# DESIGN DOCUMENT

POLITECNICO
MILANO 1863

Author: Riccardi Vincenzo 793241

Reference Professor: Mirandola Raffaela

# Summary

# 1.Introduction

## Purpose

The present document proposes to specify the architectural choices adopted to implement the requested system. In particular, it describes the interaction between users and the application. In this document will describe the system architecture, data modelling and database structures.

## Scope

The scope of this document is to provide the design of the application, describing and justifying the reason of our choices, the implementation will be precisely defined in the implementation phase.

# Definition, acronymus and abbreviation

- **DD**: design document;
- **Application**: term used to indicate the software system which will be developed;
- **Visitor**: a person who visits the application without being logged in.
- **User**: a person who has yet done registration and who can use all the functionalities.
- **HTML**: Hyper Text Mark-up Language
- **MVC**: Model-View-Controller Pattern
- **Java EE**: Java Enterprise Edition
- **DBMS**: Database Management System

# Document structure

1. Introduction
2. Architectural Design
3. Algorithm Design
4. User Interface Design
5. Appendix

# Sources and references

- Requirements Analysis and Specification Document (RASD), attached to this document.
- Software Design Document (SDD) Template:

  https://beep.metid.polimi.it/

- IEEE Standard for Information Technology-Systems Design and Software Design.

# 2.ARCHITECTURAL DESIGN

## Architectural structure

As already explained in RASD document, the application used the client-server architecture.

A client sends information at web server, which processes the request and shows dynamic web pages. The web server is connected to Database.

The infrastructure of our application is based on Java Enterprise Edition, so the architecture of the system is composed of four basic tier:

- Client Tier
- Web Tier
- Business Logic Tier
- Database Tier

The client tier running on the client machine, the client communicate through web pages running in the web tier (case of a client running in a browser). Web browser renders the pages received from the server.

The web tier manages user input and send it to the business tier for processing. The server generates content based on user interaction and is connected with user interface.

The business logic tier is responsible for data manipulation and interaction with the database. This tier contains the Enterprise Java Beans (business logic) and Java Persistence Entities (connected to the database).

To access an application that is deployed on the server, the client invokes the bean's method.

The database tier contains the data source and manages the operations of recovering data from database and writing data on it. This tier is managed by a MySQL Server.

# Architectural styles and patterns

The proposed architecture is adhering to the architectural pattern Model-View-Controller (MVC). The model directly manages the data, logic and rules of the application.
A *view* can be any output representation of information, such as a chart or a diagram; multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
The third part, the *controller*, accepts input and converts it to commands for the model or view.

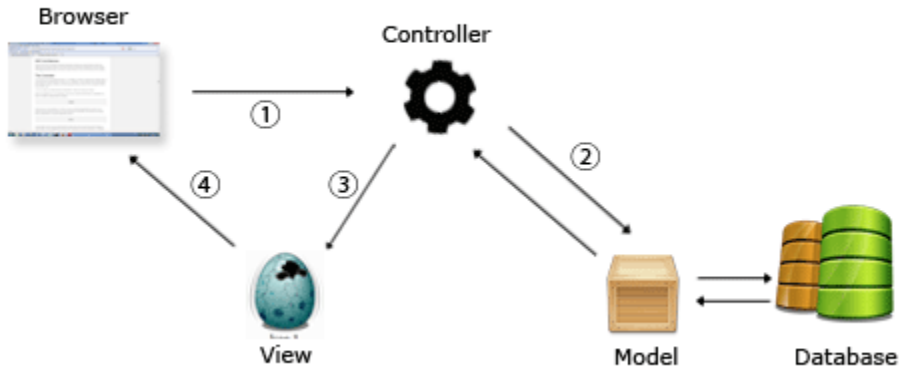
In addition to dividing the application into three kinds of components, the model–view–controller design defines the interactions between them:

- A **controller** can send commands to the model to update the model's state (e.g., editing a document). It can also send commands to its associated view to change the view's

presentation of the model (e.g., by scrolling through a document).

- A **model** stores data that is retrieved according to command from the controller and displayed in the view.
- A **view** generates an output presentation to the user based on changes in the model.
- A **view controller** generates an output view and an embedded controller

The Model-View-Controller architecture



# 3.COMPONENT VIEW

## BCE diagram

I decided to go into details of the structure of our application using a class diagram that represents its components and the relationship among them. In particular, I used the Boundary-Entity-Control pattern (BCE) pattern.

Boundary represents the view, so the interaction between the user and the application.

Controllers define the connection between the boundaries and the entities.

Finally the entities represent the model of the application.

