

# Project Report

---

## RemoteControlRelay

**Group #10**      *Stanislav Ondruš (224378)*  
*Ionela Marinuta (226484)*  
*Slavomír Šimko (224433)*  
*Valeriu Arhip (224165)*

**Course**            *SEP I2*

**Supervisors**      *Irena Valerieva Asenova*  
*Mona Wendel Andersen*

**Date**                *31 May 2015*

# Abstract

---

*This report documents each step taken from the first analyses to the final tests done for the completion of the Remote Control Relay System. It starts with an introduction which provides some background description and specifies the system's purpose and relevance. In the Analysis section, the functional and non-functional requirements can be found, as well as the use cases derived from them. The domain class model and use case/activity diagrams are also in this section. The next part of the report concerns the design aspect of the system. It includes a detailed design class model, defines the design patterns used and why, and describes how the database was designed. The design of the two Graphical User Interfaces is also explained here. The Implementation section of this report records how the designs were carried out. It generously details the steps taken to reach an MVC-architecture Client-Server system combined with Observer, Iterator, and Singleton design patterns. Following this, the Testing section consist of an overview of all the use cases and the tests conducted to determine whether or not they have been successfully completed in the system. Towards the end of the report, the Discussion focuses on the positive and negative aspects of the final product; which parts were implemented well and which parts still need improvement. Finally, the conclusion wraps up the previous sections and contains some final remarks on the project.*

# Table of Contents

---

Abstract.....	2
1. Introduction .....	5
2. Analysis.....	6
2.1 Requirements .....	6
2.1.1 Use case diagram .....	7
2.1.2 Use case summary.....	7
2.1.3 Use case description .....	8
2.2 Domain class model .....	8
2.3 Activity Diagram.....	10
3. Design.....	11
3.1 Design class model .....	11
3.2 Client-Server System: RMI .....	12
3.3 Design patterns.....	12
3.3.1 Model-View-Controller .....	12
3.3.2 Observer .....	13
3.3.3 Iterator .....	13
3.3.4 Singleton .....	13
3.4 Database design.....	14
3.5 GUI Design .....	15
3.5.1 Client GUI .....	15
3.5.2 Server GUI .....	16
3.6 Hardware.....	17
3.6.1 Relay.....	17
3.6.2 Arduino .....	18
3.7 System security.....	19
3.7.1 Database Password Encryption .....	19
3.7.2 Client Login Hash Function.....	19
4. Implementation .....	20
4.1 Client-Server System: RMI .....	20
4.2 Design patterns.....	22
4.2.1 Model-View-Controller .....	22
4.2.2 Observer .....	28
4.2.3 Iterator .....	31
4.2.4 Singleton .....	33
4.3 System security.....	34
4.3.1 Database Password Encryption .....	34
4.3.2 Client Login Encryption .....	35
5. Testing.....	36
6. Discussion .....	38
7. Conclusion.....	39
8. References .....	40

## LIST OF FIGURES

Figure 1 - Use Case diagram .....	7
Figure 2 - Use Case: Register User .....	8
Figure 3 - Domain Class Model .....	9
Figure 4 - Activity Diagram Register User .....	10
Figure 5 - Design Class Model .....	11
Figure 6 - Table History and Table User .....	14
Figure 7 - Client GUI Logged In.....	15
Figure 8 - Client GUI History .....	15
Figure 9 - Server GUI Main .....	16
Figure 10 - Relay (connected) .....	17
Figure 11 - Relay (disconnected) .....	17
Figure 12 - Arduino .....	18
Figure 13 - Encrypted Passwords in the Database .....	19
Figure 14 - RMI diagram.....	20

# 1. Introduction

---

Home automation systems, previously only a feature of science fiction writing, have been growing increasingly popular over the past few years. This is mostly due to greater affordability and simplicity through smartphone and tablet connectivity. These systems combine ease, security and energy efficiency to provide a comfortable environment to live in.

Top tier home automation systems allow control over lights, cameras, thermostats, door locks, alarm systems and the possibility to add intelligence to virtually all electronic devices found in a home. Systems such as the ones proposed by **Control4™** and **Vera™** are all-in-one solutions to cover the entire building, including indoor



and outdoor control. Since there are so many possibilities and levels of automation, a new user can start small, with a few climate control changes, and gradually expand to customize home security and entertainment systems.



It all starts with a controller, the “brain behind all the automation magic”, and each company offers a range of them, depending on the customer’s needs. **Vera™** offers controllers for basic and advanced users, as well as small businesses. An interface is then required to access the controller, usually using a mobile application, software program, or online account. **Control4™**, for example, permits the use of many interfaces, including a TV remote,

smartphone, tablet, or computer with internet access. Thus, not only can a user control the various components in their home from inside, but there is also the possibility of accessing the home remotely.

The **RemoteControlRelay** system is inspired by systems which companies like the ones mentioned above offer. Admittedly, it is much simpler and meant to demonstrate only the use of a client server system to gain remote access to some devices using a controller (the Arduino board, *Section 3.5.2*) and a few relays (*Section 3.5.1*).

The system is designed for a single household, and an account may be set up for each member, complete with customizable access rights. One person in the household will act as administrator and have access to the server side interface, where the options of adding, updating, deleting, and viewing user data are provided. The rest of the users will access the system through the client side interface, where they can login to their accounts and control devices according to their permissions. A database will store user data and their privileges, and keep track of all their actions in a history log.

Note that allowing for customizable access rights via separate accounts for each household member is a feature which sets this system apart from the others.

## 2. Analysis

---

### 2.1 Requirements

*Note: The admin has access to the server-side GUI, and as such, can control the communication link and other users' accounts (the admin can also be a user).*

#### **Functional:**

1. The admin must be able to add new accounts. The system must store a username, an encrypted password, and specific access rights for each account.
2. The admin must be able to update user information. The system must modify and save user passwords and access rights.
3. The admin must be able to delete user accounts.
4. The admin must be able to start and stop serial communication between the system and Arduino.
5. The user must be able to log in to the system using a predefined username and password.
6. The user must be able to log out of their account.
7. The user must be able to control (turn on or turn off) the available devices with the push of a button.
8. The admin must be able to view a list of current users registered in the system.
9. The admin/user must be able to view history, i.e. the list of actions taken place in the last 60 days.

#### **Non-functional:**

1. The system must use a client server architecture.
2. The system must store data in a MySQL database.
3. The system must use an Arduino circuit board.

## 2.1.1 Use case diagram

The requirements mentioned above are used for creating the use case diagram. This high-level interaction diagram shows what the system should be able to do. It will be used for presenting the functionality of the program to the Administrator and Client, shown in the diagram in *Figure 1*.

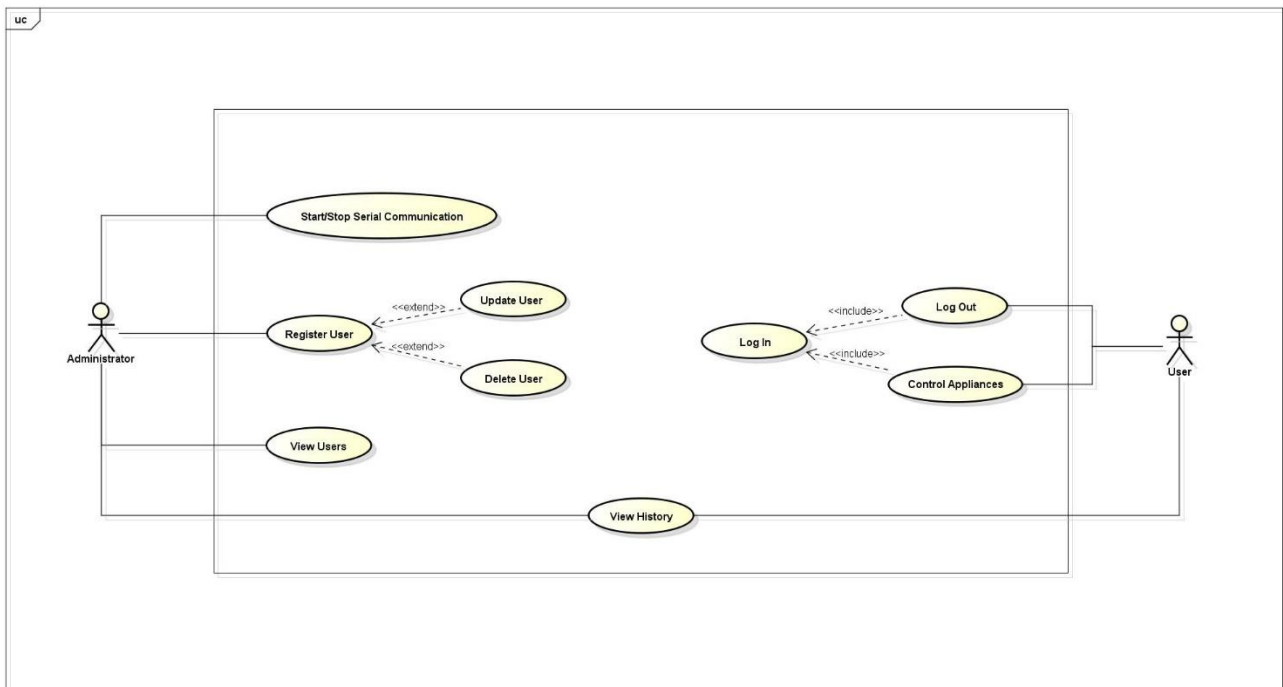


Figure 1 - Use Case diagram

## 2.1.2 Use case summary

The following list contains a short description of the actions each use case involves:

- **Log In:** The User should be able to log in to the system, without interference from other users already logged in.
- **Log Out:** The User should be able to log out from the system.
- **Control Appliances:** The User should be able to turn ON/OFF any device that is connected to the Arduino board.
- **View History:** The User and Administrator should be able to view a list of actions taken place in any selected range.
- **Start/Stop Serial Communication:** The Administrator should be able to start/stop serial communication between system and Arduino, by specifying the port number.
- **Register User:** The Administrator should be able to create new accounts that contain a legit username, password and access rights.
- **Update User:** The Administrator should be able to update User information, such as: password and access rights.
- **Delete User:** The Administrator should be able to delete existing Users.
- **View Users:** The Administrator should be able to view a list of current Users registered in the system.

## 2.1.3 Use case description

The use case diagram and summary provide a general overview of the functions of the system. In the following part, these functions will be described in further detail by examining action by action how the task will be performed, i.e. what the system does in response to actions performed by the Administrator/User.

### USE CASE: RegisterUser

<b>Name:</b>	RegisterUser
<b>Actors:</b>	Administrator
<b>Summary:</b>	The Administrator registers a new user into the system.
<b>Precondition:</b>	The user is not already registered in the system.
<b>Base sequence:</b>	<ol style="list-style-type: none"><li>1. The Administrator types in the user name.</li><li>2. The Administrator types in the user password.</li><li>3. The Administrator selects the access rights for user.</li><li>4. The Administrator presses <i>Register/Update</i> button.</li><li>5. The system adds the new user to the database.</li><li>6. The system displays an updated list of registered users.</li></ol>
<b>Exception sequence:</b>	<p><b>At 4.</b></p> <p>The admin did not type in a new username and/or password.</p> <p>System does not register any new user. System displays username and/or password fields with red background. Admin must try again.</p>
<b>Postcondition:</b>	A new user is registered in the system, with username, password, and access rights.

Figure 2 - Use Case: Register User

Figure 2 shows the use case description for **RegisterUser**. It describes how this function works under different conditions. The rest of the use case descriptions can be found in **Appendix A**.

## 2.2 Domain class model

The domain class model describes our system as a concept. It has no methods or solutions included. The only items included in this concept are classes and their expected relations. The domain model serves as a tool for setting the overall structure of the system.



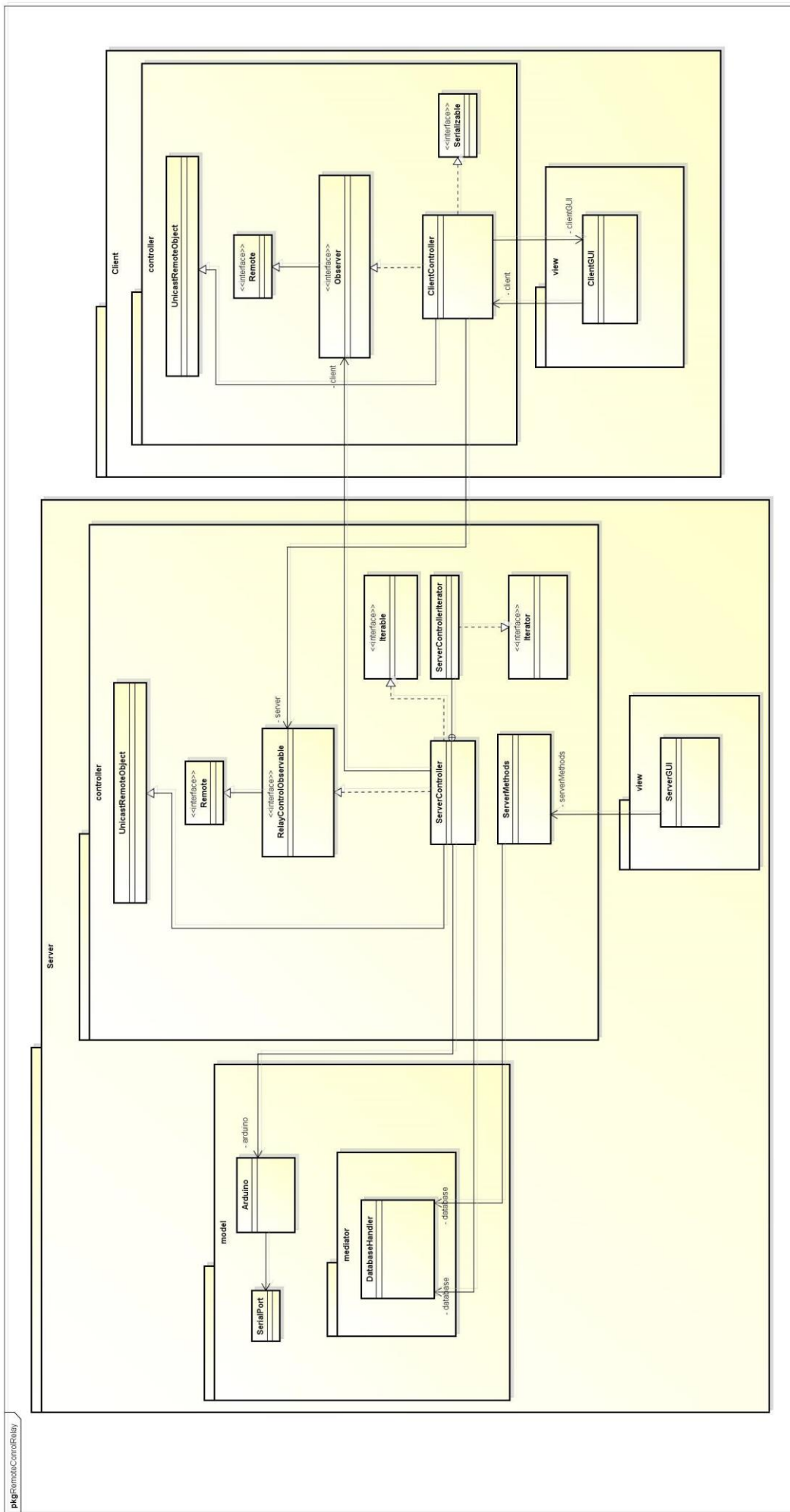


Figure 3 - Domain Class Model

## 2.3 Activity Diagram

The following activity diagram describes the use case **RegisterUser** and serves as a clear-cut, step by step illustration of how the use case works. Each text box represents an action taken by either the Administrator or the System, while the rhombus represents a decision node, where more outcomes are possible from one action.

The rest of the Activity Diagrams can be found in **Appendix B**.

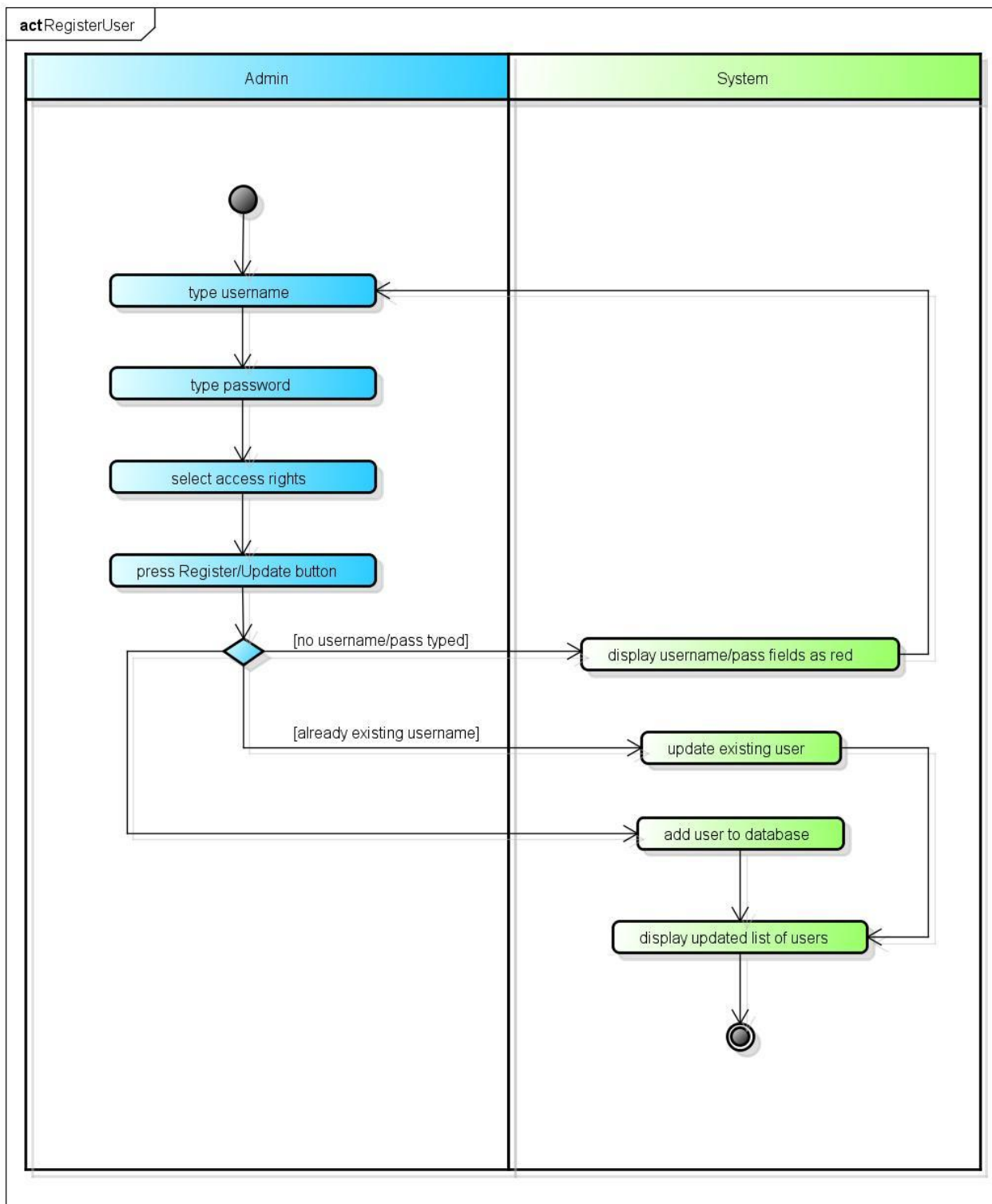


Figure 4 - Activity Diagram Register User



## 3.2 Client-Server System: RMI

To establish the client server connection, **Java Remote Method Invocation** is used in this system. **RMI** is a Java API which facilitates object function calls between Java Virtual Machines (JVMs). Thus, one JVM can invoke methods belonging to an object stored in another JVM.

