : String
- password : String
- surname : String
- address : String
- tel_number : int
- picture : Blob
- email : String
- listOfNotification : Notification[]
- listOfInvitedPeople : String[]
+ getUsername()
+ getName()
+ getPassword()
+ getSurname()
+ getAddress()
+ getTelNumber()
+ getEmail()
+ getPicture()
+ sendNotification()
+ getListOfNotification()
+ getListOfInvitedPeople()
+ sendInvitation()
+ findByUsername()
powered by Astah

4.8 Weather Managing

This sub-system manages the update of weather forecasts of the events. In that case the boundary, called *Weather Service*, doesn't manage the interaction with the user but the interaction with another system, in our case the web service which we retrieve weather information from. The controller periodically carries out the operation of updating the weather forecasts of all the events by retrieving information from the web service. Of course it has to interact with the *Notification Manager* in order to trigger new notifications in case of bad weather forecasts.


<<Boundary>>
Weather Service
+ loadWeather()
<<Controller>>
Weather Manager
+ updateWeather()
+ retrieveWeather()
<<Entity>>
Event
- visibility : Boolean
- place : String
- city : String
- nation : String
- listOfInvitations : string[]
- listOfPaticipants : User[]
- owner : User
- endHour : int
- endMinute : int
- startMinute : int
- startHour : int
- date : Date
- name : String
- description : String
+ modify() : void
+ addParticipants()
+ getParticipants()
+ getPlace()
+ getCity()
+ getNation()
+ getStartMinute()
+ getStartHour()
+ getEndMinute()
+ getEndHour()
+ getName()
+ getDescription()
+ getVisibility()
powered by Astah

5. Sequence diagrams

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

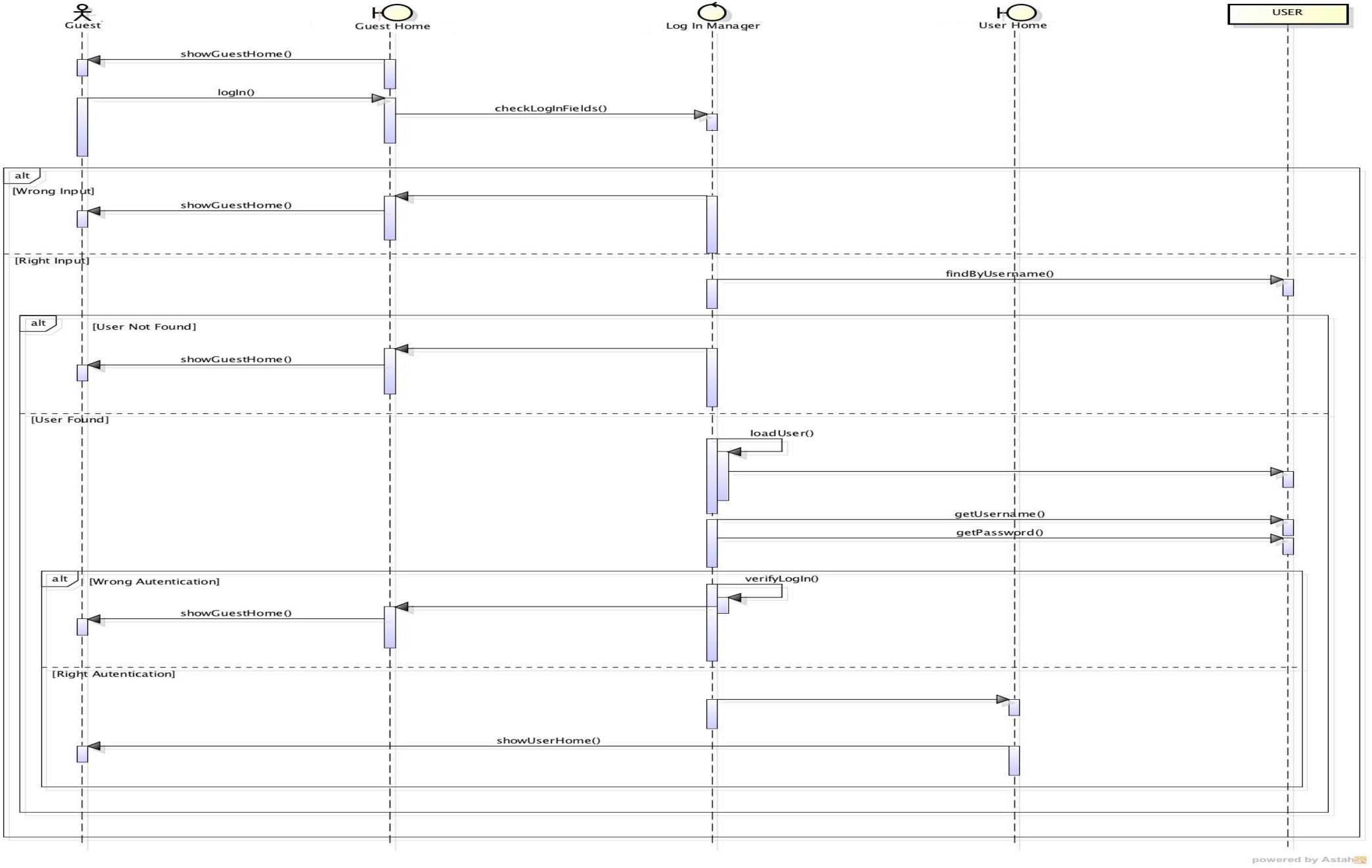
All the methods used are the methods listed into the BCE in boundaries, controls and entities.

