Project Report

Client Server

Online Banking System

ICT ENGINEERING

Semester Project 2

Tor Frøstrup Jacobsen (261565)

Faizan Yousaf (260101)

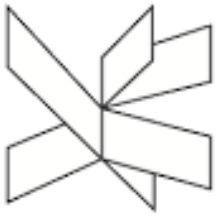
Hristo Rumenov Getov (260064)

Mohamed Guudow Sheikh Ismali Ali (162569)

Supervisor:

Troels Mortensen (TRMO)
Jens Cramer Alkjærsig (JCA)

15.12.2017



VIA University College

Abstract

This project report is aimed to provide an overview throughout the development of an Online Banking System, that prevents the use of non-dedicated applications such as internet browsers to perform their banking tasks. This includes transferring money, checking balance, maintaining their personal information, keeping track of their accounts. It also includes the ability of an administrator, registering new customers, create new accounts for customers and fulfil more administration related tasks.

This solution was found by carefully analysing and evolving the major requirements, using unified process along with SCRUM.

Java, Remote Method Invocation, JavaFX and design pattern like Model-View-Controller and adapter, are some of technical components that were used to develop the system.

The program was testes successfully, against the requirements, using scenario testing.

The result is a functional system, that fulfils the most important requirements.

Index

Abstract 1
1 Introduction 3
2 Requirements 4
2.1 Functional Requirements 4
2.2 Non-Functional Requirements 4
3 Analysis 5
3.1 Use case diagram 6
3.2 Use case description 7
3.3 Class diagram 8
4 Design 9
4.1 Class Diagrams 9
4.2 Client-Server Connection 10
4.3 Design Pattern 11
4.4 Sequence Diagram 14
4.5 GUI Design 15
4.6 Data model 20
5 Implementation 24
5.1 Shared 24
5.2 Client 26
5.3 Server 27
6 Test 35
7 Result & Discussion 37
8 Conclusion 38
9 Project Future 39
10 References 40
11 List of Appendixes 41

1 Introduction

In present days technology takes a tremendous role in the daily routine. People are connected to internet all the time and the need of fast, easy to use and secure applications is higher than ever.

Many banks often experienced problems, caused by the lack of fast, secure and easy to use online system. A online banking system which prevents the need to use traditional internet browser to have access it and which allows customers to check account balance, check transactions history, make transactions and contact the support via email is developed. For this system, administrator users are also created, who can create and fix (if necessary) the customer users and accounts. The admin users are also able to create new admin users and change them.

For security reasons customers are not going to be able to apply for loan , invest money or apply for credit card through the online system, since it requires additional check through bank employees. A double security check is done before logging in using Nem-ID, which makes the system even more secure. The system is not going to provide recovery of a forgotten password , since Nem-ID is being used to login.

The following pages explains in depth how the system was developed.

2 Requirements

The requirements have been divided into two parts non-functional and functional requirements and sorted based on their important.

2.1 Functional Requirements

1. A User can use the system from multiple computers
2. An Admin-User can create a new Customer-User
3. An Admin-User can create an account
4. An Admin-User can add account to Customer-User
5. A User can log in
6. A User can log out
7. A Customer-User can transfer money to a different account
8. A Customer-User can receive money from different account
9. A Customer-User can see balance on each account
10. A Customer-User can see transfer history of every account
11. A Customer-User gets interest on his money
12. A Customer-User can share account with another User
13. A Customer-User can transfer money on a chosen date
14. A Customer-User can change his/her personal information
15. A Customer-User can save transaction history
16. A Customer-User can see his account information
17. A Customer-User contact the bank with email
18. A Customer-User can print his history(pdf)
19. An Admin-User can see the transfer history of an account
20. An Admin-User can see the balance of a customer-user
21. An Admin-User can change a User information
22. An Admin-User can create new Admin-User

2.2 Non-Functional Requirements

1. The system must be developed in Java
2. Program must have a server connection using either RMI or sockets.
3. Server must have a PostgreSQL database

3 Analysis

Nowadays it is very important for banks to have an online tool, which provides all their customers with possibility to operate with their money. It is also very important for this tool to be secure for the customer and the bank.

For creating this online tool, a bank would need an entire system, which is going to keep track of all customers and their accounts, all transactions made between accounts. For that purpose, this system will need a database where it is going to store whole information about the customers their accounts and transactions. The system needs to be managed by special users called admins. That means beside information about customers, accounts and transactions database must store information about admins. The information about admin will be the same as for customer information with only difference, an Admin is going to have an employee number.

The system also needs a user interface, which provides the customers to manipulate their accounts (make transactions, check balance, check information about account). The customer's interface will differ from admin's. The admin will have the possibility to create different users (Customers or Admins), create accounts or change the data of all existing users or accounts.

To sum it up, the program will have to keep track of the following data:

- Person: CPR-No, first name, last name, birthday, gender, address, postal code, city, country, nationality, phone number and email.
- Admin: Everything from person + employee number.
- Customer: Everything from person + accounts and saved transactions.
- Account: Name, Account number, registration number, balance, transfer limit, credit limit, transactions and persons who the account is shared with.
- Transaction: Account number, registration number, description, transfer date, amount.

3.1 Use case diagram

From the above-mentioned requirements, it's possible to obtain 17 use cases as shown in the use case diagram (Figure 3.1.1).

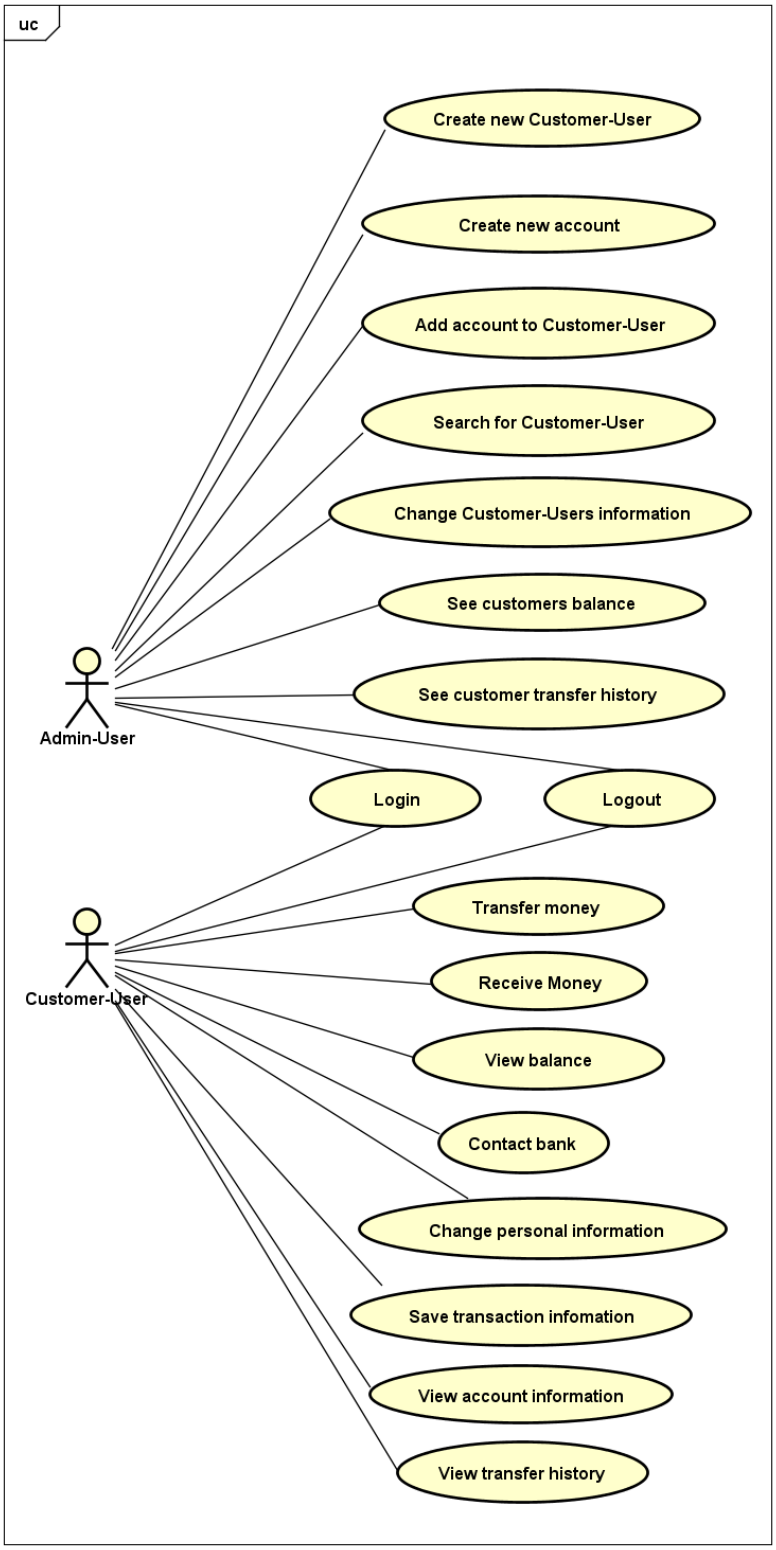


Figure: 3.1.1: *Use case diagram*

This diagram also shows what the program must be able to do and what functionalities an admin and a customer user should have. (Figure can also be found as an .png file in the appendix B.)

3.2 Use case description

This page shows a table containing the use case description for the “Login” (Figure 3.2.1) use case. This use case is one of the most complex and most important for the program.

ITEM	VALUE
UseCase	Login
Summary	An admin or a customer user can login using username, password and NEM-ID
Actor	Admin-User Customer-User
Precondition	
Postcondition	The user gains access to either the customer or the admin side of the system.
Base Sequence	1. The user chooses between administrator or customer. 2. The user types in username and password. 3. The user presses the "Login" button. 4. The program send the information to the server. 5. The server check in the database if the user exist and if the password is correct. 6. The server returns a NEM-ID value to the client. 7. The program displays the NEM-ID value in a dialog window. 8. The user types in the key. 9. The user presses the "Submit" button. 10. The program send the data to the server. 11. The server check if the key is correct. 12. The server returns an admin or a customer object. 13. The program displays the customer or admin GUI.
Branch Sequence	
Exception Sequence	Username or password wrong: 1-3 as base sequence. 4. Program display a label stating that the username or password is wrong. Use case ends. Wrong NEM-ID key: 1-9 as base sequence. 10. Program display a label stating that the NEM-ID is wrong. Use case ends.
Sub UseCase	
Note	

Figure 2.2.1: *Use case description - Login*

Use case descriptions for the use cases: “Create new user”, “Add account” and “Transfer money” can be found in the Appendix C.

3.3 Class diagram

This page shows a simplified version of the class diagram (Figure 3.3.1) It is only based on the use case diagram and descriptions, mentioned above.

It also shows the relationships and relationships types between the different classes, this is providing the base of the online banking system.

