Vinit Bharat Shah – A20350453

CS- 586 Software System Architecture

Project Report on: MDA-EFSM Account System

1. **Introduction**

The goal of this project is to design two different Account components, which are completely different from each other but still supports certain operations which are common. The design of project is such that it supports the different operations of different accounts such as entering pin, making deposit, withdrawing money, viewing balance etc. All these operations are completely different from each other for each of the account component but still they have been incorporated as a single system with the help of MDA-EFSM. Design also includes usage of three different design patterns which promotes totally de-coupling and high cohesion among components. Design patterns included in project design are State Pattern, Strategy Pattern, and Abstract Factory Pattern. State patterns helps in the transition of states while the strategy pattern that enables the algorithm behaviour to be selected at run-time. Abstract Factory pattern provides an interface to create a family of related objects.

**Model Driven Architecture of Account Components**

Output Processor

Deposit ()

Withdraw ()

……

Input Processor

MDA-EFSM

Account-1

Input events

Output Events

DataStore

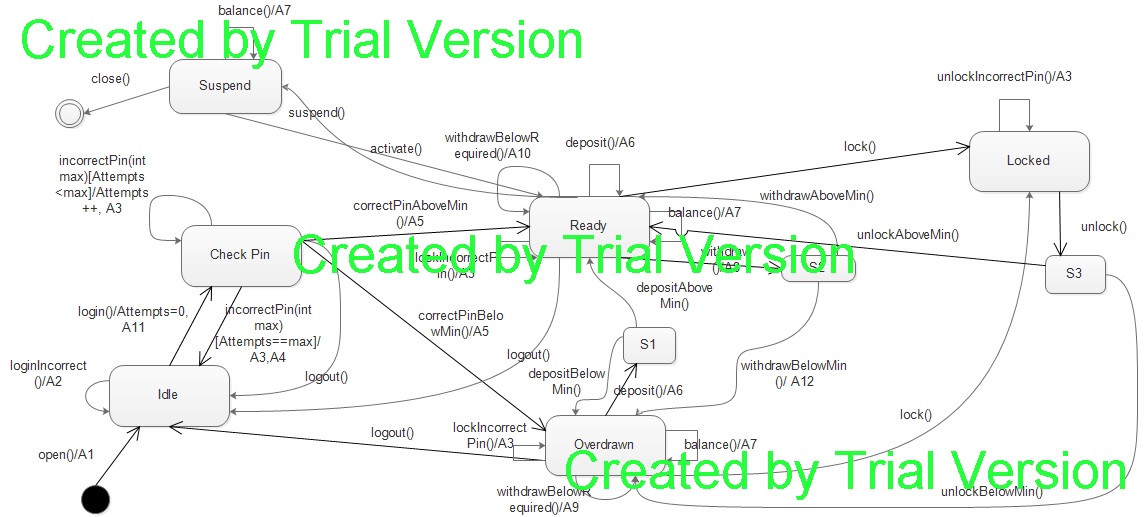
Account -2

1. **A list of events for the MDA-EFSM**
2. open()
3. login()
4. loginIncorrect()
5. incorrectPin(int max)
6. correctPinAboveMin()
7. correctPinBelowMin()
8. logout()
9. deposit()
10. depositAboveMin()
11. depositBelowMin()
12. balance()
13. withdraw()
14. withdrawAboveMin()
15. withdrawBelowMin()
16. withdrawBelowRequired()
17. lock()
18. lockIncorrectPin()
19. unlockIncorrectPin()
20. unlock()
21. unlockBelowMin()
22. unlockAboveMin()
23. suspend()
24. activate()
25. close()
26. **A list of actions for the MDA-EFSM with their descriptions**
27. A1: storeData()
28. A2: incorrectIDMsg()
29. A3: incorrectPinMsg()
30. A4: tooManyAttemptsMsg()
31. A5: displayMenu()
32. A6: makeDeposit()
33. A7: displayBalance()
34. A8: makeWithdraw()
35. A9: withdrawBelowMinMsg()
36. A10: displayNoFundsMsg()
37. A11: promptForPin()
38. A12:Penalty()

**Responsibility of each action:**

1. A1: storeData(): Stores pin, userId and balance for account from temporary data store
2. A2: incorrectIDMsg() : Display the message of Incorrect ID when login attempt is failed with invalid user ID.
3. A3: incorrectPinMsg(): Displays the message of Incorrect Pin when user enters wrong pin
4. A4: tooManyAttemptsMsg(): Display the message of Too many attempts when user exceeds the maximum number of allowed attempts.
5. A5: displayMenu() : display a menu with a list of selections
6. A6: makeDeposit(): Executes the deposit functionality for the account based on available balance.
7. A7: displayBalance(): displays the current balance for the account
8. A8: makeWithdraw() : Executes the withdraw functionality for the account based on the current balance available.
9. A9: withdrawBelowMinMsg(): Displays message notifying the account balance is below the minimum value required for the account.
10. A10: displayNoFundsMsg(): Displays the message notifying there is not sufficient balance or funds available to withdraw the money.
11. A11: promptForPin(): Displays the message asking for a pin of account after login.
12. A12: Penalty() :Penalty will get applied when the balance goes below minimum required value on withdraw.

**iii. A state diagram of the MDA-EFSM**



1. **Pseudo-code of all operations of Input Processors of *ACCOUNT-1* and *ACCOUNT-2***

Pseudo Code of operation of Input Processor for Account 1:

open (string p, string y, float a){

d->temp\_p=p

d->temp\_y=y

d->temp\_a=a;

m->open()

}

login(string y){

if(y==d->uid){

m->login();

}

Else{

m->loginIncorrect();

}}

pin (string x){

if(x==d->pin){

if(d->balance>500){

m->correctPinAboveMin();

}else{

m->correctPinBelowPin();

}}

Else{

m->incorrectPin(3);

}

}

logout(){

m->logout();

}

balance(){

m->balance();

}

deposit (float d){

d->temp\_d=d;

m->deposit();

if(d->balance>500){

m->depositAboveMin();

}

Else

m-> depositBelowMin();

}

withdraw (float w){

d->temp\_w=w;

if(d->balance>500){

m->withdraw();

if(d->balance>d->temp\_w+500){

m->withdrawAboveMin();

}else if(d->balance-d->temp\_w<=500){

m->withdrawBelowMin(); //Penalty will get applied

}

}

if(d->balance<500){

m->withdrawBelowRequired();

}

}

lock(string x){

if(x==d->pin){

m->lock();

}else{

m->lockIncorrectPin();

}

}

unlock(string x){

if(x==d->pin){

m->unlock();

if(d->balance>500){

m->unlockAboveMin();

}elseif(d->balance<=500){

m->unlockBelowMin();

}

}else{

m->unlockIncorrectPin();

}

}

Pseudo Code of operation of Input Processor for Account 2

OPEN (int p, int y, int a){

d->temp\_p=p

d->temp\_y=y

d->temp\_a=a;

m->open()

}

LOGIN(int y){

if(y==d->uid){

m->login();

}else{

m->loginIncorrect();

}

}

PIN (int x){

if(x==d->pin)

m->correctPinAboveMin();

Else

m->incorrectPin(2);

}

LOGOUT(){

m->logout();

}

BALANCE (){

m->balance();

}

DEPOSIT (int d){

d->temp\_d=d;

m->deposit();

}

WITHDRAW (int w){

d->temp\_w=w;

if(d->balance>0){

m->withdraw();

m->withdrawAboveMin();

}else{

m->withdrawBelowRequired();

}

}

suspend(){

m->suspend();

}

activate(){

m-> activate ();

}

close(){

m-> close ();

}

1. **Class diagram(s) of the MDA of the ACCOUNT components.**

**The entire class Diagram is divided into different modules along with reference of its connections with their respective modules and Design Pattern is also specified:**

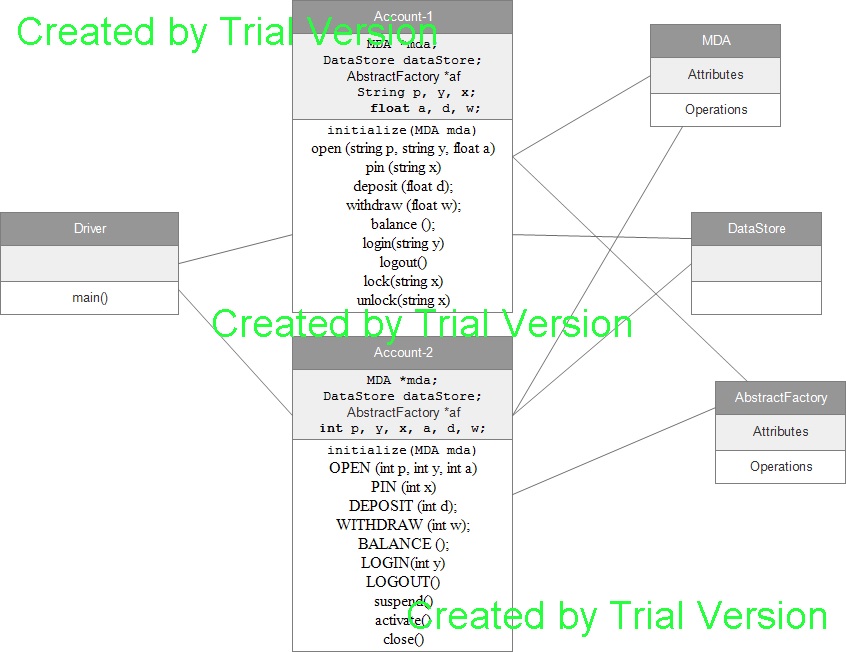
**The whole system can be seen as divided into 3 sub divisions**

1. Input Processor

2. MDA-EFSM

3. Output Processor

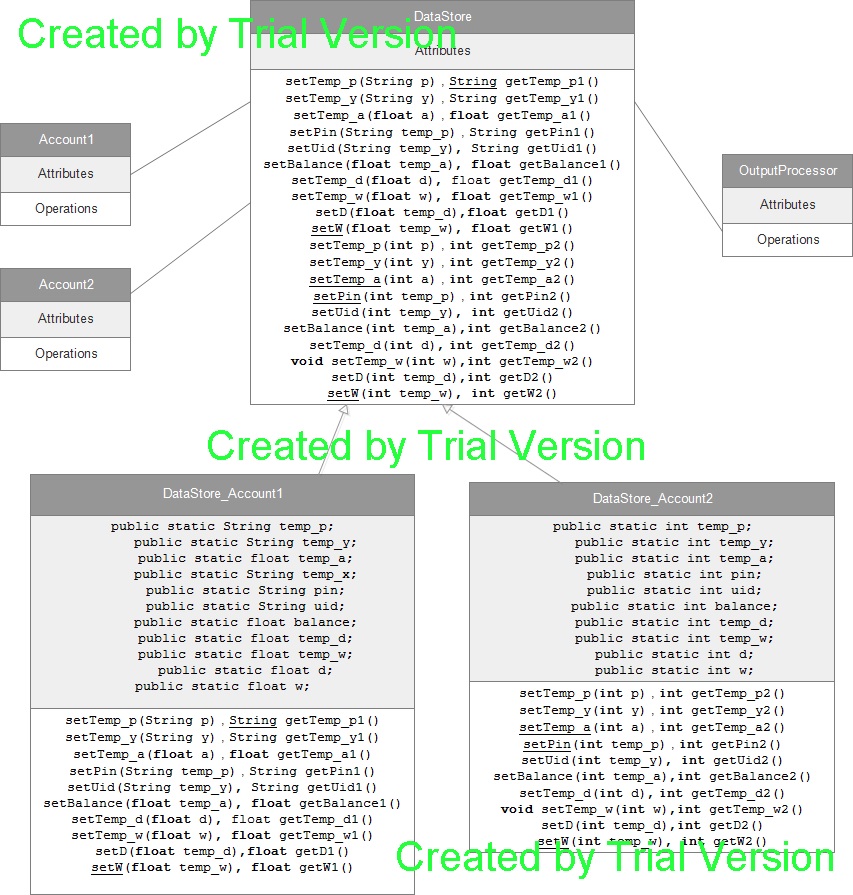
**Input Processor Class Diagram & Operations:**



This is the basic class diagram of Input Processor.

