VIA University College

VIA UNIVERSITY COLLEGE

ICT ENGINEERING

Project Description

Database for a hospital

Group Members

Georgi Jivkov (254772)

Jesper Kay Wohlgemuth (217471)

Raitis Magone (254127)

Course: Semestrial Project 2

VIA University College

Abstract

This project involves a detailed introduction through the development of the “Hospitals” new patient and treatment organizations system, which belongs to the “Hospital”. The project lasted 3 weeks. The process started with analyzing how the hospital is handling their patient and patient treatment information and how it could be improved. Then there are the features that would not be added and what methods were used to implement the desired functions. The next part consists of the functional and non-functional requirements. After that there is an activity diagram for logging in, use cases and design patterns that were used. Next is the GUI description / User Guide. The last part includes a few snippets of the code which was used to implement the system and a small description of what the code is for.

VIA University College

Contents

VIA University College

Project Description

Background description

Some hospitals abroad do not have an advanced system of keeping track of their patient information, which includes if they are sick, if they need prescription medical treatment, have they got their meds and of course exact times when they come in and have been prescribed medical care. This can lead to some tampering with the prescriptions

Proper logging – This is one of the main reasons doctors/nurses might lose or have false information. If they just log their patient info on a sheet of paper that they just put somewhere aside it might go in a wrong drawer or be mistaken for trash and get thrown out and of course they can not remember all the information about all of their patients. This can lead to small problems where the patient is sick for only a few more days or it might happen to someone who needs their medication or they could get hospitalized.

Purpose

The purpose is to create a user-friendly program that can easily edit data from the system. The issues mentioned above could be solved with an effective way of keeping track of the patient information without application.

Here we are implementing a simplified version of a system that already exists in Denmark. By creating this we will be exploring how it was made and what improvements could we possible make to the existing system

Problem formulation

The questions we set out to answer are:

- How do we sort all the data and have a user-friendly interface?

- How do we give different privileges to different types of users?
- What specific data does different types of users need?
- What does the different access tiers entail?

Delimitation

- The database should not provide a lot of information about the patients, Only the information most relevant for their treatment should be stored.
- Each user should have belong to a clearly delimited access group

Choice of model and method

What	Why	Which
How to keep data organized more easy	Sorting the information correctly and only displaying the required information for each user will improve everyone's situation	Design patterns, UML class modelling
How to grant different privileges to different users	Making the database safer and ensure no problems when someone changes information	SQL

Requirements

Requirements are based on analyzing the purpose and the product backlog, that is how we discovered the following aspects:

Functional requirements:

1. Add patient
2. Edit patient
3. Delete patient
4. Add Medical Personnel
5. Delete Medical Personnel
6. Find Patient
7. Add treatment
8. View records
9. Add prescription
10. View prescriptions
11. Access tiers

