<Assignment 2>

Student: Romina Băilă

**Group: 30433**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 4

2. Use-Case Model 4

3. System Architectural Design 5

4. UML Sequence Diagrams 6

5. Class Design 6

6. Data Model 10

7. System Testing 10

8. Bibliography 11

1. Requirements Analysis

# Assignment Specification

Use JAVA API to design and implement the API part of a web 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.

Using Java API, implement the API part of a web 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.

The regular user can 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 (Pay) utilities bills (by inserting bill information).

The administrator user can perform the following operations:

- CRUD on employees’ information.

- Generate two types of reports files (one in pdf and one in csv format) for a period containing the activities performed by an employee.

# Functional Requirements

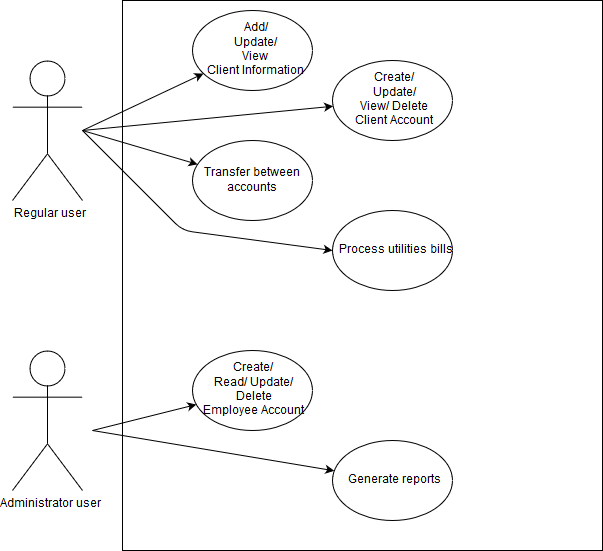
* The data must be stored in a database.
* Application must make us of the Layers design pattern.
* The Data Layer must be implemented using Spring Data.
* Each entity must have its own repository.
* All repository methods (for all entities) will be defined in repositories(interfaces). If some are not possible then hsql scripts can be used with @Query annotation on defined interface methods, or custom repositories can be created.
* All repository methods (for all entities) will be tested using unit tests.
* The business functionality of the application will be delivered by services from the Business Layer.
* Generating the reports will be done using the Factory Method design.
* The Presentation Layer will describe the API of the application using swagger.
* The Presentation Layer contains controllers that return data to the user based on request.
* Each controller exposes a specific REST API containing multiple API methods.
* Data exposed by the APIs is in JSON format.
* DTOs returned by the API methods are annotated with json annotations to be exposed in json format.
* Spring security will be used to restrict/authorize access to the exposed APIs.
* All the inputs of the application will be validated against invalid data before submitting the data and saving it in the database.

# Non-functional Requirements

Non-functional requirements ensure the usability and effectiveness of the entire system. Failing to meet any one of them can result in systems that fail to satisfy internal business, user, or market needs, or that do not fulfill mandatory requirements imposed by regulatory or standards agencies.

Some non-functional requirements are: how fast are the reports generated or the availability of the server.

2. Use-Case Model



**Use-Case:** Create Client Account

**Level:**

**Primary actor:** the regular user

**Main success scenario:** user goal

* The user enters the information provided by the client
* The account is successfully created

**Extensions:** Failure scenario

* The user enters the information provided by the client
* The account cannot be successfully created
* The user requests all the information once again from the client and tries to create the account again

3. System Architectural Design

**3.1 Architectural Pattern Description**

The design pattern used in this application is the **Layered pattern**.

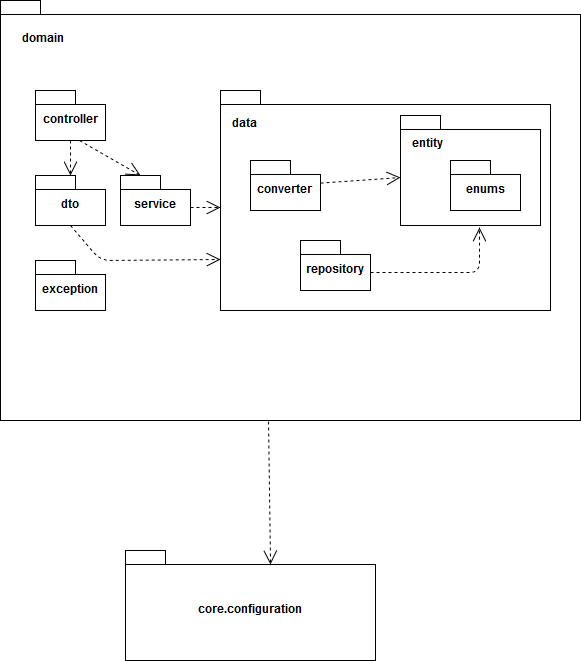
The Layered pattern is an architectural pattern, used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer.