Main Program: This consists of functions for user to select the Account version out of 2 versions and based on that it creates the concrete factory and other related links and objects which are used by MDA and output processor and other classes. It accepts arguments based on the options selected by users and passes them on to Account class requested.

**Data Store Class:**



**MDA: (State Pattern):**

MDA consists of the following components

1. MDA

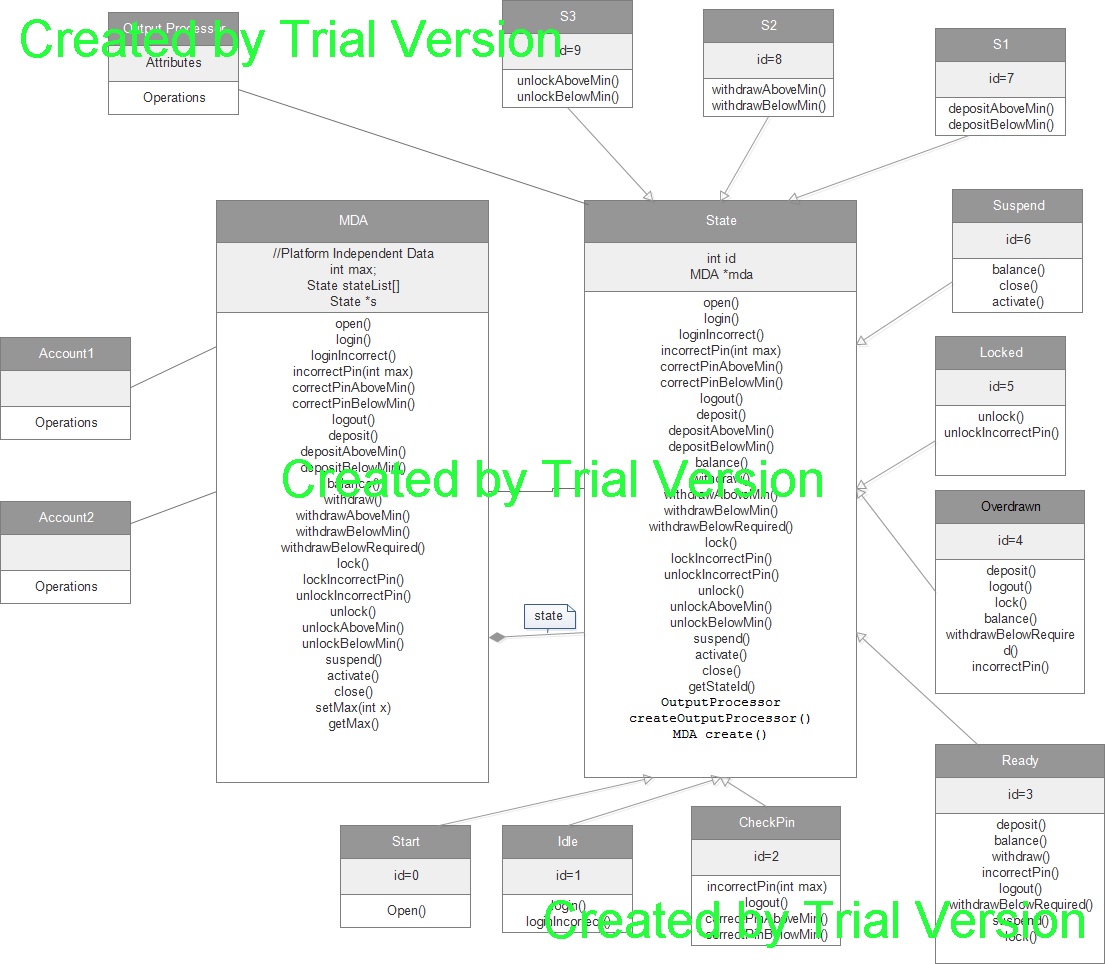
2. States

States:

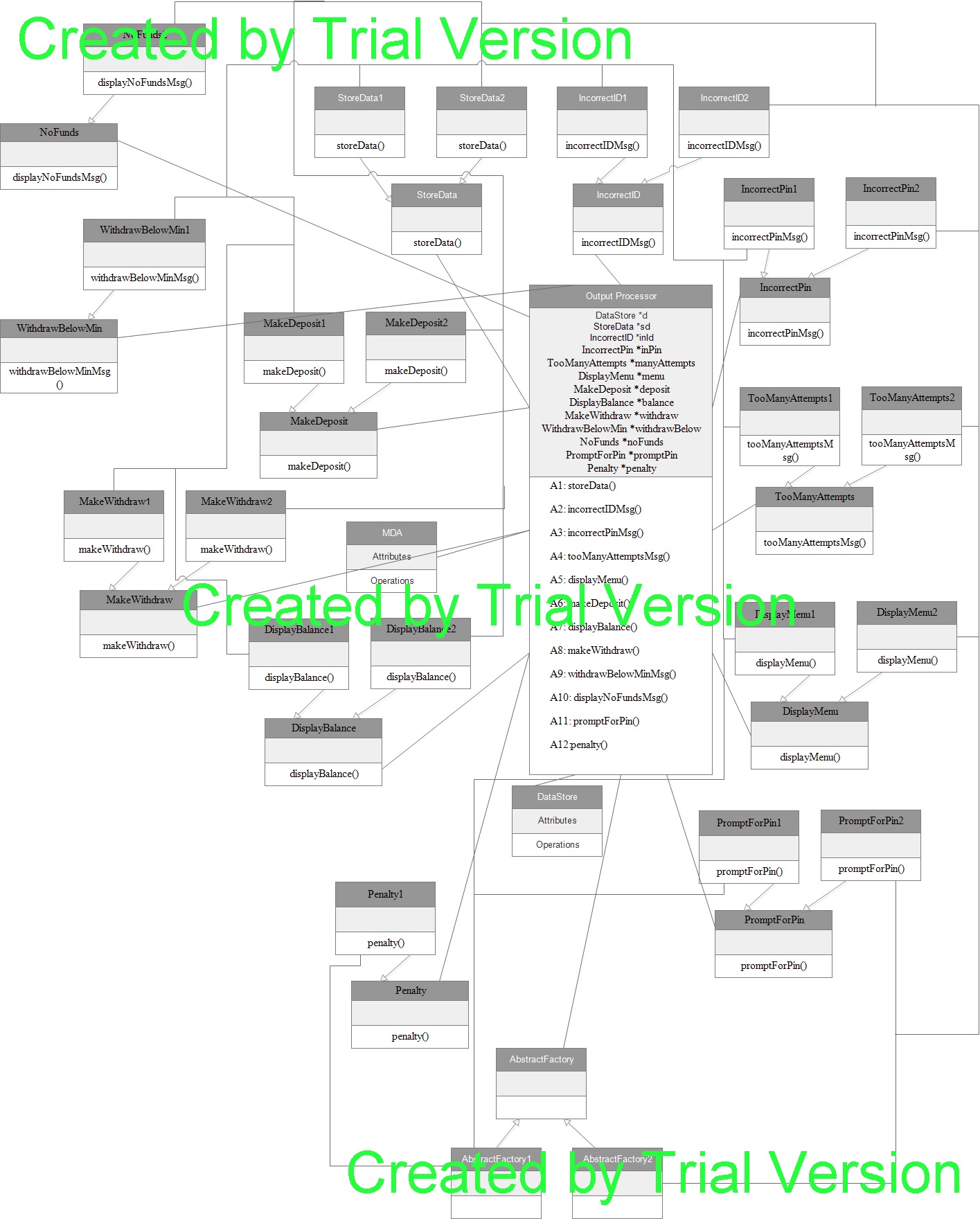
This is the class which is used by all state objects it consist of all methods that are going be called by state objects.

Attributes of State Class:

Id: it is integer that will point to current state



**Output Processor (Strategy Pattern):**



**Abstract Factory Pattern:**

****

1. **Details for each class in the class diagram(s):**

**Class Driver:**

This class is a driver class for account system, asks user’s to select between accounts.

MDA mda = **new** MDA(); //pointer to MDA

Main()// Drives an account system between account 1 and 2

**Class DataStore:**

This is an abstract class, which provides an operations to set and get the data required during processing of operations in account.

**Class DataStore\_Account1:**

Used to set and get the data required during operations of account 1, It provides setters and getters methods for all the following variables, which is useful to set and get the data during operations.

public static String *temp\_p*;//pin temp.

public static String *temp\_y*;//user id temp

public static float *temp\_a*;//account balance temp

public static String *temp\_x*;//pin temp

public static String *pin*;//pin

public static String *uid*;//userid

public static float *balance*;//balance

public static float *temp\_d*;//depeosit amount temp

public static float *temp\_w*;//withdraw amount temp

public static float *d*;//deposit amount

public static float *w*;//withdraw amount

**Class DataStore\_Account2:**

Used to set and get the data required during operations of account 2, It provides setters and getters methods for all the following variables, which is useful to set and get the data during operations

public static int *temp\_p*;//pin temp

public static int *temp\_y*;//user id temp

public static int *temp\_a*;//balance temp

public static int *pin*;//pin

public static int *uid*;//user id

public static int *balance*;//balance

public static int *temp\_d*;//deposit temp

public static int *temp\_w*;//withdraw amount temp

public static int *d*;//deposit amount

public static int *w*;//withdraw amount

**Class Account1:**

It’s an input processor

MDA mda;//mda object pointer

DataStore dataStore; //data store pointer

initialize(MDA mda) // initialize the needed parameters for account 1, such as abstract factory, and datastore

menu() : // method is used to get an input of an operation required to be performed by user for account 1, like open(),pin().deposit(), etc., based on user input it takes an action by calling a respective function in MDA.

**Class Account2:**

It’s an input processor

MDA mda;//mda object pointer

DataStore dataStore; //data store pointer

initialize(MDA mda) // initialize the needed parameters for account 2, such as abstract factory, and datastore

menu() : // method is used to get an input of an operation required to be performed by user for account 2, like open(),pin().deposit(),suspend() etc., based on user input it takes an action by calling a respective function in MDA.

