Assignment1 : Application for the front desk employees of a bank

Analysis and Design Document

Student: Lazăr Raluca-Florina

**Group: 30233**

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 3

3. System Architectural Design 3

4. UML Sequence Diagrams 3

5. Class Design 3

6. Data Model 3

7. System Testing 3

8. Bibliography 3

1. Requirements Analysis

# Assignment Specification

Use JAVA/C# API to design and implement an application for the front desk employees of a bank.

The application should have two types of users (a regular user represented by the front desk employee and an administrator user) which have to provide a username and a password in order to use the application.

# Functional Requirements

The regular user should be able to perform the following operations:

- Add/update/view client information (name, identity card number, personal numerical code, address, etc.).

- Create/update/delete/view client account (account information: identification number, type, amount of money, date of creation).

- Transfer money between accounts.

- Process utilities bills.

The administrator user should be able to perform the following operations:

- CRUD on employees’ information.

- Generate reports for a particular period containing the activities performed by an employee.

# Non-functional Requirements

External non-functional requirements:

- Usability – application is easy to use and also easy to learn by the user.

- Safety – not causing harm, injury or damage.

- Security – accessible and usable only by authorized users.

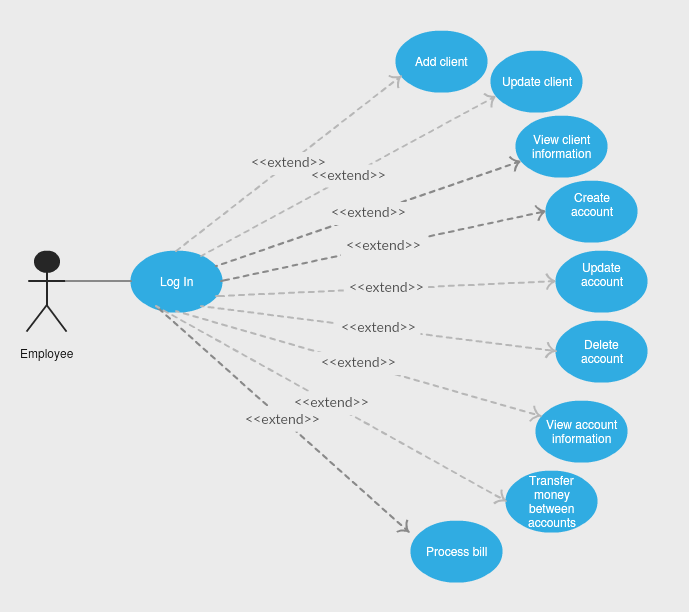
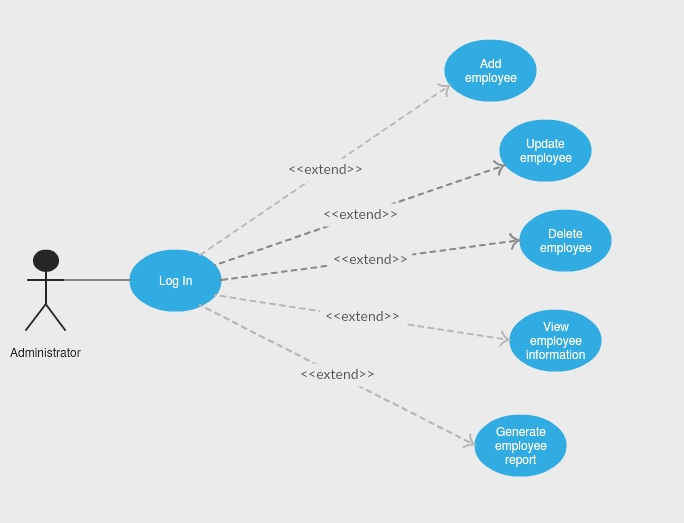
- Privacy – protecting personal information and undesired access to personal space.

Internal non-functional requirements:

- Database – structured data and integrity of stored data.

– all the inputs of the application validated against invalid data before submitting the data and saving it in the database.