The most common 4 layers are:

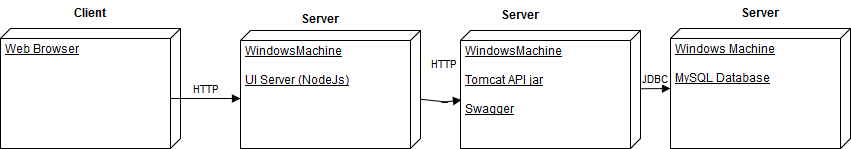
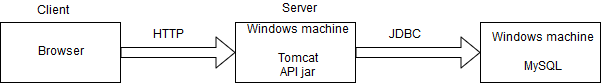
* Presentation layer
* Application layer
* Business logic layer
* Data access layer

**3.2 Diagrams**

**Package diagram:**

****

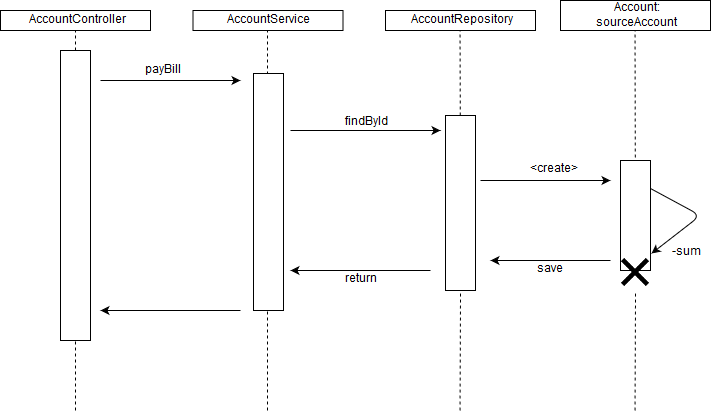
**Deployment diagram:**

**



4. UML Sequence Diagrams