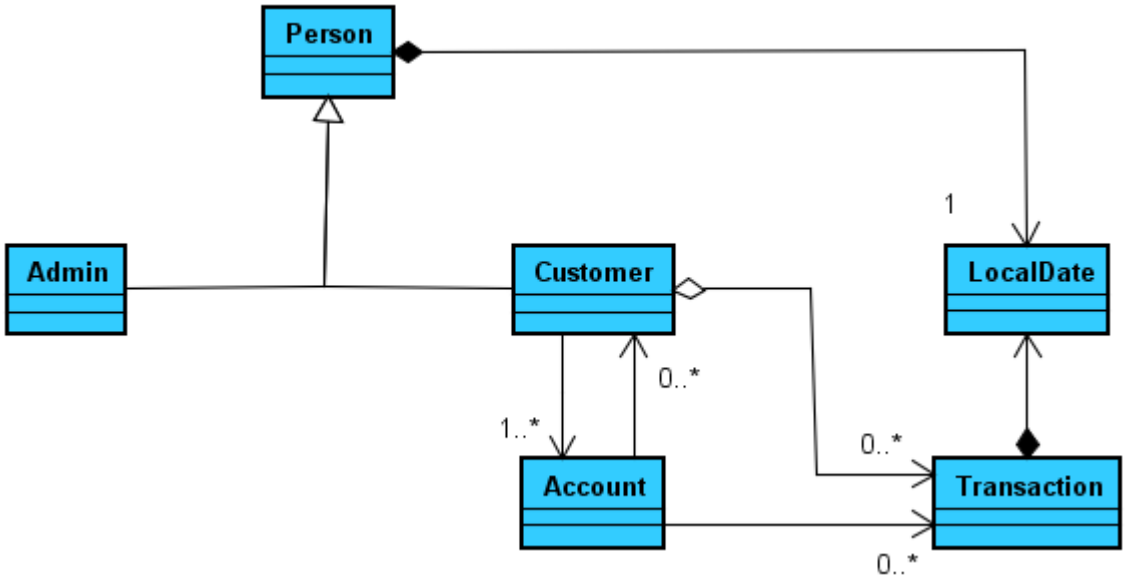


Figure 3.3.1: *Simplified Class Diagram*

The full class diagram, including the remaining classes, can be found in the appendix D.

4 Design

This section of the report explains how the system is structured. The server-client connection, the design pattern used, the class diagram, the GUI design choices along with the database models, will be explained in this part, based on the requirements giving in the analysis section.

4.1 Class Diagrams

On this page, the full design class diagram is shown, not including fields and methods (Figure 4.1.1)

The program was build using an object-oriented approach.

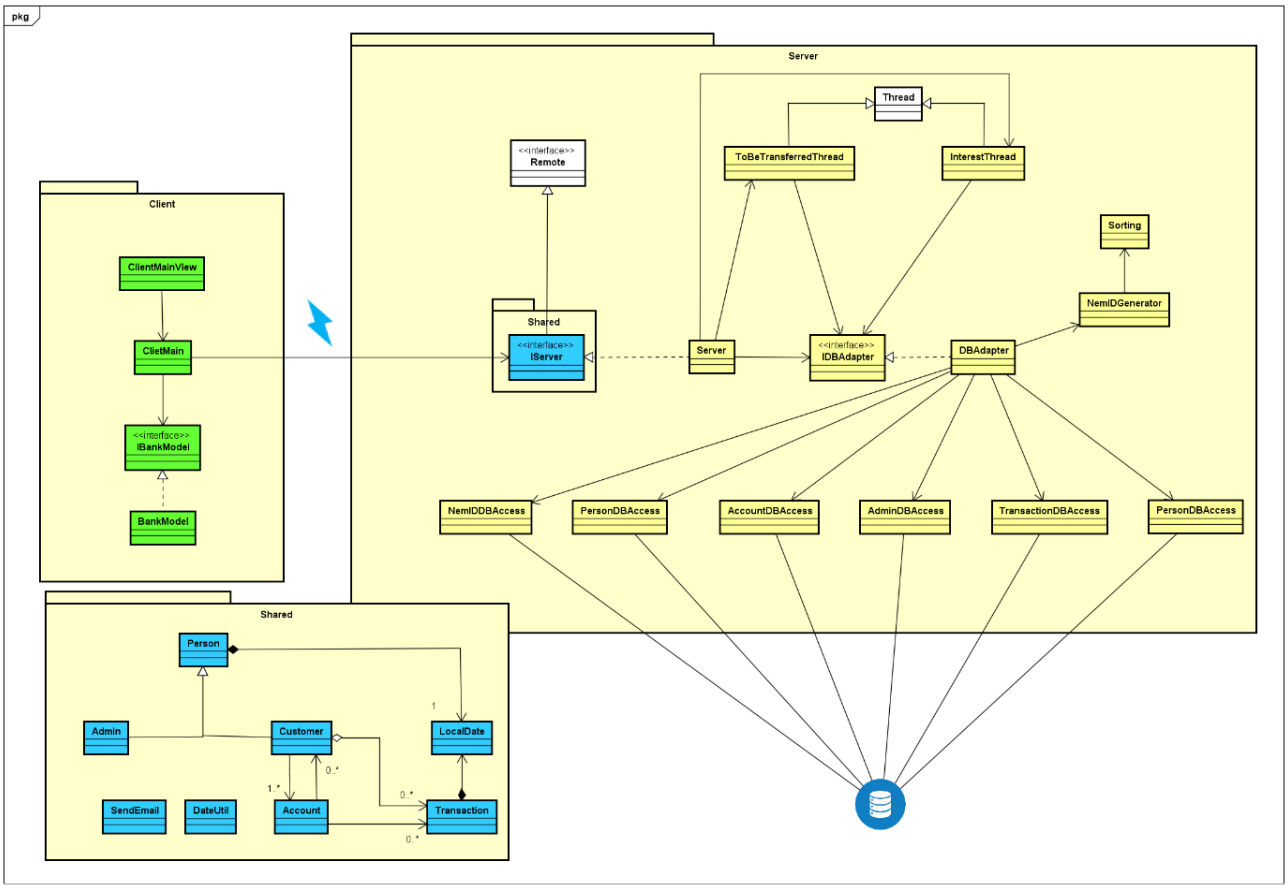


Figure 4.1.1: *Design Class Diagram*

The full class diagram can also be found in the appendix D.

4.2 Client-Server Connection

For the connection between the server and the client Remote Method Invocation (RMI) was used.

The idea behind using RMI is to create a simple client-server connection that will satisfy the requirement, to make it possible to use the system from multiple computers. For this to work the client should get the same data from the server after logging in, from any computer using the program.

For the connection between the server and client to work, the server should be running, so the client can contact the registry of the server, when the program is being started. The client will then receive instances of server stubs from the registry. Which the client can use to call methods on the server interface.

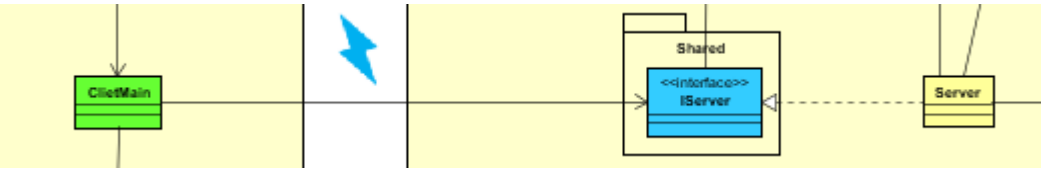


Figure 4.2.1: *Client-Server Connection*

For the connection to work, two different classes and one interface were used:

- ClientMain (Client Class)
    - Uses the IServer interface to call methods on the ServerMain.
    - Responsible for asking for data and receiving data from the server.
- IServer (Shared Interface)
    - Used by the client to invoke methods on the ServerMain class.
- ServerMain (Server Class)
    - Implements the IServer interface.
    - Responsible for saving, collecting and sending data to the clients.

4.3 Design Pattern

For the development of the system three different design pattern were used:

Model-View-Controller (MVC), Data Access Object (DAO) along with the Adapter design pattern. Below is explained where these design pattern were used and why they were used.

Model View Controller:

Model View Controller is used to separate the responsibility of the classes inside the client side of the system.

The MVC consist of three parts, each with their own responsibility.

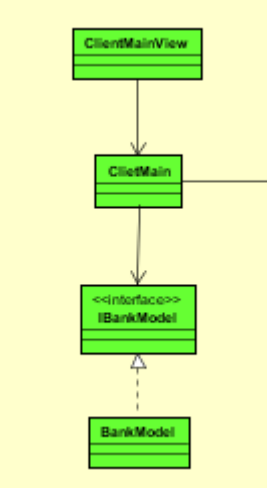


Figure 4.3.1: *Model-View-Controller*

The model side holds all the data, that the client gets from the server. It also holds all the changes that may be done before the data is send back to server to save it in the database.

The view part is responsible for displaying all the information from the model in the GUI. Along with getting user input from the GUI, that can then be forwarded to the controller.

The controller works as a translator between the model, and the view. It also holds the RMI connection to the server, so it can directly send and/or get data from the server.

Data Access Object:

Data Access Objects are used for data management.

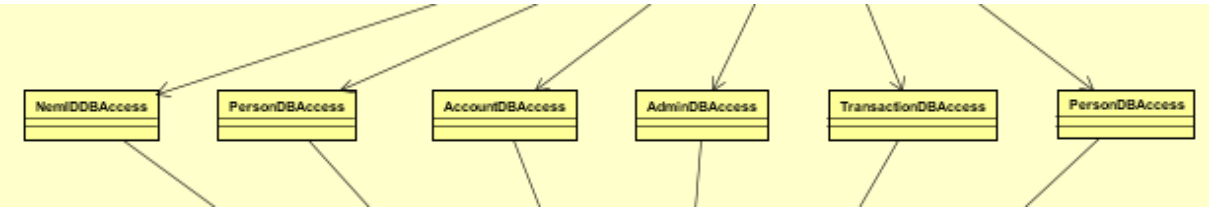


Figure 4.3.2: *Data Access Object*

On the server side of the system DAO is used for retrieving data from the database. On the client side it is being used inside the model (MVC) to keep the necessary data in the application while it is running along with changing it and deleting the data after a user logs out.

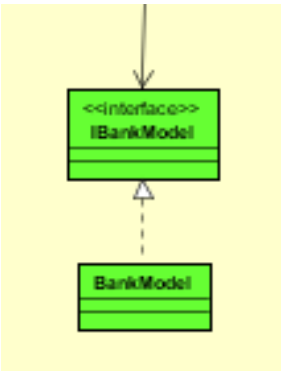


Figure 4.3.3: *Data Access Object*

If for some reason the way the data is being stored, would change , it wouldn't affect the classes using the DAO interfaces only the classes implementing the interface would need to be changes.

