Medical software for a clinic

Student: Onaciu Andreea

**Group: 30434**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 7

4. UML Sequence Diagrams 11

5. Class Design 12

6. Data Model 13

7. System Testing 14

8. Bibliography 14

1. Requirements Analysis

# Assignment Specification

Use Java/C# API to design and implement a client-server application for managing the consultations of doctors in a clinic. The application has three types of users: the clinic secretary, the doctors and an administrator.

# Functional Requirements

The clinic secretary can perform the following operations:

* Add/update patients (patient information: name, identity card number, personal numerical code, date of birth, address).
* CRUD on patients’ consultations (e.g. scheduling a consultation, assigning a doctor to a patient based on the doctor’s availability).

The doctors can perform the following operations:

* Add/view the details of a patient’s (past) consultation.

The administrator can perform the following operations:

* CRUD on user accounts.

In addition, when a patient having a consultation has arrived at the clinic and checked in at the secretary desk, the application should inform the associated doctor by displaying a message.

# Non-functional Requirements

• Secure access of confidential data.

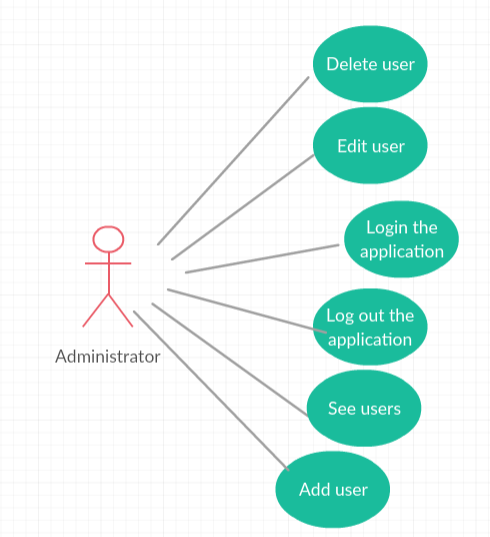
• 24X7 availability

• Better component design to get better performance at peak time

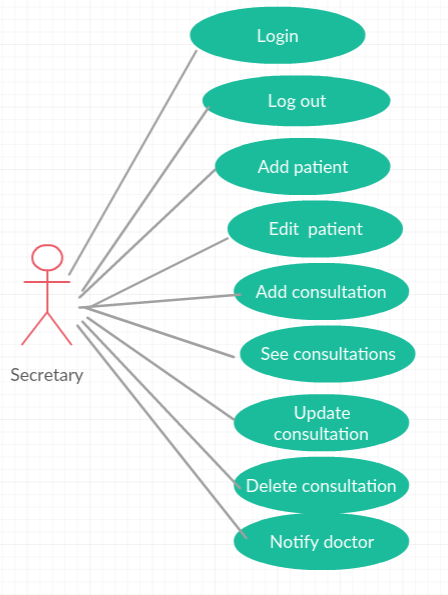
• Flexible service based architecture will be highly desirable for future extensions.

2. Use-Case Model

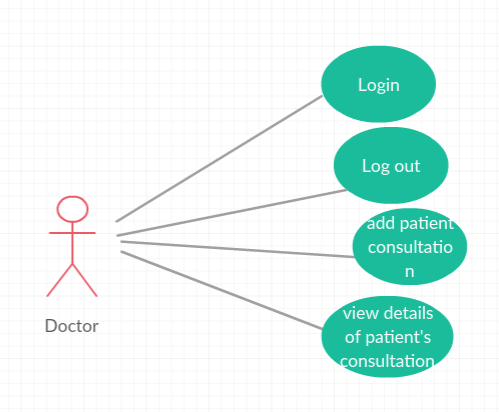
**2.1 Administrator use case diagram**



**2.2 Secretary use case diagram**

****

**2.3 Doctor use case diagram**



Use case: Add user

Level: summary level;

Primary actor: administrator;

Main success scenario: provide valid information for the new user;

Extensions: if invalid information is provided the user should not be saved.

Use case: Update user

Level: summary level;

Primary actor: administrator;

Main success scenario: search the user who’s information you want to edit and provide valid information for the new user;

Extensions: if invalid information is provided the user should not be saved.

Use case: Delete user

Level: summary level;

Primary actor: administrator;

Main success scenario: search the user you want to delete and press the delete button;

Extensions: if invalid information is provided no user should be deleted.