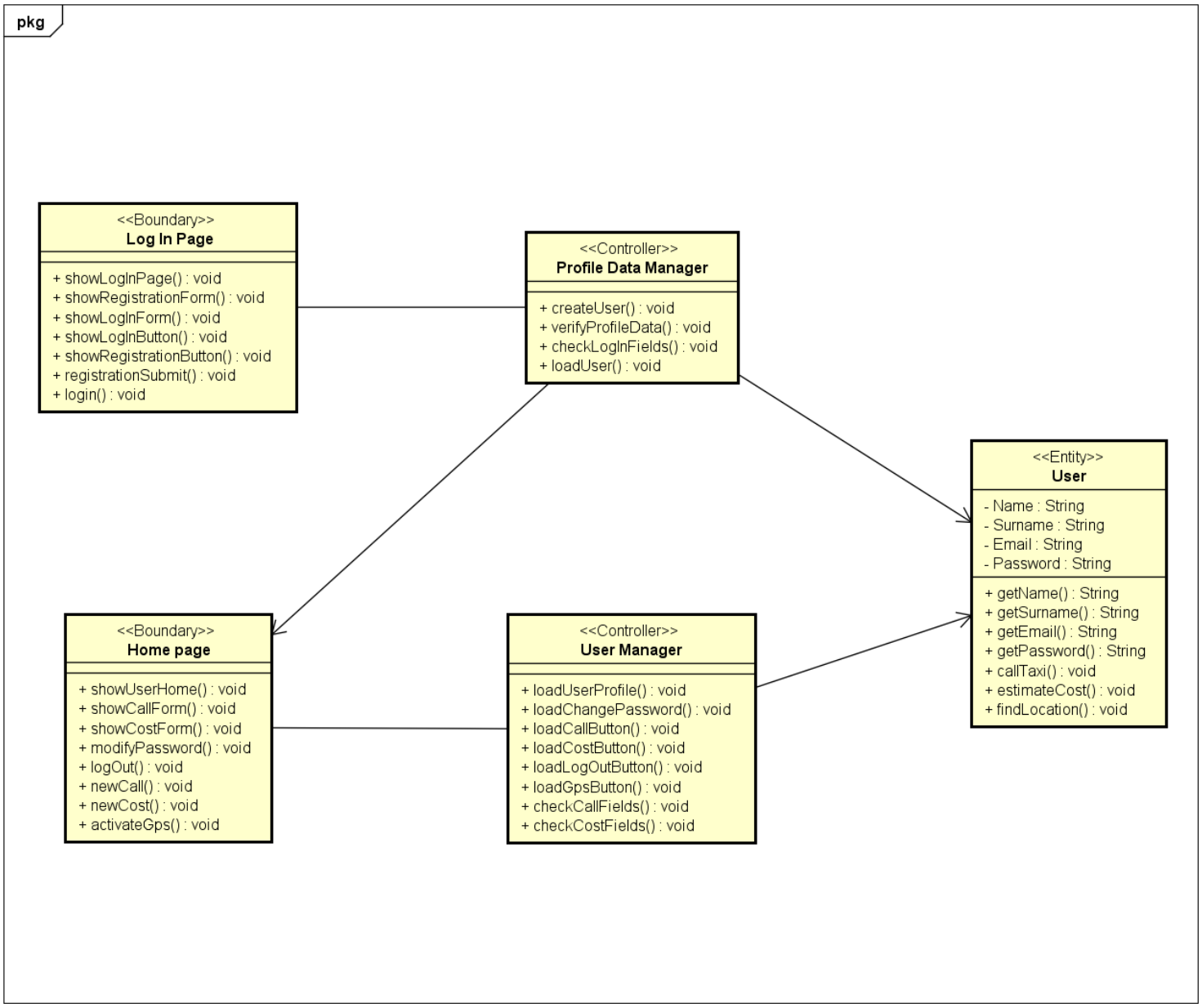
# Log in and registration

The two boundaries Home Page and Log in Page represent the interface between user and system. I have not done difference between user and visitor as they see the same log in page.

There are two main controllers:

- **Profile data manager**
  It controls the creation of a new user and the correctness of information inserted by the user/visitor during log in/registration.

- **User Manager**
  It controls the interaction with the user when he is in the home page; it manage the load of profile's information and the functionalities of the buttons.


pkg
<<Boundary>>
Log In Page
+ showLogInPage() : void
+ showRegistrationForm() : void
+ showLogInForm() : void
+ showLogInButton() : void
+ showRegistrationButton() : void
+ registrationSubmit() : void
+ login() : void
<<Controller>>
Profile Data Manager
+ createUser() : void
+ verifyProfileData() : void
+ checkLogInFields() : void
+ loadUser() : void
<<Entity>>
User
- Name : String
- Surname : String
- Email : String
- Password : String
+ getName() : String
+ getSurname() : String
+ getEmail() : String
+ getPassword() : String
+ callTaxi() : void
+ estimateCost() : void
+ findLocation() : void
<<Boundary>>
Home page
+ showUserHome() : void
+ showCallForm() : void
+ showCostForm() : void
+ modifyPassword() : void
+ logOut() : void
+ newCall() : void
+ newCost() : void
+ activateGps() : void
<<Controller>>
User Manager
+ loadUserProfile() : void
+ loadChangePassword() : void
+ loadCallButton() : void
+ loadCostButton() : void
+ loadLogOutButton() : void
+ loadGpsButton() : void
+ checkCallFields() : void
+ checkCostFields() : void

