Student: Czako Zoltan

**Group: 30235**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 6

4. UML Sequence Diagrams 9

5. Class Design 11

6. Data Model 13

7. System Testing 14

8. Bibliography 15

1. Requirements Analysis

# Assignment Specification

In this assignment we implemented a desktop application. The subject of this application is a bank app, in which helps the employees in their daily work, helps them to maintain their relationship with the clients, to manipulate the client’s data, to make transfers between accounts and so on. Besides helping the employees, it also helps the administrator of the bank, who can hire new employees, he can modify employee’s data and he can get reports about the actions of the workers.

# Functional Requirements

Our application will have two types of users. The first type is the Employee. An employee has access to the data of the clients. He can insert new clients, he can modify the personal information of an existing client and if he must, he can delete a client. An employee can see all the data of a client, he can view the account of a client, their balance, their IBAN code, their address, phone number, mail, first and last name and the date of registration. If a client requests a money transfer, the employee is able to make transfers from one account to another if the client has necessary money on he’s account and after that the employee can generate an utility bill of the last transaction.

The second type of user is the Administrator. This type of user has the ability to view the list of the bank’s employees, he can select an employee and view his personal information like username, first name, last name, mobile and mail. He can modify the user data of an employee and he can also delete an employee if needed. The admin can generate a report of an employee actions with a well specified start date.

# Non-functional Requirements

In order to obtain all this functionality we created a Windows Forms Application. Talking about accessibility, to differentiate the two type of user we created a login form. After login there are two different views, one for the admin and one for the client. We used list boxes to show all the users and all the clients in an ordered way. For creating and updating users/clients data we created another two views with text fields in which we can introduce or modify data.

To obtain availability we created a GitHub repository and after each modification that we made or new features that we created, we committed out changes to this repository and in this way we prevented data loss, wrong modifications or accidental code deletion. This repository helps us in backup, because we can backup our system striating with a well specified date(last time when our system was stabile).

Talking about dependency, we have a Layered Architecture, so we have three separate project one for the Presentation Layer, one for the Business Layer and one for the Data Access Layer. Besides these functional layers we have another project for testing our system. These Layers are loosely coupled between them, each layer can work separately, so they can be reused in another project.

We work really much on failure management and we tried to cover all the situations when a user can enter wrong data, or tries to make some illegal operation.

Talking about portability, our application is platform dependent, it work only on Windows operation systems because it is a Windows native application.

In the view of usability, this software is intended to be used in banks, it can help in client data management and for supervise the actions of employees.

To grant a well working and bugs free application, to keep business needs well satisfied we created Unit Tests for each business layer in which we have data validations or conditional statements. We tried to have a 100% of covering, we test corner cases as well as happy flow situations.

2. Use-Case Model

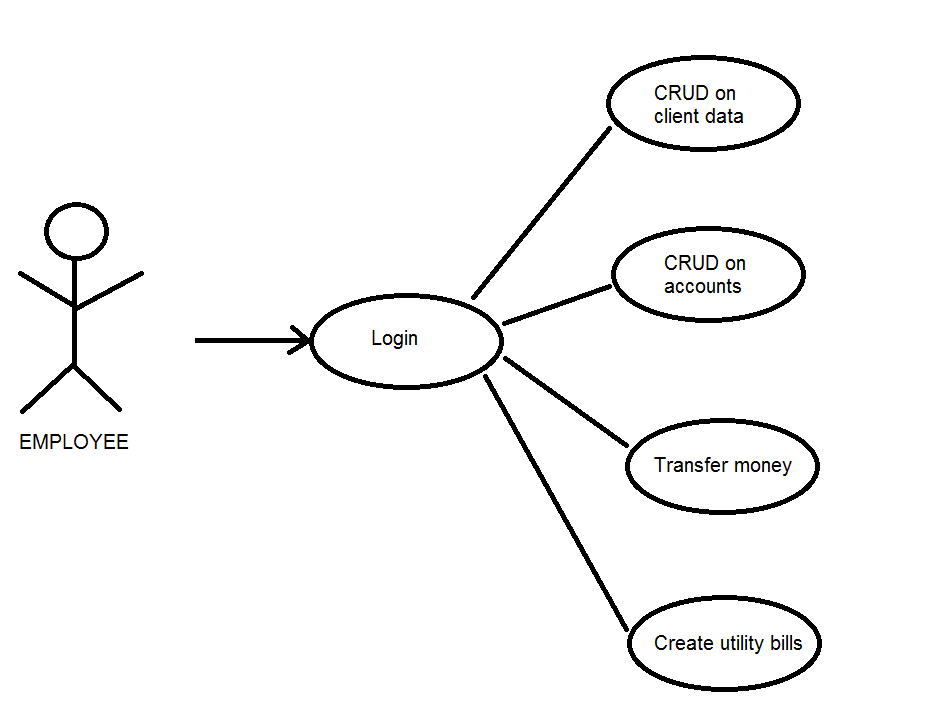
Use Case: Describe the possible actions of the employees

Level: user-goal level

Primary actor: Employee

Main success scenario:

1. Login
2. CRUD on Clients
3. CRUD on accounts
4. Transfer money between accounts
5. Generate utility bills for transfers



