POLITECNICO DI MILANO

Distributed C4C

Design Document

Written by:

Sara Magliacane 735968
Francesco Pongetti 735339

Milan, 20th of December, 2008

TABLE OF CONTENTS

ABSTRACT

This document describes the design process of the DC4C application.
We describe guidelines for the development and deployment phases.

APPLICATION ARCHITETURE

1.1 Overview

We decided to develop our system as a JEE 5 based application. The JEE 5 platform uses a distributed multi-tiered application model for enterprise applications. Application logic is divided into components according to function, and the various application components are installed on different machines.

Our system is divided in three tiers:

- **Client Tier:** the tier responsible for the interaction with the users; in our system we decided to implement an application client written in java

- **Business Tier:** the data and the business logic of the application; it is composed by Enterprise Java Beans (EJB) that represent persistent data (entity beans) and business operations (session beans).

- **Data Tier:** the relational database management system in which the date is stored

In the following pages we will start describing the system from the lowest level, the Data Tier, up to the higher ones, the Business Tier and the Client Tier.

The application structure is reassumed here:

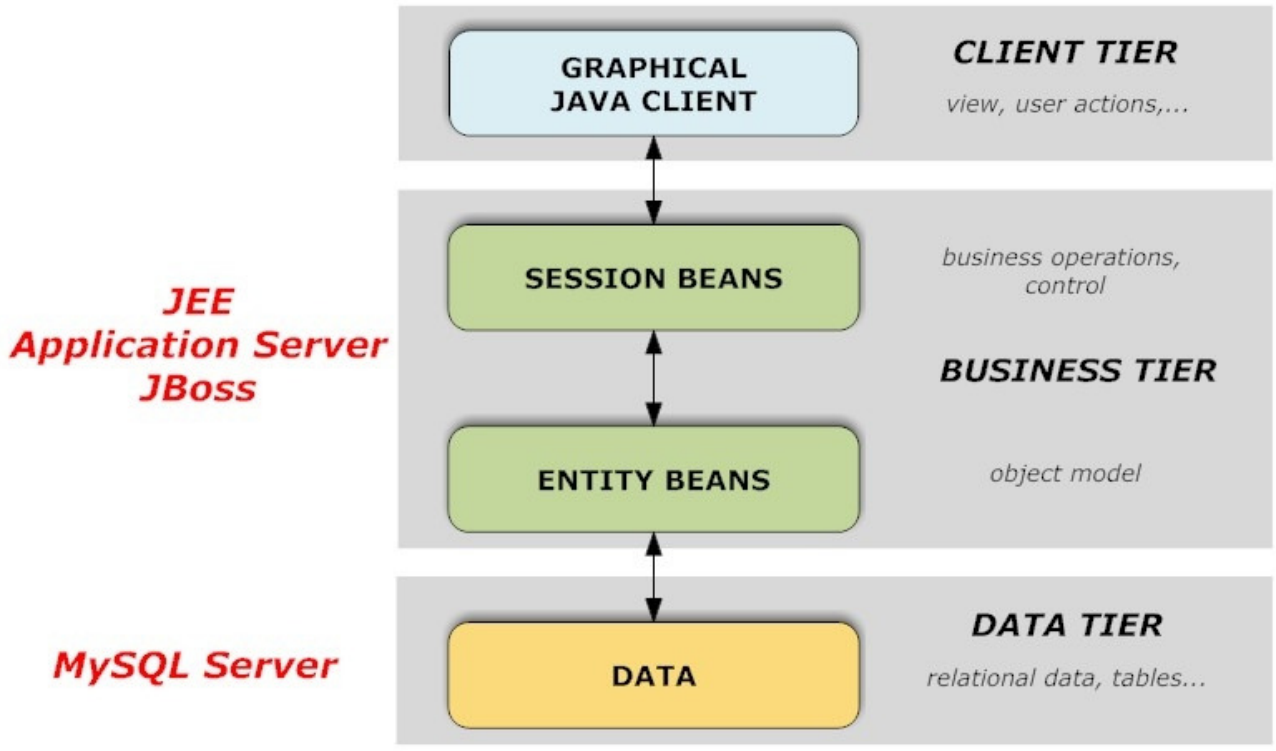


Figure 1. Application architeture diagram

DATA TIER

2.1 Introduction

We started designing the relational database that contains all the data required by our application. First we built a conceptual model of the DB using an entity-relationship diagram, after that we refined the model translating it into tables. All the information used are taken from the analysis performed in the RASD phase, especially from the conceptual class diagram and the assumptions.