Use case: See users

Level: summary level;

Primary actor: administrator;

Main success scenario: a list with all the users should be provided;

Extensions:-;

Use case: Add consultation

Level: summary level;

Primary actor: secretary, doctor;

Main success scenario: provide valid information for the new consultation;

Extensions: if invalid information is provided the consultation should not be saved.

Use case: See consultation

Level: summary level;

Primary actor: doctor;

Main success scenario: the doctor should be able to select a patient and view the consultation list for that patient;

Extensions:-;

Use case: Update consultation

Level: summary level;

Primary actor: secretary;

Main success scenario: search the consultation who’s information you want to edit and provide valid information for the new consultation;

Extensions: if invalid information is provided the consultation should not be saved.

Use case: Delete consultation

Level: summary level;

Primary actor: secretary;

Main success scenario: search the consultation you want to delete and press the delete button;

Extensions: if invalid information is provided no consultation should be deleted.

Use case: See consultations

Level: summary level;

Primary actor: secretary;

Main success scenario: a list with all the consultations should be provided;

Extensions:-;

Use case: Add patient

Level: summary level;

Primary actor: secretary;

Main success scenario: provide valid information for the new patient;

Extensions: if invalid information is provided the patient should not be saved.

Use case: Update patient

Level: summary level;

Primary actor: secretary;

Main success scenario: search the patient who’s information you want to edit and provide valid information for the new patient;

Extensions: if invalid information is provided the patient should not be saved.

Use case: Notify doctor

Level: summary level;

Primary actor: secretary;

Main success scenario: notify doctor if a patient arrives at the consultation;

Extensions: -;

3. System Architectural Design

**3.1 Architectural Pattern Description**

The project will be developed in a layered architecture pattern used in the majority of spring web applications. It respects the principles of Separation of concerns and The Keep It Simple Stupid (KISS) principle. The layers of the application will be:

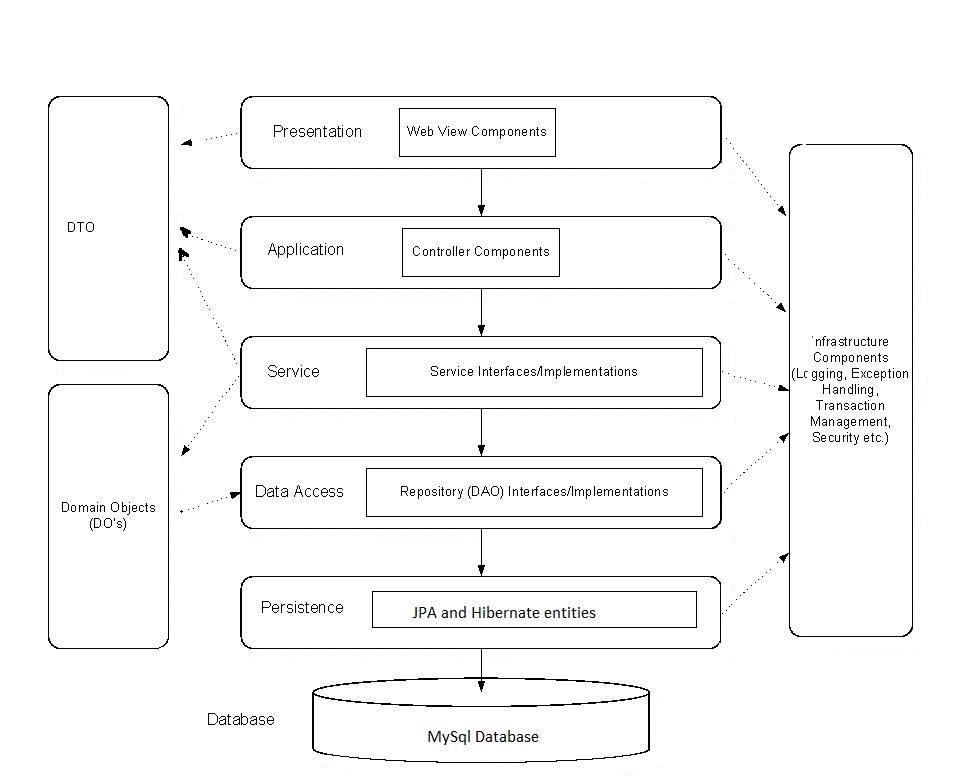
* The repository layer is the lowest layer of a web application. It is responsible of communicating with the used data storage.
* The service layer resides below the web layer. It acts as a transaction boundary and contains both application and infrastructure services. The application service provides the public API of the service layer. They also act as a transaction boundary and are responsible of authorization. The infrastructure services contain the “plumbing code” that communicates with external resources such as file systems, databases, or email servers. Often these methods are used by more than a one application service.
* The web layer is the uppermost layer of a web application. It is responsible of processing user’s input and returning the correct response back to the user. The web layer must also handle the exceptions thrown by the other layers. Because the web layer is the entry point of our application, it must take care of authentication and act as a first line of defense against unauthorized users.