### How RMI works:

- RMI uses a **network-based registry** to keep track of the distributed objects. The server object makes a **method** available for remote invocation by **binding it to a name in the registry**. The client object, in turn, can check for availability of an object by **looking up its name in the registry**. The **registry** acts as a limited **central management point for RMI**. The **registry** is simply a **name repository**. It does not address the problem of actually invoking the remote method.
- The two objects may physically reside on different machines. A mechanism is needed to transmit the client's request to invoke a method on the server object to the server object and provide a response. RMI uses an approach similar to RPC in this regard. The code for the server object must be processed by an RMI compiler called **rmic**, which is part of the JDK.
- The **rmic** compiler generates two files: a **stub** and a **skeleton**. The **stub** resides on the client machine and the **skeleton** resides on the server machine. The **stub** and **skeleton** are comprised of Java code that provides the necessary link between the two objects.
- When a **client** invokes a server method, the JVM looks at the **stub** to do *type checking* (since the class defined within the stub is an image of the server class). The request is then routed to the **skeleton** on the server, which in turn calls the appropriate method on the server object. In other words, the stub acts as a proxy to the **skeleton** and the skeleton is a proxy to the actual remote method.

## 3.3 Design patterns

The **RemoteControlRelay** system uses **4 design patterns** for conveniently organizing the code and facilitating the programming process. Design patterns provide simple and proven solutions to frequent programming problems. The following text describes the patterns chosen for this system, the reasoning behind it, and their purpose.



### 3.3.1 Model-View-Controller

The Model-View-Controller design pattern is created to split application concerns into separate packages, so as to make the code more readable and editable in the future.

It is not necessary to have a model on the client side, because the client runs only when the connection to the sever model is successful. The model on the server side is very specific. It has an indirect connection to the relays, through an Arduino board, a very flexible and easily programmable piece of hardware. The model is also connected to the database, through the database handler.

The MVC implementation includes two controllers, one for the server and one for the client. The server controller is further separated into two controllers: one containing methods accessible to the client controller (for the client view), and another containing methods exclusive to the server view. The purpose of the client controller is to allow users to manipulate the model, which then ensures that all necessary changes are made to the view.

Initially, there was some confusion about implementing this pattern on both the client and server side, specifically the model package, but a written email to **Ralph E. Johnson**, author of the textbook *Design Patterns: Elements of Reusable Object-Oriented Software* cleared up this problem.

This was his response.

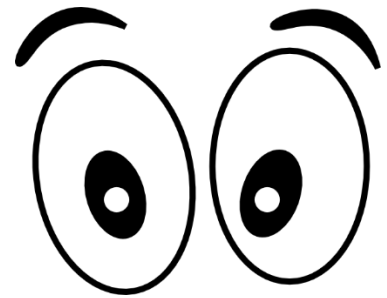
***"Just because you use a pattern like MVC in one part of a program doesn't mean you have to use it in another. In the original MVC, each is connected to each other. That is because the View is the observer of the Model. When the model changes state, the view updates without going through the controller. The controller only handles user events. When the model changes, it notifies all its views (observers) and they update themselves by communicating directly with the model.***

***Even if the second object does not have any other use inside the program. Even if it doesn't store any data.***

***But in fact, controllers usually store some data. They have a state. Controllers are often a state machine. They map user actions into operations on the model. Just like the view maps the state of the model into a picture on a display."***

### 3.3.2 Observer

The basic idea of this pattern is to update all observers (clients which need to be updated) when the subject is modified. Therefore, it is used in cases when there is a one-to-many relationship present. In this system, there are many clients and only one server. This pattern was decided on because of the need to have real time features in all client views. More specifically, intended for indicators of whether certain appliances are turned off or on. It is essential to present the information in real time because if, for example, one of several clients connected turns on a device, the others should be notified immediately of the updated state, via the interface indicators.



### 3.3.3 Iterator

This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

The Iterator is used in unison with the Observer pattern for traversing the list of clients and performing updates on each of them.

### 3.3.4 Singleton

The main goal of the singleton pattern is to ensure that there is only one instance of an object available. In this system there should be only one server interacting with many clients. This is the reason for including a singleton on the server side.

The implementation of this pattern consists of making the constructor private and writing another method which will create an instance of the object, or return the instance, in case it already exists. The pattern can be found in the server controller class.

## 3.4 Database design

The system requires a database that stores user accounts and activity logs. *Figure 6* illustrates the two tables which make up the database. The User table stores account information, such as *username*, *password*, and *access rights*, while the History table keeps track of user actions.

Table: History	
Name	Type
name	varchar(15)
port	varchar(2)
action	varchar(3)
date	datetime

Table: User	
Name	Type
name 	varchar(15)
pass	varchar(30)
controls	varchar(4)

Figure 6 - Table History and Table User

*History:*

- **Name:** the user name.
- **Port:** the number of the port which has been changed.
- **Action:** action performed by the user (ON/OFF).
- **Date:** the date and time when the action was performed.

*User:*

- **Name:** the user name (primary key).
- **Pass:** the password.
- **Controls:** the ports which the user can control.

Since critical user information is stored in the database (passwords), security measures need to be taken. The system is protected by one encryption algorithm which encrypts the password stored in the database, and one hashing algorithm which secures the password during the login process. Further explanations about system security can be found in *Section 3.7* and *Section 4.3*.

## 3.5 GUI Design

### 3.5.1 Client GUI

The client application window consists of two main parts: login and control. The left side is used for logging in and out of user accounts in the system, while the right side of the program provides user-friendly control with real-time port state indication. Port names are updated at each start of the client application. After a successful login, the buttons corresponding to the available ports for this individual user become clickable.

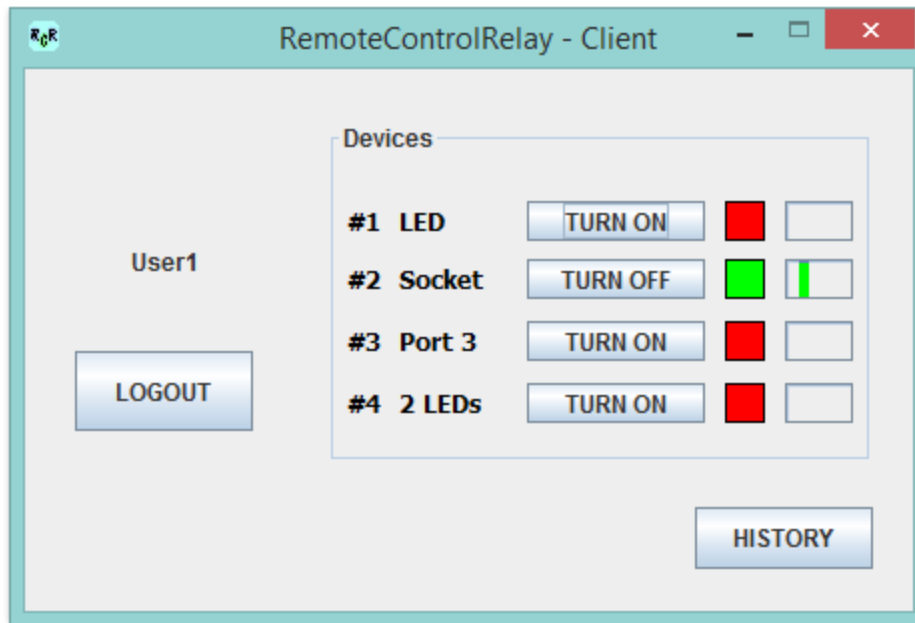


Figure 7 - Client GUI Logged In

#### History

The history panel can be accessed from both the server and the client application window, regardless of the user's login status. It contains a big scrollable, non-editable text field where all user activities are displayed. The data is organized into a table of actions, each one described by the *username*, *port* number whose state was changed, the *state* it was changed to, and the date and *time* the action was performed. It is also possible to show history in a specific date range by choosing or inputting the number of days it should cover into the combo box in the upper left corner. On the right side there are two buttons: one for refreshing the history and another to going back to the main view.

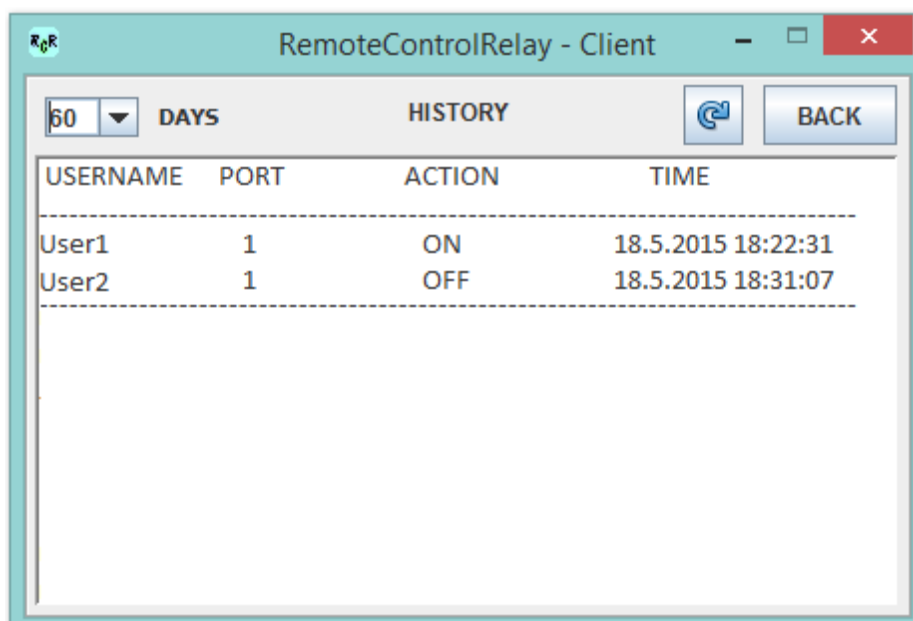


Figure 8 - Client GUI History

## 3.5.2 Server GUI

The server application window provides user-friendly interaction with the system. A panel with buttons for controlling the serial link is placed on the top side of the application window. Through this, the administrator is able to create the serial link and start/stop the communication with Arduino. The bottom part of the window consists of a section used to control all operations regarding the users (adding, deleting, viewing and updating), as well as a button to display all user history.

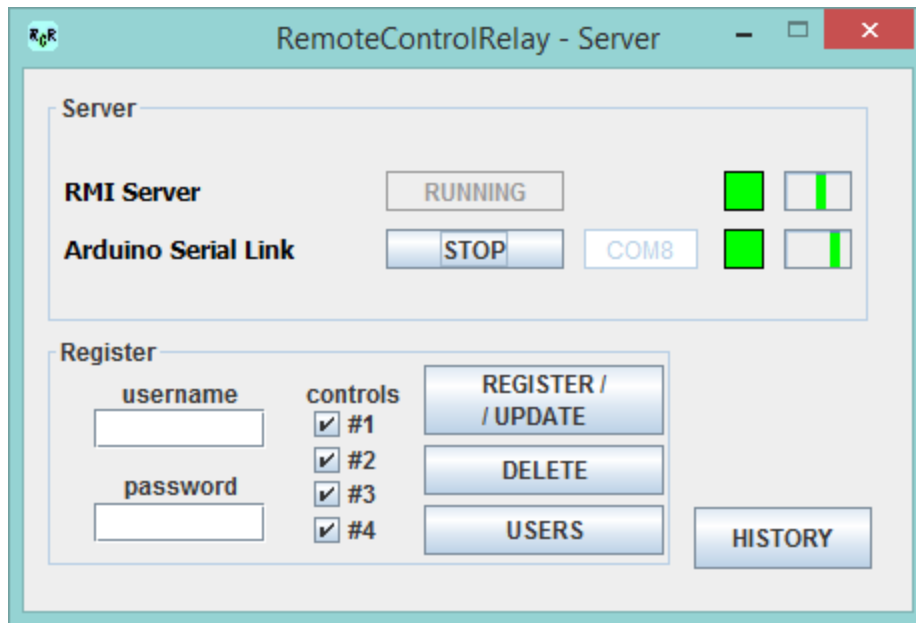


Figure 9 - Server GUI Main



## 3.6 Hardware

### 3.6.1 Relay

A relay is an electrically operated switch. It uses an electromagnet to mechanically operate a switch. Relays are often used when it is necessary to control a circuit with a low-power signal (with complete electrical isolation between control and controlled circuits).

It has usually 5 pins:

- **Input 1** and **Input 2** - terminals directly connected to the electromagnetic coil.
- **COM** - Common terminal
- **N/C** - Normally closed terminal
- **N/O** - Normally opened terminal

COM is usually connected to N/C, therefore the circuit between these terminals is usually closed (*Figure 10*).

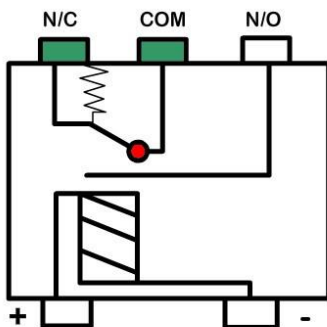


Figure 10 - Relay (connected)

When power (5V) is connected to the input terminals (+ and -), the current flowing through the coil produces an electromagnetic field which attracts the metal contact connected to the COM terminal. The contact moves away from N/C contact and towards the N/O terminal. When they touch, the circuit between COM terminal and N/O terminal is closed and a connection established (*Figure 11*).

In other words, circuits carrying up to 220V can be interrupted with only 5V. These 5V can be acquired from Arduino's output pins.

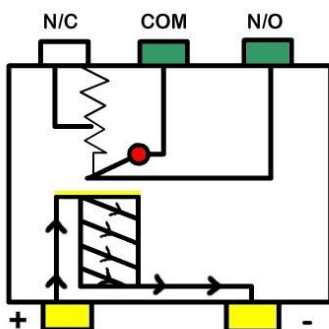


Figure 11 - Relay (disconnected)

## 3.6.2 Arduino

The whole logic of switching devices is inside the hardware aspect of this system. The most important hardware part is the Arduino UNO running a code programmed in the *Processing* programming language which reads values from the serial communication and compares them with predefined values. Each number that is sent by the program represents one state of the output port. Arduino then changes the state of individual ports according to the numbers that are received.

Output ports (digital output pins of Arduino) can have two states:

- LOW - 0V
- HIGH - 5V

Those output ports can be connected to relays which are then switched ON or OFF according to ports states.

In this implementation it is able to control 4 ports, however not all of them are connected to relays. This implementation is used for presentation purposes for better understanding the principle:

**Port #1** A LED diode in series with a current limiting resistor is connected directly to the first port to symbolize the changing state of the first digital output pin.

**Port #2** Port is connected to the relay module which simulates the press of the button on a wireless radio-controlled relay transmitter. This transmitter is able to change state of a relay which is inside the receiver module. Thus it is able to control power in the cable remotely.

**Port #3** Connected to the relay. Relay is prepared to be connected to a device.

**Port #4** Connected to the relay. Relay is used as a switch to a simple circuit. This circuit contains 2 LEDs, resistor and 2 AA batteries used to power the circuit. The main purpose of this circuit is to show that it can be opened and closed, although it is isolated from the Arduino.

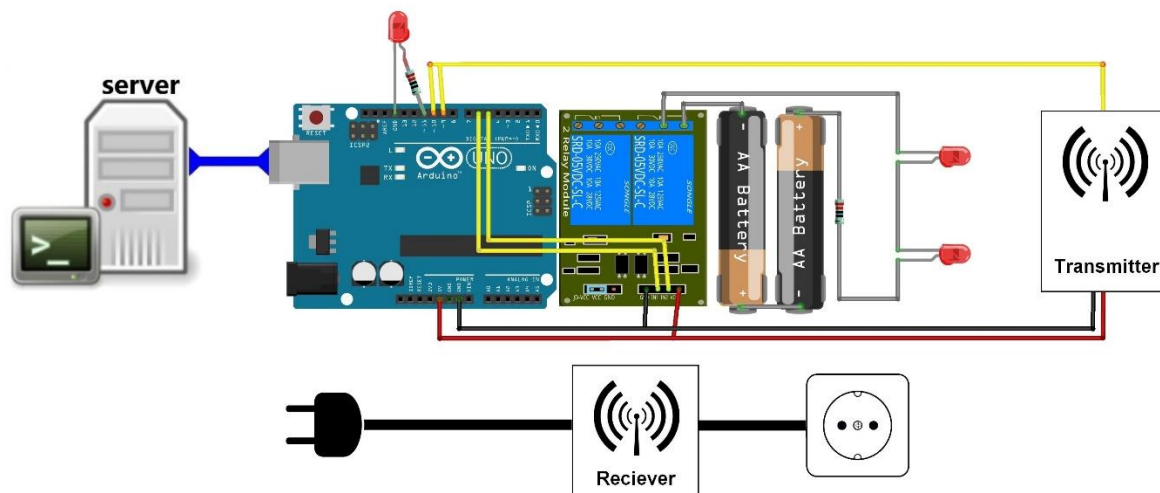


Figure 12 - Arduino

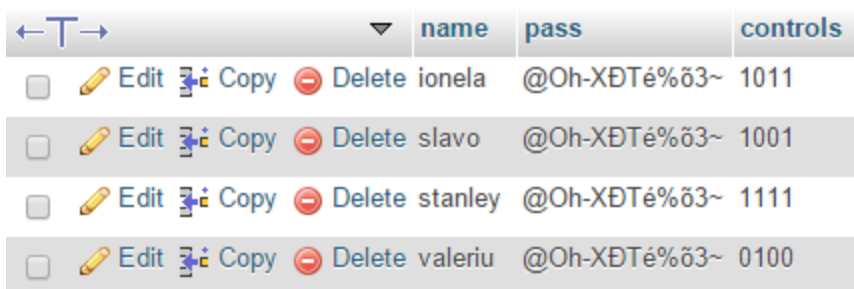
## 3.7 System security

The system implements two important features to make it secure:

- Database Password Encryption
- Client Login Hash Function

### 3.7.1 Database Password Encryption

When the administrator registers new users, their passwords are run through Advanced Encryption Standard (AES) before being stored. This is done using a special key only the java system knows.



