SWIM v2

Design Document

Authors:

Affetti Lorenzo

Canidio Andrea

December 21st 2012

Summary

1. ARCHITECTURE DESCRIPTION 5

1.1. JEE ARCHITECTURE OVERVIEW 5
1.2. IDENTIFYING SUB-SYSTEMS 6

2. PERSISTENT DATA MANAGEMENT 8

2.1. CONCEPTUAL DESIGN 8
2.2. LOGICAL DESIGN 10
2.2.1. ER Restructuration 10
2.2.2. Translation to Logical Model 11

3. USER EXPERIENCE 14

3.1. HOMES 14
3.2. MESSAGING FUNCTIONALITY 17
3.3. PROFILE MANAGING 19
3.4. USER SEARCH AND FRIENDSHIP MANAGEMENT 21
3.5. HELP REQUEST MANAGEMENT 23

4. BCE DIAGRAMS 25

4.1. ENTITY OVERVIEW 26
4.2. SIGN UP AND LOG IN 27
4.3. ADMINISTRATOR 29
4.4. USER 31
4.4.1. Search 31
4.4.2. Conversations management 33
4.4.3. Friendship management 34
4.4.4. Help requests management 35
4.4.5. Profile management 37

5. SEQUENCE DIAGRAMS 38

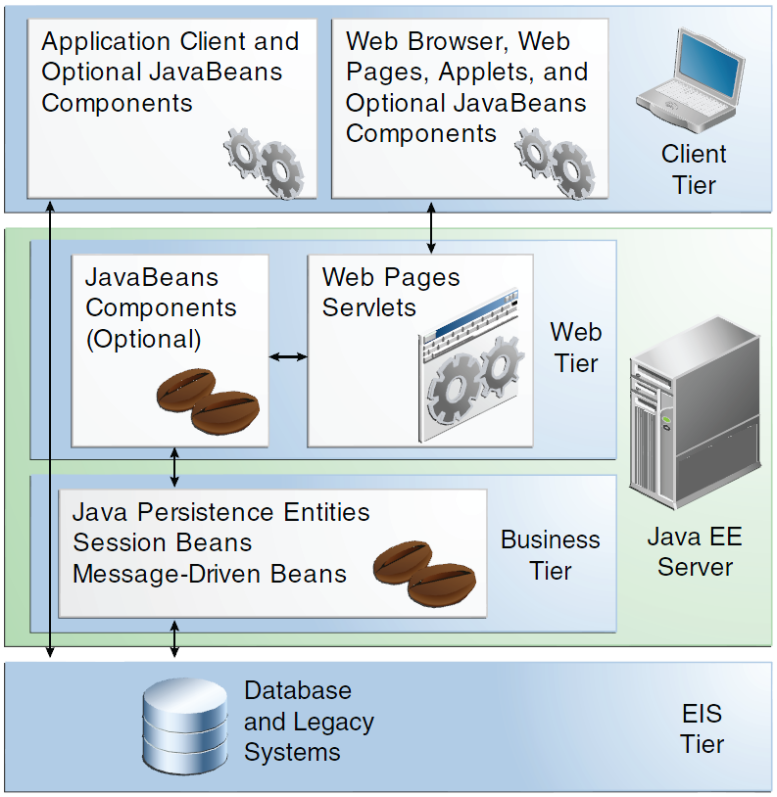
5.1. LOG IN 38
5.2. MY HELP REQUESTS 40
5.3. SEND A MESSAGE STARTING FROM A SPECIFIC CONVERSATION 42
5.4. SEND A MESSAGE STARTING FROM CONVERSATIONS 43
5.5. BROWSE HELP REQUESTS 44
5.6. SEARCH FOR A USER 45

6. FINAL CONSIDERATIONS 46

1. Architecture Description

1.1. JEE Architecture Overview

Before starting to explain our application's 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.

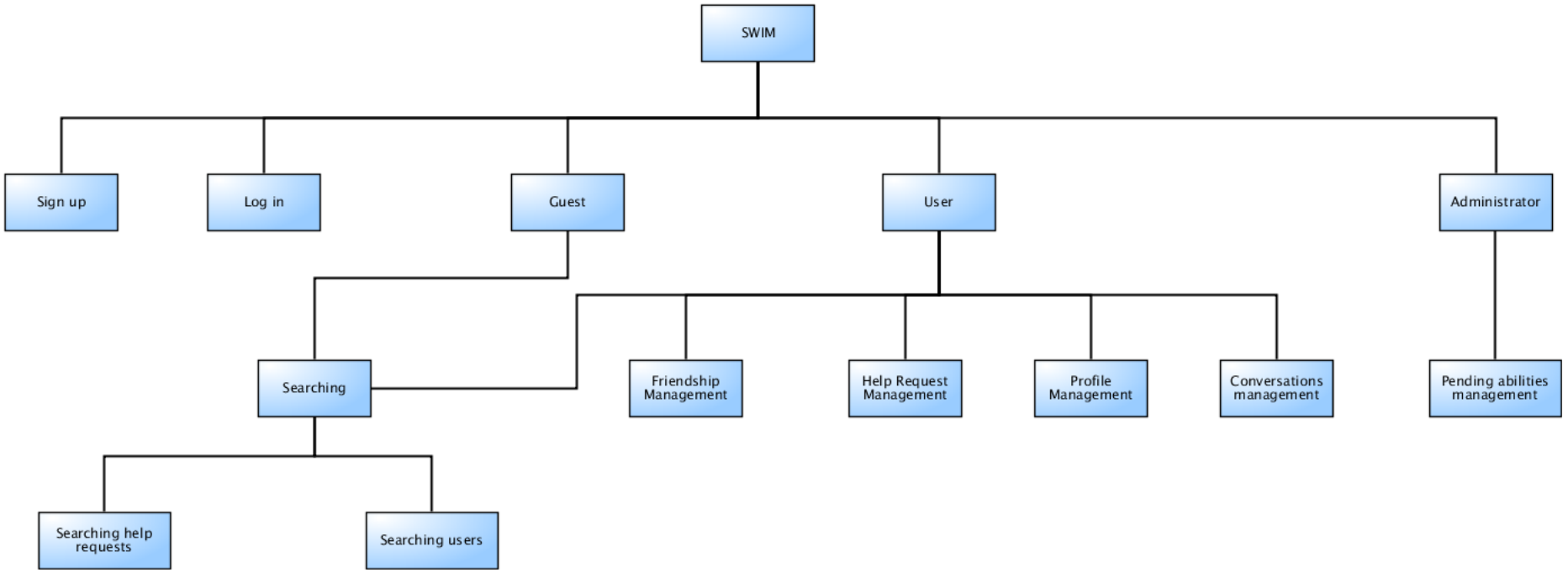
  This tier receives the requests from the client tier and forwards the pieces of data collected to the business tier waiting for processed data to be sent to the client tier, eventually formatted;

- **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 Sub-Systems

We decide to adopt a top-down approach at least at this point of the project. Maybe, once defined the sub-systems, we will adopt a bottom-up approach in order to create more reusable components.
So we think it is now necessary to decompose our system into other sub-systems, in order to make it easy to understand the issues that we will find in implementing functionalities and to separate, logically, groups of functionalities and state clearer their interaction.
We separate our systems into these sub-systems:

- Sign up sub-system;
- Log in sub-system;
- User sub-system;
    - Help request sub-system;
    - Messaging sub-system;
    - Profile managing sub-system;
    - Friendship sub-system;
- Administrator sub-system;
- Data sub-system.

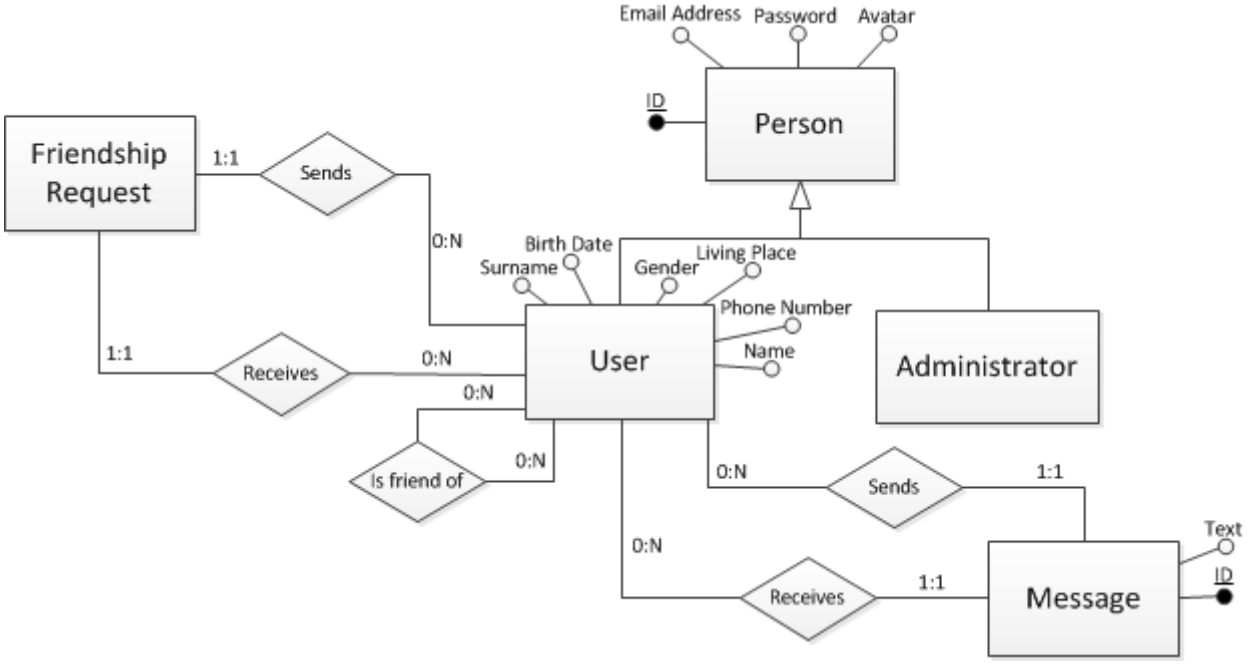

SWIM
Sign up
Log in
Guest
User
Administrator
Searching
Friendship Management
Help Request Management
Profile Management
Conversations management
Pending abilities management
Searching help requests
Searching users

2. Persistent Data Management

Our data is stored into a relational database. You can find below the design of our database in terms of Entity-Relationship Diagram.
Moreover we will explain in a detailed way entities, relations and, in general, the motivations of our design.

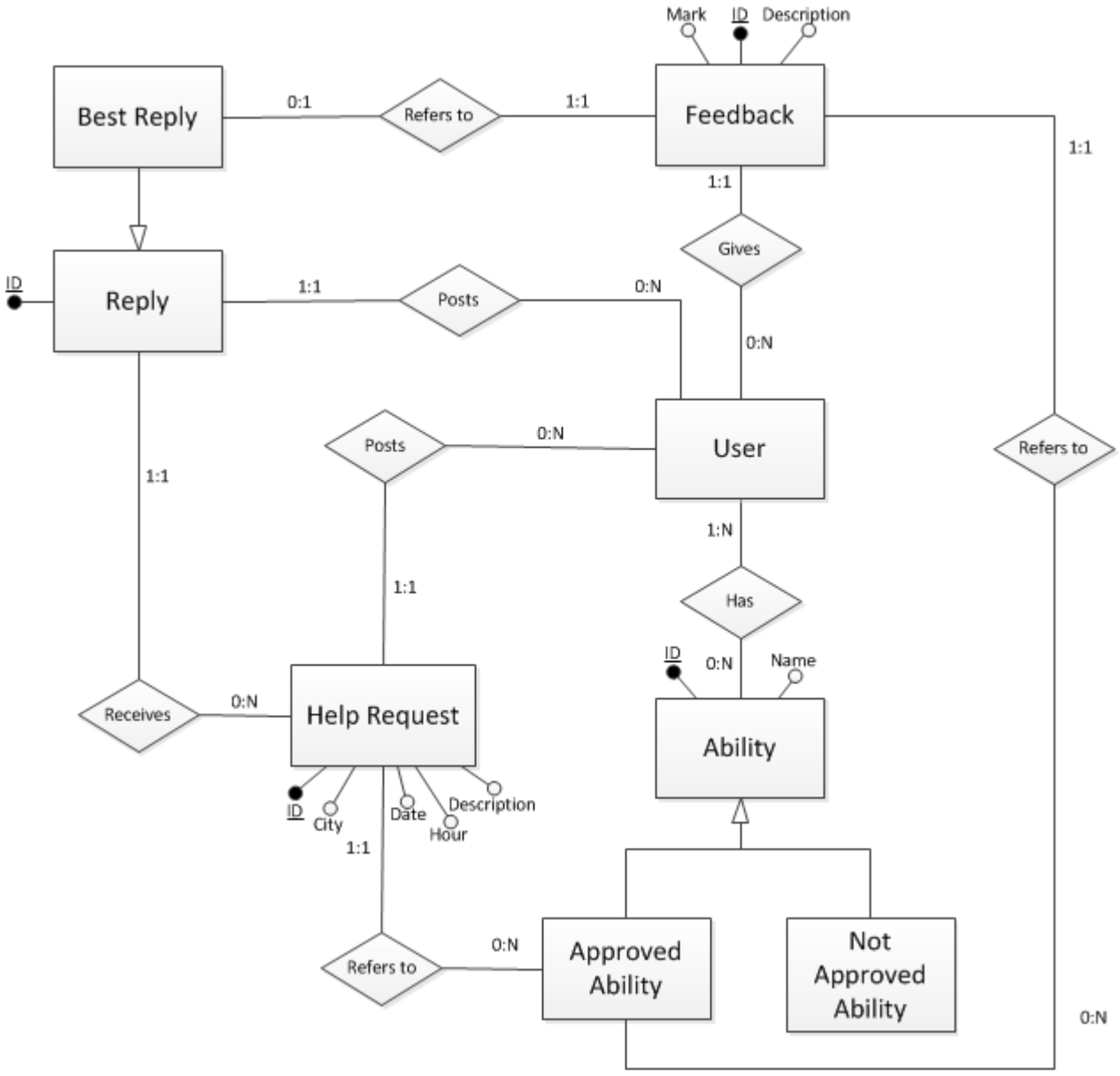
2.1. Conceptual Design

Conceptual design allows to start thinking about the data we want to stor and about the relations between them.
We decided to split up the design into two different parts as, if we gave it in an unique diagram, it would be too big and it could bring to misunderstandings. In the first scheme we represent the Friendship and Messaging part of the design.