Adapter:

The adapter design pattern is be used to make classes work together, that normally wouldn't be able to because of incompatible functionalities. In this system it is used on the server side as a translator between the ServerMain and the JDBC classes, to make it possible to save the data in the database. The ServerMain accesses the adapter through an interface (IDBAdapter). All the JDBC classes have instances of them in the DBAdapter class.

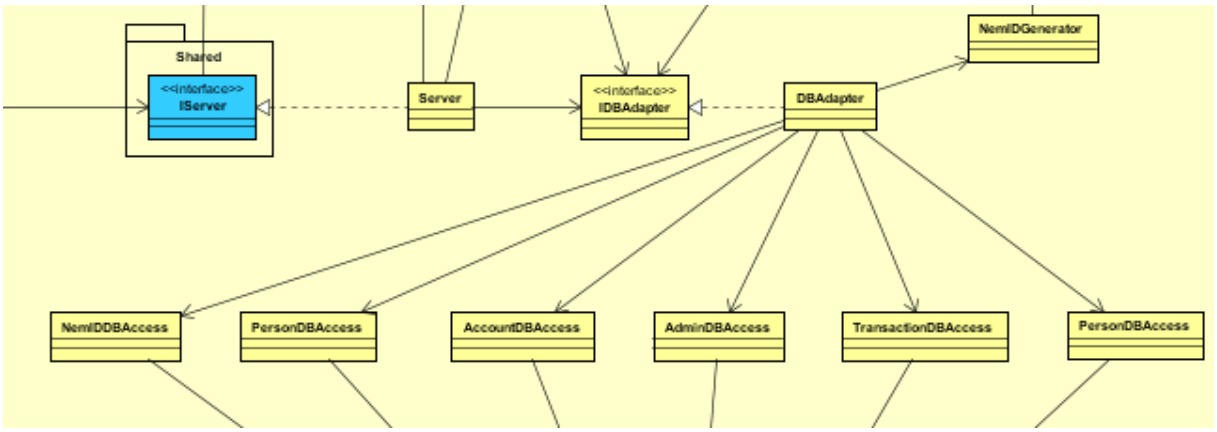


Figure 4.3.3: *Adapter design pattern*

4.4 Sequence Diagram

The figure below shows the sequence diagram for the use case “Login” used by a customer user.

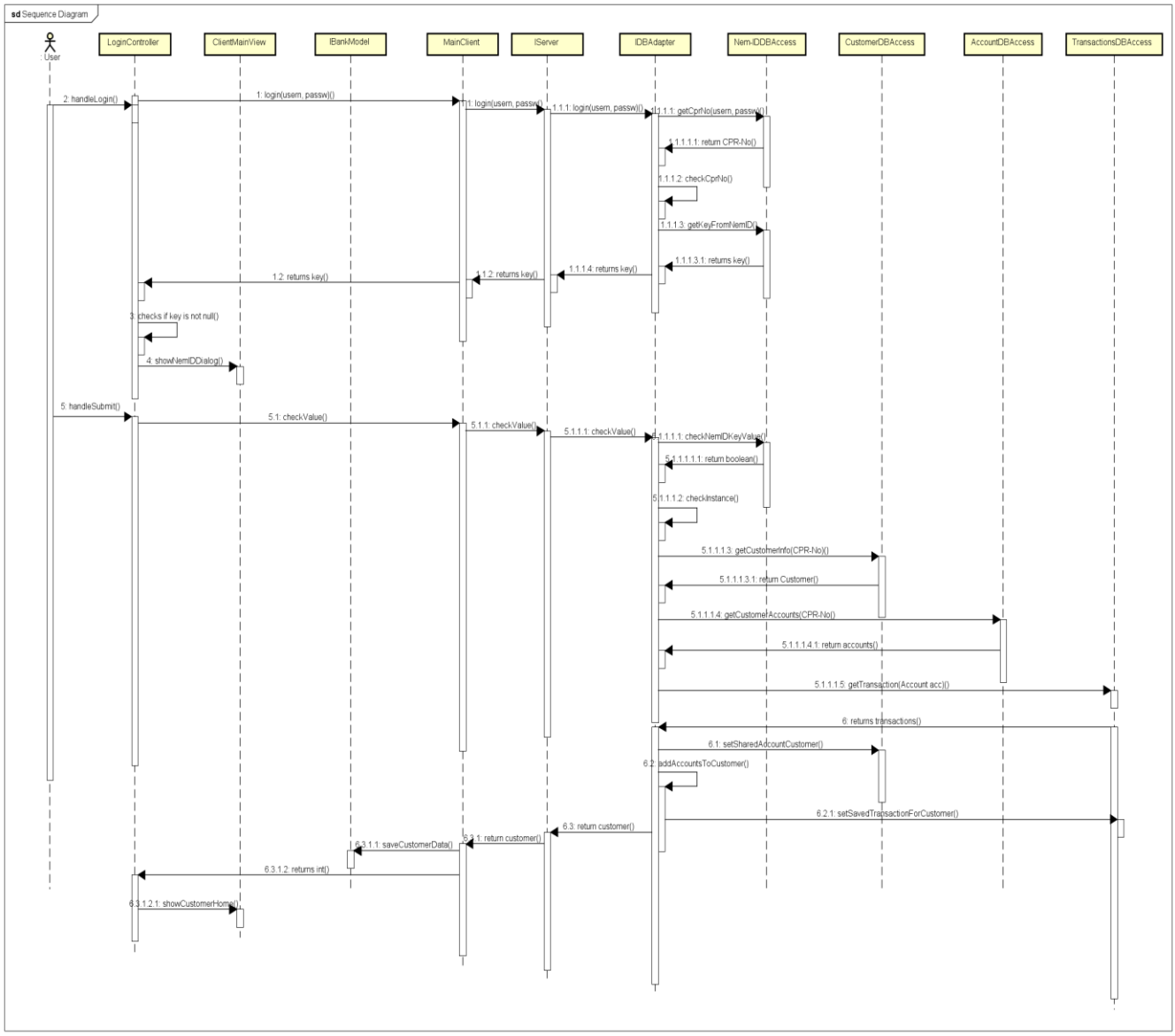


Figure 4.4.1: *Sequence Diagram - Login*

The sequence starts by the user clicking the login button, after putting in the correct username and password. The sequence diagram can also be found in the appendix E as a .svg and .png file.

4.5 GUI Design

This part will explain how the GUI has been designed and how the program is supposed to work.

For the client application to work the server must be running. When starting the program, the "Login" page will be displayed as shown below in figure 4.5.1.

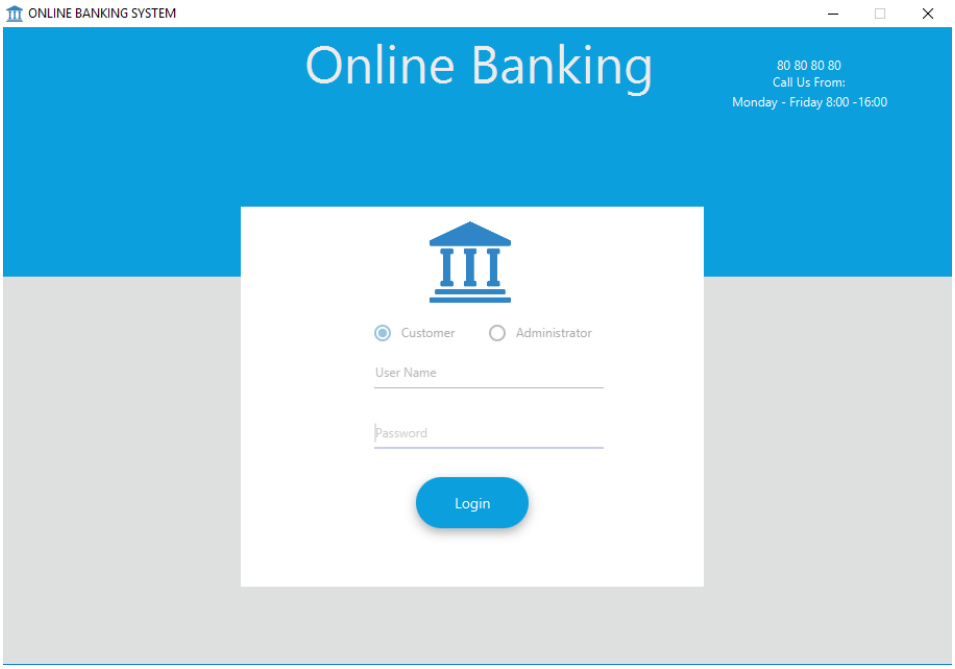


Figure 4.5.1: *Login Panel*

Here the customer user, as well as the admin user, can log in to the either the customer side or the admin side of the system. For a user to log in, the correct username and password must be used, afterwards a new dialog will show up. Here the user will need to put in the NEM-ID value for the key that is being showed in the dialog. (Figure 4.5.2)

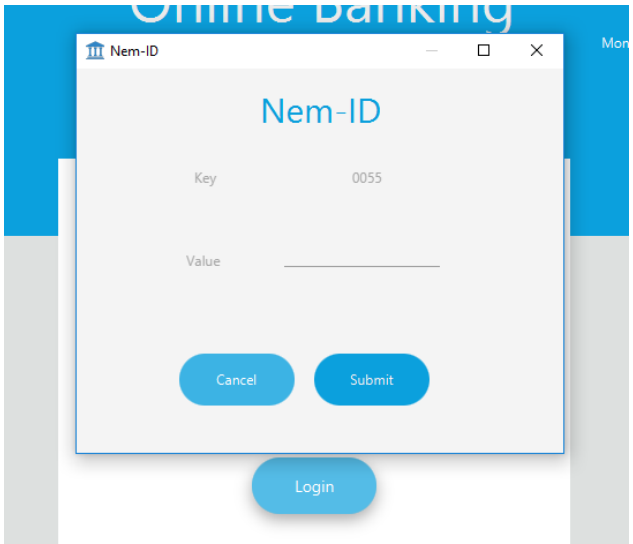


Figure 4.5.2: *NEM-ID dialog*

If a admin is logged in, the “Create New User” (figure 4.5.3) page will show up first. Here the admin can create either a new customer or a new admin user. If the user that is being created, hasn’t got an NEM-ID, a new one will also be created.