**Class MDA:**

**int** *max* = 0;// stores number of attempts of incorrect pin

State stateList[] //maintains state list for account system

State s// holds current state of account system

Calls an events on state based on operation performed by user on account, following are operations supported

Open()//called on open operation on account

login()///called on login operation on account

loginIncorrect()// called if login in correct

incorrectPin(**int** max)//if user enters wrong pin

correctPinAboveMin()//if pin is correct and balance is above required value

correctPinBelowMin() //if pin is correct and balance is below required value

logout() //logout from account system

deposit()//deposit an amount into account

depositAboveMin() //deposit an amount and that brings account balance to above required value

depositBelowMin()//deposit an amount and that brings account balance to below required value

balance ()//show balance

withdraw()//withdraw an amount from account

withdrawAboveMin()//withdraw an amount from account and balance is above required value

withdrawBelowMin()//withdraw an amount from account and balance is below required value

withdrawBelowRequired()////withdraw an amount from account and balance is below required value

lock()//lock an account

lockIncorrectPin()//if incorrect pin is entered

unlockIncorrectPin()//incorrect pin entered while unlocking

unlock()//unlock an account

unlockAboveMin()//unlock an account and balance is above required value

unlockBelowMin()//unlock an account and balance is below required value

suspend()//suspend an account

activate()//activate a suspend account

close()//close an account system

getMax()//returns number of incorrect attempts while entering pin

setMax(**int** max) //set number of incorrect attempts while entering pin

**Class OutputProcessor:**

DataStore *d*;//pointer to data store

StoreData *sd*;//pointer to store data strategy action

IncorrectID *inId*;//pointer to Incorrect ID strategy action

IncorrectPin *inPin*;//pointer to IncorrectPin strategy action

TooManyAttempts *manyAttempts*;//pointer to TooManyAttempts strategy action

DisplayMenu *menu*;//pointer to DisplayMenu strategy action

MakeDeposit *deposit*;//pointer to MakeDeposit strategy action

DisplayBalance *balance*;//pointer to DisplayBalance strategy action

MakeWithdraw *withdraw*;//pointer to MakeWithdraw strategy action

WithdrawBelowMin *withdrawBelow*;//pointer to WithdrawBelowMin strategy action

NoFunds *noFunds*;//pointer to NoFunds strategy action

PromptForPin *promptPin*;//Pointer to PromptForPin strategy action

Penalty *penalty*;//Pointer to Penalty strategy action

getDataStore()// returns data store pointer

storeData() //calls action on storedata to store temp values

incorrectIDMsg() //calls an action to print incorrect ID msg

incorrectPinMsg()////calls an action to print incorrect pin msg

tooManyAttemptsMsg()////calls an action to print too ManyAttempts msg

displayMenu()//calls an action to display menu for respective account

makeDeposit()//calls an action to deposit in account

displayBalance()////calls an action to show balance for an account

makeWithdraw()//calls an action to make withdraw from account

withdrawBelowMinMsg(){//calls a action to display a msg saying balance is below required value

displayNoFundsMsg()//calls an action to display no funds msg

promptForPin()//calls an action to ask for pin for account

penalty(){//calls an action to apply penalty when user tries to withdraw and balance goes below minimum balance

**Class State:**

It’s an abstract class

MDA create()// returns an object of MDA

OutputProcessor createOutputProcessor()//creates and returns object of output processor

It defines an abstract method for Account 1 and 2 which is to be implemented by all the states which extends this state.

**Class Start:**

Start state of system

Id=0

open()//calls to store data on output processor

**Class Idle:**

Id=1

login() //does login into the system and asks for pin along with sets max attempts to 0

loginIncorrect()// if incorrect ID then calls an incorrectID action on output processor

**Class CheckPin:**

Id=2

incorrectPin()//calls an action on output processor when user enters incorrect pin also increments number of attempts in MDA.

correctPinAboveMin()// when pin is correct and balance is above required value then calls an action on output processor

correctPinBelowMin()//when pin is correct and balance is below required value then calls an action on output processor

**Class Ready:**

Id=3

deposit() //calls an action on output processor to deposit an amount

balance()//calls an action on output processor to show balance

withdraw ()// calls an action on output processor to withdraw from account

withdrawBelowRequired ()//calls an action on output processor when balance is below the required min. value

lockIncorrectPin() // calls an action on output processor when user enters incorrect pin while lock operation

**Class Overdrawn:**

Id=4

balance()//calls an action on output processor to show balance

withdrawBelowRequired()//calls an action on output processor when balance is below the required min. value

lockIncorrectPin()//calls an action on output processor when user enters incorrect pin while lock operation

**Class Locked:**

Id=5

unlockIncorrectPin()//calls an operation on output processor when user enters incorrect pin while unlocking

**Class Suspend:**

Id=6

balance()//calls an action on output processor to show balance

**Class S1:**

Temporary state for deposit

Id=7

**Class S2:**

Temporary state for withdraw

Id=8

**Class S3:**

Temporary state for unlock

Id=9

**Class DisplayBalance:**

an abstract class with abstract method displayBalance(DataStore d)

**Class DisplayBalance1:**

displayBalance(DataStore d)//display the balance of account 1

**Class DisplayBalance2**

displayBalance(DataStore d)//display the balance of account 2

**Class DisplayMenu**

an abstract class with abstract method displayMenu()

**Class DisplayMenu1:**

displayMenu()//display a menu for account 1

**Class DisplayMenu2:**

displayMenu()//display a menu for account 2

**Class IncorrectID:**

an abstract class with abstract method incorrectIDMsg ()

**Class IncorrectID1:**

incorrectIDMsg()//show incorrect ID msg for account 1

**Class IncorrectID2:**

incorrectIDMsg()//show incorrect ID msg for account 2

**Class IncorrectPin:**

an abstract class with abstract method incorrectPinMsg ()

**Class IncorrectPin1**

incorrectPinMsg()//show IncorrectPin msg for account 1

**Class IncorrectPin2**

incorrectPinMsg()//show IncorrectPin msg for account 2

**Class MakeDeposit**

an abstract class with abstract method makeDeposit(DataStore d)

**Class MakeDeposit1**

makeDeposit(DataStore d)//perform deposit operation for account 1

**Class MakeDeposit2:**

makeDeposit(DataStore d)//perform deposit operation for account 2

**Class MakeWithdraw**

an abstract class with abstract method makeWithdraw(DataStore d)

**Class MakeWithdraw1**

makeWithdraw(DataStore d)//withdraw action for account 1

**Class MakeWithdraw2**

makeWithdraw(DataStore d)//withdraw action for account 2

**Class NoFunds:**

an abstract class with abstract method displayNoFundsMsg ()

**Class NoFunds2:**

displayNoFundsMsg()//Display No Funds msg for account 1

**Class Penalty:**

an abstract class with abstract method penalty(DataStore d)

**Class Penalty1**

penalty(DataStore d)//apply penalty for account 1

**Class PromptForPin:**

an abstract class with abstract method promptForPin()

**Class PromptForPin1**

promptForPin()//ask for Pin Account 1

**Class PromptForPin2:**

promptForPin()//ask for Pin Account 2

**Class StoreData:**

an abstract class with abstract method storeData(DataStore d)

**Class StoreData1:**

storeData(DataStore d)//storing temp. data in data store for account 1

**Class StoreData2:**

storeData(DataStore d)//storing temp. data in data store for account 2

**Class TooManyAttempts:**

an abstract class with abstract method tooManyAttemptsMsg()

**Class TooManyAttempts1:**

tooManyAttemptsMsg()//show a msg of too many attempts of incorrect pin for account 1

**Class TooManyAttempts2:**

tooManyAttemptsMsg()//show a msg of too many attempts of incorrect pin for account 2

**Class WithdrawBelowMin:**

an abstract class with abstract method withdrawBelowMinMsg()

Class WithdrawBelowMin1:

WithdrawBelowMin1()//show the msg of balance is below the required minimum balance

1. **Dynamics. Provide two sequence diagrams for two Scenarios**

**b. Scenario-II should show as to how an incorrect pin is entered three times in the *ACCOUNT-2* component, i.e., the following sequence of operations is issued: *OPEN(123,111,1000), LOGIN(111), PIN(112), PIN(222), PIN(333)***

**Part 1:**

****

**Part 2:**

****

**Part 3:**

****

***a.* Scenario-I should show as to how the deposit is made in the *ACCOUNT-1* component, i.e.,**

**the following sequence of operations is issued: *open(abc,xyz,100.5), login(xyz), pin(abc),***

***deposit(400), balance(), logout()***

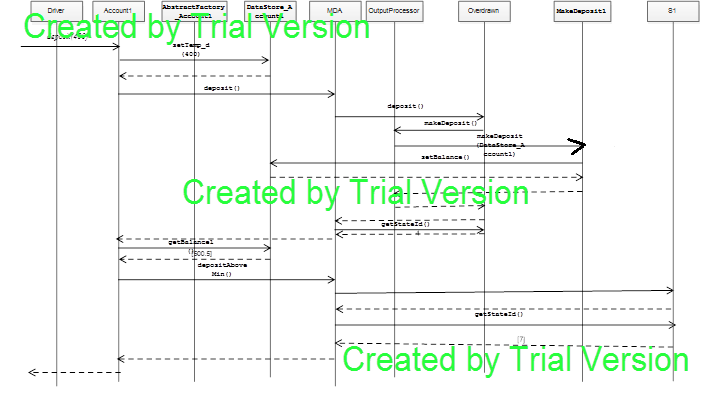
**Part1:-**



**Part 2:**



**Part 3:**

****

**Part 4:**



1. **Source-code with Design patterns and 6. Source Code**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Driver Class for Account System Start\*\*\*\*\*\*\*\*\*\*\*

package com.ssa.driver;

import java.util.Scanner;

import com.ssa.inputprocessor.Account1;

import com.ssa.inputprocessor.Account2;

import com.ssa.mda.MDA;

//Driver Class for Account 1 and Account 2

public class Driver {

public static void main(String args[]) {

MDA mda = new MDA();//MDA which act as a controller for both account 1 and 2

System.out.println("#######################################################################");

System.out.println("CS:586 Software System Architecture ");

System.out.println("Model Driven Architecture- Account System");

System.out.println("By Vinit Bharat Shah - #A20350453");

System.out.println("#######################################################################");

char choice = 'z';

while (choice != 'q') {

//Asks for user's choice to choose between Account 1 and 2

Scanner scanner = new Scanner(System.in);

System.out.println("Select the Account of your choice");

System.out.println("1. Account1");

System.out.println("2. Account2");

System.out.println("q. Exit");

choice = scanner.next().charAt(0);

switch (choice) {

//Driving the Account system based on user's choice

case '1':

System.out.println("You Selected Account1");

Account1 account1 = new Account1();

account1.initialize(mda);//initializing Account1

break;

case '2':

System.out.println("You Selected Account2");

Account2 account2 = new Account2();

account2.initialize(mda);//initializing Account2

break;

default:

System.out.println("Please select a valid Account");

break;

}

}

System.out.println("Thank you ! Good Byeee !!!");

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Driver Class End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\* Start of Input Processor Account 1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.ssa.inputprocessor;

import java.util.Scanner;

import com.ssa.datastore.DataStore;

import com.ssa.datastore.DataStore\_Account1;

import com.ssa.factory.AbstractFactory;

import com.ssa.factory.AbstractFactory\_Account1;

import com.ssa.mda.MDA;

public class Account1 {

MDA mda;//mda object pointer

DataStore dataStore; //data store pointer

String p, y, x;

float a, d, w;

public void initialize(MDA mda) {

AbstractFactory account1\_Concrete = new AbstractFactory\_Account1();//getting an abstract factory for account1

account1\_Concrete.initialize(account1\_Concrete);

this.mda = mda;

dataStore = new DataStore\_Account1();//initialization of datastore for Account 1

menu();

}

public void print() {

System.out.println("This is Account1");

}

//once started, user will given a list of options to choose from menu

public void menu() {

int choice;

do {

System.out.println("\n");

System.out.println("Select the Operation you would like to perform in Account-1");

//operations supported by Account 1

System.out.println("1. open (string p, string y, float a)");

System.out.println("2. pin (string x)");

System.out.println("3. deposit (float d)");

System.out.println("4. withdraw (float w)");

System.out.println("5. balance ()");

System.out.println("6. login(string y)");

System.out.println("7. logout()");

System.out.println("8. lock(string x) ");

System.out.println("9. unlock(string x)");

Scanner scanner = new Scanner(System.in);

choice = scanner.nextInt();

if (choice <= 9) {

Scanner scanner2 = new Scanner(System.in);

switch (choice) {

case 1://open

System.out.println("Enter the value of p");

p = scanner2.nextLine();

System.out.println("Enter the value of y");

y = scanner2.nextLine();

System.out.println("Enter the value of a");

a = scanner2.nextFloat();

open(p, y, a);

break;

case 2://pin

System.out.println("Enter the value of x");

x = scanner2.nextLine();

pin(x);

break;

case 3://Deposit

System.out.println("Enter the value of d");

d = scanner2.nextFloat();

deposit(d);

break;

case 4://Withdraw

System.out.println("Enter the value of w");

w = scanner2.nextFloat();

withdraw(w);

break;

case 5://view balance

balance();

break;

case 6://login

System.out.println("Enter the value of y");

y = scanner2.nextLine();

login(y);

break;

case 7://logout

logout();

break;

case 8://Lock Account

System.out.println("Enter the value of x");

x = scanner2.nextLine();

lock(x);

break;

case 9://Unlock Account

System.out.println("Enter the value of x");

x = scanner2.nextLine();

unlock(x);