In our system we have three types of person: *Guest*, *User* and *Administrator*.
Only the last two types need some information to be stored and some of these are common, so we decided to make the two entities inherit from the generic entity *Person*.
Moreover we added a *Friendship Request* entity to model the requests.
As they must be sent by some user and received by another we created a

relation *Send* and *Receives* between them. The *Friendship Request* entity is useful only when the *Friendship Request* is pending, so we created a relation *Is friend of* just to model the connection created after the *Friendship Request* is accepted.
To store the messages sent we created a new entity named *Message* and we created two more relations between *User* and *Message* named *Sends* and *Receives* to model the sender and the receiver of the message.



The *User* can have many abilities so we created the *Ability* entity, but as an *Ability* can be approved by the administrator we created two entities: *Approved Ability* and *Not Approved Ability*.

To model the help request mechanism we added *Help Request* (with some useful information like *City, Date, Hour* and *Description*), *Reply* (with its generalization *BestReply*) and *Feedback.* These entities are related between them through these relations:

- *Refers to* is between *Approved Ability* and *Help Request*, meaning that one *Help Request* is referred to a mandatory *Ability*.
- A *Help Request* can have some *Replies*, but a *Reply* must belong to only one *Help Request*, so we used the relation *Receives*.
- A *Help Request* is posted by one mandatory *User* and a *Reply* is given by one *User* too, so we used respectively two *Post* relations with same cardinality.
- A *Feedback* is referred to a *BestReply*, this means that only these particular *Reply* can receive an optional Feedback (*Refers to* relation);
- A *Feedback* is related to a *User* through *Gives* relation, because it is better to register which *User* gives a *Feedback* even if it is possible to calculate it (in terms of *queries*) from the *HelpRequest* associated to the *BestReply*. This redundancy is added to perform easier and faster queries over the database.
- A *Feedback* is related to an *Approved Ability* through *Refers to* relation for the same reason of *User*.
  This relations was added for the same reason as above.

2.2. Logical Design

Logical Design has the aim to better represent the database structure of our system, but, in order to build this model from the ER diagram drawn above, we have to perform some transformations.

2.2.1. ER Restructuration

- First of all we have two entities: *User* and *Administrator* which are very similar and they inherit from another entity, *Person*.
  The approach we take is to merge *User* and *Administrator* into *Person*, in

order to obtain only one entity, nevertheless we have to introduce a new attribute *Type* to discriminate *User*'s tuples from *Administrator*'s ones.

- We do the same thing between *Accepted Ability*, *Not Accepted Ability* and *Ability* merging them into *Ability* and creating a Boolean attribute *Accepted*.
- We do something very similar with *Reply* and *BestReply* merging them into the entity *Reply* and adding one Boolean attribute *Best*.

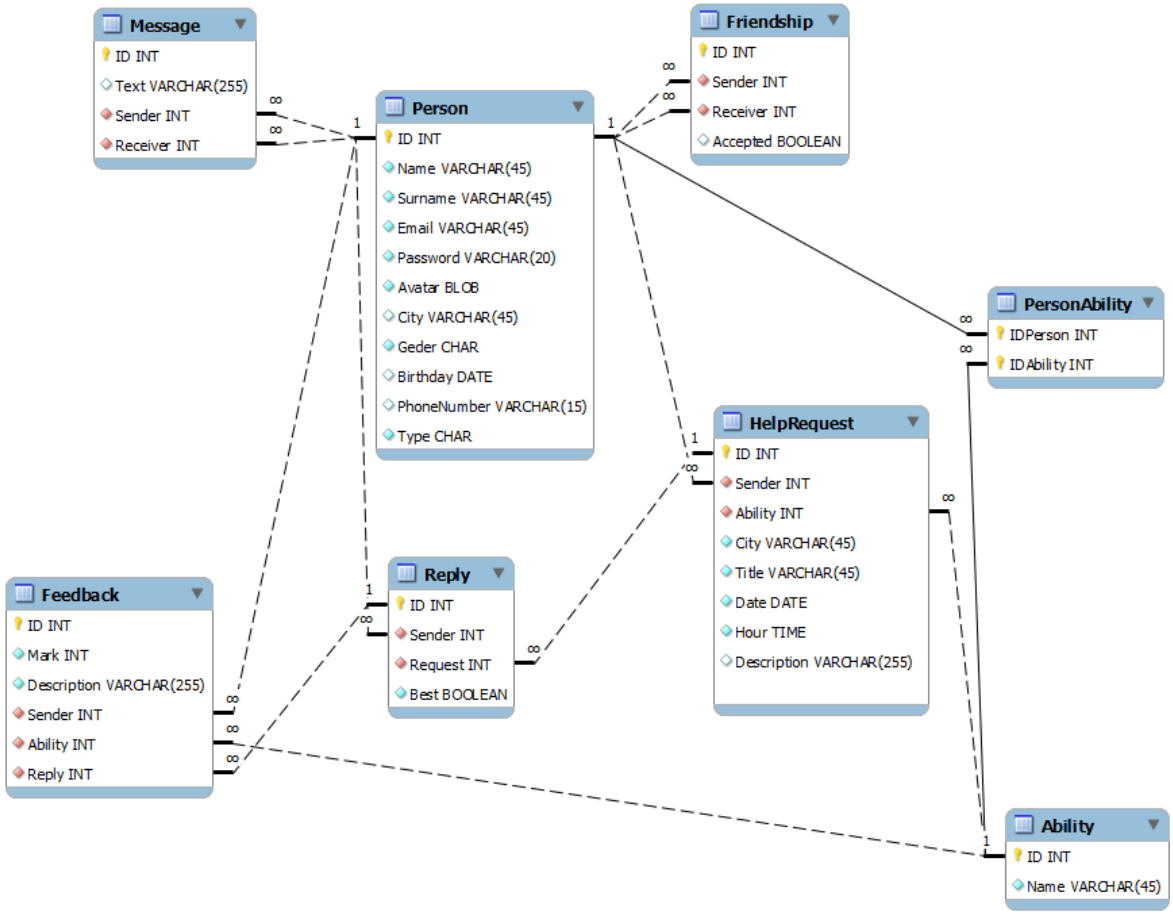
2.2.2. Translation to Logical Model

1. Relations *Sends* and *Gives* between *User* and *Message* are translated inserting two foreign keys into the table *Message* named *Sender* and *Receiver*.
2. Relation *Refers to* between *Ability* and *Feedback* is translated into the insertion of a foreign key into the table *Feedback* named *Ability*.
   We do the same thing with tables *Reply* and *User* adding respectively *Reply* and *Sender* foreign keys.
3. Relation *Has* has 0:N cardinalities on both edges so we create a new table *PersonAbility* with the two foreign keys *IDPerson* and *IDAbility*.
4. In the table *Reply* we add two foreign keys *Sender* and *Request* because of the two many to many relations *Receives* and *Posts* that associate it with *Help Request* and with *User* respectively.
5. In the table *Help Request* we add two foreign keys *Ability* and *Sender* because of the two many to many relations *Refers* and *Posts* that associate it with *Accepted Ability* and with *User* respectively.
6. *Is friend of* relation, being a many to many relationship, is translated into a table called *Friendship*.
7. For an easier management of the *Friendship* and *FriendshipRequest*, we decide to join the two tables into the single table *Friendship* (the table that was already introduced above)with the Boolean attribute *Accepted* that states if the two Users are friends.

If *Accepted* is false, there is a pending friendship request between the users.

If *Accepted* is true, the two users are friends.

We can draw below the logical model:



The final model has the following physical structure:

- **Person** (ID, Name, Surname, Email, Password, Avatar, City, Gender, Birthday, PhoneNumber, Type);
- **Message** (ID, Text, *Sender, Receiver*);
- **Friendship** (ID, *Sender, Receiver*, Accepted);
- **Ability** (ID, Name);
- **PersonAbility** (*IDPerson, IDAbility*);
- **HelpRequest** (ID, *Sender, Ability*, City, Title, Date, Hour, Description);
- **Reply** (ID, *Sender, Request*, Best);
- **Feedback** (ID, Mark, Description, *Sender, Ability, Reply*).

We have underlined the primary keys of each table and we put in Italic the foreign keys that each table contains (maybe it is not so understandable which table foreign keys refer to, but it is easily understandable watching the schema drawn above).

3. User Experience

In this paragraph we want to describe the User Experience (UX) given by our system to its users. We used a Class Diagram with appropriate stereotypes *<<screen>>*, *<<screen compartment>>* and *<<input form>>*s and normal Classes to let understand how our Experience was thought. While *<<screen>>* represents pages, *<<screen compartment>>* represents parts of the page that can be shared with others. *<<input form>>*, eventually, represents some input fields that can be fulfilled by a user (this information will be submitted to the system clicking on a button).
We decided also to split the UX Diagram in functionalities as for Use Cases in the RASD Document in order to better understand the whole Diagram.
Each of the paragraphs below is titled with the name of the functionality represented by the UX Diagram drawn.

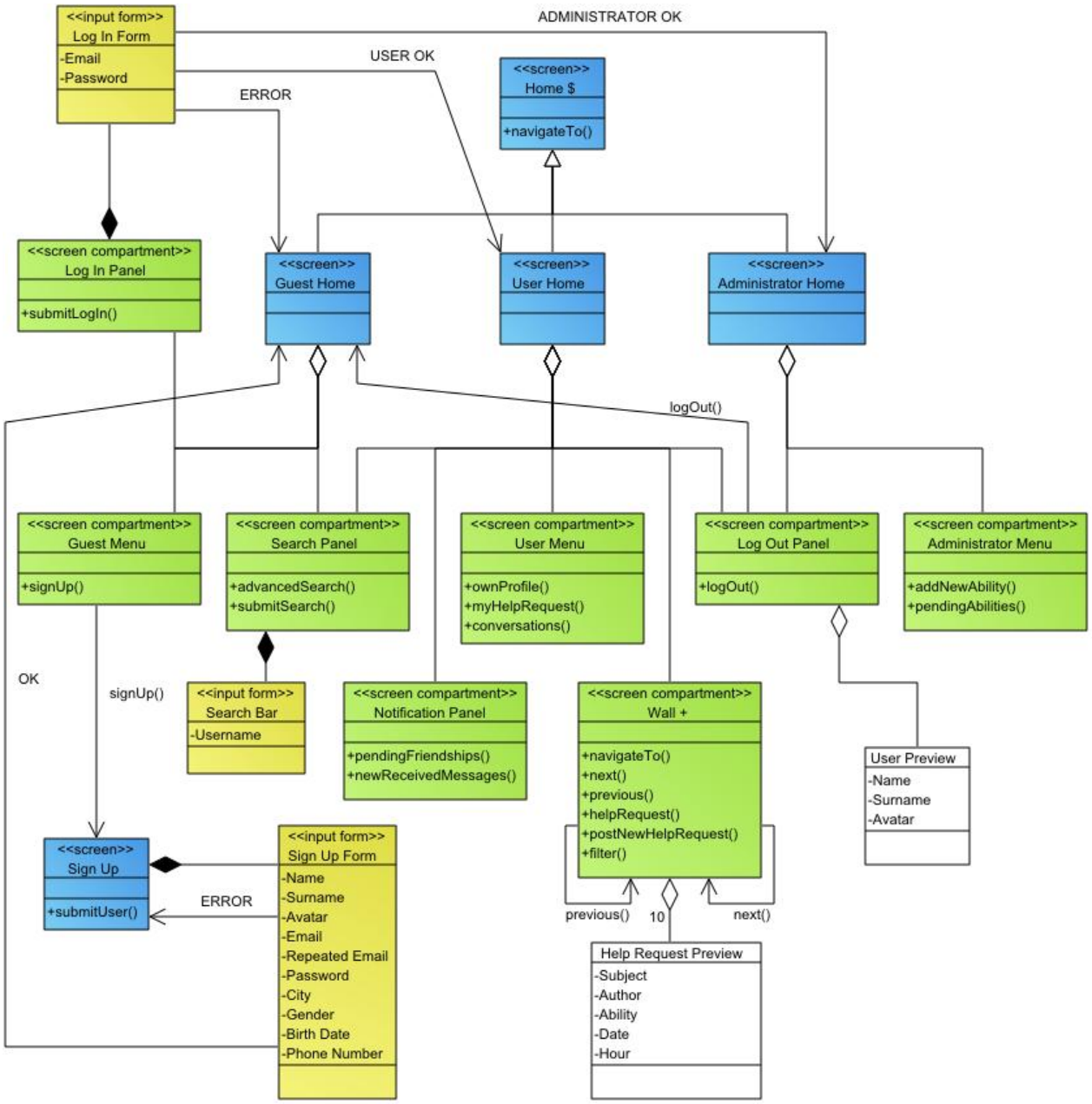
3.1. Homes

We can see below the Diagram that represents the different home pages based on the type of generic user that is interacting with the system. All of the three pages inherit from a Home screen that is also marked with the landmark *$*, so it is reachable from every page of the system.
For an easier interpretation of the schema we decided to draw some components that are contained in all the screens, but we decided to draw them only in this UX Diagram, if not, it would have been very difficult to understand other diagrams.
Here is a list of these elements:

- **Log in Panel** and **Guest Menu**:
  They are in all guest's pages;
- **Notification Panel**, **User Menu** and **Wall**:
  They are in all user's pages;