# Call taxi

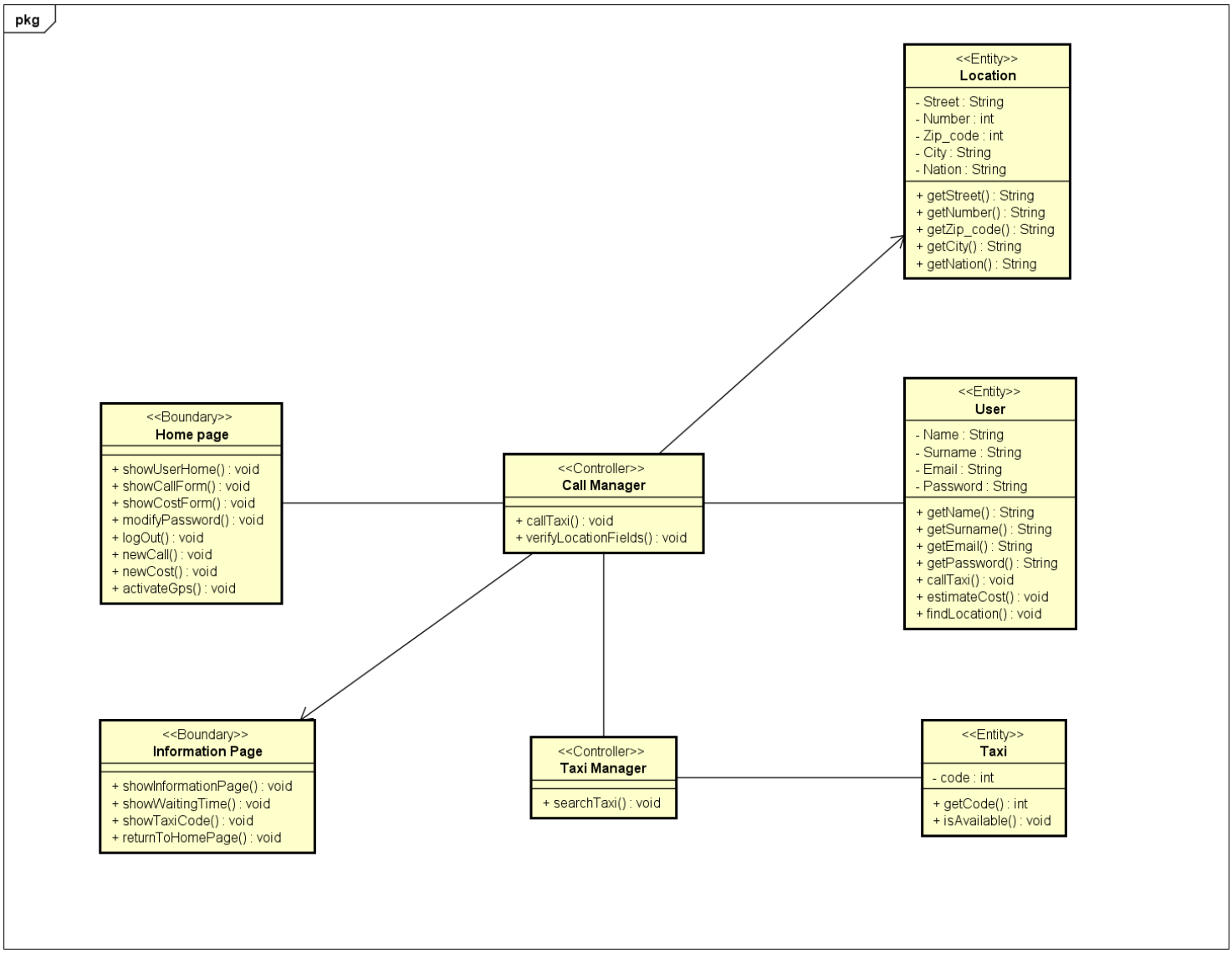
The two boundary Information Page and Home Page represent the interfaces between the system and the user.

The first exposes the main functionality that can be activate when the user clicks on call button. The second page allows the user to insert location and call taxi.

There are two controller:

- **Call Manager**
  It controls the correctness of the location information inserted by the user when he clicks on the call button.

- **Taxi Manager**
  It shall find a taxi available and communicates with the call manager to complete the user request.

