**INTRODUCTION:**

We have been tasked to design two different Bank Account components (whose EFSMs have been provided) using Model Driven Architecture and implement the design using

State Pattern

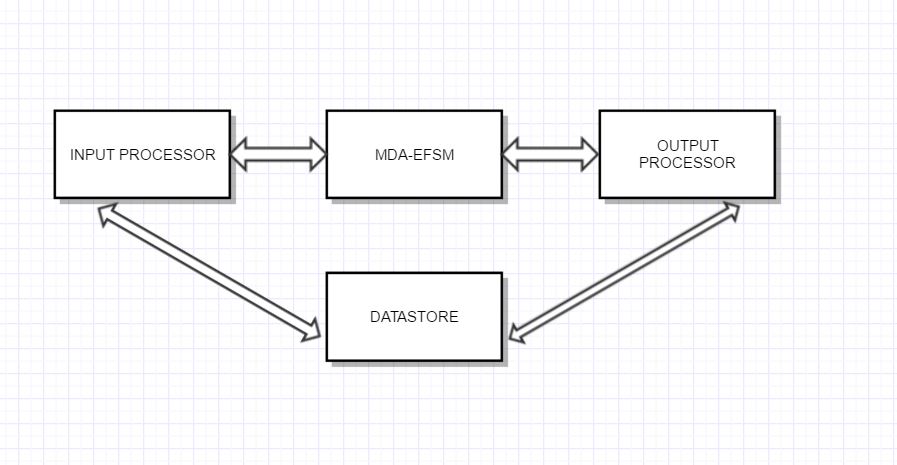
Strategy Pattern

Abstract Factory pattern

The following contains the description of the same.

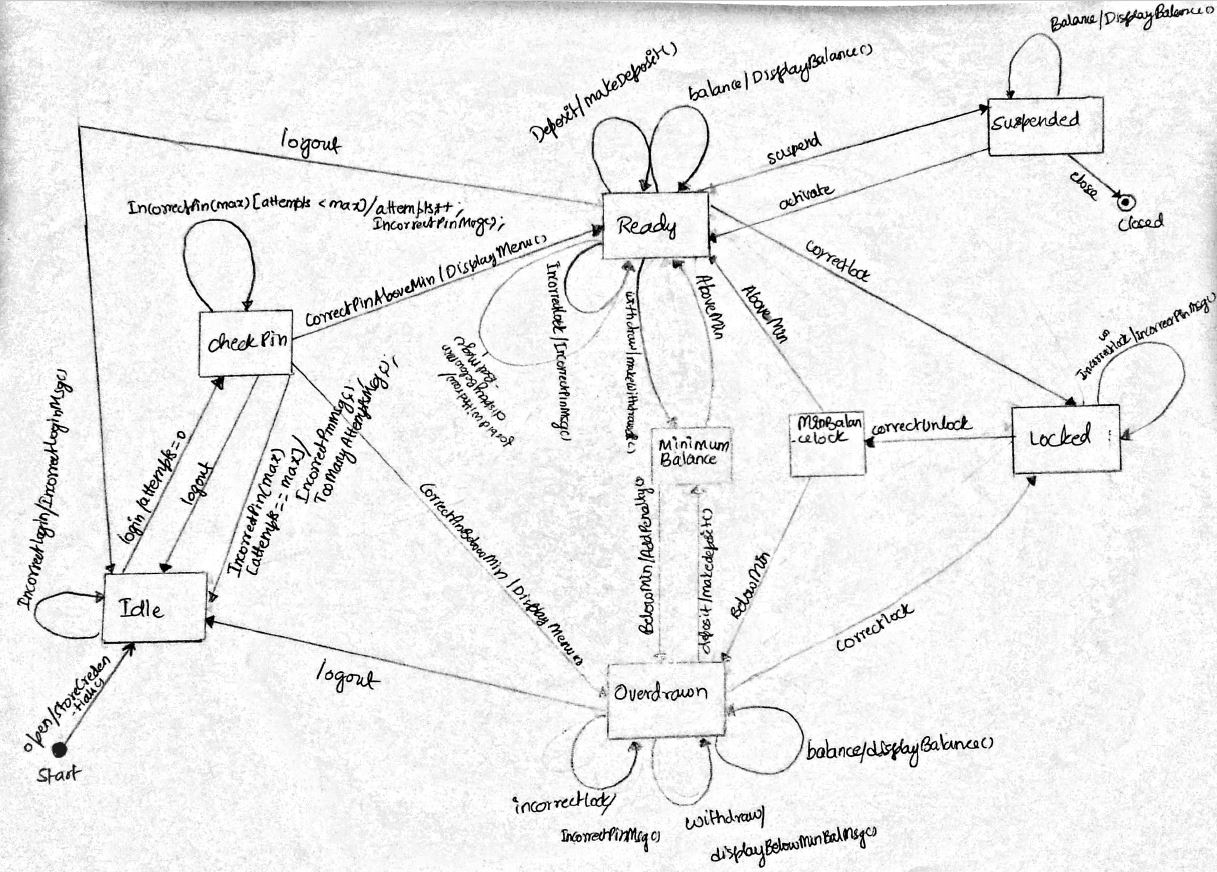
Model-driven architecture (MDA) is a software design approach for the development of software systems. It provides a set of guidelines for the structuring of specifications, which are expressed as models.

**GENERIC REPRESENTATION:**



**MDA-EFSM:**

The diagram below is an MDA-EFSM for the two account components Account 1 and Account 2.