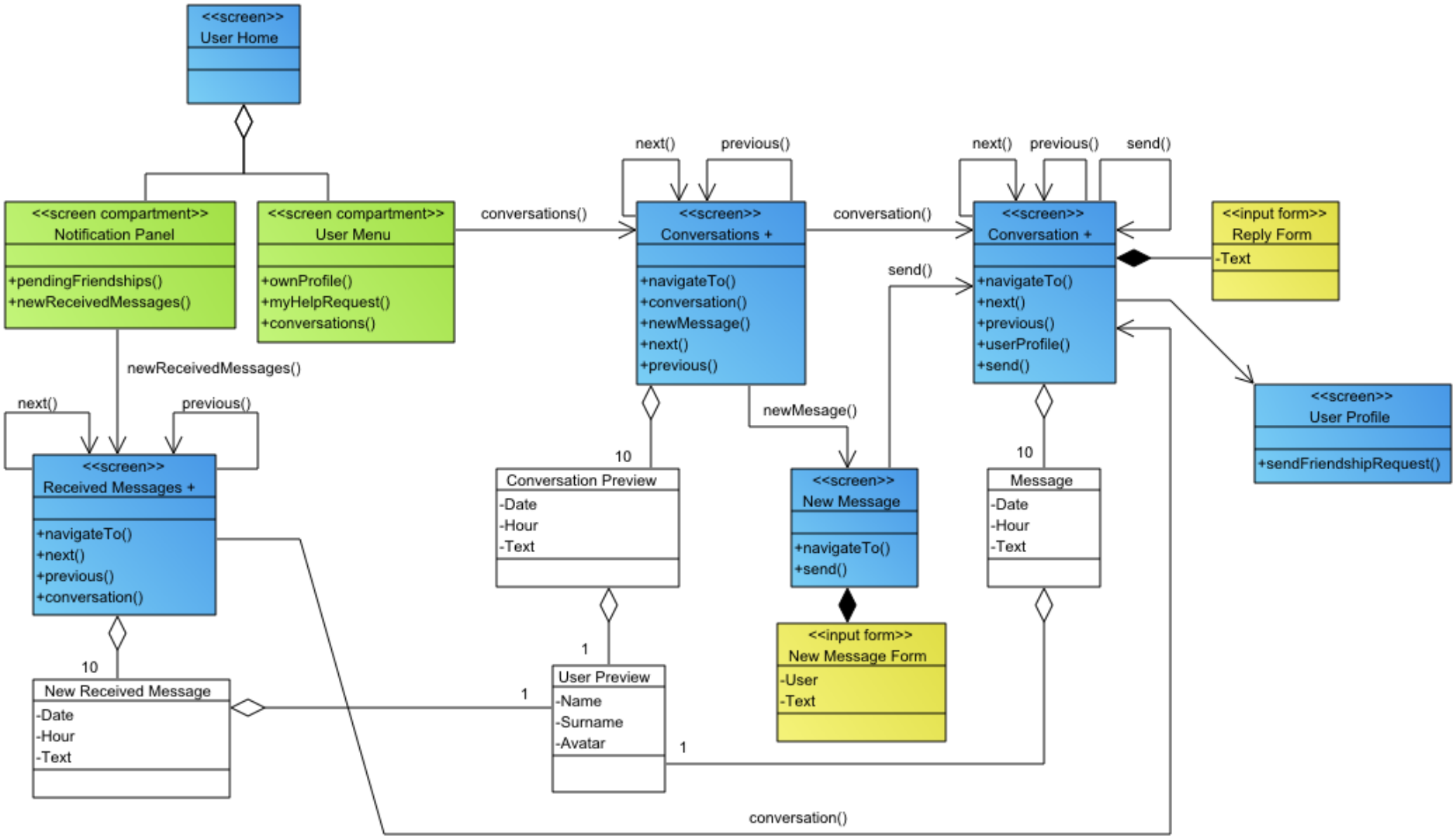
- **Administrator Menu:**
  It is in all administrator's pages;
- **Search Panel:**
  It is in all pages of guest and user;
- **Log Out Panel:**
  It is in all users and administrator's pages.

Moreover, from a *Guest Home* we, can log in through the *Log In Panel* and, if there is no error, we will be redirected to our home page based on the type of user we are.
If there is an error we will be redirected to the *Guest Home* again.
If we've already logged in we can perform a logout through the *Log Out Panel*.
Menus contain the principal functionalities exploited by each type of generic user.
The *Notification Panel* shows if there are new messages or pending friendships and allows to reach the associated pages.
The last functionality expressed by this UX Diagram is the possibility of signing up reachable from the *Guest Menu*. If we click on the right link the *Sign Up* screen is shown. This screen contains a form to be fulfilled with all the data of the new user. Eventually, we can submit the data to the system (that will have different clear behaviors in case of error or not).


<<input form>>
Log In Form
-Email
-Password
ADMINISTRATOR OK
USER OK
ERROR
<<screen>>
Home $
+navigateTo()
<<screen compartment>>
Log In Panel
+submitLogIn()
<<screen>>
Guest Home
<<screen>>
User Home
<<screen>>
Administrator Home
logOut()
<<screen compartment>>
Guest Menu
+signUp()
<<screen compartment>>
Search Panel
+advancedSearch()
+submitSearch()
<<screen compartment>>
User Menu
+ownProfile()
+myHelpRequest()
+conversations()
<<screen compartment>>
Log Out Panel
+logOut()
<<screen compartment>>
Administrator Menu
+addNewAbility()
+pendingAbilities()
OK
signUp()
<<input form>>
Search Bar
-Username
<<screen compartment>>
Notification Panel
+pendingFriendships()
+newReceivedMessages()
<<screen compartment>>
Wall +
+navigateTo()
+next()
+previous()
+helpRequest()
+postNewHelpRequest()
+filter()
User Preview
-Name
-Surname
-Avatar
<<screen>>
Sign Up
+submitUser()
<<input form>>
Sign Up Form
-Name
-Surname
-Avatar
-Email
-Repeated Email
-Password
-City
-Gender
-Birth Date
-Phone Number
ERROR
previous()
10
next()
Help Request Preview
-Subject
-Author
-Ability
-Date
-Hour

3.2. Messaging Functionality

This part of the UX Diagram allows to understand how the messaging functionalities are offered through the user experience.



From the *User Menu* we can have access to the pages related to personal conversations. This page contains only a preview of all the conversations (the recipient and some characters of the last message). Here we can both access to a specific conversation by clicking on it or send a new message through the *New Message* screen.
The first option allows to view the specific conversation (*Conversation* screen) and to send a new message to the user directly through an appropriate input form, also.
The second option, through a proper input form, allows to choose a recipient and send him a message.
Eventually, through the *Notification Panel*, clicking on a link, the user can see a screen showing a set of new received messages (*Received Messages* screen) with a proper preview.
We can also have access to the specific conversation clicking on the new incoming message.

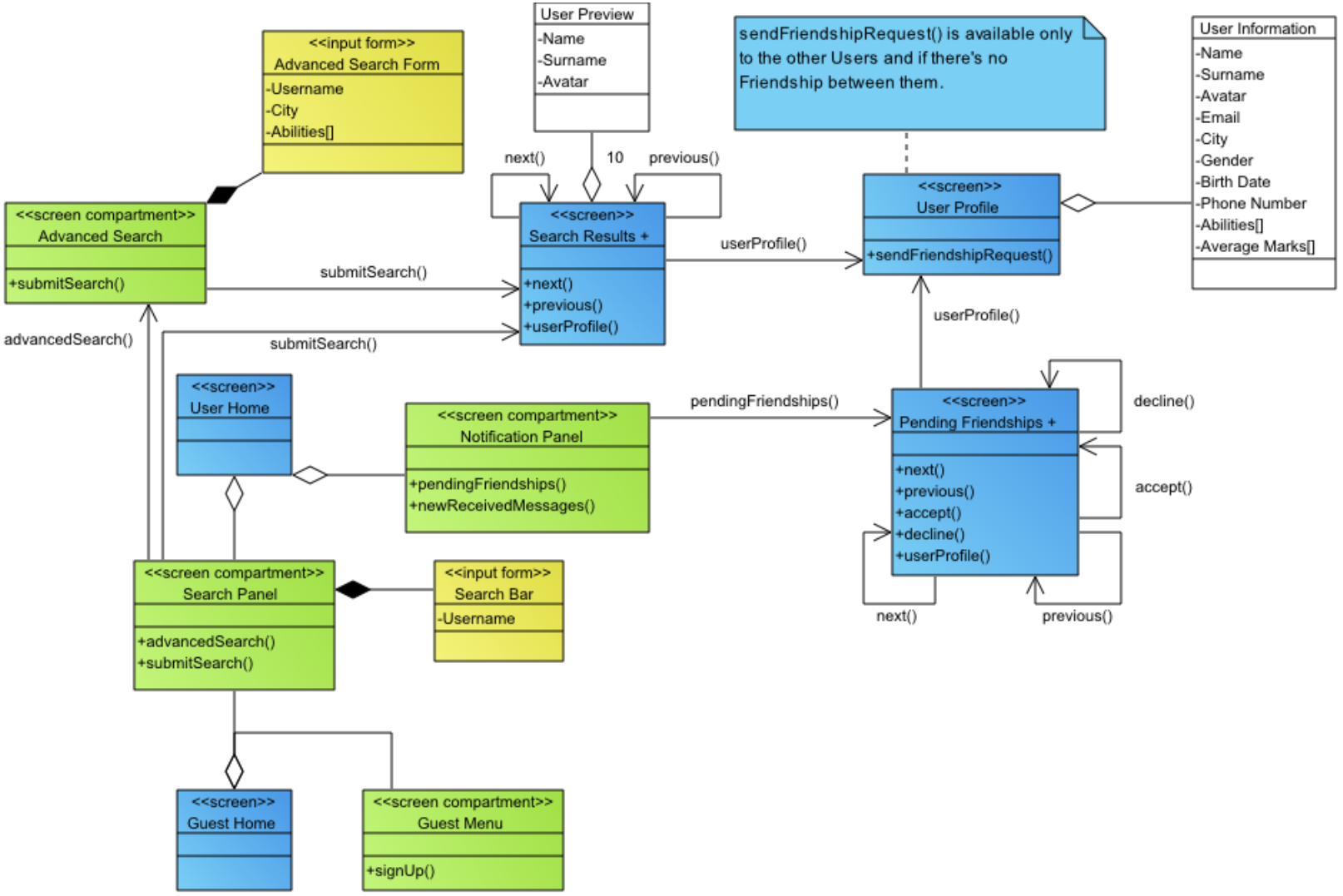
3.3. Profile Managing

This part of the UX Diagram allows to understand how the profile managing functionalities are offered through the user experience.

From the *User Menu* we can access our profile (*Own Profile* screen) and then, if we want to, modify it.
If we want to insert an ability which is not yet in the system we can do it through an appropriate screen (*Add New Ability* screen). We wanted to put out that this screen graphically the same either for a user or for an administrator, but the navigation is different, so we drew a generalization among *User Add New Ability* and *Administrator Add New Ability*, and *Add New Ability*.
Owing to the fact that if an user adds a new ability to the system an administrator have to confirm it, we have the *Pending Abilities* screen (reachable through the *Administrator Menu*) that allows the administrator to accept or decline the requests.
Every time that the administrator accepts or declines a new ability the page is reloaded with the result of the operation.

3.4. User Search and Friendship Management

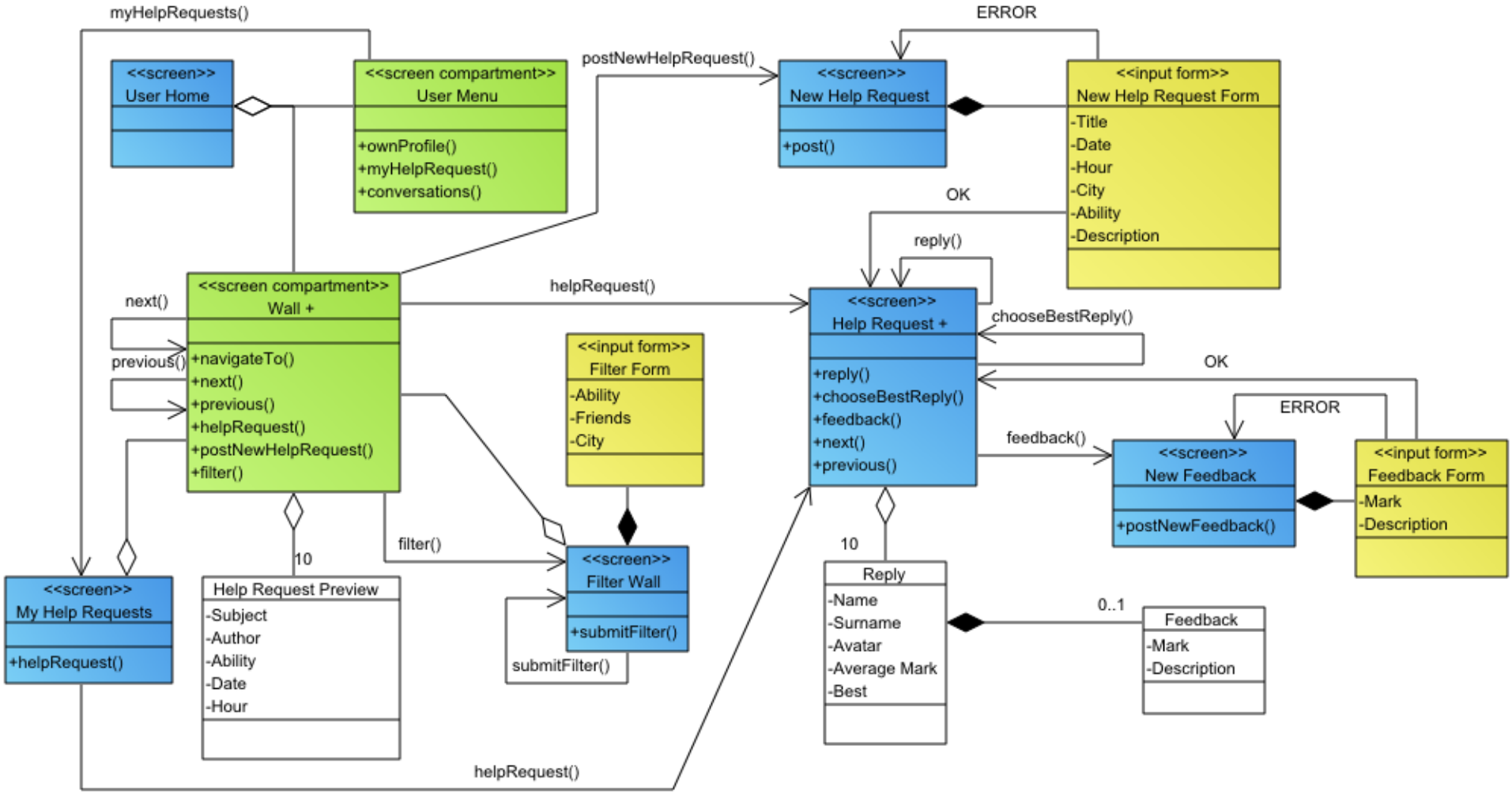
This part of the UX Diagram allows to understand how the user search and friendship management functionalities are offered through the user experience.