The list of the Sequence Diagram we are going to show is this:

- Log In
- Sign Up
- Search
- Create Event
- Three Day Before Notification
- Accept Invitation

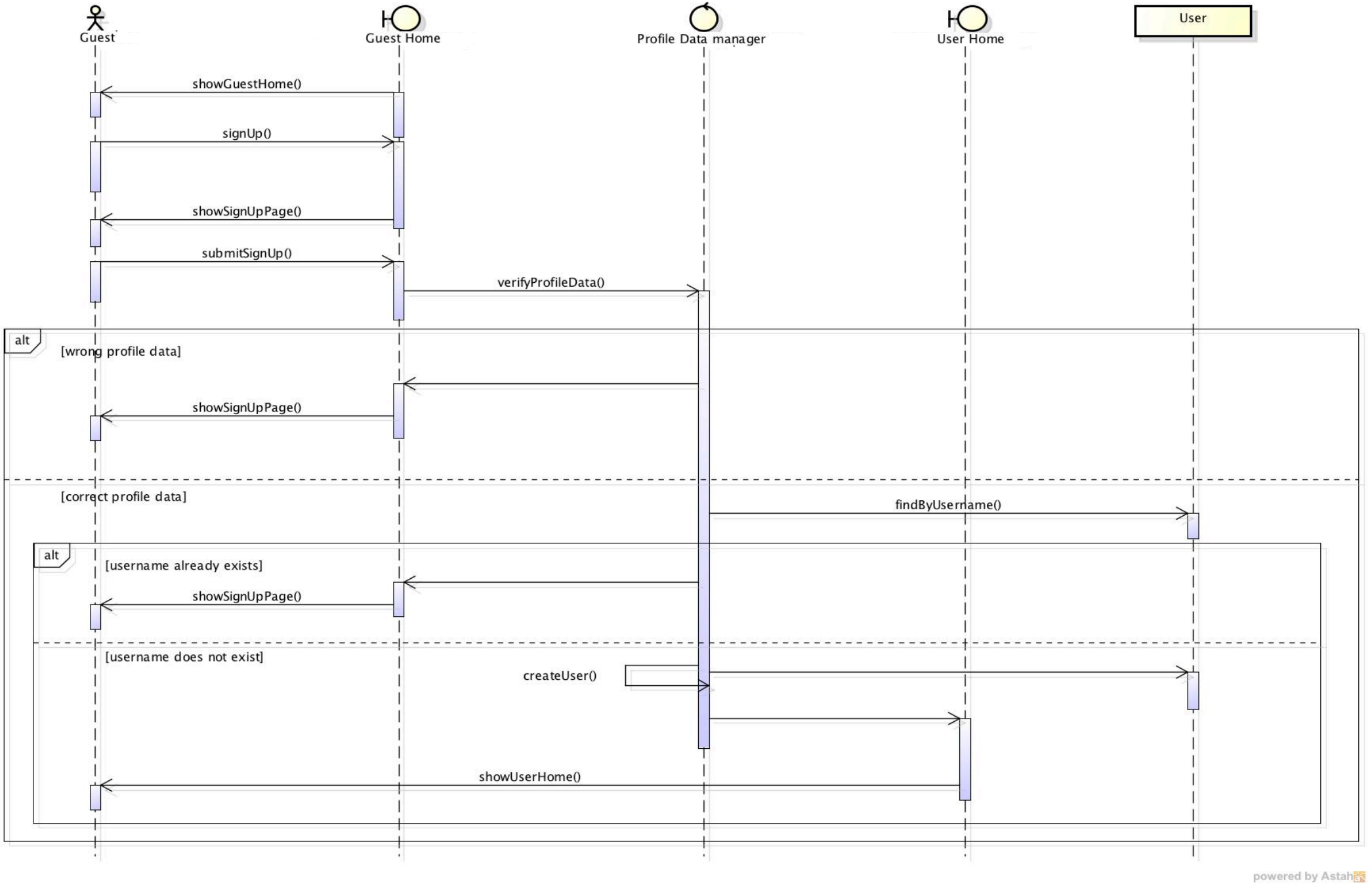
5.1 Log in Sequence Diagram

The following sequence diagram represents the Log In for a user.