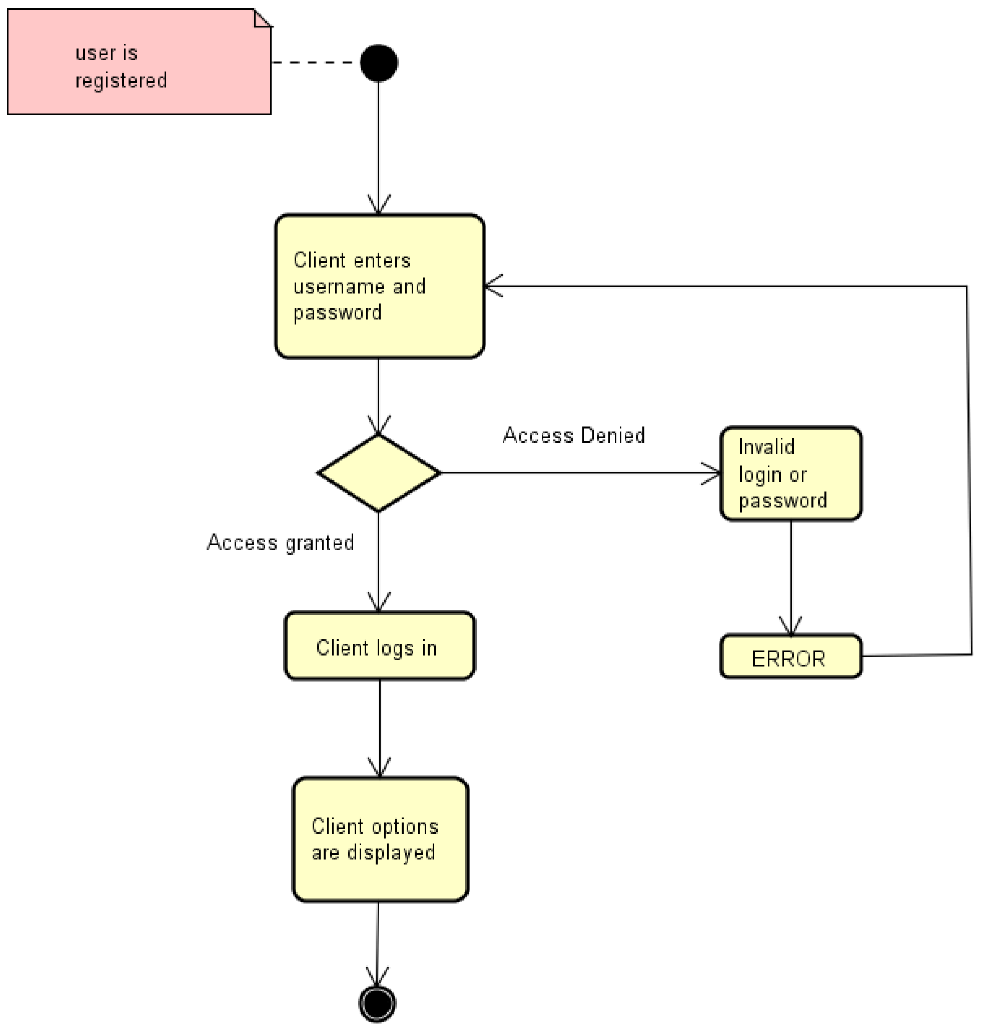
Non-Functional Requirements:

1. A SQL Database must be used
2. Must be developed in Java
3. Must be developed with client/server architecture
4. Should be developed in an agile fashion
5. It should have a GUI

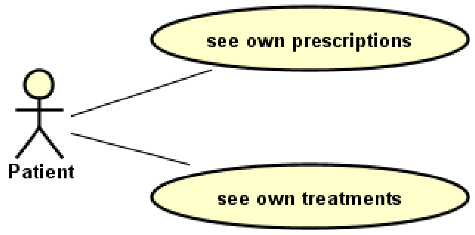
Showcasing Design

Activity Diagram for login

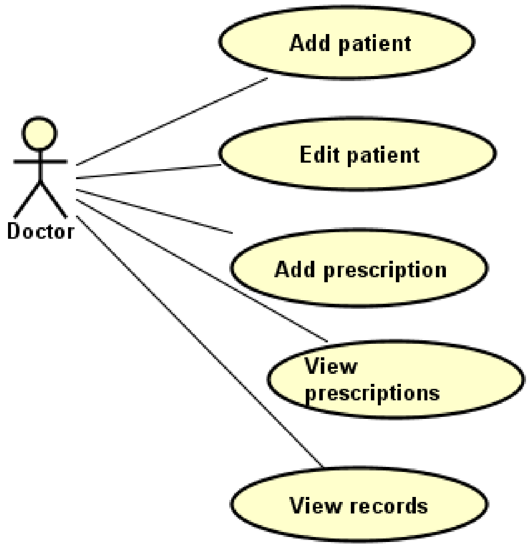
With the following figure we show the sequence of events that happens whenever someone opens the program.


user is registered
Client enters username and password
Access Denied
Invalid login or password
Access granted
Client logs in
ERROR
Client options are displayed

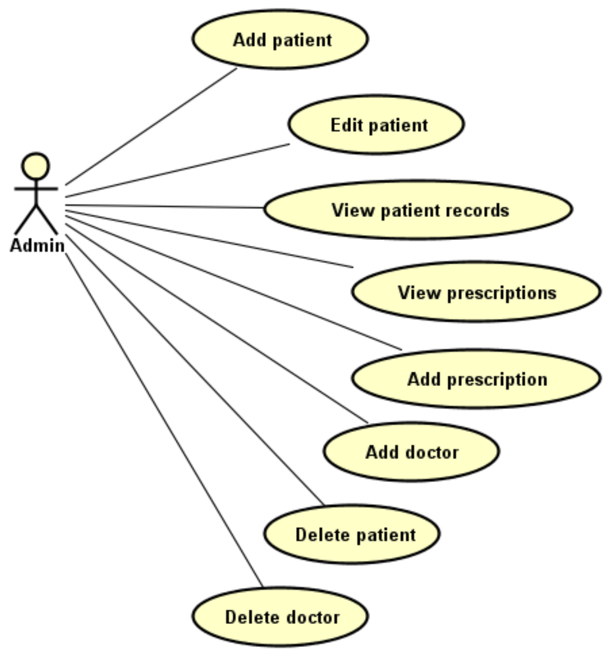
Use cases explained



Since the program is intended to deal with sensitive information 3 tier access is created. Therefore each patient can only see information related to themselves.