The screenshot shows a table with three columns: 'name', 'pass', and 'controls'. The 'pass' column contains encrypted passwords represented by a hex string '@Oh-XĐTé%ø3~'. Each row has a checkbox, an 'Edit' button, a 'Copy' button, and a 'Delete' button.

	name	pass	controls
<input type="checkbox"/>	ionela	@Oh-XĐTé%ø3~	1011
<input type="checkbox"/>	slavo	@Oh-XĐTé%ø3~	1001
<input type="checkbox"/>	stanley	@Oh-XĐTé%ø3~	1111
<input type="checkbox"/>	valeriu	@Oh-XĐTé%ø3~	0100

Figure 13 - Encrypted Passwords in the Database

The administrator can still see user passwords in the server side application window. This is achieved by decrypting the passwords, using the same key, as they are retrieved from the database.

### 3.7.2 Client Login Hash Function

When a client logs in, the entered password is hashed using the MD5 cryptographic hash function. The resulting hash is then sent over the network to the server side.

The server side decrypts the password from the database and runs it through the same MD5 cryptographic function. This hash is then compared with the hash version of the password which the user typed. If the two of them match, then the user's entry is correct and the login is successful.

## 4. Implementation

This section explains in more detail how the client server system, design patterns, and system security were implemented. The process for each of these key system components is described step-by-step with diagrams and sections of code for clarification.

### 4.1 Client-Server System: RMI

Implementing RMI consists of defining a remote server interface (**RelayControlObservable**), which extends the *java.rmi.Remote* interface, then declaring a server class which implements this interface and its methods (**ServerController**). The server must use the **rmiregistry** command to create and start a remote object registry and *bind* it to a name. The client must then call the *Naming.lookup()* function to obtain a reference to the remote object and be able to invoke methods on the reference.

The diagram below represents the RMI implementation in this system, but does not include all of the methods/variables used in the interfaces and classes, **only the more relevant** ones for this topic.

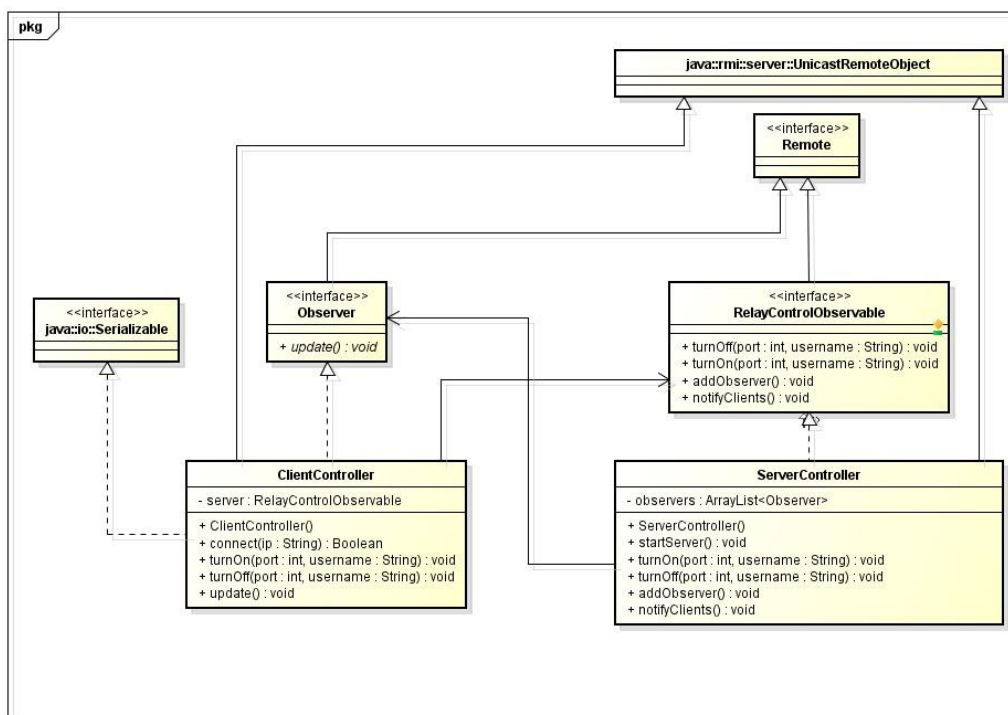


Figure 14 - RMI diagram

#### 1. Creating the RelayControlObservable remote interface

The purpose of this interface is to declare all the methods the client(s) will be able to call remotely. It extends the *java.rmi.Remote* interface, and each method declares a *java.rmi.RemoteException* in its throws clause.

```
public interface RelayControlObservable extends Remote {

    public boolean turnOn(int port, String username) throws RemoteException;
    public boolean turnOff(int port, String username) throws RemoteException;
    public int[] check() throws RemoteException;
    public String[] getPortNames() throws RemoteException;
    public int[] login(String username, char[] password) throws RemoteException;
    public String getHistory(String range) throws RemoteException;

    public void addObserver(Observer observer) throws RemoteException;
    public void deleteObserver(Observer observer) throws RemoteException;
    public void notifyClients() throws RemoteException;
}
```

## 2. Creating the ServerController class

The ServerController class implements the above remote interface and extends *java.rmi.UnicastRemoteObject*. All interface methods are implemented and declare a RemoteException. This class also consists of a *record()* method, which is only available locally.

In the server class implementation, the *startServer()* method uses the **rmiregistry** command creates and starts a remote object registry on port 1099. It then uses the remote object registry and invokes the *java.rmi.Naming.rebind()* function to bind remote objects to the “*messageServer*” name.

```
@SuppressWarnings("unused")
public void startServer()
{
    try
    {
        Registry reg = LocateRegistry.createRegistry(1099);
        Naming.rebind("messageServer", this);
        System.out.println("Starting server...");
    }
    catch (RemoteException | MalformedURLException e)
    {
        JOptionPane.showMessageDialog(null, "Server is already running.",
            "Server Instance Running", JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        e.printStackTrace();
    }
}
```

## 3. Creating the ClientController class

The client class also uses the remote object registry, but summons the *java.rmi.Naming.lookup()* function to look up remote objects on port 1099, bound to the “*messageServer*” name and make remote method invocations.

```
public boolean connect(String ip)
{
    try
    {
        server = (RelayControlObservable) Naming.lookup("rmi://" + ip +
            ":1099/messageServer");
        server.addObserver(this);
        return true;
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
        return false;
    }
}
```

## 4.2 Design patterns

### 4.2.1 Model-View-Controller

#### 4.2.1.1 Model package, Server Side

In this system, the model package is only on the server side. The connection to the Arduino is handled there. The model includes one class “Arduino.java” (there is also a Mediator package that is considered as part of the model, reviewed separately). There are 3 variables:

```
public static SerialPort serialPort;  
private int[] check;  
private String[] portNames = new String[4];
```

The `serialPort` object makes communication over wired connection to the Arduino possible from the server. The `check` variable is a 4-element integer array designed for checking whether the ports are turned on or off. If it holds the value 1, the port is active, if it holds 0 – it means it is inactive. For example `check[2]=1` means that port number 3 is turned on. The `portNames` variable is a String array containing the names of individual ports (for example: “heating” or “kitchen light”). The names can be set to anything to make the ports easily reachable.

```
public static void openPort() {...};  
public static void closePort() {...};
```

The `openPort` and `closePort` methods above work with the `serialPort` object and manage the hardware connection, as well as the data transfer.

```
public boolean turnOn(int port) throws RemoteException {...};  
public boolean turnOff(int port) throws RemoteException {...};
```

The `turnOn` and `turnOff` methods actually send commands to the Arduino to turn on/off the connected relays and, thus, the appliances. The `port` parameter, an integer value, is there to specify which port will be switched on/off. Then the corresponding integer value is sent to the Arduino over the serial connection.

```
serialPort.writeInt(2+48);
```

In conclusion, the model package represents the hardware part of the system embodied in java code.

#### 4.2.1.2 View package, Server and Client

The system has two views. One for the server side, and one for the client side. The one on the server side is in the “server.view” package and is represented by “ServerGUI.java” class. The *Server GUI* is connected to the *DatabaseHandler* via the controller (*ServerMethods* class). It is connected to the database to be able to control competencies, show history, register users, etc. Therefore, the *ServerGUI* contains an instance of the server methods class:

```
private ServerMethods serverMethods = new ServerMethods();
```

So, in the case of registering a new user, for example, only the referential method is called:

```
serverMethods.register(name, password, competencies)
```

Since the system uses referential methods like this one, the GUI has no direct influence towards controlling the database. The visual aspects are created using JButtons, JLabels, JPanels etc. Here are some examples:

```
private JPanel registerPanel = new JPanel();  
private JButton btnHistory = new JButton("HISTORY");  
private JLabel lblUsername = new JLabel("username");
```

After all the elements are created, implementing the actual methods consists of adjusting their visibility inside the methods, as seen in the code below for the *showHistory* method.

```
public void showHistory()  
{  
    panel1.setVisible(false);  
    panel2.setVisible(false);  
    server.setVisible(false);  
    registerPanel.setVisible(false);  
    progressBar1.setVisible(false);  
    progressBar2.setVisible(false);  
    btnHistory.setVisible(false);  
  
    historyPanel.setVisible(true);  
    refreshHistory();  
}
```

In this figure the visibility of the history panel is set to true, while the other elements, unrequired at the moment, are hidden.

In conclusion, the *ServerGUI* class only handles the visual elements. To access the functionality of the program, an instance of the *ServerMethods* class is available as the middleman to the *DatabaseHandler*.

Similarly, there is a view class on the client side. The class is in the "client.view" package and is named "ClientGUI.java". The implementation logic is very similar to the server's. The class has JButtons, JTextFields etc. and a *ClientController* object:

```
private ClientController client;
```

The *client* object is the connection to the client controller. That way the view has no influence on the business logic of the system. Another important variable is:

```
private boolean loggedIn = false;
```

Keeping track of whether a user is logged in or not is done using this variable. This is also used to manage some key visual elements.

### 4.2.1.3 Controller Package, Server and Client

The system has two controllers, one for the server and one for a client. In the server controller package there are 5 classes – the interface, two classes regarding the iterator pattern, the server controller itself – `ServerController.java` class, and the `ServerMethods.java` class. The server controller acts as a middleman between the server GUI and both – the database and the Arduino. The server controller class is intended for communicating with a client and the server methods class for the server view. To have a connection to the other parts of the system in the server controller, it is necessary to have an instance for each:

```
private DatabaseHandler database = new DatabaseHandler();
private Arduino arduino = new Arduino();
private ServerGUI window;
```

In this system, if a server is started, it is started through the controller class. Connection to the Arduino and the database is already started in the variables declaration and the GUI is started by the constructor:

```
private ServerController() throws RemoteException
{
    super();

    startServer();

    window = new ServerGUI();
    window.frame.setVisible(true);

    observers = new ArrayList<Observer>();
}
```

The server controller is a singleton in this case, therefore the `getInstance()` method is also stated. When the server is started, the registry is also created. If a user tries to open the server twice, exception is handled. The error window would be shown in that case:

```
public void startServer()
{
    try
    {
        Registry reg = LocateRegistry.createRegistry(1099);
        Naming.rebind("messageServer", this);
        System.out.println("Starting server...");
    }
    catch (RemoteException | MalformedURLException e)
    {
        JOptionPane.showMessageDialog(null, "Server is
already running.", "Server Instance Running",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        e.printStackTrace();
    }
}
```



Other methods included are referential to the model, such as:

```
@Override
public int[] check() throws RemoteException
{
    return arduino.check();
}

@Override
public String[] getPortNames() throws RemoteException
{
    return arduino.getPortNames();
}
```

There are also referential methods to the database handler. More details about the referential methods can be found in the model and mediator part of this report. The other methods included in the server controller are regarding the iterator and the observer pattern.

The server methods class is a connection to the database handler and also to the model. As mentioned before, in compare to the server controller class, it is intended just for the server view purposes. The methods are referential to the model and the database. The link to the database handler is in the instance:

```
private DatabaseHandler database = new DatabaseHandler();
```

And the connection to the Arduino part of the model is in the constructor:

```
public void startSerialCommunication(String port)
{
    Arduino.serialPort = new SerialPort(port);
}
```

The controller package on the client side contains two classes, the interface (important for the observer functionality) and the `ClientController` class. That is necessary to manage a connection to the GUI and the server (remote connection). The view is created in the constructor:

```
public ClientController() throws RemoteException
{
    super();

    clientGUI = new ClientGUI(this);
    clientGUI.frame.setVisible(true);
    this.username = null;
}
```

The remote connection to the server is established in the following method:

```
public boolean connect(String ip)
{
    try
    {
        server = (RelayControlObservable) Naming.lookup("rmi://" + ip +
            ":1099/messageServer");
        server.addObserver(this);
        return true;
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
        return false;
    }
}
```

It is done in the separate method (not in the constructor) because the method is called from the view, when a user is asked to enter the IP of the server. The other methods are referential to the server, they pass data from/to the view. The only other method is the update method for the observer functionality.

#### 4.2.1.4 Mediator package

In the mediator package, there is just a single class – DatabaseHandler.java. In the database handler all the functionality of the database management is implemented because it is the only class directly connected to the database. Basically the handler executes MySQL queries. To picture the process here is the example of the method which is called if the user should be unregistered:

```
public boolean unregister(String name)
{
    try
    {
        sql = "DELETE FROM `users` WHERE `name` = ?";
        update(sql, name);
    }
    catch (SQLException e)
    {
        e.printStackTrace();
        return false;
    }

    return true;
}
```

As pictured above, the name from the parameter is inserted in the query instead of the question mark. The query is then sent to the database and executed.

The connection to the database is established in the handler's constructor. If the database does not exist it is automatically created:

```
public DatabaseHandler()
{
    super();

    try
    {
        Class.forName(driver);
        connection = DriverManager.getConnection(url, user, password);
    }
    catch(ClassNotFoundException e)
    {
        System.out.println("MySQL driver not found");
    }
    catch(SQLException e)
    {
        System.out.println("DATABASE NOT FOUND");
        createDatabase();
    }
}
```

To improve security of the system, an AES encryption is used. The password is stored in the database in the encrypted form. Even if someone would hack the access to the database, it won't be possible to read the passwords.

The password is never transferred through the network in its readable form. This method is used for creating hash from the decrypted password from the database so it can be compared with client's hash.

```
public char[] md5e(char[] password) throws NoSuchAlgorithmException
{
    MessageDigest md = MessageDigest.getInstance("MD5");
    md.update(new String(password).getBytes());
    byte[] digest = md.digest();
    StringBuffer sb = new StringBuffer();

    for (byte b : digest)
    {
        sb.append(String.format("%02x", b & 0xff));
    }

    return sb.toString().toCharArray();
}
```

In conclusion mediator handles all the database functionality through sending queries and returning result tables.

## 4.2.2 Observer

The Observer pattern is used, in this system, to update the Client GUIs when changes are made in the model's state. Specifically, it will update the displayed state of the ports each time they are turned on or off. Thus, an Observable party is required, along with one or more Observers.

The diagram below represents the Observer implementation in this system, but does not include all of the methods/variables used in the interfaces and classes, **only the more relevant** ones for this topic.

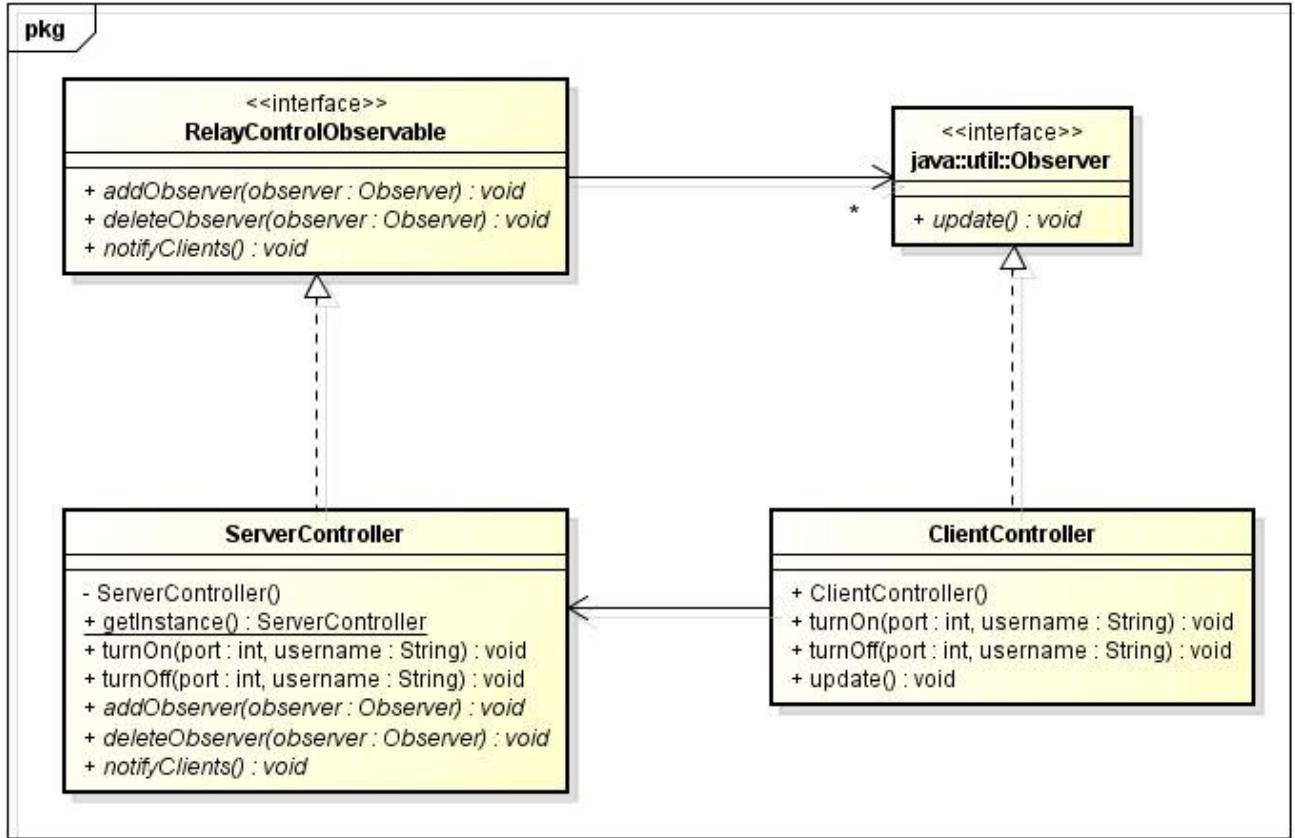


Figure 19. Observer diagram

Since the model is on the server side, the Observable object will also be implemented there. The subject maintains a collection of Observers and notifies them of its changes, so they may change their own state to reflect this.