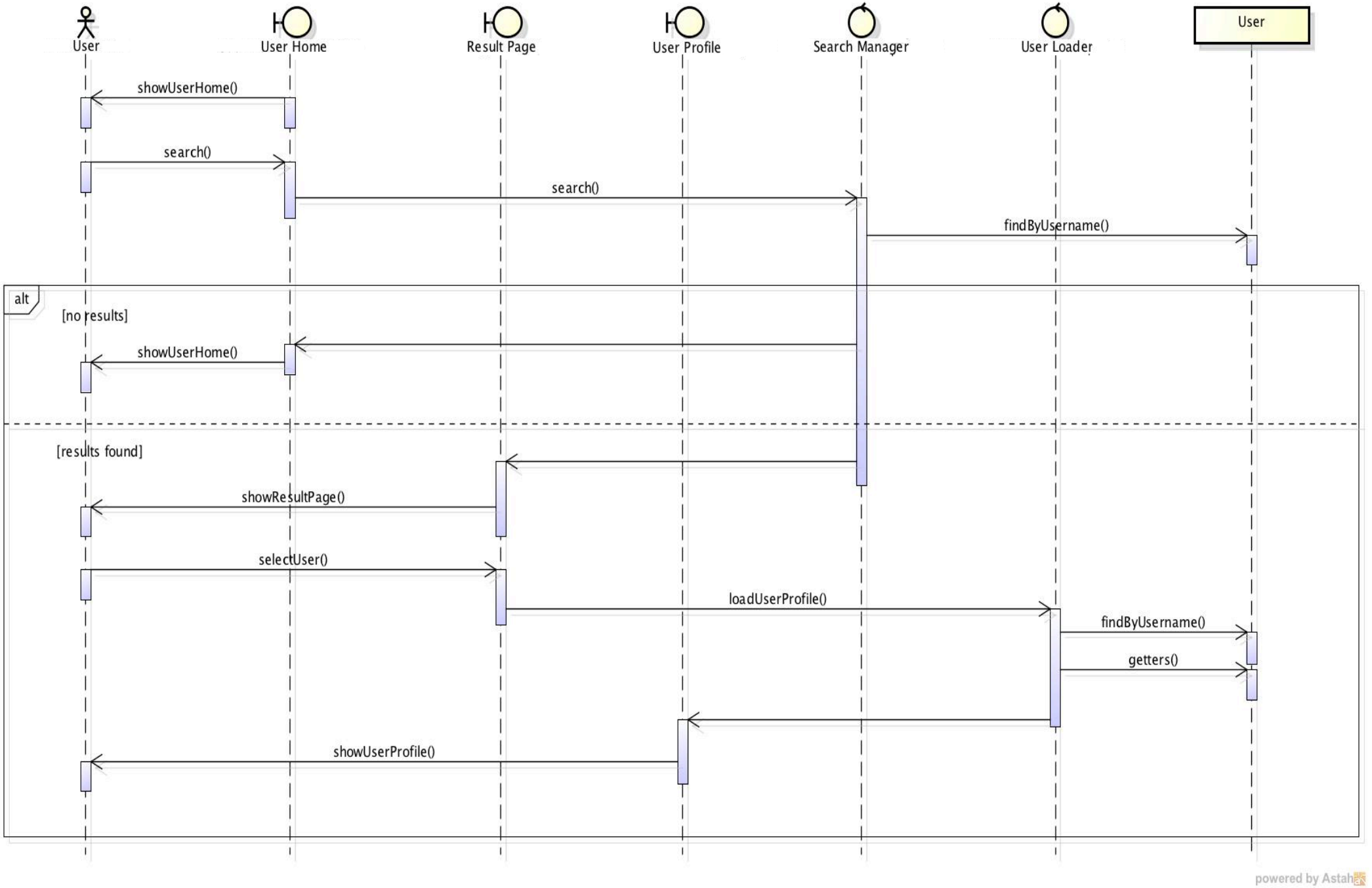
5.2 Sign Up Sequence Diagram

The following sequence diagram represents the Sign Up of a new user.