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

} else {

System.out.println("\nOperation not permitted in this state");

menu();

}

} while (choice <= 9);

}

public void unlock(String x) {

//if pin matches then unlocked otherwise stay in same state

if (x.equals(dataStore.getPin1())) {

mda.unlock();

if (dataStore.getBalance1() > 500) {

mda.unlockAboveMin();//to ready state

} else {

mda.unlockBelowMin();//to overdrawn state

}

} else {

mda.unlockIncorrectPin();//to Locked state

}

}

public void lock(String x) {

//lock an account if Pin matches else stay in same state

if (x.equals(dataStore.getPin1())) {

mda.lock();

} else {

mda.lockIncorrectPin();

}

}

//logout from account

public void logout() {

mda.logout();

}

//login to Account to perform required action

public void login(String y) {

//if user id matches then login to system

if (y.equals(dataStore.getUid1())) {

mda.login();

} else {

mda.loginIncorrect();//incorrect login

}

}

//show balance of account

public void balance() {

mda.balance();

}

//withdraw an amount from account

public void withdraw(float w) {

dataStore.setTemp\_w(w);

if (dataStore.getBalance1() > 500) {//if in ready state

mda.withdraw();

if (dataStore.getBalance1() > (dataStore.getTemp\_w1() + 500)) {//after withdraw balance is above 500

mda.withdrawAboveMin();

} else if ((dataStore.getBalance1() - dataStore.getTemp\_w1()) <= 500) {//after withdraw balance is below 500

mda.withdrawBelowMin();

}

}

else if (dataStore.getBalance1() <= 500) {//if balance is below 500 in overdrawn state

mda.withdrawBelowRequired();

}

}

//deposit an amount in account

public void deposit(float d) {

dataStore.setTemp\_d(d);

mda.deposit();

if (dataStore.getBalance1() > 500) {//deposit makes balance above 500, go to ready state

mda.depositAboveMin();

} else {

mda.depositBelowMin();//deposit makes balance below 500, go to Overdrawn

}

}

//pin operation on account

public void pin(String x) {

if (x.equals(dataStore.getPin1())) {//if pin matches

if (dataStore.getBalance1() > 500) {

mda.correctPinAboveMin();//ready

} else {

mda.correctPinBelowMin();//overdrawn

}

} else {

mda.incorrectPin(3);//if in correct pin, set the max number o attempts

}

}

//open operation on account

public void open(String p, String y, float a) {

dataStore.setTemp\_p(p);// storing data in temp. variables

dataStore.setTemp\_y(y);

dataStore.setTemp\_a(a);

mda.open();// calling function on mda

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* End of Input Processor Account 1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Start of Input Processor Account 2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.ssa.inputprocessor;

import java.util.Scanner;

import com.ssa.datastore.DataStore;

import com.ssa.datastore.DataStore\_Account2;

import com.ssa.factory.AbstractFactory;

import com.ssa.factory.AbstractFactory\_Account2;

import com.ssa.mda.MDA;

public class Account2 {

MDA mda;//pointer to mda

DataStore dataStore;//pointer to data store

int p, y, x, a, d, w;

public void initialize(MDA mda) {

AbstractFactory account2\_Concrete = new AbstractFactory\_Account2();//getting abstract factory for account 2

account2\_Concrete.initialize(account2\_Concrete);

this.mda = mda;

dataStore = new DataStore\_Account2();//initializing datastore

menu();

}

public void print() {

System.out.println("This is Account2");

}

//once started, user will given a list of options to choose from menu

public void menu() {

int choice;

do {

System.out.println("\n");

System.out.println("Select the Operation you would like to perform in Account-1");

//operations supported by Account 2

System.out.println("1. OPEN (int p, int y, int a)");

System.out.println("2. PIN (int x)");

System.out.println("3. DEPOSIT (int d)");

System.out.println("4. WITHDRAW (int w)");

System.out.println("5. BALANCE ()");

System.out.println("6. LOGIN(int y)");

System.out.println("7. LOGOUT()");

System.out.println("8. suspend()");

System.out.println("9. activate()");

System.out.println("10. close()");

Scanner scanner = new Scanner(System.in);

choice = scanner.nextInt();

if (choice <= 10) {

switch (choice) {

case 1://open

System.out.println("Enter the value of p");

p = scanner.nextInt();

System.out.println("Enter the value of y");

y = scanner.nextInt();

System.out.println("Enter the value of a");

a = scanner.nextInt();

open(p, y, a);

break;

case 2://pin

System.out.println("Enter the value of x");

x = scanner.nextInt();

pin(x);

break;

case 3://deposit

System.out.println("Enter the value of d");

d = scanner.nextInt();

deposit(d);

break;

case 4://withdraw

System.out.println("Enter the value of w");

w = scanner.nextInt();

withdraw(w);

break;

case 5://show balance

balance();

break;

case 6://login

System.out.println("Enter the value of y");

y = scanner.nextInt();

login(y);

break;

case 7://logout

logout();

break;

case 8://suspend

suspend();

break;

case 9://activate

activate();

break;

case 10://close

close();

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

} else {

System.out.println("Operation not permitted in this state");

menu();

}

} while (choice <= 10);

}

//close the account system

public void close() {

mda.close();

}

//activate an account system from suspend state

public void activate() {

mda.activate();

}

//suspend an account system

public void suspend() {

mda.suspend();

}

//logout from account system

public void logout() {

mda.logout();

}

//login into account system

public void login(int y) {

if (y == dataStore.getUid2()) {//if user is matches

mda.login();

} else {

mda.loginIncorrect();

}

}

//show balance

public void balance() {

mda.balance();

}

//withdraw an amount

public void withdraw(int w) {

dataStore.setTemp\_w(w);

if ((dataStore.getBalance2()) > 0) {//if balance is above 0

mda.withdraw();

mda.withdrawAboveMin();

} else {

mda.withdrawBelowRequired();//if balance is below 0, No funds

}

}

//deposit an amount into account

public void deposit(int d) {

dataStore.setTemp\_d(d);

mda.deposit();

}

//pin operation in account

public void pin(int x) {

if (x == dataStore.getPin2()) {//if pin matches

mda.correctPinAboveMin();

} else {

mda.incorrectPin(2);

}

}

//open operation on account 2

public void open(int p, int y, int a) {

dataStore.setTemp\_p(p);

dataStore.setTemp\_y(y);

dataStore.setTemp\_a(a);

mda.open();

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* End of Input Processor Account 2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MDA Start Includes State pattern \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.ssa.mda;

import com.ssa.states.CheckPin;

import com.ssa.states.Idle;

import com.ssa.states.Locked;

import com.ssa.states.Overdrawn;

import com.ssa.states.Ready;

import com.ssa.states.S1;

import com.ssa.states.S2;

import com.ssa.states.S3;

import com.ssa.states.Start;

import com.ssa.states.State;

import com.ssa.states.Suspend;

public class MDA {

public static int max = 0;// stores number of attempts of incorrect pin

private final State stateList[] = { new Start(), new Idle(), new CheckPin(), new Ready(), new Overdrawn(),

new Locked(), new Suspend(), new S1(), new S2(), new S3()};//state list for account system

public State s = stateList[0];//maintains current state of account system

//open operation on account

public void open() {

s.open();

switch (s.getStateId()) {

case 0:

s = stateList[1];

System.out.println("State changed to Idle. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//login operation on account

public void login() {

s.login();

switch (s.getStateId()) {

case 1:

s = stateList[2];

System.out.println("State changed to CheckPin. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//if login in correct

public void loginIncorrect() {

s.loginIncorrect();

switch (s.getStateId()) {

case 1:

s = stateList[1];

System.out.println("State changed to Idle. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//if user enterns wrong pin

public void incorrectPin(int max) {

s.incorrectPin(max);

switch (s.getStateId()) {

case 2:

if (MDA.max == max) {//if max attempts reached

s = stateList[1];

System.out.println("State changed to Idle. ");

} else if (MDA.max < max) {

s = stateList[2];

System.out.println("State changed to CheckPin. ");

}

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//if pin is correct and balance is above required value

public void correctPinAboveMin() {

s.correctPinAboveMin();

switch (s.getStateId()) {

case 2:

s = stateList[3];

System.out.println("State changed to Ready. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//if pin is correct and balance is below required value

public void correctPinBelowMin() {

s.correctPinBelowMin();

switch (s.getStateId()) {

case 2:

s = stateList[4];

System.out.println("State changed to Overdrawn. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//logout from account system

public void logout() {

s.logout();

switch (s.getStateId()) {

case 2:

s = stateList[1];

System.out.println("State changed to Idle. ");

break;

case 3:

s = stateList[1];

System.out.println("State changed to Idle. ");

break;

case 4:

s = stateList[1];

System.out.println("State changed to Idle. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//deposit an amount into account

public void deposit() {

s.deposit();

switch (s.getStateId()) {

case 3:

s = stateList[3];

System.out.println("State changed to Ready. ");

break;

case 4:

s = stateList[7];

System.out.println("State changed to S1 temporary state. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//deposit an amount and that brings account balance to above required value

public void depositAboveMin() {

s.depositAboveMin();

switch (s.getStateId()) {

case 3:

s = stateList[3];

break;

case 7:

s = stateList[3];

System.out.println("State changed to Ready. ");

break;

default:

System.out.println("Operation not permitted in this state.");

break;

}

}

//deposit an amount and that brings account balance to below required value

public void depositBelowMin() {

s.depositBelowMin();

switch (s.getStateId()) {

case 7:

s = stateList[4];

System.out.println("State changed to Overdrawn. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//show balance

public void balance() {

s.balance();

switch (s.getStateId()) {

case 3:

s = stateList[3];

System.out.println("State changed to Ready. ");

break;

case 4:

s = stateList[4];

System.out.println("State changed to Overdrawn. ");

break;

case 6:

s = stateList[6];

System.out.println("State changed to Suspend. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//withdraw an amount from account

public void withdraw() {

s.withdraw();

switch (s.getStateId()) {

case 3:

s = stateList[8];

System.out.println("State changed to S2. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//withdraw an amount from account and balance is above required value

public void withdrawAboveMin() {

s.withdrawAboveMin();

switch (s.getStateId()) {

case 8:

s = stateList[3];

System.out.println("State changed to Ready. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//withdraw an amount from account and balance is below required value

public void withdrawBelowMin() {

s.withdrawBelowMin();

switch (s.getStateId()) {

case 8:

s = stateList[4];

System.out.println("State changed to Overdrawn. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

////withdraw an amount from account and balance is below required value

public void withdrawBelowRequired() {

s.withdrawBelowRequired();

switch (s.getStateId()) {

case 3:

s = stateList[3];

System.out.println("State changed to Ready. ");

break;

case 4:

s = stateList[4];

System.out.println("State changed to Overdrawn. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//lock an account

public void lock() {

s.lock();

switch (s.getStateId()) {

case 3:

s = stateList[5];

System.out.println("State changed to Locked. ");

break;

case 4:

s = stateList[5];

System.out.println("State changed to Locked. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//if incorrect pin is entered

public void lockIncorrectPin() {

s.lockIncorrectPin();

switch (s.getStateId()) {

case 3:

s = stateList[3];

System.out.println("State changed to Ready. ");

break;

case 4:

s = stateList[4];

System.out.println("State changed to Overdrawn. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//incorrect pin entered while unlocking

public void unlockIncorrectPin() {

s.unlockIncorrectPin();

switch (s.getStateId()) {

case 5:

s = stateList[5];

System.out.println("State changed to Locked. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//unlock an account

public void unlock() {

s.unlock();

switch (s.getStateId()) {

case 5:

s = stateList[9];