Search functionalities are can be performed through the *Search Bar* and the *Advanced Search Panel* (which is reachable through the *Search Bar* itself).
In modeling the User Experience in this way we thought that, once clicked on the *Advanced Search* link on the *Search Bar* the page would be reloaded with the new *Advanced Search* panel allowing this new advanced functionality.
Both from the input forms associated to the compartments we can submit parameters of research and send them to the system that will answer with the *Search Result* page, showing a set of *User Preview*s. Starting from here we can also click on the user's name and see its related profile.
Viewing the *User Profile* page we have the possibility to send a friendship request.
We have to remember that only users (not guests) that are not already friends with the target user can exploit this functionality.
We can also receive friendship requests, this event will be notified by the *Notification Panel*.
Through the appropriate link we can access to the *Pending Friendships* screen that shows us the set of pending friendships and, for each of them, it allows to accept it or decline it.

3.5. Help Request Management

This part of the UX Diagram allows to understand how the help request managing functionalities are offered through the user experience.



Starting from the *Wall* compartment we can submit a new help request with an input form and, if there is no error, we will be directed to the *Help Request* screen.
This screen gathers many functionalities in itself:

- A user can reply to the Help Request;
- The user that posted the help request can choose the best reply (from that moment on no one can reply no more to that help request).
- From the moment that a best reply is chosen by the user, he can also add a feedback to the best reply through the *New Feedback* screen.

As we want to explore the *Wall* it can be useful to have the possibility to filter it. To reach this aim we added the *Filter Wall* screen that contains an input form and, every time the data is submitted, in turn, it contains the filtered *Wall*.
If someone wants to reach quickly its (posted) help requests it can be possible for a user to reach them, through the *User Menu,* in *My Help Requests* screen.
In this screen it will be shown a set of help request preview and it can be possible to reach the specific help request through the *Help Request* screen.

4. BCE Diagrams

We decided to give a further design schema of SWIM using the Boundary-Control-Entity pattern, because it is very close to the Model-View-Controller pattern (in fact we can say that boundaries maps to the view, controls map to the controller and entities map to the model) and UML defines a standard to represent it.

It is important to know that boundaries are partially derived from the Use Cases Diagram provided in the RASD (so they don't need further explanations) and they gather some screen in the UX Diagram.

All the methods of the screens are written into the appropriate boundaries (maybe some names changed a bit only to make them more understandable in this context).

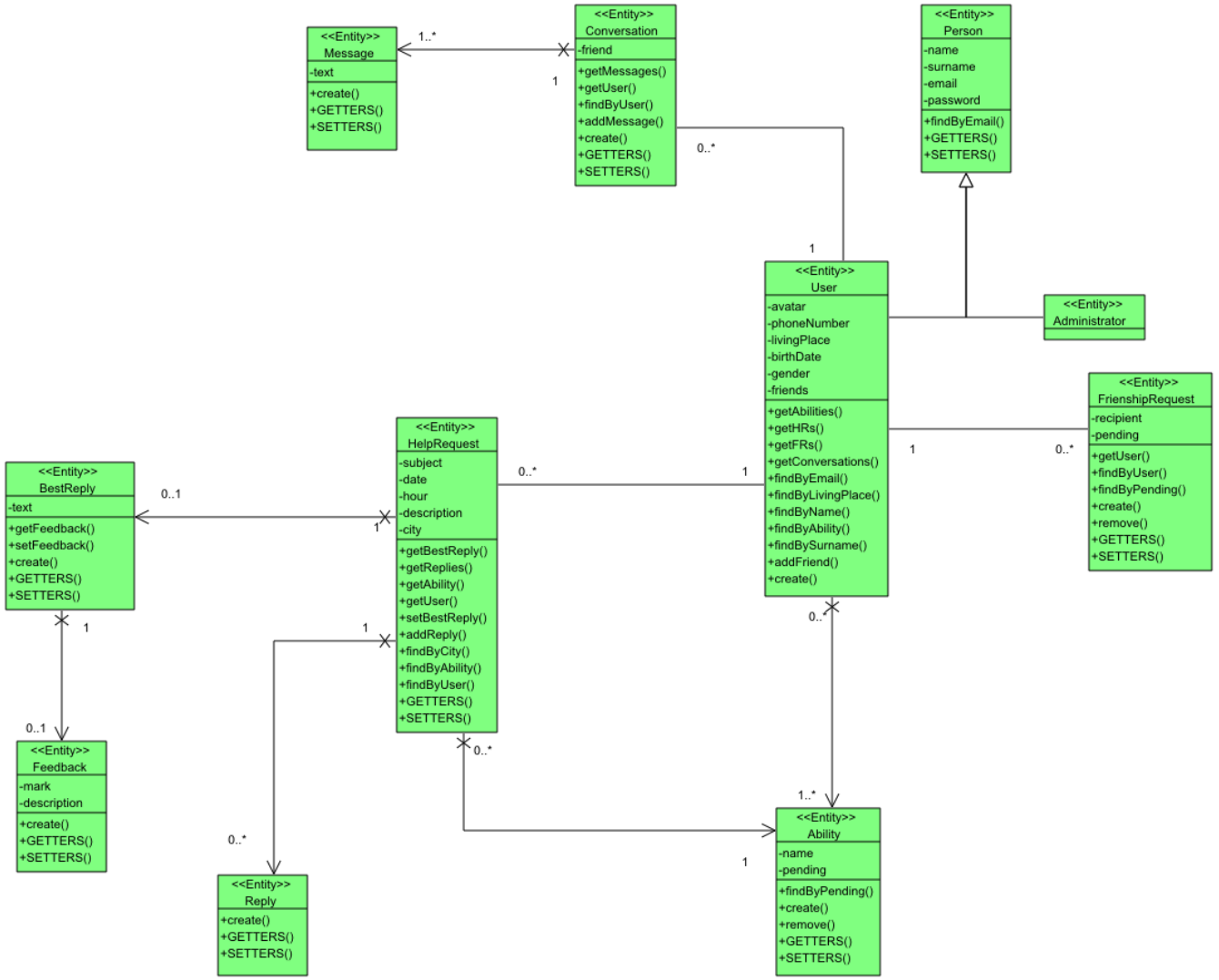
There is a method *showXXX()* for every screen in the UX Diagram. Screen compartment doesn't have a "show method" because we considered them as pieces of screens and not as standalone screens.

Finally it is important to say that entities do not represent the ER Diagram, but only a conceptual view of the entities used in the BCE.

We decided to separate the BCE diagram into the sub-systems of SWIM. In this way everything will be clearer.

4.1. Entity overview

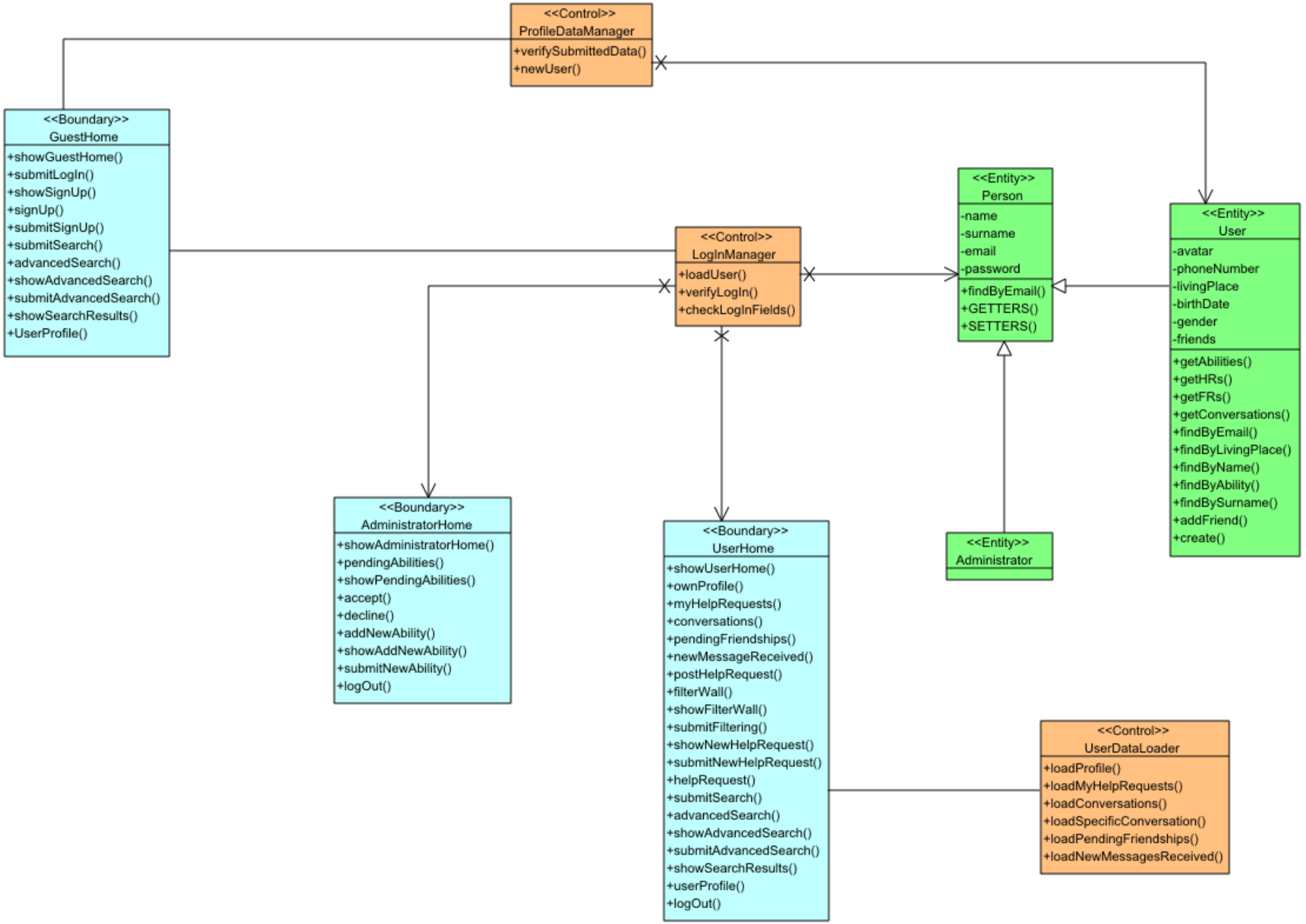
All entities of the BCE model will be presented in a very fragmented way, to prevent the diagrams to be too chaotic. For this reason we provide an entity overview, in the way that the reader can always refer to this diagram.



4.2. Sign up and Log in

The three boundaries *GuestHome*, *AdministratorHome* and *UserHome* represent the interface with generic users and the basic functionalities that they can reach at a first glance of the system (in other words, they represent the functionalities that a generic user can exploit in his home page).
We write down now a synthetic description of the controllers used:

- ***ProfileDataManager:*** this controller manages the verification of profile data in case of sign up or modification of profile information.
  It also creates a new user in case of signing up correctly.
- ***LogInManager:*** this controller manages the verification of log in fields. The logic is that firstly, it examines if the fields are filled correctly (we need *checkLogInFields()* because it might happen that a user fills the password field with only three characters, and we know that, for instance, our password has to be at least eight characters long. In this case there is no need to look for the user in the database, because we know that the log in will be incorrect in any case). Then the controller can load the correspondent user finding it by e-mail address and then check the password. This mechanism will be better examined later in the Sequence Diagrams section.


<<Control>>
ProfileDataManager
+verifySubmittedData()
+newUser()
<<Boundary>>
GuestHome
+showGuestHome()
+submitLogIn()
+showSignUp()
+signUp()
+submitSignUp()
+submitSearch()
+advancedSearch()
+showAdvancedSearch()
+submitAdvancedSearch()
+showSearchResults()
+UserProfile()
<<Control>>
LogInManager
+loadUser()
+verifyLogIn()
+checkLogInFields()
<<Entity>>
Person
-name
-surname
-email
-password
+findByEmail()
+GETTERS()
+SETTERS()
<<Entity>>
User
-avatar
-phoneNumber
-livingPlace
-birthDate
-gender
-friends
+getAbilities()
+getHRs()
+getFRs()
+getConversations()
+findByEmail()
+findByLivingPlace()
+findByName()
+findByAbility()
+findBySurname()
+addFriend()
+create()
<<Boundary>>
AdministratorHome
+showAdministratorHome()
+pendingAbilities()
+showPendingAbilities()
+accept()
+decline()
+addNewAbility()
+showAddNewAbility()
+submitNewAbility()
+logOut()
<<Boundary>>
UserHome
+showUserHome()
+ownProfile()
+myHelpRequests()
+conversations()
+pendingFriendships()
+newMessageReceived()
+postHelpRequest()
+filterWall()
+showFilterWall()
+submitFiltering()
+showNewHelpRequest()
+submitNewHelpRequest()
+helpRequest()
+submitSearch()
+advancedSearch()
+showAdvancedSearch()
+submitAdvancedSearch()
+showSearchResults()
+userProfile()
+logOut()
<<Entity>>
Administrator
<<Control>>
UserDataLoader
+loadProfile()
+loadMyHelpRequests()
+loadConversations()
+loadSpecificConversation()
+loadPendingFriendships()
+loadNewMessagesReceived()