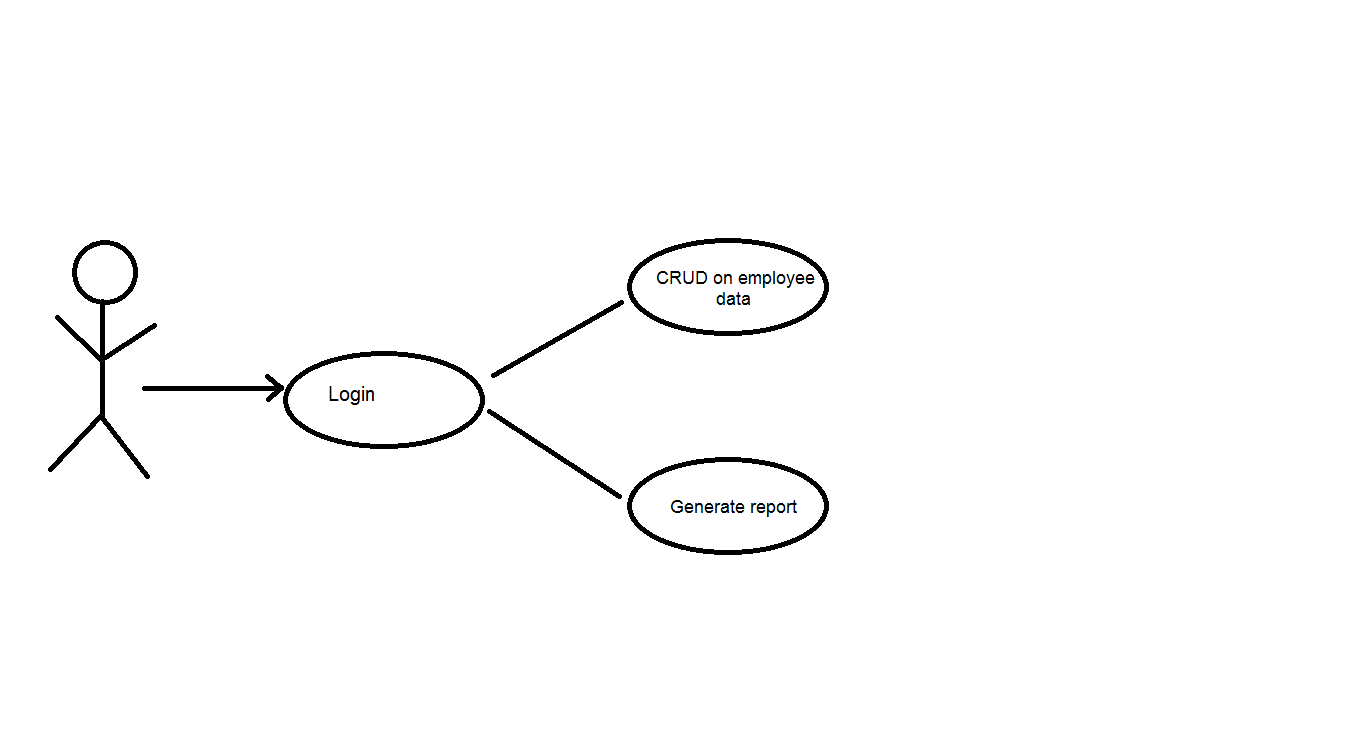
Use Case: Describe the possible actions of the aministrator

Level: user-goal level

Primary actor: Administrator

Main success scenario:

1. Login
2. CRUD on Employees
3. Report on employee’s actions starting from a specified date



Administrator

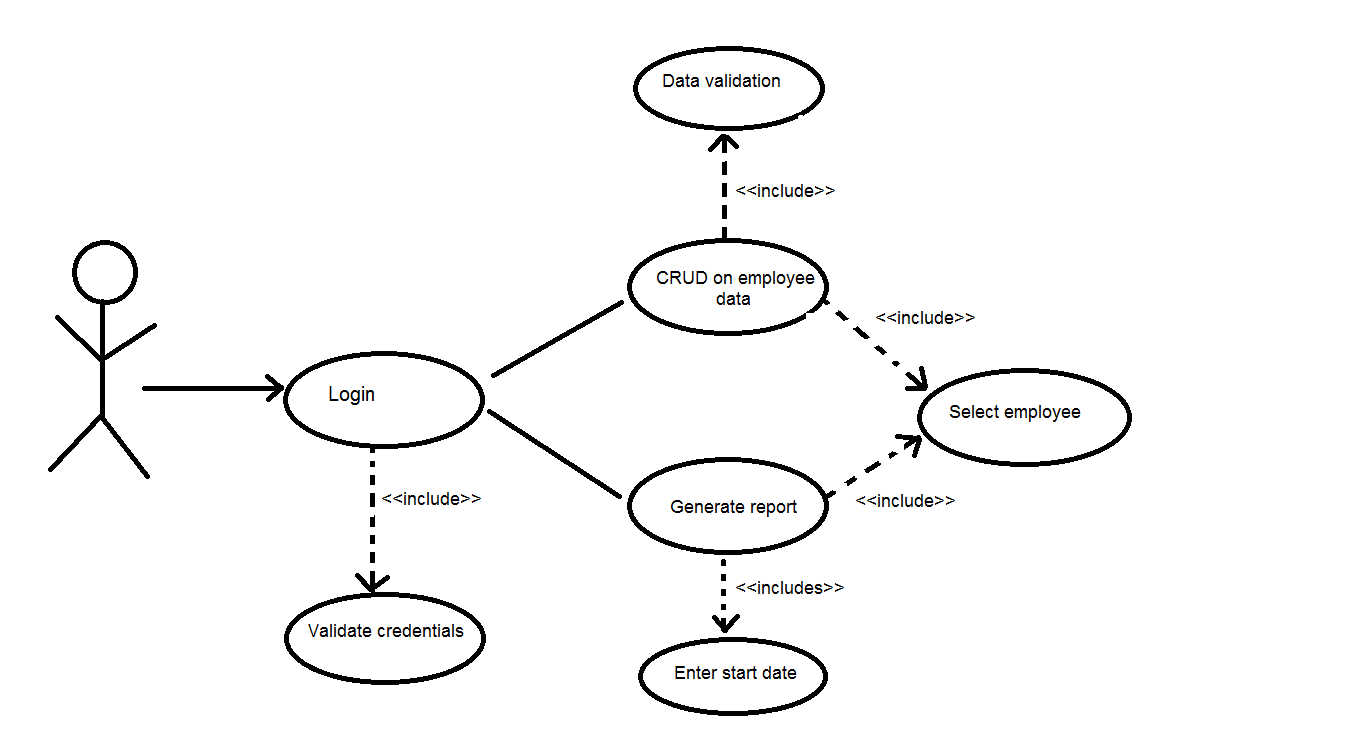
Use Case: Describe the possible actions of the aministrator