System.out.println("State changed to S3. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//unlock an account and balance is above required value

public void unlockAboveMin() {

s.unlockAboveMin();

switch (s.getStateId()) {

case 9:

s = stateList[3];

System.out.println("State changed to Ready. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//unlock an account and balance is below required value

public void unlockBelowMin() {

s.unlockBelowMin();

switch (s.getStateId()) {

case 9:

s = stateList[4];

System.out.println("State changed to Overdrawn. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//suspend an account

public void suspend() {

s.suspend();

switch (s.getStateId()) {

case 3:

s = stateList[6];

System.out.println("State changed to Suspend. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//activate a suspend account

public void activate() {

s.activate();

switch (s.getStateId()) {

case 6:

s = stateList[3];

System.out.println("State changed to Ready. ");

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//close an account system

public void close() {

s.close();

switch (s.getStateId()) {

case 6:

System.out.println("Account System is Closed, Now you can't perform any action. Thank you");

System.exit(0);

break;

default:

System.out.println("Operation not permitted in this state");

break;

}

}

//returns number of incorrect attempts while entering pin

public static int getMax() {

return max;

}

//set number of incorrect attempts while entering pin

public static void setMax(int max) {

MDA.max = max;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MDA End \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Start of Data Store \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.ssa.datastore;

//an abstract class of datastore

public abstract class DataStore {

// --------------- For Account 1 start-------------------------

public void setTemp\_p(String p) {

}

public String getTemp\_p1() {

return null;

}

public void setTemp\_y(String y) {

}

public String getTemp\_y1() {

return null;

}

public void setTemp\_a(float a) {

}

public float getTemp\_a1() {

return 0;

}

public void setPin(String temp\_p) {

}

public String getPin1() {

return null;

}

public void setUid(String temp\_y) {

}

public String getUid1() {

return null;

}

public void setBalance(float temp\_a) {

}

public float getBalance1() {

return 0;

}

public void setTemp\_d(float d) {

}

public float getTemp\_d1() {

return 0;

}

public void setTemp\_w(float w) {

}

public float getTemp\_w1() {

return 0;

}

public void setD(float temp\_d) {

}

public float getD1() {

return 0;

}

public void setW(float temp\_w) {

}

public float getW1() {

return 0;

}

// --------------- For Account 1 End-------------------------

// -----------For Account 2 start --------------------

public void setTemp\_p(int p) {

}

public int getTemp\_p2() {

return 0;

}

public void setTemp\_y(int y) {

}

public int getTemp\_y2() {

return 0;

}

public void setTemp\_a(int a) {

}

public int getTemp\_a2() {

return 0;

}

public void setPin(int temp\_p) {

}

public int getPin2() {

return 0;

}

public void setUid(int temp\_y) {

}

public int getUid2() {

return 0;

}

public void setBalance(int temp\_a) {

}

public int getBalance2() {

return 0;

}

public void setTemp\_d(int d) {

}

public int getTemp\_d2() {

return 0;

}

public void setTemp\_w(int w) {

}

public int getTemp\_w2() {

return 0;

}

public void setD(int temp\_d) {

}

public int getD2() {

return 0;

}

public void setW(int temp\_w) {

}

public int getW2() {

return 0;

}

// --------------- For Account 2 End-------------------------

}

//\*\*\*\*\*\*\*\*\*\*\*Data Store for Account 1 Start\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**package** com.ssa.datastore;

//DataStore for Account1, this will helps to store data for Account1

**public** **class** DataStore\_Account1 **extends** DataStore {

**public** **static** String *temp\_p*;//pin temp.

**public** **static** String *temp\_y*;//user id temp

**public** **static** **float** *temp\_a*;//account balance temp

**public** **static** String *temp\_x*;//pin temp

**public** **static** String *pin*;//pin

**public** **static** String *uid*;//userid

**public** **static** **float** *balance*;//balance

**public** **static** **float** *temp\_d*;//depeosit amount temp

**public** **static** **float** *temp\_w*;//withdraw amount temp

**public** **static** **float** *d*;//deposit amount

**public** **static** **float** *w*;//withdraw amount

@Override

**public** **void** setTemp\_p(String p) {

*temp\_p* = p;

}

@Override

**public** String getTemp\_p1() {

**return** *temp\_p*;

}

@Override

**public** **void** setTemp\_y(String y) {

*temp\_y*=y;

}

@Override

**public** String getTemp\_y1() {

**return** *temp\_y*;

}

@Override

**public** **void** setTemp\_a(**float** a) {

*temp\_a*=a;

}

@Override

**public** **float** getTemp\_a1() {

**return** *temp\_a*;

}

@Override

**public** **void** setPin(String temp\_p) {

*pin*=temp\_p;

}

@Override

**public** String getPin1() {

**return** *pin*;

}

@Override

**public** **void** setUid(String temp\_y) {

*uid*=temp\_y;

}

@Override

**public** String getUid1() {

**return** *uid*;

}

@Override

**public** **void** setBalance(**float** temp\_a) {

*balance*=temp\_a;

}

@Override

**public** **float** getBalance1() {

**return** *balance*;

}

@Override

**public** **void** setTemp\_d(**float** d) {

*temp\_d*=d;

}

@Override

**public** **float** getTemp\_d1() {

**return** *temp\_d*;

}

@Override

**public** **void** setTemp\_w(**float** w) {

*temp\_w*=w;

}

@Override

**public** **float** getTemp\_w1() {

**return** *temp\_w*;

}

@Override

**public** **void** setD(**float** temp\_d) {

*d*=temp\_d;

}

@Override

**public** **float** getD1() {

**return** *d*;

}

@Override

**public** **void** setW(**float** temp\_w) {

*w*=temp\_w;

}

@Override

**public** **float** getW1() {

**return** *w*;

}

}

//\*\*\*\*\*\*\*\*\*\*\*Data Store for Account 1 End \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*Data Store for Account 2 Start\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**package** com.ssa.datastore;

//DataStore for Account2, this will helps to store data for Account2

**public** **class** DataStore\_Account2 **extends** DataStore {

**public** **static** **int** *temp\_p*;//pin temp

**public** **static** **int** *temp\_y*;//user id temp

**public** **static** **int** *temp\_a*;//balance temp

**public** **static** **int** *pin*;//pin

**public** **static** **int** *uid*;//user id

**public** **static** **int** *balance*;//balance

**public** **static** **int** *temp\_d*;//deposit temp

**public** **static** **int** *temp\_w*;//withdraw amount temp

**public** **static** **int** *d*;//deposit amount

**public** **static** **int** *w*;//withdraw amount

@Override

**public** **void** setTemp\_p(**int** p) {

*temp\_p* = p;

}

@Override

**public** **int** getTemp\_p2() {

**return** *temp\_p*;

}

@Override

**public** **void** setTemp\_y(**int** y) {

*temp\_y* = y;

}

@Override

**public** **int** getTemp\_y2() {

**return** *temp\_y*;

}

@Override

**public** **void** setTemp\_a(**int** y) {

*temp\_a* = y;

}

@Override

**public** **int** getTemp\_a2() {

**return** *temp\_a*;

}

@Override

**public** **void** setPin(**int** temp\_p) {

*pin* = temp\_p;

}

@Override

**public** **int** getPin2() {

**return** *pin*;

}

@Override

**public** **void** setUid(**int** temp\_y) {

*uid* = temp\_y;

}

@Override

**public** **int** getUid2() {

**return** *uid*;

}

@Override

**public** **void** setBalance(**int** temp\_a) {

*balance* = temp\_a;

}

@Override

**public** **int** getBalance2() {

**return** *balance*;

}

@Override

**public** **void** setTemp\_d(**int** d) {

*temp\_d* = d;

}

@Override

**public** **int** getTemp\_d2() {

**return** *temp\_d*;

}

@Override

**public** **void** setTemp\_w(**int** w) {

*temp\_w* = w;

}

@Override

**public** **int** getTemp\_w2() {

**return** *temp\_w*;

}

@Override

**public** **void** setD(**int** temp\_d) {

*d* = temp\_d;

}

@Override

**public** **int** getD2() {

**return** *d*;

}

@Override

**public** **void** setW(**int** temp\_w) {

*w* = temp\_w;

}

@Override

**public** **int** getW2() {

**return** *w*;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*Data Store for Account 2 End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End of Data Store \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Start of State Pattern \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.ssa.states;

import com.ssa.mda.MDA;

import com.ssa.outputprocessor.OutputProcessor;

//an abstract class defined for all the states of account system

public abstract class State {

//MDA

public MDA create(){

return new MDA();

}

public OutputProcessor outputProcessor;

//output processor

public OutputProcessor createOutputProcessor(){

return new OutputProcessor();

}

public abstract void open();

public abstract void login();

public abstract void loginIncorrect();

public abstract void incorrectPin(int max);

public abstract void correctPinAboveMin();

public abstract void correctPinBelowMin();

public abstract void logout();

public abstract void deposit();

public abstract void depositAboveMin();

public abstract void depositBelowMin();

public abstract void balance();

public abstract void withdraw();

public abstract void withdrawAboveMin();

public abstract void withdrawBelowMin();

public abstract void withdrawBelowRequired();

public abstract void lock();

public abstract void lockIncorrectPin();

public abstract void unlockIncorrectPin();

public abstract void unlock();

public abstract void unlockAboveMin();

public abstract void unlockBelowMin();

public abstract void suspend();

public abstract void activate();

public abstract void close();

public abstract int getStateId();

}

**package** com.ssa.states;

**import** com.ssa.outputprocessor.OutputProcessor;

//start state

**public** **class** Start **extends** State {

@Override

**public** **void** open() {

OutputProcessor outputProcessor=createOutputProcessor();

outputProcessor.storeData();

}

@Override

**public** **void** login() {

//Do Nothing

}

@Override

**public** **void** loginIncorrect() {

//Do Nothing

}

@Override

**public** **void** incorrectPin(**int** max) {

//Do Nothing

}

@Override

**public** **void** correctPinAboveMin() {

//Do Nothing

}

@Override

**public** **void** correctPinBelowMin() {

//Do Nothing

}

@Override

**public** **void** logout() {

//Do Nothing

}

@Override

**public** **void** deposit() {

//Do Nothing

}

@Override

**public** **void** depositAboveMin() {

//Do Nothing

}

@Override

**public** **void** depositBelowMin() {

//Do Nothing

}

@Override

**public** **void** balance() {

//Do Nothing

}

@Override

**public** **void** withdraw() {

//Do Nothing

}

@Override

**public** **void** withdrawAboveMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowRequired() {

//Do Nothing

}

@Override

**public** **void** lock() {

//Do Nothing

}

@Override

**public** **void** lockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlock() {

//Do Nothing

}

@Override

**public** **void** unlockAboveMin() {

//Do Nothing

}

@Override

**public** **void** unlockBelowMin() {

//Do Nothing

}

@Override

**public** **void** suspend() {

//Do Nothing

}

@Override

**public** **void** activate() {

//Do Nothing

}

@Override

**public** **void** close() {

//Do Nothing

}

@Override

**public** **int** getStateId() {

**return** 0;

}

}

**package** com.ssa.states;

**import** com.ssa.outputprocessor.OutputProcessor;

//idle state

**public** **class** Idle **extends** State {

@Override

**public** **void** open() {

//DO Nothing

}

//does login into the system and asks for pin along with sets max attempts to 0

@Override

**public** **void** login() {

OutputProcessor outputProcessor = createOutputProcessor();

outputProcessor.promptForPin();

create().*setMax*(0);

}

//if incorrect ID then calls an incorrectID action on output processor

@Override

**public** **void** loginIncorrect() {

OutputProcessor outputProcessor = createOutputProcessor();

outputProcessor.incorrectIDMsg();

