<Assignment A1 - Architectural Patterns>

Student: Cozma Andreea

**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

Proiectati si implementati o aplicatie pentru angajatii unei banci. Aceasta va avea doua tipuri de utilizatori, un utilizator care va reprezenta angajatul bancii de la ghiseu si un administrator. Acestia se vor putea avea acces in aplicatie cu ajutorul unui username si al unei parole.

# Functional Requirements

Utilizatorul va putea efectua urmatoarele operatiu:

* Adaugarea/modificarea/vizualizarea informatiilor despre client (despre client se vor retine informatii precum nume, CNP, numarul de identificare al cardului, adresa)
* Adaugarea/modificarea/vizualizarea contului unui client (informatiile despre cont vor fi: numarul de identificare al cardului, tipul acestuia, soldul, data crearii).
* Transfera bani intre conturi
* Va procesa un raport despre tranzactii

Administratorul va putea:

* Adauga/sterge/modifica/vizualiza angajatii
* Genera rapoarte despre activitatea unui angajat pentru o anumita perioada

Toate datele introduse de utilizatori vor fi validate inainte ca acestea sa fie salvate in baza de date.

# Non-functional Requirements

Aplicatia va fi independent de platforma, fiind portabila pe orice sistem de operare datorita

implementarii in Java.

Designul aplicatiei va in in asa fel gandit incat dezolvarile ulterioare sa poata fi

implementate cat cu un efort minim. Se vor putea adauga usor generari de rapoarte noi precum sunt cele de ActivityClient si ActivityAccount care sa extinda clasa de Activity.

Se va pune accent pe securitate aplicatiei si a datelor din moment ce este o aplicatie bancara care tine informarii despre angajati, client si in special conturile acestora. Din partea de Presentation nu vor putea fi accesate direct datele si nu exista dependenta spre Layer-ul de data source.

Datele introduse de catre utilizatori sunt verificate si validate, doar in cazurile in care acestea sunt valide vor fi si persistate.

Pattern-ul de Data Mapper este util datorita testarii usoare, clasele de entity nestiind de prezenta bazei de date sau de modul in care acestea sunt create, functionalitate inglobata in Mapper-ele associate fiecare clase de domain. Pattern-ul de Domain Model este util in aplicatii complexe, de aceea s-a ales acesta in detrimentul unui pattern precum Transaction Script pentru o face posibila o dezvoltare ulterioara usoara si adaugarea unor functionalitati noi.

2. Use-Case Model

Use case: transfer bani intre conturi

Level: user-goal level

Primary actor: front desk employee

Main success scenario: suma de bani va fi retrasa din contul sursa si adaugata in contul destinatie

Extensions: pentru situatia de esec conturile raman nemodificate

3. System Architectural Design

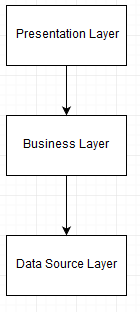
**3.1 Architectural Pattern Description**

Pattern-ul arhitectural folosit pentru organizarea aplicatiei este “Layer Architecture” cu trei nivele.

Se foloseste pattern-ul pentru domain layer: “Domain Model” iar pentru accesul la date pattern-ul “Data Mapper” care decupleaza clasele de model de baza de date.