The following are the various **states** in the MDA-EFSM.

Start

Idle

CheckPin

Ready

OverDrawn

MinimumBalance

MinBalanceLock

Locked

Suspended

Closed

The following are the different **meta-events** in the MDA-EFSM

open

incorrectlogin

login

incorrectPin(max)

logout

correctPinAboveMin

correctPinBelowMin

deposit

balance

incorrectLock

incorrectUnlock

correctLock

correctUnlock

aboveMin

belowMin

withdraw

forbidWithdraw

suspend

activate

close

The following are the different **meta-actions** in the MDA-EFSM

StoreCredentials()

//stores login credentials like initial balance, pin, user id in the data store

IncorrectLoginMsg()

//displays incorrect login message

IncorrectPinMsg()

//displays incorrect pin message each time user enters wrong pin

makeDeposit()

//deposit procedure

makeWithdrawal()

//withdraw procedure

displayBalance()

//displays balance amount in the account

displayBelowMinBalMsg()

//displays below minimum balance when the user tries withdrawing some amount when the balance is less than 0

AddPenalty()

//penalizes the user if he tries to withdraw money when the balance is less than $500

DisplayMenu()

//displays list of actions in the form of menu when the user enters the correct pin and ready to do his transactions

TooManyAttemptsMsg()

//displays 'too many incorrect pin attempts' message when the user enter wrong pin three times

**PSEUDO-CODE:**

**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->id){

m->login();

else

m->incorrectlogin();

}

}

pin(String x){

if(x == d->pin){

if(d->b > 500)

m->correctPinAboveMin();

else

m->correctPinBelowMin();

}else

m->incorrectPin(3);

}

logout(){

m->logout();

}

balance(){

m->balance();

}

deposit(float d1){

d->temp\_d1 = d1;

m->Deposit();

if(d->b > 500)

m->AboveMin();

else

m->BelowMin();

}

withdraw(float w){

d->temp\_w = w;

if(d->b > 500){

m->withdraw();

m->AboveMin();

}

else if(d->b > 0 && d->b <= 500){

m->withdraw();

m->BelowMin();

}

else{

m->BelowMin();

m->forbidWithdraw();

}

}

lock(String x){

if(x == d->pin)

m->correctLock();

else

m->incorrectLock();

}

unlock(String x){

if(x == d->pin){

m->correctUnlock();

if(d->b > 500)

m->AboveMin();

else

m->BelowMin();

}

else

m->incorrectUnlock();

}

**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->id){

m->login();

else

m->incorrectLogin();

}

}

PIN(int x){

if(x == d->pin)

m->correctPinAboveMin();

else

m->incorrectPin(2);

}

LOGOUT(){

m->logout();

}

balance(){

m->balance();

}

DEPOSIT(int d1){

d->temp\_d1 = d1;

m->Deposit();

}

suspend(){

m->suspend();

}

activate(){

m->activate();

}

WITHDRAW(float w){

d->temp\_w = w;

if(d->b > 0){

m->withdraw();

m->AboveMin();

}

else{

m->forbidWithdraw();

}

}

close(){

m->close();

}

**CLASS DIAGRAM:**

**Note:**

Since the class diagram is huge with many components and many relationships between the components, I have used the following connectives and the same have been shown (in circles in the class diagram)

AF – AbstractFactory connection

OP – Output\_Processor connection

CF1 – ConcreteFactoryAcc1 connection

CF2 – ConcreteFactoryAcc2 connection

A1 – Account\_1 connection

A2- Account\_2 connection

**CLASSES, ATTRIBUTES, AND THEIR RESPONSIBILITIES:**

**CLASS ACCOUNT-1**

This class contains operations that are specific to account 1.

**Attributes:**

MDA\_EFSM m

This attribute is a reference to the object of the MDA EFSM class.

DataStore d

This attribute is a parent object reference to object of the DataStoreAcc1 class.

**Operations:**

Account\_1(MDA\_EFSM, AbstractFactory)

This is a constructor for instantiating an object of MDA EFSM as well as a concrete factory class of Acc1.

open(String, String, float)

This method takes three parameters namely account pin, account id, and initial balance. The account is opened with this call.

pin(String)

This method takes a parameter namely pin using which the user can do his transaction.

deposit(float)

This method takes a parameter namely amount d (amount to be deposited)

withdraw(float)

This method takes a parameter namely amount w (amount to be withdrawn)

balance()

This method is for displaying the current balance in the account.

login(String)

This method is for logging into the account. It takes a parameter namely account id.

logout()

This method is for logging out of the account.

lock(String)

This method is for locking the account. It takes a parameter namely pin.

unlock(String)

This method is for unlocking the account. It takes a parameter namely pin.

**CLASS ACCOUNT-2**

This class contains operations that are specific to account 2.

**Attributes:**

MDA\_EFSM m

This attribute is a reference to the object of the MDA EFSM class.

DataStore d

This attribute is a parent object reference to object of the DataStoreAcc2 class.

**Operations:**

Account\_2(MDA\_EFSM, AbstractFactory)

This is a constructor for instantiating an object of MDA EFSM as well as a concrete factory class of Acc1.

OPEN(int, int, int)

This method takes three parameters namely account pin, account id, and initial balance. The account is opened with this call.

PIN(int)

This method takes a parameter namely pin using which the user can do his transaction.

DEPOSIT(int)

This method takes a parameter namely amount d (amount to be deposited)

WITHDRAW(int)

This method takes a parameter namely amount w (amount to be withdrawn)

BALANCE()

This method is for displaying the current balance in the account.