Medical Personnel - Doctor in this case, has to deal with multiple patients and prescriptions, therefore they need more options than a patient. When the doctor uses view prescriptions and view patient info they are asked for a cpr number, so they are not overwhelmed with every single record stored in the database but only the ones pertaining to the patient in question.

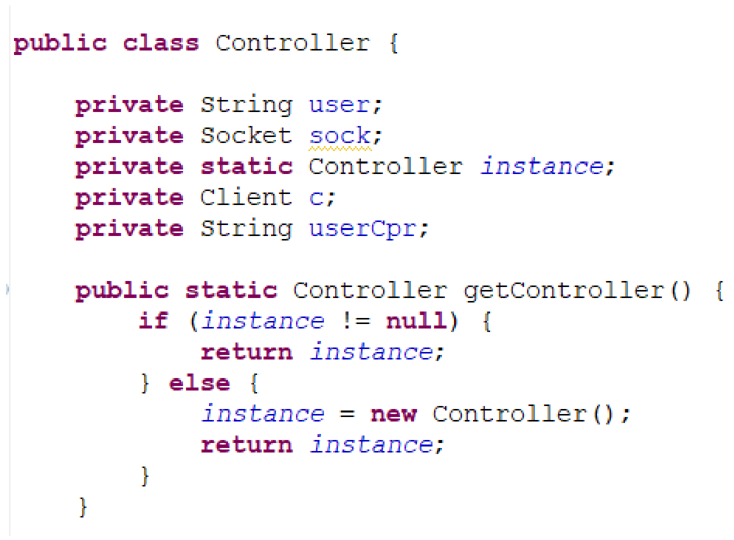


The admin has all the same privileges as the doctor. They also have the option to add or delete doctors. The reason for the admin being the only one to have deletion permissions is twofold:

1. Public offices have to keep any info for longer than 7 years.
2. We expected the admin to be a more tech savvy person, so we expect few accidental deletions.

Design Patterns

Singleton



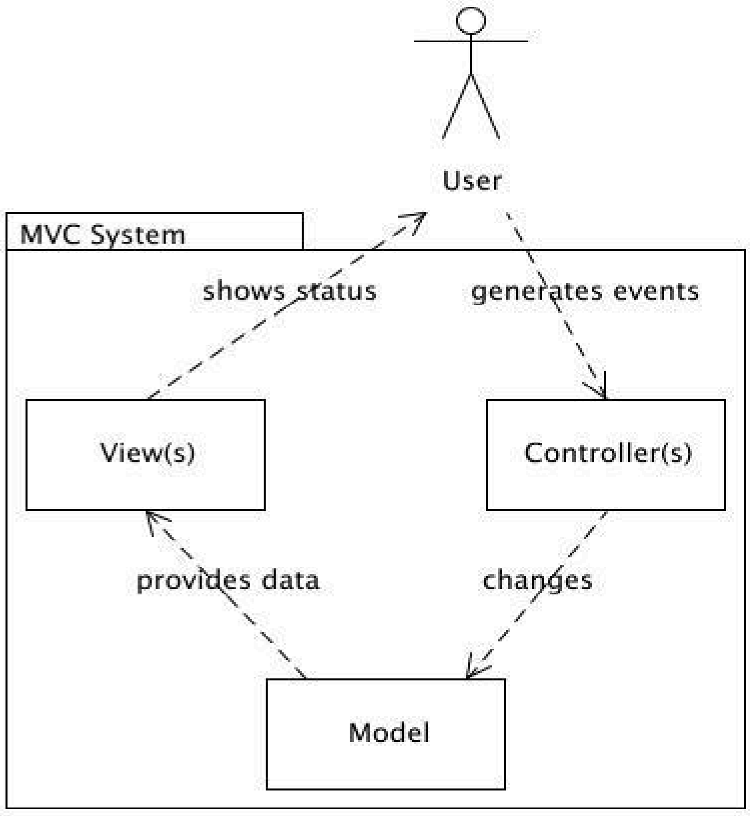
```
public class Controller {

    private String user;
    private Socket sock;
    private static Controller instance;
    private Client c;
    private String userCpr;

    public static Controller getController() {
        if (instance != null) {
            return instance;
        } else {
            instance = new Controller();
            return instance;
        }
    }
```