## 1. Interface for the subject: RelayControlObservable

The RelayControlObservable interface specifies the methods which the Subject must implement. It consists of, among others, three fairly straight forward methods relating to the observer pattern: *addObserver*, *deleteObserver*, and *notifyClients*.

```
public interface RelayControlObservable extends Remote {  
  
    //... a few other methods  
  
    public void addObserver(Observer observer) throws RemoteException;  
    public void deleteObserver(Observer observer) throws RemoteException;  
    public void notifyClients() throws RemoteException;  
}
```

## 2. Interface for the Observer and Observer class

The Observer interface only contains one method, *update*, and extends the *java.rmi.Remote* class since RMI is used to connect the observers and subject.

```
public interface Observer extends Remote {  
  
    public void update() throws RemoteException;  
}
```

The **ClientController** class implements the Observer interface and its *update* method. This method calls the *updateCheck* method in the **clientGUI** class, which refreshes the displayed port states (TURN ON button with red square or TURN OFF button with green square).

```
@Override  
    public void update() throws RemoteException {  
        clientGUI.updateCheck();  
    }
```

### 3. Subject class

The `ServerController` class implements the `RelayControlObservable` interface methods, along with more of its own. The `addObserver` method adds one more `Observer` to the `ArrayList`, while the `deleteObserver` method removes a specific observer from the `ArrayList`. The `notifyClients` method iterates through the `ArrayList` and calls the `update` method for each client `Observer`.

```
@Override
public void addObserver(Observer observer) {
    this.observers.add(observer);
}

@Override
public void deleteObserver(Observer observer) {
    this.observers.remove(observer);
}

@Override
public void notifyClients() {

    ArrayList<Observer> temporaryArrayList = new ArrayList<Observer>();

    Iterator iterator = getIterator();

    while (iterator.hasNext())
    {
        Observer temporaryObserver = (Observer) iterator.next();
        try
        {
            temporaryObserver.update();
            temporaryArrayList.add(temporaryObserver);
        }
        catch (RemoteException e)
        {
            e.printStackTrace();
        }
    }
}
```

Thus, for every method which changes the model's state and is reflected in the client GUI, `notifyClients` must be executed in the method. This can be seen, for example, in the `turnOn` method in the `ServerController` subject.

```
@Override
public boolean turnOn(int port, String username) throws RemoteException {
    if (arduino.turnOn(port, username))
    {
        record(username, port, "ON");
        notifyClients();
        return true;
    }
    else
    {
        return false;
    }
}
```

## 4.2.3 Iterator

*Iterator pattern* is used to provide a standard way to traverse through a group of Objects. In this case it is a list of *Observers*.

The first step of the implementation is to create an interface which narrates navigation methods.

```
package server.controller;

public interface Iterator {

    public boolean hasNext();
    public Object next();
}
```

Next an *Iterable* interface needs to be created. This interface will contain only one method which is responsible to return the *Iterator* object.

```
package server.controller;

public interface Iterable {

    public Iterator getIterator();
}
```

In the following step a private class named *ServerControllerIterator* is created inside *ServerController* class. The inner class will implement the *Iterator* interface.

```
private class ServerControllerIterator implements Iterator {

    int index = 0;

    public boolean hasNext() {
        if (index < observers.size())
            return true;

        return false;
    }

    public Observer next() {
        if (this.hasNext())
            return observers.get(index++);

        return null;
    }
}
```

In order to use the Iterator object, an *Iterable* interface needs to be implemented in the *ServerController* class and *getIterator()* method has to be overridden.

```
public Iterator getIterator() {  
    return new ServerControllerIterator();  
}
```

Last but not least, *Iterator* object needs to be created inside *notifyClients()* method and use it to update the observers. Explanation of the *notifyClients()* method is below.

```
public void notifyClients() {  
    1. ArrayList<Observer> temporaryArrayList = new ArrayList<Observer>();  
    2. Iterator iterator = getIterator();  
    3. while (iterator.hasNext())  
    {  
        4. Observer temporaryObserver = (Observer) iterator.next();  
        try  
        {  
            temporaryObserver.update();  
            5. temporaryArrayList.add(temporaryObserver);  
        }  
        catch (RemoteException e)  
        {  
            e.printStackTrace();  
        }  
    }  
}
```

The aim of *notifyClients()* method is to notify the Users when the states of the ports change. The following methods works like this:

1. A *List* is created that has as purpose to store the updated observers.
2. An *Iterator* object is created.
3. A loop with the argument *iterator.hasNext()* is used to stop the iterator from going out of the observer collection.
4. A temporary *Observer* object is created to store a unique object.
5. The *Observer* object is updated and stored in the *List* created at *Step 1*.



## 4.2.4 Singleton

The *Singleton's* purpose is to control object creation, limiting the number of instances to a single one. In this system, the pattern will solve the problem of creating multiple *ServerControllers*. Instead, the *Singleton* pattern will provide global access to one instance of the *ServerController* class.

1. The first step is to make the constructor of the *ServerController* class **private**.

```
private ServerController()
{
    super();

    startServer();

    window = new ServerGUI();
    window.frame.setVisible(true);

    observers = new ArrayList<Observer>();
}
```

2. Next step is to create a static instance of the *ServerController* class and assign it to *null*.

```
public class ServerController extends UnicastRemoteObject implements
RelayControlObservable, Iterable {

    private static final long serialVersionUID = 1L;

    private static ServerController instance = null;
    private DatabaseHandler database = new DatabaseHandler();
    private Arduino arduino = new Arduino();
    private ArrayList<Observer> observers;
    private ServerGUI window;
```

3. To make sure that the singleton instance is only created once, a static **getInstance()** method is implemented. This method returns the singleton instance if it is already created.

```
public static ServerController getInstance() throws RemoteException
{
    if (instance == null)
    {
        instance = new ServerController();
    }
    return instance;
}
```

## 4.3 System security

### 4.3.1 Database Password Encryption

When the administrator registers new users, their passwords are sent to the database to be processed using AES encryption with a special key.

```
sql = "INSERT INTO `users` (`name`, `pass`, `controls`)\n      VALUES (?,AES_ENCRYPT(?, 'rcrelay'),?)\n      ON DUPLICATE KEY UPDATE pass=AES_ENCRYPT(?, 'rcrelay'), controls=?";\n\nupdate(sql, name, String.valueOf(password), controls, String.valueOf(password),\ncontrols);
```

The administrator can retrieve a list of users and their actual (unencrypted) passwords in the application window. This is done by decrypting the passwords using the same key and using a **SELECT** statement.

```
public String getUsers()\n{\n    String users = "";\n\n    try\n    {\n        preparedStatement = connection.prepareStatement(\n            "SELECT *, AES_DECRYPT(pass, 'rcrelay') FROM users");\n        resultSet = preparedStatement.executeQuery();\n\n        while (resultSet.next())\n        {\n            users += resultSet.getString(1) + "\\t\\t"\n                + resultSet.getString(4) + "\\t"\n                + resultSet.getString(3) + "\\n";\n        }\n    }\n    catch (SQLException e)\n    {\n        e.printStackTrace();\n    }\n\n    return users;\n}
```

## 4.3.2 Client Login Encryption

When a client logs in, the entered password is hashed. The *MessageDigest* class provides the functionality of a message digest algorithm. In this implementation, the program is using an MD5 cryptographic hash function.

```
public char[] md5e(char[] password) throws NoSuchAlgorithmException
{
    MessageDigest md = MessageDigest.getInstance("MD5");
    md.update(new String(password).getBytes());
    byte[] digest = md.digest();
    StringBuffer sb = new StringBuffer();

    for (byte b : digest)
    {
        sb.append(String.format("%02x", b & 0xff));
    }

    return sb.toString().toCharArray();
}
```

The server side decrypts the password from the database using AES Decryption as a part of **Database Password Encryption**. Afterwards, the returned password is send to the same MD5 cryptographic function to create a 16-byte hash in the form of a 32 digit hexadecimal number. This number is then compared with the hashed password from the user. If they are a match, the login is successful.

```
...
pw = md5e(resultSet.getString(4).toCharArray());
```

## 5. Testing

---

Testing is one of the most crucial aspects of developing any system. It is needed to ensure that the implemented functionality matches the initial requirements. It provides a detailed picture of the system and decreases the likeliness of blind spots and errors. Internal tests were performed before the final release of the **RemoteControlRelay** system.

In this test, the internal structure of the system remains a “black box”. For the tester, it is only needed to know what the expected response as a consequence of the input should be. All tests are based on the use case descriptions (**Section 2.1.3 and Appendix A**). This test checks that the system follows the use case base and exception sequences as the tester inputs both valid and invalid content.

Below is an overview of all the requirements the system must adhere to and the results of their testing.

Use Case	Test Result
Log In	Working as expected
Log Out	Working as expected
Control Ports	Working as expected
Register/Update User	Working as expected
Delete User	Working as expected
Show User	Working as expected
Show History	Working as expected
Start/Stop Serial Communication	Working as expected

In the following table, test results from the **Login** use case are presented. More detailed results for the rest of the use cases can be found in **Appendix D**.

Action	Description	Expected Result	Final Result
<b>Log in</b>	The User inputs a legal username and password.	The ports available for the account are loaded in the <i>Device</i> panel.	Working as expected
	The User inputs an illegal username and password.	Username and password text fields are cleared and a “ <i>wrong credentials</i> ” message is display below them.	Working as expected
	The User inputs an illegal username.	Username and password text fields are cleared and a “ <i>wrong credentials</i> ” message is display under them.	Working as expected
	The User inputs an illegal password.	Username and password text fields are cleared and a “ <i>wrong credentials</i> ” message is display under them.	Working as expected
	The User inputs the username only.	Username text field is cleared and a “ <i>wrong credentials</i> ” message is display under it.	Working as expected
	The User inputs the password only.	Password text field is cleared and a “ <i>wrong credentials</i> ” message is display under it	Working as expected
	The User does not input a username and password.	A “ <i>wrong credentials</i> ” message is display under the password text field	Working as expected

This table covers all the scenarios possible for the **login** use case, including when the username and password are incorrect, when only one of the credentials is specified, etc. Conducting such thorough tests minimizes the chances of encountering problems when further developing the system.

## 6. Discussion

---

While the system is indeed ready to be deployed, with all the intended features implemented and working as expected, there are some considerations to be made. First of all, the installment of the hardware part of the system requires some skill. Although this is a limitation to anyone wanting to do it themselves, it is not uncommon for home automation systems to offer this service as part of their product.

A notable feature of the system is that it allows multiple accounts, each with customizable access rights, in a single household. Also, having separate interfaces for the server and client side ensures that specific actions are only available for the Administrator of the system. The users should only be concerned with controlling devices from their GUIs, which was designed to be simple and easy to understand.

A number of improvements can be considered for future version of the **RemoteControlRelay** system. An obvious possibility would be to implement more control over device settings, rather than just have the option of turning on or off. Another idea is to increase the number of ports and, thus, the number of devices which can be connected. Another improvement would be to customization options to the history view, such as sorting by user, action, or device, displaying statistics, etc.

In the future, the hardware component could also be upgraded to a more compact size and be connected to the server wirelessly.

All of these features would have been implemented if there were more time available, but even now, all the requirements and objectives have been accomplished and the **RemoteControlRelay** system is working and ready for deployment.

## 7. Conclusion

---

The project proposed for this semester was one of great interest to all members of this team. It started with a relatively vague idea, but progressed to a fully functioning and easy to use system. The initial idea was taken and reformed, research was done, and the system began to take shape. Analyses then had to be done, to determine the requirements, and further derive the use cases and activity diagrams. After that, much consideration had to be put into designing system. Questions like: *Which design patterns to incorporate and where? What connection type? How should the database look? How should the interface look?* Were answered. Also, the hardware aspect of the system was then decided upon. The first version of the class diagram soon emerged. While coding the classes, many alterations had to be made in the model diagram. After each section of code was implemented, tests had to be run to ensure it was working as planned and fix any unexpected surprises. New elements were added to the system, which were not proposed in the project's inception. System security was implemented in the form of encrypting passwords before storing and using a hash algorithm during the login process.

Each step over the course of development was documented as well as possible. In the end, what is left is a fully functioning client server system, using RMI, facilitated with design patterns, which allows users to connect and control devices *in their homes* remotely.

## 8. References

---

- [1] Serialization, [http://www.tutorialspoint.com/java/java\\_serialization.html](http://www.tutorialspoint.com/java/java_serialization.html), December 2014.
- [2] Relay, <http://en.wikipedia.org/wiki/Relay>
- [3] MVC, [http://www.tutorialspoint.com/design\\_pattern/mvc\\_pattern.htm](http://www.tutorialspoint.com/design_pattern/mvc_pattern.htm)
- [4] RMI, <http://www.javacoffeebreak.com/articles/javarmi/javarmi.html>
- [5] Iterator, [http://www.tutorialspoint.com/design\\_pattern/iterator\\_pattern.htm](http://www.tutorialspoint.com/design_pattern/iterator_pattern.htm)
- [6] Singleton, <http://www.javaworld.com/article/2073352/core-java/simply-singleton.html>
- [7] Control4™, <http://www.control4.com/>
- [8] Vera™, <http://getvera.com/>
- [9] Black Box testing, <http://www.softwaretestinghelp.com/black-box-testing/>
- [10] RMI, <http://www-itec.uni-klu.ac.at/~harald/ds2001/rmi/rmi.html> (accessed 24.05.15)
- [11] Use Case Diagrams: <http://www.agilemodeling.com/artifacts/useCaseDiagram.htm>
- [12] Database creation: <http://stackoverflow.com/questions/717436/create-mysql-database-from-java>
- [13] Study material from SDJ12, CON11, RDB11.



# APPENDIX

# **USE CASE DESCRIPTIONS**

RemoteControlRelay System

## USE CASE: RegisterUser

---

<b>Name:</b>	RegisterUser
--------------	--------------

---

<b>Actors:</b>	Administrator
----------------	---------------

---

<b>Summary:</b>	The Administrator registers a new user into the system.
-----------------	---

---

<b>Precondition:</b>	The user is not already registered in the system.
----------------------	---

---

<b>Base sequence:</b>	<ol style="list-style-type: none"><li>1. The Administrator types in the user name.</li><li>2. The Administrator types in the user password.</li><li>3. The Administrator selects the access rights for user.</li><li>4. The Administrator presses <i>Register/Update</i> button.</li><li>5. The system adds the new user to the database.</li><li>6. The system displays an updated list of registered users.</li></ol>
-----------------------	---

---

<b>Exception sequence:</b>	<p><b>At 4.</b></p> <p>The admin did not type in a new username and/or password.</p> <p>System does not register any new user. System displays username and/or password fields with red background. Admin must try again.</p>
----------------------------	---

---

<b>Postcondition:</b>	A new user is registered in the system, with username, password, and access rights.
-----------------------	---

---

## USE CASE: DeleteUser

---

<b>Name:</b>	DeleteUser
<b>Actors:</b>	Administrator
<b>Summary:</b>	The Administrator deletes a user from the database.
<b>Precondition:</b>	The user already exists in the database.
<b>Base sequence:</b>	<ol style="list-style-type: none"><li>1. The Administrator fills in the username field.</li><li>2. The Administrator presses the <i>Delete</i> button.</li><li>3. The system removes the user from the database.</li><li>4. The system displays an updated list of registered users.</li></ol>
<b>Exception sequence:</b>	<p><b>At 2.</b></p> <p>The admin did not type in an existing username.</p> <p>System will display a list of existing users, without deleting any of them.</p>
<b>Postcondition:</b>	The specified user is deleted from the database.

---

## USE CASE: UpdateUser

---

<b>Name:</b>	UpdateUser
<b>Actors:</b>	Administrator
<b>Summary:</b>	The Administrator makes changes regarding a user's password and/or access rights.
<b>Precondition:</b>	The user is already registered in the system.
<b>Base sequence:</b>	<ol style="list-style-type: none"><li>1. The Administrator types in the user name.</li><li>2. The Administrator fills in the password field with the new password.</li><li>3. The Administrator selects the new access rights.</li><li>4. The Administrator presses the <i>Register/Update</i> button.</li><li>5. System updates user's information.</li><li>6. The system displays an updated list of registered users.</li></ol>
<b>Exception sequence:</b>	<p><b>At 4.</b></p> <p>The admin did not type in an existing user or mistyped the username, but filled in the password and selected access rights.</p> <p>System registers a new user.</p>
<b>Exception sequence:</b>	<p><b>At 4.</b></p> <p>The admin did not type in a username (or password).</p> <p>System displays username fields with red background.</p> <p>Admin must try again.</p>
<b>Postcondition:</b>	The selected user's password and/or access rights have been modified.

---

## USE CASE: ViewUsers

---

<b>Name:</b>	ViewUsers
<b>Actors:</b>	Administrator
<b>Summary:</b>	The Administrator sees a list of registered users.
<b>Precondition:</b>	-
<b>Base sequence:</b>	<ol style="list-style-type: none"><li>1. The Administrator clicks the <i>Show</i> button.</li><li>2. The system displays an updated list of registered users.</li></ol>
<b>Exception sequence:</b>	<b>At 1.</b> There's a problem with the database connection. He's fucked.
<b>Postcondition:</b>	-

---

## USE CASE: Start/Stop Serial Communication

---

<b>Name:</b>	Start/Stop Serial Communication
<b>Actors:</b>	Administrator
<b>Summary:</b>	The Administrator sees a list of registered users.
<b>Precondition:</b>	Arduino is connected to the computer.
<b>Base sequence:</b>	<ol style="list-style-type: none"><li>1. The Administrator inputs the communication port name (only on first start).</li><li>2. The Administrator clicks the <i>Create</i> button.</li><li>3. The system establishes the connection.</li><li>4. Administrator now has the possibility to <i>Stop</i> and <i>Start</i> the connection.</li></ol>
<b>Exception sequence:</b>	<b>At 2.</b> Administrator specified the wrong communication port. No connection is established. Admin must restart the program and try again.
<b>Postcondition:</b>	Connection is established between Arduino and the system.

---

## USE CASE: ViewHistory

---

<b>Name:</b>	ViewHistory
<b>Actors:</b>	Administrator
<b>Summary:</b>	The Administrator sees a list of all user activities since forever.
<b>Precondition:</b>	-
<b>Base sequence:</b>	<ol style="list-style-type: none"><li>1. The Administrator clicks the <i>History</i> button.</li><li>2. The system displays an updated list of user activity (including user, date, time, appliance and state)</li></ol>
<b>Exception sequence:</b>	<b>At 1.</b> There's a problem with the database connection. He's fucked.
<b>Postcondition:</b>	-

---



## USE CASE: LoginUser

---

<b>Name:</b>	ShowHistory
<b>Actors:</b>	User
<b>Summary:</b>	The user inputs a username and password to gain access to the sytem.
<b>Precondition:</b>	The user is registered in the system.
<b>Base sequence:</b>	<ol style="list-style-type: none"><li>1. The user fills in the username field.</li><li>2. The user fills in the password field.</li><li>3. The user clicks the <i>Login</i> button.</li><li>4. The system displays the account window.</li></ol>
<b>Exception sequence:</b>	<b>At 3.</b> The username or password is incorrect. System displays <i>wrong credentials</i> message.
<b>Postcondition:</b>	The user has access and can now use the system.

---

## USE CASE: LogoutUser

---

<b>Name:</b>	LogoutUser
<b>Actors:</b>	User
<b>Summary:</b>	The user presses a button and logs out of the system.
<b>Precondition:</b>	The user is logged into the system.
<b>Base sequence:</b>	<ol style="list-style-type: none"><li>1. The user clicks the <i>Logout</i> button.</li><li>2. The system displays the main window.</li></ol>
<b>Postcondition:</b>	The user has terminated access to his/her account, but can still see the states of devices.