Level: sub-function level

Primary actor: Administrator

Main success scenario:

1. Login 🡪 Validate credentials
2. CRUD on Employees 🡪 Select employee 🡪 Validate data
3. Report on employee’s actions starting from a specified date 🡪 Select employee 🡪 Select start date



Administrator

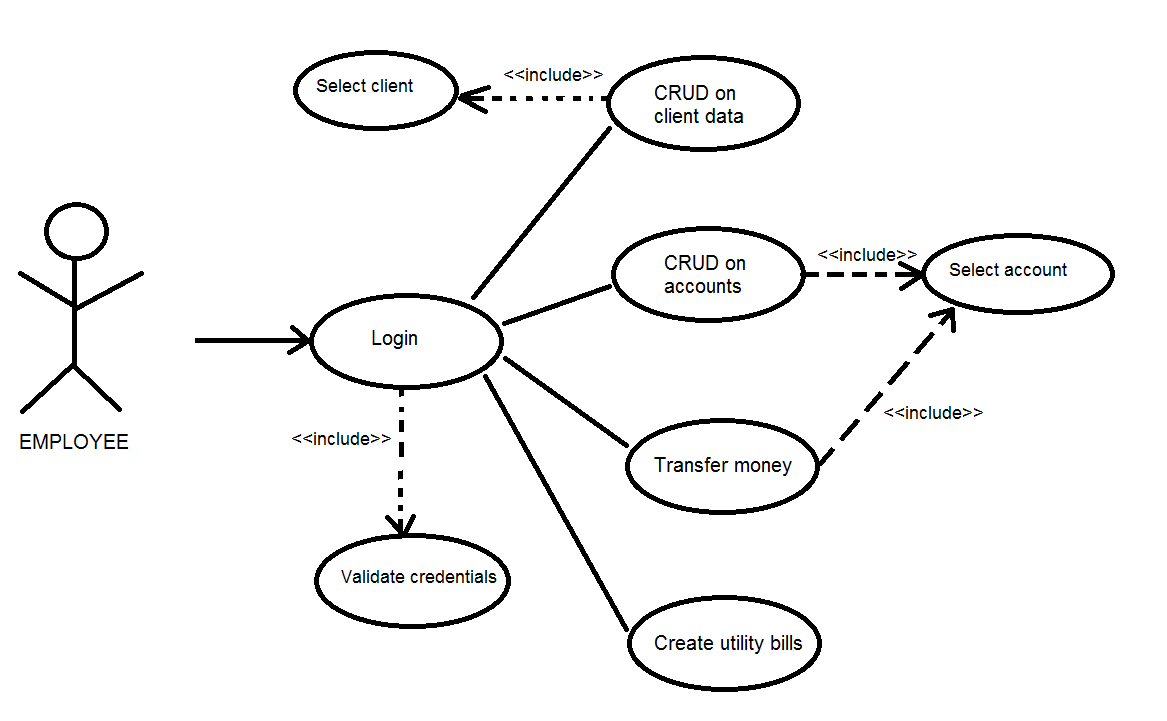
Use Case: Describe the possible actions of the employees

Level: sub-function level

Primary actor: Employee

Main success scenario:

1. Login 🡪 Validate credentials
2. CRUD on Clients 🡪 Select client
3. CRUD on accounts 🡪 Select account
4. Transfer money between accounts 🡪 Select accounts
5. Generate utility bills for transfers



3. System Architectural Design

**3.1 Architectural Pattern Description**

Layered architecture focuses on the grouping of related functionality within an application into distinct layers that are stacked vertically on top of each other. Functionality within each layer is related by a common role or responsibility. Communication between layers is explicit and loosely coupled. Layering your application appropriately helps to support a strong separation of concerns that, in turn, supports flexibility and maintainability.