The benefits of this kind of architecture is that it is a low coupling architecture, it offers the possibility of improving performance (sometimes).The maintenance of your application is easier because of the low coupling between layers. Adding more functionality to your application is made easier. Layers make your application more testable.

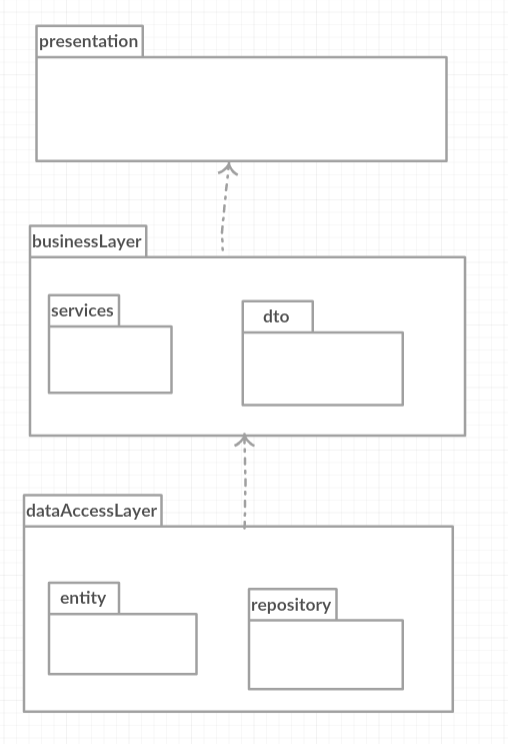
**3.2 Diagrams**

3.2.1 Conceptual architecture design

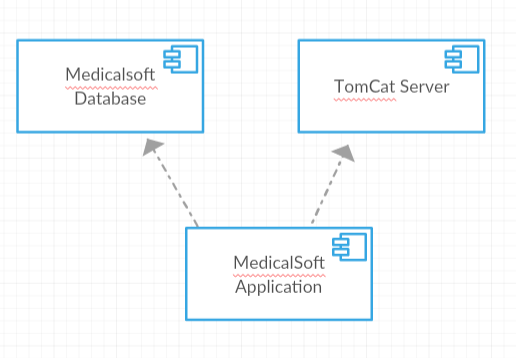
In the figure it can be seen how the architecture is structured on layers: presentation layer, controller layer, service layer, data access layer, database layer.



