CS 586

SOFTWARE SYSTEM ARCHITECTURE

PROJECT REPORT

ROSHNI MASAND

A20355488

INTRODUCTION

The goal of this project is to design two different ACCOUNT components using a Model-Driven Architecture (MDA) and then implement these ACCOUNT components based on this design. Aspects that vary between two ACCOUNT components are maximum number of times incorrect pin can be entered, minimum balance, display menu(s), messages, penalties, operation and data types. Three distinct designs patterns have to be used to design and implement this Machine, namely, State Pattern, Strategy Pattern and Abstract Factory Pattern.

1. MDA-EFSM MODEL FOR THR ACCOUNT COMPONENTS

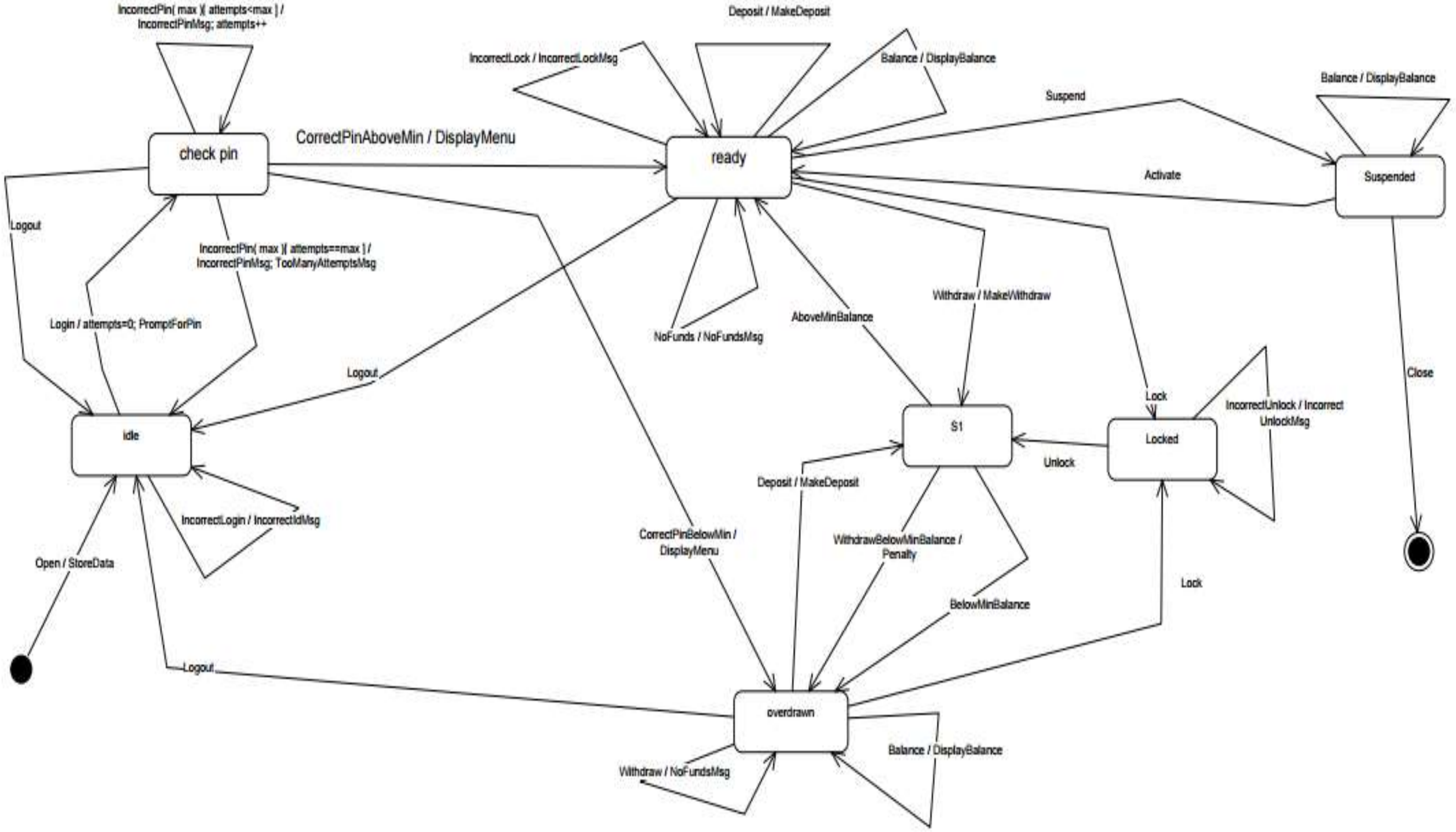
1.1 List of events for MDA-EFSM

Open()
Login()
IncorrectLogin()
IncorectPin(int max)
CorrectPinBelowMin()
CorrectPinAboveMin()
Deposit()
BelowMinBalance()
AboveMinBalance()
Logout()
Balance()
Withdraw()
WithdrawBelowMinBalance()
NoFunds()
Lock()
IncorrectLock()
Unlock()
IncorrectUnlock()
Suspend()
Activate()
Close()

1.2 MDA-EFSM actions with their descriptions

Action No	Action	Description
A1	StoreData()	stores pin from temporary data store to pin in data store
A2	IncorrectIdMsg()	displays incorrect ID message
A3	IncorrectPinMsg()	displays incorrect pin message
A4	TooManyAttemptsMsg()	display too many attempts message
A5	DisplayMenu()	display a menu with a list of transactions
A6	MakeDeposit()	makes deposit (increases balance by a value stored in temp. data store)
A7	DisplayBalance()	displays the current value of the balance
A8	PromptForPin()	prompts to enter pin
A9	MakeWithdraw()	makes withdraw (decreases balance by a value stored in temp. data store)
A10	Penalty()	applies penalty (decreases balance by the amount of penalty)
A11	IncorrectLock Msg()	displays incorrect lock msg
A12	IncorrectUnlock Msg()	displays incorrect unlock msg
A13	NoFundsMsg()	Displays no sufficient funds msg

1.3 MDA-EFSM State Diagram


IncorrectPin(max)[attempts<max] / IncorrectPinMsg; attempts++
check pin
CorrectPinAboveMin / DisplayMenu
Deposit / MakeDeposit
IncorrectLock / IncorrectLockMsg
Balance / DisplayBalance
ready
Suspend
Balance / DisplayBalance
Activate
Suspended
Logout
IncorrectPin(max)[attempts==max] / IncorrectPinMsg; TooManyAttemptsMsg
Login / attempts=0; PromptForPin
NoFunds / NoFundsMsg
AboveMinBalance
Withdraw / MakeWithdraw
Logout
Close
Lock
IncorrectUnlock / Incorrect UnlockMsg
idle
S1
Locked
Unlock
Deposit / MakeDeposit
IncorrectLogin / IncorrectIdMsg
CorrectPinBelowMin / DisplayMenu
WithdrawBelowMinBalance / Penalty
Open / StoreData
BelowMinBalance
Lock
Logout
overdrawn
Balance / DisplayBalance
Withdraw / NoFundsMsg

1.4 Pseudo Code of all operations of Input Processors of ACCOUNT-1 and ACCOUNT-2

ACCOUNT1

```
open (string p, string y, float a)
{
        // store p, y and a in temp data store ds->temp_p=p;
        ds->temp_y=y;
        ds->temp_a=a;
        m->Open();
}
pin (string x)
{
        if (x==ds->pin)
        {
                if (d->balance > 500)
                        m->CorrectPinAboveMin ();
                else m->CorrectPinBelowMin();
        }
else
        m->IncorrectPin(3) ;
}
deposit (float d)
{
        ds->temp_d=d;
        m->Deposit();
        if (ds->balance>500)
                m->AboveMinBalance();
        else
                m->BelowMinBalance();
}
withdraw (float w)
{
        ds->temp_w=w;
        m->withdraw();
        if ((ds->balance>500)
                m->AboveMinBalance();
        else
                m->WithdrawBelowMinBalance();
}
balance()
{
        m->Balance();
}
login (string y)
{
        if (y==ds->uid)
                m->Login();
        else
                m->IncorrectLogin();
}
```

```
logout()
{
        m->Logout();
}
lock (string x)
{
        if (ds->pin==x)
                m->Lock();
        else
                m->IncorrectLock();
}
unlock (string x)
{
        if (x==ds->pin)
        {
                m->Unlock();
                if (ds->balance > 500)
                        m->AboveMinBalance ();
                else
                        m->BelowMinBalance();
        }
        else
                m->IncorrectUnlock();
}
```