The main benefits of the 3-tier architectural style are:

* **Maintainability**. Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.
* **Scalability**. Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.
* **Flexibility**. Because each tier can be managed or scaled independently, flexibility is increased.
* **Availability**. Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

In our project we created 3 tier, 3 separate layer. The first layer is the Presentation Layer, which we named as BankPL which stands for Bank Presentation Layer. In this layer we used Windows Forms Application because it has a lot of different components, so we can make a very good design for the UI, and because it is a desktop application, the security is even better (no internet connection, less viruses or hackers).

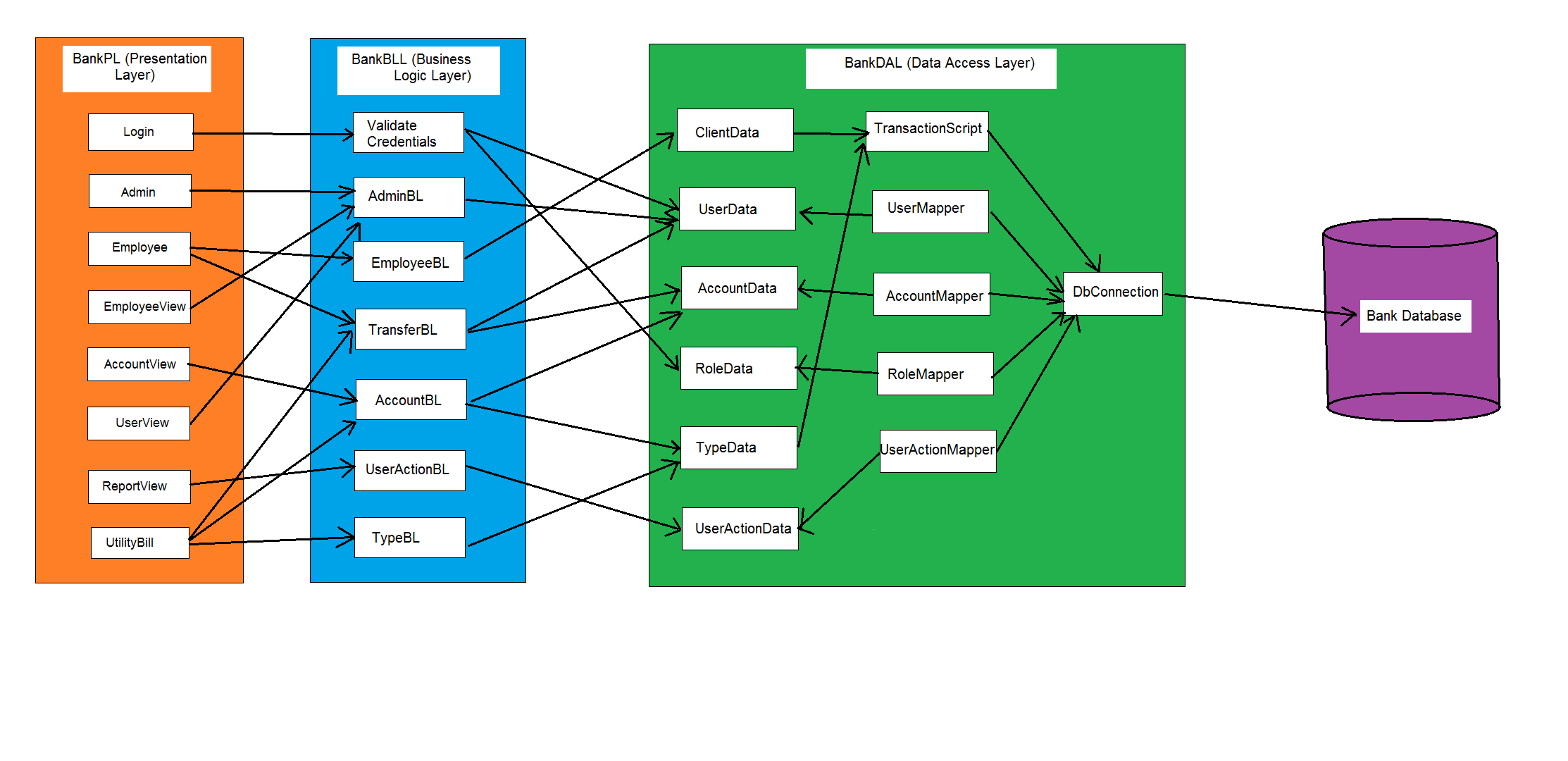
The second layer is the Business Layer, in our project it’s name is Bank BLL (Bank Business Logic Layer). In this layer we make all the validation that the business needs, here we have data formatters, data format checkers, e-mail, password and account validations, error custom error handling and all that the business demands.

The third layer is the data Access Layer, which name is BankDAL (Bank Data Access Layer). The goal of this layer is to separate data access and data manipulation from other layers. In this layer we implemented two design pattern for a clear way of accessing our data. The first pattern is the Transaction Scrip pattern, which contains all the methods with SQL Scripts to manipulate data of the clients. The second Design Pattern is the Data Mapper Pattern. For each data object we have a data mapper object. The goal of these mappers is to hide operations used to access and manipulate data from database, the mapper creates an object with the information needed for the next layer. Other layers do not needs to know anything about SLQ Scripts or other server side configurations.