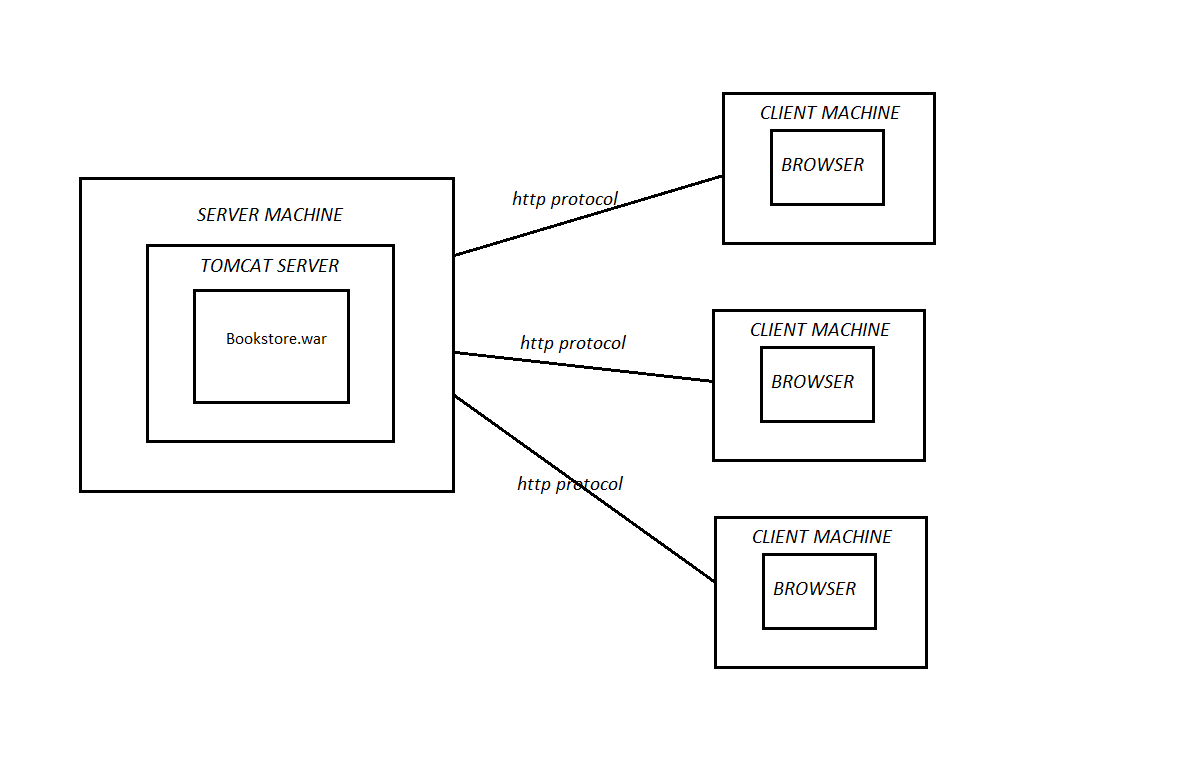
3.2.2 Package diagram



3.2.3 Component diagram

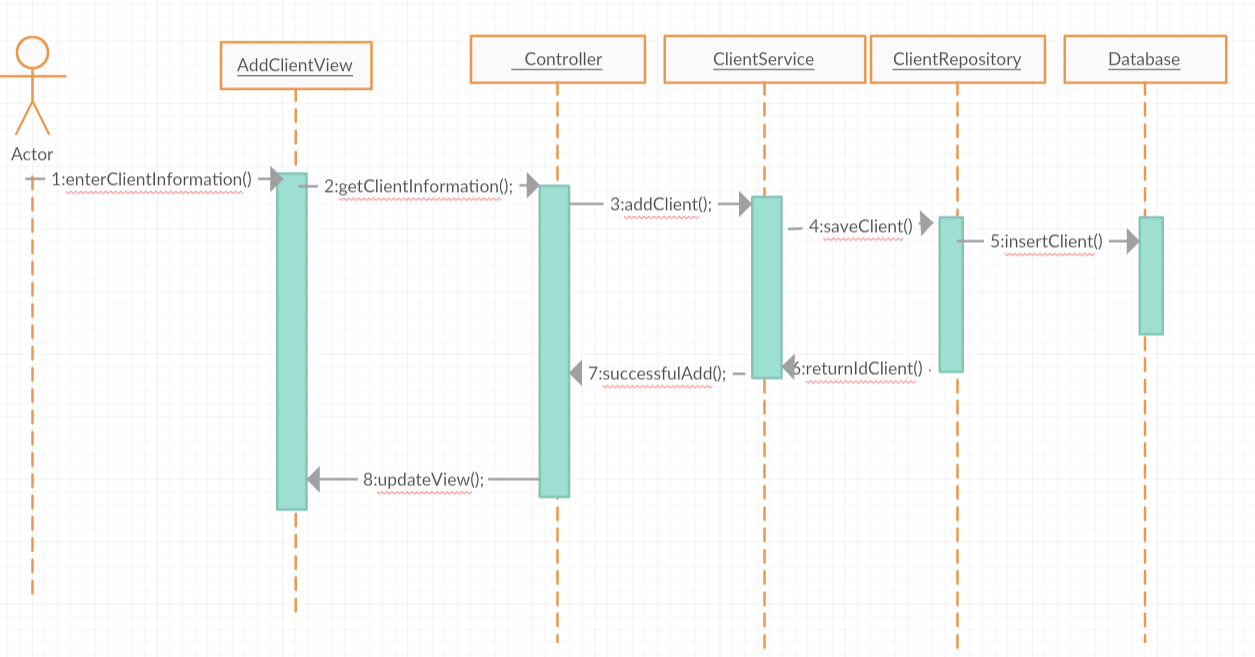


3.2.4 Deployment diagram

**

4. UML Sequence Diagrams

Add patient/client sequence diagram

**

5. Class Design

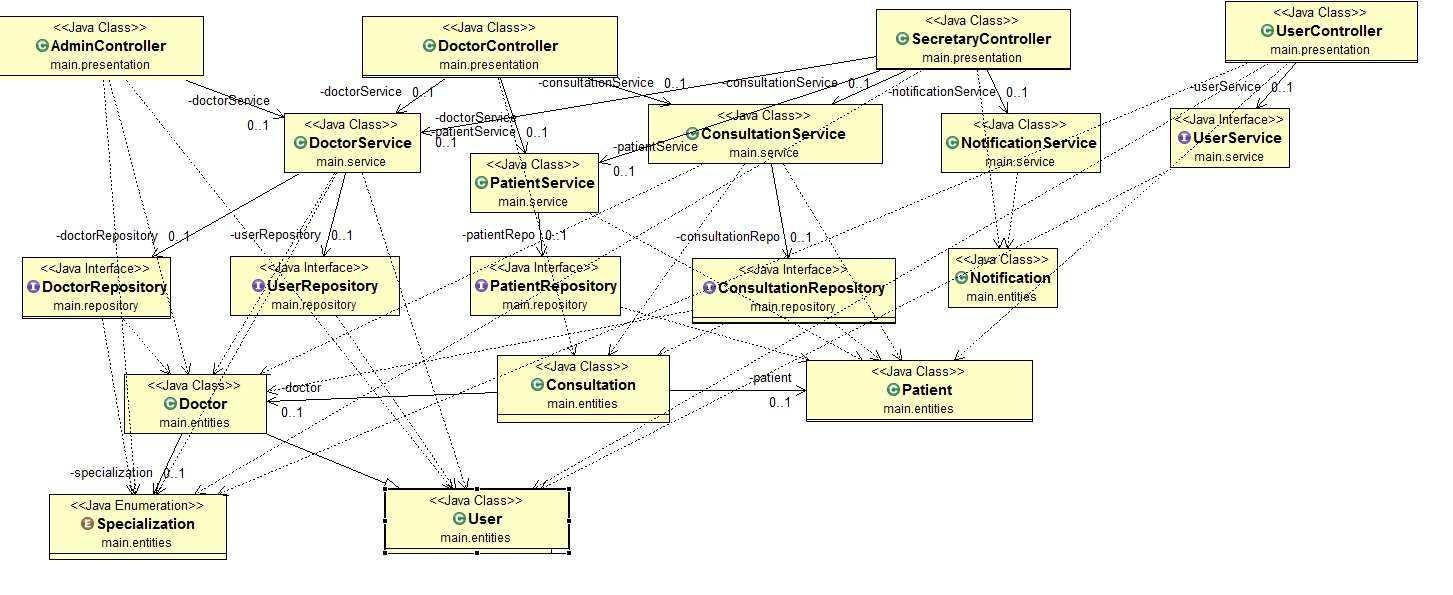
**5.1 Design Patterns Description**

For developing the application I used as some design patterns for each layer, some of them are used due to the frameworks in which the application is made. For the data access level I used the Spring Hibernate framework which uses the Data Model design pattern and Proxy Pattern. At runtime, Hibernate dynamically generates proxies from the entity code through bytecode generation. These proxies are aware of the ORM functionality and implement the object lifecycle, while pretending to belong the class that the programmer created.