ACCOUNT2

```
OPEN (int p, int y, int a)
{
        // store p, y and a in temp data store ds->temp_p=p;
        ds->temp_y=y;
        ds->temp_a=a;
        m->Open();
}
PIN (int x)
{
        if (x==ds->pin)
                m->CorrectPinAboveMin ();
        else
                m->IncorrectPin(2);
}
DEPOSIT (int d)
{
        ds->temp_d=d;
        m->Deposit();
}
WITHDRAW (int w)
{
        ds->temp_w=w;
        if (ds->balance>0)
                m->Withdraw();
                m->AboveMinBalance();
        else
                m->NoFunds();
}
BALANCE()
{
        m->Balance();
}
LOGIN (int y)
{
        if (y==ds->uid)
                m->Login();
        else
                m->IncorrectLogin();

}
LOGOUT()
{
        m->Logout();
}
suspend ()
{
        m->Suspend();
}
activate ()
{
        m->Activate();
}
close ()
{
        m->Close();
}
```

2. Class Diagrams

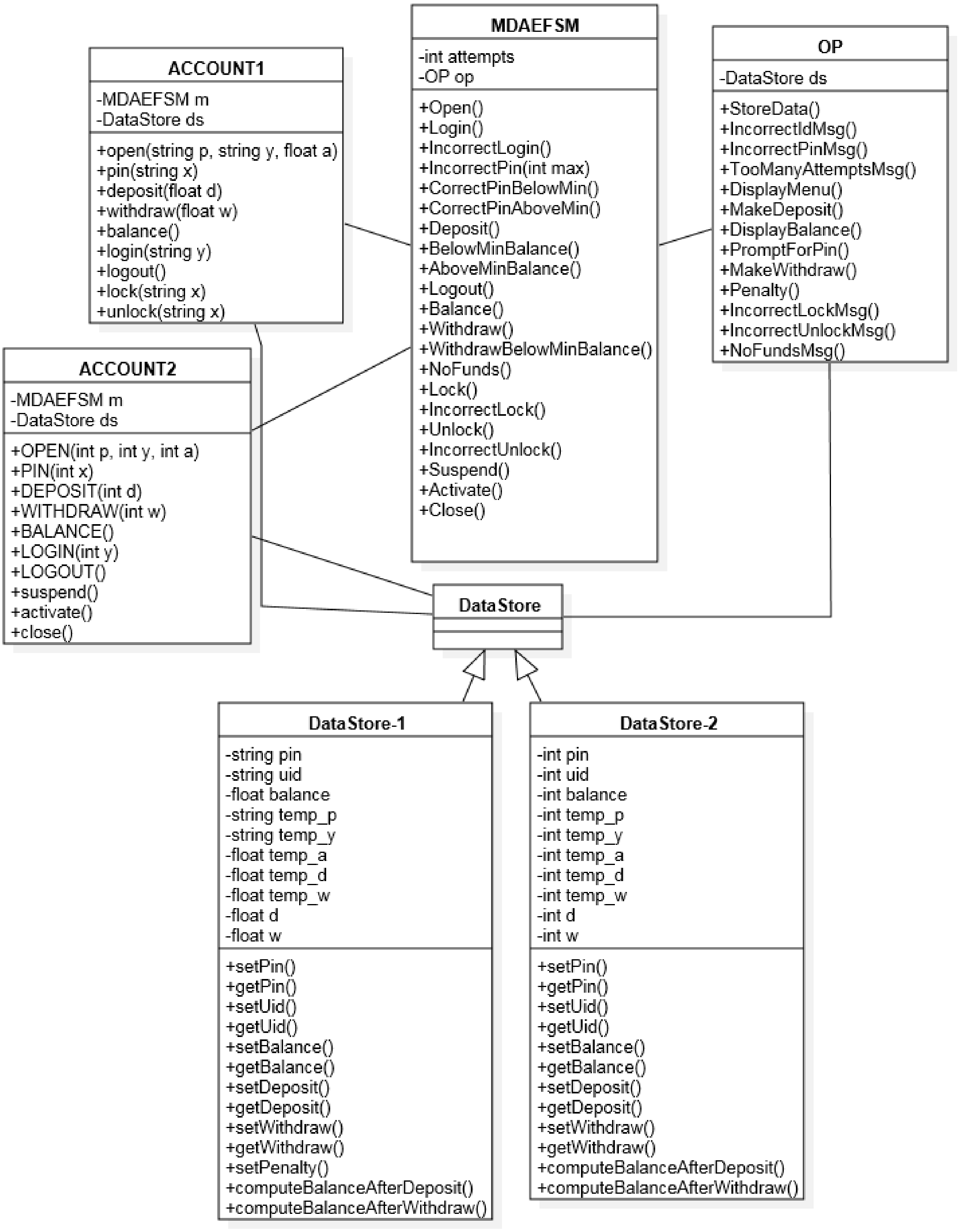
The class diagrams have been represented in the following order.

1.1 General MDA-EFSM Diagram showing Account1, Account2, MDAEFSM, OP and DataStore
1.2 State Pattern highlighting the State Classes
1.3 Strategy Pattern highlighting Strategy Connection
1.4 Abstract Factory Pattern

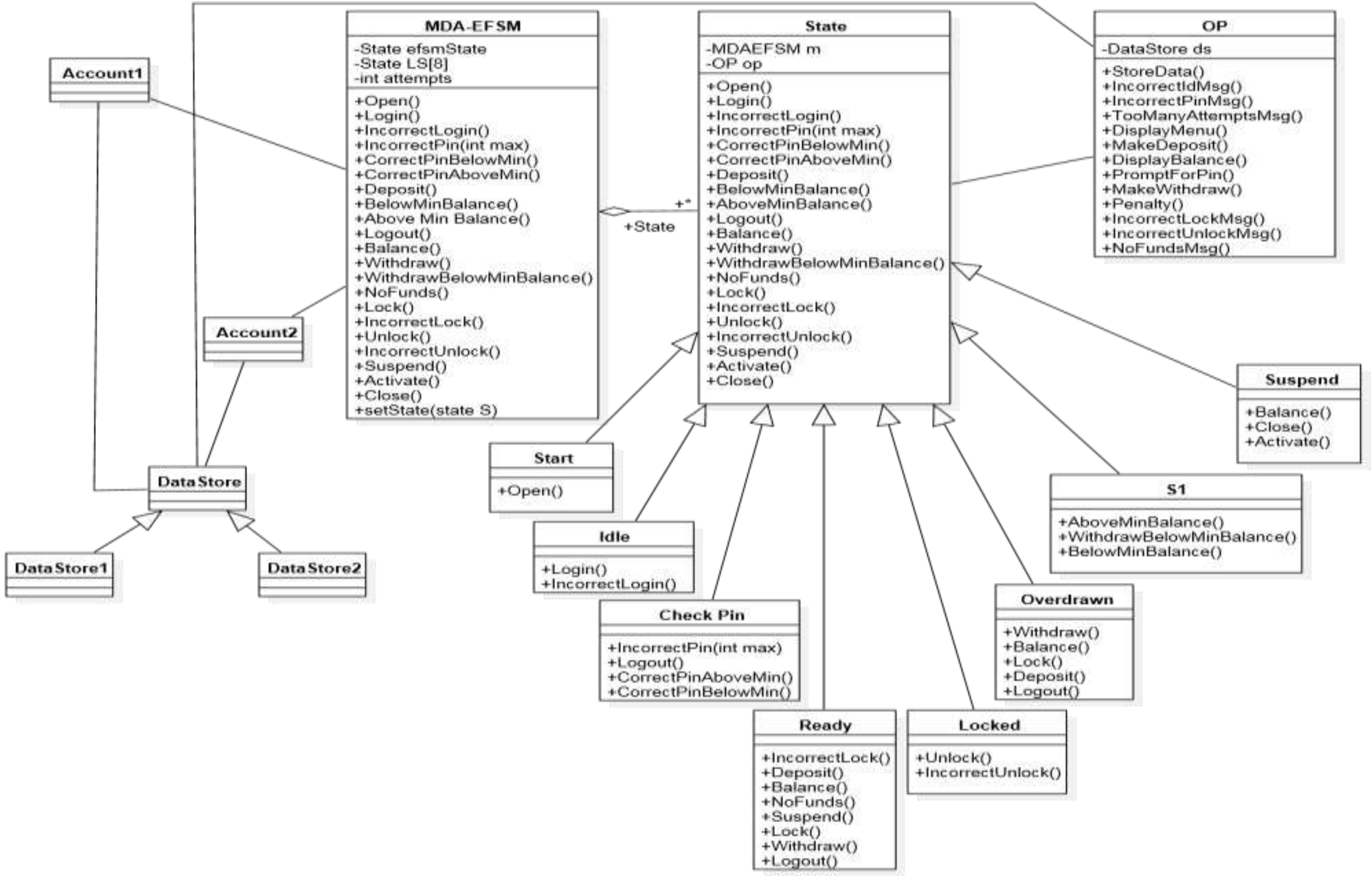
Note1: The part of general MDA-EFSM diagram is the other diagrams is represented by only blocks for simplicity in readability of the diagram.