---

## USE CASE: ControlAppliances

---

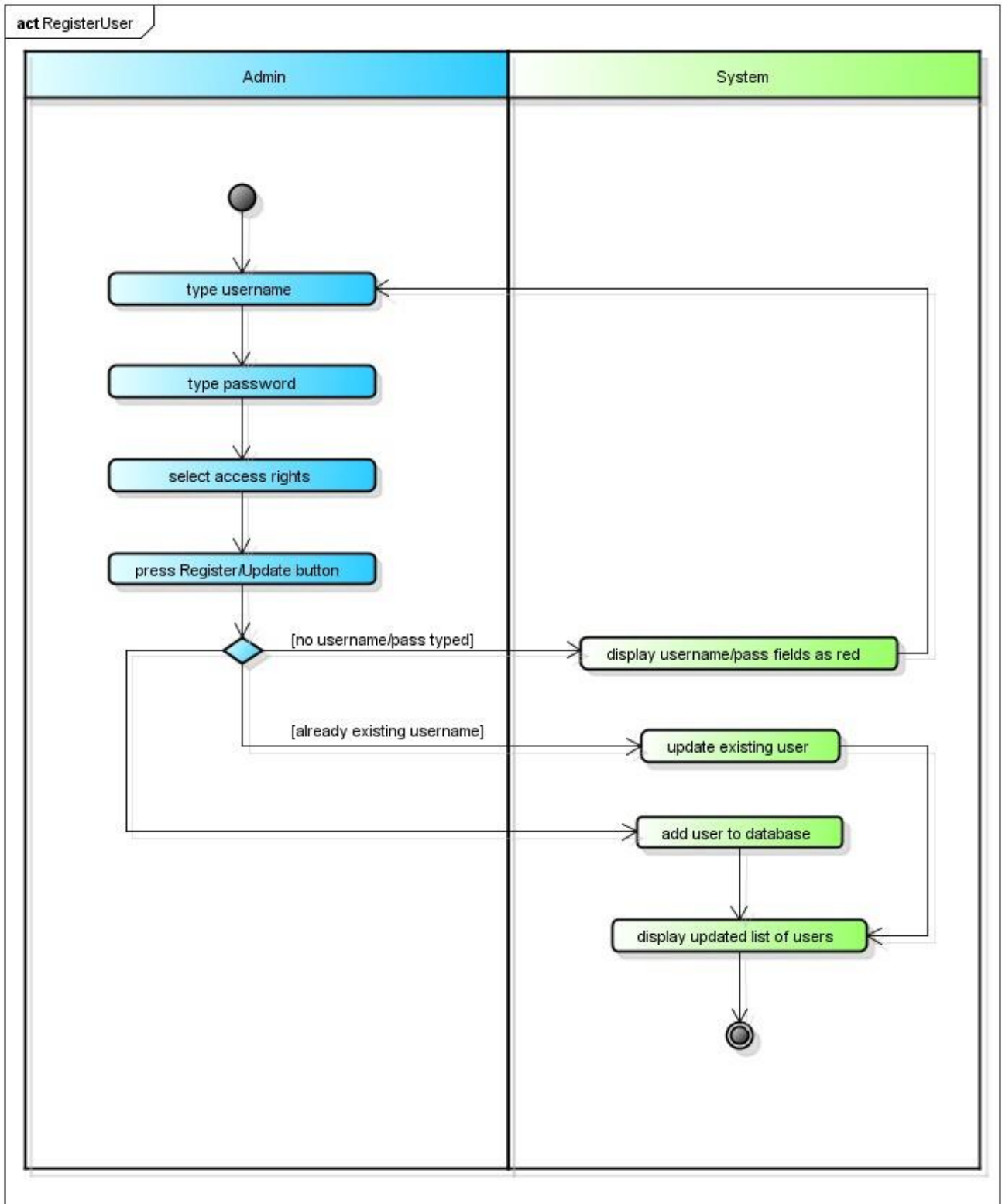
<b>Name:</b>	ControlAppliances
<b>Actors:</b>	User
<b>Summary:</b>	The user presses buttons for the corresponding devices to remotely control them.
<b>Precondition:</b>	The user is logged into the system.
<b>Base sequence:</b>	<ol style="list-style-type: none"><li>1. The user selects which port to control.</li><li>2. The user clicks the <i>TURN ON</i> or <i>TURN OFF</i> button (depending on the current state of the port).</li><li>3. The system turns on/off the device connected to that particular port.</li><li>4. Change button according to next state. (If user clicked <i>TURN ON</i>, after the action the button would change to <i>TURN OFF</i>).</li><li>5. The system logs the action into the database.</li></ol>
<b>Exception sequence:</b>	User
<b>Postcondition:</b>	One or more device states have been modified, all actions have been recorded in the database.