**3.2 Diagrams**

*[Create the system’s conceptual architecture; use architectural patterns and describe how they are applied. Create package, component and deployment diagrams]*

The architecture of our project is a 3-tier Layered Architectural pattern. As we can see in the following image, we have three layers and the database. Every layer has a connection to the immediate inferior layer. In the following image we can see these connections.



For each layer we created a separate project, separate namespace, so every layer can be reutilized in other projects. Besides of the three layers we have a project for testing our system. Here is a package diagram of our system:

BankPL

(Presentation Layer)

<<references>>

<<references>>

BankTest

Bank

Database

BankDAL

(Data Access Layer)

BankBLL

(Business Logic Layer)

<<references>>

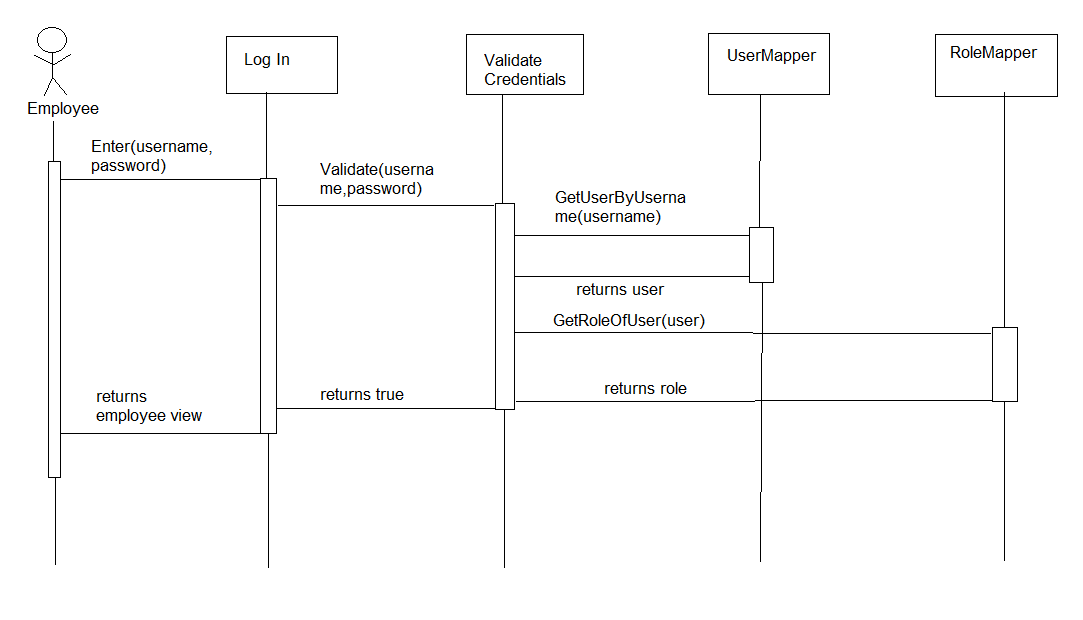
<<references>>

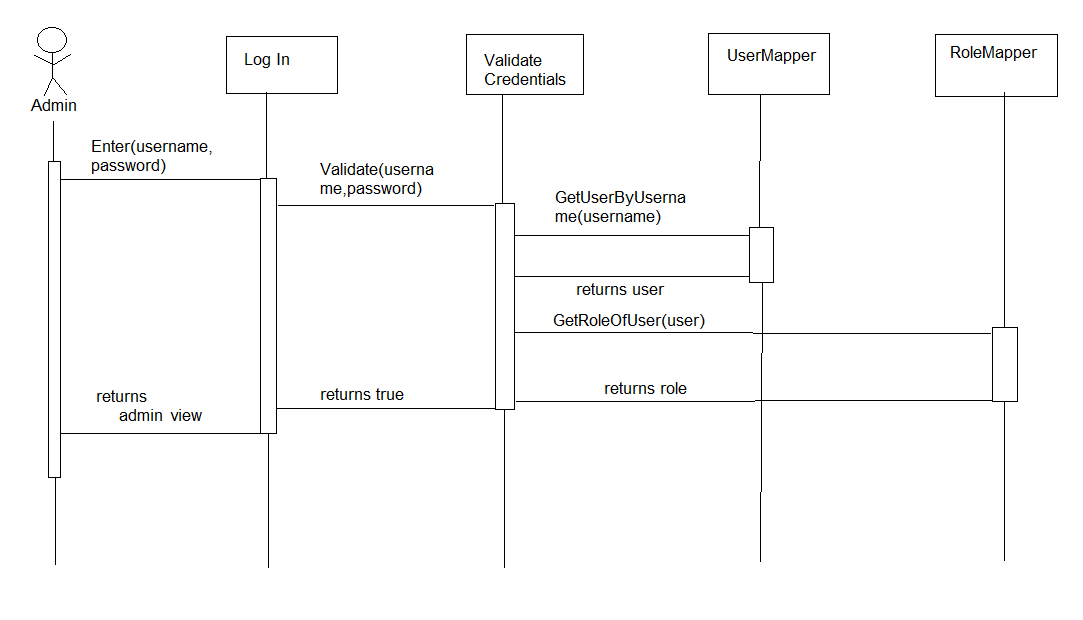
<<references>>

<<references>>

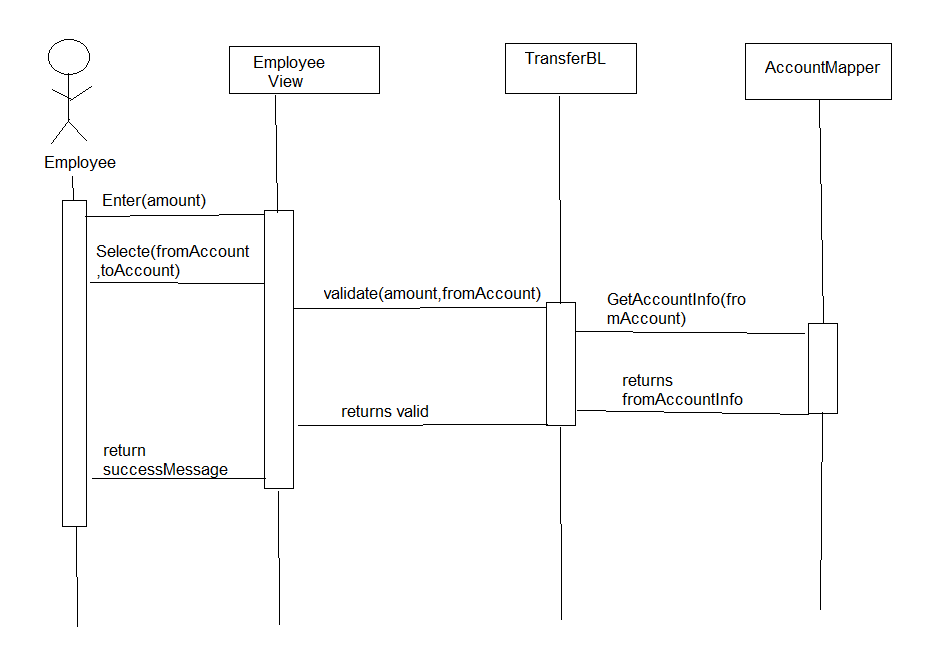