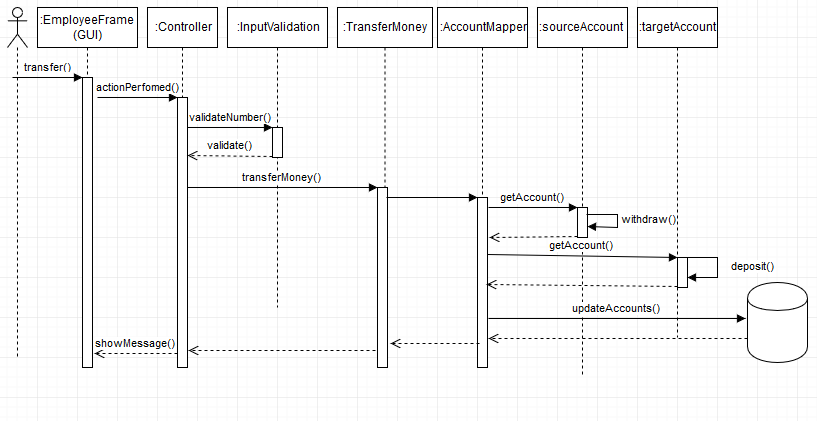
**3.2 Diagrams**

Architecture Diagram: Layer Architecture



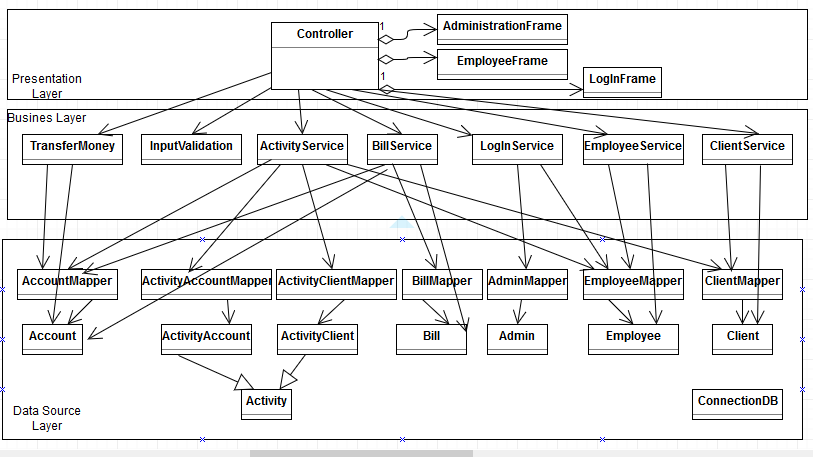
4. UML Sequence Diagrams

Scenariu: transfer de bani intre doua conturi



5. Class Design

Diagrama de clase de mai jos surprinde si arhitectura structurata pe cele 3 nivele si dependetele care exista intre acestea.



**5.1 Design Patterns Description**

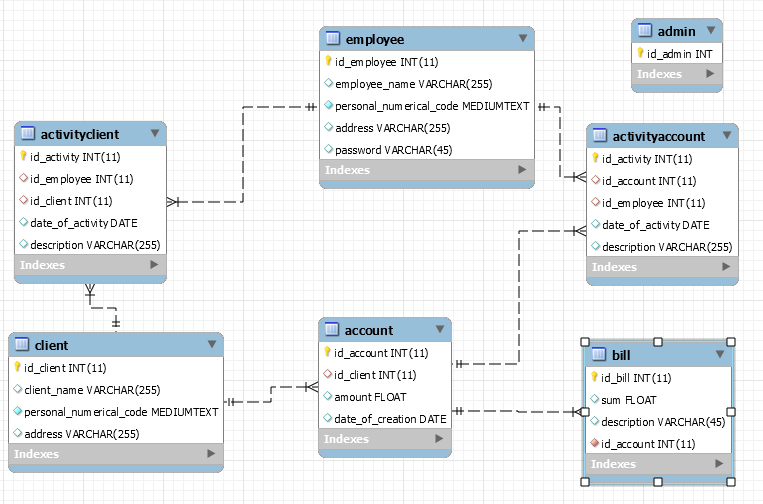
Folosirea Data Mapper-ului ca design pattern pentru accesul la date a facut decuplarea intre clasele de model ale bazei de date si clasele de entity aferente. Acestea pot evolua in mod diferit, iar fiecare clasa de Mapper care mediaza intre cele doua poate fi inlocuita/adaptata astfel incat sa suporte modificari ulterioare sau o diferenta mare intre modelul de date si modelul de entitati. Functia de map() are scopul de a crea obiecte care vor fi ulterior folosite. Am ales acest pattern datorita avantajului pe care il ofera legat de decuplarea celor doua parti.

Folosirea unui pattern precum Domain Model este utila la aplicatii cu logica mai complexa, datorita faptului ca clasele de entitati cuprind atat date cat si comportament, dupa cum se spune si in cartea “Patterns of Enterprise Application Architecture”. Se creeaza astfel un model mult mai util decat daca s-ar fi ales spre exemplu un Transaction Script care ar contine mult cod redundant.

Cele doua pattern-uri se preteaza a fi folosite impreuna.

6. Data Model

Datele sunt stocate in entitati precum Employee, Bill, Client, Admin, Account, iar datele necesare generarii rapoartelor sunt stocate in tabelele de Activity, unde se tin cheile spre client si angajat, respectiv spre cont si angajat, pentru a se genera atat rapoarte despre actiunile facute asupra conturilor, cat si asupra clientilor.

**

7. System Testing

Fiecare functionalitate nou integrata a fost testate dupa finalizarea acesteia.

Datorita Pattern-ului de Data Mapper a fost usoara testarea functionalitatilor si cu date fictive, deoarece clasele de entity nu au dependente spre baza de date. Ulterior a fost facuta testarea cu datele reale persistente introduse in tabele prezentate in diagrama de Data Model.

8. Bibliography

1. Martin Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2003