LOGIN(int)

This method is for logging into the account. It takes a parameter namely account id.

LOGOUT()

This method is for logging out of the account.

suspend()

This method is for suspending the account.

activate(String)

This method is for activating the suspended account.

close()

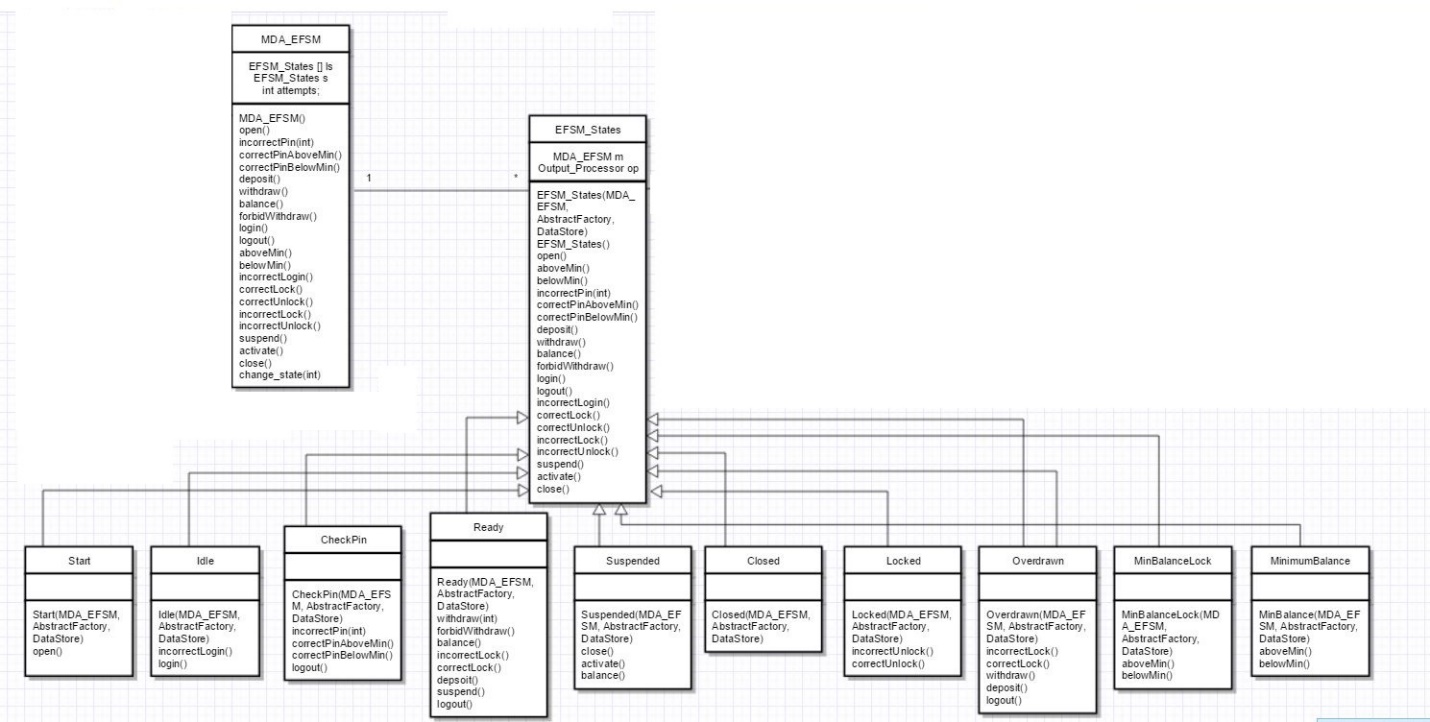
This method closes the account.

**DECENTRALIZED STATE PATTERN**

The Account components exhibit state behavior. The component behaves differently based on the state it is in. To the outside world, it is just one component, but internally a lot of different behaviors are exhibited as state of the account changes as it goes through the lifecycle which makes perfect sense to incorporate State Pattern.

I have incorporated the Decentralized state pattern for the context MDA-EFSM and the generic EFSM-State classes so that MDA-EFSM does minimal functionality which it’s primarily designed for.

MDA-EFSM class is a generic implementation of both account classes. Contains meta-events each of which calls a particular state object’s counterpart method that does account specific operations.



**CLASS MDA\_EFSM**

**Attributes:**

EFSM\_States [] ls

Contains a reference to the list of all state objects.

EFSM\_States s

Contains a reference to the current state object

int attempts

contains the attempts of the user while entering the pin.

**Operations:**

MDA\_EFSM()

Constructor for instantiating the attributes of the MDA EFSM class.

open()

contains a call to the current state’s counterpart

incorrectPin(int)

contains a call to the current state’s counterpart.

correctPinAboveMin()

contains a call to the current state’s counterpart.

correctPinBelowMin()

contains a call to the current state’s counterpart.

deposit()

contains a call to the current state’s counterpart.

withdraw()

contains a call to the current state’s counterpart.

balance()

contains a call to the current state’s counterpart.

forbidWithdraw()

contains a call to the current state’s counterpart.

login()

contains a call to the current state’s counterpart.

logout()

contains a call to the current state’s counterpart.

aboveMin()

contains a call to the current state’s counterpart.

belowMin()

contains a call to the current state’s counterpart.

incorrectLogin()

contains a call to the current state’s counterpart.

correctLock()

contains a call to the current state’s counterpart.

correctUnlock()

contains a call to the current state’s counterpart.

incorrectLock()

contains a call to the current state’s counterpart.

incorrectUnlock()

contains a call to the current state’s counterpart.

suspend()

contains a call to the current state’s counterpart.

activate()

contains a call to the current state’s counterpart.

close()

contains a call to the current state’s counterpart.

change\_state(int)

changes the reference of the current state reference s to point to a different state object based on the operation that’s being called.