2.1 Entity-Relationship diagram

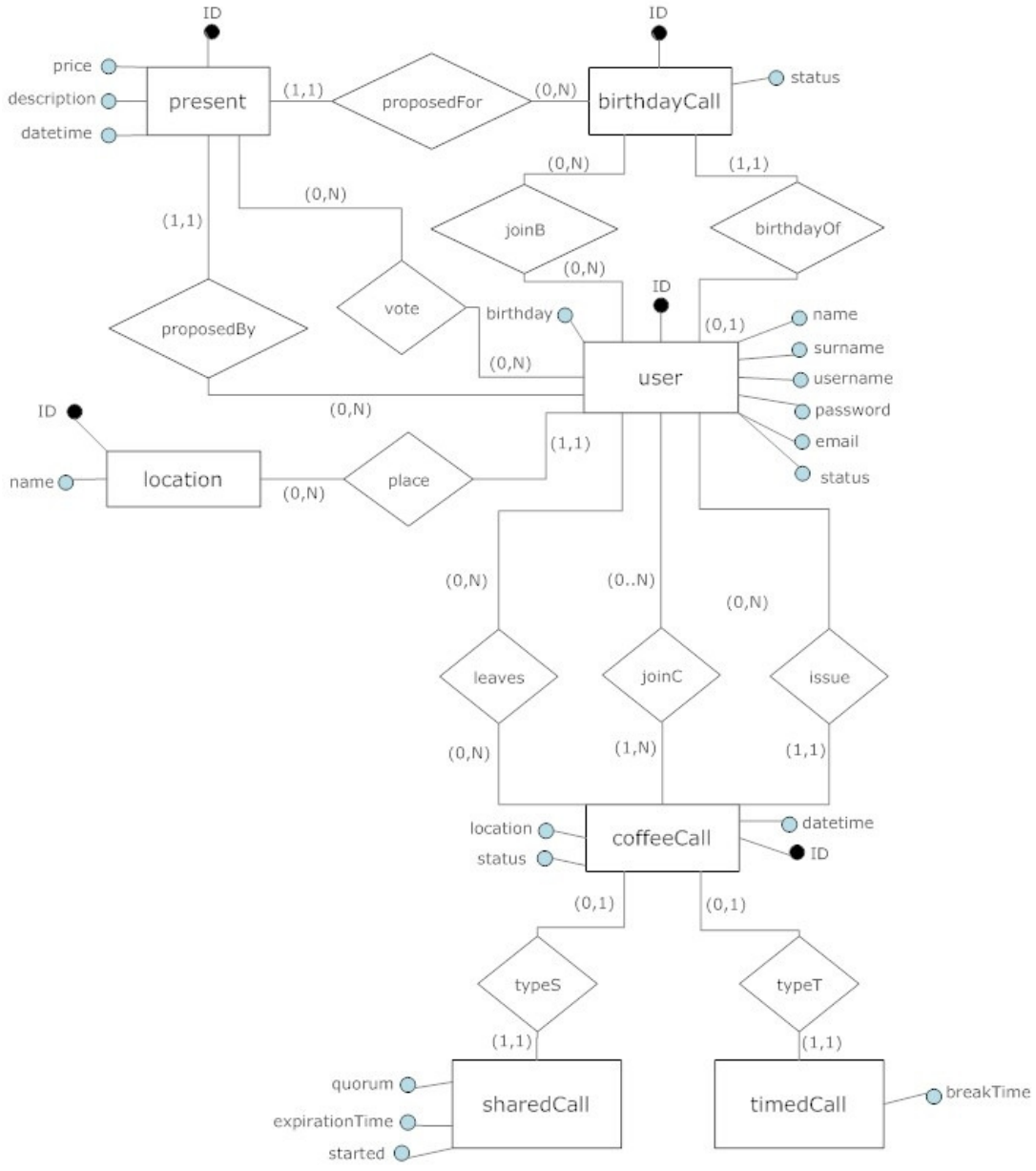


Figure 2. Entity-Relationship Diagram

2.1.1 Entities

user: registered user that uses the DC4C system
- ID: an unique numeric identifier generated automatically by the system for each user
- name: the real name of the user
- surname: the real surname of the user
- username: an username for logging in the system, specified by the user during the registration
- password: a password for logging in the system, specified by the user during registration
- email: a valid email that the user has to provide during the registration
- birthday: user's birthday
- status: specifies the status of the user: 0 for working, 1 for busy, 2 gone to take a coffee

location: the places where you can have a coffee.
- ID: an unique numeric identifier generated automatically by the system for each location
- name: name of the location

birthdayCall: a "call for birthday present" automatically generated by the system one week before the birth date
- ID: an unique numeric identifier generated automatically by the system for each birthday call
- status: a Boolean value. False for closed call, true for open call

present: all the present proposed by the users
- ID: an unique numeric identifier generated automatically by the system for each present
- price: present's price
- description: present's description
- datetime: date and time when the present was proposed on

coffeeCall: a "call for coffee" that can be issued by any user; it is a generalization of a timed and shared call
- ID: an unique numeric identifier generated automatically by the system for each coffee call
- location: the place where the call will be hold
- status: a Boolean value. False for closed call, true for open call
- datetime: date and time when the call was issued
-

timedCall: a coffee call that is triggered on a specific time, no matter how many users join it
- breakTime: what time the call will be hold

sharedCall: a coffee call that is triggered when a threshold of users have joined it.
- quorum: the minimum number of people that have to join to start the call
- expirationTime: date and time until while the system will wait for people joining the shared call
- started: date and time the shared call started

2.1.2 Relationships

proposedFor: each present is proposed for a specific birthday call

proposedBy: each present is proposed by an user

vote: each present can be voted by some users

birthdayOf: each birthday call is generated for an user's birthday

joinB: each user can join to a birthday call

issue: each user can issue a coffee call

joinC: each user can join to coffee call

leaves: an user can leave a joined coffee call

typeS: a coffee call can be shared or timed. This relationship is for shared call

typeT: a coffee call can be shared or timed. This relationship is for timed call

place: the location where the user has gone to take a coffee

2.2 Translation into tables

We translated the entity-relationship diagram into tables that are contained in the real database. We identified the primary and foreign keys of each tables. The translation from entities and relationships into tables was not 1-to-1. Indeed we decided to:

1) Not create a table for each different kind of coffee call (shared or timed). Instead we created only one table for every coffee call. The table contains all the attributes from the two entities plus the attributes of *coffeCall* entity. A new attribute specifies the type of each coffee call. In this way also the *typeS* and *typeT* relationships become useless.
2) Eliminate the *leaves* relationship. When an user decide to leave a joined coffee call, the business logic simply takes care to update the information about his joins.
3) Translate the *proposedFor* relationship into an attribute of each present
4) Translate the *proposedBy* relationship into an attribute of each present
5) Translate the *birthdayOf* relationship into an attribute of each birthday call
6) Translate the *issue* relationship into an attribute of each coffee call
7) Translate the place relationship into an attribute of each user

The final model of the DB, looks in this way:

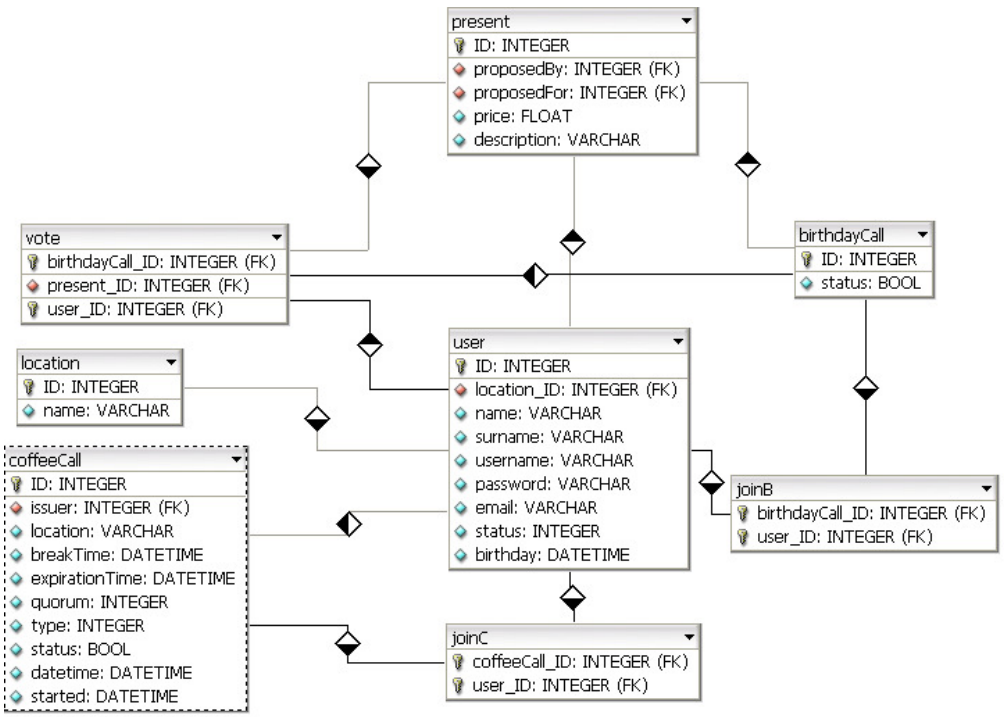


Figure 3. The DB Relational Schema

- In the *vote* table we use *(birthdayCall_ID,user_ID)* as primary key, because from the assumptions in the RASD we know that an user can vote only for a present at a time in a specific birthday call that he has joined.
- The same consideration can be applied to the *joinC* and *joinB* tables where the primary keys are *(coffeeCall_ID, user_ID)* and *(birthdayCall_ID,user_ID),* because we know that an user can join only one time to specific coffee calls or birthday calls.