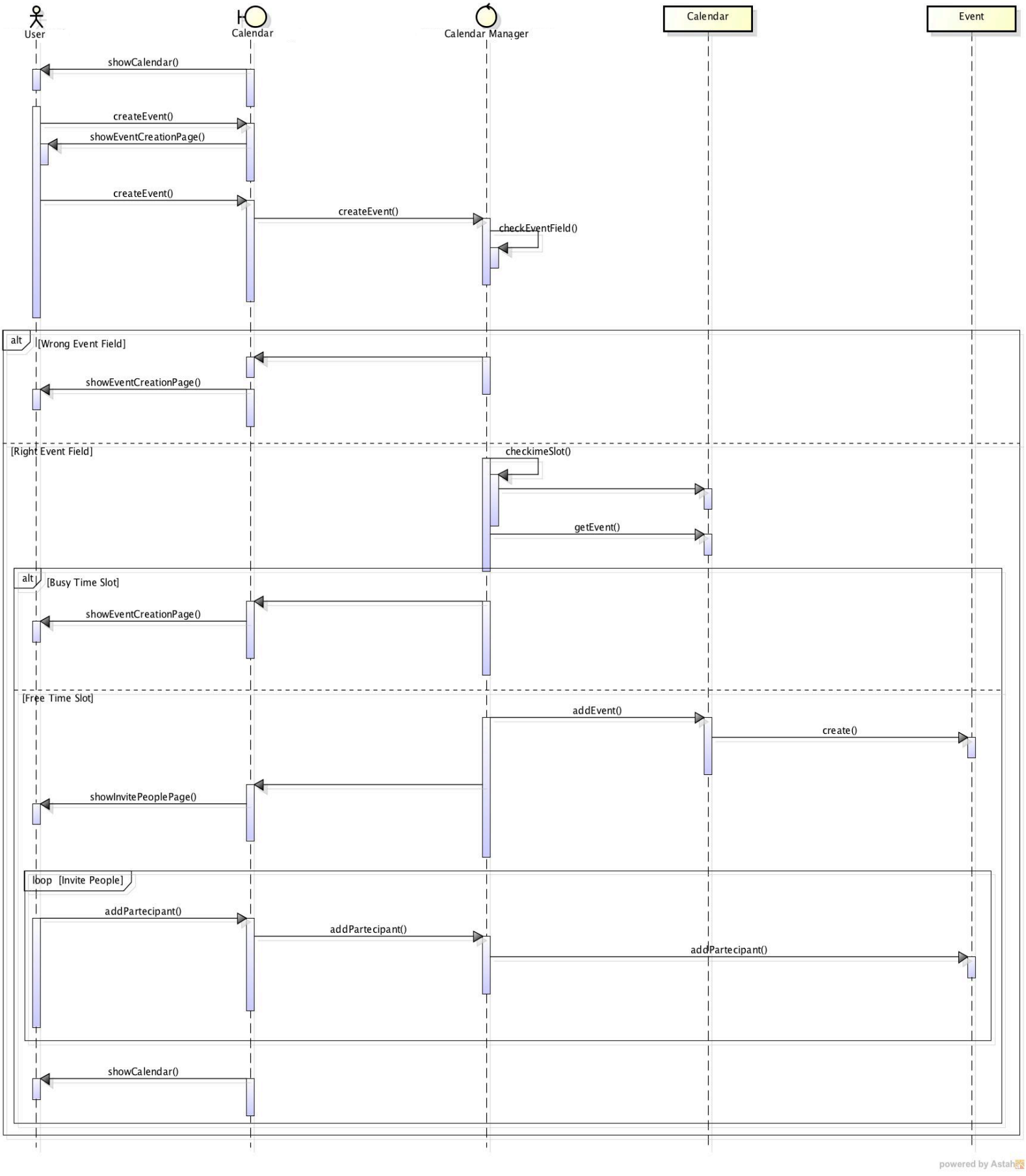
5.3 Search Sequence diagram

The following sequence diagram represents the search functionality.