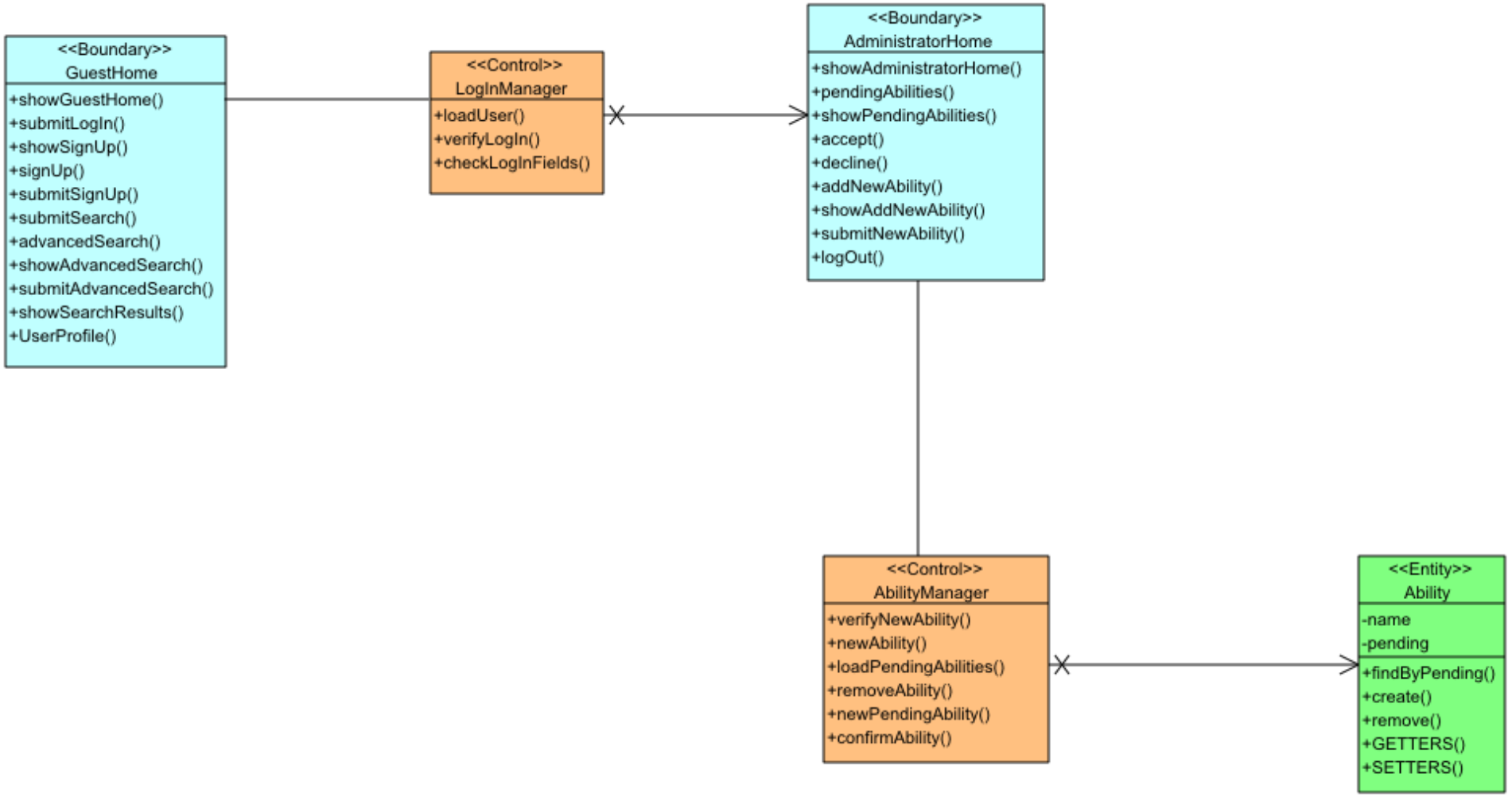
4.3. Administrator

In this diagram the *GuestHome* boundary and the *LogInManager* controller are repeated for a better comprehension.

In this diagram the *AbilityManager* controller appears.

This controller is shared between the administrator's and the user's features, as concerning the administrator its functionalities are:

- Loading pending abilities;
- Confirming pending abilities, in case the administrator confirms an ability request (sent by an user);
- Removing abilities, in case the administrator refuses an ability request;
- Creating a new ability;
- Verifying a new ability, in fact the system, before storing a new ability, will check if the same ability already exists into the system and will notify the administrator.


<<Boundary>>
GuestHome
+showGuestHome()
+submitLogIn()
+showSignUp()
+signUp()
+submitSignUp()
+submitSearch()
+advancedSearch()
+showAdvancedSearch()
+submitAdvancedSearch()
+showSearchResults()
+UserProfile()
<<Control>>
LogInManager
+loadUser()
+verifyLogIn()
+checkLogInFields()
<<Boundary>>
AdministratorHome
+showAdministratorHome()
+pendingAbilities()
+showPendingAbilities()
+accept()
+decline()
+addNewAbility()
+showAddNewAbility()
+submitNewAbility()
+logOut()
<<Control>>
AbilityManager
+verifyNewAbility()
+newAbility()
+loadPendingAbilities()
+removeAbility()
+newPendingAbility()
+confirmAbility()
<<Entity>>
Ability
-name
-pending
+findByPending()
+create()
+remove()
+GETTERS()
+SETTERS()

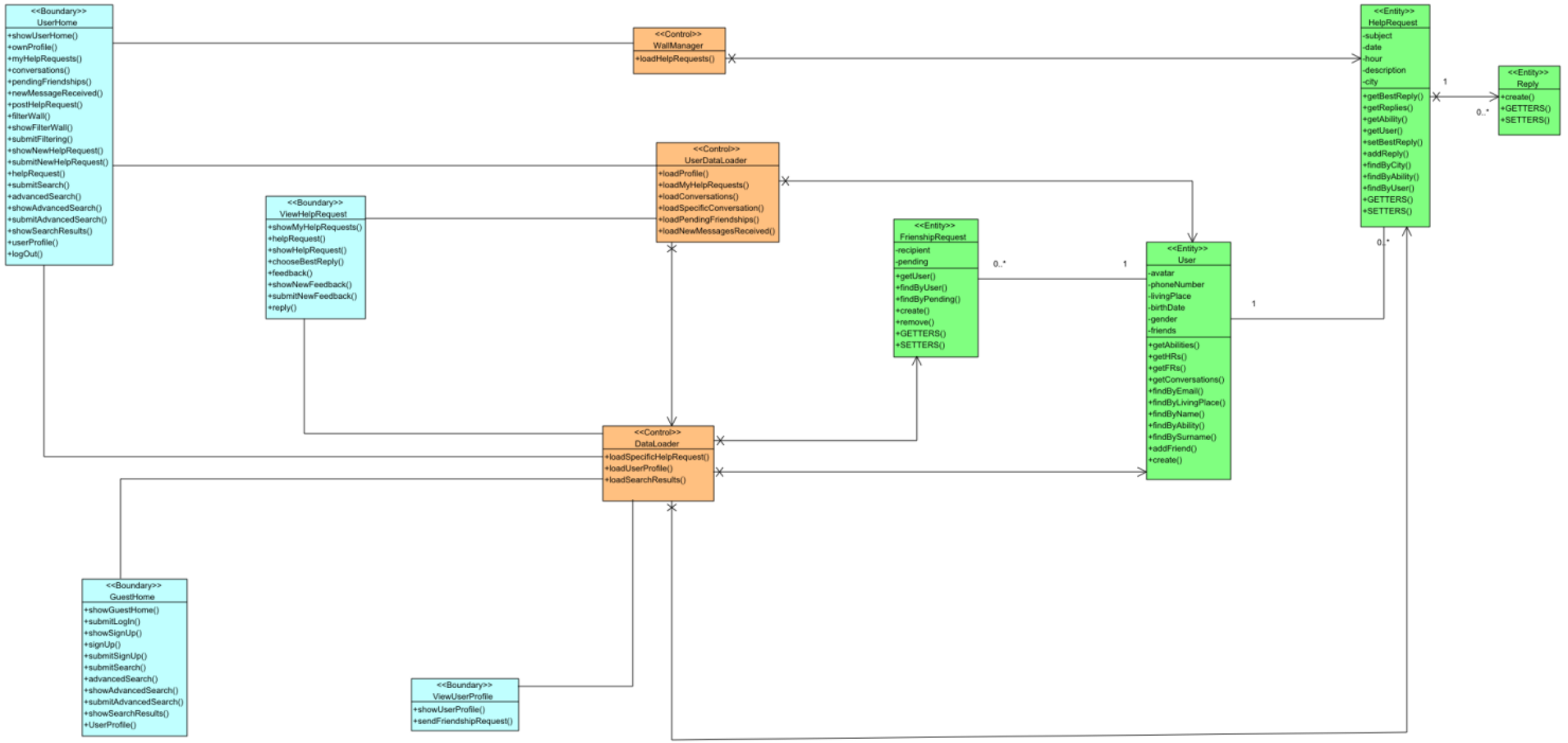
4.4. User

In this paragraph we will show all modules that are in the *User* sub-system (with one sharing exception).

4.4.1. Search

Searching functionalities are shared between the user and the guest.
The only feature that cannot be exploited by the guest is into the *ViewUserProfile* boundary and it is *sendFriendshipRequest()*. In fact only a user can send a friendship request to another user.
The controllers are:

- ***WallManager:*** the one that has to load all right help requests in case that a user filters the wall.
- ***UserDataLoader:*** the loader of personal data about the user. The reader will notice that the only link that this controller has with entities is with the User entity. He manages the notification panel of the user. In fact it loads personal help requests, pending friendship requests, new messages received and personal conversations.
- ***DataLoader:*** this loader loads information that are not dependent from the user. In fact it loads generic help requests, other users' profile and search results (with "search results" we mean the results of a search performed to find users, not a wall filtering).

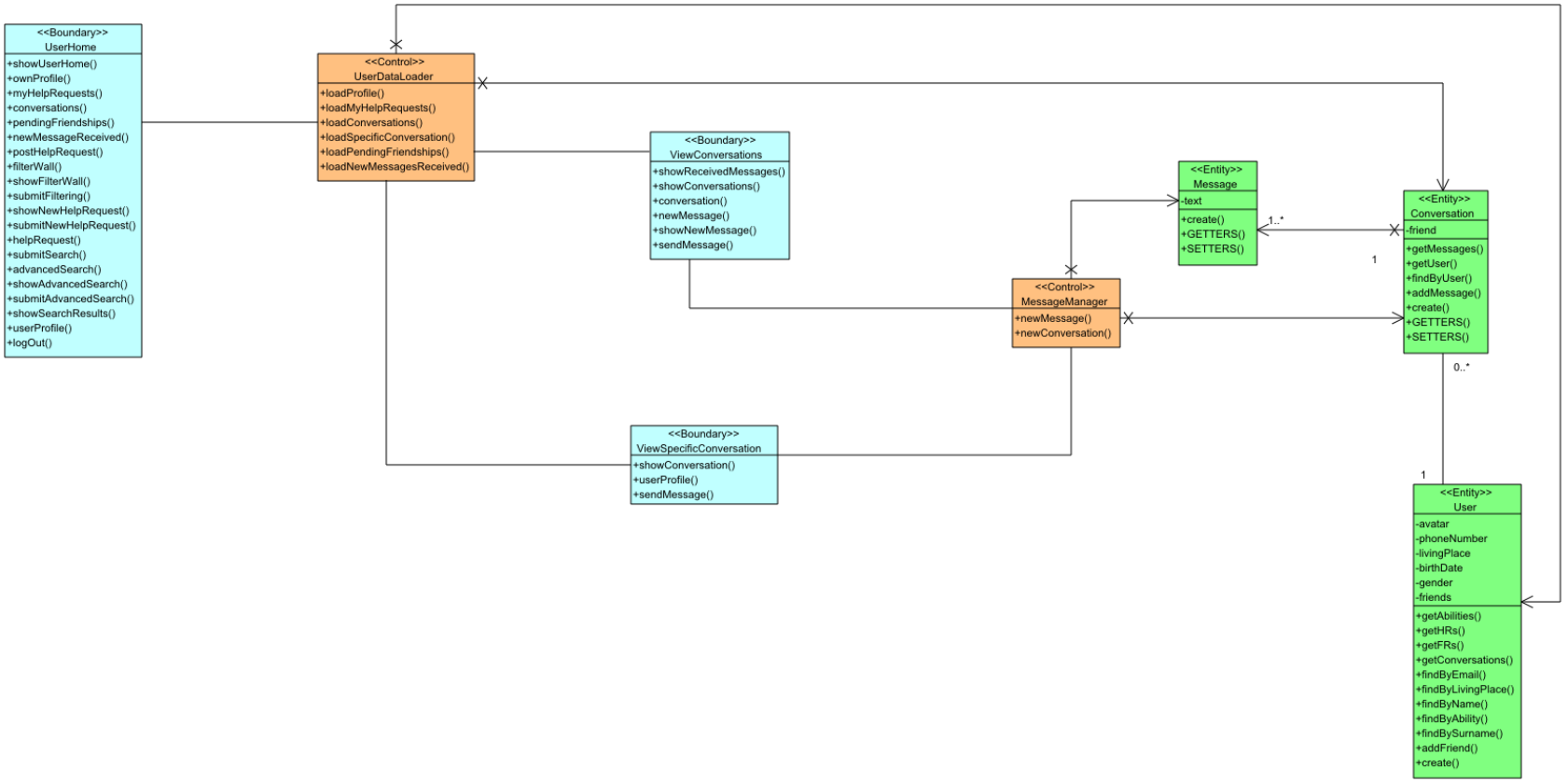

<<Boundary>>
UserHome
+showUserHome()
+ownProfile()
+myHelpRequests()
+conversations()
+pendingFriendships()
+newMessageReceived()
+postHelpRequest()
+filterWall()
+showFilterWall()
+submitFiltering()
+showNewHelpRequest()
+submitNewHelpRequest()
+helpRequest()
+submitSearch()
+advancedSearch()
+showAdvancedSearch()
+submitAdvancedSearch()
+showSearchResults()
+userProfile()
+logOut()
<<Control>>
WallManager
+loadHelpRequests()
<<Entity>>
HelpRequest
-subject
-date
-hour
-description
-city
+getBestReply()
+getReplies()
+getAbility()
+getUser()
+setBestReply()
+addReply()
+findByCity()
+findByAbility()
+findByUser()
+GETTERS()
+SETTERS()
1
0..*
<<Entity>>
Reply
+create()
+GETTERS()
+SETTERS()
<<Control>>
UserDataLoader
+loadProfile()
+loadMyHelpRequests()
+loadConversations()
+loadSpecificConversation()
+loadPendingFriendships()
+loadNewMessagesReceived()
<<Boundary>>
ViewHelpRequest
+showMyHelpRequests()
+helpRequest()
+showHelpRequest()
+chooseBestReply()
+feedback()
+showNewFeedback()
+submitNewFeedback()
+reply()
<<Entity>>
FrienshipRequest
-recipient
-pending
+getUser()
+findByUser()
+findByPending()
+create()
+remove()
+GETTERS()
+SETTERS()
0..*
1
<<Entity>>
User
-avatar
-phoneNumber
-livingPlace
-birthDate
-gender
-friends
+getAbilities()
+getHRs()
+getFRs()
+getConversations()
+findByEmail()
+findByLivingPlace()
+findByName()
+findByAbility()
+findBySurname()
+addFriend()
+create()
1
0..*
<<Control>>
DataLoader
+loadSpecificHelpRequest()
+loadUserProfile()
+loadSearchResults()
<<Boundary>>
GuestHome
+showGuestHome()
+submitLogIn()
+showSignUp()
+signUp()
+submitSignUp()
+submitSearch()
+advancedSearch()
+showAdvancedSearch()
+submitAdvancedSearch()
+showSearchResults()
+UserProfile()
<<Boundary>>
ViewUserProfile
+showUserProfile()
+sendFriendshipRequest()

4.4.2. Conversations management

We write down the functionalities managed by the *MessageManager:*

- Creating a new message;
- Creating a new conversation.