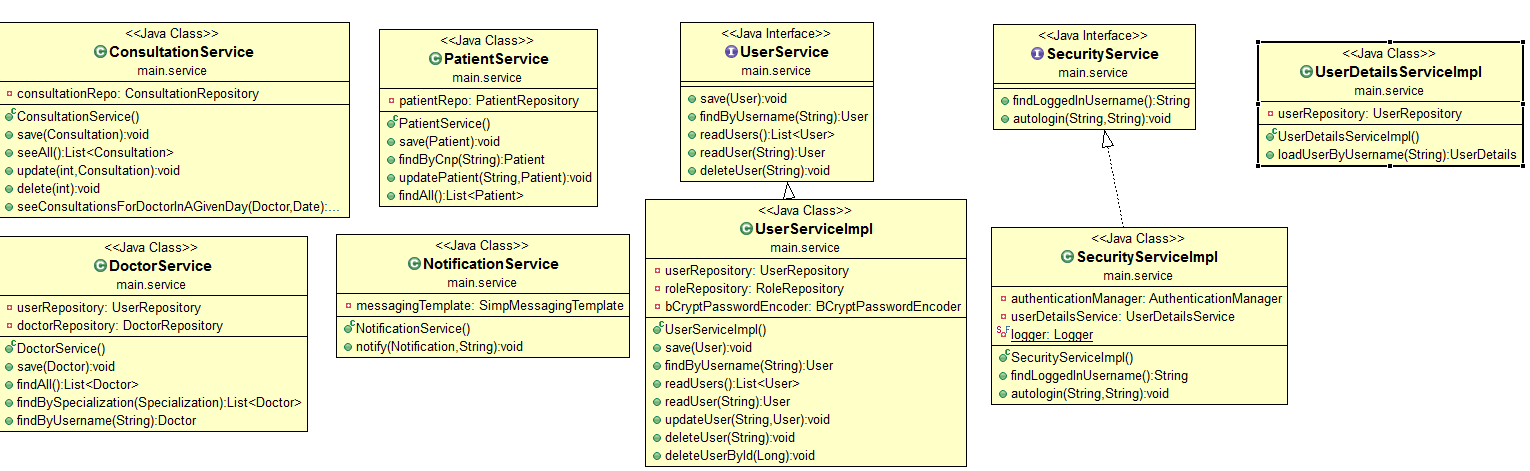
The business logic is created using the Services layer pattern, services for each domain model entity, they include all the needed logic to provide information for the user information and for saving into the database.

The presentation layer is created using MVC pattern. The Model handles the state of the application. The View is the representation of the user interface. User actions on the View are sanded to the Controller.