4. UML Sequence Diagrams

In the next image we will present a sequence diagram in which we describe the flow of login:





The next sequence diagram shows the way an employee can make money transfers between accounts.



5. Class Design

**5.1 Design Patterns Description**

The first design pattern that we used in our implementation is the **Data Mapper Pattern**. The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With Data Mapper the in-memory objects needn't know even that there's a database present; they need no SQL interface code, and certainly no knowledge of the database schema. (The database schema is always ignorant of the objects that use it.) Since it's a form of Mapper, Data Mapper itself is even unknown to the domain layer. Here is a schema of this pattern

<<Entity>>Mapper

<<Entity>>

Database

-propr1

-propr2

….

-proprN

+insert

+update

+delete

+view

+find

….

+method1

+method2

….

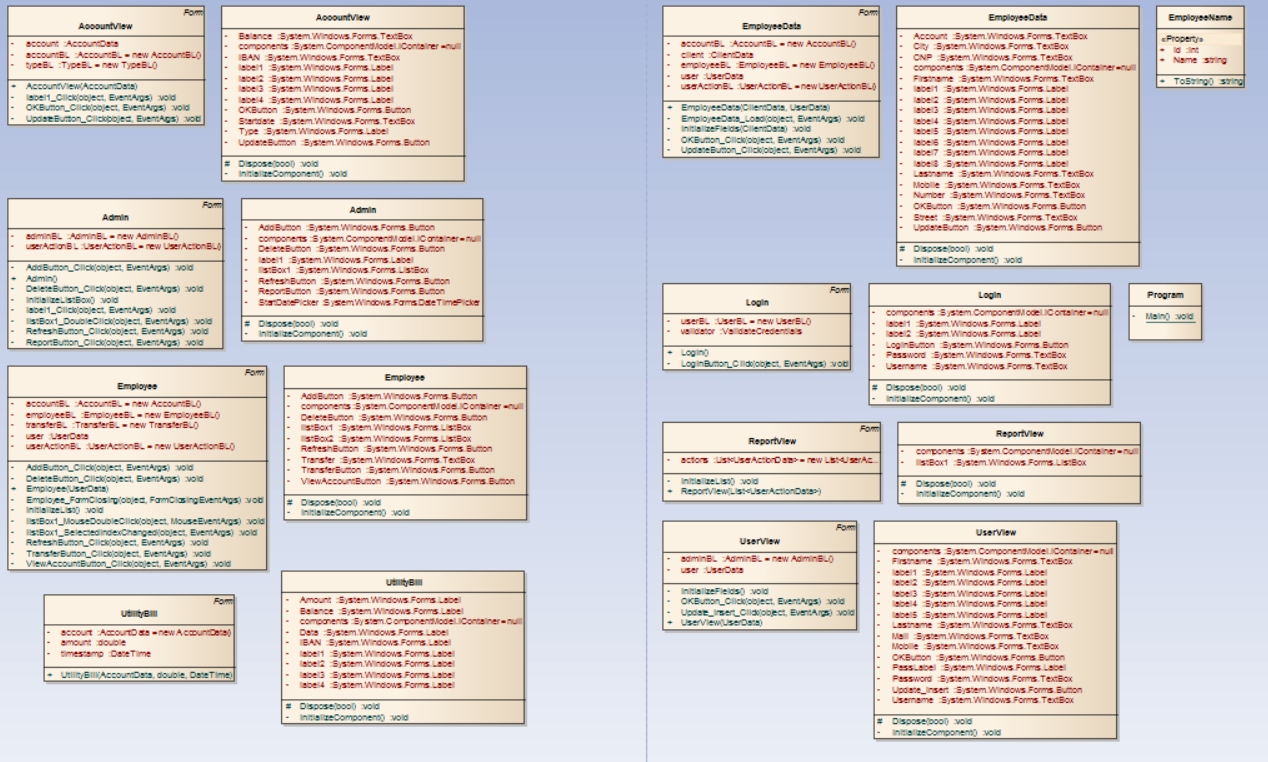
+methodN

Where <<Entity>> is a user defined class.