Note2: The description of all the classes is provided in section 3 which provides the attributes used and the pointers for the different classes. If the pointers are not clear in the Class Diagrams, they can be viewed in Section 3 along with their purpose.

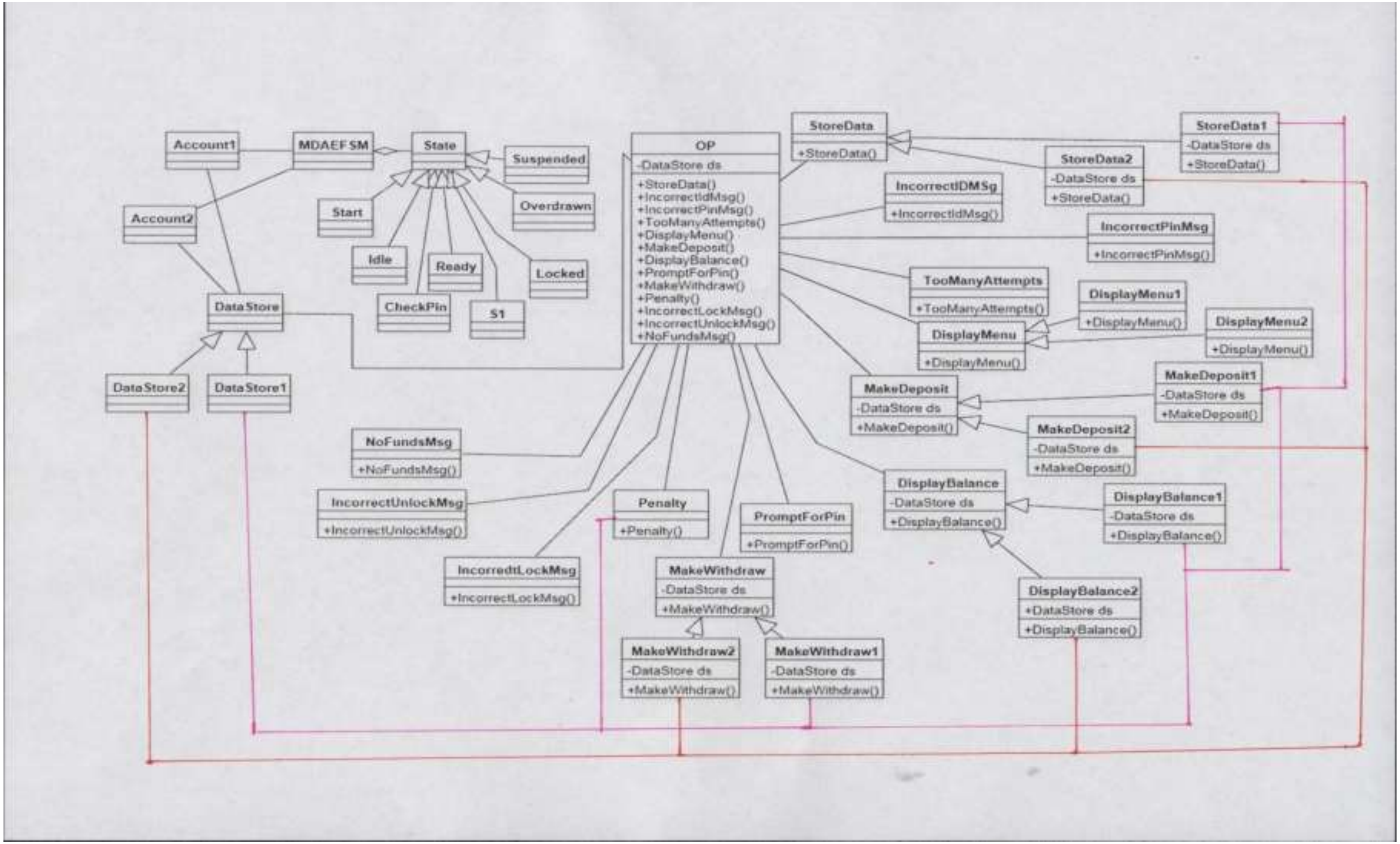
1.1 General MDA-EFSM Diagram



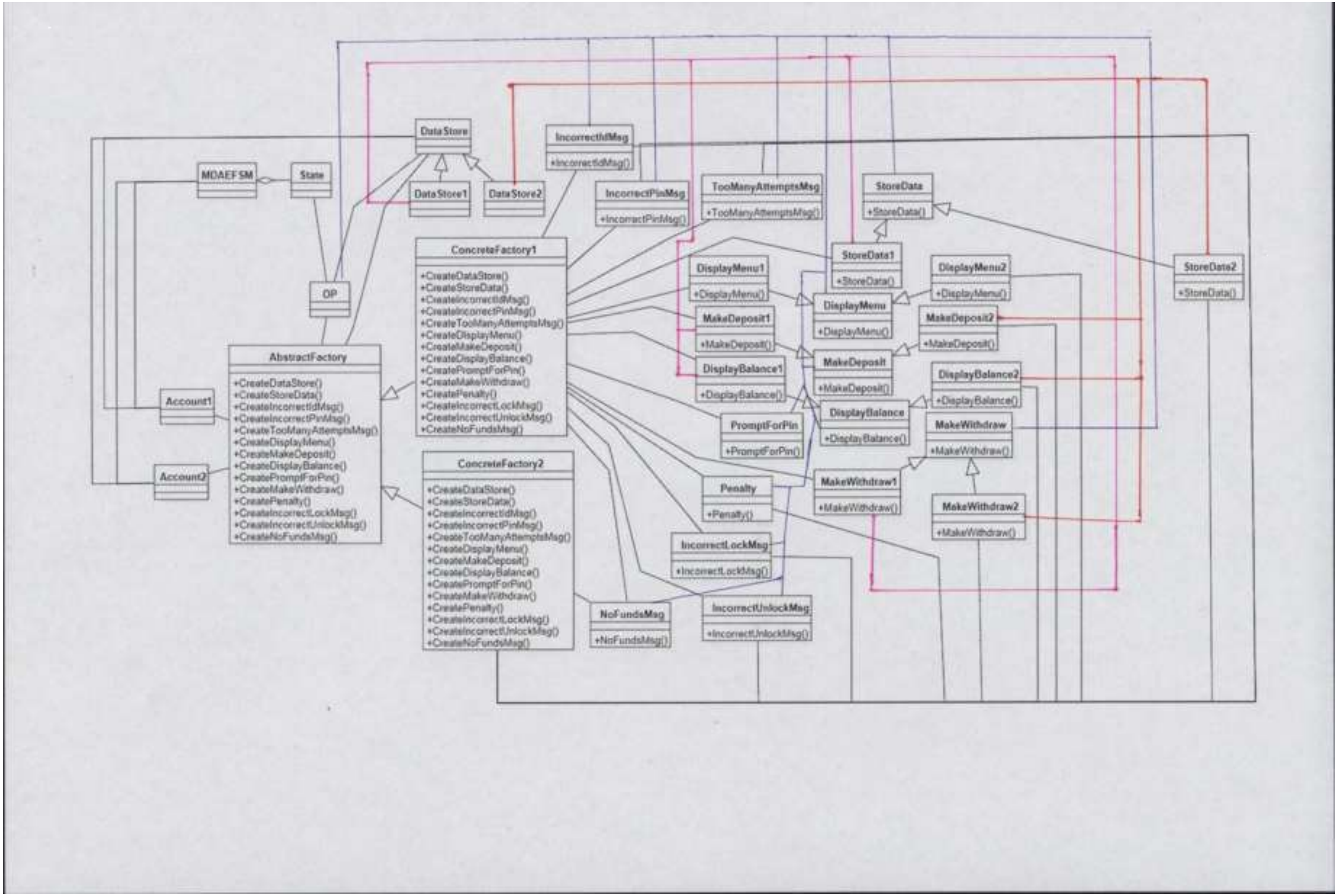
1.2 State Pattern



1.3 Strategy Pattern



1.4 Abstract Factory Pattern



3. CLASSES FOR EACH DIAGRAM WITH THEIR DESCRIPTIONS

3.1 Class Account 1

Purpose	Account 1 represents the implementation for the class that contains input functions for the users. Account 1 stores the data and calls MDAEFSM for further processing.
Attributes	**m** is an object of class MDAEFSM (pointer to MDAEFSM)
	ds is an object of class DataStore(pointer to DataStore)
Operations	**Account2()** is a constructor for the class used for initialization.
	open(String p, String y, float a) is the operation used to initialize the values of Pin, User ID and balance. Pseudo Code is provided in the section above.
	pin(String x) is the operation to check if the entered pin matches the store pin. It also checks for minimum balance of 500.
	deposit(float d) is the operation that takes input the amount to be deposited into the account and performs the operation based on its conditions.
	withdraw(float w) is the operation that takes input the amount to be withdrawn from the account and performs the operation based on its conditions.
	balance() is the operation that displays the current balance present in the account.
	login(String y) is the operation to check if the enter user ID matches the stored user ID.
	logout() is the operation that logs out of the account on call and moves back to idle state.
	lock(String x) is the operation that locks the account if the user pin entered matches the stored pin.
	Unlock(String x) is the operation that unlocks a locked account by checking if the pin entered matches.

3.2 Class Account2

Purpose	Account 2 represents the implementation for the class that contains input functions for the users. Account 2 stores the data and calls MDAEFSM for further processing.
Attributes	**m** is an object of class MDAEFSM (pointer to MDAEFSM)
	ds is an object of class DataStore(pointer to DataStore)
Operations	**Account2()** is a constructor for the class used for initialization.
	OPEN(int p, int y, int a) is the operation used to initialize the values of Pin, User ID and balance. Pseudo Code is provided in the section above.
	PIN(int x) is the operation to check if the entered pin matches the store pin. It also checks for minimum balance of 500.
	DEPOSIT(int d) is the operation that takes input the amount to be deposited into the account and performs the operation based on its conditions.
	WITHDRAW(int w) is the operation that takes input the amount to be withdrawn from the account and performs the operation based on its conditions.

	BALANCE() is the operation that displays the current balance present in the account.
	LOGIN(int y) is the operation to check if the enter user ID matches the stored user ID.
	LOGOUT() is the operation that logs out of the account on call and moves back to idle state.
	suspend() is the operation that suspends the account and moves to suspended state.
	activate() is the operation that activates an account after it has been suspended.
	close() is the operation that closes the account.

3.3 Class DataStore

Purpose	DataStore class is an abstract class which helps us access its sub classes DataStore1 or DataStore2 for Account1 and Account2 respectively.
Attributes	-
Operations	-

3.4 Class DataStore1

Purpose	DataStore1 is a subclass of DataStore. Its main purpose is to store and access data provided from the Account1 class. It stores the temporary data also. The DataStore1 class is accessed anytime the data of Account1 is operated on.
Attributes	**String temp_p** stores the temporary value for pin.
	String temp_y stores the temporary value for user ID.
	float temp_a stores the temporary value for balance.
	String pin stores the permanent(actual value used in comparisons) value for pin.
	String uid stores the permanent(actual value used in comparisons) value for user ID.
	float balance stores the permanent(actual value used in comparisons) value for balance.
	float temp_d stores the temporary value for deposit amount.
	float d stores the permanent(actual value used in comparisons) value for deposit amount.
	float temp_w stores the temporary value for withdraw amount.
	float w stores the permanent(actual value used in comparisons) value for withdraw amount.
Operations	**setPin()** is the operation that sets the pin value with the value of the temporary pin variable (temp_p)
	getPin() returns the set value for pin .
	setUid()is the operation that sets the uid value with the value of the temporary uid variable (temp_y)
	getUid()returns the set value for uid.