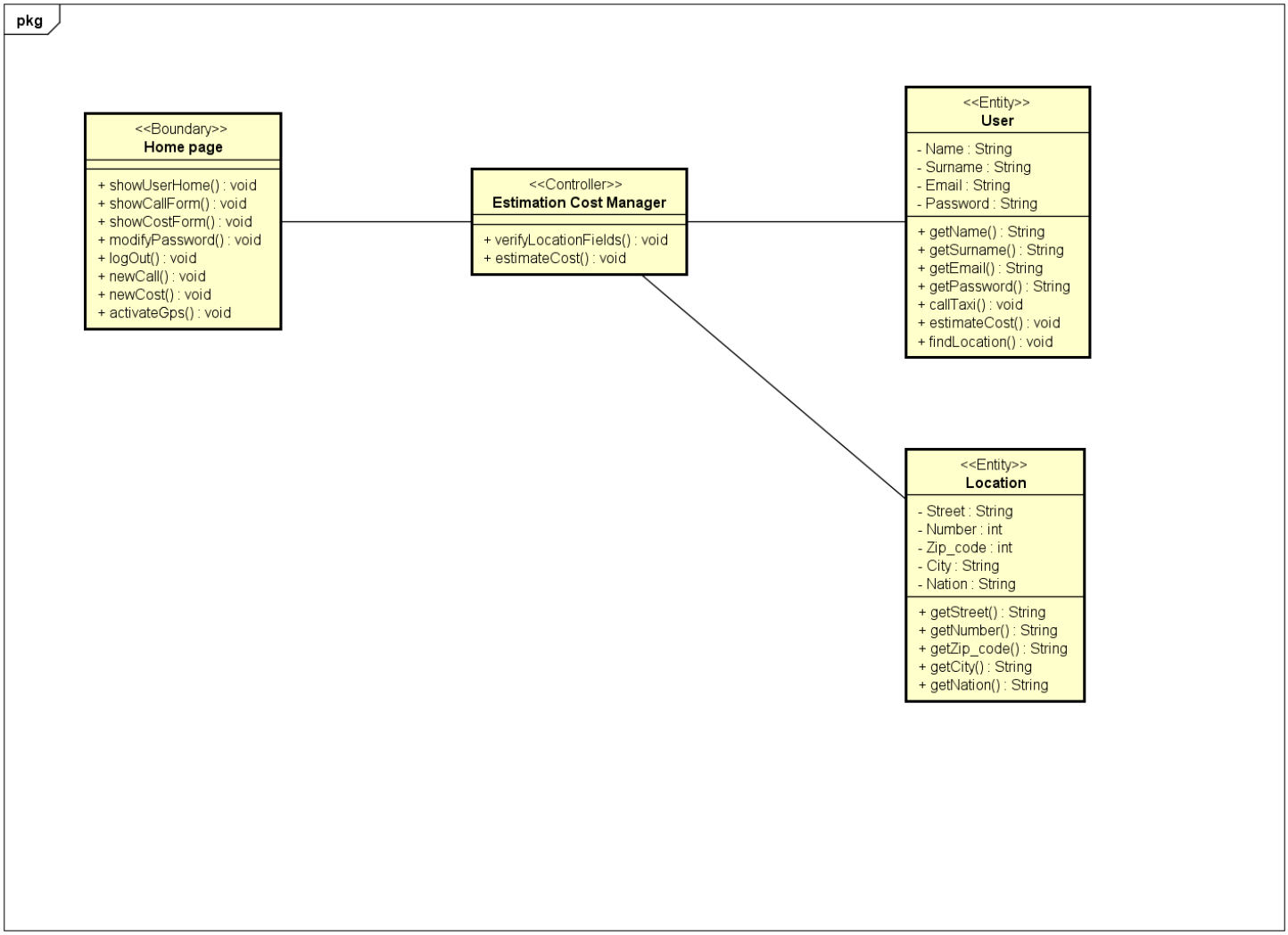

pkg
<<Entity>>
Location
- Street : String
- Number : int
- Zip_code : int
- City : String
- Nation : String
+ getStreet() : String
+ getNumber() : String
+ getZip_code() : String
+ getCity() : String
+ getNation() : String
<<Boundary>>
Home page
+ showUserHome() : void
+ showCallForm() : void
+ showCostForm() : void
+ modifyPassword() : void
+ logOut() : void
+ newCall() : void
+ newCost() : void
+ activateGps() : void
<<Controller>>
Call Manager
+ callTaxi() : void
+ verifyLocationFields() : void
<<Entity>>
User
- Name : String
- Surname : String
- Email : String
- Password : String
+ getName() : String
+ getSurname() : String
+ getEmail() : String
+ getPassword() : String
+ callTaxi() : void
+ estimateCost() : void
+ findLocation() : void
<<Boundary>>
Information Page
+ showInformationPage() : void
+ showWaitingTime() : void
+ showTaxiCode() : void
+ returnToHomePage() : void
<<Controller>>
Taxi Manager
+ searchTaxi() : void
<<Entity>>
Taxi
- code : int
+ getCode() : int
+ isAvailable() : void

# Estimation Cost

The boundary Home Page represent the interfaces between the system and the user. This boundary exposes the functionality that can be activated when the user click on cost button.

There is one controller:

- **Estimation Cost Manager**
  It provides an estimate of the cost of a specific trip and controls the correctness of the location information inserted by the user when he clicks on the cost button.


pkg
<<Boundary>>
Home page
+ showUserHome() : void
+ showCallForm() : void
+ showCostForm() : void
+ modifyPassword() : void
+ logOut() : void
+ newCall() : void
+ newCost() : void
+ activateGps() : void
<<Controller>>
Estimation Cost Manager
+ verifyLocationFields() : void
+ estimateCost() : void
<<Entity>>
User
- Name : String
- Surname : String
- Email : String
- Password : String
+ getName() : String
+ getSurname() : String
+ getEmail() : String
+ getPassword() : String
+ callTaxi() : void
+ estimateCost() : void
+ findLocation() : void
<<Entity>>
Location
- Street : String
- Number : int
- Zip_code : int
- City : String
- Nation : String
+ getStreet() : String
+ getNumber() : String
+ getZip_code() : String
+ getCity() : String
+ getNation() : String