BUSINESS TIER

3.1 Entity Beans

3.1.1 Introduction

An instance of an "entity bean" object is a component that represents domain elements (relational data) as objects. We have already showed the structure of the relational database, now we translate this structure in a set of entity beans. Each entity beans represents an element in the model of our system.
In the relational schema and in the entity-relationship diagram we saw that there are a lot of relationships between the data elements in our model. In the relational schema these relationships were satisfied using specific attributes, foreign keys and linking tables, like *joinB* and *joinC*.
Going on into the object representation offered by entity beans, these relationships can be satisfied in a different way. For example, we see that the attribute subscribers of the *coffeeCall* entity bean is mapped to a list of *user* entity beans. This offers a practical way to manage data and relationships efficiently. In the class diagram below, we also specified the multiplicity of the relationships between the various entity beans.
All the attributes of an entity bean are marked as private, so it's clear that to manipulate the data are necessary some kind of operations. Thus each entity bean provides some methods (*getters* and *setters*), marked public, to manipulate his attribute.

3.1.2 Entity Beans Class Diagram

- **<<entity>> user:** this entity bean encapsulates all the information associated to an user registred in the system

- **<<entity>> coffeeCall:** this entity bean encapsulates all the information related to a coffeeCall, either shared call or timed call

- **<<entity>> vote:** this entity bean encapsulates all the information regarding a vote in a specific birthday call

- **<<entity>> location:** this entity bean encapsulates all the information associated to a location in which an user can go to take a coffee

- **<<entity>> birthdayCall:** this entity beans encapsulates all the information about a birthdayCall

- **<<entity>> present:** this entity bean encapsulates all the information regarding a proposed present

The complete class diagram looks in this way:

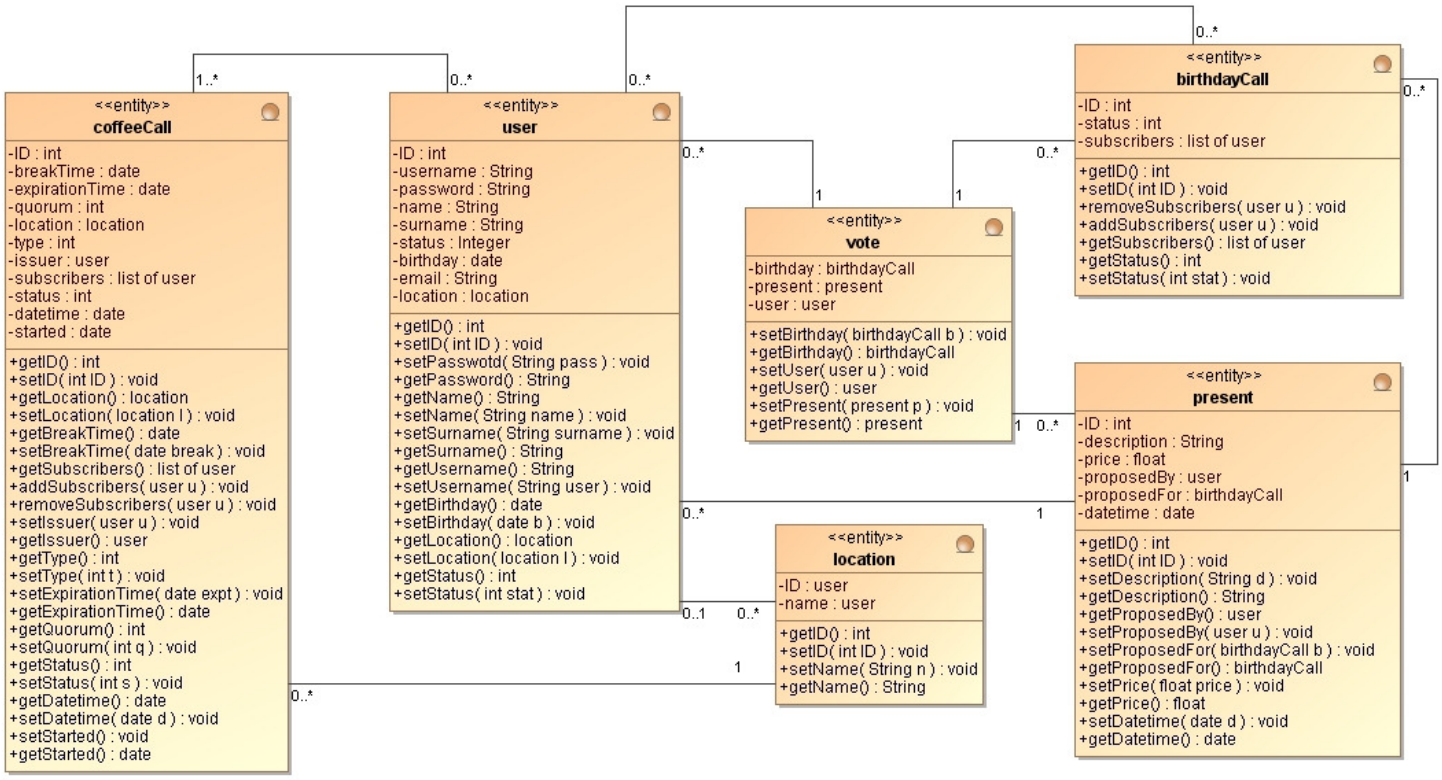


Figure 4. Class Diagram for entity beans

3.2 Session Beans

3.2.1 Introduction

Session beans are the mechanism provided by the JEE platform to implement the business operations of the application. While the entity beans provide a way to represent the data model, the session beans encapsulate the operations and the methods necessary to perform actions in the applications. Session beans deal with mechanisms to create, query and modify data in the model. Session beans continuously act using the persistence information that they collect from the entity beans, and they manage the entity beans through the *EntityManager*.
There two types of session beans, statefull session beans and stateless session beans. In our application both the types are used.

3.2.2 Functional View

As you can see in the following diagram we decomposed the system into five main functional components. In particular we created a component to manage the birthday call aspects (*BirthdayManager*), a component to manage the coffee call aspects (*CoffeeManager*), a main component that interacts with the client application and the users (*UserManager*), and finally two other components that provide specific functionalities (*NotificationManager* and *PresentManager*).