The program is designed in a way that there is only one controller intended for each client. Learning from our mistake of accidentally creating 2 controllers, we decided that we should use the singleton design pattern. This patterns makes it impossible to create more than one instance. If you attempt to create a second controller in simply returns the existing one.

“The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; in the same time it provides a global point of access to that instance. In this case the same instance can be used from everywhere, being impossible to invoke directly the constructor each time.”[1]

[1] *https://www.oodesign.com/singleton-pattern.html*

Model-View-Controller



To make the code easy to read and to edit, we used Model-View-Controller design pattern. This pattern creates a clear separation between the database and GUI and the base classes. Hence The name, Model-View-Controller. This clear separation makes it easier to debug and create fixes because of the clear responsibility separation. Since the design pattern is done over a multitude of classes we can't show an exact code snippet where we show our implementation.

"MVC proposes three types of objects in an application, the Model, Views and Controllers. These objects are separated by abstract boundaries which makes MVC more of a paradigm rather than an actual pattern since the communication with each other across those boundaries is not further specified. How

the objects inside MVC communicate differs not only by the type of application you are describing (GUI, web) but also by which part of the application you are currently looking at (frontend, backend).”[2]

Modified Front controller pattern

“The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers. ”[3]

We use a modified version without the dispatcher because we handed off that responsibility to the login “screen”. because of this our login screen functions as both dispatcher and front controller. Since we don’t need to switch between the different “screens” it was not a necessity to have a full dispatcher.

[2] http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/mvc.html
[3] *https://www.tutorialspoint.com/design_pattern/front_controller_pattern.htm*

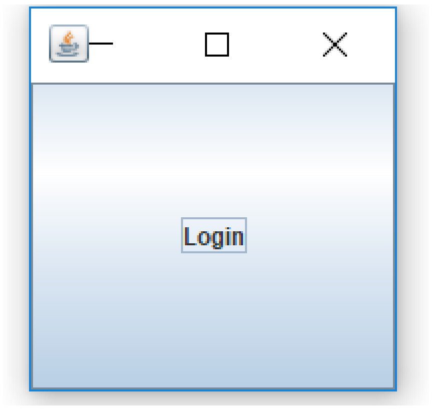
Gui Description / User Guide

For setup first run the SQL script provided named:

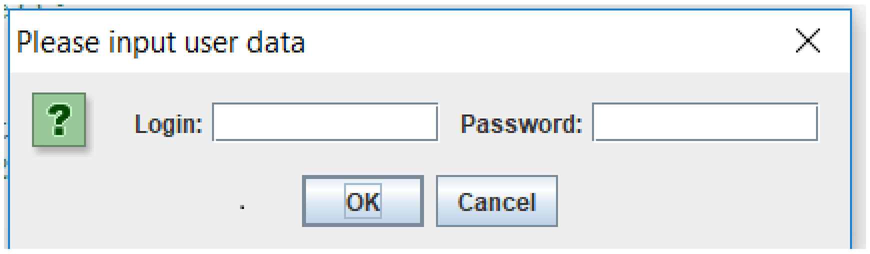
Then start the server by running RMI.java in the package RMI