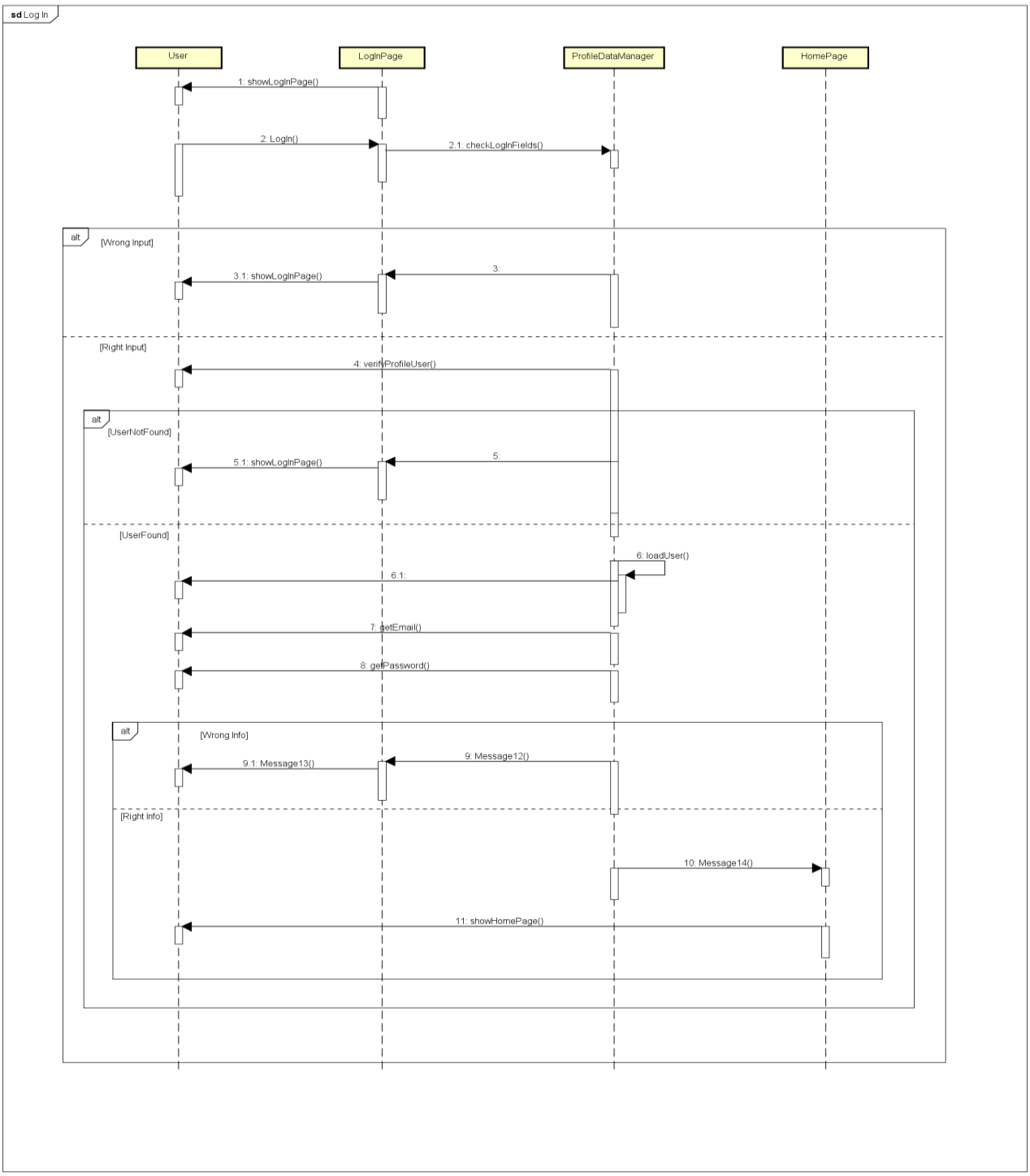
# 4.RUNTIME VIEW

## Sequence diagram

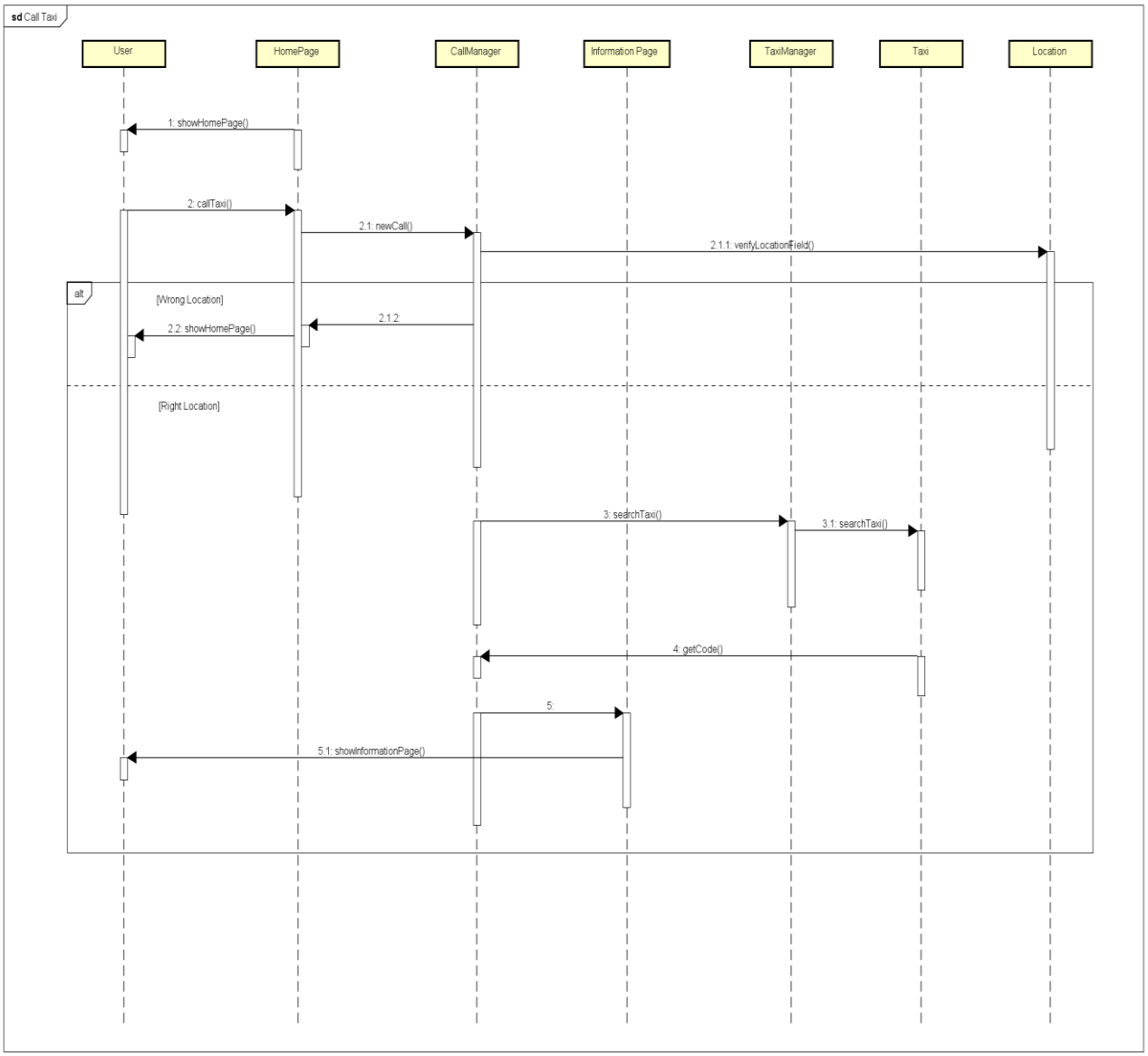
A Sequence diagram is an interaction diagram that shows how processes operate with one another and in what order.