	setBalance()is the operation that sets the balance with the value of the temporary balance variable (temp_a)
	getBalance()returns the set value for balance.
	setDeposit()is the operation that sets the deposit(d) value with the value

	of the temporary deposit variable (temp_d)
	getDeposit()returns the set value for deposit.
	setWithdraw()is the operation that sets the withdraw(w) value with the value of the temporary withdraw variable (temp_w)
	getWithdraw()returns the set value for withdraw.
	setPenalty()is the operation that subtracts the penalty(20) from the balance .
	computeBalanceAfterDeposit() is the operation that calculates the balance after deposit is performed.
	computeBalanceAfterWithdraw() is the operation that calculates the balance after withdraw is performed.

3.5 Class DataStore2

Purpose	DataStore2 is a subclass of DataStore. Its main purpose is to store and access data provided from the Account2 class. It stores the temporary data also. The DataStore2 class is accessed anytime the data of Account2 is operated on.
Attributes	**int temp_p** stores the temporary value for pin.
	int temp_y stores the temporary value for user ID.
	int temp_a stores the temporary value for balance.
	int pin stores the permanent(actual value used in comparisons) value for pin.
	int uid stores the permanent(actual value used in comparisons) value for user ID.
	int balance stores the permanent(actual value used in comparisons) value for balance.
	int temp_d stores the temporary value for deposit amount.
	int d stores the permanent(actual value used in comparisons) value for deposit amount.
	int temp_w stores the temporary value for withdraw amount.
	int w stores the permanent(actual value used in comparisons) value for withdraw amount.
Operations	**setPin()** is the operation that sets the pin value with the value of the temporary pin variable (temp_p)
	getPin() returns the set value for pin .
	setUid()is the operation that sets the uid value with the value of the temporary uid variable (temp_y)
	getUid()returns the set value for uid.
	setBalance()is the operation that sets the balance with the value of the temporary balance variable (temp_a)
	getBalance()returns the set value for balance.
	setDeposit()is the operation that sets the deposit(d) value with the value of the temporary deposit variable (temp_d)
	getDeposit()returns the set value for deposit.
	setWithdraw()is the operation that sets the withdraw(w) value with the value of the temporary withdraw variable (temp_w)
	getWithdraw()returns the set value for withdraw.
	computeBalanceAfterDeposit() is the operation that calculates the balance after deposit is performed.

	computeBalanceAfterWithdraw() is the operation that calculates the balance after withdraw is performed.

3.6 Class OP

Purpose	OP Class represents the Output Processor of the MDA and implements various action strategies. Each action having more than one strategy has an abstract class that is associated to the OP, while actions having only one strategy are directly associated with the OP.
Attributes	**af** is a pointer to the AbstractFactory
	ds is a pointer to the DataStore
Operations	OP() is the class constructor used for initialization. It sets the value of af to the AbstractFactory and the value of ds to the DataStore.
	StoreData() creates an object on the appropriate ConcreteFactory and calls the appropriate StoreData() operation from Strategy
	IncorrectIdMsg() creates an object on the appropriate ConcreteFactory and calls IncorrectIdMsg() operation from Strategy
	IncorrectPinMsg() creates an object on the appropriate ConcreteFactory and calls IncorrectPinMsg () operation from Strategy
	TooManyAttemptsMsg() creates an object on the appropriate ConcreteFactory and calls TooManyAttemptsMsg() from Strategy
	DisplayMenu() creates an object on the appropriate ConcreteFactory and calls the appropriate DisplayMenu() operation from Strategy
	MakeDeposit() creates an object on the appropriate ConcreteFactory and calls the appropriate MakeDeposit() operation from Strategy
	DisplayBalance() creates an object on the appropriate ConcreteFactory and calls the appropriate DisplayBalance() operation from Strategy
	PromptForPin() creates an object on the appropriate ConcreteFactory and calls PromptForPin() operation from Strategy
	MakeWithdraw() creates an object on the appropriate ConcreteFactory and calls the appropriate MakeWithdraw() operation from Strategy
	Penalty() creates an object on the appropriate ConcreteFactory and calls Penalty() operation from Strategy
	IncorrectLockMsg() creates an object on the appropriate ConcreteFactory and calls IncorrectLockMsg() operation from Strategy
	IncorrectUnlockMsg() creates an object on the appropriate ConcreteFactory and calls IncorrectUnlockMsg() operation from Strategy
	NoFundsMsg() creates an object on the appropriate ConcreteFactory and calls NoFundsMsg() operation from Strategy