2. Use-Case Model



*Use case: Add employee*

*Level: User-goal level*

*Primary actor: Administrator*

*Main success scenario: 1. Administrator chooses his field in the main window of the application interface and presses OK.*

*2. Log in page for administrators pop up.*

*3. Administrator fill in the blanks for his personal username and password and presses OK.*

*4. System displays a list with the actions an administrator could perform.*

*5. Administrator chooses the ‘Add employee’ action.*

*6. A pane with 3 empty fields pop up.*

*7. Administrator fill in the information (name, username and password) for the employee he wants to register.*

*8. System registers the employee and store the information in the database.*

*Extensions: 3. Data introduced is not valid – the system will show a message ‘Invalid username or password’.*

*7.1. Employee data is not correctly introduced – the system will not store the wrong information.*

*7.2. Employee already exists – it will not be stored again, the system will show an appropriate message.*

3. System Architectural Design

**3.1 Architectural Pattern Description**

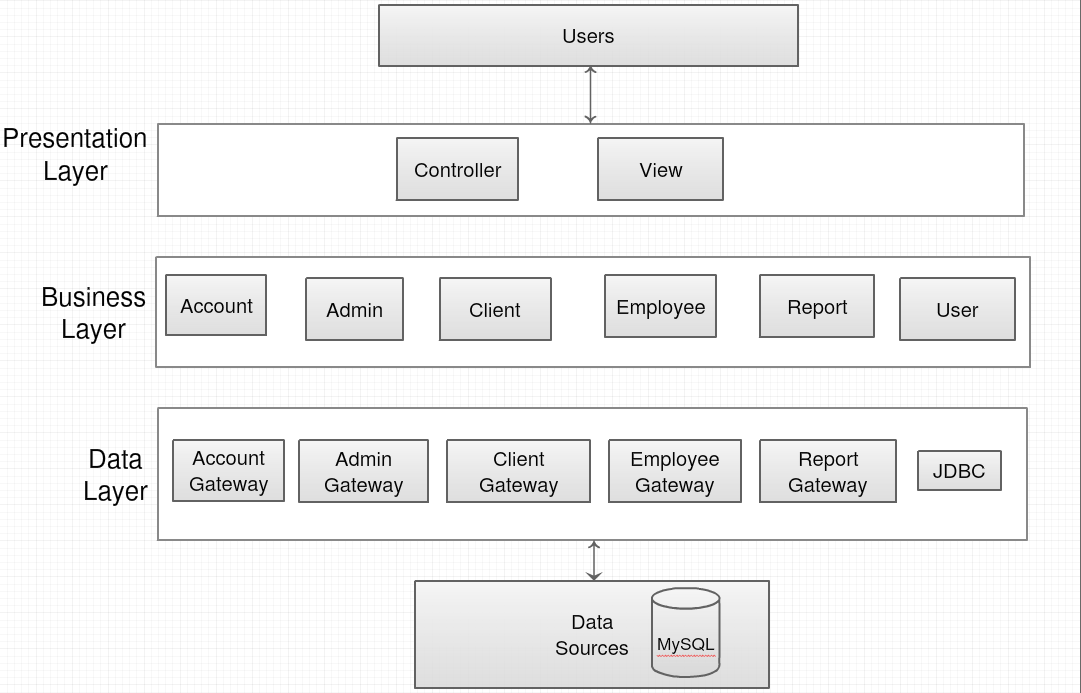
First of all, I used the Layers architectural pattern to organize my application. Talking about the three principal layers of this architectural pattern, I could start with the Presentation layer. As long as presentation logic is about how to handle the interaction between the user and the software, this is achieved by a quite tiny client graphics UI. Then, Data sourcelogic is contained in the Data source layer and it is about communicating with other systems that carry out tasks on behalf of the application. The last one is the domain logic, also reffered to as business logic. It is contained in the Business layer and it involves, in turn, calculations based on inputs and stored data.

Furtermore, from the data source hybrid patterns, I chose Table Module, which is in many ways a middle ground between Transaction Script and Domain Model. A Table Module organizes domain logic with one class per table in the database. Organizing the domain logic around tables rather than straight procedures provides more structure and makes it easier to find and remove duplication.