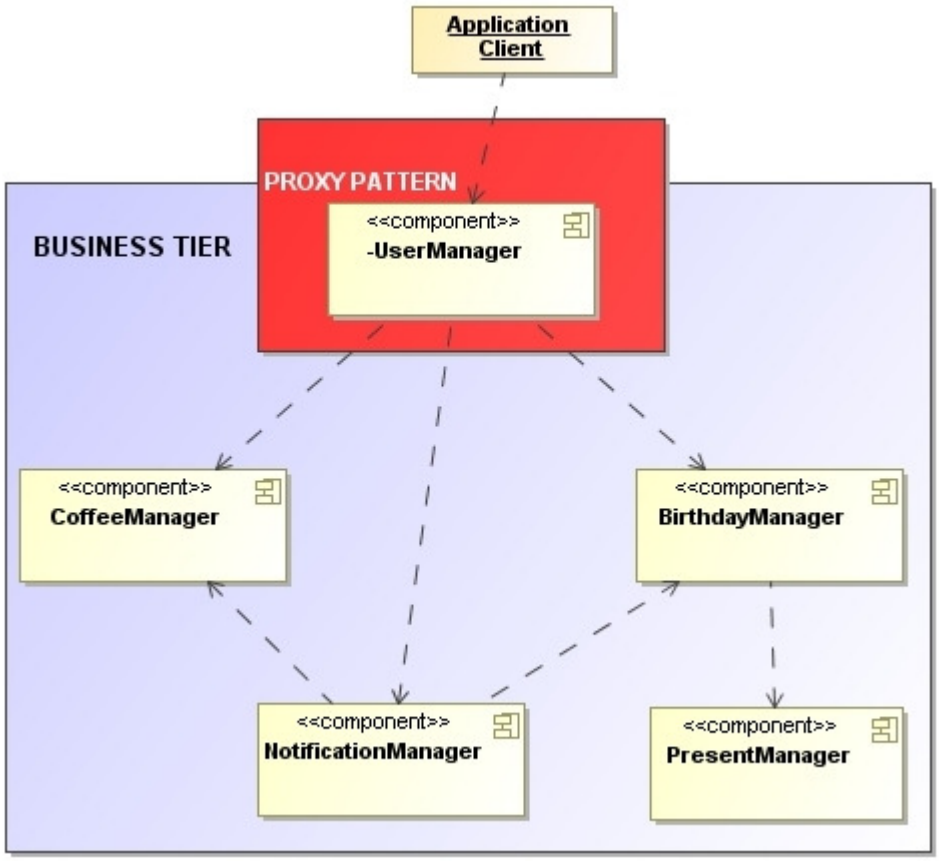


Figure 5. Functional View

- **<<components>> UserManager:**

UserManager deals with the client application and thus with the users. It provides mechanisms to register new users, to login and logout. In addiction *UserManager* encapsulates methods for setting and reading the user status and location. Moreover we can see that the application client controlled by the user, interacts with only one this component. This design was chosen for various reasons:

- First this approach guaranties a strong security. Indeed the client can't directly communicate with all the components on the server, but has to pass always through the control of the *UserManager*. In this way the *UserManager* controls the input provided by the client and can refuse to do some operations if the input is insane.

- Second this mechanism permits an efficient **management of the login, logout** procedures. Indeed the *UserManager* takes care that the client had already

provided is credential before performing any request, and store these credentials during the communication session.

We can say that UserManager acts as a proxy for the others components of the system. This mechanism is definitely inspired by the **proxy design pattern**.

- **<<components>> CoffeeManager:**

CoffeeManager deals with the management of the coffee calls, both shared and timed. It provides methods for creating, joining, listing and leaving coffee calls.

- **<<components>> NotificationManager:**

NotificationManager deals with the management of notifications to the users. Indeed the system triggers a new notification for each new event it may occur.
The notification mechanism is based on the continuous polling by the UserManager to the NotificationManager, for new events. Every time a request of new notifications comes from the UserManager, the NotificationManager checks for planned coffee call, new birthday calls, and so on, and reports the collected information.
NotificationManager also creates and closes birthday calls. Finally *NotificationManager* closes coffee calls that are expired.

- **<<components>> BirthdayManager:**

BirthdayManager deals with the management of the birthday calls. It provides methods for creating, joining, listing birthday call.

- **<<components>> PresentManager:**

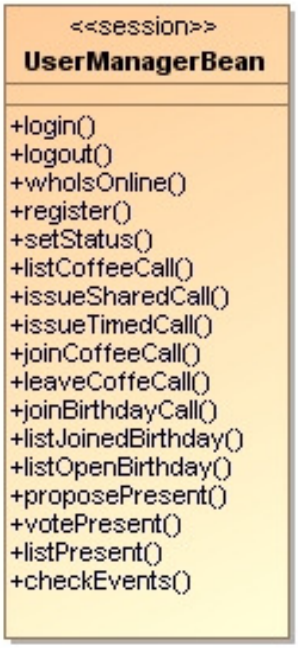
PresentManager deals with the management of the birthday's presents. It provides methods for proposing and voting present.

3.2.3 Detailed View

Each components presented in the functional view has been translated in a session bean. These session beans encapsulate all the methods necessary to run DC4C. In the following paragraphs, each session bean will be discussed in details.

3.2.3.1 UserManagerBean

Type: Statefull session bean



• *login*

– Provide the authentication mechanism

– **Prototype**: public int login(String user, String pass)

```
/** if the authentication goes right it return 0, otherwise it return a specific integer
for each possible error
@param user Username introduced by the user for login
@param pass Password introduced by the user for login
@returns 0 or error code
```

• *logout*

– Close the current session.

– **Prototype**: public int logout()

```
/** if everything go right return 0, otherwise an error code different from 0
@returns 0 or error code
```

• *whoIsOnline*

– Provide a list of users that are currently logged in the system.

– **Prototype**: public user[] whoIsOnline()

```
/** provide a list of user's instances
@returns a list of user's instances
```

• *register*

– Create a new user profile in the system

– **Prototype**: public int login(String name, String surname, String user, String pass, String email, Date birthday)

```
/** if the registration go right it return 0, otherwise an error code
@param name Real name of the user
@param surname Real surname of the user
@param user An username chosen by the user
@param password A password chosen by the user
@param email User's e-mail
@param birthday User's birthdate
```

• *setStatus*

– Set the user status

– **Prototype**: public void setStatus(int status)

```
/** set the new status for the user.
@param status 0 for working, 1 for busy
```

• *listCoffeeCall*

– Provide a list of the active coffee calls in the system
– **Prototype**: public coffeeCall[] listCoffeeCall()
```
@returns a list of coffeeCalls
```

• *issueSharedCall*
– Create a new shared coffee call
– **Prototype**: public int issueSharedCall(location location, int quorum, date expirationTime)
```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param location Location where to take the coffee call
@param quorum The minimum people that have to be present for the coffee call
@param expirationTime when the coffee call will expire if the quorum isn't reached
@return 0 or error code
```