**CLASS EFSM\_STATES**

This generic class contains a list of methods supported by the MDA EFSM, methods that each state subclass can handle in its own way based on the current state’s pointer.

**Attributes:**

MDA\_EFSM m

Contains a reference to the MDA EFSM class’s object.

Output\_Processor op

Contains a reference to the Output\_Processor class’s object.

**Operations:**

EFSM\_States(MDA\_EFSM, AbstractFactory, DataStore)

This constructor sets the AbstractFactory reference, datastore reference, MDA EFSM reference to point to the same object created in the account and driver classes respectively.

The following operations contain empty implementations. The subclasses implements each MDA EFSM operation based on the account that was called.

open()

aboveMin()

belowMin()

incorrectPin(int)

correctPinAboveMin()

correctPinBelowMin()

deposit()

withdraw()

balance()

forbidWithdraw()

login()

logout()

incorrectLogin()

correctLock()

correctUnlock()

incorrectLock()

incorrectUnlock()

suspend()

activate()

close()

**CLASS START**

**Operations:**

Start(MDA\_EFSM, AbstractFactory, DataStore)

Instantiates the attributes MDA EFSM, AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class.

open()

Account specific implementation where in the entered pin, id, and balances are stored appropriately. State gets changed to the next state accordingly based on the account.

**CLASS IDLE**

**Operations:**

Idle(MDA\_EFSM, AbstractFactory, DataStore)

Instantiates the attributes MDA EFSM, AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class.

incorrectLogin()

Appropriate message is displayed about incorrect login.

login()

Account specific implementation where in the user logs in using his ID and is directed to the check pin state.

**CLASS CHECKPIN**

**Operations:**

CheckPin(MDA\_EFSM, AbstractFactory, DataStore)

Instantiates the attributes MDA EFSM, AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class.

incorrectPin(int)

Appropriate messages are displayed using the op reference

correctPinAboveMin()

Account specific implementation where the user is moved to the Ready state (based on the initial balance checked in account 1 class implementation), along with displaying a menu for various transactions.

correctPinBelowMin()

Account specific implementation where the user is moved to the Overdrawn state (based on the initial balance checked in account 1 class implementation), along with displaying a menu for various transactions, stays in the Ready state in account 2 as there is no constraint on the initial account balance.

logout()

User is logged out of the account.

**CLASS READY**

**Operations:**

Ready(MDA\_EFSM, AbstractFactory, DataStore)

Instantiates the attributes MDA EFSM, AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class.

withdraw(int)

Account specific implementation for withdraw. State is changed based on the accou t balance. In case of account 1 when the balance is below minimum at any point, the user is directed to overdrawn state. If the user withdraws an amount and balance is still below minimum, a penalty is applied.

forbidWithdraw()

Appropriate messages are displayed when the tries to withdraw when the balance is below minimum

balance()

displays the appropriate balance using op reference

incorrectLock()

displays the appropriate message when the user fails to lock in

correctLock()

user is directed to locked state upon successful locking

depsoit()

Account specific implementation where the user is moved to ready state if the balance after depositing is above minimum balance.

suspend()

Account specific implementation where in the account 2 is suspended temporarily.

logout()

user is logged out of the account.

**CLASS SUSPENDED**

**Operations:**

Suspended(MDA\_EFSM, AbstractFactory, DataStore)

Instantiates the attributes MDA EFSM, AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class.

close()

Account specific implementation where the account 2 is closed.

activate()

Account specific implementation where the account is activated after temporary suspension

balance()

Displays the current balance

**CLASS CLOSED**

**Operations:**

Closed(MDA\_EFSM, AbstractFactory, DataStore)

Instantiates the attributes MDA EFSM, AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class

**CLASS LOCKED**

**Operations:**

Locked(MDA\_EFSM, AbstractFactory, DataStore)

Instantiates the attributes MDA EFSM, AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class

incorrectUnlock()

Account specific implementation where the account 1 is not unlocked and appropriate message is displayed.

correctUnlock()

Account specific implementation where the account 1 is unlocked and appropriate message is displayed.

**CLASS OVERDRAWN**

**Operations:**

Overdrawn(MDA\_EFSM, AbstractFactory, DataStore)

Instantiates the attributes MDA EFSM, AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class

incorrectLock()

Account specific implementation where the account 1 is not locked and appropriate message is displayed.

correctLock()

Account specific implementation where the account 1 is locked and state is appropriately changed.

withdraw()

Account specific implementation where in account 1’s withdraw is called. Appropriate message is displayed as weithdraw cannot happen in overdrawn state.

deposit()

Account specific implementation where in account 1’s deposit is called and state is changed to Ready if the balance is above minimum. If the user deposits an amount and balance is still below minimum, a penalty is applied.

logout()

user is logged out of the account.

**CLASS MINBALANCELOCK**

**Operations:**

MinBalanceLock(MDA\_EFSM, AbstractFactory, DataStore)

Instantiates the attributes MDA EFSM, AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class

aboveMin()

Upon locking in Account 1, this method appropriately directs the user to be in Ready state based on the current balance.

belowMin()

Upon locking in Account 1, this method appropriately directs the user to be in Overdrawn state based on the current balance.