}

@Override

**public** **void** incorrectPin(**int** max) {

//DO Nothing

}

@Override

**public** **void** correctPinAboveMin() {

//DO Nothing

}

@Override

**public** **void** correctPinBelowMin() {

//DO Nothing

}

@Override

**public** **void** logout() {

//DO Nothing

}

@Override

**public** **void** deposit() {

//DO Nothing

}

@Override

**public** **void** depositAboveMin() {

//DO Nothing

}

@Override

**public** **void** depositBelowMin() {

//DO Nothing

}

@Override

**public** **void** balance() {

//DO Nothing

}

@Override

**public** **void** withdraw() {

//DO Nothing

}

@Override

**public** **void** withdrawAboveMin() {

//DO Nothing

}

@Override

**public** **void** withdrawBelowMin() {

//DO Nothing

}

@Override

**public** **void** withdrawBelowRequired() {

//DO Nothing

}

@Override

**public** **void** lock() {

//DO Nothing

}

@Override

**public** **void** lockIncorrectPin() {

//DO Nothing

}

@Override

**public** **void** unlockIncorrectPin() {

//DO Nothing

}

@Override

**public** **void** unlock() {

//DO Nothing

}

@Override

**public** **void** unlockAboveMin() {

//DO Nothing

}

@Override

**public** **void** unlockBelowMin() {

//DO Nothing

}

@Override

**public** **void** suspend() {

//DO Nothing

}

@Override

**public** **void** activate() {

//DO Nothing

}

@Override

**public** **void** close() {

//DO Nothing

}

@Override

**public** **int** getStateId() {

//DO Nothing

**return** 1;

}

}

**package** com.ssa.states;

**import** com.ssa.outputprocessor.OutputProcessor;

//check pin state

**public** **class** CheckPin **extends** State {

@Override

**public** **void** open() {

//Do Nothing

}

@Override

**public** **void** login() {

//Do Nothing

}

@Override

**public** **void** loginIncorrect() {

//Do Nothing

}

//calls an action on output processor when user enters incorrect pin also increments number of attempts in MDA.

@Override

**public** **void** incorrectPin(**int** max) {

OutputProcessor outputProcessor = createOutputProcessor();

outputProcessor.incorrectPinMsg();

**if** (max == create().*getMax*()) {

outputProcessor.tooManyAttemptsMsg();

create().*setMax*(create().*getMax*() + 1);

} **else** {

create().*setMax*(create().*getMax*() + 1);

}

}

//when pin is correct and balance is above required value then calls an action on output processor

@Override

**public** **void** correctPinAboveMin() {

createOutputProcessor().displayMenu();

}

//when pin is correct and balance is below required value then calls an action on output processor

@Override

**public** **void** correctPinBelowMin() {

createOutputProcessor().displayMenu();

}

@Override

**public** **void** logout() {

}

@Override

**public** **void** deposit() {

//Do Nothing

}

@Override

**public** **void** depositAboveMin() {

//Do Nothing

}

@Override

**public** **void** depositBelowMin() {

//Do Nothing

}

@Override

**public** **void** balance() {

//Do Nothing

}

@Override

**public** **void** withdraw() {

//Do Nothing

}

@Override

**public** **void** withdrawAboveMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowRequired() {

//Do Nothing

}

@Override

**public** **void** lock() {

//Do Nothing

}

@Override

**public** **void** lockIncorrectPin() {

}

@Override

**public** **void** unlockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlock() {

//Do Nothing

}

@Override

**public** **void** unlockAboveMin() {

//Do Nothing

}

@Override

**public** **void** unlockBelowMin() {

//Do Nothing

}

@Override

**public** **void** suspend() {

//Do Nothing

}

@Override

**public** **void** activate() {

//Do Nothing

}

@Override

**public** **void** close() {

//Do Nothing

}

@Override

**public** **int** getStateId() {

//Do Nothing

**return** 2;

}

}

**package** com.ssa.states;

//ready state

**public** **class** Ready **extends** State {

@Override

**public** **void** open() {

//Do Nothing

}

@Override

**public** **void** login() {

//Do Nothing

}

@Override

**public** **void** loginIncorrect() {

//Do Nothing

}

@Override

**public** **void** incorrectPin(**int** max) {

//Do Nothing

}

@Override

**public** **void** correctPinAboveMin() {

//Do Nothing

}

@Override

**public** **void** correctPinBelowMin() {

//Do Nothing

}

@Override

**public** **void** logout() {

//Do Nothing

}

//calls an action on output processor to deposit an amount

@Override

**public** **void** deposit() {

createOutputProcessor().makeDeposit();

}

@Override

**public** **void** depositAboveMin() {

//Do Nothing

}

@Override

**public** **void** depositBelowMin() {

//Do Nothing

}

//calls an action on output processor to show balance

@Override

**public** **void** balance() {

createOutputProcessor().displayBalance();

}

//calls an action on output processor to withdraw from account

@Override

**public** **void** withdraw() {

createOutputProcessor().makeWithdraw();

}

@Override

**public** **void** withdrawAboveMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowMin() {

//Do Nothing

}

//calls an action on output processor when balance is below the required min. value

@Override

**public** **void** withdrawBelowRequired() {

createOutputProcessor().displayNoFundsMsg();

}

@Override

**public** **void** lock() {

//Do Nothing

}

//calls an action on output processor when user enters incorrect pin while lock operation

@Override

**public** **void** lockIncorrectPin() {

createOutputProcessor().incorrectPinMsg();

}

@Override

**public** **void** unlockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlock() {

//Do Nothing

}

@Override

**public** **void** unlockAboveMin() {

//Do Nothing

}

@Override

**public** **void** unlockBelowMin() {

//Do Nothing

}

@Override

**public** **void** suspend() {

//Do Nothing

}

@Override

**public** **void** activate() {

//Do Nothing

}

@Override

**public** **void** close() {

//Do Nothing

}

@Override

**public** **int** getStateId() {

//Do Nothing

**return** 3;

}

}

**package** com.ssa.states;

//over drawn state

**public** **class** Overdrawn **extends** State {

@Override

**public** **void** open() {

//Do Nothing

}

@Override

**public** **void** login() {

//Do Nothing

}

@Override

**public** **void** loginIncorrect() {

//Do Nothing

}

@Override

**public** **void** incorrectPin(**int** max) {

//Do Nothing

}

@Override

**public** **void** correctPinAboveMin() {

//Do Nothing

}

@Override

**public** **void** correctPinBelowMin() {

//Do Nothing

}

@Override

**public** **void** logout() {

//Do Nothing

}

//deposit an amount

@Override

**public** **void** deposit() {

createOutputProcessor().makeDeposit();

}

@Override

**public** **void** depositAboveMin() {

//Do Nothing

}

@Override

**public** **void** depositBelowMin() {

//Do Nothing

}

//calls an action on output processor to show balance

@Override

**public** **void** balance() {

createOutputProcessor().displayBalance();

}

@Override

**public** **void** withdraw() {

//Do Nothing

}

@Override

**public** **void** withdrawAboveMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowMin() {

//Do Nothing

}

//calls an action on output processor when balance is below the required min. value

@Override

**public** **void** withdrawBelowRequired() {

createOutputProcessor().withdrawBelowMinMsg();

}

@Override

**public** **void** lock() {

//Do Nothing

}

//calls an action on output processor when user enters incorrect pin while lock operation

@Override

**public** **void** lockIncorrectPin() {

createOutputProcessor().incorrectPinMsg();

}

@Override

**public** **void** unlockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlock() {

//Do Nothing

}

@Override

**public** **void** unlockAboveMin() {

//Do Nothing

}

@Override

**public** **void** unlockBelowMin() {

//Do Nothing

}

@Override

**public** **void** suspend() {

//Do Nothing

}

@Override

**public** **void** activate() {

//Do Nothing

}

@Override

**public** **void** close() {

//Do Nothing

}

@Override

**public** **int** getStateId() {

//Do Nothing

**return** 4;

}

}

**package** com.ssa.states;

//Locked state

**public** **class** Locked **extends** State {

@Override

**public** **void** open() {

//DO Nothing

}

@Override

**public** **void** login() {

//DO Nothing

}

@Override

**public** **void** loginIncorrect() {

//DO Nothing

}

@Override

**public** **void** incorrectPin(**int** max) {

//DO Nothing

}

@Override

**public** **void** correctPinAboveMin() {

//DO Nothing

}

@Override

**public** **void** correctPinBelowMin() {

//DO Nothing

}

@Override

**public** **void** logout() {

//DO Nothing

}

@Override

**public** **void** deposit() {

//DO Nothing

}

@Override

**public** **void** depositAboveMin() {

//DO Nothing

}

@Override

**public** **void** depositBelowMin() {

//DO Nothing

}

@Override

**public** **void** balance() {

//DO Nothing

}

@Override

**public** **void** withdraw() {

//DO Nothing

}

@Override

**public** **void** withdrawAboveMin() {

//DO Nothing

}

@Override

**public** **void** withdrawBelowMin() {

//DO Nothing

}

@Override

**public** **void** withdrawBelowRequired() {

//DO Nothing

}

@Override

**public** **void** lock() {

//DO Nothing

}

@Override

**public** **void** lockIncorrectPin() {

//DO Nothing

}

//calls an operation on output processor when user enters incorrect pin while unlocking

@Override

**public** **void** unlockIncorrectPin() {

createOutputProcessor().incorrectPinMsg();

}

@Override

**public** **void** unlock() {

//DO Nothing

}

@Override

**public** **void** unlockAboveMin() {

//DO Nothing

}

@Override

**public** **void** unlockBelowMin() {

//DO Nothing

}

@Override

**public** **void** suspend() {

//DO Nothing

}

@Override

**public** **void** activate() {

//DO Nothing

}

@Override

**public** **void** close() {

//DO Nothing

}

@Override

**public** **int** getStateId() {

//DO Nothing

**return** 5;

}

}

**package** com.ssa.states;

//suspend state

**public** **class** Suspend **extends** State {

@Override

**public** **void** open() {

//Do Nothing

}

@Override

**public** **void** login() {

//Do Nothing

}

@Override

**public** **void** loginIncorrect() {

//Do Nothing

}

@Override

**public** **void** incorrectPin(**int** max) {

//Do Nothing

}

@Override

**public** **void** correctPinAboveMin() {

//Do Nothing

}

@Override

**public** **void** correctPinBelowMin() {

//Do Nothing

}

@Override

**public** **void** logout() {

//Do Nothing

}

@Override

**public** **void** deposit() {

//Do Nothing

}

@Override

**public** **void** depositAboveMin() {

//Do Nothing

}

@Override

**public** **void** depositBelowMin() {

//Do Nothing

}

//calls an action on output processor to show balance

@Override

**public** **void** balance() {

createOutputProcessor().displayBalance();