A sequence diagram shows object interactions arranged in time sequence. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. This allows the specification of simple **runtime scenarios** in a graphical manner.

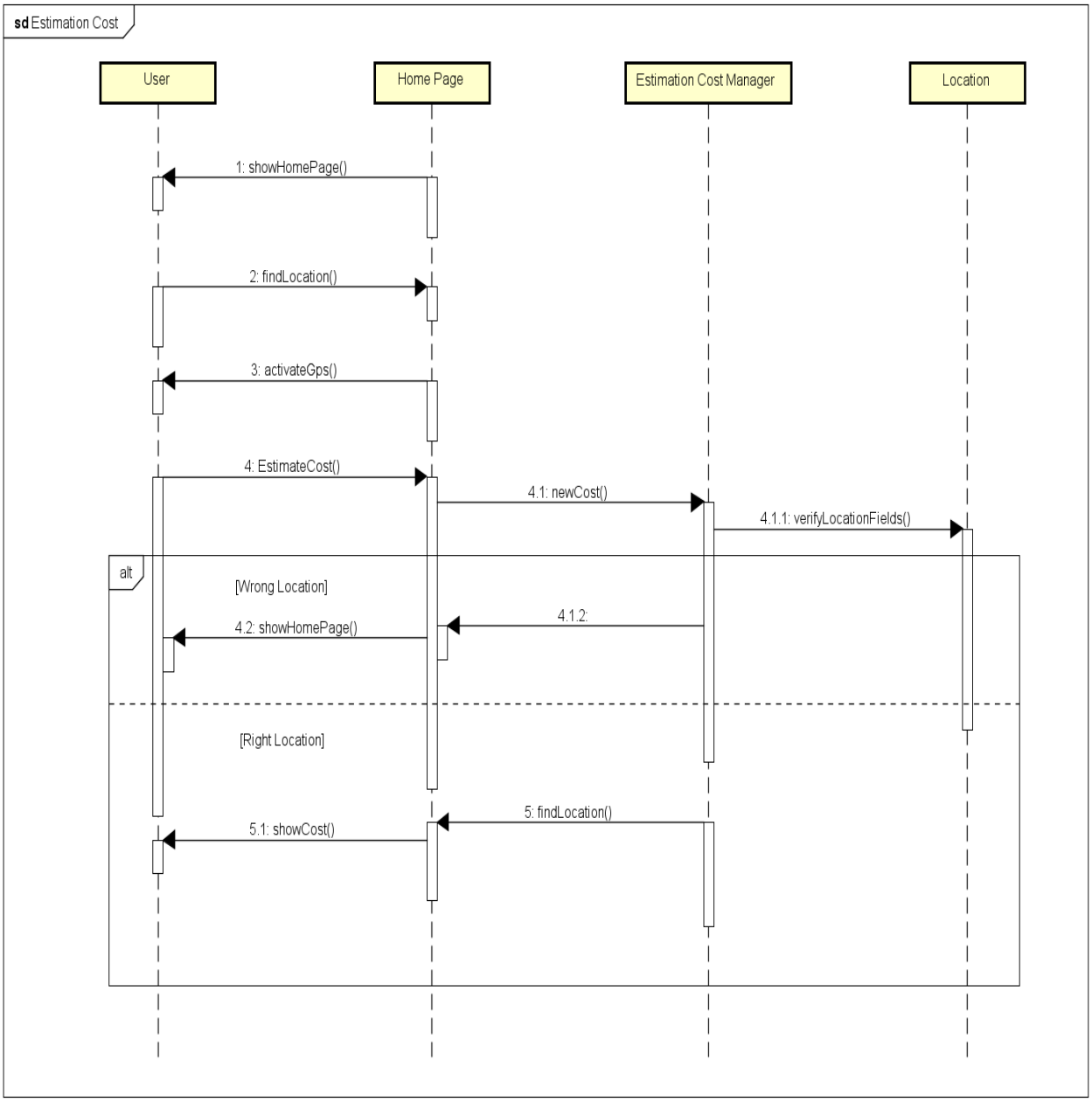
# Log in


sd Log In
User
LogInPage
ProfileDataManager
HomePage
1: showLogInPage()
2: LogIn()
2.1: checkLogInFields()
alt
[Wrong Input]
3.1: showLogInPage()
3:
[Right Input]
4: verifyProfileUser()
alt
[UserNotFound]
5.1: showLogInPage()
5:
[UserFound]
6: loadUser()
6.1:
7: getEmail()
8: getPassword()
alt
[Wrong Info]
9.1: Message13()
9: Message12()
[Right Info]
10: Message14()
11: showHomePage()