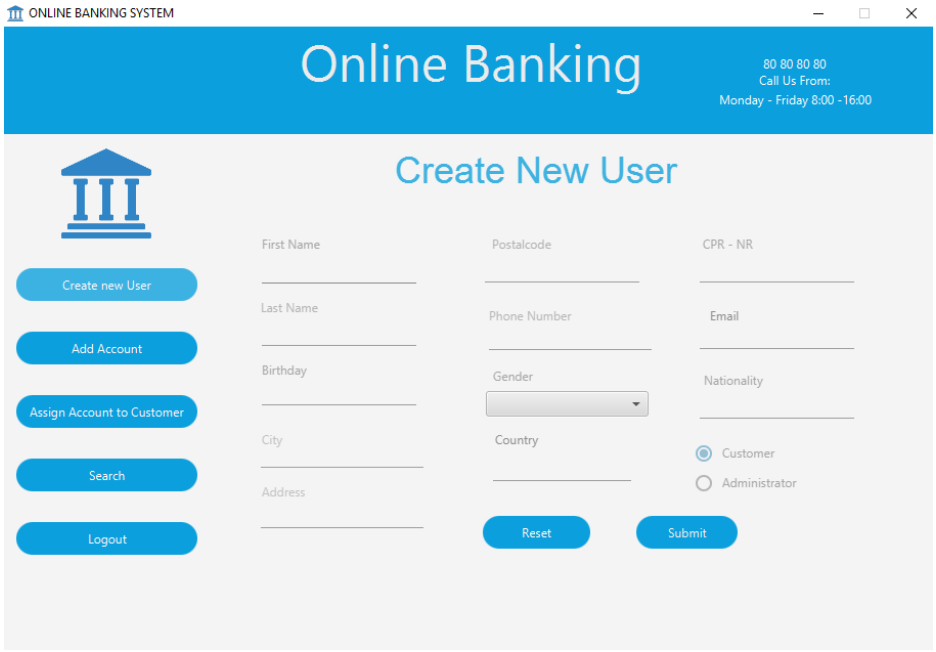


Figure 4.5.3: *Create New User panel*

On the left side the admin user has a menu navigation. Here the user can navigate between the “Create New User”, “Add Account”, “Assign account to customer” and the “Search” panels.

The “Add Account” page is used to create a new account and assign it to a customer user.

The “Assign account to customer” page is used to assign an already existing account to another user so the it can be shared among two or more users.

The last panel “Search” (figure 4.5.4) is for looking up detail about customer user as well as making it possible for the admin user to edit the information saved about the user. By clicking the “Show Accounts” button a new panel will show up where the balance of each account can be seen. By clicking on the “See transactions” button, the admin can also see all the transaction made or received by the selected user.

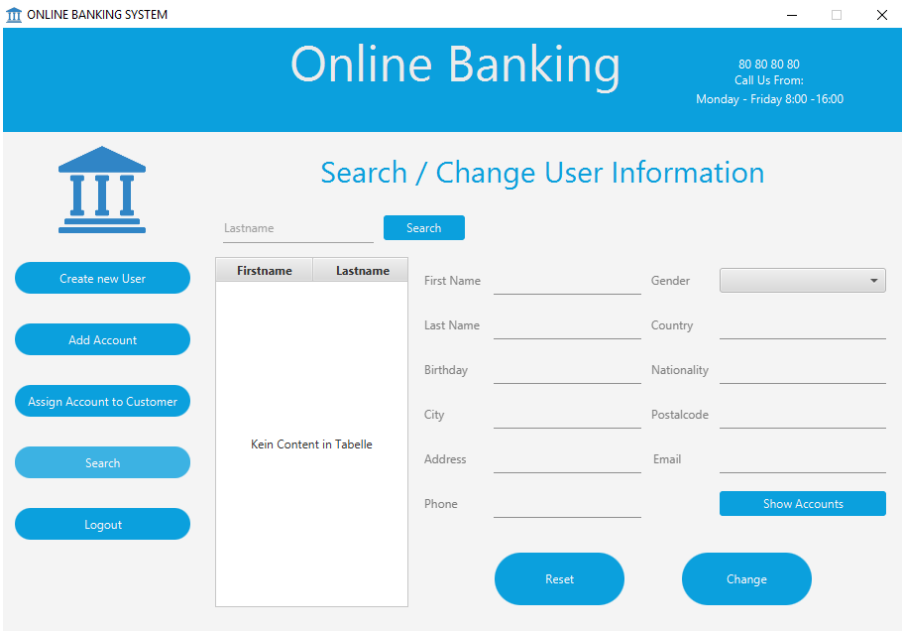


Figure 4.5.4: *Search/ Change User Information panel*

The logout button is used for the admin to logout of the admin side. After logging out the "Login" panel will show up again.

In case a customer is logged in, the first page that will show up is the "Home" panel. (Figure 4.5.5)

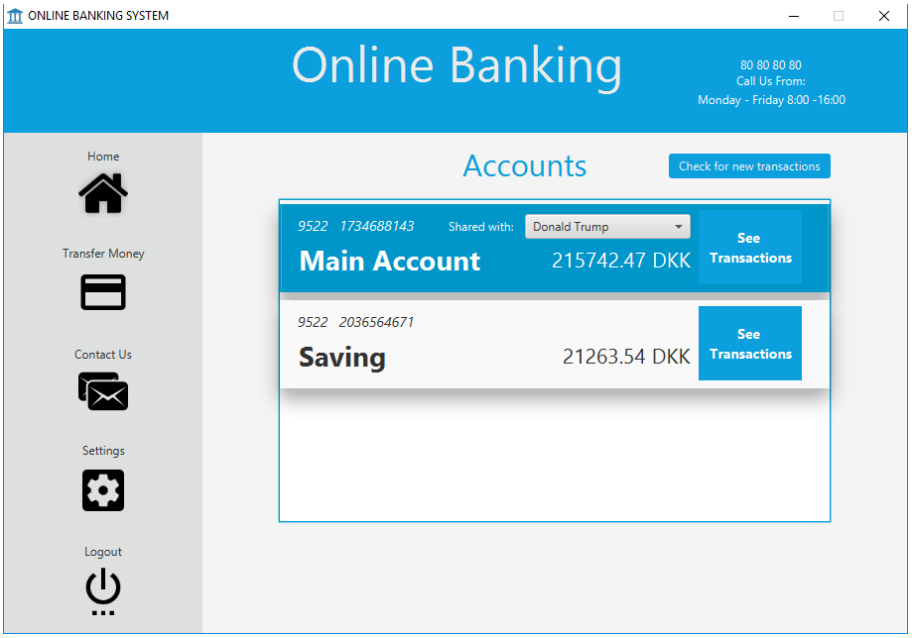


Figure 4.5.5: *Home panel*

Here the customer user can see all of his/her accounts and the balance on each of them. By clicking on the "Show Transaction" button, the user can see all transaction made from a specific chosen date. (Figure 4.5.6)

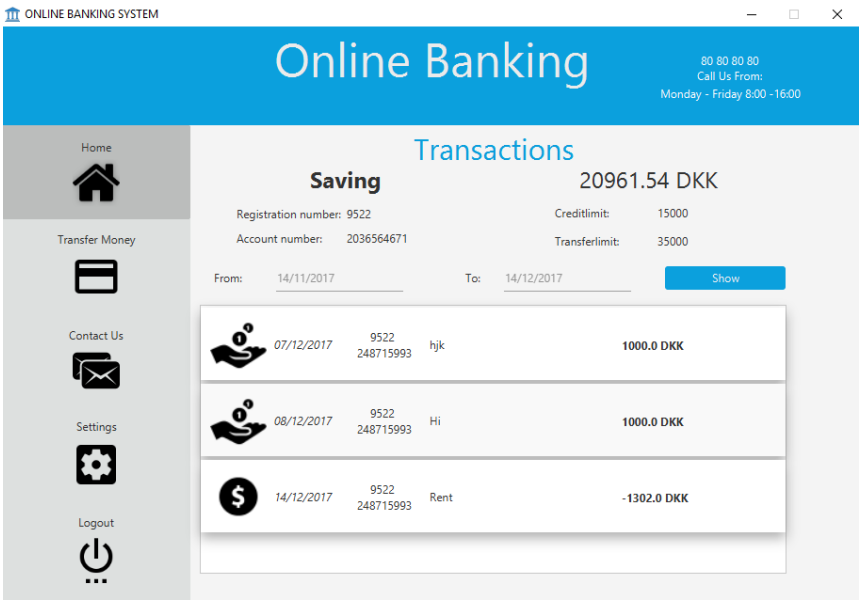


Figure 4.5.6: *See transaction panel*

The customer user also has a menu navigation, where the user can choose between the "Home", "Transfer Money", "Contact Us" and "Settings" panel. The logout button is also inside the menu. If the user presses this button, all the information kept in the application, about the user will be deleted and the "Login" panel will show again.

In the "Transfer Money" page the user can transfer money to another account, by filling in the field correctly and then providing the password for extra security. The user can also use previous saved transaction information by selecting one in the choice box at the top and the pressing the "Apply" button.

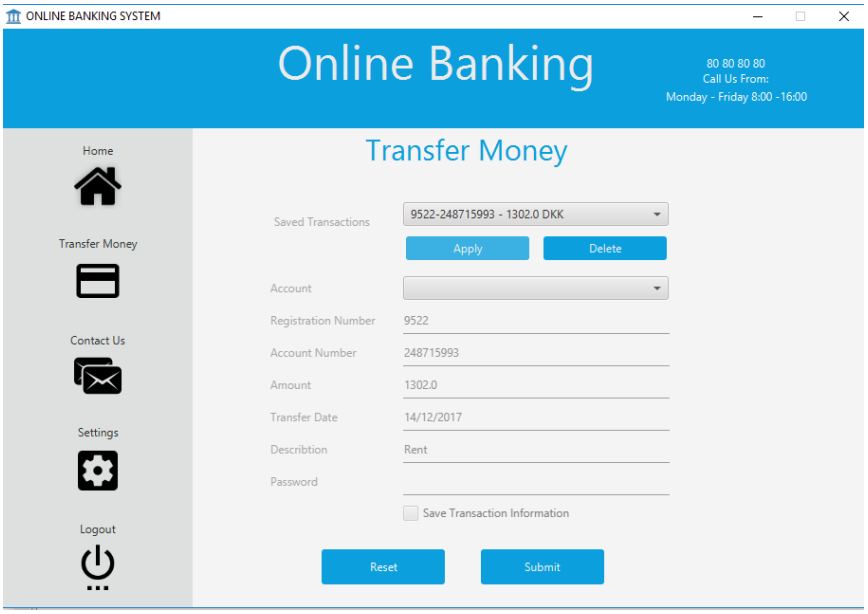


Figure 4.5.7: *Transfer Money panel*

The page "Contact Us" (Figure 4.5.8) can be used by the customer user to contact the bank by email. The name, phone and email will automatically be filled out by the application, getting the data from the model, where the customer information is being saved after login. The user can also choose the option to get a copy of the email send to their email.