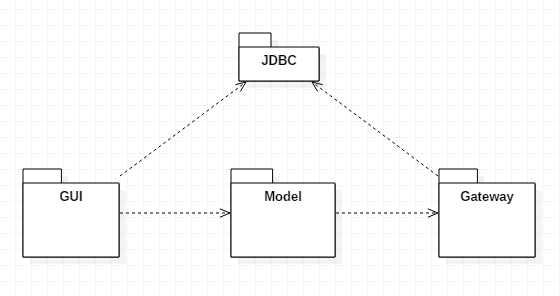
Moreover, as a data source pure pattern, I chose Table Data Gateway. In this case, the Table Data Gateway classes hold all the SQL for accessing a single table or view: selects, inserts, updates, and deletes. Other code calls its methods for all interaction with the database.

**3.2 Diagrams**

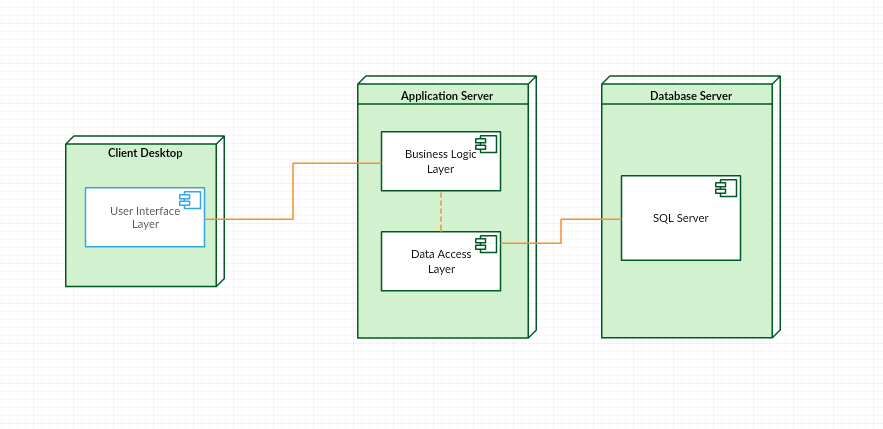
Conceptual architecture:



Package diagram:

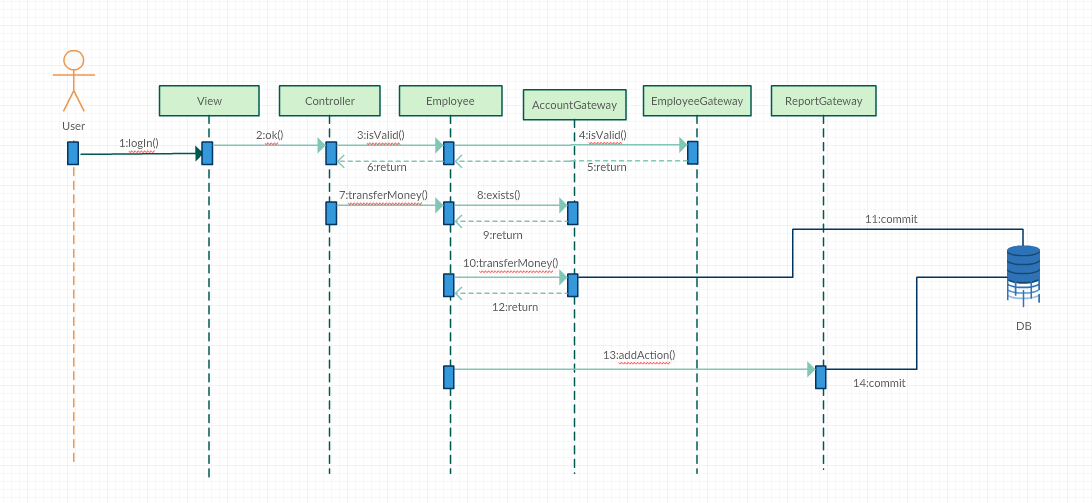


Deployment diagram:



4. UML Sequence Diagrams

Scenario: Transfer money between accounts



5. Class Design

**5.1 Design Patterns Description**

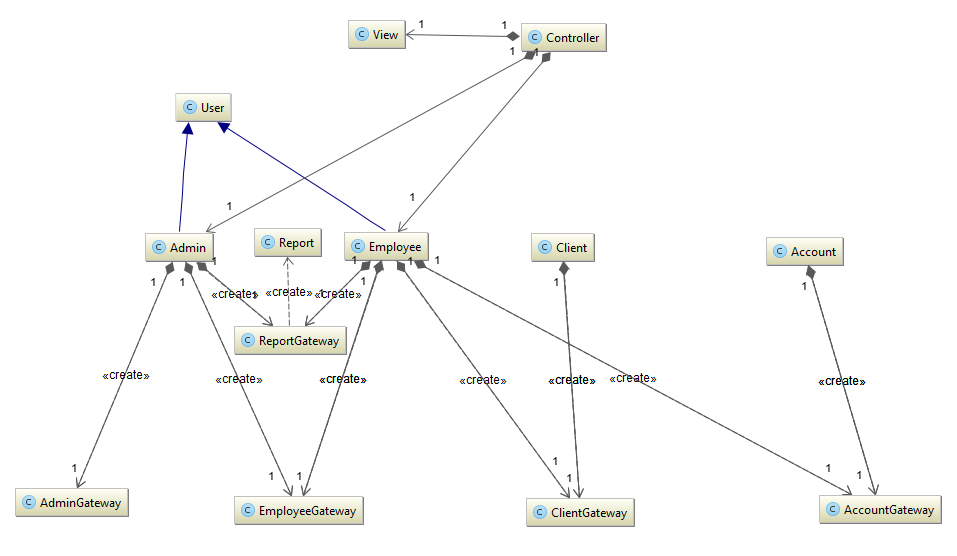
Besides using the architectural design patterns mentioned above, I used for a well-structured graphic user interface implementation, the Model-View-Controller design pattern, which  separates the modeling of the domain, the presentation, and the actions based on user input into three separate classes.

The model manages the behavior and data of the application domain, responds to requests for information about its state (from the view), and responds to instructions to change state.

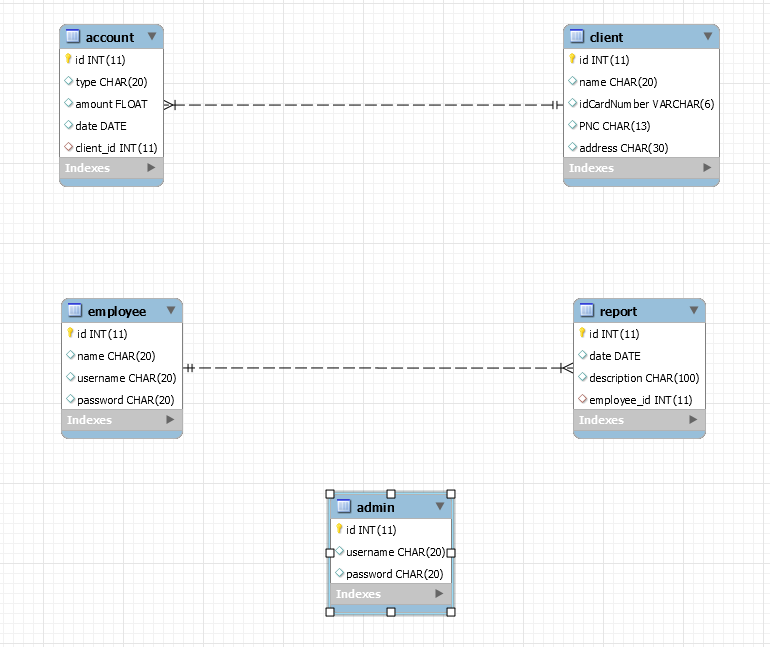
The view only manages the display of information.

Finally, the controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.

**5.2 UML Class Diagram**



6. Data Model

**

7. Bibliography

1. Patterns of Enterprise Application Architecture, By Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford
2. .NET Application Architecture xxiv Guide, 2nd Edition
3. Software architecture patterns, Mark Richards.
4. <http://searchsoftwarequality.techtarget.com/tip/Using-a-nonfunctional-requirements-template-plus-examples>
5. <https://msdn.microsoft.com/en-us/library/ff649643.aspx>