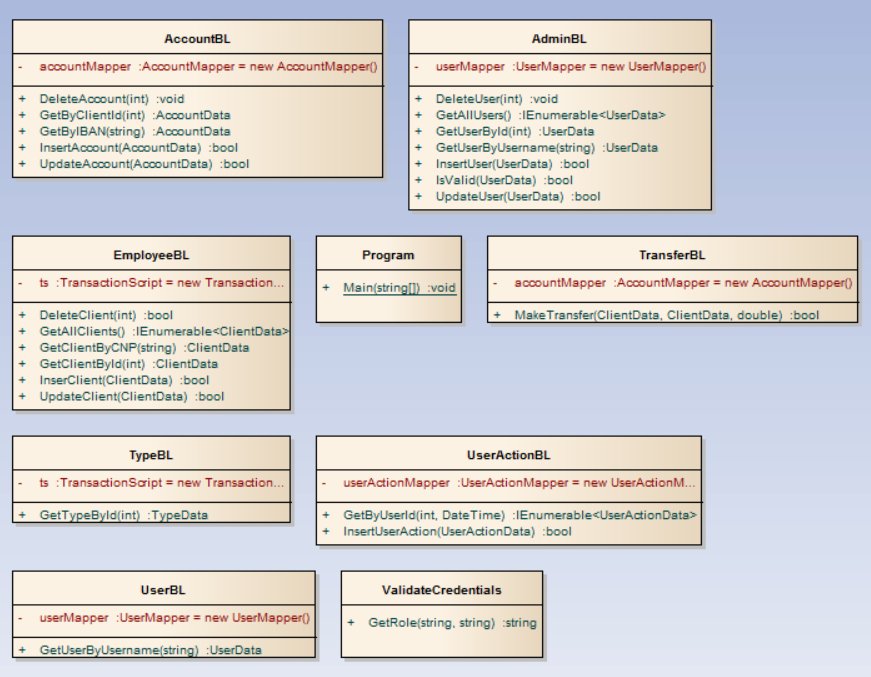
The second design pattern that we used is the Transaction Script. A Transaction Script organizes all this logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each transaction will have its own Transaction Script, although common subtasks can be broken into subprocedures.

**5.2 UML Class Diagram**

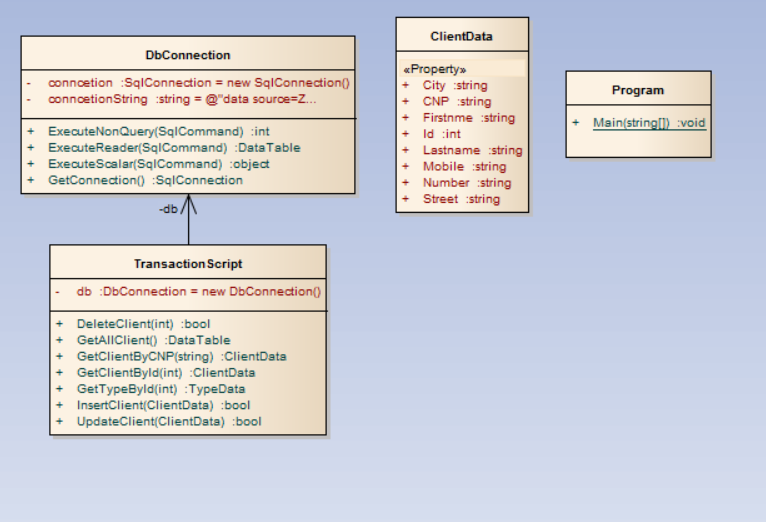
**Class diagram for BankPL (Presentation Layer)**