• *issueTimedCall*
– Create a new timed coffee call
– **Prototype**: public int issueTimedCall(location location, date breakTime)
```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param location Location where to take the coffee call
@param breakTime when the coffee call will be taken
@return 0 or error code
```

• *joinCoffeeCall*
– Join a coffee call
– **Prototype**: public int joinCoffeeCall(int coffeeID)
```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param coffeeID The ID of the coffee call to join
@returns 0 or error code
```

• *leaveCoffeeCall*
– Leave a coffee call
– **Prototype**: public int leaveCoffeeCall(int coffeeID)
```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param coffeeID The ID of the coffee call to leave
@returns 0 or error code
```

• *listOpenBirthday*
– Provide a list of all open birthday call
– **Prototype**: public birthdayCall[] listOpenBirthday()
```
@returns a list of open birthday calls
```

• *listJoinedBirthday*
– Provide a list of all joined birthday calls
– **Prototype**: public birthdayCall[] listOpenBirthday()
```
@returns a list of joined birthday calls
```

• *joinBirthdayCall*
– Join a birthday call
– **Prototype**: public int joinBirthdayCall(int birthdayID)
```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param birthdayID The ID of the birthday call to join
@returns 0 or error code
```

• *proposePresent*
– Propose a new present
– **Prototype**: public int proposePresent(String description, float price, int birthdayID)
```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param birthdayID The ID of the birthday call for which propose a present
@param description The present's description
@param price The present's price
@returns 0 or error code
```

• *votePresent*
– Vote a present
– **Prototype**: public int votePresent(int presentID)
```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param presenID The ID of the present for which vote
@returns 0 or error code
```

• *listPresent*
– Provide a list of the presents for a specific birthday call
– **Prototype**: public present[] listPresent(int birthdayID)

```
@param birthdayID The ID of the birthday call
@returns a list of presents
```

• *checkEvents*
– Provide a a list of notifications for the user
– **Prototype**: public notification[] checkEvents()

```
@returns a list of notifications
```

3.2.3.2 CoffeeManagerBean

Type: Stateless session bean

• *listCoffeeCall*
– Provide a list of the active coffee calls in the system
– **Prototype**: public coffeeCall[] listCoffeeCall()

```
@returns a list of coffeeCalls
```

• *issueCall*
– Create a new coffee call
– **Prototype**: public int issueCall(location location, date breakTime, int quorum, date expirationTime, int type, int issuer)

```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param location Location where to take the coffee call
@param breakTime when the coffee call will be taken
@param quorum The minimum people that have to be present for the coffee call
@param expirationTime When the coffee call will expire if the quorum isn't reached
@param type The coffee call's type. 0 for timed, 1 for shared
@param issuer The ID of the call's issuer
@return 0 or error code
```

• *joinCall*
– Join a coffee call
– **Prototype**: public int joinCall(int coffeeID, int userID)

```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param coffeeID The ID of the coffee call to join
@param userID The ID of the user that wants to join
@returns 0 or error code
```

• *leaveCall*
– Leave a coffee call
– **Prototype**: public int leaveCall(int coffeeID, int userID)

```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param coffeeID The ID of the coffee call to leave
@param userID The ID of the user that wants to leave
@returns 0 or error code
```

3.2.3.3 BirthdayManagerBean

Type: Stateless session bean

• *listOpenBirthday*
– Provide a list of all open birthday call
– **Prototype**: public birthdayCall[] listOpenBirthday()

```
@returns a list of open birthday calls
```

• *listJoinedBirthday*
– Provide a list of all joined birthday calls
– **Prototype**: public birthdayCall[] listOpenBirthday(int userID)

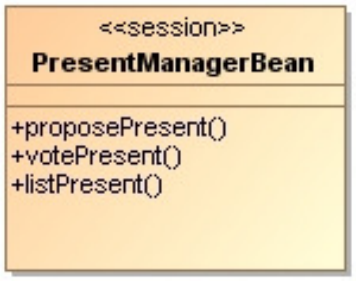
```
@param userID The ID of the user that joined the birthday calls
@returns a list of joined birthday calls
```

• *joinBirthdayCall*
– Join a birthday call
– **Prototype**: public int joinBirthdayCall(int birthdayID, int userID)

```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param birthdayID The ID of the birthday call to join
@param userID The ID of the user that wants to join
@returns 0 or error code
```

3.2.3.3 PresentManagerBean

Type: Stateless session bean



• *proposePresent*
– Propose a new present
– **Prototype**: public int proposePresent(String description, float price, int birthdayID, int userID)

```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param birthdayID The ID of the birthday call for which propose a present
@param description The present's description
@param price The present's price
@param userID The ID of the user that proposes the present
@returns 0 or error code
```

• *votePresent*
– Vote a present
– **Prototype**: public int votePresent(int presentID, int userID)

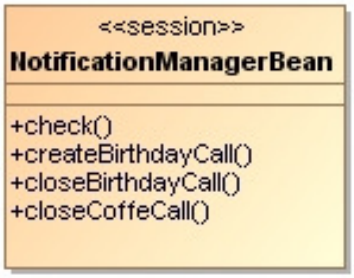
```
/** if everything goes right it return 0, otherwise return an error code different from 0
@param presenID The ID of the present for which vote
@param userID The ID of the user that wants to vote
@returns 0 or error code
```

• *listPresent*
– Provide a list of the presents for a specific birthday call
– **Prototype**: public present[] listPresent(int birthdayID)

```
@param birthdayID The ID of the birthday call
@returns a list of presents
```

3.2.3.4 NotificationManagerBean

Type: Stateless session bean



• *check*
– Check the new events for a specific user and return a list of notification instances
– **Prototype**: public notification[] check(int userID)

```
@param userID The ID of the user whose the birthday is
@return a list of notifications
```

The implementation of this method can be illustrated through a simple algorithm:

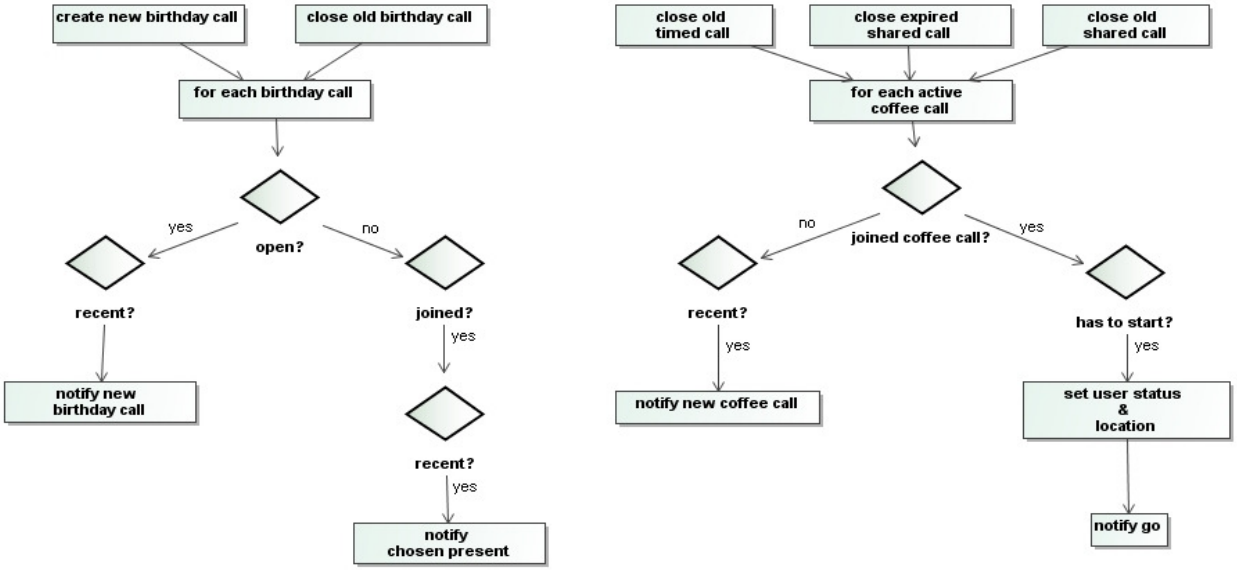
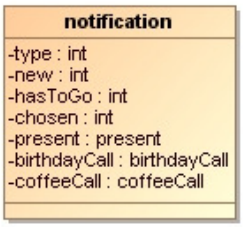


Figure 6. Algorithm for check notifications

This is the class diagram of the class notification*:*



• *createBirthdayCall*
– Create a birthday call
– **Prototype**: public void createBirthdayCall(int userID)
@param userID The ID of the user whose the birthday is

• *closeBirthdayCall*
– Close a birthday call
– **Prototype**: public void closeBirthdayCall(int birthdayID)
@param birthdayID The ID of the birthday call to close

• *closeCoffeeCall*
– Close a coffee call
– **Prototype**: public void closeBirthdayCall(int coffeeID)
@param coffeeID The ID of the coffee call to close

3.3 Complete Class Diagram

The following is the complete class diagram of the application. We don't show the attributes and the methods of each class to improve the readability of the diagram. In the diagram are represented all the entity beans as the session beans. The relationships between the various components are showed as well.

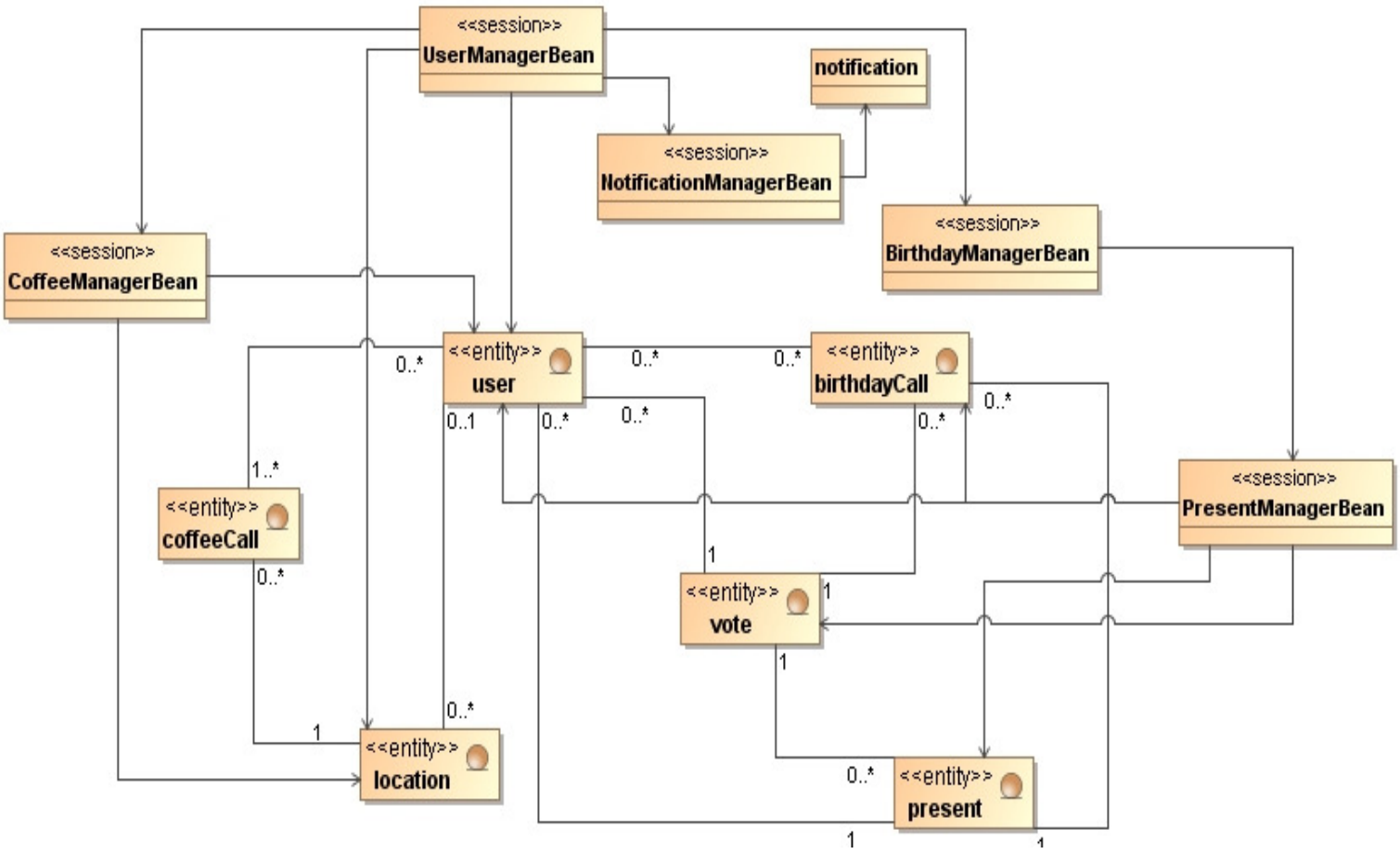


Figure 7. Complete Class Diagram

3.4 Sequence Diagram

To explain the interaction among the various components of the system, we present two sequence diagrams for two common situations:

1. join a birthday call and propose a present
2. join a coffee call, wait for notification and go to take the coffee