To connect run clientMain.java in the package Client

A login button will appear



To continue click the "Login" button. Next a Login screen will appear

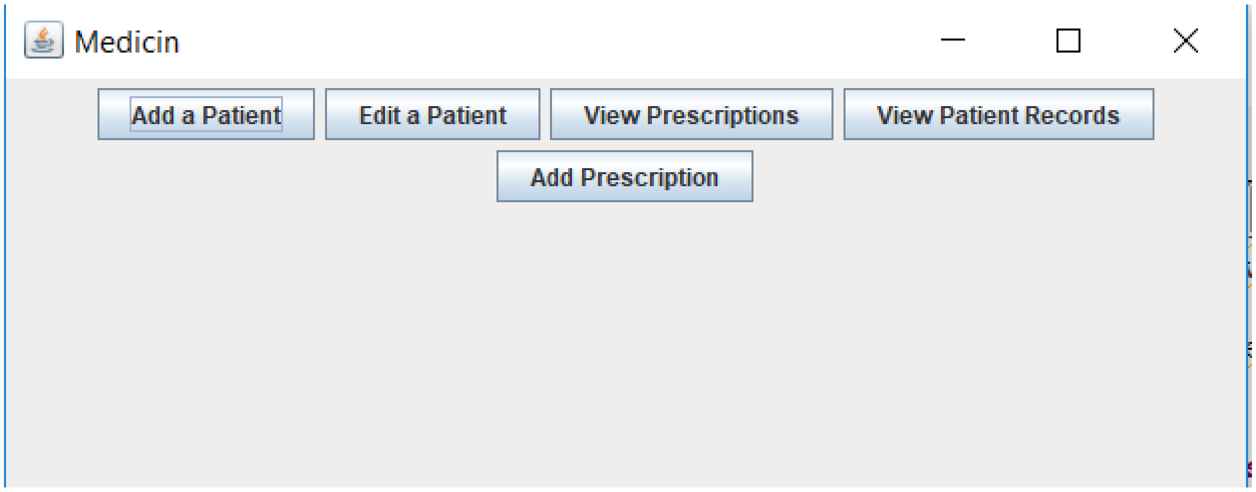


Please input user data

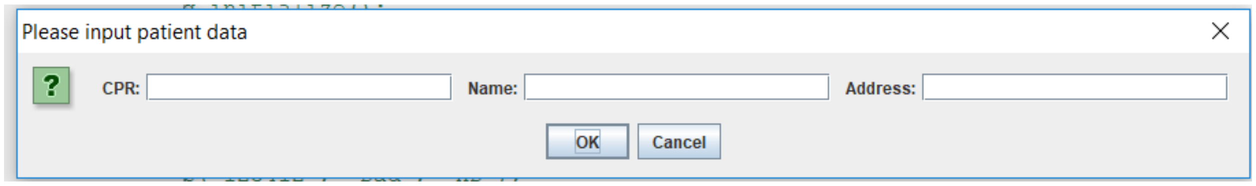
Login:

Password:

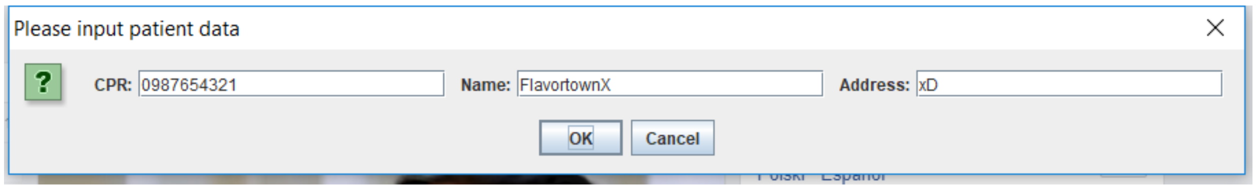
In login input your username. In most cases it should be equal to your ID or CPR. Then input your password. To continue press OK. The program will check if your login and password is correct and give you access if it is.



Options for client are displayed. The titles for the buttons are self explanatory for their functionality. The window presented is what the medical personnel would see as their options. If you press Add a Patient the following screen will open.



In here you must input CPR, Name and address of the patient. The program expects the CPR, therefore if you have anything else than a 10 digit number, the program will throw an error