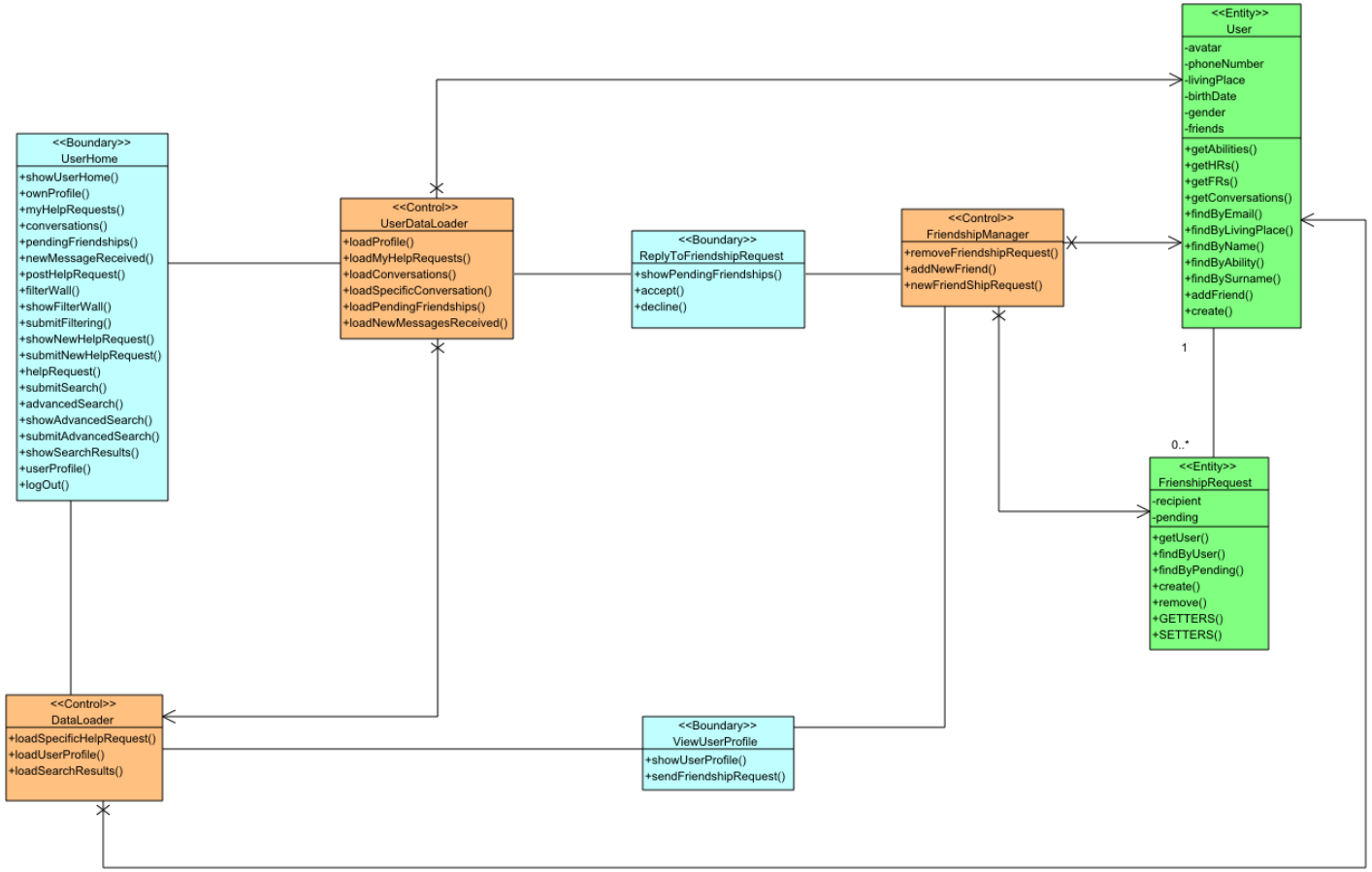
A new conversation is created in case that the user sends for the first time a message to another user (this details are specified clearly into the Sequence Diagrams section).



4.4.3. Friendship management

The *FriendShipManager*:

- Removes a friendship request in case that the recipient declines it;
- Adds a new friend to the user that sent the friendship request in case that the recipient accepts it;
- Creates a new friendship request.



4.4.4. Help requests management

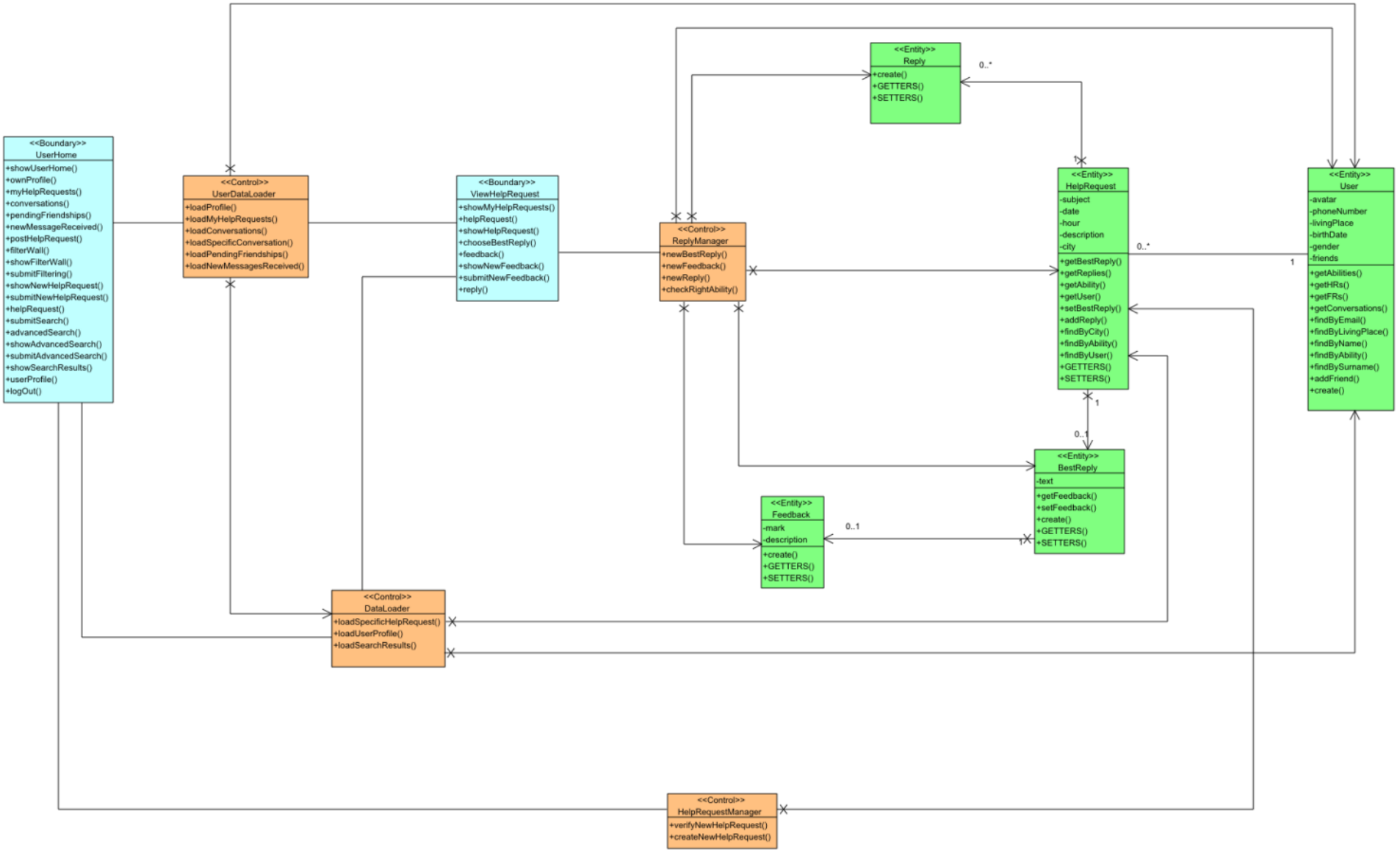
The *ReplyManager*:

- Creates a new reply;
- Creates a new best reply;
- Creates a new feedback;
- Checks if the user that is replying to a help request has the right ability to do it.

Replies, best replies and feedbacks are put in relation with the right entities using the entities' methods (see Sequence Diagram section).

The *HelpRequestManager* is used in case that a user posts a new help request, in fact it:

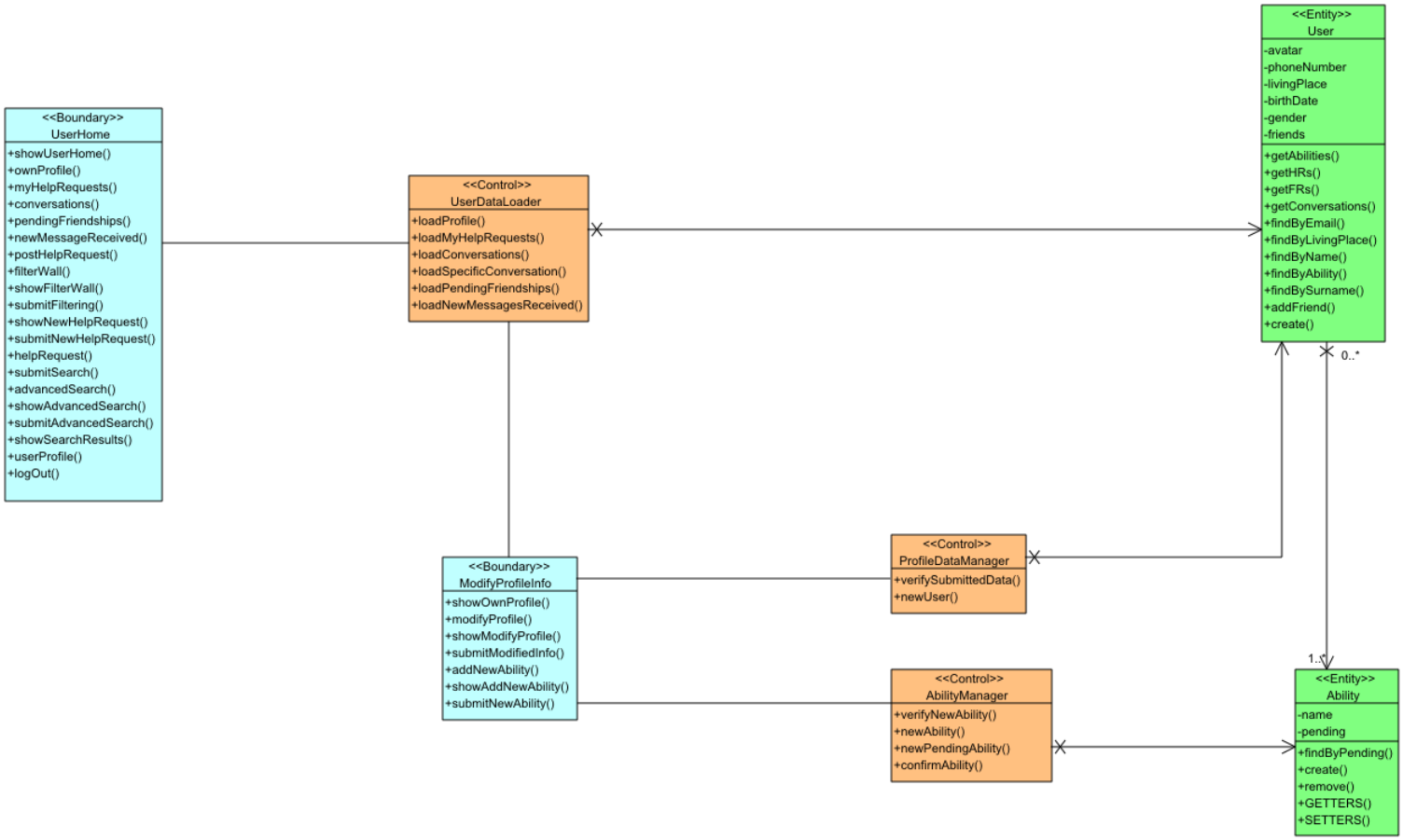
- Checks if the new help request has been fulfilled properly;
- Creates a new help request if the check ended in a positive way.