# Call Taxi

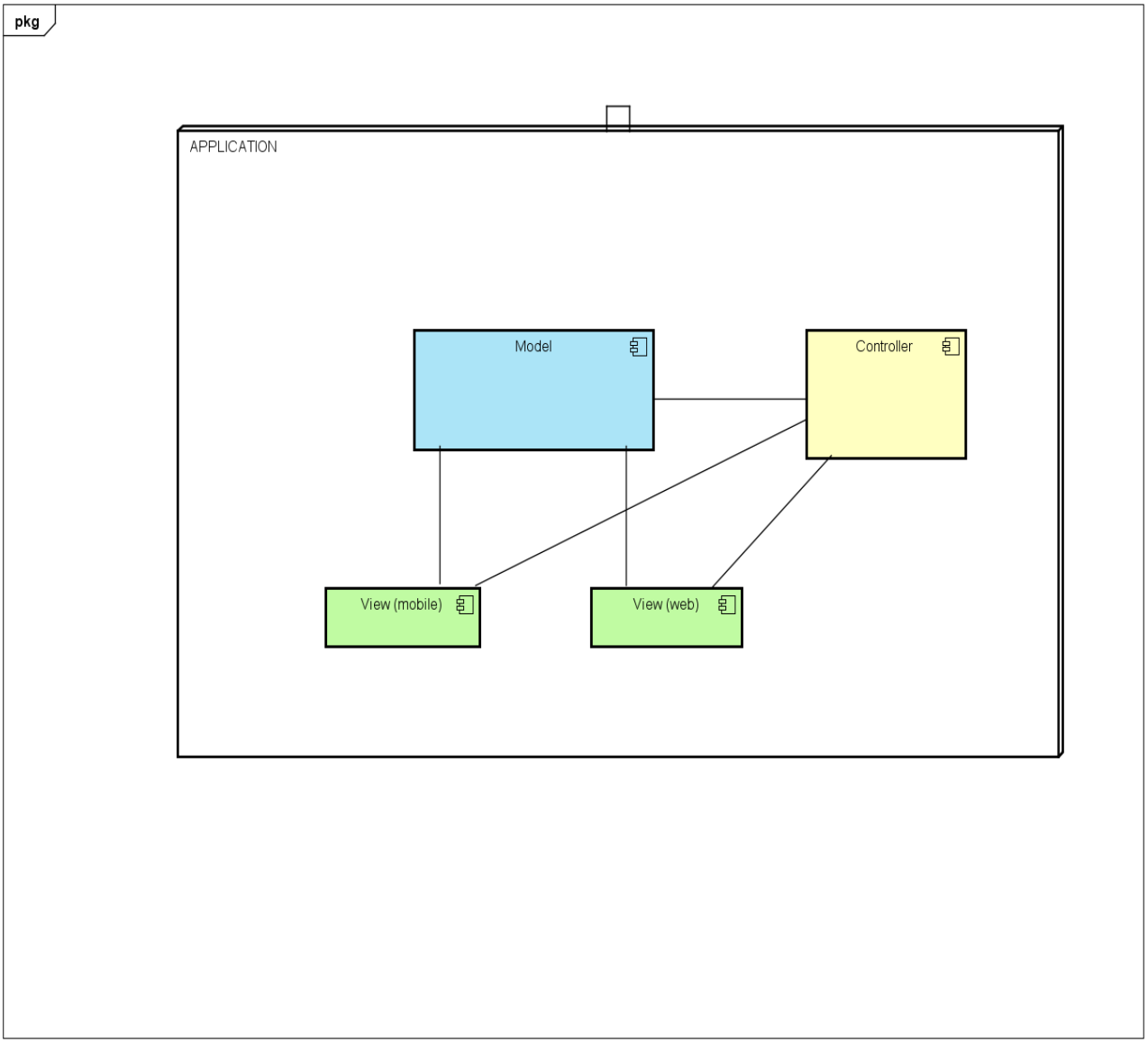

sd Call Taxi
User
HomePage
CallManager
Information Page
TaxiManager
Taxi
Location
1: showHomePage()
2: callTaxi()
2.1: newCall()
2.1.1: verifyLocationField()
alt
[Wrong Location]
2.2: showHomePage()
2.1.2:
[Right Location]
3: searchTaxi()
3.1: searchTaxi()
4: getCode()
5:
5.1: showInformationPage()