}

@Override

**public** **void** withdraw() {

//Do Nothing

}

@Override

**public** **void** withdrawAboveMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowRequired() {

//Do Nothing

}

@Override

**public** **void** lock() {

//Do Nothing

}

@Override

**public** **void** lockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlock() {

//Do Nothing

}

@Override

**public** **void** unlockAboveMin() {

//Do Nothing

}

@Override

**public** **void** unlockBelowMin() {

//Do Nothing

}

@Override

**public** **void** suspend() {

//Do Nothing

}

@Override

**public** **void** activate() {

//Do Nothing

}

@Override

**public** **void** close() {

//Do Nothing

}

@Override

**public** **int** getStateId() {

//Do Nothing

**return** 6;

}

}

**package** com.ssa.states;

//S1 Temp. state

**public** **class** S1 **extends** State {

@Override

**public** **void** open() {

//Do Nothing

}

@Override

**public** **void** login() {

//Do Nothing

}

@Override

**public** **void** loginIncorrect() {

//Do Nothing

}

@Override

**public** **void** incorrectPin(**int** max) {

//Do Nothing

}

@Override

**public** **void** correctPinAboveMin() {

//Do Nothing

}

@Override

**public** **void** correctPinBelowMin() {

//Do Nothing

}

@Override

**public** **void** logout() {

//Do Nothing

}

@Override

**public** **void** deposit() {

//Do Nothing

}

@Override

**public** **void** depositAboveMin() {

}

@Override

**public** **void** depositBelowMin() {

//Do Nothing

}

@Override

**public** **void** balance() {

//Do Nothing

}

@Override

**public** **void** withdraw() {

//Do Nothing

}

@Override

**public** **void** withdrawAboveMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowRequired() {

//Do Nothing

}

@Override

**public** **void** lock() {

//Do Nothing

}

@Override

**public** **void** lockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlock() {

//Do Nothing

}

@Override

**public** **void** unlockAboveMin() {

//Do Nothing

}

@Override

**public** **void** unlockBelowMin() {

//Do Nothing

}

@Override

**public** **void** suspend() {

//Do Nothing

}

@Override

**public** **void** activate() {

//Do Nothing

}

@Override

**public** **void** close() {

//Do Nothing

}

@Override

**public** **int** getStateId() {

//Do Nothing

**return** 7;

}

}

**package** com.ssa.states;

//S2 temp state

**public** **class** S2 **extends** State {

@Override

**public** **void** open() {

//Do Nothing

}

@Override

**public** **void** login() {

//Do Nothing

}

@Override

**public** **void** loginIncorrect() {

//Do Nothing

}

@Override

**public** **void** incorrectPin(**int** max) {

//Do Nothing

}

@Override

**public** **void** correctPinAboveMin() {

//Do Nothing

}

@Override

**public** **void** correctPinBelowMin() {

//Do Nothing

}

@Override

**public** **void** logout() {

//Do Nothing

}

@Override

**public** **void** deposit() {

//Do Nothing

}

@Override

**public** **void** depositAboveMin() {

//Do Nothing

}

@Override

**public** **void** depositBelowMin() {

//Do Nothing

}

@Override

**public** **void** balance() {

//Do Nothing

}

@Override

**public** **void** withdraw() {

//Do Nothing

}

@Override

**public** **void** withdrawAboveMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowMin() {

createOutputProcessor().penalty();

}

@Override

**public** **void** withdrawBelowRequired() {

}

@Override

**public** **void** lock() {

//Do Nothing

}

@Override

**public** **void** lockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlock() {

//Do Nothing

}

@Override

**public** **void** unlockAboveMin() {

//Do Nothing

}

@Override

**public** **void** unlockBelowMin() {

//Do Nothing

}

@Override

**public** **void** suspend() {

//Do Nothing

}

@Override

**public** **void** activate() {

//Do Nothing

}

@Override

**public** **void** close() {

//Do Nothing

}

@Override

**public** **int** getStateId() {

//Do Nothing

**return** 8;

}

}

**package** com.ssa.states;

//S3 temp state

**public** **class** S3 **extends** State {

@Override

**public** **void** open() {

//Do Nothing

}

@Override

**public** **void** login() {

//Do Nothing

}

@Override

**public** **void** loginIncorrect() {

//Do Nothing

}

@Override

**public** **void** incorrectPin(**int** max) {

//Do Nothing

}

@Override

**public** **void** correctPinAboveMin() {

//Do Nothing

}

@Override

**public** **void** correctPinBelowMin() {

//Do Nothing

}

@Override

**public** **void** logout() {

//Do Nothing

}

@Override

**public** **void** deposit() {

//Do Nothing

}

@Override

**public** **void** depositAboveMin() {

//Do Nothing

}

@Override

**public** **void** depositBelowMin() {

//Do Nothing

}

@Override

**public** **void** balance() {

//Do Nothing

}

@Override

**public** **void** withdraw() {

//Do Nothing

}

@Override

**public** **void** withdrawAboveMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowMin() {

//Do Nothing

}

@Override

**public** **void** withdrawBelowRequired() {

//Do Nothing

}

@Override

**public** **void** lock() {

//Do Nothing

}

@Override

**public** **void** lockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlockIncorrectPin() {

//Do Nothing

}

@Override

**public** **void** unlock() {

//Do Nothing

}

@Override

**public** **void** unlockAboveMin() {

//Do Nothing

}

@Override

**public** **void** unlockBelowMin() {

//Do Nothing

}

@Override

**public** **void** suspend() {

//Do Nothing

}

@Override

**public** **void** activate() {

//Do Nothing

}

@Override

**public** **void** close() {

//Do Nothing

}

@Override

**public** **int** getStateId() {

//Do Nothing

**return** 9;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End of State Pattern \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Start of Abstract Factory Pattern\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.ssa.factory;

import com.ssa.datastore.DataStore;

import com.ssa.strategy.DisplayBalance;

import com.ssa.strategy.DisplayMenu;

import com.ssa.strategy.IncorrectID;

import com.ssa.strategy.IncorrectPin;

import com.ssa.strategy.MakeDeposit;

import com.ssa.strategy.MakeWithdraw;

import com.ssa.strategy.NoFunds;

import com.ssa.strategy.Penalty;

import com.ssa.strategy.PromptForPin;

import com.ssa.strategy.StoreData;

import com.ssa.strategy.TooManyAttempts;

import com.ssa.strategy.WithdrawBelowMin;

//abstract class defined for an abstract factory implementation which will help to invoke an object to class

public abstract class AbstractFactory {

public abstract DataStore getdatastore();

public abstract void initialize(AbstractFactory abstractFactory);

public abstract StoreData getstoreData();

public abstract IncorrectID getincorrectIDMsg();

public abstract IncorrectPin getincorrectPinMsg();

public abstract TooManyAttempts gettooManyAttemptsMsg();

public abstract DisplayMenu getdisplayMenu();

public abstract MakeDeposit getmakeDeposit();

public abstract DisplayBalance getdisplayBalance();

public abstract MakeWithdraw getmakeWithdraw();

public abstract WithdrawBelowMin getwithdrawBelowMinMsg();

public abstract NoFunds getdisplayNoFundsMsg();

public abstract PromptForPin getpromptForPin();

public abstract Penalty getPenalty();

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*Start of Abstract Factory for Account 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.ssa.factory;

import com.ssa.datastore.DataStore;

import com.ssa.datastore.DataStore\_Account1;

import com.ssa.outputprocessor.OutputProcessor;

import com.ssa.strategy.DisplayBalance;

import com.ssa.strategy.DisplayBalance1;

import com.ssa.strategy.DisplayMenu;

import com.ssa.strategy.DisplayMenu1;

import com.ssa.strategy.IncorrectID;

import com.ssa.strategy.IncorrectID1;

import com.ssa.strategy.IncorrectPin;

import com.ssa.strategy.IncorrectPin1;

import com.ssa.strategy.MakeDeposit;

import com.ssa.strategy.MakeDeposit1;

import com.ssa.strategy.MakeWithdraw;

import com.ssa.strategy.MakeWithdraw1;

import com.ssa.strategy.NoFunds;

import com.ssa.strategy.Penalty;

import com.ssa.strategy.Penalty1;

import com.ssa.strategy.PromptForPin;

import com.ssa.strategy.PromptForPin1;

import com.ssa.strategy.StoreData;

import com.ssa.strategy.StoreData1;

import com.ssa.strategy.TooManyAttempts;

import com.ssa.strategy.TooManyAttempts1;

import com.ssa.strategy.WithdrawBelowMin;

import com.ssa.strategy.WithdrawBelowMin1;

//abstract factory for Account 1 used to get objects used by Account1

public class AbstractFactory\_Account1 extends AbstractFactory {

@Override

public void initialize(AbstractFactory abstractFactory) {

OutputProcessor outputProcessor = new OutputProcessor();

outputProcessor.initialize(this);//initializing the output processor

}

@Override

public StoreData getstoreData() {

return new StoreData1();

}

@Override

public IncorrectID getincorrectIDMsg() {

return new IncorrectID1();

}

@Override

public IncorrectPin getincorrectPinMsg() {

return new IncorrectPin1();

}

@Override

public TooManyAttempts gettooManyAttemptsMsg() {

return new TooManyAttempts1();

}

@Override

public DisplayMenu getdisplayMenu() {

return new DisplayMenu1();

}

@Override

public MakeDeposit getmakeDeposit() {

return new MakeDeposit1();

}

@Override

public DisplayBalance getdisplayBalance() {

return new DisplayBalance1();

}

@Override

public MakeWithdraw getmakeWithdraw() {

return new MakeWithdraw1();

}

@Override

public WithdrawBelowMin getwithdrawBelowMinMsg() {

return new WithdrawBelowMin1();

}

@Override

public NoFunds getdisplayNoFundsMsg() {

return null;

}

@Override

public PromptForPin getpromptForPin() {

return new PromptForPin1();

}

@Override

public Penalty getPenalty() {

return new Penalty1();

}

@Override

public DataStore getdatastore() {

return new DataStore\_Account1();

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End of abstract Factory for Account 1 \*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*Start of abstract Factory for Account 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.ssa.factory;

import com.ssa.datastore.DataStore;

import com.ssa.datastore.DataStore\_Account2;

import com.ssa.outputprocessor.OutputProcessor;

import com.ssa.strategy.DisplayBalance;

import com.ssa.strategy.DisplayBalance2;

import com.ssa.strategy.DisplayMenu;

import com.ssa.strategy.DisplayMenu2;

import com.ssa.strategy.IncorrectID;

import com.ssa.strategy.IncorrectID2;

import com.ssa.strategy.IncorrectPin;

import com.ssa.strategy.IncorrectPin2;

import com.ssa.strategy.MakeDeposit;

import com.ssa.strategy.MakeDeposit2;

import com.ssa.strategy.MakeWithdraw;

import com.ssa.strategy.MakeWithdraw2;

import com.ssa.strategy.NoFunds;

import com.ssa.strategy.NoFunds2;

import com.ssa.strategy.Penalty;

import com.ssa.strategy.PromptForPin;

import com.ssa.strategy.PromptForPin2;

import com.ssa.strategy.StoreData;

import com.ssa.strategy.StoreData2;

import com.ssa.strategy.TooManyAttempts;

import com.ssa.strategy.TooManyAttempts2;

import com.ssa.strategy.WithdrawBelowMin;

//abstract factory for Account 2 used to get objects used by Account2

public class AbstractFactory\_Account2 extends AbstractFactory {

@Override

public DataStore getdatastore() {

// TODO Auto-generated method stub

return new DataStore\_Account2();

}

@Override

public void initialize(AbstractFactory abstractFactory) {

OutputProcessor outputProcessor = new OutputProcessor();

outputProcessor.initialize(this);

}

@Override

public StoreData getstoreData() {

// TODO Auto-generated method stub

return new StoreData2();

}

@Override

public IncorrectID getincorrectIDMsg() {

// TODO Auto-generated method stub

return new IncorrectID2();

}

@Override

public IncorrectPin getincorrectPinMsg() {

// TODO Auto-generated method stub

return new IncorrectPin2();

}

@Override

public TooManyAttempts gettooManyAttemptsMsg() {

// TODO Auto-generated method stub

return new TooManyAttempts2();

}

@Override

public DisplayMenu getdisplayMenu() {

// TODO Auto-generated method stub

return new DisplayMenu2();

}

@Override

public MakeDeposit getmakeDeposit() {

// TODO Auto-generated method stub

return new MakeDeposit2();

}

@Override

public DisplayBalance getdisplayBalance() {

// TODO Auto-generated method stub

return new DisplayBalance2();

}

@Override

public MakeWithdraw getmakeWithdraw() {

// TODO Auto-generated method stub

return new MakeWithdraw2();

}

@Override

public WithdrawBelowMin getwithdrawBelowMinMsg() {

// TODO Auto-generated method stub

return null;

}

@Override

public NoFunds getdisplayNoFundsMsg() {

// TODO Auto-generated method stub

return new NoFunds2();

}

@Override

public PromptForPin getpromptForPin() {

// TODO Auto-generated method stub

return new PromptForPin2();

}

@Override

public Penalty getPenalty() {

// TODO Auto-generated method stub

return null;