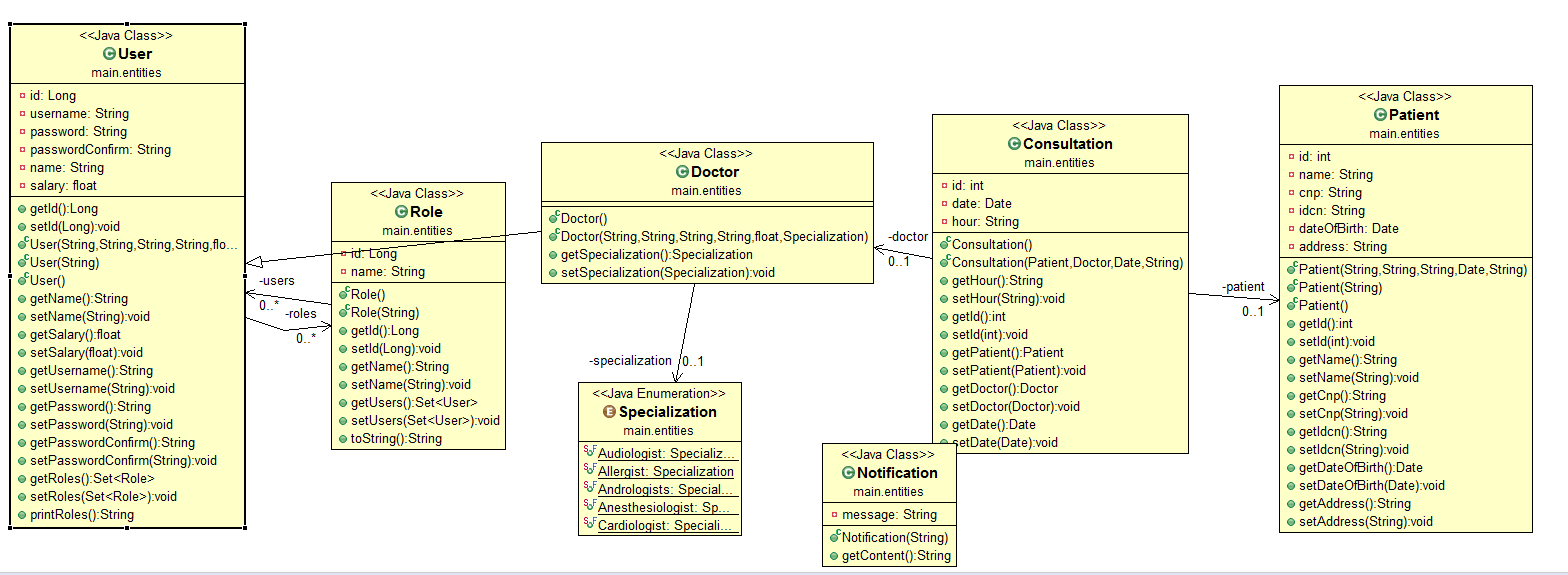
**5.2 UML Class Diagram**

****

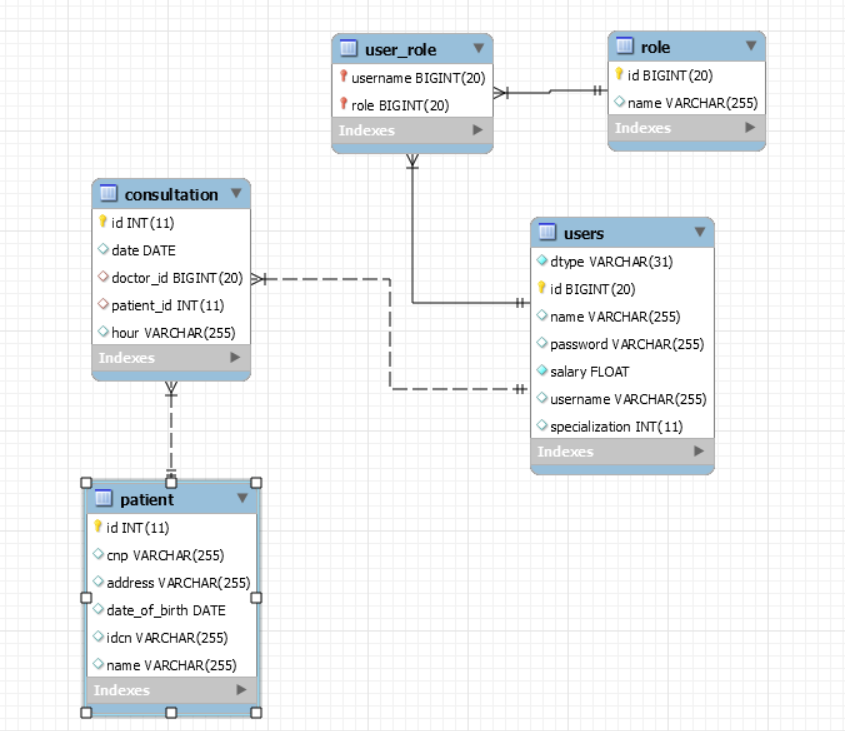
Services classes:



Entities:



6. Data Model

**

7. System Testing

For testing the system the following strategies were used: unit testing, validation testing and testing methods like boundary analysis. The system contains unit tests for the important methods from the services of the application.

Validation testing is made in the business logic of the application by testing each user input to have correct and valid values, if invalid values are placed then an error message which communicates the error is shown. The validation methods contain CNP, identity card number, salary validation, saving username and password validation.

Example: Login Test Scenario

* Black box

-Verify that the login screen is having option to enter username and password with submit button and option of forgot password.

-Verify that user is able to login with valid username and password.

-Verify that user is not able to login with invalid username and password.

-Verify that validation message gets displayed in case user leaves username or password field as blank

-Verify that the password is in encrypted form when entered

-For security point of view, in case of in correct credentials user is displayed the message like "incorrect username or password" instead of exact message pointing at the field that is incorrect.

-Verify if the password can be copy-pasted or not.

* White box testing

-Test method which tests the login method from the UserService. The test tests the validity of the username and password, the existence in the db and retrival of correct messages.

8. Bibliography

* <https://zeekat.nl/articles/mvc-for-the-web.html>
* <http://stackoverflow.com/questions/2637114/what-are-the-benefits-of-an-n-layered-architecture>