---

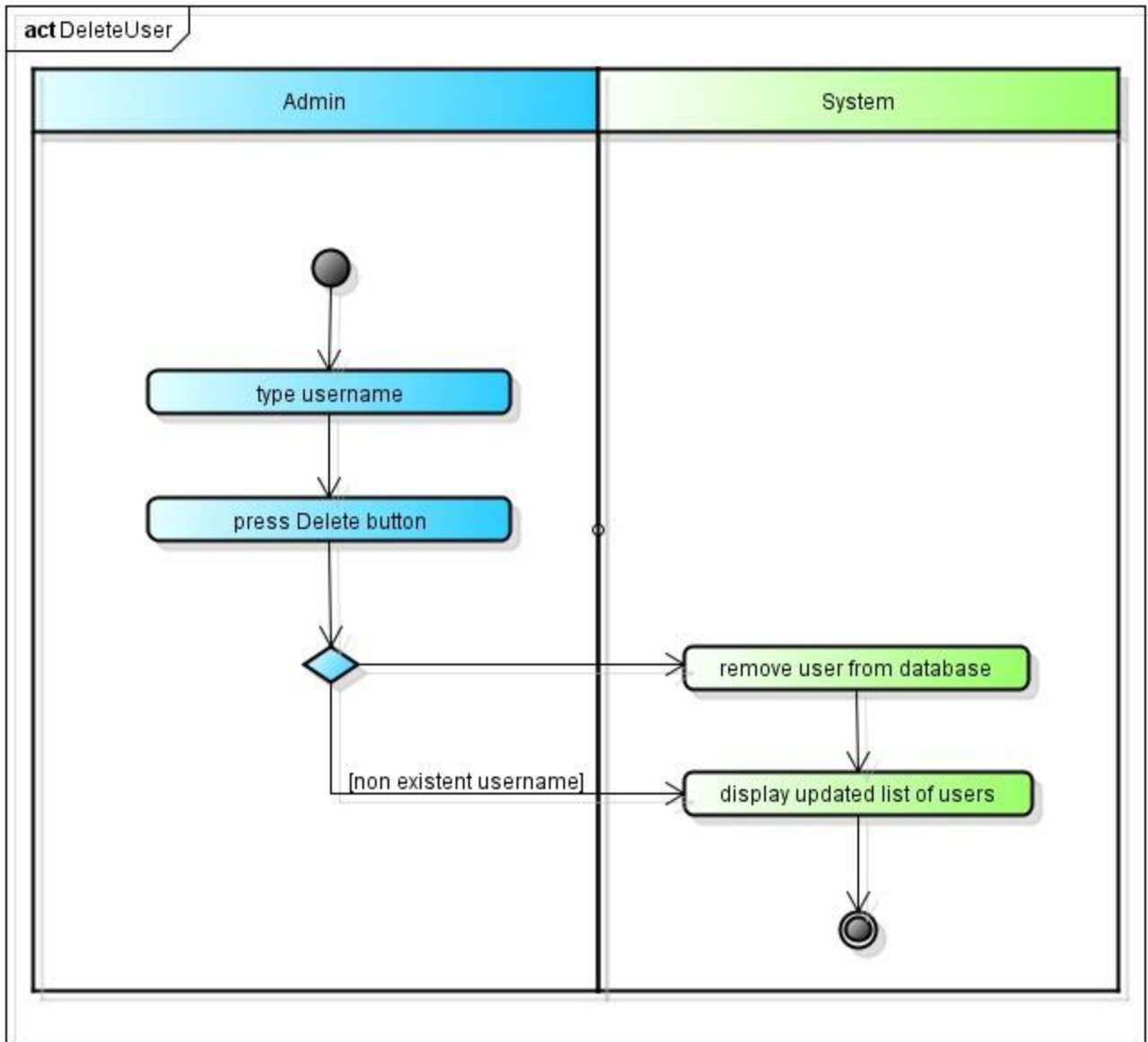
# **ACTIVITY DIAGRAMS**

RemoteControlRelay System

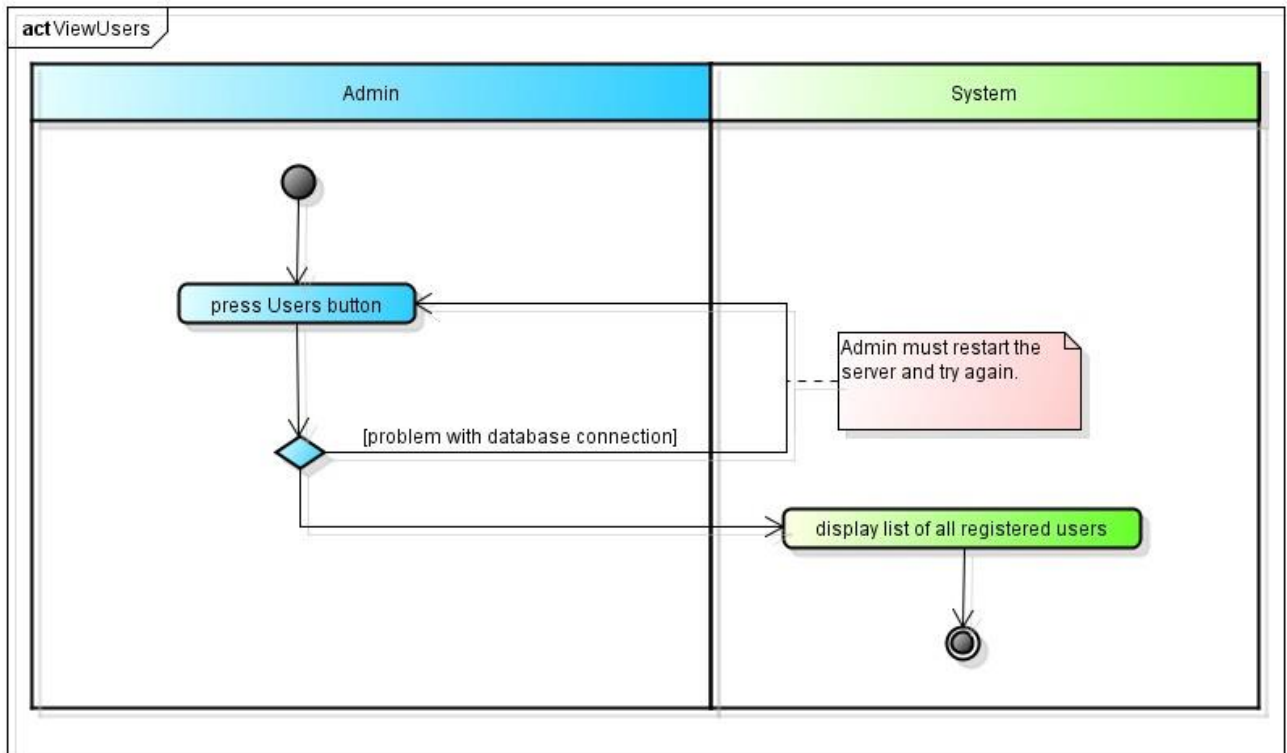
# Register User



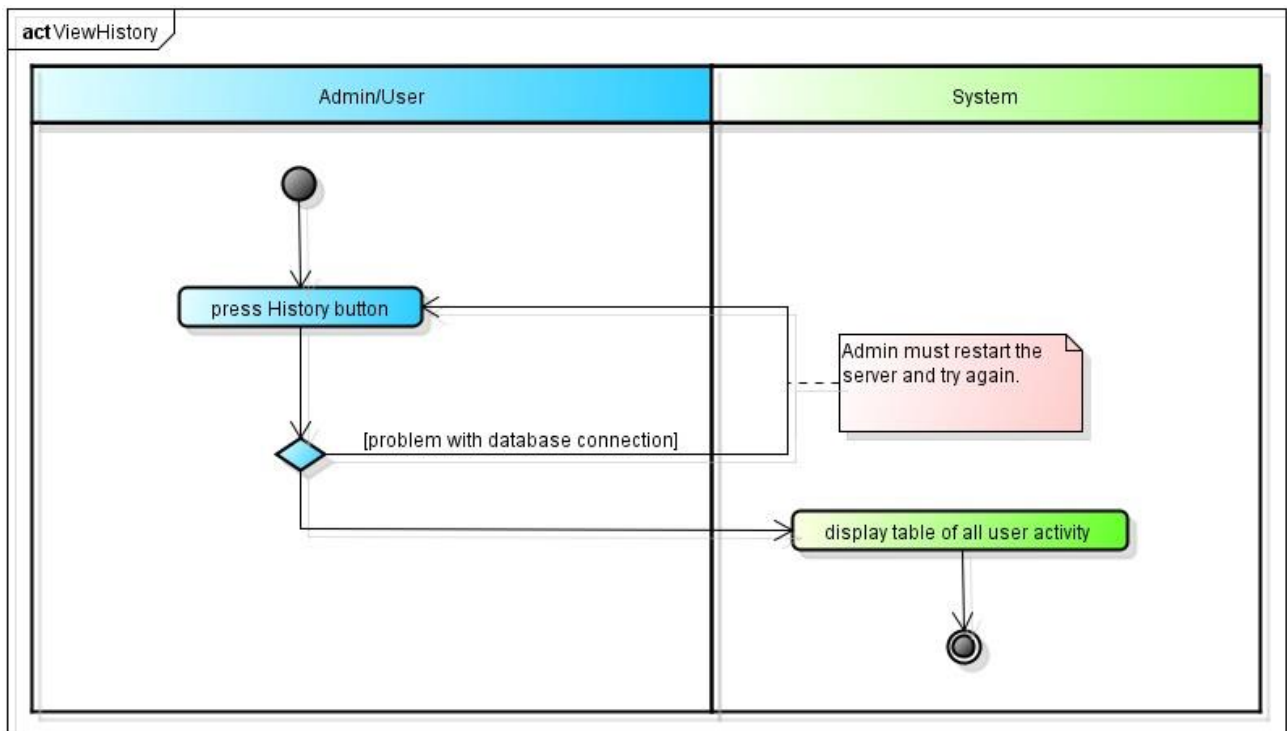
# Delete User



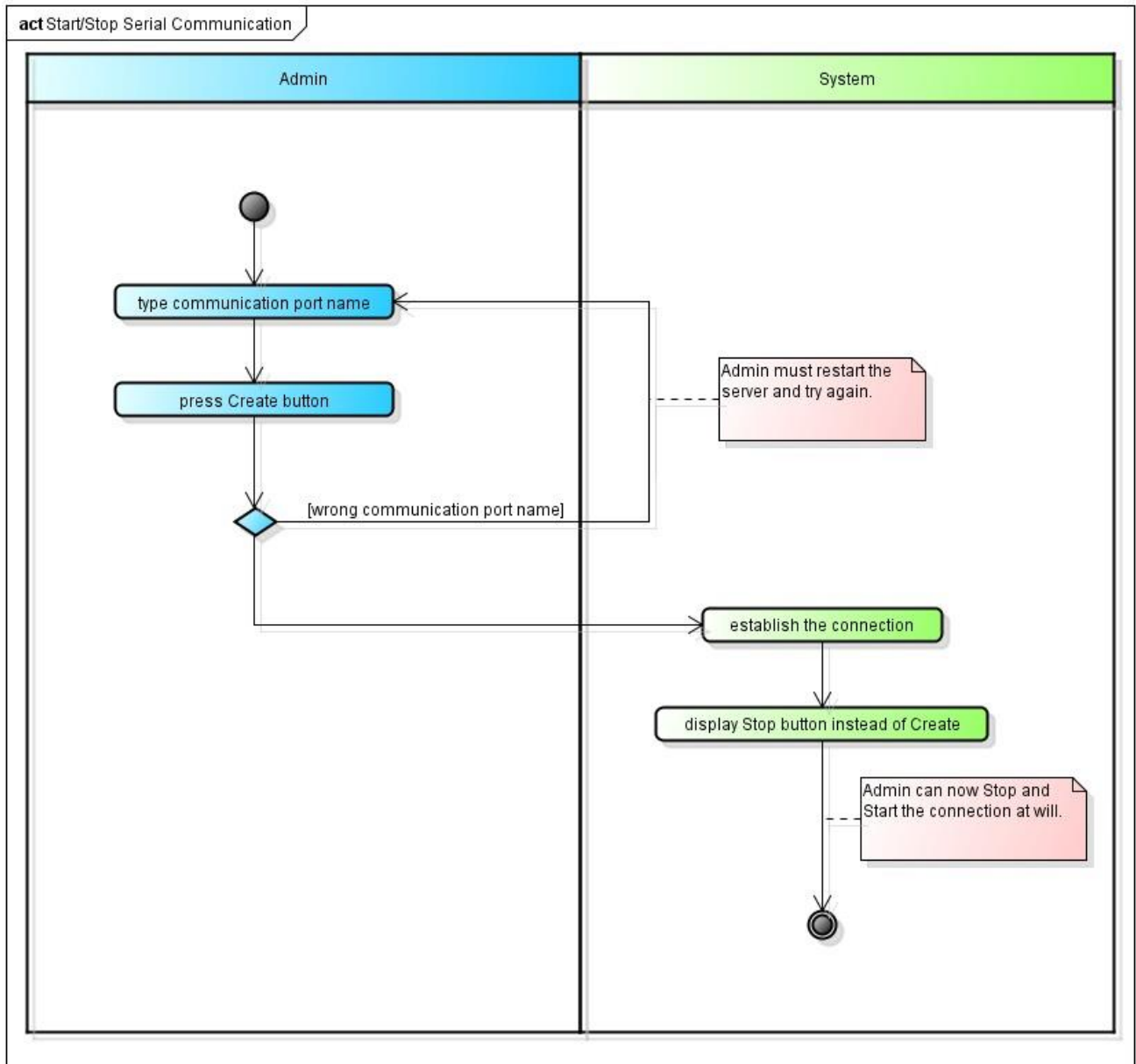
# View Users



# View History

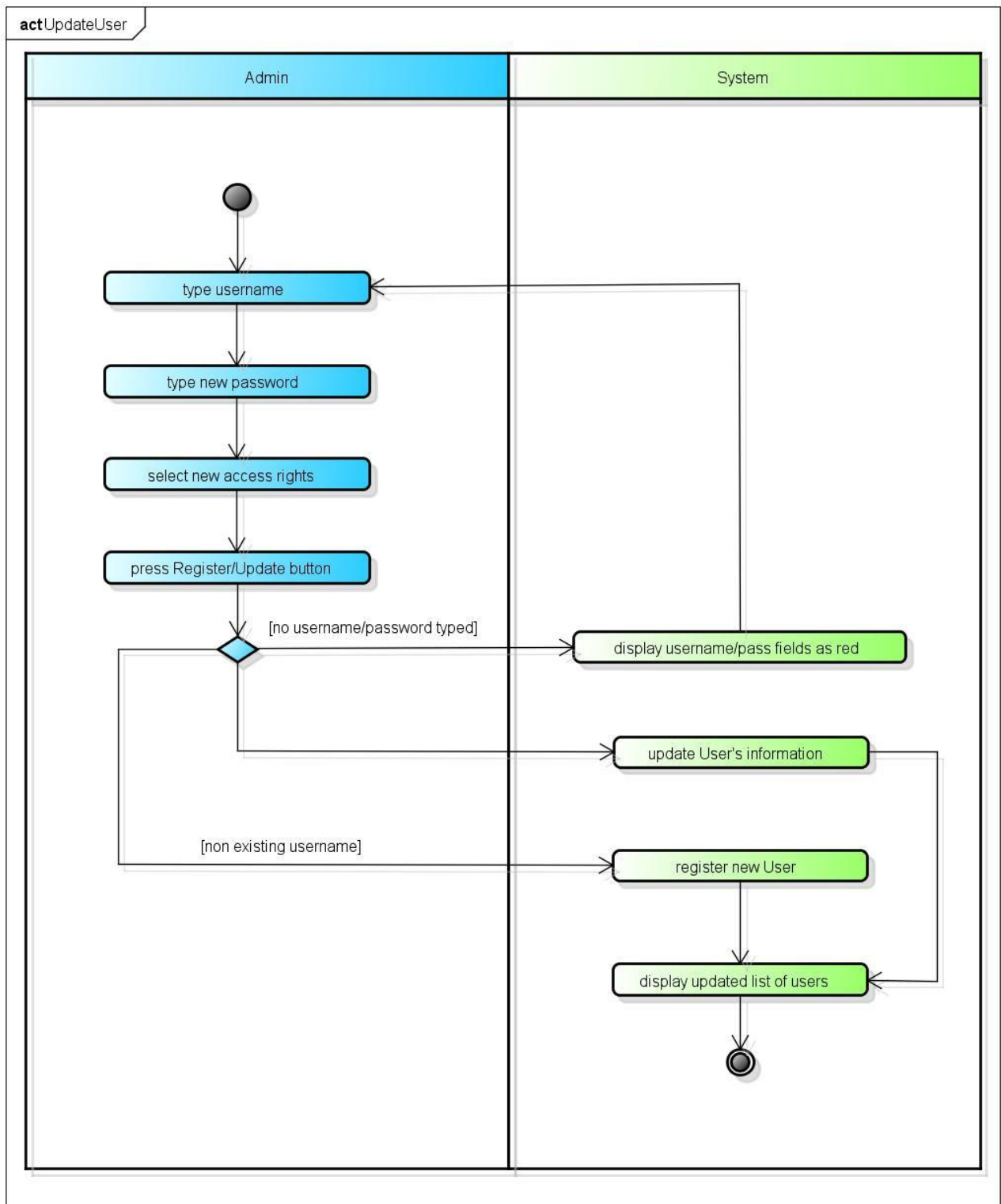


# Start/Stop Serial Communication

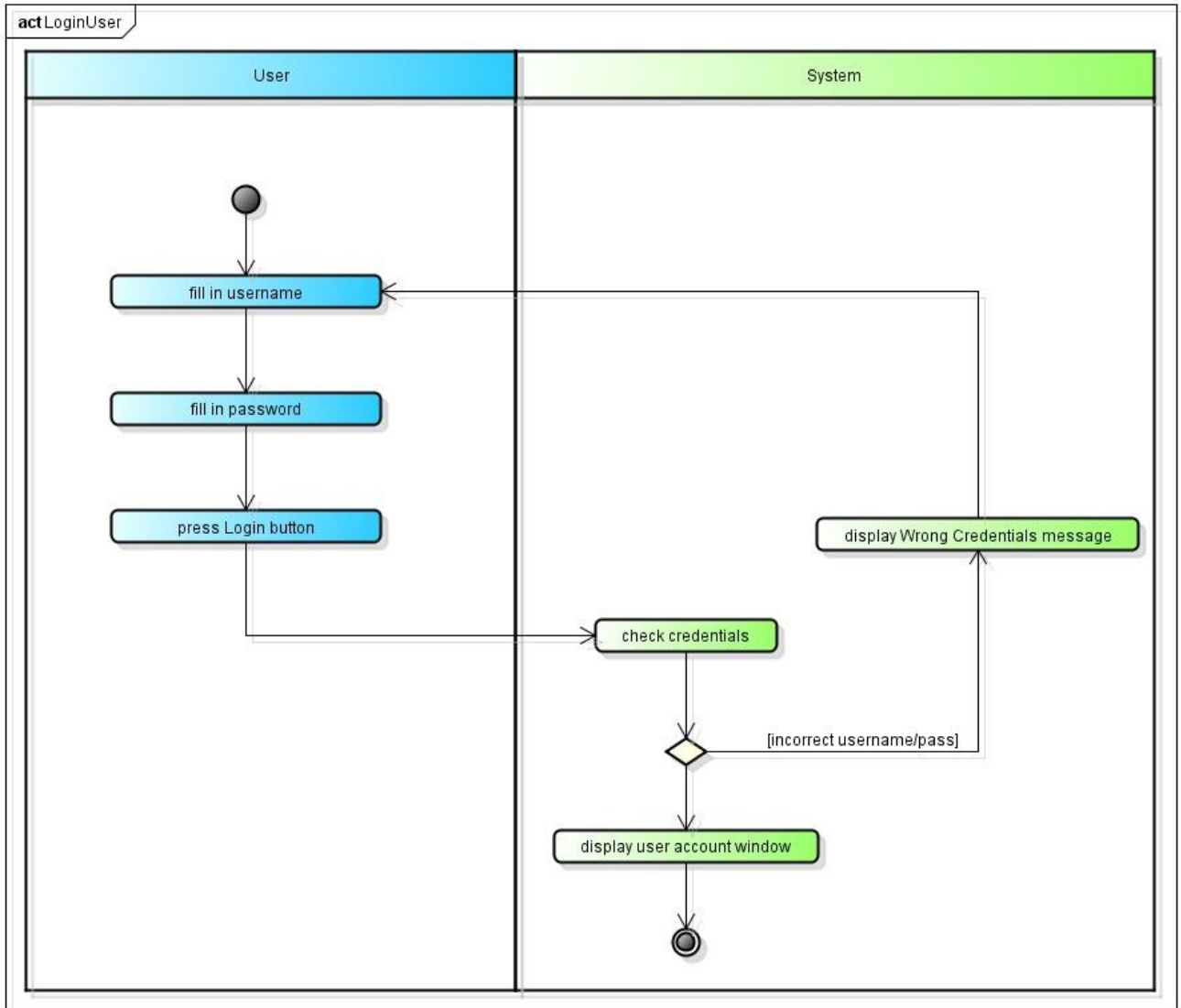




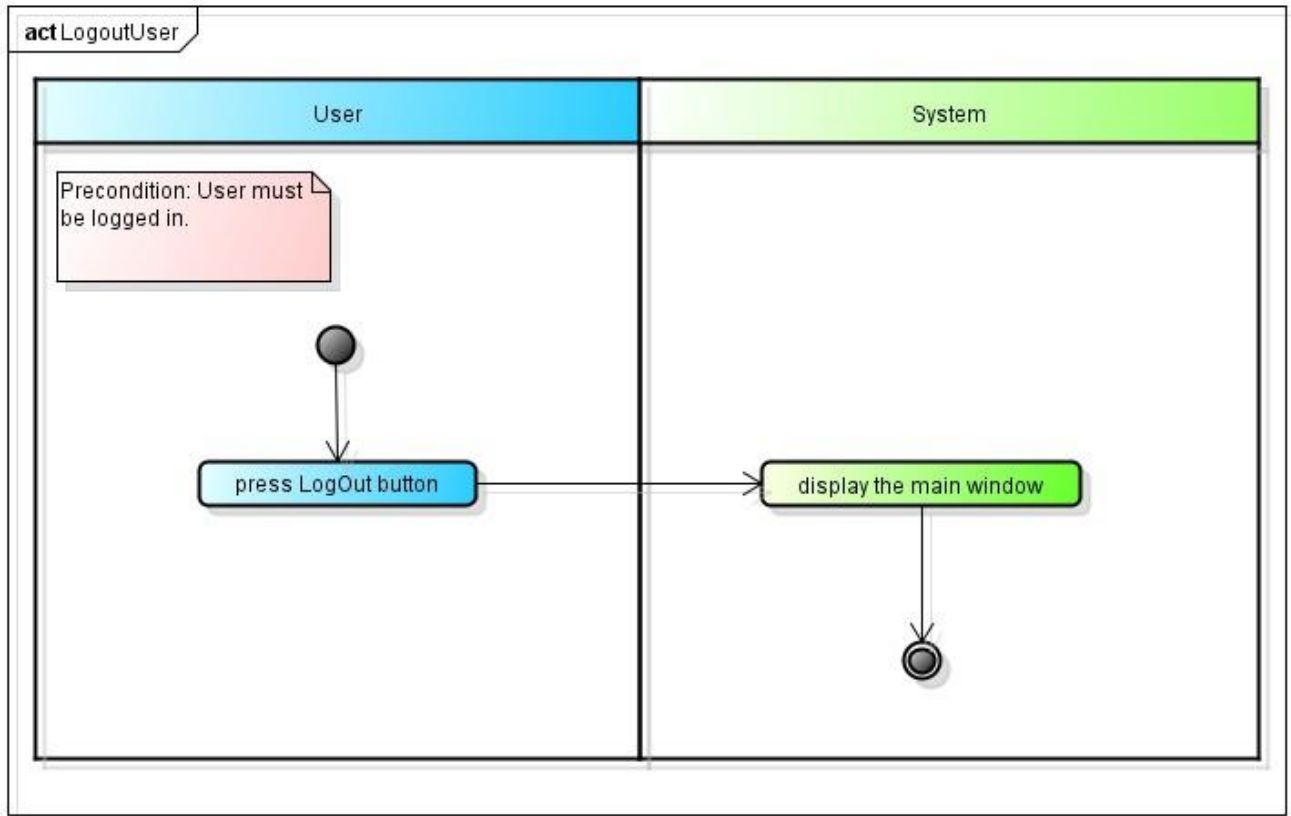
# Update User



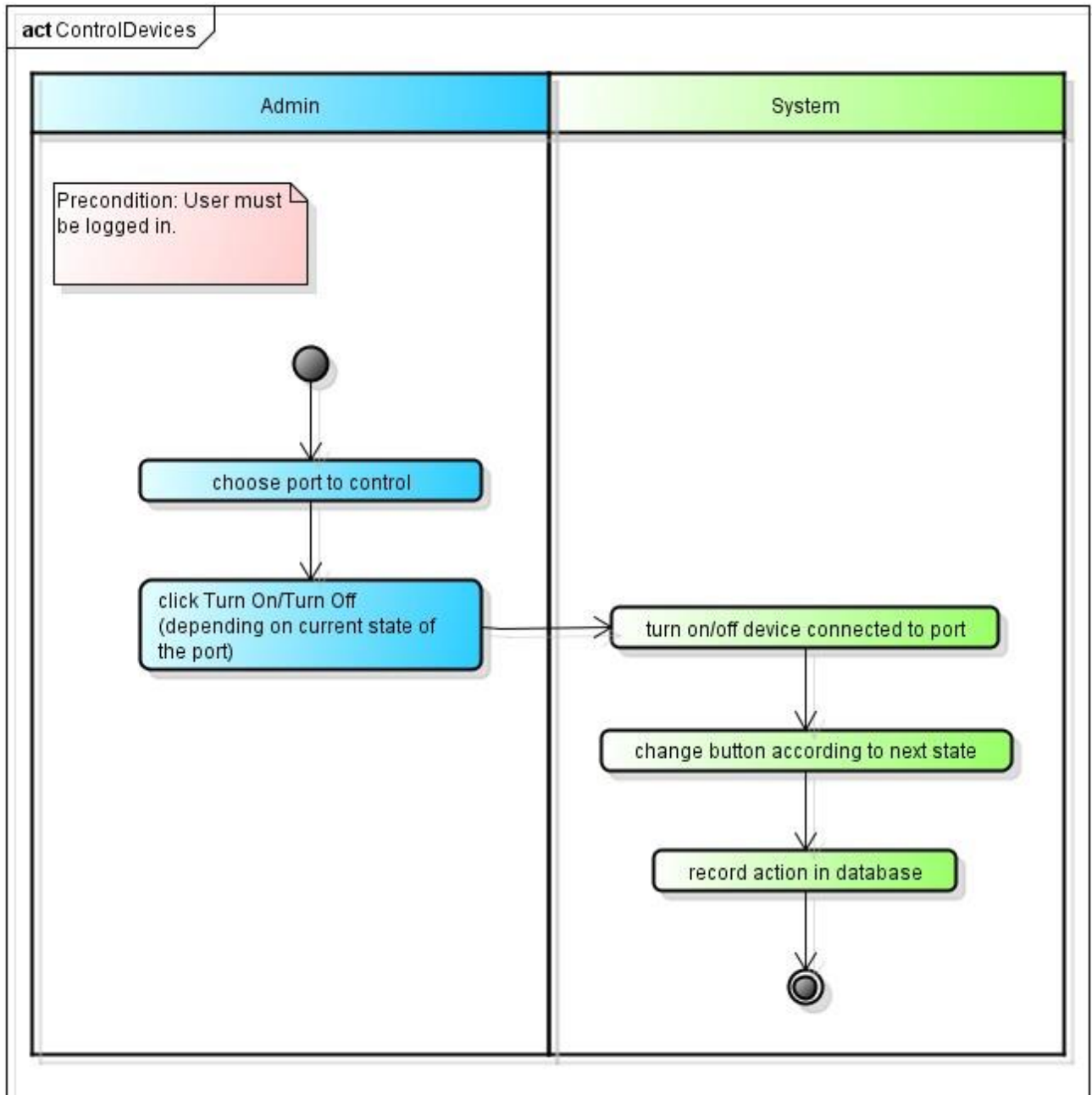
# Login User



# Logout User

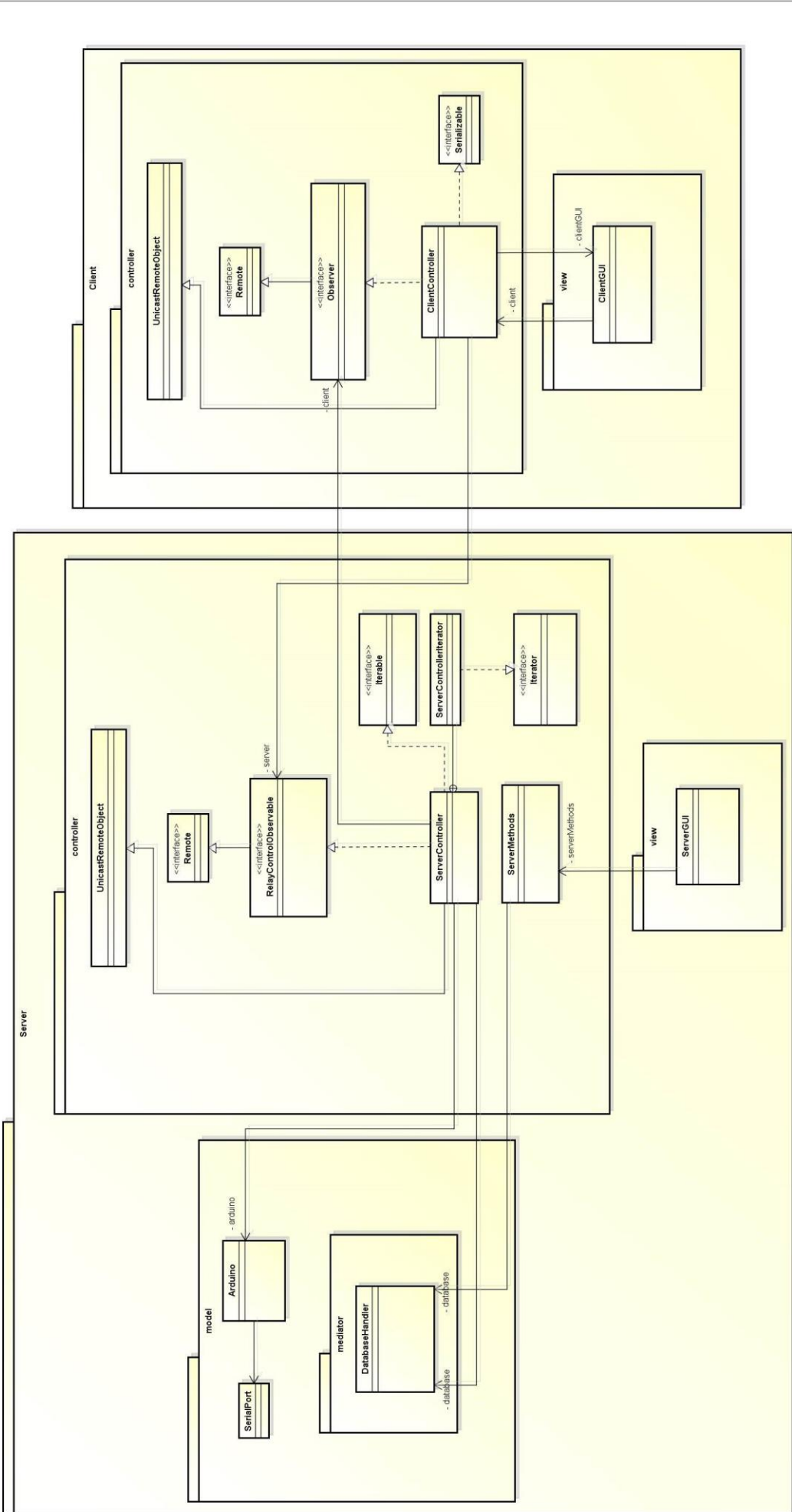


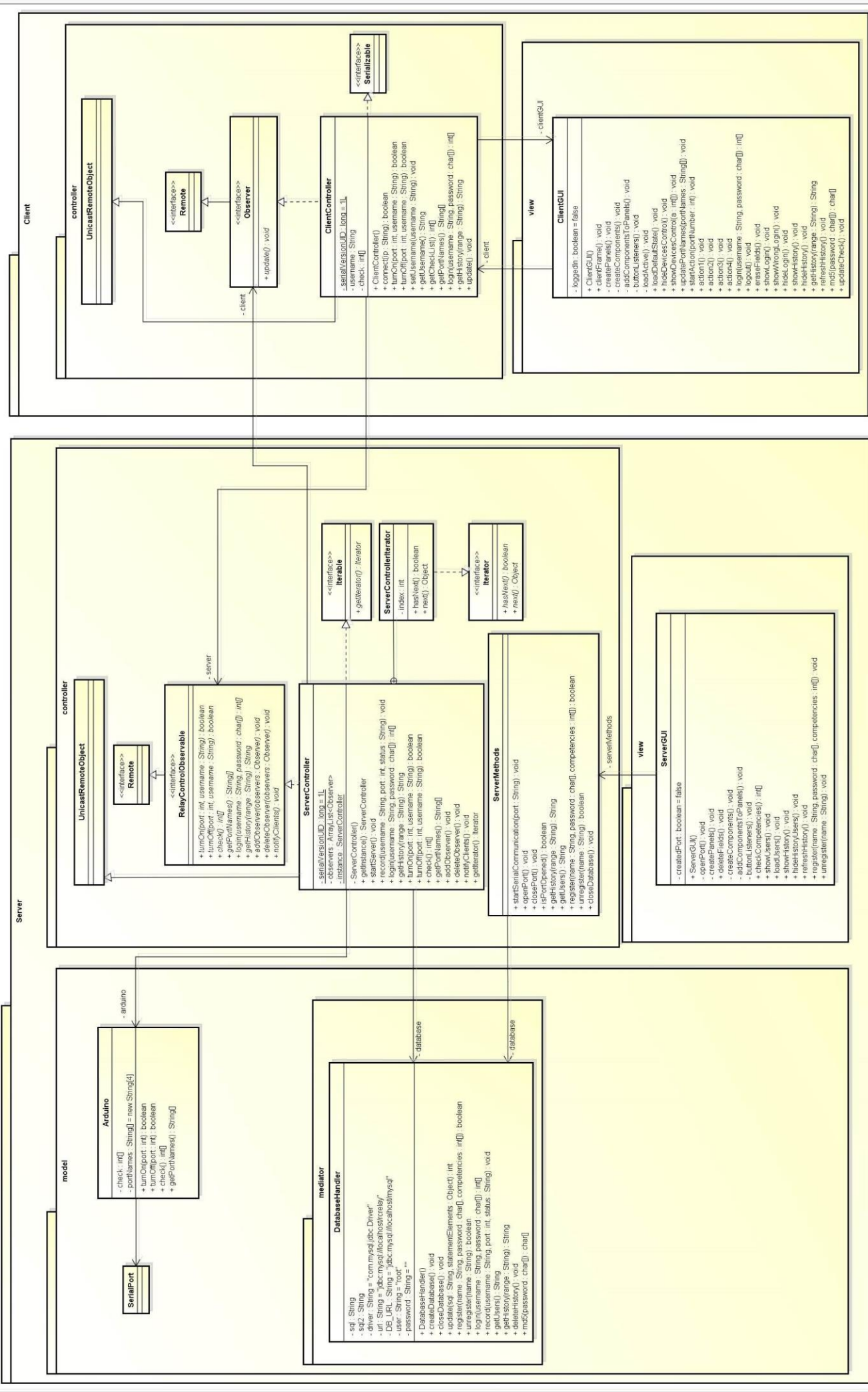
# Control Devices



# **CLASS DIAGRAMS**

RemoteControlRelay System





# TEST CASES

RemoteControlRelay System





## Use Case: Log In

---

Action	Description	Expected Result	Final Result
Log in	The User inputs a legal username and password.	The ports available for the account are loaded in the <i>Device</i> panel.	Working as expected
	The User inputs an illegal username and password.	Username and password text fields are cleared and a “ <i>wrong credentials</i> ” message is display below them.	Working as expected
	The User inputs an illegal username.	Username and password text fields are cleared and a “wrong credentials” message is display under them.	Working as expected
	The User inputs an illegal password.	Username and password text fields are cleared and a “wrong credentials” message is display under them.	Working as expected
	The User inputs the username only.	Username text field is cleared and a “wrong credentials” message is display under it.	Working as expected
	The User inputs the password only.	Password text field is cleared and a “wrong credentials” message is display under it	Working as expected
	The User does not input a username and password.	A “wrong credentials” message is display under the password text field	Working as expected

## Use Case: Log Out

---

Action	Description	Expected Result	Final Result
Log Out	The User presses the “LOGOUT” button	The main windows is refreshed. Text fields for the username and password appears and all 4 ports become inaccessible.	Working as expected

---

# Use Case: Register/Update User

Action	Description	Expected Result	Final Result
<b>Register/Update User</b>	The Administrator inputs a non-existing username, a legit password and also selects which ports the new account should control. <b>(Register User)</b>	The main window is refreshed and a list of all accounts appears. (The account is registered into the database)	Working as expected
	The Administrator inputs an existing username, a new password and also selects which ports the account should control <b>(Update User)</b>	The main window is refreshed, and a list of all accounts appears. (The account is updated with a new password and access rights)	Working as expected
	The Administrator inputs an existing username, the same password and also selects new ports which the account should control <b>(Update User)</b>	The main window is refreshed, and a list of all accounts appears. (The account is updated with new access rights)	Working as expected
	The Administrator inputs an existing username, the new password and also selects the same access rights <b>(Update User)</b>	The main window is refreshed, and a list of all accounts appears. (The account is updated with a new password)	Working as expected
	The Administrator inputs a non-existing username, an illegal password and also selects which ports the new account should control. <b>(Register User)</b>	The username, password fields are cleared and they turn red	Working as expected