****

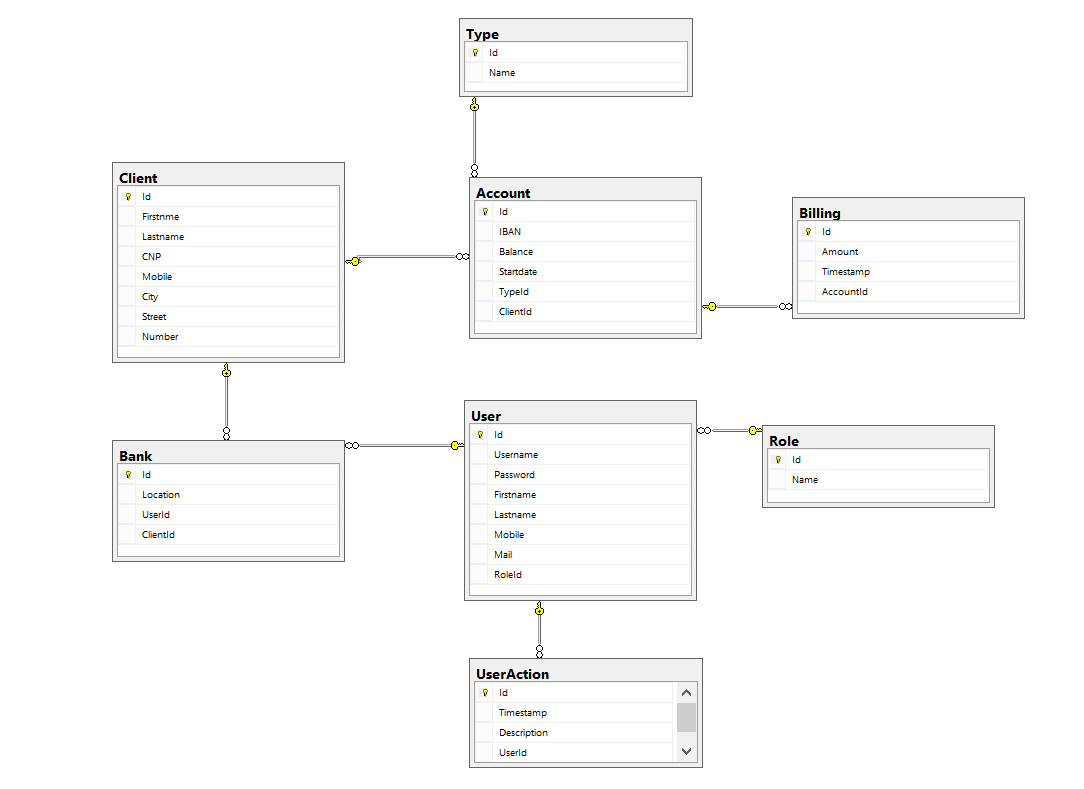
**Class diagram for BankBLL (Business Logic Layer)**



**Class diagram for BankDAL (Data Access Layer)**

****

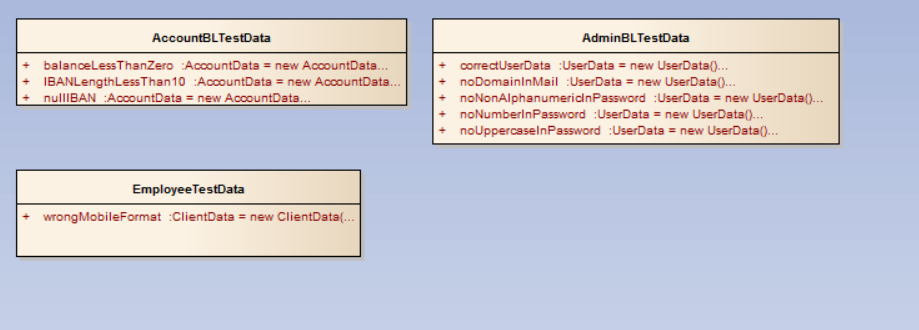
6. Data Model

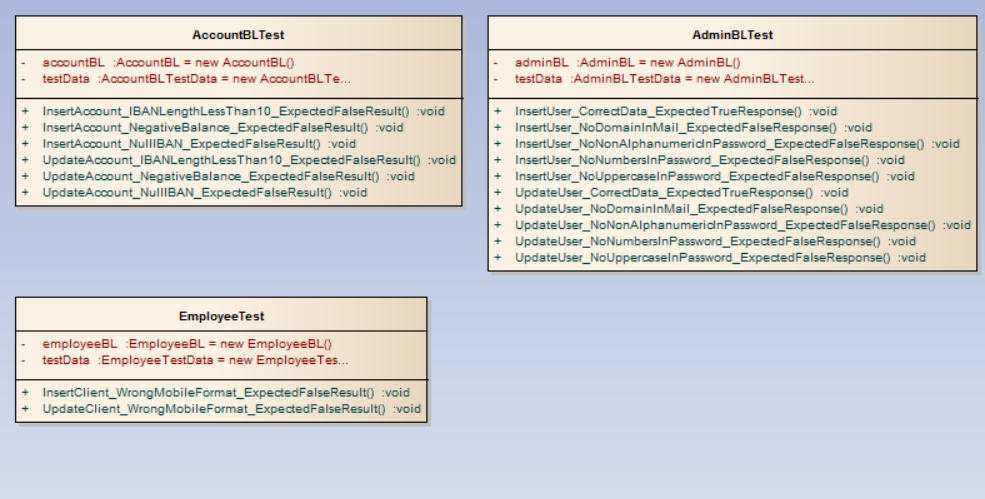


7. System Testing

We wanted to create a reliable and functionally correct project so we created test for each functionality, for each data validation and conditional statements. We used the Unit test framework and we tested the Business layer, because in this part of the project we had a lot of data validation as e-mail format, password format, null values, balance less than 0. We tested the corner cases as well as the happy flow situations.

The next pictures shows our test units, classes and methods that we used to cover as many situations as we could.





8. Bibliography

[1] <http://martinfowler.com/eaaCatalog/transactionScript.html>

[2] <http://gunnarpeipman.com/2013/10/transaction-script-pattern/>

[3] <http://msdn.microsoft.com/en-us/library/54xbah2z(VS.80).aspx/>

[4] <http://msdn.microsoft.com/en-us/library/e80y5yhx(VS.80).aspx/>