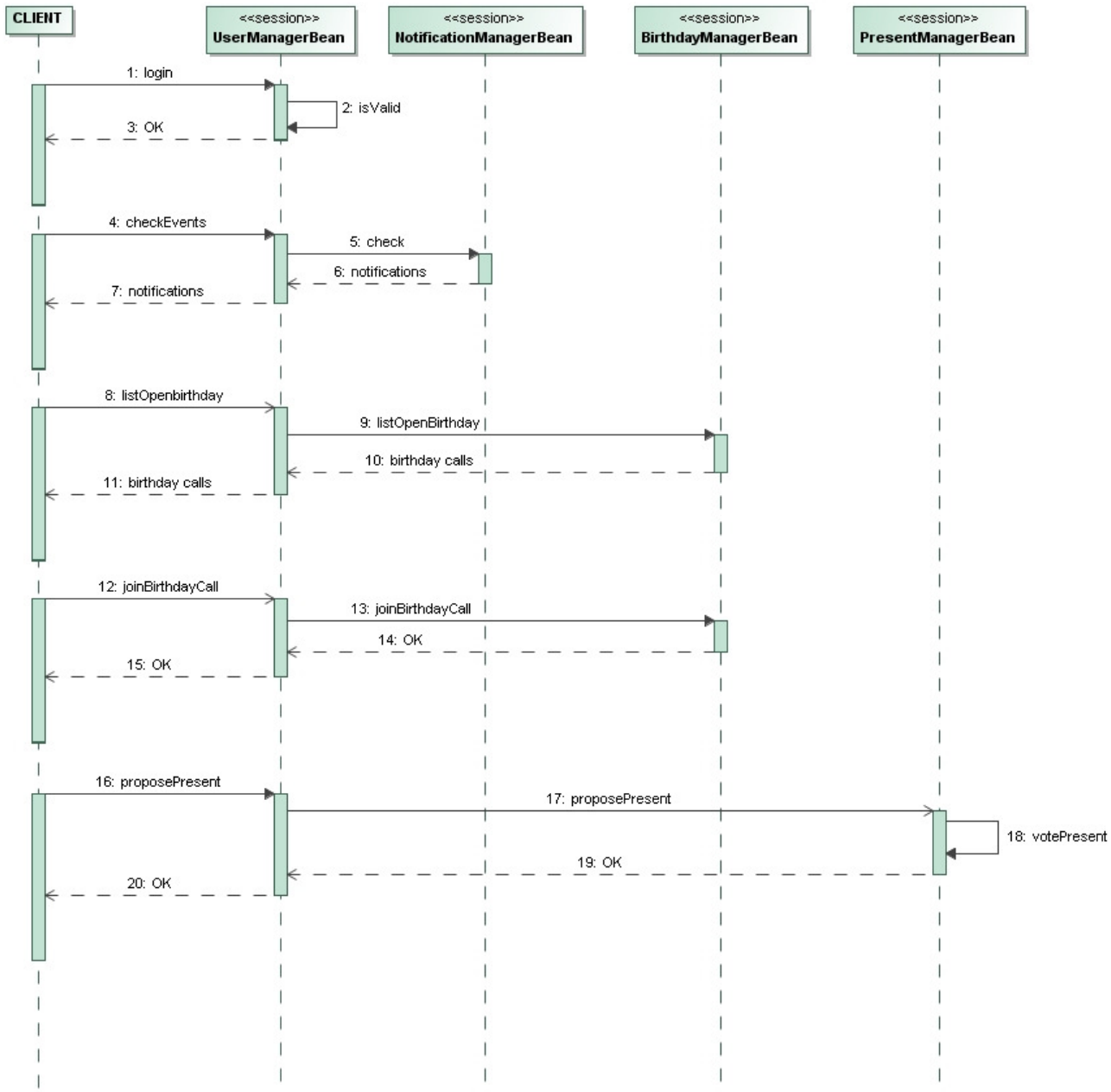


Figure 8. Birthday Call Sequence Diagram

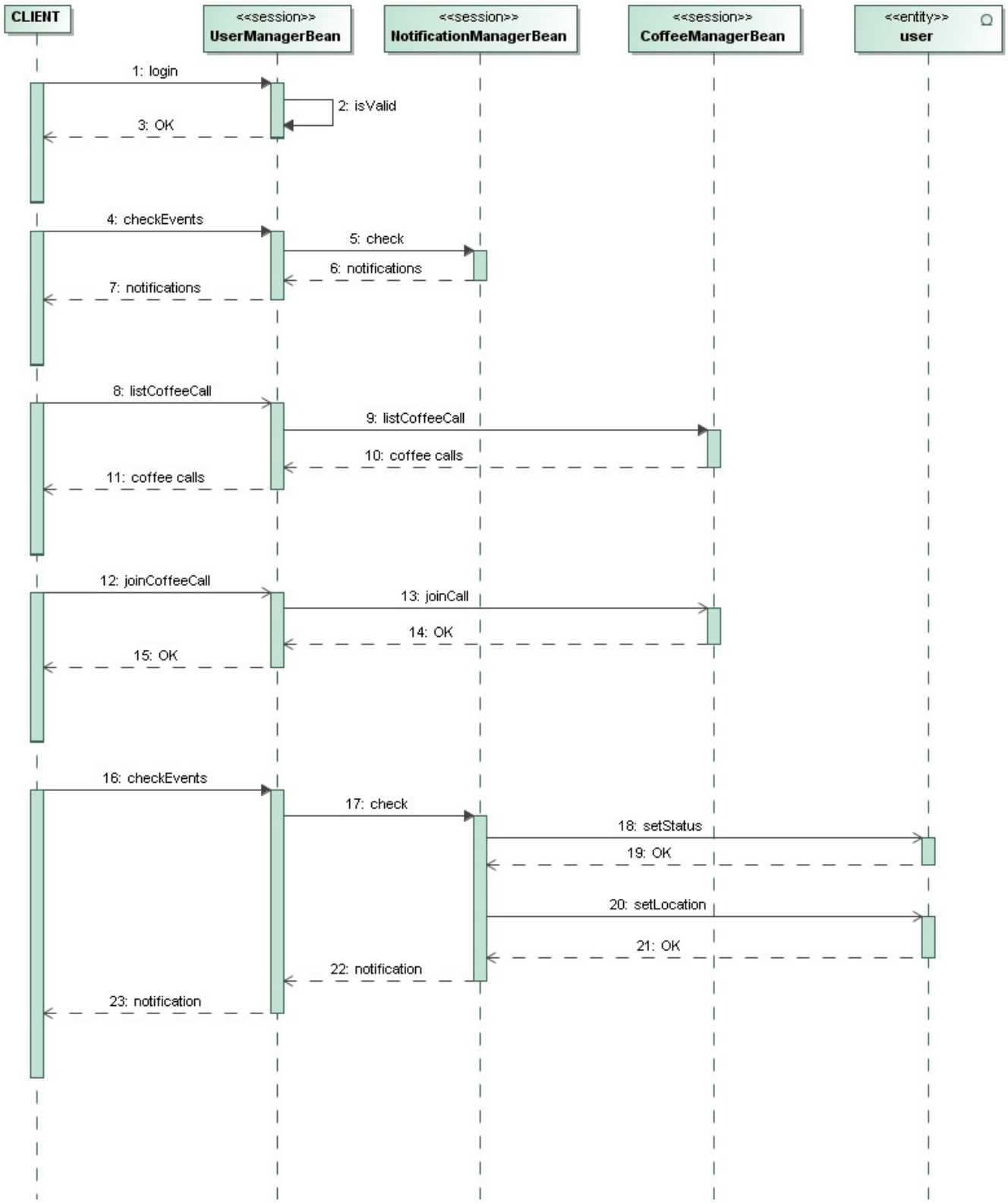


Figure 9. Coffee Call Sequence Diagram

CLIENT TIER

4.1 Introduction

The client of our application will be a graphical java application that will interact with the component on the server through shared interfaces.
To design the client tier of our application, we build a User eXperience diagram (UX diagram). UX diagram takes in account only the part of the system which is visible to users.