STATE PATTERN CLASSES:

3.7 MDAEFSM

Purpose	MDAEFSM represent the events that will be called after an operation in Account1 or Account2 is called. It represents the platform independent logic of operations from Account1 and Account2.
Attributes	**efsmState** is a pointer to State Class which points to the current State Class the machine is in.
	attempts is the number of attempts which is maintained for the Pin() operation.
Operations	**MDAEFSM (AbstractFactory af, OP op)** is the class constructor and is fired upon creation.This creates a new state and addresses any initialization needed. (Initialization receives a pointer to the abstract factory. This pointer is then passed along to the OP to perform the appropriate function.)
	Open() operation calls the state method **efsmState.Open(),** completes further computation and prints the current state after the operation.
	Login() operation calls the state method **efsmState.Login(),** completes further computation and prints the current state after the operation.
	IncorrectLogin() operation calls the state method **efsmState.IncorrectLogin(),** completes further computation and prints the current state after the operation.
	IncorrectPin(int max) operation calls the state method **efsmState.IncorrectPin(int max),** completes further computation and prints the current state after the operation.
	CorrectPinAboveMin() operation calls the state method **efsmState.CorrectPinAboveMin(),** completes further computation and prints the current state after the operation.
	CorrectPinBelowMin() operation calls the state method **efsmState.CorrectPinBelowMin(),** completes further computation and prints the current state after the operation.
	Deposit() operation calls the state method **efsmState.Deposit(),** completes further computation and prints the current state after the operation.
	BelowMinBalance() operation calls the state method **efsmState.BelowMinBalance(),** completes further computation and prints the current state after the operation.
	AboveMinBalance() operation calls the state method **efsmState.AboveMinBalance(),** completes further computation and prints the current state after the operation.
	Logout() operation calls the state method **efsmState.Logout(),** completes further computation and prints the current state after the operation.
	Balance() operation calls the state method **efsmState.Balance(),** completes further computation and prints the current state after the operation.
	Withdraw() operation calls the state method **efsmState.Withdraw(),** completes further computation and prints the current state after the operation.
	WithdrawBelowMinBalance() operation calls the state method **efsmState.WithdrawBelowMinBalance(),** completes further computation

	and prints the current state after the operation.
	NoFunds() operation calls the state method **efsmState NoFunds(),** completes further computation and prints the current state after the operation.
	Lock() operation calls the state method **efsmState.Lock(),** completes further computation and prints the current state after the operation.
	IncorrectLock() operation calls the state method **efsmState.IncorrectLock(),** completes further computation and prints the current state after the operation.
	Unlock() operation calls the state method **efsmState.Unlock(),** completes further computation and prints the current state after the operation.
	IncorrectUnlock() operation calls the state method **efsmState.IncorrectUnlock(),** completes further computation and prints the current state after the operation.
	Suspend() operation calls the state method **efsmState.Suspend(),** completes further computation and prints the current state after the operation.
	Activate() operation calls the state method **efsmState.Activate(),** completes further computation and prints the current state after the operation.
	Close() operation calls the state method **efsmState.Close(),** completes further computation and prints the current state after the operation.
	setState(State efsmState) is the operation that sets the currents state of the machine.
	getStartState(), getIdleState(), getCheckPinState(), getReadyState(), getSIState(), getLockedState(), getOverdrawnState(), getSuspendedState() operations return their respective states.
	printCurrentState() prints the current machine state.

3.8 Class State

Purpose	This is an interface for the methods of MDAEFSM maintains and is implemented by the various State classes.
Attributes	-
Operations	**Open(), Login(), IncorrectLogin(), IncorrectPin(int max), CorrectPinBelowMin(), CorrectPinAboveMin(), Deposit, AboveMinBalance(), BelowMinBalance(), Logout(), Balance(), Withdraw(), WithdrawBelowMinBalance(), NoFunds(), Lock(), IncorrectLock(), Unlock(), IncorrectUnlock(), Suspend(), Activate() and Close()** have no logic implemented as they will be overridden by their subclasses

3.9 Class Start

Purpose	This is the class for the Start State. Any operation performed leaving this state are implemented here and called when the machine is in this state
Attributes	**m** is the pointer to the MDAEFSM
	op is the pointer to OP
Opearations	**Start()** is a constructor that initiates the state.
	Open() calls the StoreData() action in OP. Performs the functions and then

	changes the state to Idle.
	Login(), IncorrectLogin(), IncorrectPin(int max), CorrectPinBelowMin(), CorrectPinAboveMin(), Deposit, AboveMinBalance(), BelowMinBalance(), Logout(), Balance(), Withdraw(), WithdrawBelowMinBalance(), NoFunds(), Lock(), IncorrectLock(), Unlock(), IncorrectUnlock(), Suspend(), Activate() and Close() are left empty and they do no leave from the Start State in the EFSM.

3.10 Class Idle

Purpose	This is the class for the Idle State. Any operation performed leaving this state are implemented here and called when the machine is in this state
Attributes	**m** is the pointer to the MDAEFSM
	op is the pointer to OP
Opearations	**Idle()** is a constructor that initiates the state.
	Login() calls the PromptForPin() action in OP. Performs the functions and then accordingly changes the state to CheckPin.
	IncorrectLogin() calls the IncorrectIdMsg() action in OP. This remains in the same state.
	Open(),IncorrectPin(int max), CorrectPinBelowMin(), CorrectPinAboveMin(), Deposit(), AboveMinBalance(), BelowMinBalance(), Logout(), Balance(), Withdraw(), WithdrawBelowMinBalance(), NoFunds(), Lock(), IncorrectLock(), Unlock(), IncorrectUnlock(), Suspend(), Activate() and Close() are left empty and they do no leave from the Idle State in the EFSM.

3.11 Class CheckPin

Purpose	This is the class for the CheckPin State. Any operation performed leaving this state are implemented here and called when the machine is in this state
Attributes	**m** is the pointer to the MDAEFSM
	op is the pointer to OP
Opearations	**CheckPin()** is a constructor that initiates the state.
	IncorrectPin(int max) checks for the attempts. If more than three failed attempts are made it calls IncorrectIdMsg() action and TooManyAttempts() action in OP and changes the state to Idle, otherwise it just calls IncorrectIdMsg() action and incrememnts the number of attempts and stays in the same CheckPin State.
	CorrectPinBelowMin() calls the DisplayMenu() action in OP, performs the function and changes the state to Overdrawn State.
	CorrectPinAboveMin()calls the DisplayMenu() action in OP, performs the function and changes the state to Ready State.
	Logout() changes the state to Idle State
	Open(), IncorrectLogin(), Deposit(), AboveMinBalance(), BelowMinBalance(), Logout(), Balance(), Withdraw(), WithdrawBelowMinBalance(), NoFunds(), Lock(), IncorrectLock(), Unlock(), IncorrectUnlock(), Suspend(), Activate() and Close() are left empty and they do no leave from the CheckPin State in the EFSM.

3.12 Class Ready

Purpose	This is the class for the Ready State. Any operation performed leaving this state are implemented here and called when the machine is in this state
Attributes	**m** is the pointer to the MDAEFSM
	op is the pointer to OP
Operations	**Ready()** is a constructor that initiates the state.
	Deposit () calls the MakeDeposit() action in OP, performs the function and stays in the same(Ready) state.
	Balance() calls the DisplayBalance() action in OP, performs the function and stays in the same Ready State.
	Withdraw()calls the MakeWithdraw() action in OP, performs the function and changes the state to S1 State.
	Logout() changes the state to Idle State.
	NoFunds() calls the NoFundsMsg() action in OP, performs the function and stays in the same Ready State.
	Lock() changes the state to Locked State.
	IncorrectLock() calls the IncorrectLockMessage() action in OP, performs the function and stays in the same Ready State.
	Suspend() changes from Ready State to Suspended State.
	Open() , Login(), IncorrectLogin(), IncorrectPin(int max), CorrectPinAboveMin(), CorrectPinBelowMin(), AboveMinBalance(), BelowMinBalance(), WithdrawBelowMinBalance(),Unlock(), IncorrectUnlock(), Activate() and Close() are left empty and they do no leave from the Ready State in the EFSM.