# Estimation Cost



# 5. DEPLOYMENT E USER VIEW

## Deployment Diagram



# UX diagram

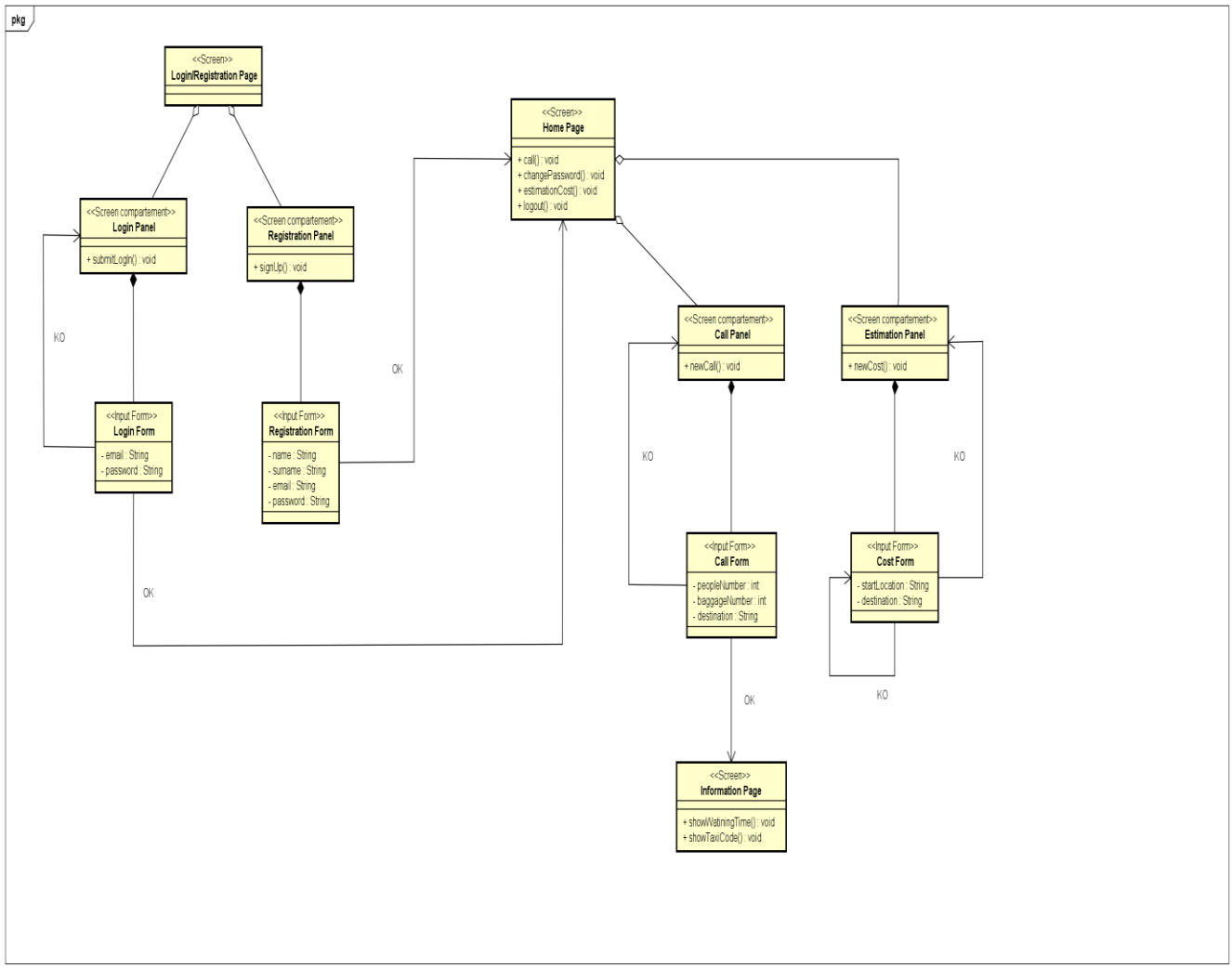
User interface design user interface engineering is the design of user interfaces for machines and software, such as computers, home appliances, mobile devices, and other electronic devices, with the focus on maximizing the user experience. The goal of user interface design is to make the user's interaction as simple and efficient as possible, in terms of accomplishing user goals. 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.


pkg
<<Screen>>
LoginRegistration Page
<<Screen>>
Home Page
+ call() : void
+ changePassword() : void
+ estimationCost() : void
+ logout() : void
<<Screen compartement>>
Login Panel
+ submitLogin() : void
<<Screen compartement>>
Registration Panel
+ signUp() : void
<<Screen compartement>>
Call Panel
+ newCall() : void
<<Screen compartement>>
Estimation Panel
+ newCost() : void
<<Input Form>>
Login Form
- email : String
- password : String
<<Input Form>>
Registration Form
- name : String
- surname : String
- email : String
- password : String
<<Input Form>>
Call Form
- peopleNumber : int
- baggageNumber : int
- destination : String
<<Input Form>>
Cost Form
- startLocation : String
- destination : String
<<Screen>>
Information Page
+ showWaitingTime() : void
+ showTaxiCode() : void
KO
OK
KO
OK
KO
KO
OK
KO

# 6.ALGORITHM DESIGN

## Pseudocode of user call request

```
While ( true ) {
   flag = 0;
   If ( receiveRequest() ){
      idZone = request.idRequestZone();
      queue = findQueueByIdZone( idZone);
      Taxi taxi = (Taxi) queue.first.getElement();
      While ( flag == 0 ){
         If( taxi.isAvailable() ){
            sendConfirmation( user );
            flag = 1;
         }
         else {
```

```
            queue.end.setElemen( queue.first.
            getElement() );
            queue.first.delete();
            taxi = queue.first.getElement();
            }
        }
    }
}
```

# 7.USER INTERFACE DESIGN

I included the description of User interface in the section **3.2** of the **RASD**. With this document I want to focus on the structure of the user interface.

# 8.APPENDIX

The tools we used to create the Design Document are:

- Microsoft Office Word 2011.
- Astah: to create UML Models.

I spent 40 hours to redacting and writing this document.