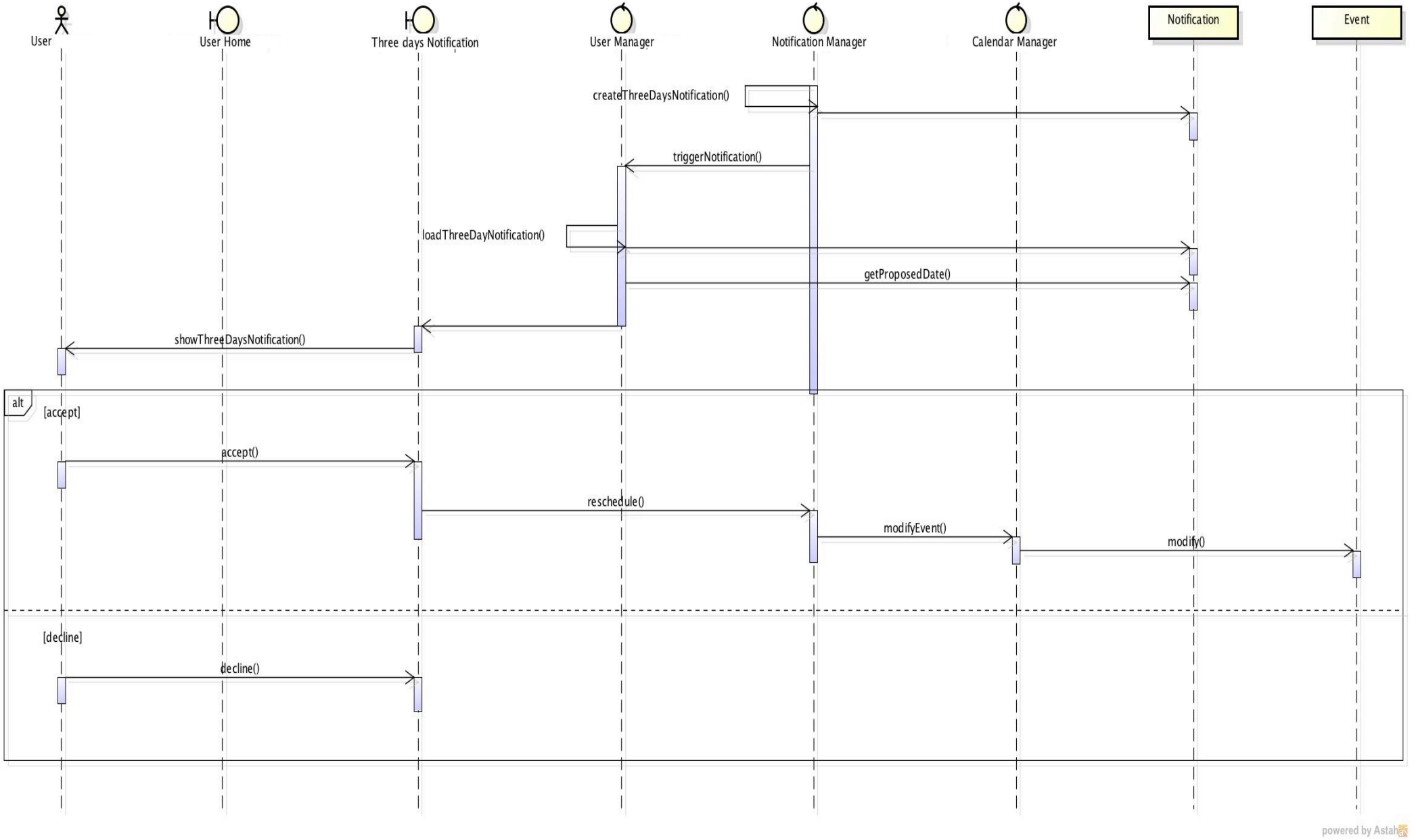
5.4 Create Event Sequence Diagram

The following sequence diagram represents the Create an Event functionality.