Sequence diagram for payBill



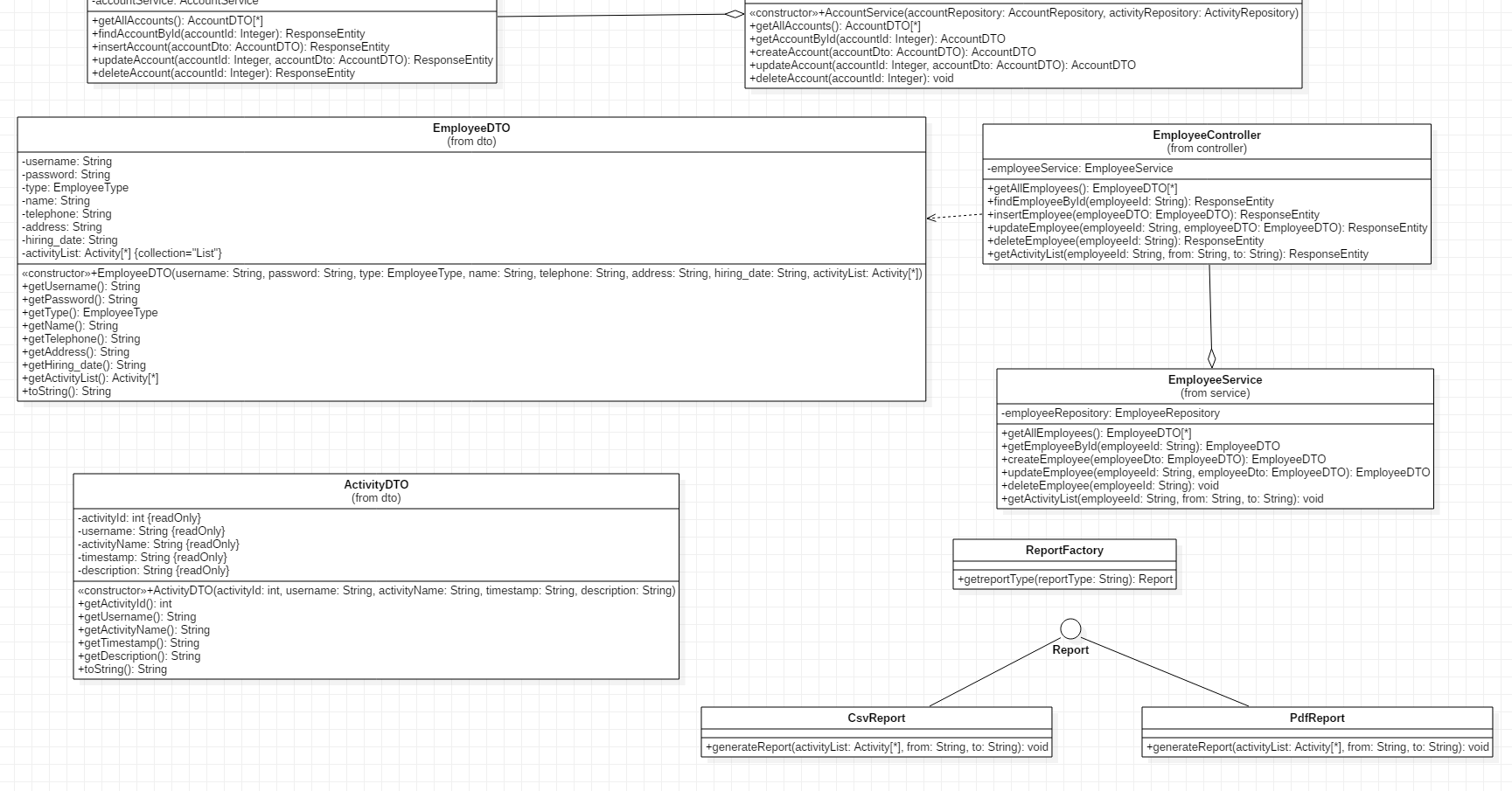
5. Class Design

**5.1 Design Patterns Description**

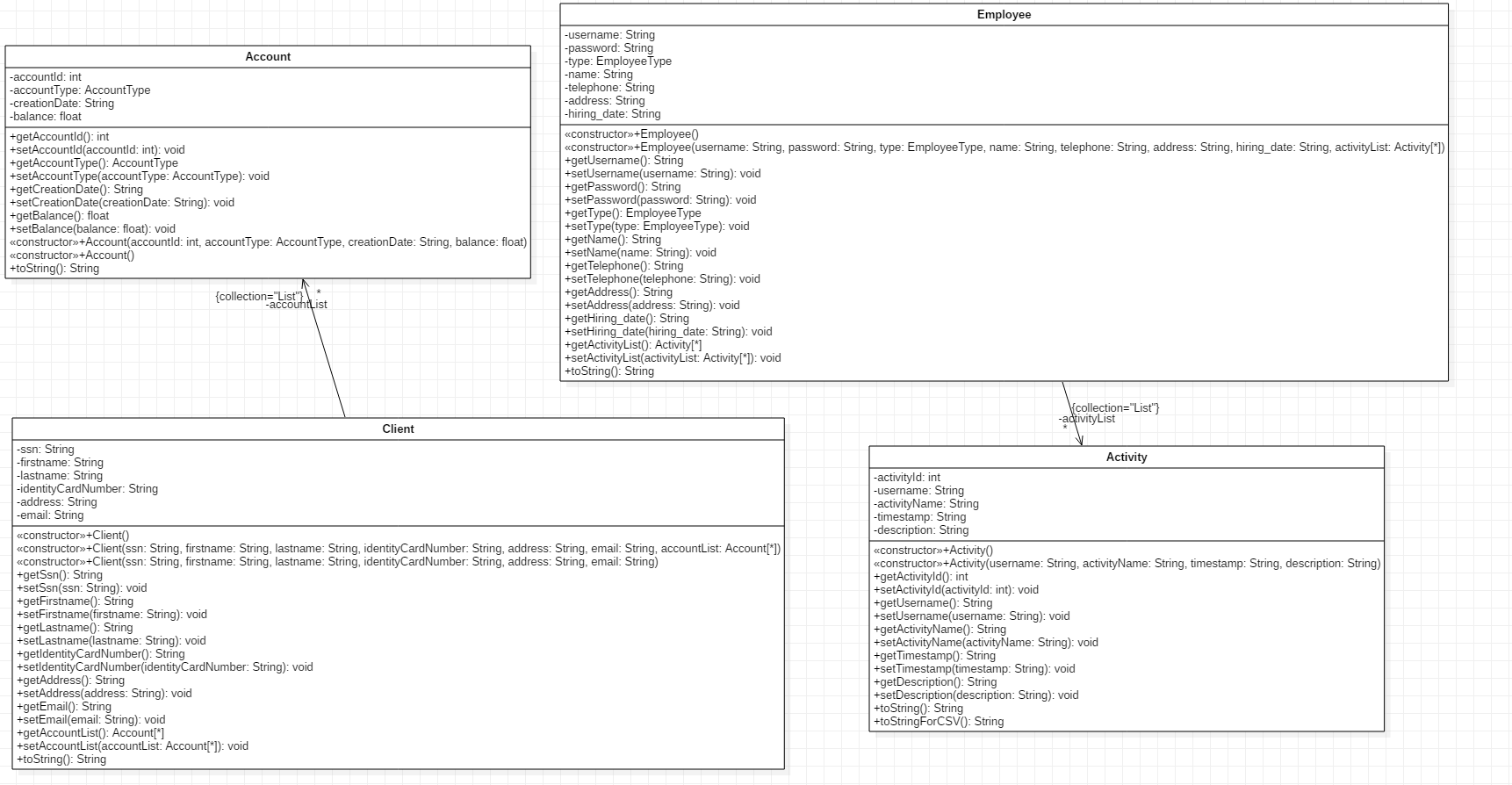
**Factory design pattern** is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. It provides a way to delegate the instantiation logic to child classes.