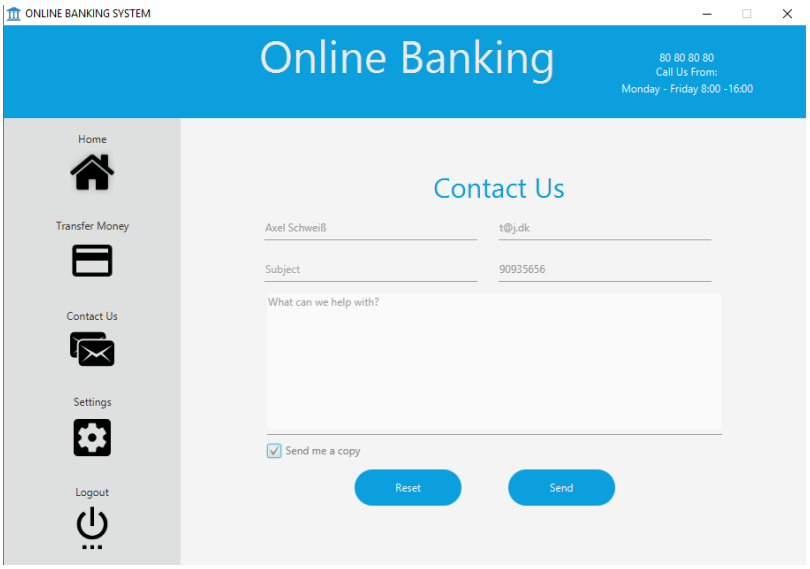


Figure 4.5.8: *Contact Us panel*

The last panel "Settings" (Figure 4.5.9) makes it possible for the user to change the information saved about him/her. But not all information can be changed by a customer user, for example birthday and gender only a admin user can change it. The user can also change the name of each account that is assigned to him/her in this panel.

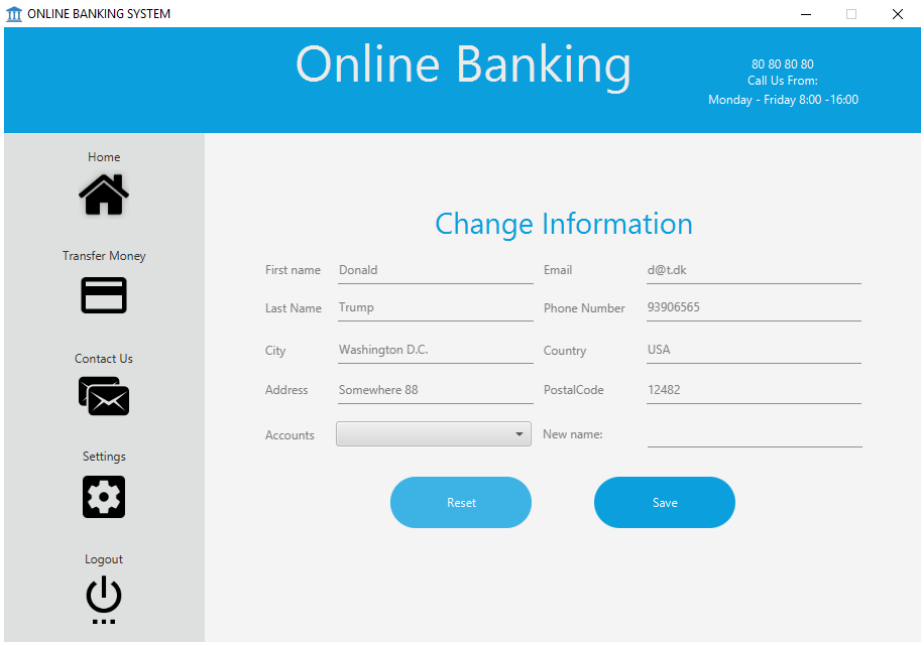


Figure 4.5.9: *Change Information panel*

4.6 Data model

Based on the required information for person relation was created, which holds all the person's information, CPR number(CPR_No), which is a primary key, first name (fname), last name (lname), date of birth (DOB), gender (Gender), address (address), postal code (postalCode), city (city), country (country), phone number (phoneNo), email (email), nationality (nationality).

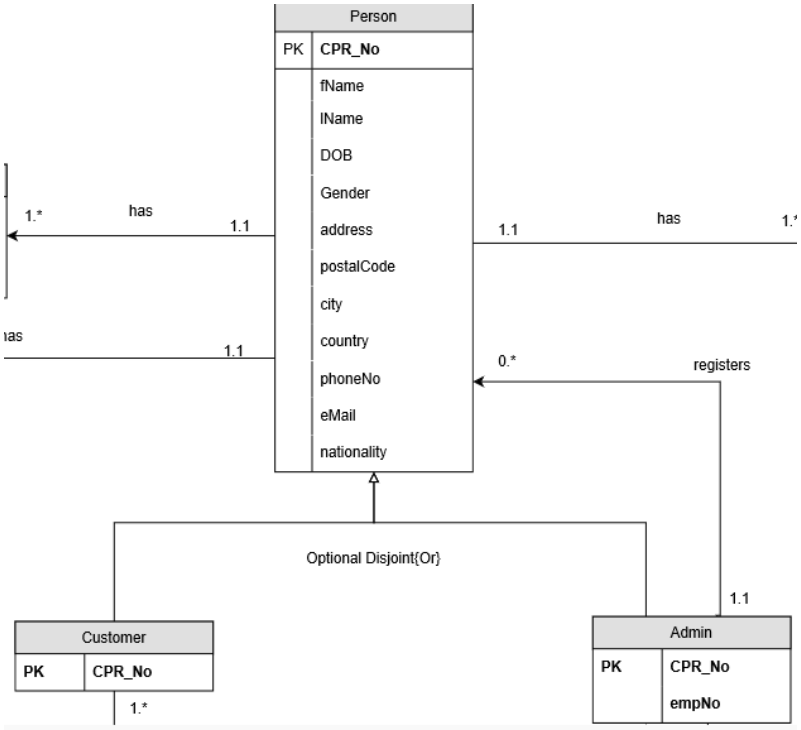


Figure 4.6.1: *Person, Customer and Admin relations.*

Since in the system, users are divided into two types (Customer and Admin), two more tables were created Customer and Admin. Customer table contains CPR number (CPR_No), which is a foreign key and primary key for the table. And the Admin table contains CPR number (CPR_No) as a primary and foreign key and employee number (emp_No). The connection is between Person-Customer and Person-Admin is "Optional Disjoint{Or}", because a person could be Admin or Customer or both Admin and Customer.

A relation between Admin and Person table was created (registers), because the admin can create a Person (Customer or Admin). This relationship is one to many (1…*) – one Admin can create many Persons (Customer, Admin).

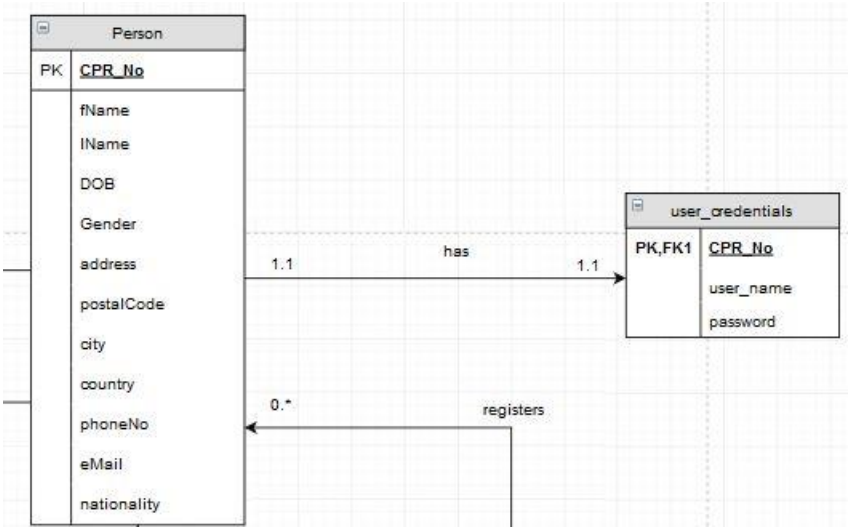


Figure 4.6.2: *User_Credentials relation*

A user_credentials table was created to holds the user name and password for a person (Customer or Admin). The relationship (has) between Person and user_credentials is one to one (1…1) – One person has one user credential (username and password). The table contains CPR_No, which is a primary key and foreign key (reference to Person), user name (user_name) and password (password).

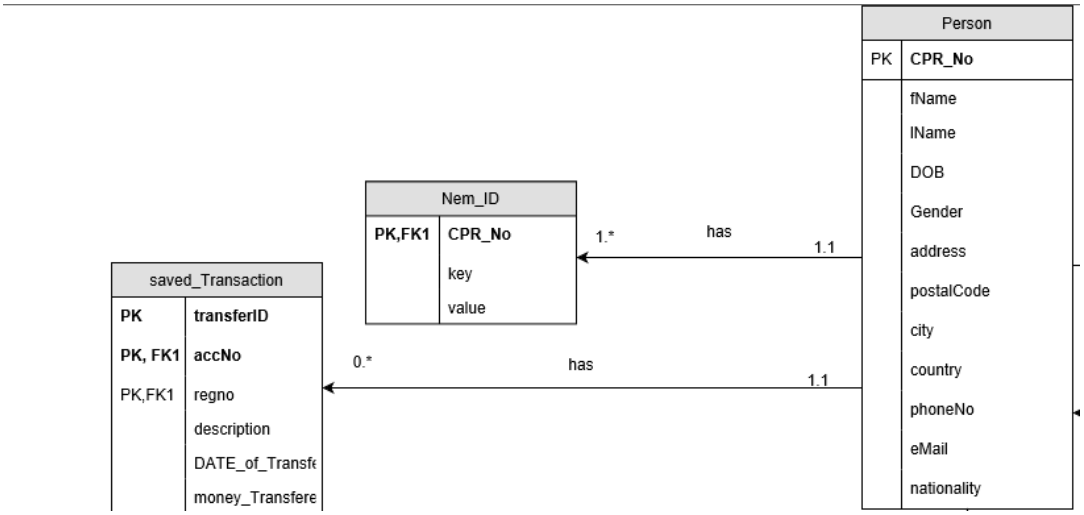


Figure 4.6.3: *NEM-ID, saved_Transaction relations*

Nem_ID table was created to contain all existing Nem ID's in the system. The relationship (has) is one to many (1…*) – One customer has many Nem_ID keys. The table contains CPR number (CPR_No), which is a primary key and foreign key, key (key), value (value).

saved_Transaction table was created to hold all the information about the receiver's account, to which customer made a transaction. Person table has a relationship (has) with saved_Transaction table. This relationship is one to many (1…*) – one person has many saved transactions (information). Saved_Transaction table contains transfer Id, which is Primary key, account number (accNo), which is also primary key and foreign key, description (description), date of the transfer (DATE_of_transfer) and the money which were transferred (money_Transfered).