**CLASS MINIMUMBALANCE**

**Operations:**

MinBalance(MDA\_EFSM, AbstractFactory, DataStore)

Instantiates the attributes MDA EFSM, AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class

aboveMin()

Upon each deposit and withdraw especially in Account 1 implementation of the operations, this method appropriately directs the user to be in Ready state based on the account current balance.

belowMin()

Upon each deposit and withdraw especially in Account 1 implementation of the operations, this method appropriately directs the user to be in Overdrawn state based on the account current balance. A Penalty of $20 is applied when the user tries making a deposit that wouldn’t push the account balance to cross $500 mark.

**CLASS OUTPUT\_PROCESSOR**

This class contains the generic methods that either calls account 1’s operations or account 1’s operations at a time. This is a handler for meta-actions.

**Attributes:**

StoreCredentials sc;

A reference that points to either Account 1 or Account 2’s implementation of storing credentials.

TooManyAttemptsMsg tm;

A reference that points to either Account 1 or Account 2’s implementation of displaying invalid attempts.

AddPenalty ap;

A reference that points to Account 1’s implementation of the adding penalty.

DisplayBalance db;

A reference that points to either account 1’s or account 2’s implementation of the displaying balance.

DisplayBelowMinBalMsg bm;

A reference that points to an implementation of displaying message

DisplayMenu dm;

A reference that points to either account 1 or account 2’s implementation of displaying balance

IncorrectLoginMsg il;

A reference that points to an implementation of displaying menu for either account 1 or account 2

IncorrectPinMsg ip;

A reference that points to an implementation of displaying incorrect pin message

MakeDeposit md;

A reference that points to an implementation of making deposits for either account 1 or account 2

MakeWithdrawal mw;

A reference that points to an implementation of making withdrawal for either account 1 or account 2

Close cl;

A reference that points to an implementation of closing the account 2

DataStore d;

A reference that points to account specific datastore object

**Operations:**

Output\_Processor(AbstractFactory)

Instantiates the attribute AbstractFactory based on the value it was pointed with in the account class and the driver class

Output\_Processor(AbstractFactory, DataStore)

Instantiates the attributes AbstractFactory and DataStore based on the values it was pointed with in the account class and the driver class

storeCredentials()

Calls the account specific storeCredentials operation by instantiating the account specific class for the same.

tooManyAttemptsMsg()

Calls the account specific tooManyAttemptsMsg operation by instantiating the account specific class for the same

addPenalty()

calls the addPenalty operation by instantiating the class that does the same

displayBalance()

Calls the account specific displaybalance operation by instantiating the account specific class for the same

displayBelowMinBalMsg()

calls the displayBelowMinBalMsg operation by instantiating the class that does the same

displayMenu()

Calls the account specific displayMenu operation by instantiating the account specific class for the same

incorrectLoginMsg()

calls the incorrectLoginMsg operation by instantiating the class that does the same

incorrectPinMsg()

calls the incorrectPininMsg operation by instantiating the class that does the same

makeDepsoit()

Calls the account specific makeDeposit operation by instantiating the account specific class for the same

makeWithdrawal()

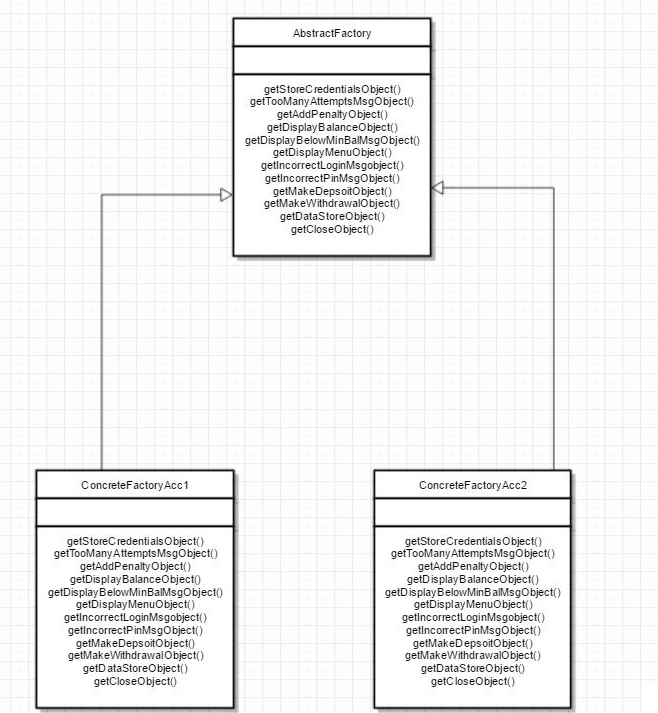
Calls the account specific makeWithdrawal operation by instantiating the account specific class for the same

close()

calls the close operation by instantiating the class that does the same

**ABSTRACT FACTORY PATTERN**

In Abstract Factory pattern, an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Account 1 has a group of classes that is just specific to it. For example, StoreCredentialsAcc1, DisplayMenuAcc1, MakeDepositAcc1. Likewise for account 2 as well. Instead of creating objects of these classes based on conditions, it makes a better design if we segregate all account-specific objects in one place. So, Abstract Factory pattern is appropriate for this scenario.