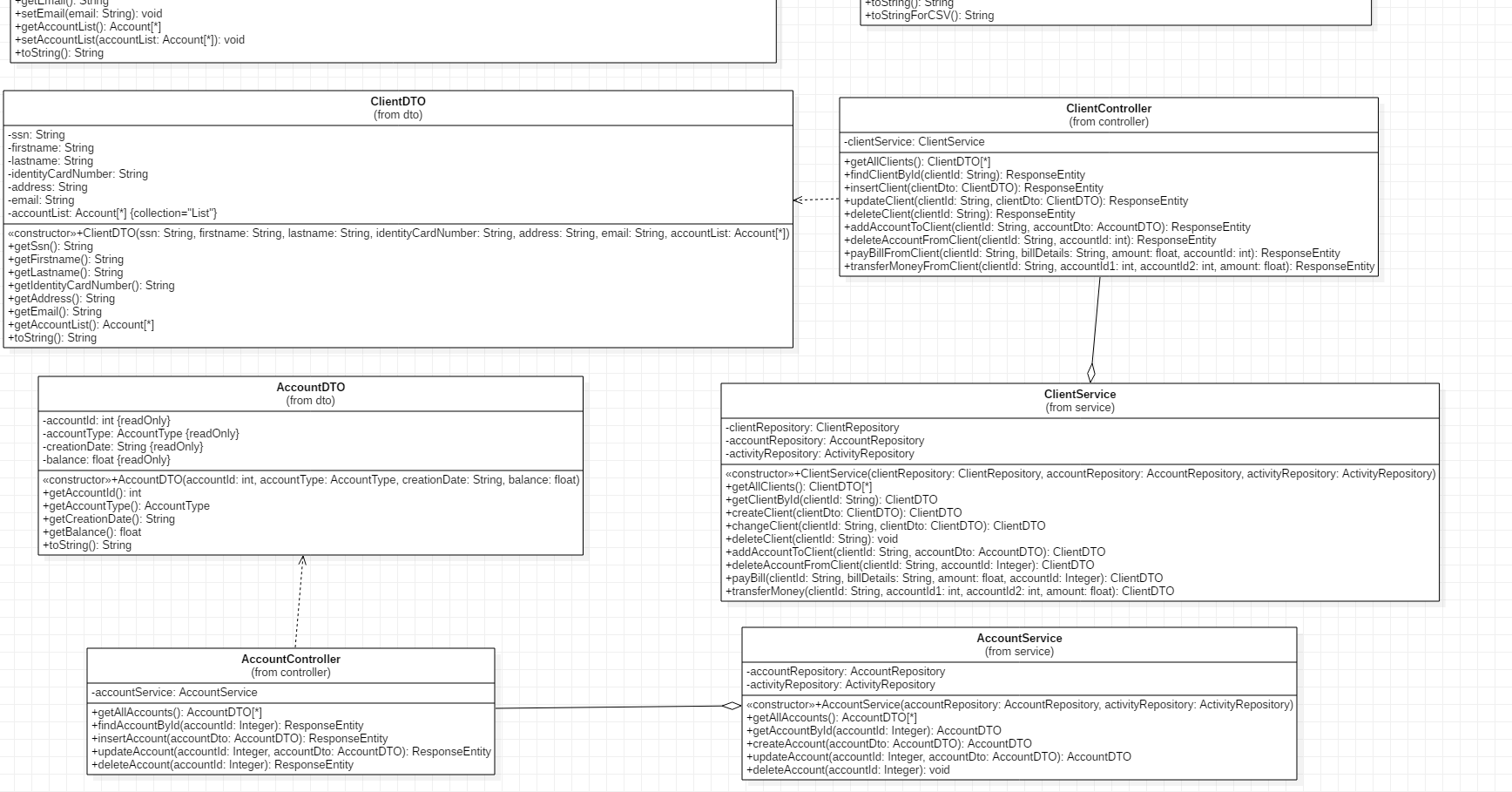
Factory method is useful when it doesn’t know what exact sub-class will be required during execution. It can be used when there is common/generic implementation in a class but the required sub-class is decided dynamically during runtime.