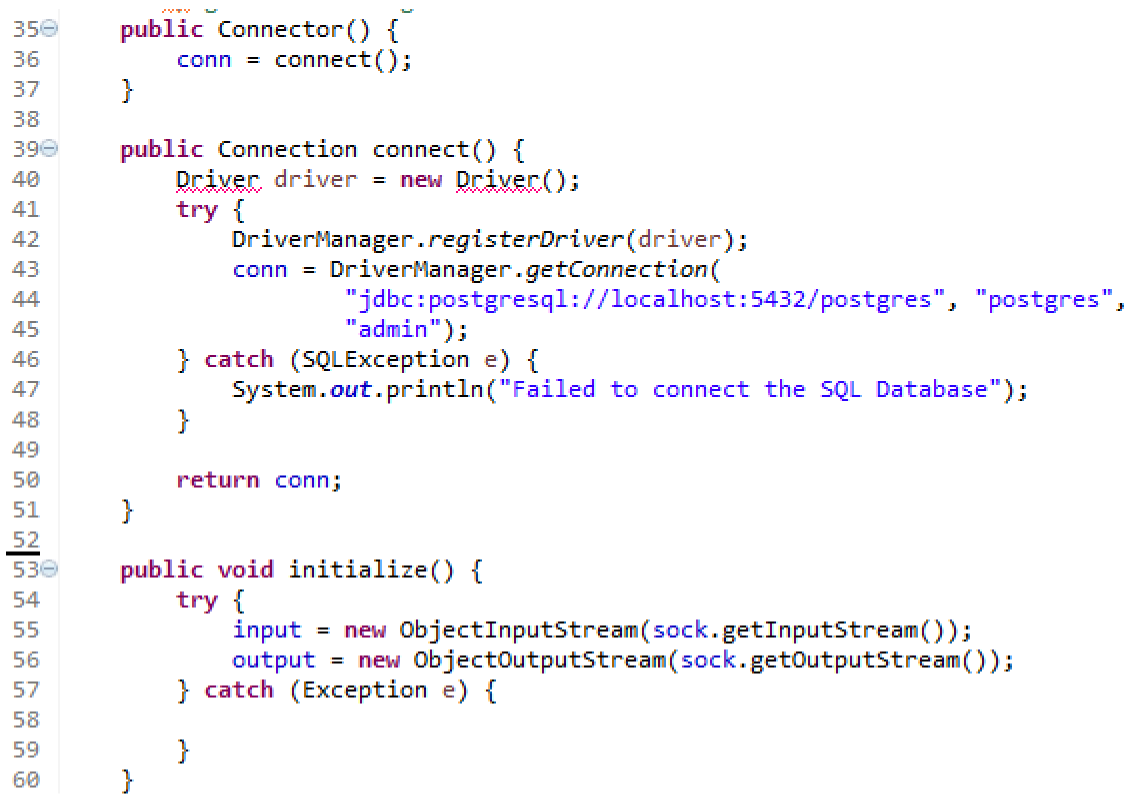

Please input patient data

CPR: 0987654321 Name: FlavortownX Address: xD

If the patient is added successfully the screen closes and the patient is added.

Implementations

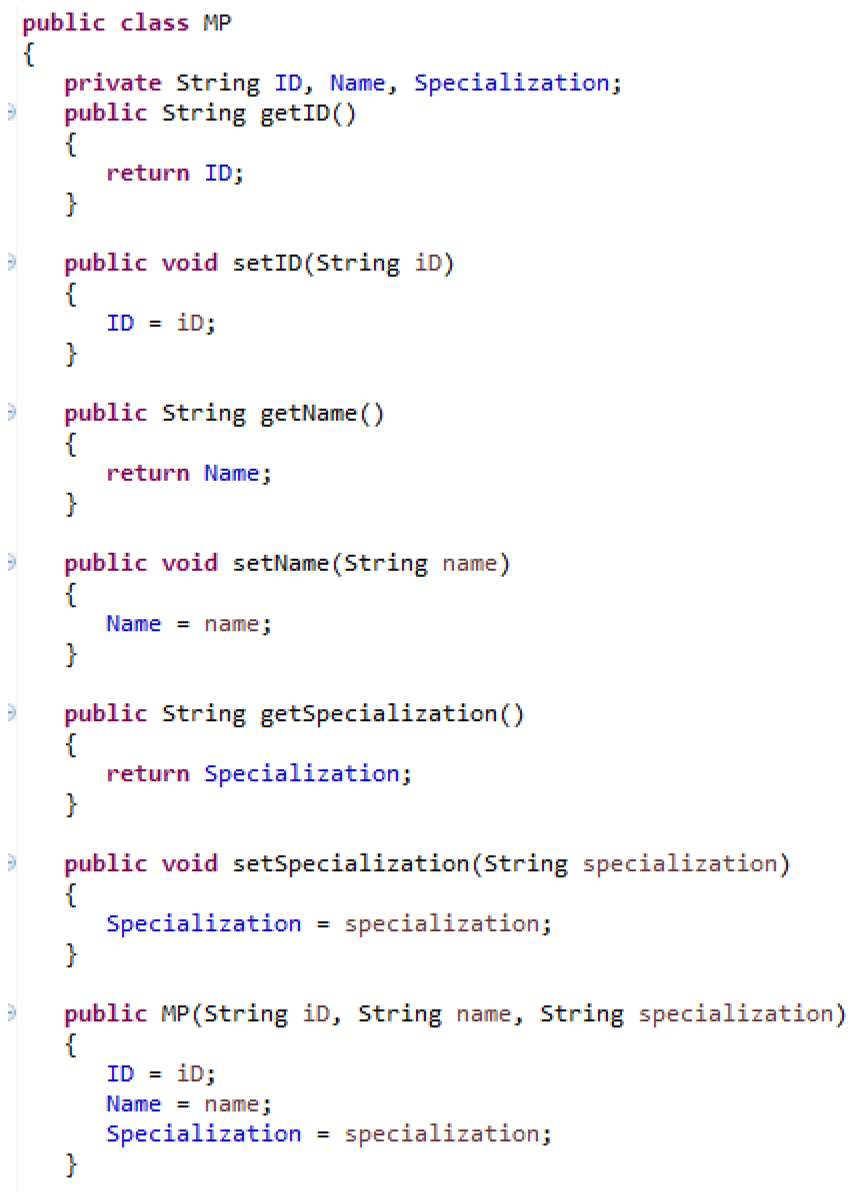
Connector



```java
    public Connector() {
        conn = connect();
    }

    public Connection connect() {
        Driver driver = new Driver();
        try {
            DriverManager.registerDriver(driver);
            conn = DriverManager.getConnection(
                    "jdbc:postgresql://localhost:5432/postgres", "postgres",
                    "admin");
        } catch (SQLException e) {
            System.out.println("Failed to connect the SQL Database");
        }

        return conn;
    }

    public void initialize() {
        try {
            input = new ObjectInputStream(sock.getInputStream());
            output = new ObjectOutputStream(sock.getOutputStream());
        } catch (Exception e) {

        }
    }
```