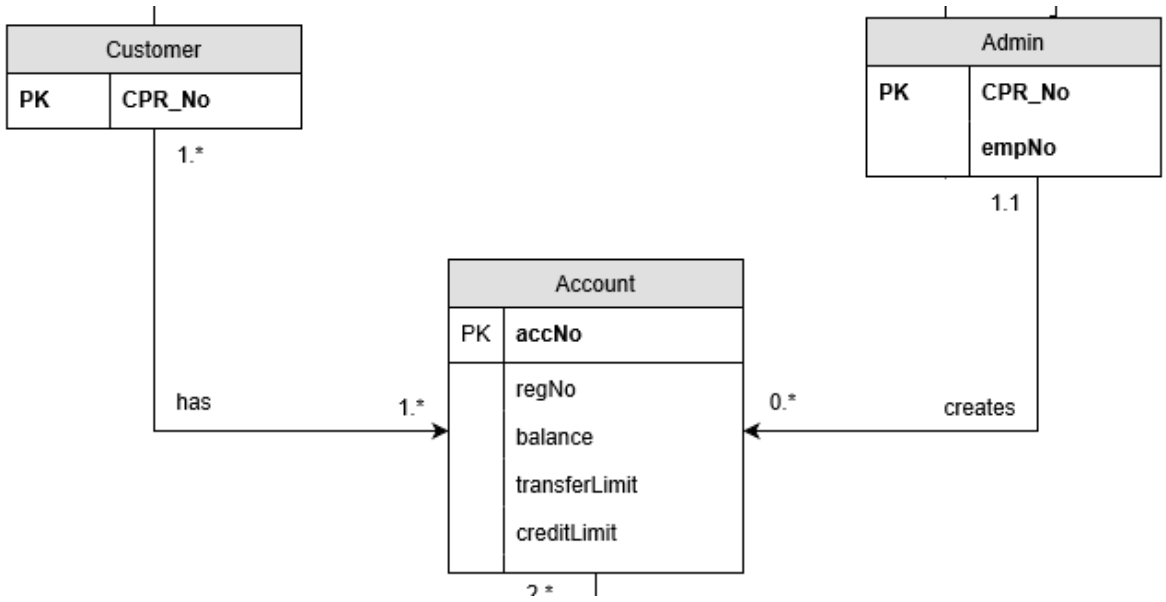


Figure 4.6.4: *Account relation*

Another table was created, which holds all accounts. This table contains account number (accNo), registration number (regNo), which together are the primary key, balance (balance), transfer limit (transferLimit) and credit limit (creditLimit). Admin table has also relationship (creates) to Account table. The relationship is one to many (1…*) – one Admin can create zero or many accounts. Customer table has a relationship (has) to Account many to many (*...*) one customer can have one or many account. As a result, from this relationship another table is created - Customer_Account. This table contains CPR number (CPR_No) of the customer, which is Primary key and Foreign key and the account number (accNo) of customer's account, which is Primary key and Foreign key.

A table Transaction was created for holding all transactions made by customers. The table contains transfer ID (transferID), which is a primary key, account number (accNo1) of the customer's account, registration number (regNo1) of the customer's account, account number (accNo2) of the receiver's account, registration number (regNo2) of the receiver's account, description (description) if there is any, date of the transaction has been made (date_of Transfer), the amount of money has been transferred (money_Transfered). There is a relationship (has) between Account and Transaction. This relationship is two to many (2…*) – at least two accounts have one or many transactions.

A table tobe_transfered was created to hold all transactions, which has to be made in a different date then current. The table contains same attributes as Transaction table.

The full EER diagram can be found in appendix H.

5 Implementation

All classes are implement in accordance with the class diagram which can be found in the appendix D.

The classes and interface necessary for the Client-Server connection were implemented before any other, making the core of the system.

Afterwards, in each sprint (SCRUM) more use cases were implemented and added to the core of the system, each building upon the previous added functionalities.

For the implementation, the classes were divided into three main parts: Client, Server and Shared. Below each part will be explained how it was implemented, in more detail.

5.1 Shared

The classes inside this package are shared by both the client and the server.

The classes Person, Customer, Admin, Account and Transaction all needs to implement the java.io.Serializable to make it possible to send and receive these classes from the server or client.

"Serialization in Java is a mechanism of writing the state of an object into a byte stream." – javatpoint.com.

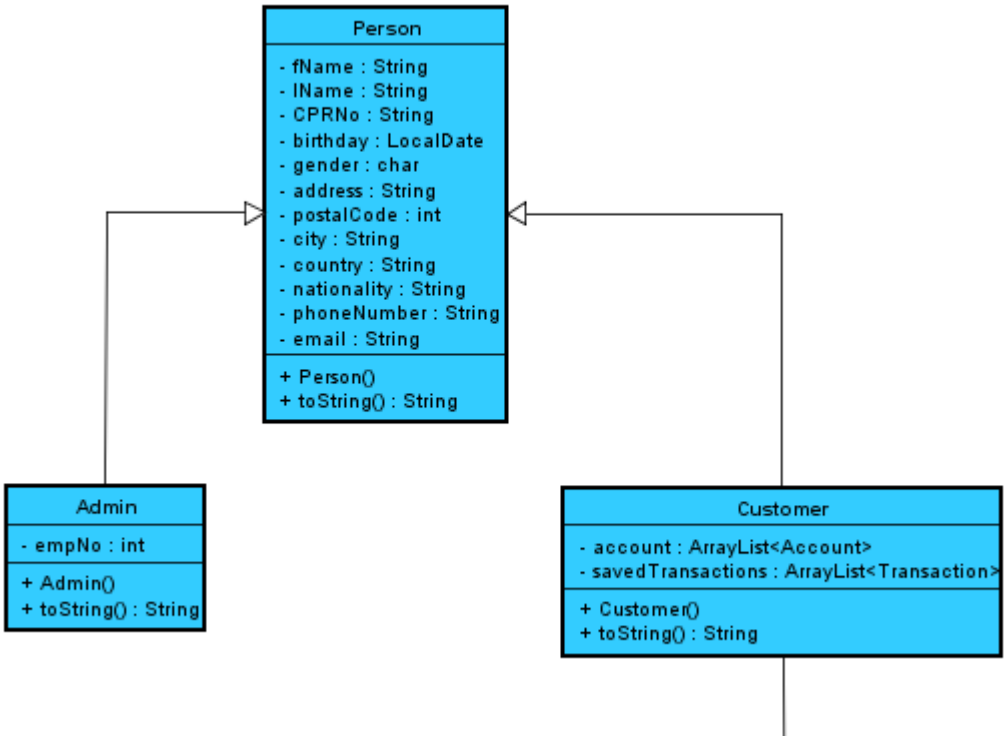


Figure 5.1.1: *Person, Admin and Customer classes*

Figure 5.1.1 shows the Person class one of the most important ones in the system.

This class is a superclass of the Customer and Admin classes. It holds information like name, birthday and CPR-No.

The Customer class has two ArrayLists inside it. One of them keeps the transaction that has been saved, so it can be used again. The other one keeps all the Accounts object inside, that the customer is assigned to. Each account object also has two ArrayList. One that keeps all the transactions made or received. The other one keeps the customer, that the account is shared with.

The IServer interface was implemented and shared with both the client and server side. Methods provided through this interface can be used by client to access certain functionalities on the server.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.ArrayList;


public interface IServer extends Remote {
    void addObjectToDatabase(Object obj) throws RemoteException;
    void deleteObjectFromDatabase(String cprNo, Object obj) throws RemoteException;
    void changeObjectInDatabase(Object obj) throws RemoteException;
```

***Figure: 5.1.2**: IServer Code Snippet*

Certain requirements had to be fulfilled by the end of its implementation to use RMI functionality. As shown in the code snippet above, IServer extends Remote (interface provided by RMI) and every method throws RemoteException.

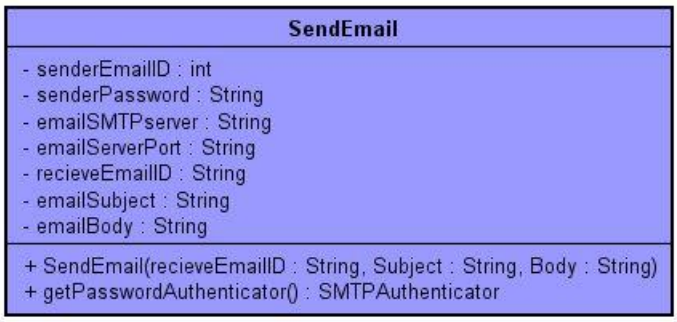


***Figure 5.1.2:** SendEmail class*

SendEmail (figure 5.1.2) class does exactly what its name suggests. Upon registration of a new person Server uses this class to notify the person by sending an e-mail with User credentials and Nem-id keys-values. It's also used to send email to the bank email account, in case a customer-user contacts the bank, using the "Contact Us" panel.

This class was setup by saving the default sender's e-mail and password. When instantiated, the constructor must receive receivers e-mail and subject and body of the e-mail message.

5.2 Client

As explained above in the design section, the Model-View-Controller design pattern was used on the client side. How each part was implement is being explained below.

Controller:

The controller consists of one class, the ClientMain. The class instances of the IServer object and the IBankModel model.

In the constructor of the class, the connection to the server is established on port 1099. So, the class can use the IServer interface to call method on the server.

The controller class is also being used by the view to get data from the model and server and commit changes.

View:

For the implementation of the graphical user interface, JavaFX was used. The designing was done using SceneBuilder, which creates a FXML file, with all components for each panel. This file can then be loaded in Java and shown in the GUI.

In SceneBuilder and in the Java code the JFoenix.jar was added. JFoenix is a material design library, which gives the developer more design option, with less programming.

The GUI has a main controller class, which has an instance of the MainClient class. From here the FXML files are loaded and has all the method to show the panels (Figure 5.2.1).

The RootLayout.fxml has a BorderPane layout, which has five options (TOP, CENTER, BOTTOM, LEFT, RIGHT) where the panel from the other FXML files can be inserted.

On the left the menu for either the customer or the admin was inserted. On the top the banner FXML file was placed. The center is set to the panel that the user chooses from the menu navigation.

```
public void showLogin(){
    try{
    // Load  login page.
    FXMLLoader loader = new FXMLLoader();
    loader.setLocation(ClientMainView.class.getResource("login/Logind.fxml"));
    this.login = (AnchorPane) loader.load();

    // Give the create login controller class access to the client main class and this class.
    Login loginController = loader.getController();
    loginController.setViewController(this);
    loginController.setClientController(clientController);

    // Set login page into the center of root layout.
    rootLayout.setCenter(login);
    rootLayout.setLeft(null);

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 5.2.1: *showLogin method*

Each fxml beside the banner.fxml has a controller class. From here each class can get the user input and show output to the user and control what happens when a button is pressed. Some of the

control classes also have an instance of the MainView class and the ClientMain so it can change which panel should be shown and, so it can make changes to the model and/or save changes onto the server.

Model:

The model package has one interface (IBankModel) and one class (BankModel) which implements the interface.

The BankModel class is responsible for keeping the data, that the application gets after a customer user logs in, changing the data, along with deleting it after logging out.