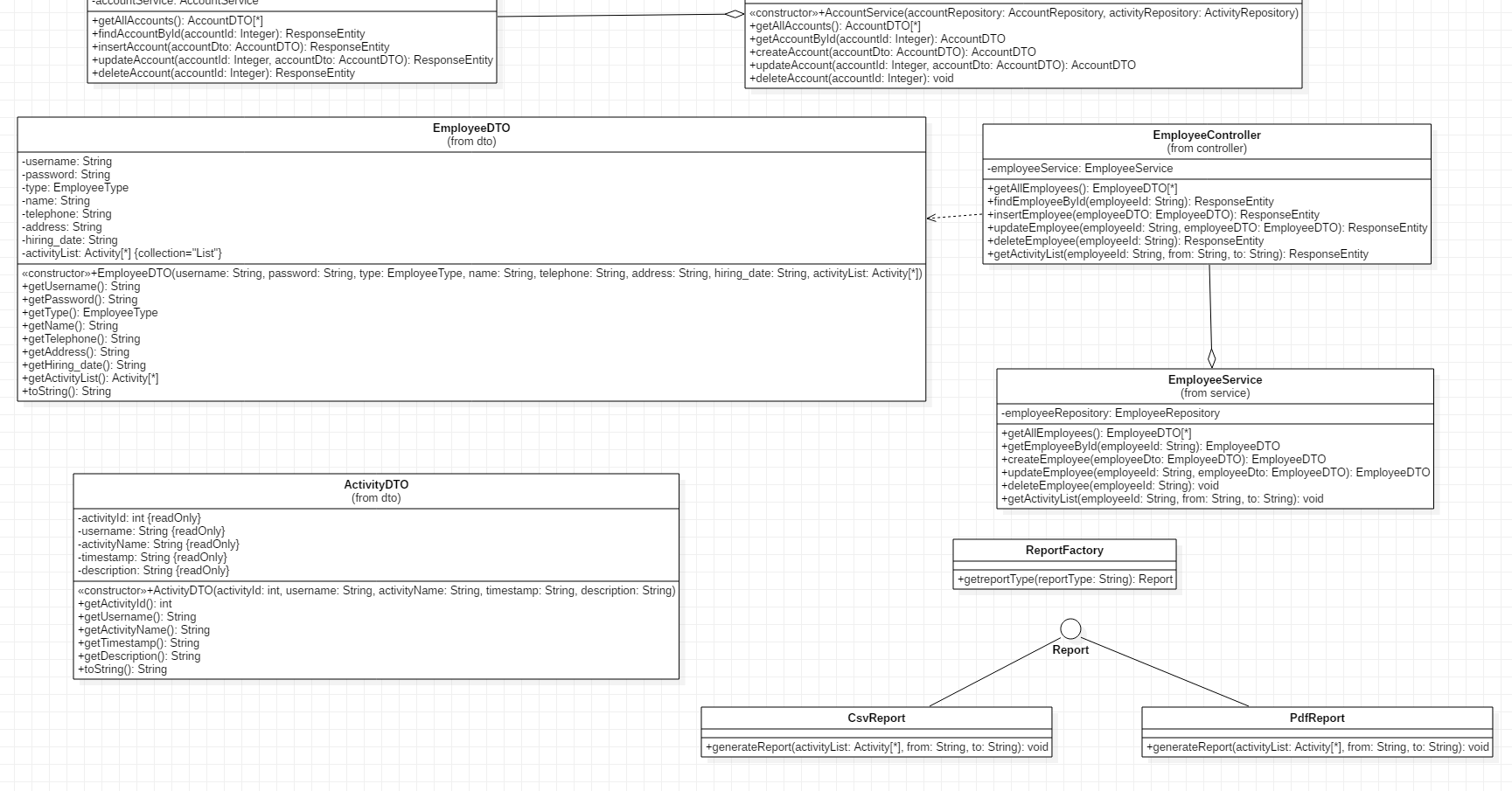
The diagram from below shows how the Factory design pattern is implemented in this application.