<<Boundary>>
UserHome
+showUserHome()
+ownProfile()
+myHelpRequests()
+conversations()
+pendingFriendships()
+newMessageReceived()
+postHelpRequest()
+filterWall()
+showFilterWall()
+submitFiltering()
+showNewHelpRequest()
+submitNewHelpRequest()
+helpRequest()
+submitSearch()
+advancedSearch()
+showAdvancedSearch()
+submitAdvancedSearch()
+showSearchResults()
+userProfile()
+logOut()
<<Control>>
UserDataLoader
+loadProfile()
+loadMyHelpRequests()
+loadConversations()
+loadSpecificConversation()
+loadPendingFriendships()
+loadNewMessagesReceived()
<<Boundary>>
ViewHelpRequest
+showMyHelpRequests()
+helpRequest()
+showHelpRequest()
+chooseBestReply()
+feedback()
+showNewFeedback()
+submitNewFeedback()
+reply()
<<Entity>>
Reply
+create()
+GETTERS()
+SETTERS()
0..*
<<Control>>
ReplyManager
+newBestReply()
+newFeedback()
+newReply()
+checkRightAbility()
<<Entity>>
HelpRequest
-subject
-date
-hour
-description
-city
+getBestReply()
+getReplies()
+getAbility()
+getUser()
+setBestReply()
+addReply()
+findByCity()
+findByAbility()
+findByUser()
+GETTERS()
+SETTERS()
1
0..*
<<Entity>>
User
-avatar
-phoneNumber
-livingPlace
-birthDate
-gender
-friends
+getAbilities()
+getHRs()
+getFRs()
+getConversations()
+findByEmail()
+findByLivingPlace()
+findByName()
+findByAbility()
+findBySurname()
+addFriend()
+create()
1
0..1
<<Entity>>
BestReply
-text
+getFeedback()
+setFeedback()
+create()
+GETTERS()
+SETTERS()
<<Entity>>
Feedback
-mark
-description
+create()
+GETTERS()
+SETTERS()
0..1
1
<<Control>>
DataLoader
+loadSpecificHelpRequest()
+loadUserProfile()
+loadSearchResults()
<<Control>>
HelpRequestManager
+verifyNewHelpRequest()
+createNewHelpRequest()

4.4.5. Profile management

In this case the *ProfileDataManager* and the *AbilityManager* is used in the case of a modification of user's profile. In fact, the only methods used into the *AbilityManager* are:

- *verifyNewAbility()*, used for the same reason ad in the administrator's case;
- *newPendingAbility()*, used in the case that a user adds an ability which has not already inserted into the system.



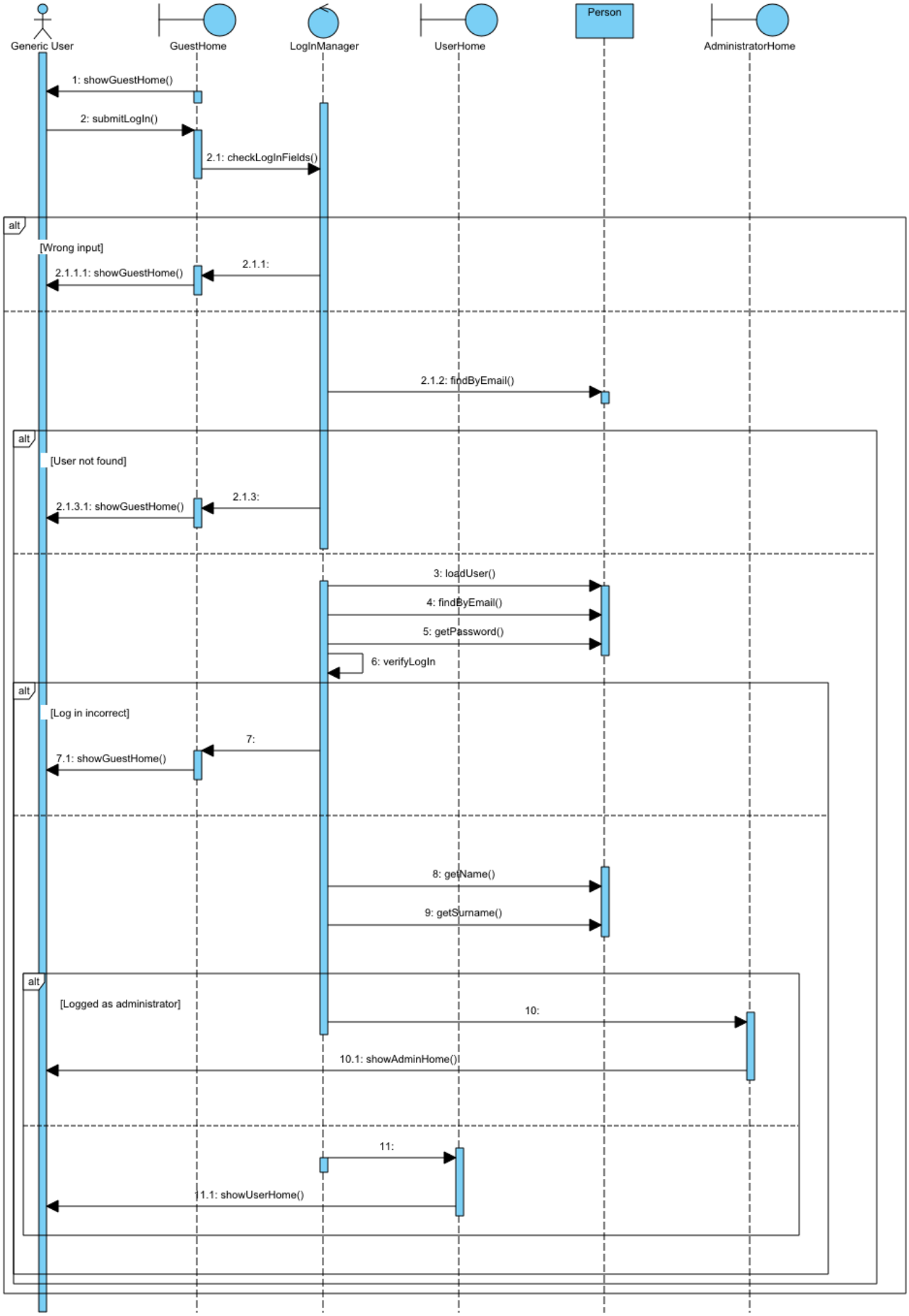
5. Sequence diagrams

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.

5.1. Log In

A generic user:

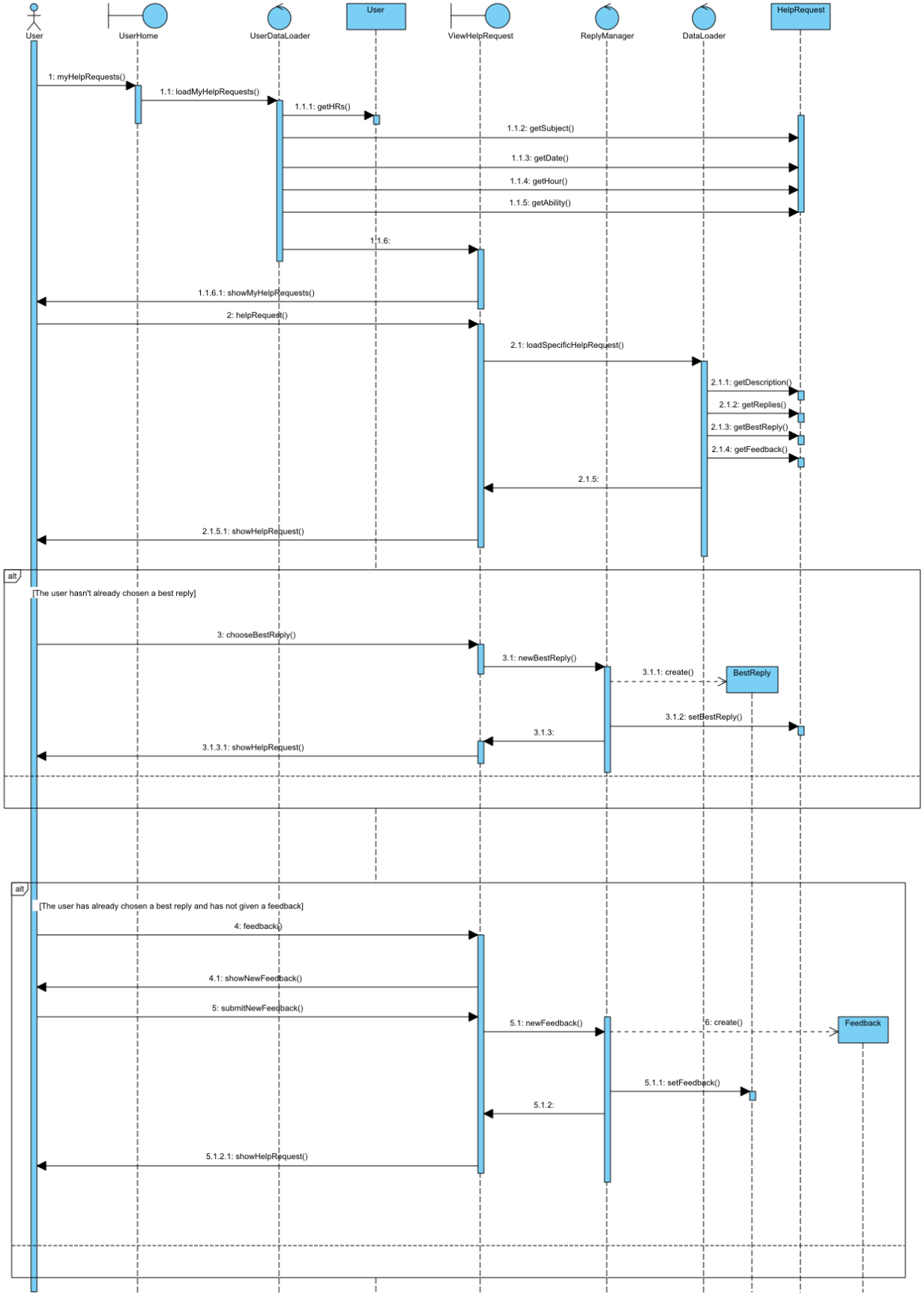
- Logs in.


Generic User
GuestHome
LogInManager
UserHome
Person
AdministratorHome
1: showGuestHome()
2: submitLogIn()
2.1: checkLogInFields()
alt
[Wrong input]
2.1.1:
2.1.1.1: showGuestHome()
2.1.2: findByEmail()
alt
[User not found]
2.1.3:
2.1.3.1: showGuestHome()
3: loadUser()
4: findByEmail()
5: getPassword()
6: verifyLogIn
alt
[Log in incorrect]
7:
7.1: showGuestHome()
8: getName()
9: getSurname()
alt
[Logged as administrator]
10:
10.1: showAdminHome()
11:
11.1: showUserHome()

5.2. My help requests

An user:

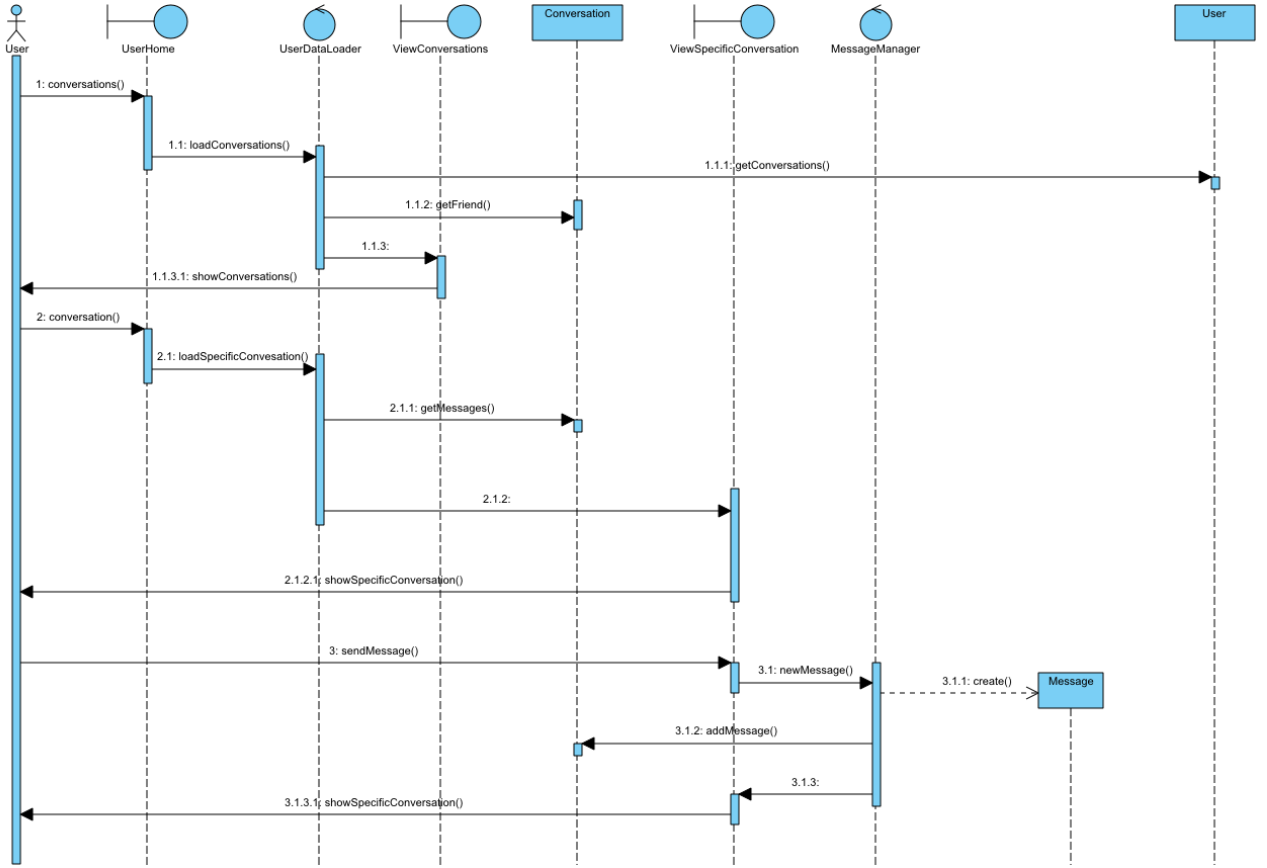
- Accesses to his help requests;
- Selects a specific help request;
- (optionally) chooses a best reply;
- (optionally) gives a feedback.