5.3 Server

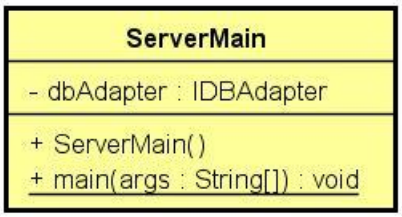


Figure 5.3.1: *ServerMain Class*

ServerMain is the class that implements the interface IServer and is responsible for running the server. One of this class's implementation requirement is "UnicastRemoteObject.exportObject(this,0)" (Fig.: 5.4) in its constructor to export itself when instantiated in the main method.

```
public ServerMain() throws RemoteException
{
    UnicastRemoteObject.exportObject(this, 0);
    dbAdapter = new DBAdapter();

}
```

Figure 5.3.2: *ServerMain - constructor code snippet*

Upon instantiation it also starts 2 Threads. Their functionality is discussed further later in this report.

The code snippet below shows the main method from Server main. Upon exe cution it creates a registry on Port 1099, instantiate the server, as the type of the interface and puts the server instance into the registry, as "bankingServer". So, then any client looking up on port 1099 should find the server and be able to connect with it.

```
36    public static void main(String[] args) {
37        // TODO Auto-generated method stub
38        try
39        {
40            LocateRegistry.createRegistry(1099);
41            IServer server = new ServerMain();
42            Naming.rebind("bankingServer", server);
43            System.out.println("Starting server....");
44        }
45        catch (Exception ex)
46        {
47            ex.printStackTrace();
48        }
49    }
```

Figure: 5.3.3: *ServerMain code snippet*

ServerMain uses DBAdapter to provide the functionality for all the methods from IServer. Since the system is to be accessible from multiple clients and there is a possibility of them calling same methods at same time, therefore all the methods are declared synchronized.

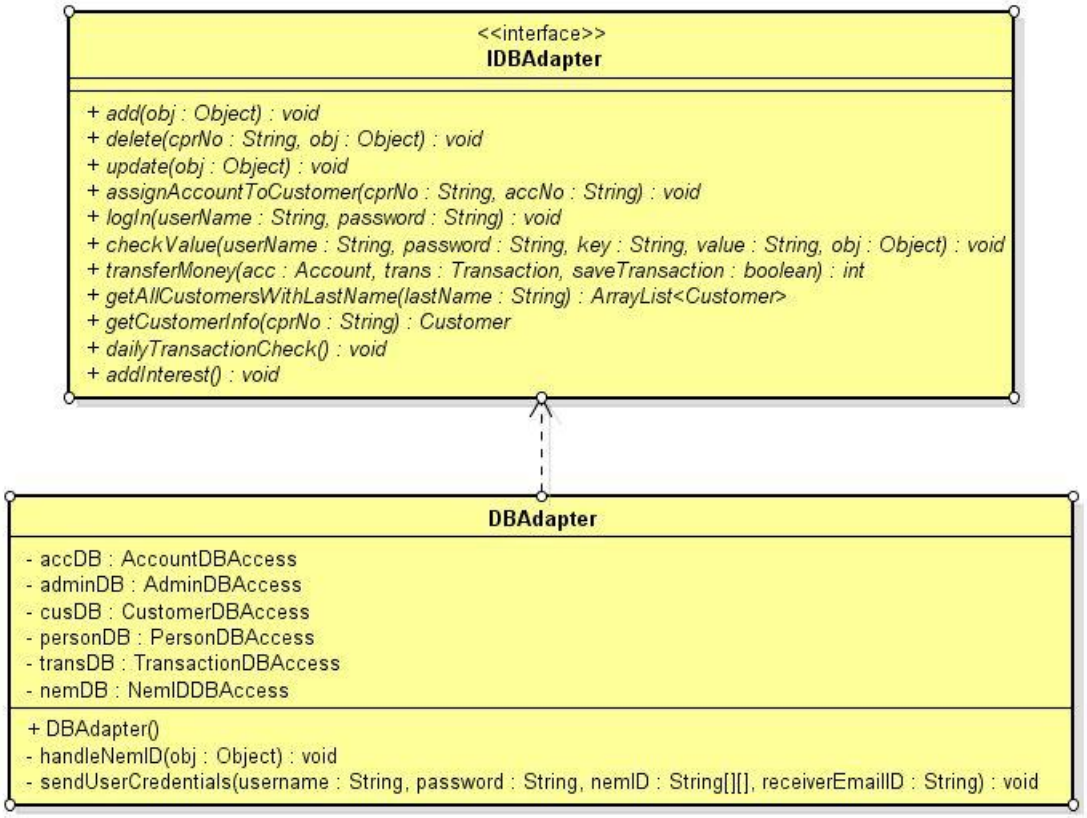


Figure 5.3.4: *IDBAdapter & DBAdapter*

DBAdapter class and IDBAdapter interface (Figure 5.3.4) are the result of the adapter design pattern. They act as an intermediary between Database access classes and the rest.

DBAdapter uses all 6 DBAccess classes and with extra helper methods it provides the desired functionality. Some of its methods are detailed below.

```
96      @Override
97      public String login(String username, String password) {
98          // TODO Auto-generated method stub
99          String cprNo = nemDB.getCPRFromUserCredentials(username, password);
100         if (cprNo.equals(null)) {
101             return null;
102         }
103
104         String key = nemDB.getKeyFromNemID(cprNo);
105         return key;
106     }
107
```

Figure 5.3.5: *DBAdapter code snippet*

The login method (Fig.: 5.3.5) is what every client has to use at beginning. It takes username and password of client as parameters and checks using nemDB if there is a CPR-No. correspondent to these parameters. If the CPR-no. does not exist, it returns null else it fetches a random key from NemID table in Database and returns it for further security check.

```
109     @Override
110     public Object checkValue(String username, String password, String key, String value, Object obj) {
111         Object object = null;
112         // TODO Auto-generated method stub
113         System.out.println("get cprNo");
114         String cprNo = nemDB.getCPRFromUserCredentials(username, password);
115
116         System.out.println("cpr " + cprNo + " checking key and value");
117         boolean check = nemDB.checkNemIDKeyValue(cprNo, key, value);
118
119         System.out.println("if statement for checked key and value" + check);
120         if (check == true)
121         {
122             if (obj instanceof Admin)
```

Figure 5.3.6: *DBAdapter code snippet*

After receiving a key, the client shall verify its value to successfully login. The checkValue method (Figure 5.3.6) serves that purpose. It receives username, password, key, value and an instance of requested object as parameters. This time, username and password are needed again to get the CPR-No.

After getting CPR-No, method checks if the key and value matches the ones in Database, if not it returns null, otherwise it checks if the provided object in parameters is an instance of Admin or of Customer. (Figure 5.3.7) Depending on that, it contacts the database again, this time using adminDB or cusDB, transDB and accDB, populates the object with relevant information and returns it.

```
        if (check == true)
        {
            if (obj instanceof Admin)
            {
                System.out.println("Admin object to return");
                // getting admin information( if no admin is found then null is returned)
                object = adminDB.getAdminInfo(cprNo, (Admin) obj);

            }
            if (obj instanceof Customer)
            {
                System.out.println("Customer object to return");
                object = cusDB.getCustomerInfo(cprNo);

                if (object == null)
                {
                    return null;
                }
                // getting arraylist of accounts
                ArrayList<Account> accounts = accDB.getCustomerAcounts(((Customer)obj).getCPRNo());

                System.out.println("Populating accounts with account transactions and adding them to customer");
                for (int i = 0; i < accounts.size(); i ++)
                {
                    accounts.get(i).setTransactions(transDB.getAccountTransactions(accounts.get(i)));
                    // adding every customer info to account, who shares it
                    cusDB.setSharedAccountCustomers(((Customer)obj).getCPRNo(), accounts.get(i));
                    System.out.println(accounts.get(i).toString());
                    // adding account to customer
                    ((Customer) obj).addAccount(accounts.get(i));
                    System.out.println("Account added to customer");
                }
            transDB.setSavedTransactionsForCustomer(((Customer)obj)); // adding savedTransaction to customer
            } else { System.out.println("object not recognized");
                return null; }
        } else {
            System.out.println("Value did not match");
            return null; }
        System.out.println(object.toString());
        return object;
    }
```

Figure 5.3.7: *checkValue Method*

However, if the requested object instance is wrong (e.g. Customer trying to log-in as an admin, or vice versa), the method returns null.

All DBAccess classes are the result of DAO pattern. Every class uses JDBC (Java Database Connection) drivers to connect with Database and run SQL queries to read and write information. Each class has its own unique connection which makes them more flexible, e.g. if two clients trying to call methods from two different DBAccess classes, then they won't interfere with each other.

Since all the DBAccess classes have same structure and design, therefore only one of them is detailed. Below are details of the connection from TransactionDBAccess class.

```
19      private Connection connection;
20      private PreparedStatement pStatement;
21      private String sql, driver, url, user, pw;
22      private ResultSet resultSet;
23
24      public TransactionDBAccess()
25      {
26          /*
27           *  Setting up a connection to the Database
28           *  1. Loading JDBC driver
29           *  2. URL for database
30           *  3. Username to database
31           *  4. password to database
32           */
33
34          driver = "org.postgresql.Driver";
35          try {
36              Class.forName(driver);
37          } catch (ClassNotFoundException e) {
38              // TODO Auto-generated catch block
39              e.printStackTrace();
40          }
41          url = "jdbc:postgresql://localhost:5432/postgres";
42          user = "postgres";
43          //pw = "8803096680";
44           pw = "postgres"; //
45          //pw = "521765";
46
47          sql = "";
48          pStatement = null;
49          resultSet = null;
50
51      }
52
```

***Figure 5.3.8**: TransactionDBAccess constructor*

The constructor of TransactionDBAccess (Figure 5.3.8) is where a part of the connection is setup. Using Class.*forName*(driver) it looks up the JDBC driver and loads it. Three variables hold important connection information, url (the address of postgresql Database), user (the username for the connection) and pw (the password).

With that, the only remaining thing for the connection is to use DriverManager (Figure 5.3.9) to get it, using all saved information.

```
52
53      private void openConnection() throws SQLException
54      {
55              connection = DriverManager.getConnection(url, user, pw);
56      }
57
```

***Figure 5.3.9:** openConnetion method*

Since it is a common practice among IT developers to open the connection and statement when needed and close them afterwards when finished using them, the methods openConnection() and closeStatementConnection() (Figure 5.3.10) were implemented as private methods.

```
58      private void closeStatementConnection()
59      {
60          try {
61              pStatement.close();
62              connection.close();
63          } catch (SQLException e) {
64              // TODO Auto-generated catch block
65              e.printStackTrace();
66          }
67      }
68
69
```

Figure 5.3.10: *closeStatementConnection method*

For further clarification of DBAccess classes' structure, one of the methods from insertTransfer (Figure 5.3.11) is detailed below.

```
71      public synchronized void insertTransfer(String accNo1, String regNo1, Transaction trans) {
72          // TODO Auto-generated method stub
73          sql = "INSERT INTO onlinebankingsystem.transaction "
74                  + "(accNo1, regNo1, accNo2, regNo2, description, date, money_transfered)"
75                  + " VALUES (?, ?, ?, ?, ?, ?, ?)";
76                  try {
77                      openConnection();
78                      pStatement = connection.prepareStatement(sql);
79                      pStatement.setString(1, accNo1);
80                      pStatement.setString(2, regNo1);
81                      pStatement.setString(3, trans.getTransactionAccNo());
82                      pStatement.setString(4, trans.getRegNo());
83                      pStatement.setString(5, trans.getDescription());
84                      pStatement.setDate(6, Date.valueOf(trans.getTransferDate()));
85                      pStatement.setDouble(7, trans.getAmount());
86                      pStatement.executeUpdate();
87                      System.out.println("Transaction added to Database.");
88                  } catch (SQLException e) {
89                      // TODO Auto-generated catch block
90                      e.printStackTrace();
91                  }
92                  finally {
93                      closeStatementConnection();
94                  }
95      }
```

Figure 5.3.11*: insertTransfer method*

insertTransfer is a very basic method, serves for populating transaction relation in the system's database. It receives as parameters, account no. and registration no. of the person transferring money and trans, an instance of Transaction object which includes further transaction information.