	The Administrator inputs the username only <b>(Register User/Update User)</b>	Username text field is cleared and it turns red. The password text field also turns red	Working as expected
	The Administrator inputs password only <b>(Register User/Update User)</b>	Password text field is cleared and it turns red. The username text field also turns red	Working as expected
	The Administrator selects which ports the account should control. The text field for username and password remains empty. <b>(Register User/Update User)</b>	Password and username text field turn red	Working as expected
	The Administrator inputs an empty username, password and doesn't select none of the ports. <b>(Register/Update User)</b>	Password and username text field turn red	Working as expected

## Use Case: Delete User

---

Action	Description	Expected Result	Final Result
Delete User	The User inputs an existing username	The main window is refreshed and a list of all accounts appears. (The account is deleted from the database)	Working as expected
	The User inputs a non-existing username	The main window is refreshed and a list of all accounts appears. (Nothing is deleted)	Working as expected
	The Administrator inputs an empty username	Nothing happens	Working as expected

## Use Case: Show Users

---

Action	Description	Expected Result	Final Result
Show Users	The Administrator presses "USERS" button	The main window is refreshed and a list of all accounts appears.	Working as expected

## Use Case: Show History

---

Action	Description	Expected Result	Final Result
Show History	The Administrator/User presses 'HISTORY' button	The main window is refreshed and a list of all accounts appears.	Working as expected

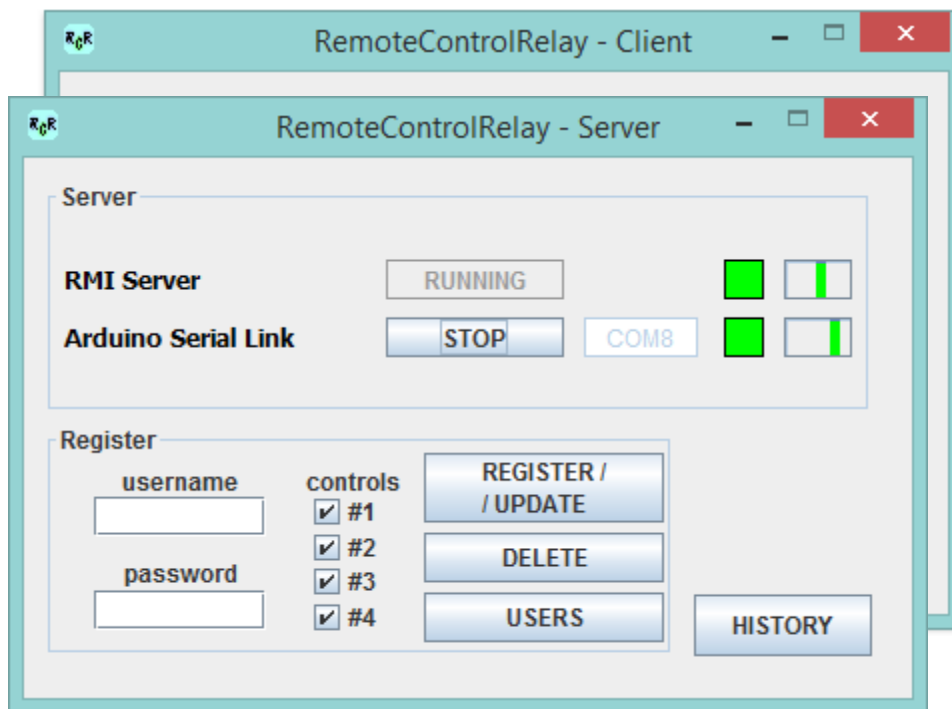
## Use Case: Start/Stop Serial Communication

---

Action	Description	Expected Result	Final Result
Start/Stop Serial Communication	The Administrator inputs a legal communication port name	In the Server panel, state indicator turns green	Working as expected
	The Administrator inputs an illegal communication port name	Nothing happens	Working as expected

# USERGUIDE

## RemoteControlRelay System



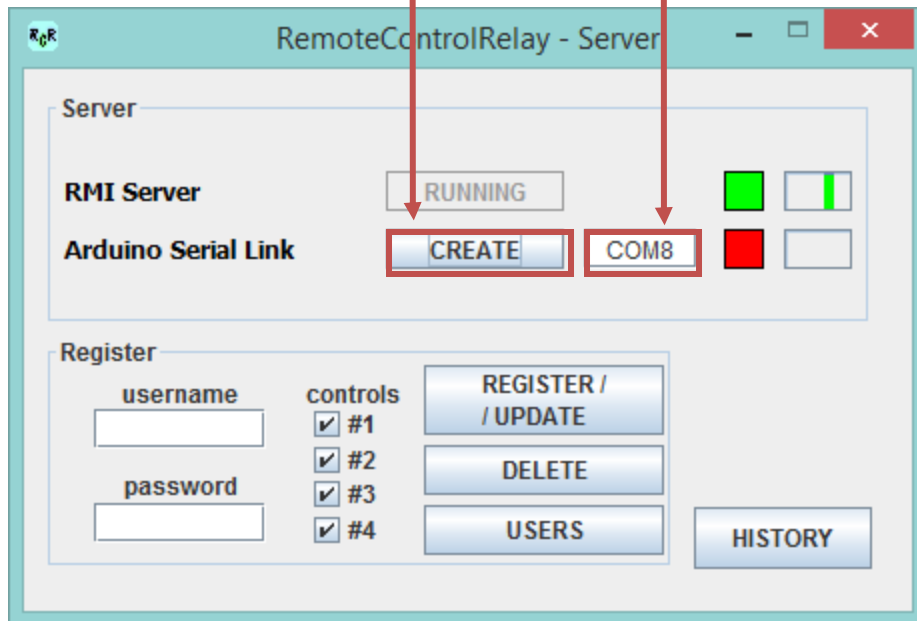
## Table of Contents

Start Serial Communication.....	3
Register a User .....	4
Update a User .....	5
Delete a User .....	6
Show Users .....	7
Show History .....	8
Control Devices .....	9

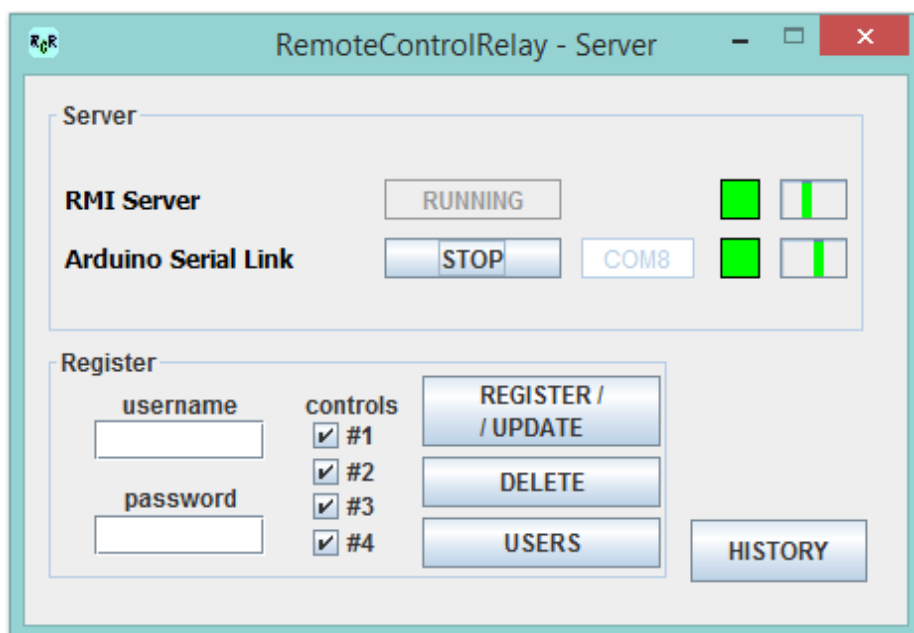
# Start Serial Communication

To allow users to control devices, we need to start serial communication between the program and Arduino. Only the system administrator can do this.

1. Make sure that **Arduino** is connected to the USB port of your computer.
2. Open **RemoteControlRelay - Server**.
3. Specify a correct COM port Arduino is connected to.
4. Press the **CREATE** button.



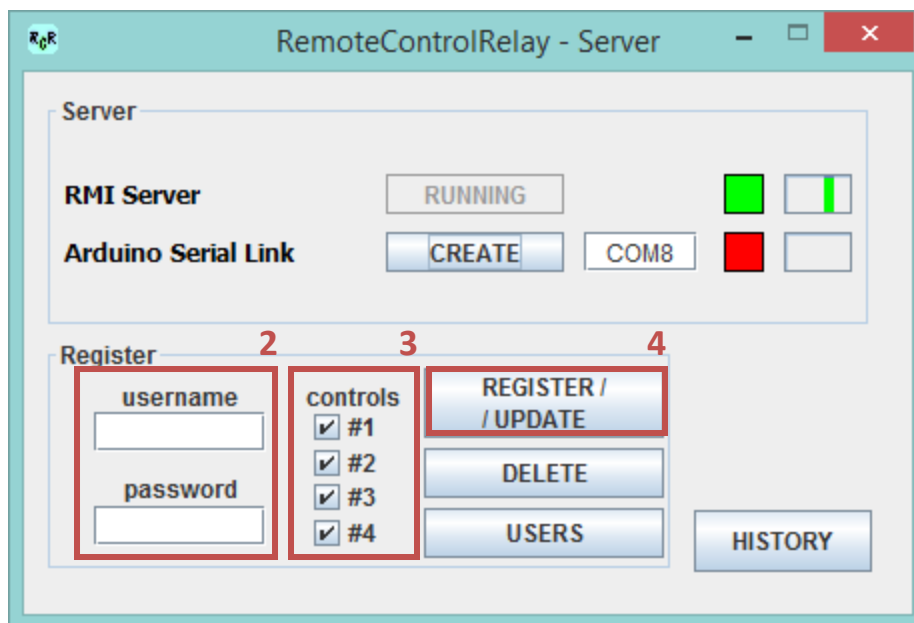
Communication link is created and started. Notice how the button changes from **CREATE** to **STOP**. You can now *Stop/Start* the communication freely.



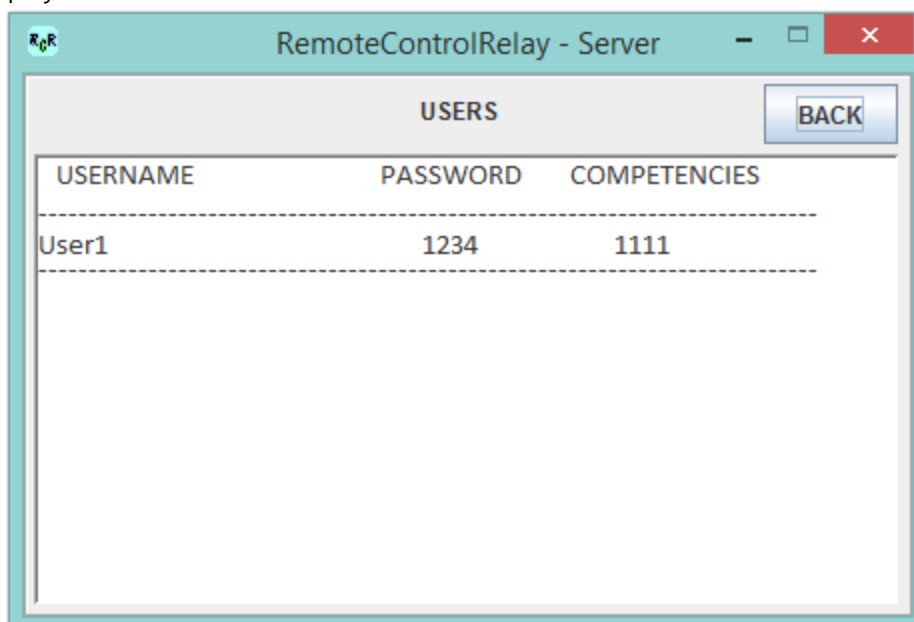
# Register a User

To allow users to control devices, we need to register them into a database. Only the system administrator can do this.

1. Open **RemoteControlRelay - Server**.
2. Enter a unique **username** and a **password** of maximum 4 characters.
3. Tick the boxes corresponding to the port numbers the user will be able to control.
4. Press the **REGISTER/UPDATE** button.



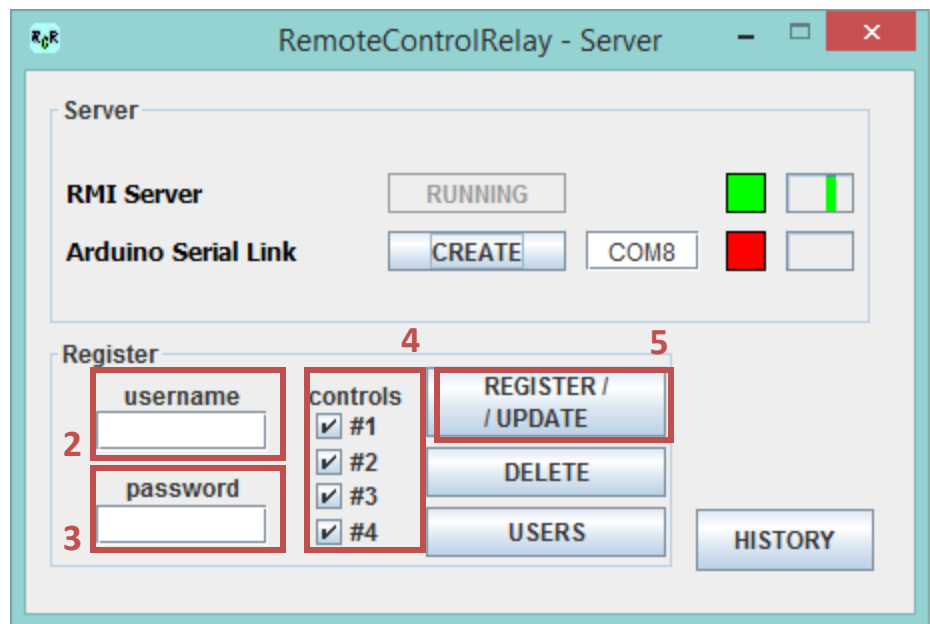
If the action is successfully completed, a window containing a table of all registered users (including the new one) is displayed.