3.13 Class Suspended

Purpose	This is the class for the Suspended State. Any operation performed leaving this state are implemented here and called when the machine is in this state
Attributes	**m** is the pointer to the MDAEFSM
	op is the pointer to OP
Opearations	**Suspended()** is a constructor that initiates the state.
	Balance() calls the DisplayBalance() action in OP, performs the functions and then stays in the same Suspended State.
	Activate() changes the state from Suspended State to Ready State.
	Close() closes the account and terminates the code.
	Open(), Login(), IncorrectLogin(), IncorrectPin(int max), CorrectPinBelowMin(), CorrectPinAboveMin(), Deposit, AboveMinBalance(), BelowMinBalance(), Logout(), Withdraw(), WithdrawBelowMinBalance(), NoFunds(), Lock(), IncorrectLock(), Unlock(), IncorrectUnlock() and Suspend()are left empty and they do no leave from the Suspended State in the EFSM.

3.14 Class S1

Purpose	This is the class for the S1 State. Any operation performed leaving this

	state are implemented here and called when the machine is in this state
Attributes	**m** is the pointer to the MDAEFSM
	op is the pointer to OP
Opearations	**S1()** is a constructor that initiates the state.
	BelowMinBalance() changes the state from S1 to Overdrawn State.
	AboveMinBalance()changes the state from S1 to Ready State.
	WithdrawBelowMinBalance() calls the Penalty() action in OP, performs the function and then changes the state to Overdrawn State.
	Open(),Login(), IncorrectPin(int max), CorrectPinBelowMin(), CorrectPinAboveMin(), Deposit(), Logout(), Balance(), Withdraw(), NoFunds(), Lock(), IncorrectLock(), Unlock(), IncorrectUnlock(), Suspend(), Activate() and Close() are left empty and they do no leave from the S1 State in the EFSM.

3.15 Class Overdrawn

Purpose	This is the class for the Overadrawn State. Any operation performed leaving this state are implemented here and called when the machine is in this state
Attributes	**m** is the pointer to the MDAEFSM
	op is the pointer to OP
Opearations	**Overdrawn()** is a constructor that initiates the state.
	Deposit()calls the MakeDeposit() action in OP, performs the function and then changes the state to S1State.
	Logout()changes the state from Overdrawn State to Idle State.
	Balance() calls the DisplayBalance() action in OP, performs the function and stays in the same overdrawn state.
	Withdraw() calls the NoFundsMsg() action in OP, performs the function and stays in the same Overdrawn State.
	Lock() changes the state from Overdrawn State to Locked State.
	Open(),Login(), IncorrectPin(int max), CorrectPinBelowMin(), CorrectPinAboveMin(),BelowMinBalance(), AboveMinBalance(), WithdrawBelowMinBalance(), NoFunds(), IncorrectLock(), Unlock(), IncorrectUnlock(), Suspend(), Activate() and Close() are left empty and they do no leave from the Overdrawn State in the EFSM.

3.16 Class Locked

Purpose	This is the class for the Start State. Any operation performed leaving this state are implemented here and called when the machine is in this state
Attributes	**m** is the pointer to the MDAEFSM
	op is the pointer to OP
Opearations	**Locked()** is a constructor that initiates the state.
	IncorrectUnlock() calls the IncorrectUnlockMsg() action in OP, performs the functions and then changes the state to Ready State.
	Unlock() changes the state from Locked State to S1 State.
	Open(), Login(), IncorrectLogin(), IncorrectPin(int max), CorrectPinBelowMin(), CorrectPinAboveMin(), Deposit(), AboveMinBalance(), BelowMinBalance(), Logout(), Balance(), Withdraw(), WithdrawBelowMinBalance(), NoFunds(), Lock(),

	IncorrectLock(), Suspend(), Activate() and Close() are left empty and they do no leave from the Locked State in the EFSM.

STRATEGY PATTERN CLASSES

3.17 Class StoreData

Purpose	StoreData is the abstract class for the Action A1:StoreData() and is used to group its subsequent strategies.
Attributes	**ds** is an object of class DataStore
Operations	StoreData() is the class constructor and has no other methods as this is an abstract class.

3.18 Class StoreData1

Purpose	StoreData1 is the class that extends StoreData and is used to store the data for Account1, that is, String pin, String uid, and float balance in the DataStore.
Attributes	**ds** is an object of class DataStore
Operations	**setPin()** sets the DataStore1's pin to the temporary value of pin.
	getPin() returns the value of the pin
	setUid() sets the DataStore1's uid to the temporary value of uid.
	getUid() returns the value of the uid
	setBalance() sets the DataStore1's balance to the temporary value of balance.
	getBalance() returns the value of the balance

3.19 Class StoreData2

Purpose	StoreData2 is the class that extends StoreData and is used to store the data for Account2, that is, int pin,int uid, and int balance in the DataStore.
Attributes	**ds** is an object of class DataStore
Operations	**setPin()** sets the DataStore2's pin to the temporary value of pin.
	getPin() returns the value of the pin
	setUid() sets the DataStore2's uid to the temporary value of uid.
	getUid() returns the value of the uid
	setBalance() sets the DataStore2's balance to the temporary value of balance.
	getBalance() returns the value of the balance

3.20 Class IncorrectIdMsg

Purpose	IncorrectIdMsg is used to display a message to the user if they have entered an incorrect id during the login call.
Attributes	-
Operations	**IncorrectIdMsg()** prints the message "Incorrect Id entered."

3.21 Class IncorrectPinMsg

Purpose	IncorrectPinMsg is used to display a message to the user if they have entered an incorrect pin during the pin call.
Attributes	-
Operations	**IncorrectPinMsg()** prints the message “Incorrect Pin Entered.”

3.22 Class TooManyAttemptsMsg

Purpose	TooManyAttemptsMsg is used to display a message to the user if they have entered an incorrect pin more than the maximum limit during the pin call.
Attributes	-
Operations	**TooManyAttemptsMsg()** prints the message “Too Many Attempts”

3.23 Class DisplayMenu

Purpose	DisplayMenu is the abstract class for the action DisplayMenu() and is used to group its subsequent strategies.
Attributes	**ds** is an object of class DataStore
Operations	DisplayMenu() is the class constructor and has no other methods as this is an abstract class.

3.24 Class DisplayMenu1

Purpose	DisplayMenu1 is the class that extends DisplayMenu and is used to display the menu for Account1.
Attributes	**-**
Operations	**DisplayMenu1()** just displays the menu of the operations.

3.25 Class DisplayMenu2

Purpose	DisplayMenu2 is the class that extends DisplayMenu and is used to display the menu for Account2.
Attributes	**-**
Operations	**DisplayMenu2()** just displays the menu of the operations.

3.26 Class MakeDeposit

Purpose	MakeDeposit is the abstract class for the action MakeDeposit() and is used to group its subsequent strategies.
Attributes	**ds** is an object of class DataStore
Operations	MakeDeposit() is the class constructor and has no other methods as this is an abstract class.

3.27 Class MakeDeposit1

Purpose	MakeDeposit1 is the class that extends MakeDeposit and is used to make the deposit in Account 1 and then compute the balance.
Attributes	**ds** is an object of class DataStore
Operations	**setDeposit()** sets the DataStore1’s d(deposit) to the temporary value of d.

	computeBalanceAfterDepsoit () computes the balance after the deposit is performed.
	getBalance() returns the computed balance.

3.28 Class MakeDeposit2

Purpose	MakeDeposit2 is the class that extends MakeDeposit and is used to make the deposit in Account2 and then compute the balance.
Attributes	**ds** is an object of class DataStore
Operations	**setDeposit()** sets the DataStore2's d(deposit) to the temporary value of d.
	computeBalanceAfterDepsoit () computes the balance after the deposit is performed.
	getBalance() returns the computed balance.