User
UserHome
UserDataLoader
User
ViewHelpRequest
ReplyManager
DataLoader
HelpRequest
1: myHelpRequests()
1.1: loadMyHelpRequests()
1.1.1: getHRs()
1.1.2: getSubject()
1.1.3: getDate()
1.1.4: getHour()
1.1.5: getAbility()
1.1.6:
1.1.6.1: showMyHelpRequests()
2: helpRequest()
2.1: loadSpecificHelpRequest()
2.1.1: getDescription()
2.1.2: getReplies()
2.1.3: getBestReply()
2.1.4: getFeedback()
2.1.5:
2.1.5.1: showHelpRequest()
alt
[The user hasn't already chosen a best reply]
3: chooseBestReply()
3.1: newBestReply()
3.1.1: create()
BestReply
3.1.2: setBestReply()
3.1.3:
3.1.3.1: showHelpRequest()
alt
[The user has already chosen a best reply and has not given a feedback]
4: feedback()
4.1: showNewFeedback()
5: submitNewFeedback()
5.1: newFeedback()
6: create()
Feedback
5.1.1: setFeedback()
5.1.2:
5.1.2.1: showHelpRequest()

5.3. Send a message starting from a specific conversation

An user:

- Accesses to his conversations;
- Accesses to a specific conversation;
- Sends a new message.

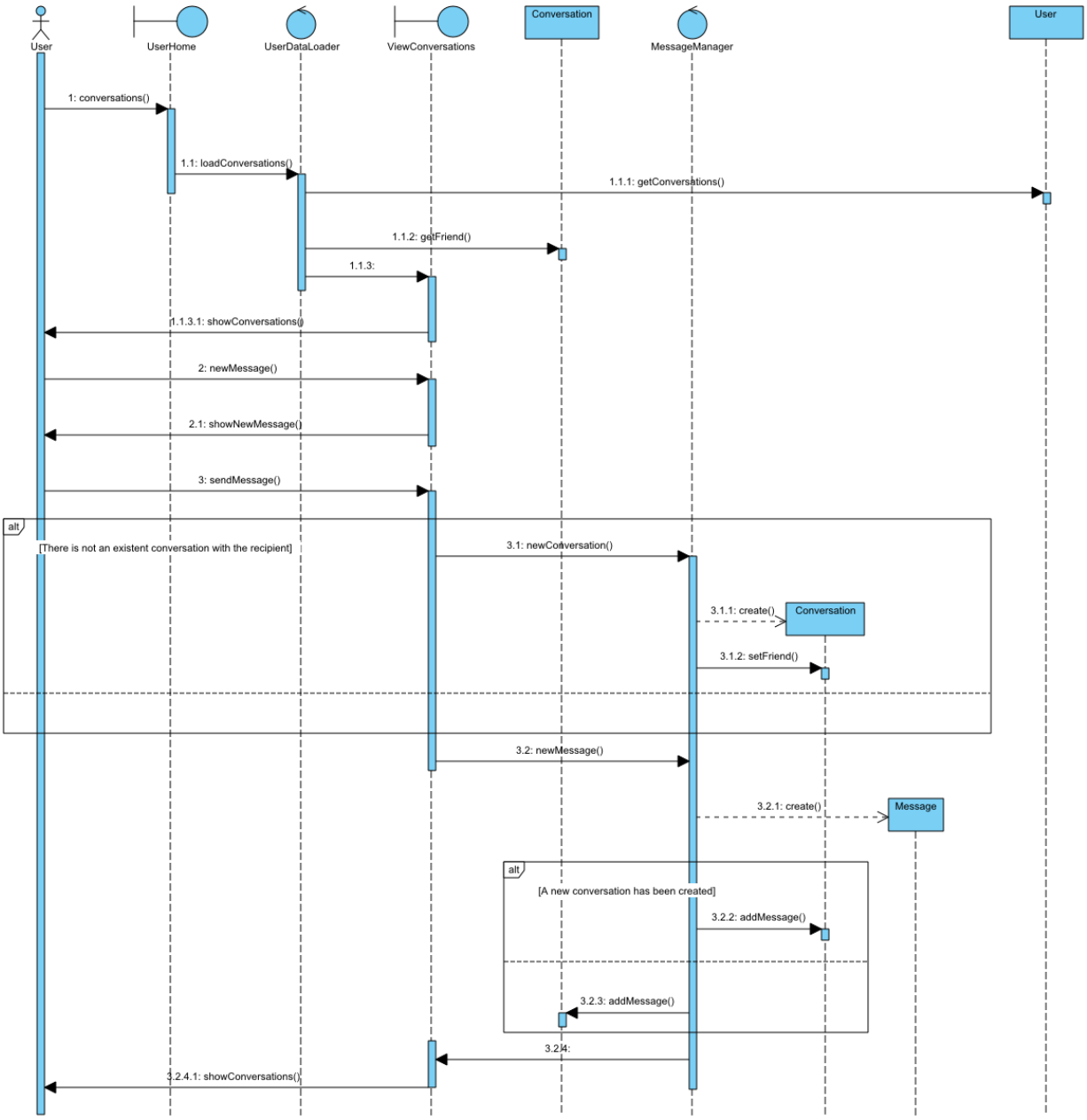


5.4. Send a message starting from conversations

An user:

- Accesses to his conversations;
- Sends a new message.

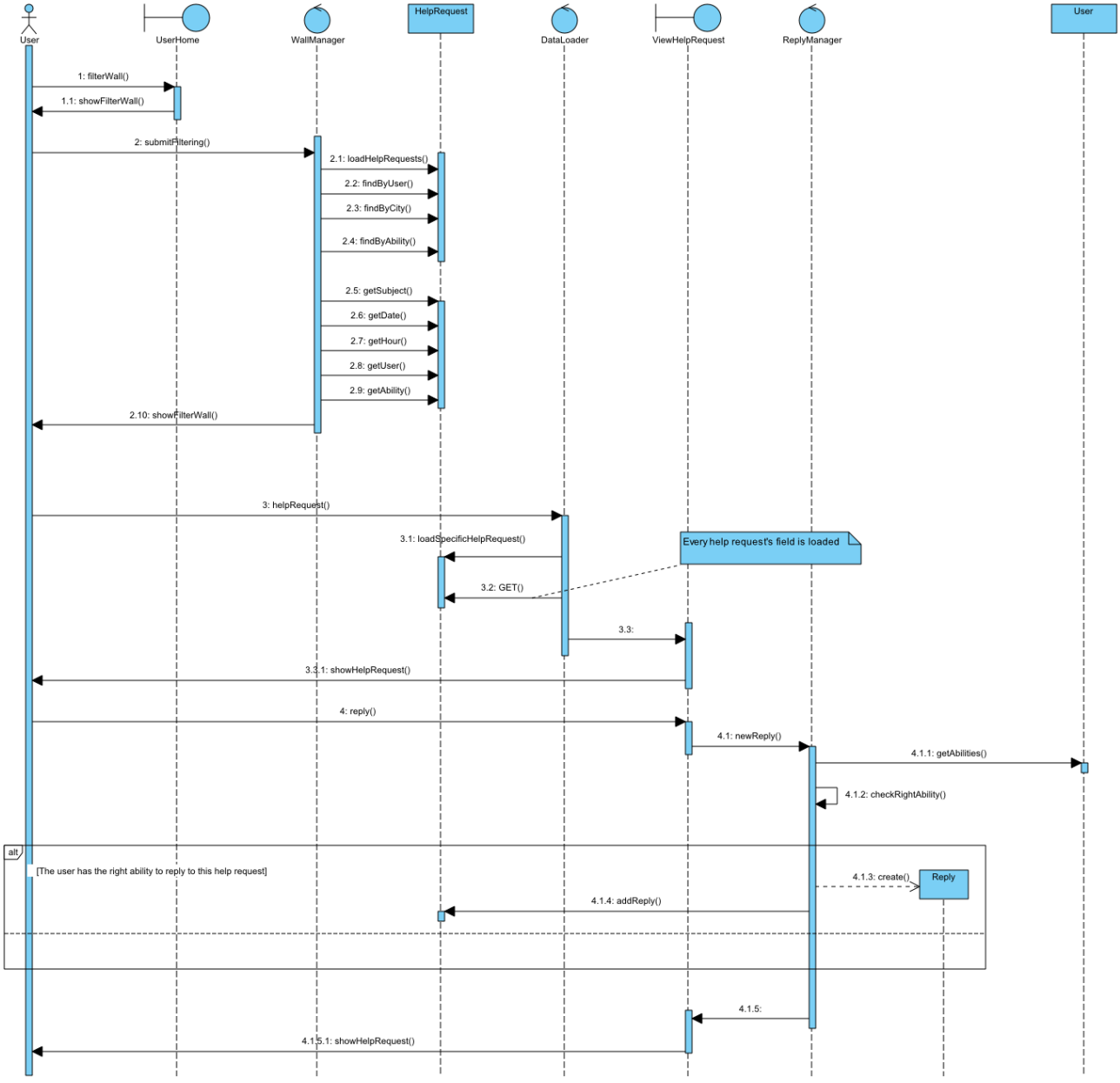
In this case a new conversation is created if the message sent to the recipient is the first one.



5.5. Browse help requests

An user:

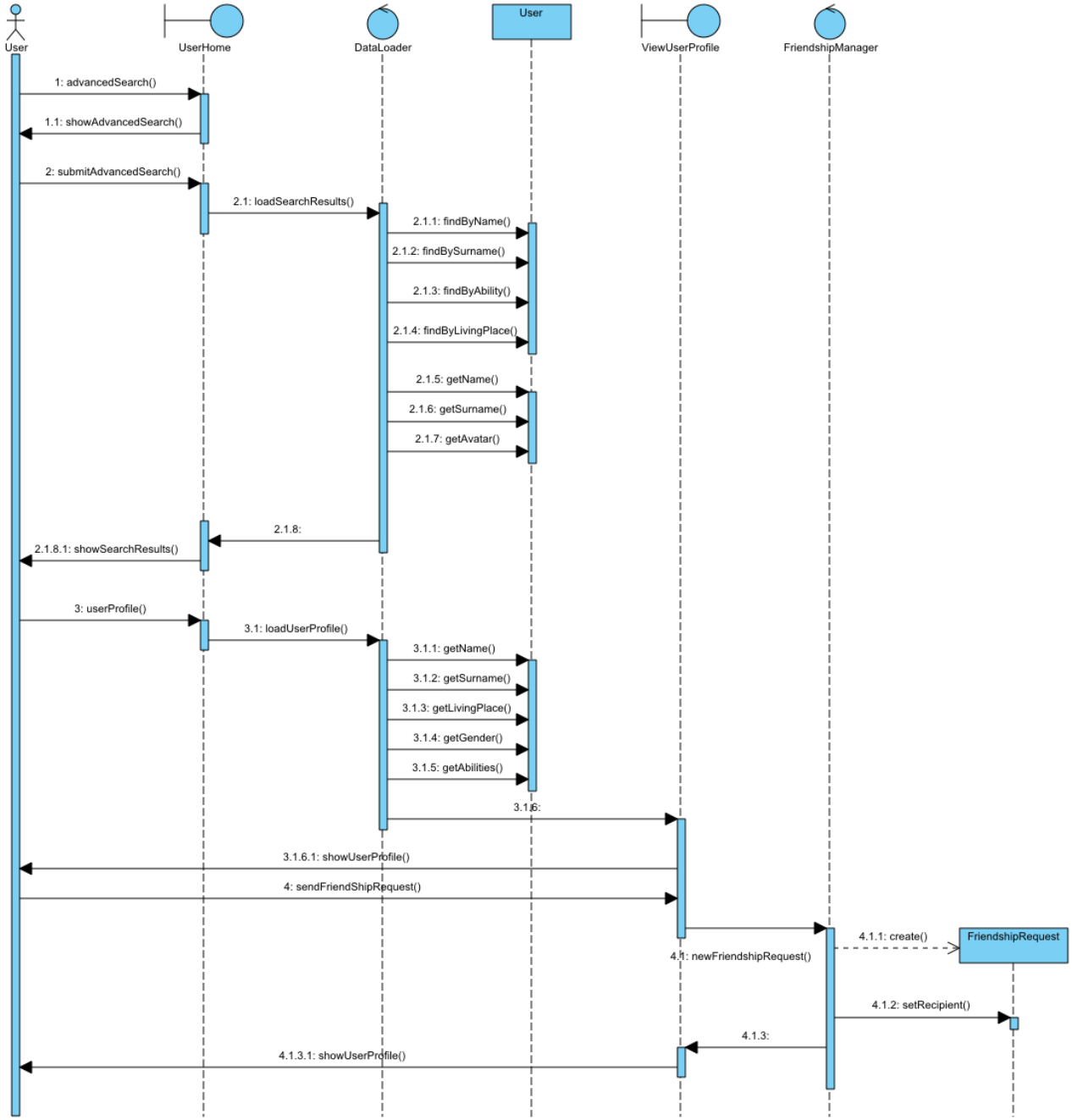
- Is in his home page, so he views the wall;
- Clicks on a help request;
- Replies to it.



5.6. Search for a user

An user:

- Performs an advanced search to find a user;
- Clicks on the user's name and views his profile;
- Sends a new friendship request.



6. Final Considerations

We decided not to draw any detailed diagram, because we think that a standard detailed diagram (with *Server Page*, *Client Page*, *HTML Form* and *Control* stereotypes) wouldn't have added meaning to our Design Document.
In fact, with this diagram, we only had to had *Server Page*s if *Client Page*s (the same thing as *Screen*s in the UX Diagram) are built dynamically, and we know that almost a large part of our pages will be dynamic.
Moreover, it is not clear if *Server Page*s and *Control*s represent directly *Servlet*s or *Bean*s.
For this reason we think that the standard detailed diagram couldn't bring us to a more specific knowledge of the implementation of our project.
Eventually, we drew UX Diagrams and BCE Diagrams instead, that are diagrams very detached from the architecture that lays under the project, but we decided not to draw more specific diagrams (such as *Deployment View* and *Run-Time View*), because we don't know so much *JEE* architecture to go into details.
We know, in fact, that from now on we have to take a very big effort to understand the architecture well and to start implementing our project.