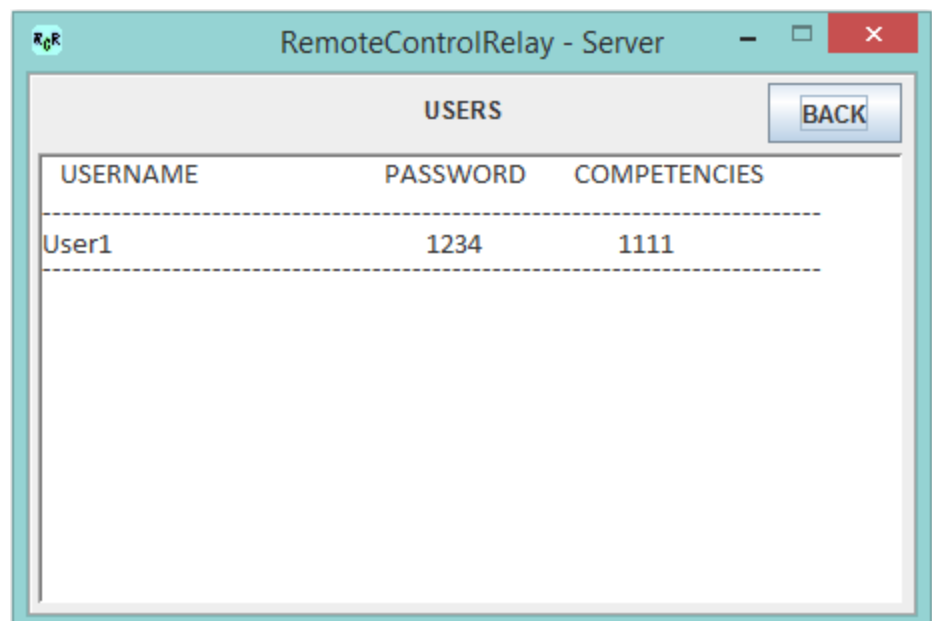
# Update a User

The update function is used in cases when a user's password must be changed or some modifications to access rights must be made. Only the system administrator can do this.

1. Open **RemoteControlRelay – Server**.
2. Enter the **username** of an already registered user.
3. Enter a new password.
4. Tick the boxes corresponding to the port numbers the user will be able to control.
5. Press the **REGISTER/UPDATE** button.



As when registering a user, the action is considered successful if a new window with a table of all registered users is shown.

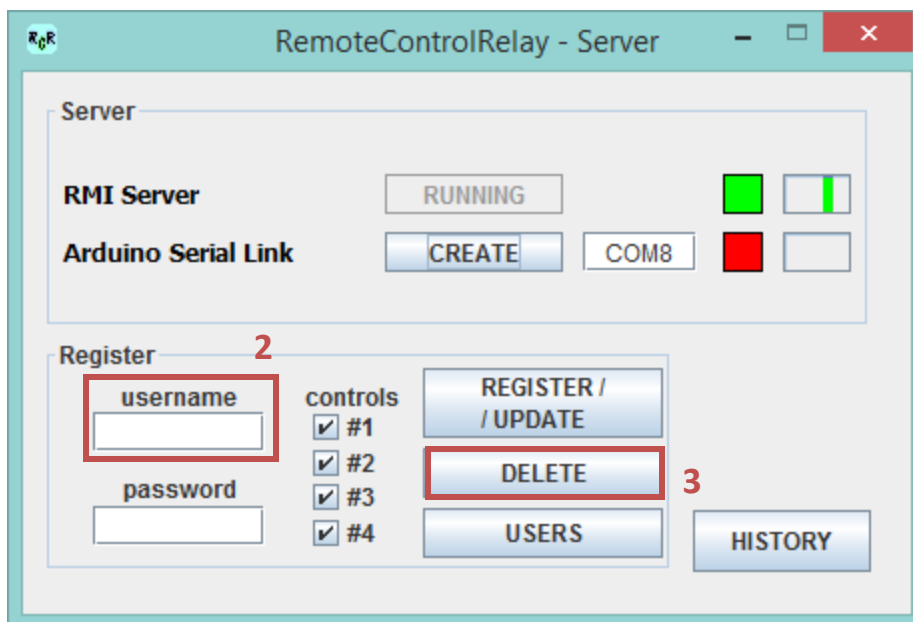




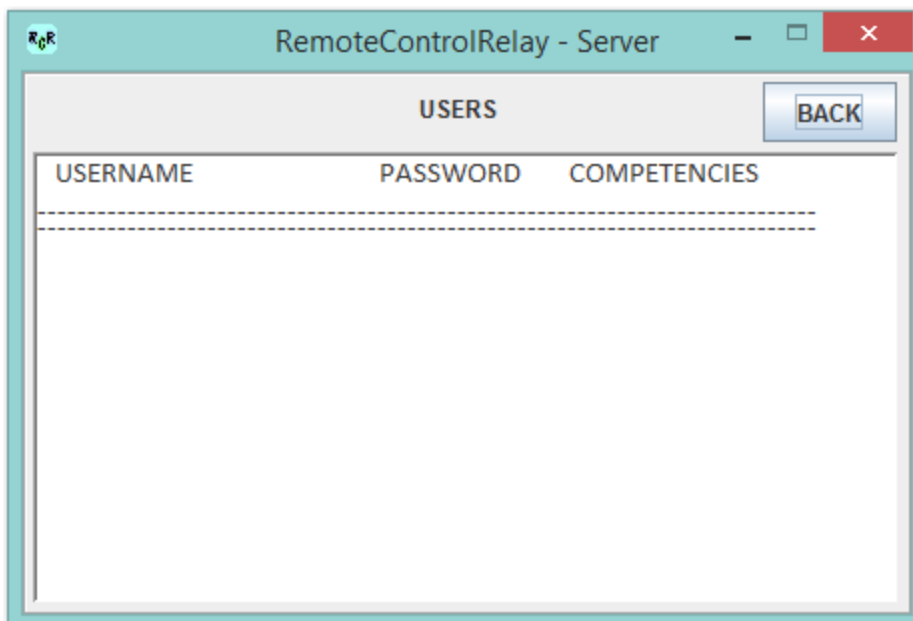
# Delete a User

This action removes a user from the system, but maintains their records in the database. Only the system administrator can do this.

1. Open **RemoteControlRelay - Server**.
2. Enter **username** of the existing user.
3. Press the **DELETE** button.



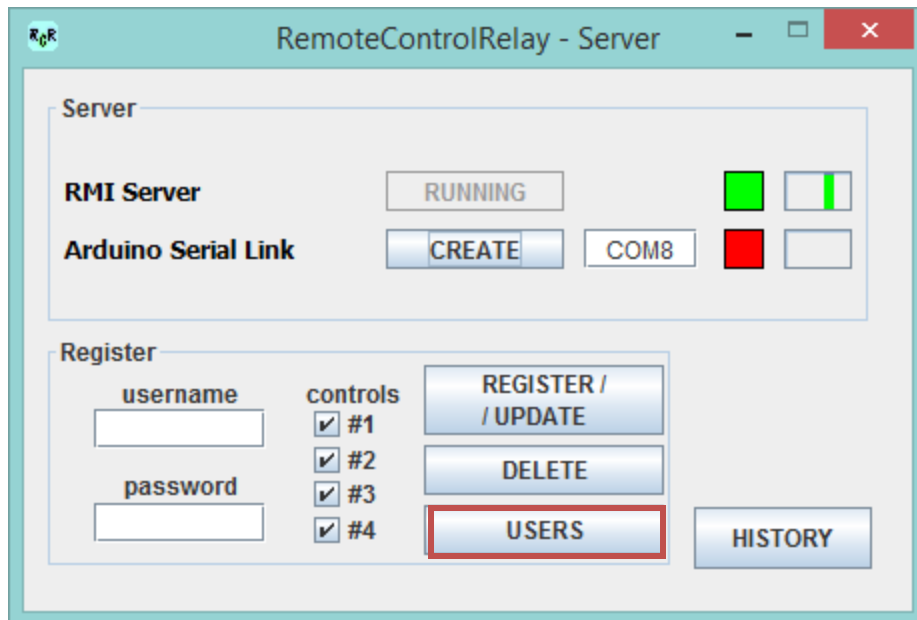
The user is no longer registered and the updated table of all users is displayed.



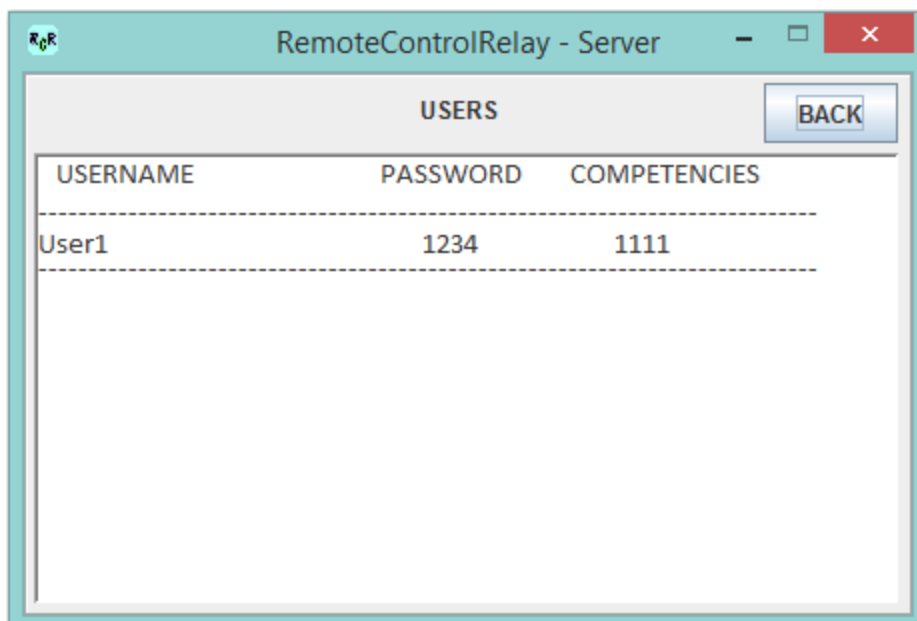
# Show Users

Displays a table of all the users registered in the system, along with their passwords and access rights.

1. Open **RemoteControlRelay – Server**.
2. Press the **USERS** button.



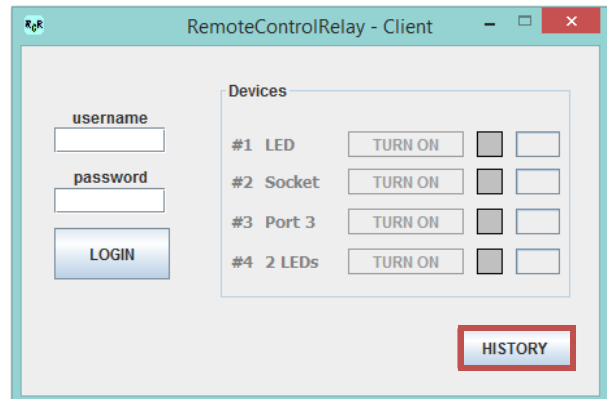
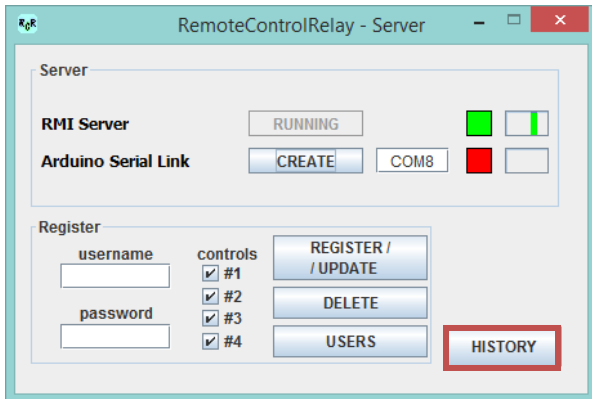
Result:



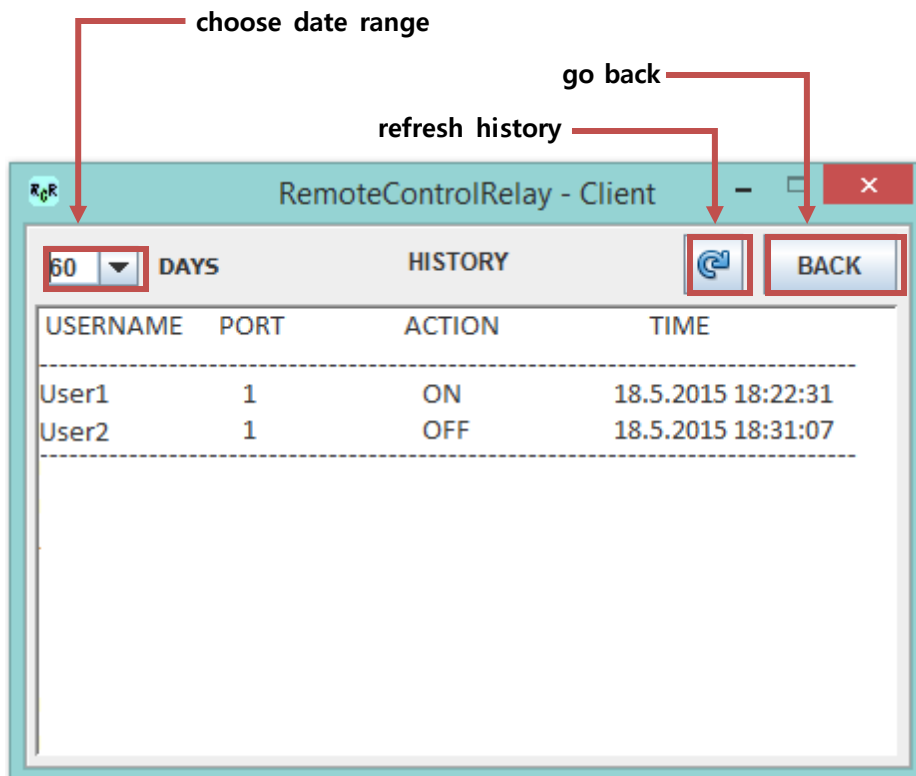
# Show History

Displays a history log with all the actions performed by all users in the past two months. The table specifies the user, the port, the action, and date/time. This action can be done from both *server* and *client* side.

1. Open **RemoteControlRelay – Server** or **RemoteControlRelay – Client**.
2. Press the **HISTORY** button.



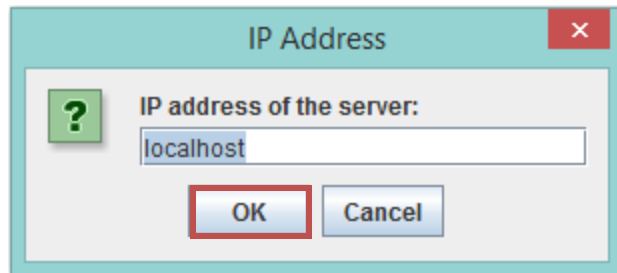
After history window is shown, the following actions are available:



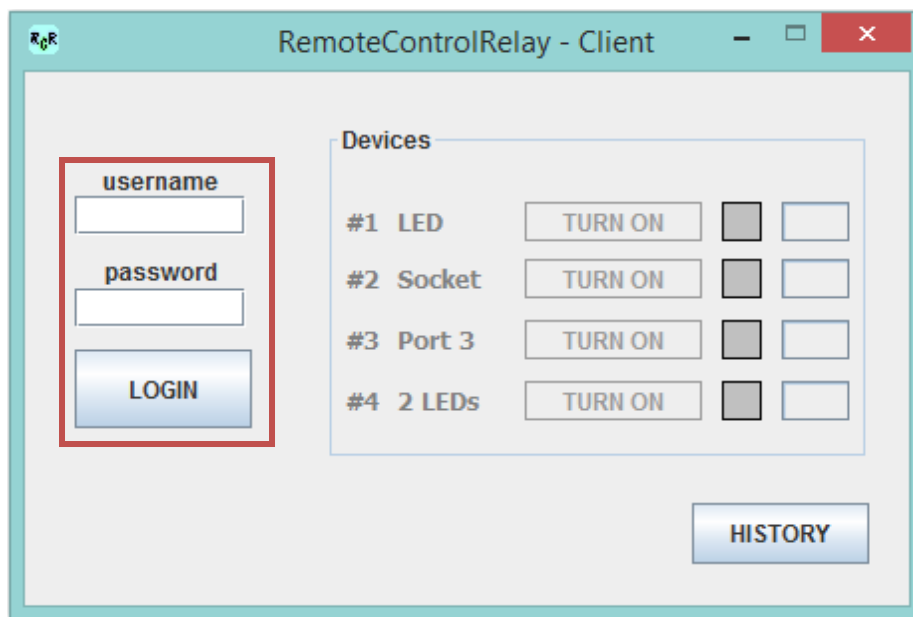
# Control Devices

Shows how to login to an account and control devices remotely. The *RemoteControlRelay - Server* must be running with a serial link created.

1. Start *RemoteControlRelay - Client*.
2. Specify the IP address of the server.
3. Press **OK**.

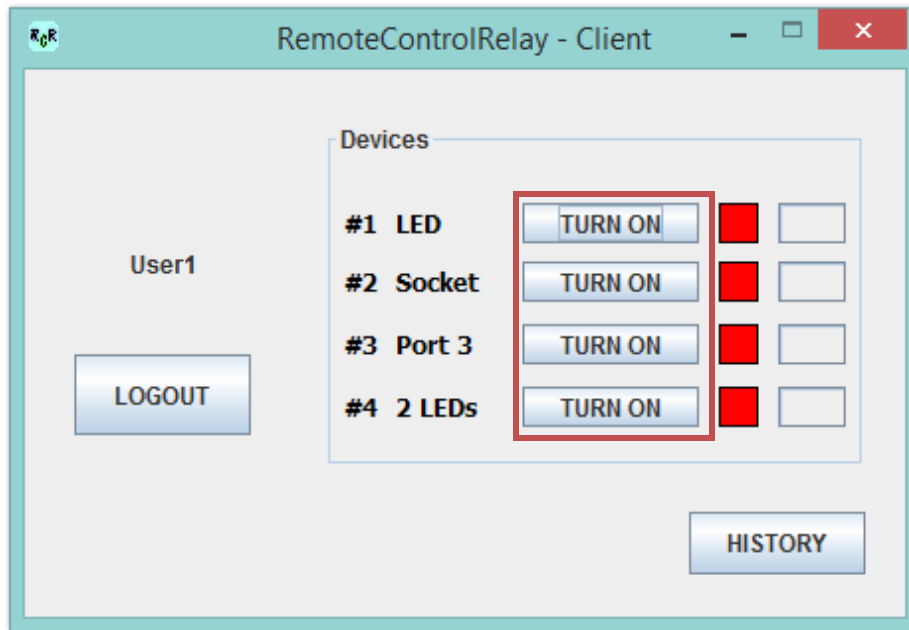


4. Enter a *username* and *password*.
5. Press the **LOGIN** button.



Communication with the server is established and the ports available for the account are loaded. The restricted ports are displayed in gray.

3. Choose a port to control and press the corresponding **TURN ON** button. If the port is already on, a **TURN OFF** button will be displayed instead.



The device connected to the port is turned on and indicators turns green.

4. Press the **TURN OFF** button to disable the device.