}

}

//\*\*\*\*\*\*\* End of abstract Factory for Account 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End of Abstract Factory Pattern\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*Start Output Processor, participates in Strategy Pattern \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.ssa.outputprocessor;

import com.ssa.datastore.DataStore;

import com.ssa.factory.AbstractFactory;

import com.ssa.strategy.DisplayBalance;

import com.ssa.strategy.DisplayMenu;

import com.ssa.strategy.IncorrectID;

import com.ssa.strategy.IncorrectPin;

import com.ssa.strategy.MakeDeposit;

import com.ssa.strategy.MakeWithdraw;

import com.ssa.strategy.NoFunds;

import com.ssa.strategy.Penalty;

import com.ssa.strategy.PromptForPin;

import com.ssa.strategy.StoreData;

import com.ssa.strategy.TooManyAttempts;

import com.ssa.strategy.WithdrawBelowMin;

//responsible to call an action being invoked by MDA

public class OutputProcessor {

//maintains pointers to all output strategies

public static DataStore d;//pointer to data store

public static StoreData sd;//pointer to store data strategy action

public static IncorrectID inId;//pointer to Incorrect ID strategy action

public static IncorrectPin inPin;//pointer to IncorrectPin strategy action

public static TooManyAttempts manyAttempts;//pointer to TooManyAttempts strategy action

public static DisplayMenu menu;//pointer to DisplayMenu strategy action

public static MakeDeposit deposit;//pointer to MakeDeposit strategy action

public static DisplayBalance balance;//pointer to DisplayBalance strategy action

public static MakeWithdraw withdraw;//pointer to MakeWithdraw strategy action

public static WithdrawBelowMin withdrawBelow;//pointer to WithdrawBelowMin strategy action

public static NoFunds noFunds;//pointer to NoFunds strategy action

public static PromptForPin promptPin;//Pointer to PromptForPin strategy action

public static Penalty penalty;//Pointer to Penalty strategy action

public void initialize(AbstractFactory abstractFactory) {

d = abstractFactory.getdatastore();

sd = abstractFactory.getstoreData();

inId = abstractFactory.getincorrectIDMsg();

inPin = abstractFactory.getincorrectPinMsg();

manyAttempts = abstractFactory.gettooManyAttemptsMsg();

menu = abstractFactory.getdisplayMenu();

deposit = abstractFactory.getmakeDeposit();

balance = abstractFactory.getdisplayBalance();

withdraw = abstractFactory.getmakeWithdraw();

withdrawBelow = abstractFactory.getwithdrawBelowMinMsg();

noFunds = abstractFactory.getdisplayNoFundsMsg();

promptPin = abstractFactory.getpromptForPin();

penalty = abstractFactory.getPenalty();

}

public DataStore getDataStore(){//returns DataStore pointer

return d;

}

public void storeData(){//calls action on storedata to store temp values

sd.storeData(d);

}

public void incorrectIDMsg(){//calls an action to print incorrect ID msg

inId.incorrectIDMsg();

}

public void incorrectPinMsg(){//calls an action to print incorrect pin msg

inPin.incorrectPinMsg();

}

public void tooManyAttemptsMsg(){////calls an action to print too ManyAttempts msg

manyAttempts.tooManyAttemptsMsg();

}

public void displayMenu(){//calls an action to display menu for respective account

menu.displayMenu();

}

public void makeDeposit(){//calls an action to deposit in account

deposit.makeDeposit(d);

}

public void displayBalance(){//calls an action to show balance for an account

balance.displayBalance(d);

}

public void makeWithdraw(){//calls an action to make withdraw from account

withdraw.makeWithdraw(d);

}

public void withdrawBelowMinMsg(){//calls a action to display a msg saying balance is below required value

withdrawBelow.withdrawBelowMinMsg();

}

public void displayNoFundsMsg(){//calls an action to display no funds msg

noFunds.displayNoFundsMsg();

}

public void promptForPin(){//calls an action to ask for pin for account

promptPin.promptForPin();

}

public void penalty(){//calls an action to apply penalty when user tries to withdraw and balance goes below minimum balance

penalty.penalty(d);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End Output Processor \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Start of Strategy Pattern \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**package** com.ssa.strategy;

//abstract class for display balance

**import** com.ssa.datastore.DataStore;

**public** **abstract** **class** DisplayBalance {

**public** **abstract** **void** displayBalance(DataStore d);

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

**public** **class** DisplayBalance1 **extends** DisplayBalance {

//showing balance for account 1

@Override

**public** **void** displayBalance(DataStore d) {

System.***out***.println("Balance: $" + d.getBalance1());

}

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

**public** **class** DisplayBalance2 **extends** DisplayBalance {

//showing balance for account 2

@Override

**public** **void** displayBalance(DataStore d) {

System.***out***.println("Balance: $" + d.getBalance2());

}

}

**package** com.ssa.strategy;

//an abstract class for Display menu

**public** **abstract** **class** DisplayMenu {

**public** **abstract** **void** displayMenu();

}

**package** com.ssa.strategy;

**public** **class** DisplayMenu1 **extends** DisplayMenu {

//Display menu for account 1

@Override

**public** **void** displayMenu() {

System.***out***.println("Display Menu for Account 1");

}

}

**package** com.ssa.strategy;

**public** **class** DisplayMenu2 **extends** DisplayMenu {

//Display menu for account 2

@Override

**public** **void** displayMenu() {

System.***out***.println("Display Menu for Account 2");

}

}

**package** com.ssa.strategy;

//an abstract class for IncorrectID action

**public** **abstract** **class** IncorrectID {

**public** **abstract** **void** incorrectIDMsg();

}

**package** com.ssa.strategy;

**public** **class** IncorrectID1 **extends** IncorrectID {

//show incorrect ID msg for account 1

@Override

**public** **void** incorrectIDMsg() {

System.***out***.println("Incorrect ID for Account 1");

}

}

**package** com.ssa.strategy;

**public** **class** IncorrectID2 **extends** IncorrectID {

//show incorrect ID msg for account 2

@Override

**public** **void** incorrectIDMsg() {

System.***out***.println("Incorrect ID for Account 2");

}

}

**package** com.ssa.strategy;

//an abstract class for IncorrectPin

**public** **abstract** **class** IncorrectPin {

**public** **abstract** **void** incorrectPinMsg();

}

**package** com.ssa.strategy;

**public** **class** IncorrectPin1 **extends** IncorrectPin {

//show IncorrectPin msg for account 1

@Override

**public** **void** incorrectPinMsg() {

System.***out***.println("Incorrect Pin for Account 1");

}

}

**package** com.ssa.strategy;

**public** **class** IncorrectPin2 **extends** IncorrectPin {

//show IncorrectPin msg for account 2

@Override

**public** **void** incorrectPinMsg() {

System.***out***.println("Incorrect Pin for Account 2");

}

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

//an abstract class for Deposit

**public** **abstract** **class** MakeDeposit {

**public** **abstract** **void** makeDeposit(DataStore d);

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

**public** **class** MakeDeposit1 **extends** MakeDeposit {

//perform deposit operation for account 1

@Override

**public** **void** makeDeposit(DataStore d) {

d.setBalance(d.getBalance1() + d.getTemp\_d1());

}

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

**public** **class** MakeDeposit2 **extends** MakeDeposit {

//perform deposit operation for account 2

@Override

**public** **void** makeDeposit(DataStore d) {

d.setBalance(d.getBalance2() + d.getTemp\_d2());

}

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

//an abstract class for withdraw operation

**public** **abstract** **class** MakeWithdraw {

**public** **abstract** **void** makeWithdraw(DataStore d);

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

**public** **class** MakeWithdraw1 **extends** MakeWithdraw {

//withdraw action for account 1

@Override

**public** **void** makeWithdraw(DataStore d) {

d.setBalance(d.getBalance1()-d.getTemp\_w1());

}

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

**public** **class** MakeWithdraw2 **extends** MakeWithdraw {

//withdraw action for account 2

@Override

**public** **void** makeWithdraw(DataStore d) {

d.setBalance(d.getBalance2() - d.getTemp\_w2());

}

}

**package** com.ssa.strategy;

//an abstract class for NoFunds

**public** **abstract** **class** NoFunds {

**public** **abstract** **void** displayNoFundsMsg();

}

**package** com.ssa.strategy;

**public** **class** NoFunds2 **extends** NoFunds {

//Display No Funds msg for account 1

@Override

**public** **void** displayNoFundsMsg() {

System.***out***.println("No Funds Left in your Account");

}

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

//an abstract class for penalty action

**public** **abstract** **class** Penalty {

**public** **abstract** **void** penalty(DataStore d);

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

**public** **class** Penalty1 **extends** Penalty {

//apply penalty for account 1

@Override

**public** **void** penalty(DataStore d) {

System.***out***.println("Penalty Applied of $20");

d.setBalance(d.getBalance1() - 20);

}

}

**package** com.ssa.strategy;

//an abstract class for PromptForPin action

**public** **abstract** **class** PromptForPin {

**public** **abstract** **void** promptForPin();

}

**package** com.ssa.strategy;

**public** **class** PromptForPin1 **extends** PromptForPin {

//ask for Pin Account 1

@Override

**public** **void** promptForPin() {

System.***out***.println("Enter Pin for Account 1");

}

}

**package** com.ssa.strategy;

**public** **class** PromptForPin2 **extends** PromptForPin {

//ask for Pin Account 2

@Override

**public** **void** promptForPin() {

System.***out***.println("Enter Pin for Account 2");

}

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

//abstract class for store data in datastore

**public** **abstract** **class** StoreData {

**public** **abstract** **void** storeData(DataStore d);

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

**public** **class** StoreData1 **extends** StoreData {

//storing temp. data in data store for account 1

@Override

**public** **void** storeData(DataStore d) {

d.setPin(d.getTemp\_p1());

d.setUid(d.getTemp\_y1());

d.setBalance(d.getTemp\_a1());

}

}

**package** com.ssa.strategy;

**import** com.ssa.datastore.DataStore;

**public** **class** StoreData2 **extends** StoreData {

//storing temp. data in data store for account 2

@Override

**public** **void** storeData(DataStore d) {

d.setPin(d.getTemp\_p2());

d.setUid(d.getTemp\_y2());

d.setBalance(d.getTemp\_a2());

}

}

**package** com.ssa.strategy;

//An abstract class for TooManyAttempts action msg

**public** **abstract** **class** TooManyAttempts {

**public** **abstract** **void** tooManyAttemptsMsg();

}

**package** com.ssa.strategy;

**public** **class** TooManyAttempts1 **extends** TooManyAttempts {

//show a msg of too many attempts of incorrect pin for account 1

@Override

**public** **void** tooManyAttemptsMsg() {

System.***out***.println("Too Many Attempts for Account 1");

}

}

**package** com.ssa.strategy;

**public** **class** TooManyAttempts2 **extends** TooManyAttempts {

//show a msg of too many attempts of incorrect pin for account 2

@Override

**public** **void** tooManyAttemptsMsg() {

System.***out***.println("Too Many Attempts for Account 2");

}

}

**package** com.ssa.strategy;

//an abstract class for WithdrawBelowMin action

**public** **abstract** **class** WithdrawBelowMin {

**public** **abstract** **void** withdrawBelowMinMsg();

}

**package** com.ssa.strategy;

**public** **class** WithdrawBelowMin1 **extends** WithdrawBelowMin {

//show the msg of balance is below the required minimum balance

@Override

**public** **void** withdrawBelowMinMsg() {

System.***out***.println("Balance is below minimum required value.");

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* End of Strategy Pattern \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Conclusion:**

Through this project, I learned about how the system can be designed and programmed with completely decoupled architecture. The use of MDA-EFSM, along with three different design pattern supported with a successful completion of an Account system. The Project is successfully implemented in Java Programming Language, and behaves as expected. The learning through this implementation helped to understand the design concepts very well.