3.29 Class DisplayBalance

Purpose	DisplayBalance is the abstract class for the action DisplayBalance() and is used to group its subsequent strategies.
Attributes	**ds** is an object of class DataStore
Operations	DisplayBalance() is the class constructor and has no other methods as this is an abstract class.

3.30 Class DisplayBalance1

Purpose	DisplayBalance1 is the class that extends DisplayBalance and is used to display the current balance in Account 1
Attributes	**ds** is an object of class DataStore
Operations	**getBalance()** returns the computed balance.

3.31 Class DisplayBalance2

Purpose	DisplayBalance2 is the class that extends DisplayBalance and is used to display the current balance in Account2
Attributes	**ds** is an object of class DataStore
Operations	**getBalance()** returns the computed balance.

3.32 Class PromptForPin

Purpose	PromptForPin is used to display a message to the user to enter the pin if the login is successful.
Attributes	-
Operations	PromptForPin **()** prints the message "Please enter Pin"

3.33 Class MakeWithdraw

Purpose	MakeWithdraw is the abstract class for the action MakeWithdraw () and is used to group its subsequent strategies.
Attributes	**ds** is an object of class DataStore
Operations	MakeWithdraw () is the class constructor and has no other methods as

	this is an abstract class.

3.34 Class MakeWithdraw1

Purpose	MakeWithdraw1 is the class that extends MakeWithdraw and is used to withdraw amount from Account 1 and then compute the balance.
Attributes	**ds** is an object of class DataStore
Operations	**setWithdraw ()** sets the DataStore1's w(withdraw) to the temporary value of w.
	computeBalanceAfterWithdraw () computes the balance after the withdraw is performed.
	getBalance() returns the computed balance.

3.35 Class MakeWithdraw2

Purpose	MakeWithdraw2 is the class that extends MakeWithdraw and is used to withdraw amount from Account 2 and then compute the balance.
Attributes	**ds** is an object of class DataStore
Operations	**setWithdraw ()** sets the DataStore1's w(withdraw) to the temporary value of w.
	computeBalanceAfterWithdraw () computes the balance after the withdraw is performed.
	getBalance() returns the computed balance.

3.36 Class Penalty

Purpose	Penalty is the class that is used to apply a penalty for Account1 if the conditions are not met.
Attributes	**ds** is an object of class DataStore
Operations	**setPenalty ()** applies the minus 20 penalty to the balance and sets the new value as the current value of balance.

3.37 Class IncorrectLockMsg

Purpose	IncorrectLockMsg is used to display a message to the user if they have entered an incorrect pin during the lock call.
Attributes	-
Operations	**IncorrectLockMsg()** prints the message "Incorrect Lock"

3.38 Class IncorrectUnlockMsg

Purpose	IncorrectUnlockMsg is used to display a message to the user if they have entered an incorrect pin during the unlock call.
Attributes	-
Operations	**IncorrectUnlockMsg()** prints the message "Incorrect Unlock"

3.39 Class NoFundsMsg

Purpose	NoFundsMsg is used to display a message to there are no funds in the

	account.
Attributes	-
Operations	**NoFundsMsg()** prints the message "No Funds"

ABSTRACT FACTORY PATTERN CLASSES:

3.40 Class Abstract Factory

Purpose	AbstractFactory is an abstract class for the Factory and groups the ConcreteFactory1 and ConcreteFactory2 for Account1 and Account2 respectively.
Attributes	-
Operations	**CreateDataStore(), CreateStoreData(), CreateIncorrectIdMsg(), CreateIncorrectPinMsg(), CreateTooManyAttemptsMsg(), CreateDisplayMenu(), CreateMakeDeposit(), CreateDisplayBalance(), CreatePromptForPin(), CreateMakeWithdraw(), CreatePenalty(), CreateIncorrectLockMsg(), CreateIncorrectUnlockMsg() and CreateNoFundsMsg()** are abstract methods which will be overridden by the subclasses ConcreteFactory1 and ConcreteFactory2.

3.41 Class ConcreteFactory1

Purpose	ConcreteFactory1 class represents the concrete class for Account1's factory and is used to handle the creation of class objects specific for Account1.
Attributes	**ds** is an object of class DataStore1
	store is an object of class StoreData1
	disp_bal is an object of class DisplayBalance1
	disp_menu is an object of class DisplayMenu1
	id_msg is an object of class IncorrectIdMsg
	pin_msg is an object of class IncorrectPinMsg
	lock_msg is an object of class IncorrectLockMsg
	unlock_msg is an object of class IncorrectUnlockMsg
	make_deposit is an object of class MakeDeposit1
	make_withdraw is an object of class MakeWithdraw1
	no_funds is an object of class NoFundsMsg
	penalty is an object of class Penalty
	prompt_pin is an object of class PromptForPin
	too_many_attempts_msg is an object of class TooManyAttemptsMsg
Operations	**ConcreteFactory1()** is the class constructor.
	CreateDataStore() returns the ds object which can be used to create the datastore.
	CreateStoreData() returns the store object which can be used to call strategy StoreData1()
	CreateIncorrectIdMsg()returns the id_msg object which can be used to call strategy IncorrectIdMsg()
	CreateIncorrectPinMsg()returns the pin_msg object which can be used to call strategy IncorrectPinMsg()
	CreateTooManyAttemptsMsg()returns the too_many_attempts_msg

	object which can be used to call strategy TooManyAttempts()
	CreateDisplayMenu()returns the disp_menu object which can be used to call strategy DisplayMenu1()
	CreateMakeDeposit()returns the make_deposit object which can be used to call strategy MakeDeposit1()
	CreateDisplayBalance()returns the disp_bal object which can be used to call strategy DisplayBalance1()
	CreatePromptForPin()returns the prompt_pin object which can be used to call strategy PromptForPin()
	CreateMakeWithdraw()returns the make_withdraw object which can be used to call strategy MakeWithdraw1()
	CreatePenalty()returns the penalty object which can be used to call strategy Penalty()
	CreateIncorrectLockMsg()returns the lock_msg object which can be used to call strategy IncorrectLockMsg()
	CreateIncorrectUnlockMsg() returns the unlock_msg object which can be used to call strategy IncorrectUnlockMsg()
	CreateNoFundsMsg()returns the no_funds object which can be used to call strategy NoFundsMsg()

3.42 Class ConcreteFactory2

Purpose	ConcreteFactory2 class represents the concrete class for Account2's factory and is used to handle the creation of class objects specific for Account2.
Attributes	**ds** is an object of class DataStore2
	store is an object of class StoreData2
	disp_bal is an object of class DisplayBalance2
	disp_menu is an object of class DisplayMenu2
	id_msg is an object of class IncorrectIdMsg
	pin_msg is an object of class IncorrectPinMsg
	lock_msg is an object of class IncorrectLockMsg
	unlock_msg is an object of class IncorrectUnlockMsg
	make_deposit is an object of class MakeDeposit2
	make_withdraw is an object of class MakeWithdraw2
	no_funds is an object of class NoFundsMsg
	penalty is an object of class Penalty
	prompt_pin is an object of class PromptForPin
	too_many_attempts_msg is an object of class TooManyAttemptsMsg
Operations	**ConcreteFactory2()** is the class constructor.
	CreateDataStore() returns the ds object which can be used to create the datastore.
	CreateStoreData() returns the store object which can be used to call strategy StoreData2()
	CreateIncorrectIdMsg()returns the id_msg object which can be used to call strategy IncorrectIdMsg()
	CreateIncorrectPinMsg()returns the pin_msg object which can be used to call strategy IncorrectPinMsg()
	CreateTooManyAttemptsMsg()returns the too_many_attempts_msg object which can be used to call strategy TooManyAttempts()
	CreateDisplayMenu()returns the disp_menu object which can be used to

	call strategy DisplayMenu2()
	CreateMakeDeposit()returns the make_deposit object which can be used to call strategy MakeDeposit2()
	CreateDisplayBalance()returns the disp_bal object which can be used to call strategy DisplayBalance2()
	CreatePromptForPin()returns the prompt_pin object which can be used to call strategy PromptForPin()
	CreateMakeWithdraw()returns the make_withdraw object which can be used to call strategy MakeWithdraw2()
	CreatePenalty()returns the penalty object which can be used to call strategy Penalty()
	CreateIncorrectLockMsg()returns the lock_msg object which can be used to call strategy IncorrectLockMsg()
	CreateIncorrectUnlockMsg() returns the unlock_msg object which can be used to call strategy IncorrectUnlockMsg()
	CreateNoFundsMsg()returns the no_funds object which can be used to call strategy NoFundsMsg()