**5.2 UML Class Diagram**

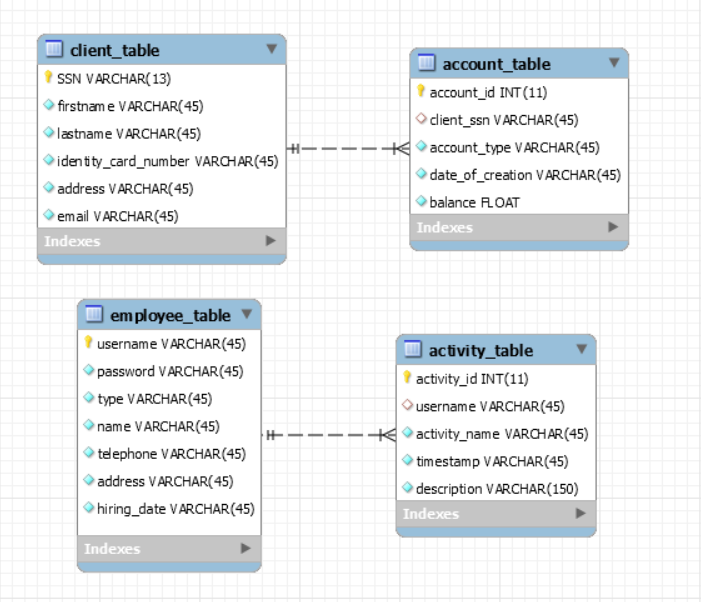
**





6. Data Model

Database diagram:



7. System Testing

The system testing was done using the Mockito framework. I chose Mockito, because during unit testing of the application, sometimes it is not possible to replicate exact production environment. Sometimes database is not available and sometimes network access is not allowed. There can be many more such restrictions. To deal with such limitations, we have to create mock for these unavailable resources. Thus, Mockito allows us to easily create test doubles (mocks).

The tests were performed on the services’ methods. The two test classes, AccountServiceTest and ClientServiceTest were implemented using Mockito.

After setting up the “mock” environment (with the help of @Before and @After annotations), I was able to write the desired tests. For AccountServiceTest those were tests for getAllAccounts, getAccountById, createAccount and updateAccount. For the ClientServiceTest class, I chose to implement the two most interesting methods, namely payBill and transferMoney.

For the UI part of the project, I did manual testing, because how can we detect error the best, if not by trial and error?

8. Bibliography

<https://stackoverflow.com/>

<https://www.scaledagileframework.com/nonfunctional-requirements/>

<https://www.baeldung.com/spring-boot-security-autoconfiguration>

<https://medium.com/@gustavo.ponce.ch/spring-boot-spring-mvc-spring-security-mysql-a5d8545d837d>

<https://javacodehouse.com/blog/mockito-tutorial/>

<https://howtodoinjava.com/mockito/junit-mockito-example/>

<https://www.baeldung.com/java-spring-mockito-mock-mockbean>

<https://medium.com/nestedif/java-factory-method-976cf4e12ec0>

<https://www.tutorialspoint.com/design_pattern/factory_pattern.htm>

<https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013>