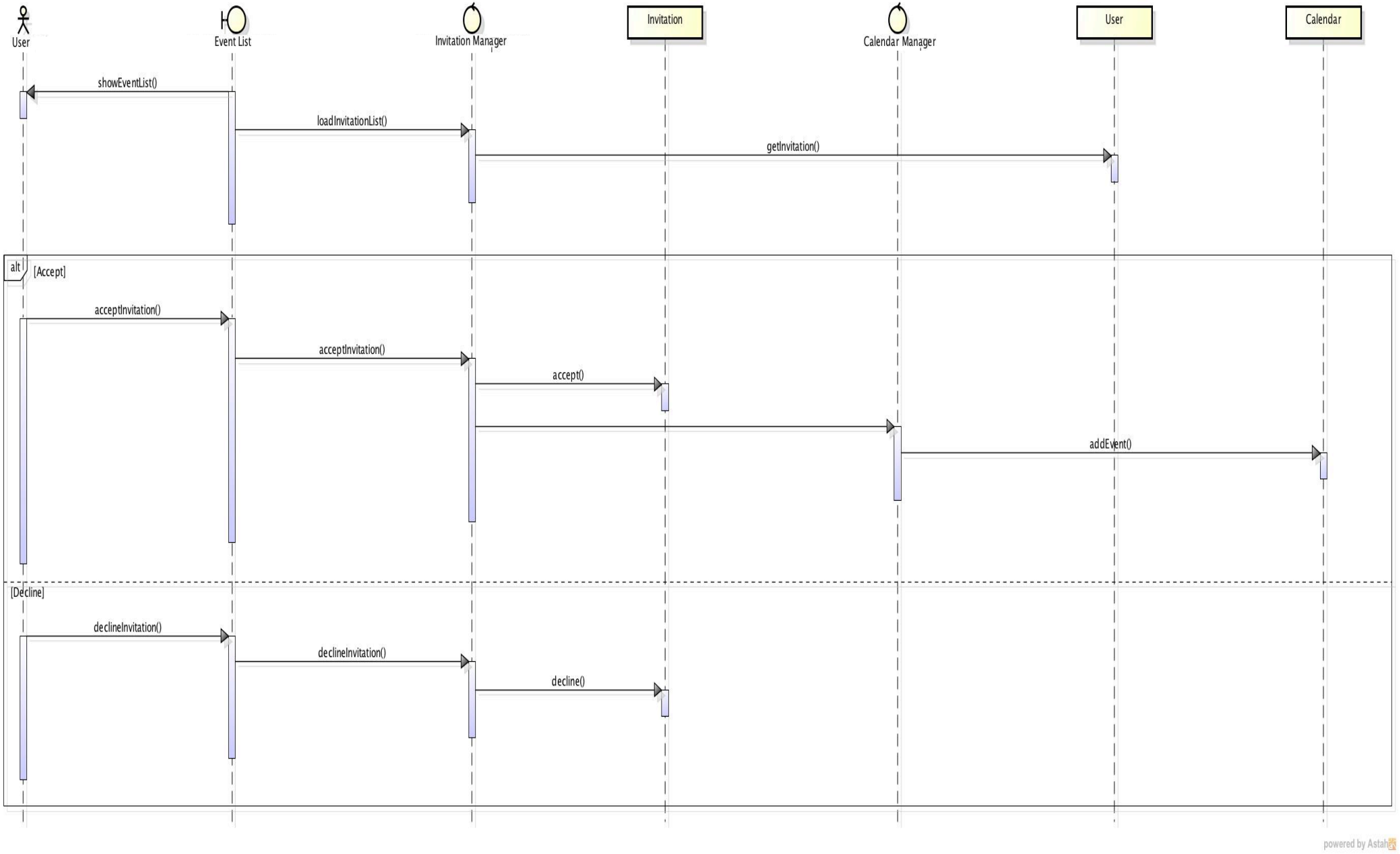
5.5 Three Days Before Notification Sequence Diagram

The following sequence diagram represents the visualization of a Three Days Notification; it assumes that it's starting the creation of a new modification, so all previous operations are not displayed in the diagram.



5.6 Accept Invitations

The following sequence diagram represents the Accept Invitations functionality.



For redacting and writing this document we spent 20 hours per person