****

**CLASS ABSTRACTFACTORY**

**Operations:**

getStoreCredentialsObject()

gets either Account 1’s or Account 2’s object of StoreCredentials

getTooManyAttemptsMsgObject()

gets either Account 1’s or Account 2’s object of TooManyAttemptsMsg

getAddPenaltyObject()

gets an object of the class AddPenalty

getDisplayBalanceObject()

gets either Account 1’s or Account 2’s object of DisplayBalance

getDisplayBelowMinBalMsgObject()

gets an object of the class DisplayBelowMinBalMsg

getDisplayMenuObject()

gets either Account 1’s or Account 2’s object of DisplayMenu

getIncorrectLoginMsgobject()

gets an object of IncorrectLoginMsg

getIncorrectPinMsgObject()

gets an object of IncorrectPinMsg

getMakeDepsoitObject()

gets either Account 1’s or Account 2’s object of the MakeDeposit

getMakeWithdrawalObject()

gets either Account 1’s or Account 2’s object of MakeWithdrawal

getDataStoreObject()

gets either Account 1’s or Account 2’s object of DataStore

getCloseObject()

gets an object of Close

**CLASS CONCRETEFACTORYACC1**

Provides account-1 specific behavior for all the methods listed in AbstractFactory class

**Operations:**

getStoreCredentialsObject()

gets Account 1’s object of StoreCredentials

getTooManyAttemptsMsgObject()

gets Account 1’s object of TooManyAttemptsMsg

getAddPenaltyObject()

gets an object of the class AddPenalty

getDisplayBalanceObject()

gets Account 1’s object of DisplayBalance

getDisplayBelowMinBalMsgObject()

gets an object of the class DisplayBelowMinBalMsg

getDisplayMenuObject()

gets Account 1’s object of DisplayMenu

getIncorrectLoginMsgobject()

gets Account 1’s object of tIncorrectLoginMsg

getIncorrectPinMsgObject()

gets Account 1’s object of IncorrectPinMsg

getMakeDepsoitObject()

gets Account 1’s object of the same

getMakeWithdrawalObject()

gets Account 1’s object of MakeWithdrawal

getDataStoreObject()

gets either Account 1’s object of DataStoreAcc1

getCloseObject()

gets a null pointer as there is no implementation for account 1

**CLASS CONCRETEFACTORYACC2**

Provides account-2 specific behavior for all the methods listed in AbstractFactory class

**Operations:**

getStoreCredentialsObject()

gets Account 2’s object of StoreCredentials

getTooManyAttemptsMsgObject()

gets Account 2’s object of TooManyAttemptsMsg

getAddPenaltyObject()

gets an object of the class AddPenalty

getDisplayBalanceObject()

gets Account 2’s object of DisplayBalance

getDisplayBelowMinBalMsgObject()

gets an object of the class DisplayBelowMinBalMsg

getDisplayMenuObject()

gets Account 2s object of DisplayMenu

getIncorrectLoginMsgobject()

gets Account 2’s object of tIncorrectLoginMsg

getIncorrectPinMsgObject()

gets Account 2’s object of IncorrectPinMsg

getMakeDepsoitObject()

gets Account 2’s object of the same

getMakeWithdrawalObject()

gets Account 2’s object of MakeWithdrawal

getDataStoreObject()

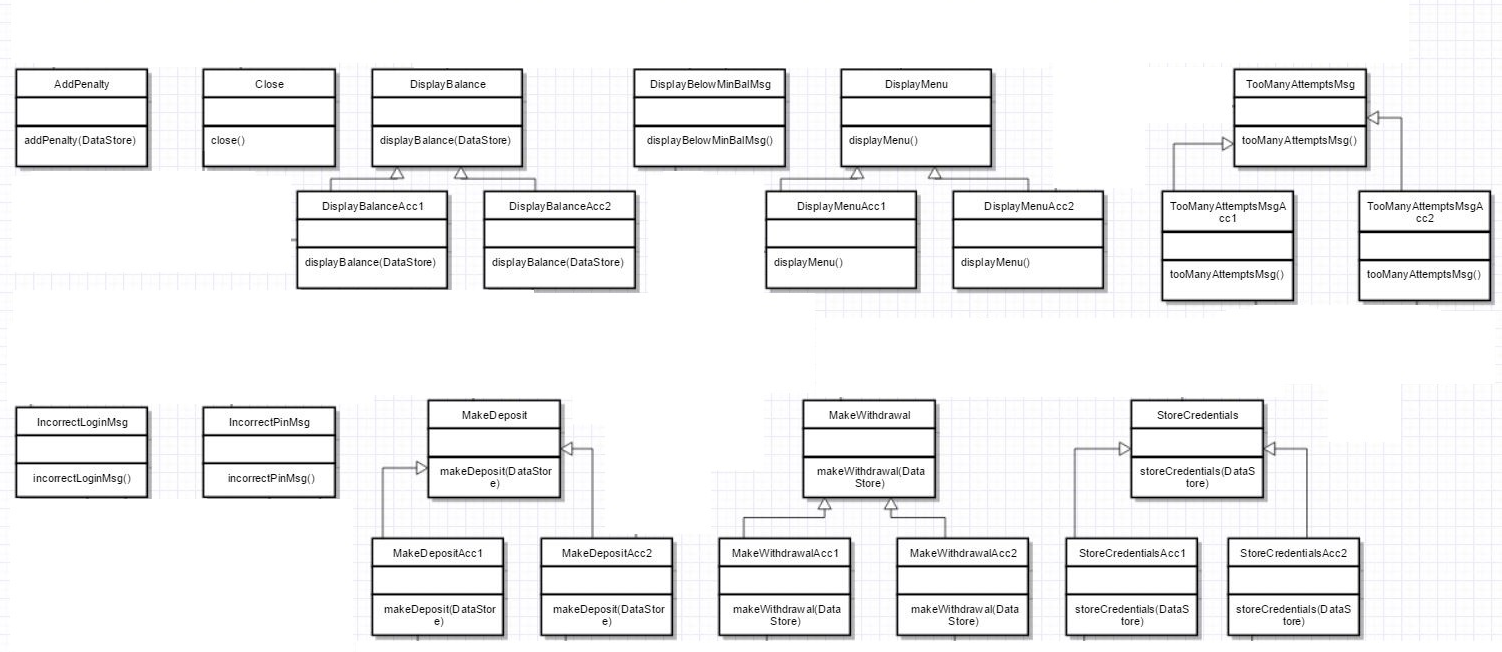
gets Account 2’s object of DataStoreAcc1