The Connector class is used to make a connection between the program and its database.

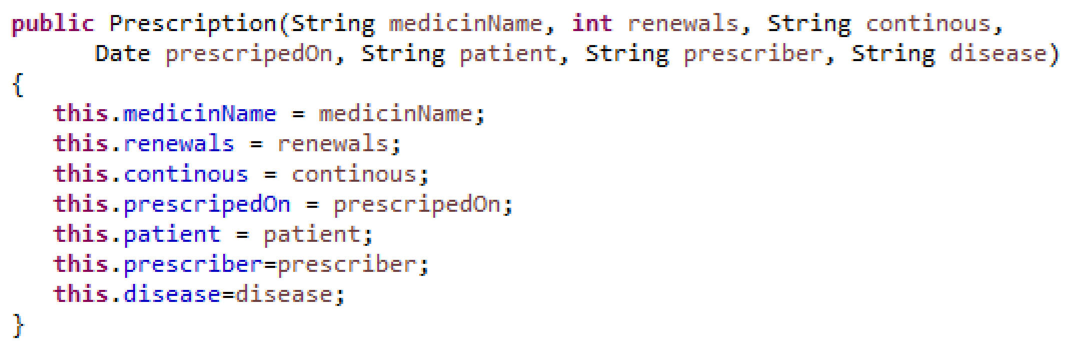
Medical Personnel



```
public class MP
{
   private String ID, Name, Specialization;
   public String getID()
   {
      return ID;
   }

   public void setID(String iD)
   {
      ID = iD;
   }

   public String getName()
   {
      return Name;
   }

   public void setName(String name)
   {
      Name = name;
   }

   public String getSpecialization()
   {
      return Specialization;
   }

   public void setSpecialization(String specialization)
   {
      Specialization = specialization;
   }

   public MP(String iD, String name, String specialization)
   {
      ID = iD;
      Name = name;
      Specialization = specialization;
   }
```

this is our class to represent our MP database object.