A SQL query is written to the sql variable. The connection to Database is opened and using this connection an instance of prepared statement is created. Later the preparedStatement prepares the sql query by filling the missing details and executes it. Finally, it closes both statement and connection. This method was declared as synchronized, since it is used by ToBeTransThread to on a daily routine check and could also be used by a client trying to communicate to database.

That covers the explanation for most of the methods in DBAccess classes since their structure is almost the same.

Next is elaboration on one of the interesting SQL queries (5.3.12) used in AccountDBAccess in method called getCustomerAccounts, which should return an ArrayList of account for certain customer.

```
sql="SELECT customer_account.cpr_no, account.*" +
        " FROM onlinebankingsystem.account" +
        " INNER JOIN onlinebankingsystem.customer_account" +
        " ON customer_account.accno=account.accno AND "
        + "customer_account.regno=account.regno AND customer_account.cpr_no = ?";
```

Figure 5.3.12: *SQL query*

Implementing this query was necessary for getting all the accounts of a customer and also their information from Database, which was spread on two different tables (customer_account and account).

Upon execution of this query, a resultSet is return with information. Every row contains an account information with customers CPR-No. It is a result of SELECT query with INNER JOIN of relations customer_account and account on attributes accno (account number), regno (registration number) and cpr_no.

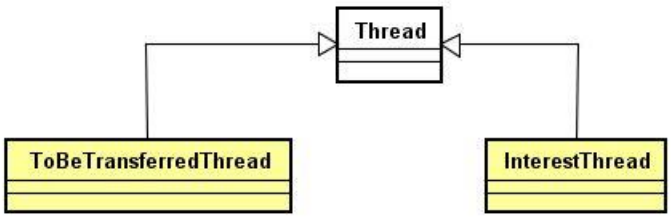


Figure 5.3.13: *Threads classes*

These threads shown above serve a key role in the system. Both Thread are started when ServerMain is instantiated.

ToBeTransThread does a daily check on tobe_transferred relation in database. If there are any transactions pending for that date, it will transfer them to the transaction relation.

InterestThread gives interest on money in every account. It does that every week.

Both of these threads use IDBAdapter's instance from ServerMain to communicate with database.

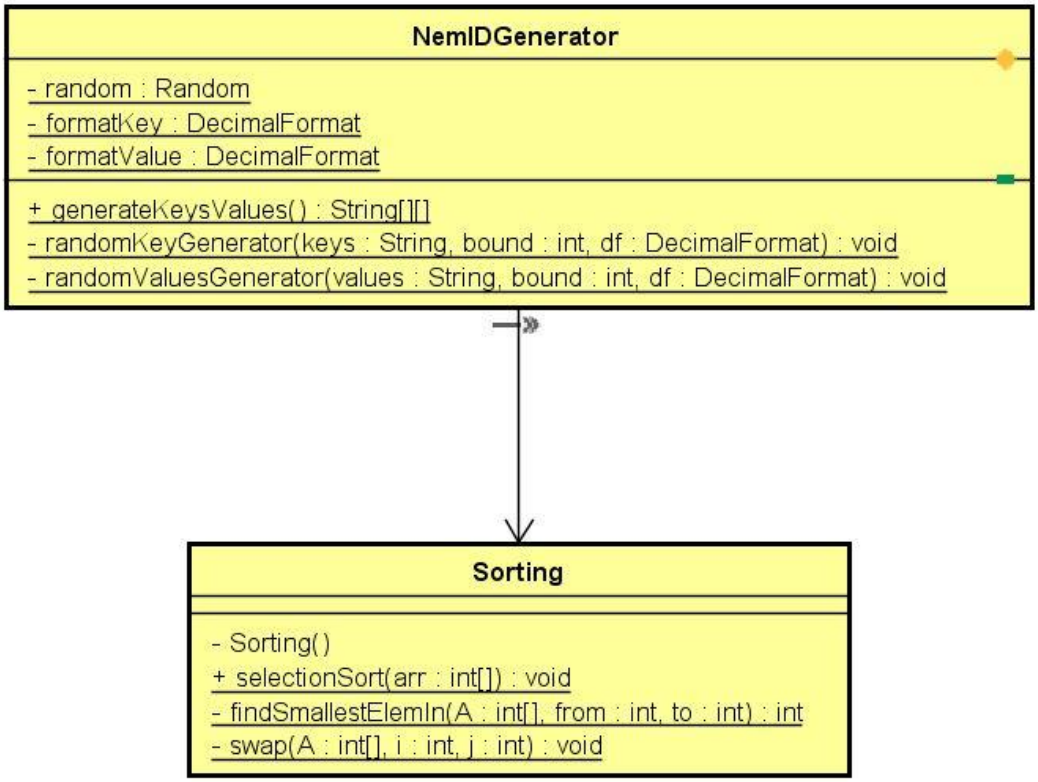


Figure 5.3.14: *NEM-ID and Sorting class*

NemID class (Figure 5.3.14) shown above is mainly used to generate a two-dimensional Array of Strings of random NemID keys and values.

Sorting class is used by NemID class during the process of generating random keys. It provides a method called SelectionSort(array) to sort an array into ascending order.

6 Test

The program is tested in accordance with the requirement mentioned in the begging.

Each requirement was tested using scenario-testing. For each requirement, between one and five different scenario-tests were done.

Below is an overview of all 22 requirements and the result of the scenario testing. All requirements passed the scenario testing beside the user story "A Customer-User can print his history (pdf)" which was not implemented, due to lack of time. A more detailed overview of the scenario testing can be found in the appendix G – Testing, were every step of each scenario test is being described along with all the scenario tests.

No.	Requirement	Result
1	A User can use the system from multiple computers	PASSED
2	An Admin-User can create a new Customer-User	PASSED
3	An Admin-User can create an account	PASSED
4	An Admin-User can add account to Customer-User	PASSED
5	A User can log in	PASSED
6	A User can log out	PASSED
7	A Customer-User can transfer money to a different account	PASSED
8	A Customer-User can receive money from a different user	PASSED
9	A Customer-User can see balance on each account	PASSED
10	A Customer-User can see transfer history on every account	PASSED
11	A Customer-User gets interest on his money	PASSED
12	A Customer-User can share an account with another user	PASSED
13	A Customer-User can transfer money on a chosen date	PASSED
14	A Customer-User can change personal information	PASSED
15	A Customer-User can save transaction information	PASSED
16	A Customer-User can see his account information	PASSED
17	A Customer-User can contact the bank with email	PASSED
18	A Customer-User can print his history (pdf)	NOT IMPLEMENTED
19	An Admin-User can see the transfer history of an account	PASSED

20	An Admin-User can see the balance of a customer-user	PASSED
21	An Admin-User can change a User’s information	PASSED
22	An Admin-User can create new admin-user	PASSED

7 Result & Discussion

The outcome of this project was rather satisfactory then perfect. Some functionalities were left off unimplemented due to lack of time. However, all the important requirements were implemented successfully according to their user stories.

Database functionalities like triggers and functions were not very useful in this system, although some other functionalities could have been a nice addition to the System. Instead of implementing ToBeTransferThread and InterestThread threads two schedule procedures written in SQL in Database would have done the job perfectly. Since it was not a part of SEP2 project, it was not included, and the functionality was handed to the two threads.

Security of the system was a success. The system was implemented with double security check upon log-in. Client is asked for username and password, if they are correct then it is asked to confirm a value for a certain key, from the Nem-ID. This double security check made the system secure, however there are still room for improvement.

8 Conclusion

The requirements have been analysed, designed, implemented and tested successfully, besides one. The requirements were analysed and checked thoroughly by creating Use Cases, Use Cases description and class diagrams.

Different design patterns were implemented with client/server system as a core of the system and a data model was provided to save user related information. A graphic user interface was designed and implemented for easy user interaction with the system. The final product was tested successfully, using scenario testing.

To conclude, the system has fulfilled all the important requirements. The developed system prevents the need of a web browser to do online banking tasks, since it has all the important functionalities e.g. client can transfer money, client can see transaction history, admin can register person and create account and more.

9 Project Future

This section explains what would be implemented or improved in case, more time would have been available.

1. As a result of the short time one of the features from customer requirements ("The customer can print his/her account history as pdf file") was not implemented, this would be implemented as one of the first things.
2. An improvement of the system could be reducing the redundancy in saved_Transaction table in database.
3. Another improvement to the system could be adding a transaction isolation level which will help avoiding over-writing or losing data from database and handling "Dirty Read Problem", "Nonrepeatable read" and "Phantom read".
4. Another improvement could be, that instead of using threads in Java for daily checking of transactions in Database and adding interest to account, could be used stored procedures from SQL.
5. Also for security purposes the user credentials could be saved as hash key using technique called "SALT".
6. Another feature which could be "A Customer-User can apply for loan".
7. A live chat connection between customer and admin, to provide a better support for the customer.
8. Another feature could be "A Customer can apply online for new credit/debit card".

10 References

Lewis, J. and Chase, J. D. (2010) *Java Software Structures: Designing & Using Data Structures*.

Kurose, J. F. and Ross, K. W. (no date) *COMPUTER NETWORKING A Top-Down Approach*.

Thomas M. Connoly, C. E. B. (no date) '[Thomas_M._Connolly,_Carolyn_E._Begg]_Database_Sys(BookZZ.org).pdf'.

SDJ_Home_page_Session Material - Default (no date). Available at: https://studienet.via.dk/Class/IT-SDJ2X-A17/Session Material/Forms/Default.aspx (Accessed: 14 December 2017).

JavaFX 8 Tutorial - Part 1: Scene Builder | code.makery.ch (no date). Available at: http://code.makery.ch/library/javafx-8-tutorial/part1/ (Accessed: 14 December 2017).

11 List of Appendixes

- Appendix A: User Guide
- Appendix B: Use Case Diagram
- Appendix C: Use Case Descriptions
- Appendix D: Class Diagrams
- Appendix E: Sequence Diagram
- Appendix F: Project Description
- Appendix G: Testing
- Appendix H: EER Diagram