4.2 User eXperience

- The stereotype *<<screen>>* represents a screen presented to the user during the navigation, and the methods attached to it are all the actions that a user can perform
- The stereotype *<<inputForm>>* represents an interaction module
- The stereotype *<<item>>* represents a possible item in a list showed to the user

When the user starts the system, the system will display a *LoginScreen*. From this screen the user will have the possibility to perform the *login()* action.

If he has not registered yet, he cannot perform the *login()* action, but he will have to perform the *register()* action first. In this case he will have to fill the register form with all the relevant data (username, password, etc.). At this stage he will control the data and perform the submit() action, that will save all the data in the newly created profile.

After performing the *login()* action in the *LoginScreen* the user can reach his personal *HomeScreen*. From this screen he can access the three main functionalities of the system, by performing the appropriate actions. If the user performs the *whoIsOnline()* action he receives the list of all the online users. If he performs the *manageCoffee()* or *manageBirthday(),* the system will display the corresponding screen.
In the *HomeScreen* the user can also see a list of notification for him.

When the user navigates to the *ManageCoffee(),* he can further decide to perform the *setStatus(), issueSharedCall()* or *issueTimedCall()* actions by completing the corresponding form. In this case he does not exit the screen. In addition the user can perform the *listCall()* action in order to get the list of the available timed and shared coffee calls. For each of them he can either decide to perform the *join()* or *leave()* action.

When the user navigates to the *ManageBirthday(),* he can further decide to perform the *listOpenCall()* or *listJoinedCall()* actions. By performing the former the user accesses the *ListOpenCall* screen displaying the list of all open birthday calls. For each of them he can decide to perform the *join()* action.

By performing the *listJoinedCall()* action the user can see all the birthday balls he has joined. For each of them he can perform the *proposePresent()* or *listPresents()* actions. The former action implies the filling of the propose form. The latter action displays the *ListPresent* screen, which represents the list of all the presents for the chosen birthday Call. For each of them the user can perform the *votePresent()* action.

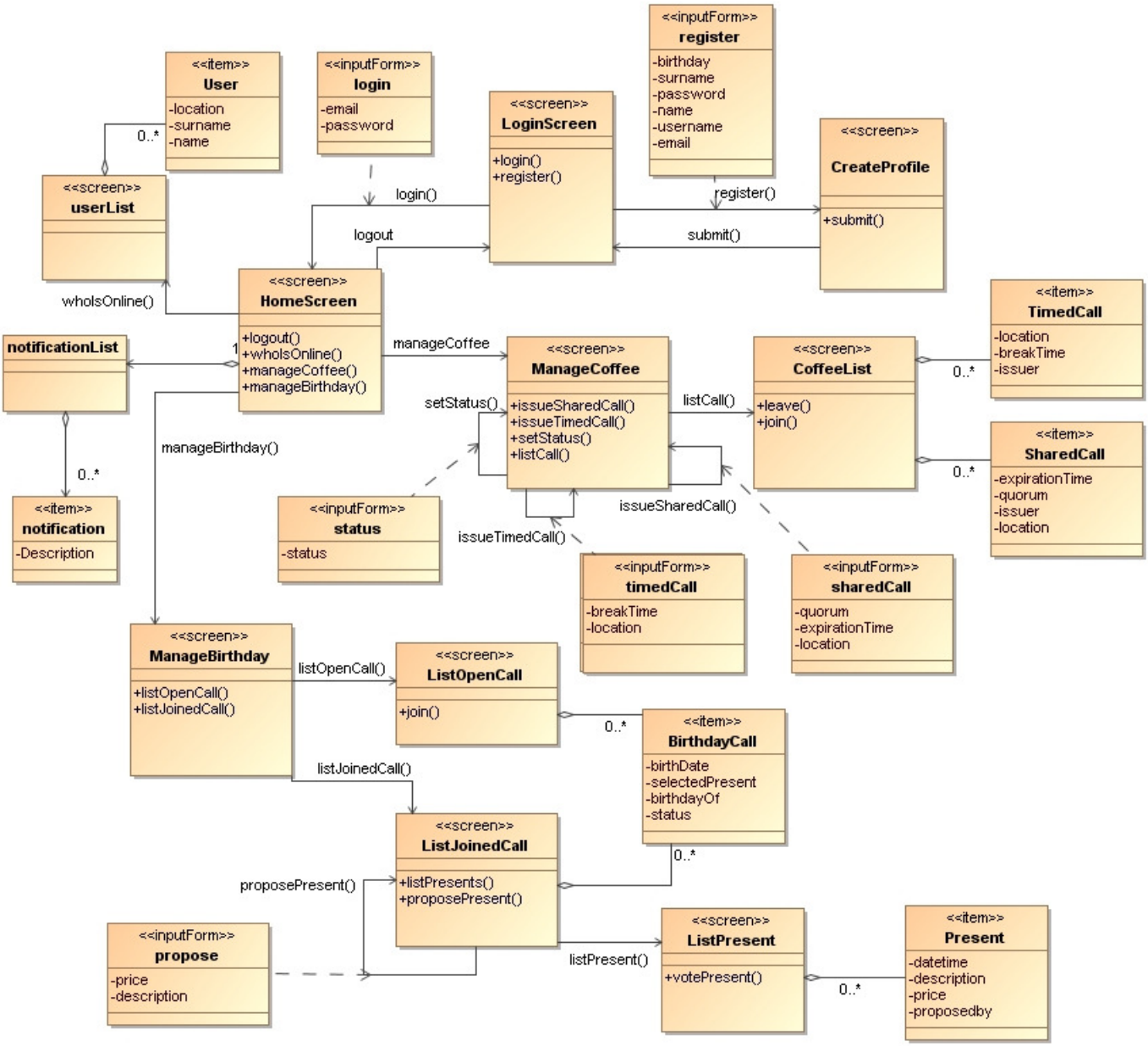


Figure 10. User Experience Model