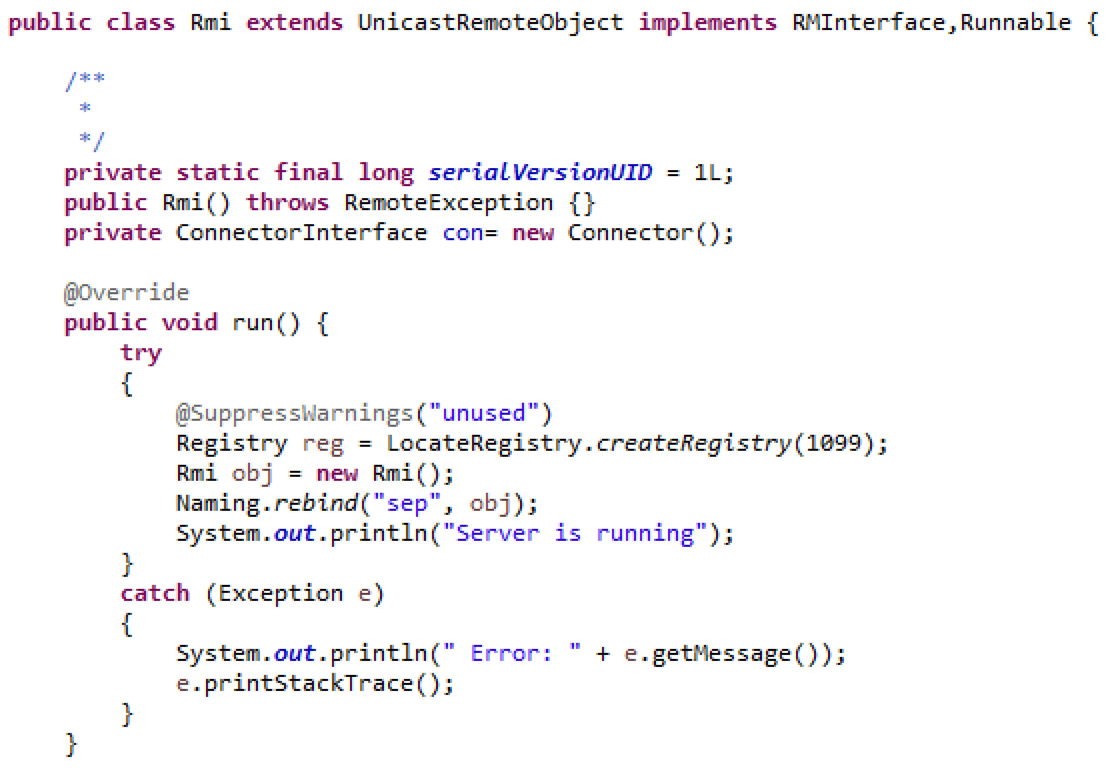
Prescription



```java
public Prescription(String medicinName, int renewals, String continous,
      Date prescripedOn, String patient, String prescriber, String disease)
{
   this.medicinName = medicinName;
   this.renewals = renewals;
   this.continous = continous;
   this.prescripedOn = prescripedOn;
   this.patient = patient;
   this.prescriber=prescriber;
   this.disease=disease;
}
```

Prescription is a very important part of treating a patient so we had to decide what is the most important things about it and these are the parameters we came up with : Name of the needed medicine, how many time you can renew it, whether or not the doctor needs to continuously prescribe this or they just prescribe it the once, the date the medicine was prescribed on, its patient, the medical personnel who prescribed it and last the disease it is intended to treat.

RMI



```
public class Rmi extends UnicastRemoteObject implements RMInterface,Runnable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;
    public Rmi() throws RemoteException {}
    private ConnectorInterface con= new Connector();

    @Override
    public void run() {
        try
        {
            @SuppressWarnings("unused")
            Registry reg = LocateRegistry.createRegistry(1099);
            Rmi obj = new Rmi();
            Naming.rebind("sep", obj);
            System.out.println("Server is running");
        }
        catch (Exception e)
        {
            System.out.println(" Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

We use rmi to pass info from our client to our server then the server either sends or receives our info.

The next step

In this chapter we will describe what some of the possibilities we could add on to the program in the future. All of these were found as a result of our testing phase, our scrum meetings, and our continuous talks with the customer representative.

1. Adding buttons for creating logins in the program (right now credentials are added through sql inserts)
2. Editing of of both prescriptions and treatment records.
3. Fine tuning of the gui
4. create the ability to generate statistics from some of our records, like how many were treated for which disease.

VIA University College

Requirements Test

Requirement	Implementation	Test
A patient can be added.	+	+
A patient can be edited.	+	+
A patient can be deleted.	+	+
Medical personnel can be added	+	+
Medical personnel can be edited	-	-
Medical personnel can be deleted	+	+
A patient can be found	+	+
A treatment can be added	+	+
Records can be viewed	+	+
Prescriptions can be added	+	+
Prescriptions can be viewed	+	+
The users must have different access tiers	+	+