getCloseObject()

gets an object of Close

**STRATEGY PATTERN FOR META-ACTIONS**

Actions are almost always Account specific. For example, the 2 accounts vary in terms of displaying menus, displaying balance messages, storing credentials etc. So, it is appropriate to use strategy pattern for these scenarios.

****

**CLASS ADDPENALTY**

**Operations:**

addPenalty(DataStore)

contains the logic for negating $20 from current balance

**CLASS CLOSE**

**Operations:**

close()

contains the logic for closing the account 2

**CLASS DISPLAYBALANCE**

**Operation:**

displayBalance(DataStore)

contains no implementation

**CLASS DISPLAYBALANCEACC1**

**Operation:**

displayBalance(DataStore)

contains account 1 implementation of displaying balance

**CLASS DISPLAYBALANCEACC2**

**Operation:**

displayBalance(DataStore)

contains account 2 implementation of displaying balance

**CLASS DISPLAYBELOWMINBALMSG**

**Operation:**

displayBelowMinBalMsg()

contains the text message that would be displayed for both accounts

**CLASS DISPLAYMENU**

**Operation:**

displayMenu()

contains no implementation

**CLASS DISPLAYMENUACC1**

**Operations:**

displayMenu()

contains the transaction menu “Withdraw, Deposit, Lock, Balance” that would be displayed for account 1

**CLASS DISPLAYMENUACC2**

**Operation:**

displayMenu()

contains the transaction menu “WITHDRAW, DEPOSIT, SUSPEND” that would be displayed for account 2

**CLASS TOOMANYATTEMPTSMSG**

**Operation:**

tooManyAttemptsMsg()

contains no implementation

**CLASS TOOMANYATTEMPTSMSGACC1**

**Operation:**

tooManyAttemptsMsg()

contains the text message that would be displayed for account 1

**CLASS TOOMANYATTEMPTSMSGACC2**

**Operation:**

tooManyAttemptsMsg()

contains the text message that would be displayed for account 2

**CLASS INCORRECTLOGINMSG**

**Operation:**

incorrectLoginMsg()

displays the text message that would be displayed when the user enters wrong id.

IncorrectPinMsg

incorrectPinMsg()

displays the text message that would be displayed when the user enters wrong pin

**CLASS MAKEDEPOSIT**

**Operation:**

makeDeposit(DataStore)

contains no implementation

**CLASS MAKEDEPOSITACC1**

**Operation:**

makeDeposit(DataStore)

contains account 1’s implementation logic for deposit()

**CLASS MAKEDEPOSITACC2**

**Operation:**

makeDeposit(DataStore)

contains account 2’s implementation logic for deposit()

**CLASS MAKEWITHDRAWAL**

**Operation:**

makeWithdrawal(DataStore)

contains no implementation

**CLASS MAKEWITHDRAWALACC1**

**Operation:**

makeWithdrawal(DataStore)

contains account 1’s implementation logic for withdraw()

**CLASS MAKEWITHDRAWALACC2**

**Operation:**

makeWithdrawal(DataStore)

contains account 2’s implementation logic for withdraw()

**CLASS STORECREDENTIALS**

**Operation:**

storeCredentials(DataStore)

contains no implementation

**CLASS STORECREDENTIALSACC1**

**Operation:**

storeCredentials(DataStore)

contains account 1’s implementation logic for storing credentials

**CLASS STORECREDENTIALSACC2**

**Operation:**

storeCredentials(DataStore)

contains account 2’s implementation logic for storing credentials

**CLASS DATASTORE**

**Attributes:**

String temp\_p, temp\_y, pin, id;

Temp\_p is for storing entered pin

Temp\_y is for storing entered id

Pin and Id are for storing with the value that was entered. These values are used throughout the lifecycle of the account 1

float temp\_a, temp\_d, temp\_w, balance;

temp\_a, temp\_d, temp\_w are used for storing values of initial balance, deposit amount, withdraw amount respectively. Balance is set with the value entered while opening account 1

int key, aid, b, temp\_key, temp\_aid, temp\_b, temp\_d1, temp\_w1, temp\_b1;

Temp\_key is for storing entered pin

Temp\_aid is for storing entered id

key and aid are for storing with the value that was entered. These values are used throughout the lifecycle of the account 2

temp\_b1, temp\_d1, temp\_w1 are used for storing values of initial balance, deposit amount, withdraw amount respectively. b is set with the value entered while opening account 2.

**Operations:**

The following methods contains no implementations in this class. Implementations are provided by the subclasses “DataStoreAcc1” and “DataStoreAcc2”.

These variables are all setters and getters for the attributes declared in the class.

getTemp\_p()

getTemp\_y()

getPin()

getId()

getTemp\_a()

getTemp\_d()

getTemp\_w()

getBalance()

getKey()

getAid()

getB()

getTemp\_Key()

getTemp\_aid()

getTemp\_b()

gettemp\_d1()

getTemp\_w1()

getTemp\_b1()

setTemp\_p()

setTemp\_y()

setPin()

setId()

setTemp\_a()

setTemp\_d()

setTemp\_w()

setBalance()

setKey()

setAid()

setB()

setTemp\_Key()

setTemp\_aid()

setTemp\_b()

settemp\_d1()

setTemp\_w1()

setTemp\_b1()

incrementBalance()

decrementBalance()

penalize()

**CLASS DATASTOREACC1**

**Operations:**

getTemp\_p()

getTemp\_y()

getTemp\_a()

getTemp\_d()

getTemp\_w()

The Account 1 specific methods above gets temporarily stored values of pin, id, initial balance, deposit amount, withdraw amount.

getPin()

getId()

getBalance()

The Account 1 specific methods above gets stored values of pin, id, balance. These methods are used throughout the lifecycle of account 1.

setTemp\_p()

setTemp\_y()

setTemp\_a()

setTemp\_d()

setTemp\_w()

The Account 1 specific methods above sets values for temp\_p, temp\_y, temp\_a, temp\_d, temp\_w which are used for holding, temporary values for pin, id, initial balance, deposit amount, withdraw amount.

setPin()

setId()

setBalance()

The Account 1 specific methods above sets values for pin, id, balance

incrementBalance()

contains logic for incrementing balance. Called during deposit()

decrementBalance()

contains logic for decrementing balance. Called during withdraw()

penalize()

contains logic for adding penalty. Called after withdraw() in case of account 1

**CLASS DATASTOREACC2**

**Operations:**

getKey()

getAid()

getB()

The Account 2 specific methods above gets stored values of pin, id, balance. These methods are used throughout the lifecycle of account 2.

getTemp\_Key()

getTemp\_aid()

getTemp\_b1()

gettemp\_d1()

getTemp\_w1()

The Account 2 specific methods above gets temporarily stored values of pin, id, initial balance, deposit amount, withdraw amount.

setKey()

setAid()

setB()

The Account 2 specific methods above sets values for pin, id, balance

setTemp\_Key()

setTemp\_aid()

setTemp\_b()

setTemp\_d1()

setTemp\_w1()

setTemp\_b1()

The Account 2 specific methods above sets values for temp\_p, temp\_y, temp\_a, temp\_d, temp\_w which are used for holding, temporary values for pin, id, initial balance, deposit amount, withdraw amount.

incrementBalance()

contains logic for incrementing balance. Called during deposit()

decrementBalance()

contains logic for decrementing balance. Called during withdraw()

**CLASS DRIVER**

This is the main class of the project using which the user does his transactions on his chosen account.

**Operations:**

main()

This method accepts the user’s input for the type of account, the type of operation for the cosen account.

Acc1Operation()

Contains a set of calls to the account 1 specific set of operations

Acc2Operation()

Contains a set of calls to the account 2 specific set of operations

**DYNAMICS WITH SEQUENCE DIAGRAMS:**

1. Making a deposit in Account-1
2. Behavior when an Incorrect pin is entered three times in Account -2

The objects in the following sequence diagrams have been abbreviated. They refer to the following classes.

**Objects in 1st sequence diagram**

D – Driver

A1 – Account-1

M – MDA-EFSM

ST – Start State

Id – Idle State

C – Check Pin State

R – Ready State

MB – Minimum Balance State

OD – Overdrawn State

OP – Output\_Processor

CFA1 – ConcreteFactoryAcc1

SCA1 – StoreCredentialsAcc1

DSA1 – DataStoreAcc1

MDA1 – MakeDepositAcc1

DBA1 – DisplayBalanceAcc1

**Objects involved in 2nd sequence diagram**

D – Driver

A2 – Account-2

M – MDA-EFSM

ST – Start State

Id – Idle State

C – Check Pin State

OP – Output\_Processor

CFA2 – ConcreteFactoryAcc2

SCA2 – StoreCredentialsAcc2

DSA2 – DataStoreAcc2

IPM - IncorrectPinMsg

TMA2 – TooManyAttemptsPinMsgAcc2

**CONCLUSION:**

The two account components Account-1 and Account-2 were successfully designed and implemented using Model Driven Architecture and 3 patterns namely Decentralized State pattern, Strategy Pattern and Abstract Factory Pattern, and is programmed in Java.

The design has been well implemented without any dependency on the platform or database although it goes without saying that Object Oriented Programming Language like Java or C++ is inevitable in this case where the architecture and the components in the architecture need an Object Oriented approach to design.

There is a bit of complexity that is involved in terms of number of classes (count is 45-46) that have been used to depict the actions, behavior and state changes in this architecture. But, it is very clear cut as to what each class does. In other words, the components that is the classes are all strongly decoupled and very cohesive.

Any functional defect/bug can be easily handled with absolutely minimum code changes. Model driven architecture along with design patterns enables programmers/developers to easily come up with solutions to fix the bug as there is clarity as to what each module in each of the loosely coupled, strongly cohesive components (classes) does. I am someone who has experienced working on spaghetti code in organizations. So, I am certain this architecture holds ground strongly from maintainability point of view, over longevity.

**SOURCE CODE:**