4. DYNAMICS:

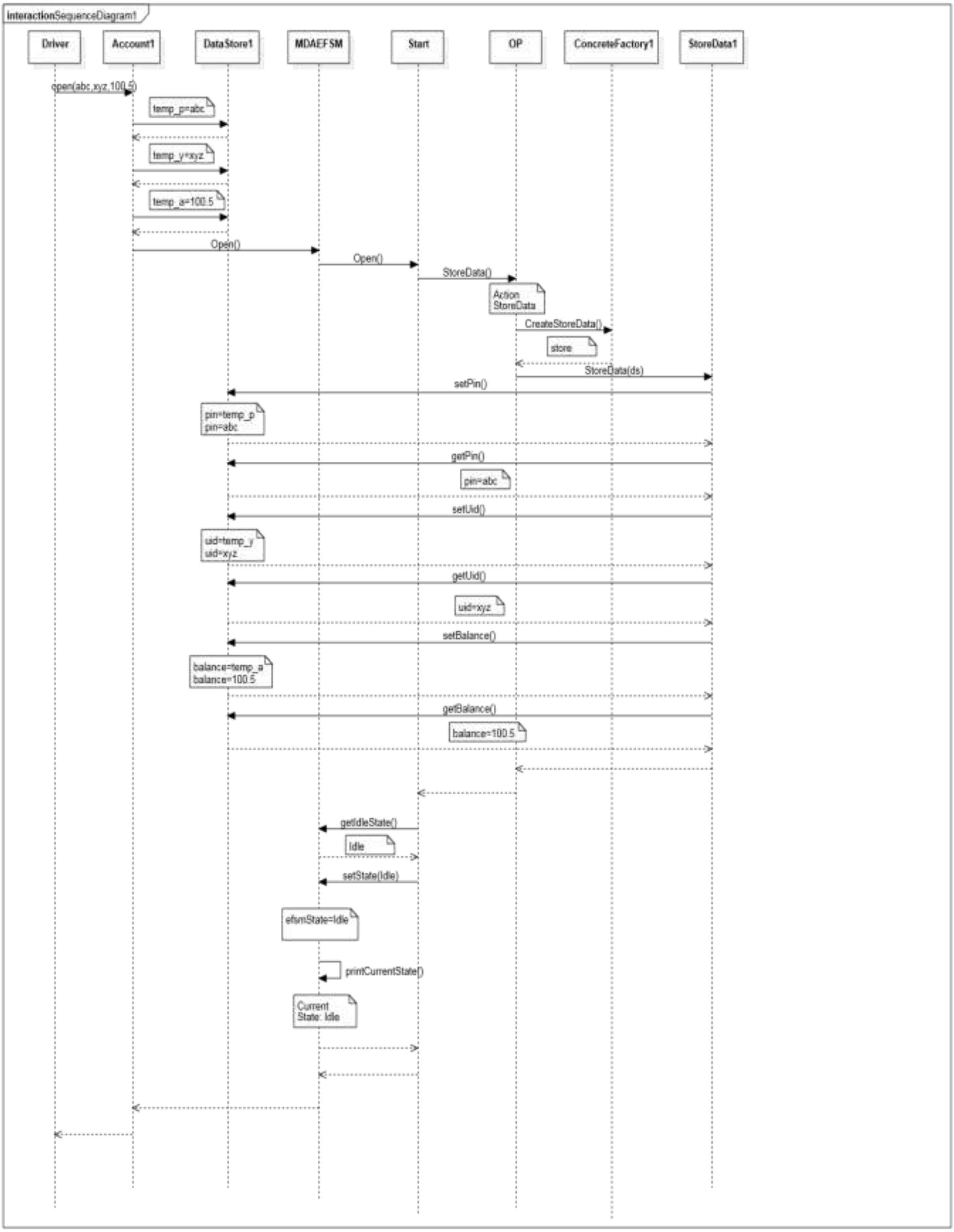
Sequence Diagram for the 2 Scenarios is provided in the section below.

Scenario 1: The following operations are called on Account1: open(abc,xyz,100.5), login(xyz), pin(abc), deposit(400), balance(), logout()

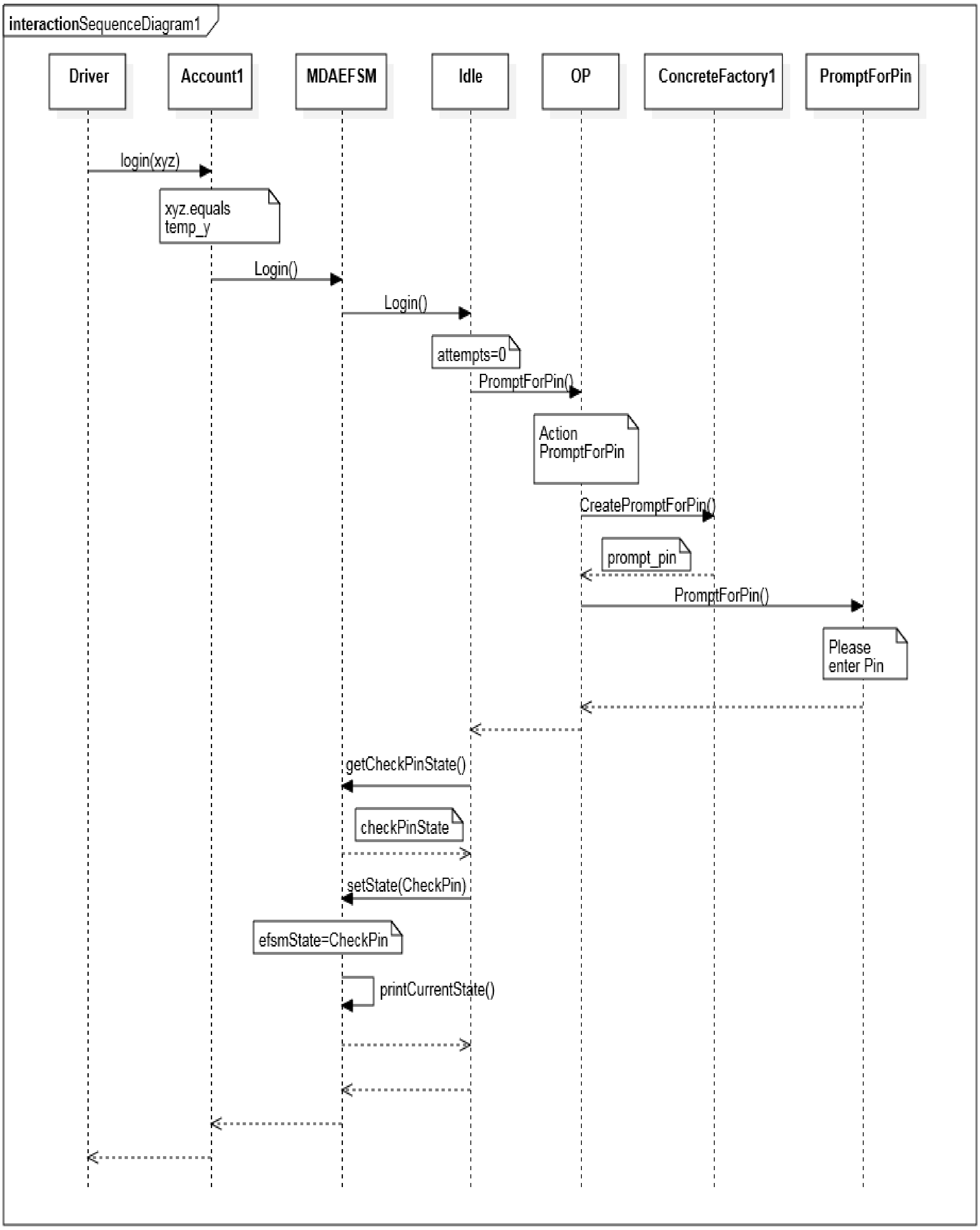
Scenario 2: The following operations are called on Account2: OPEN(123,111,1000), LOGIN(111), PIN(112), PIN(222), PIN(333)

Note: Each operation call is represented as an individual sequence diagram for convenience and readability.

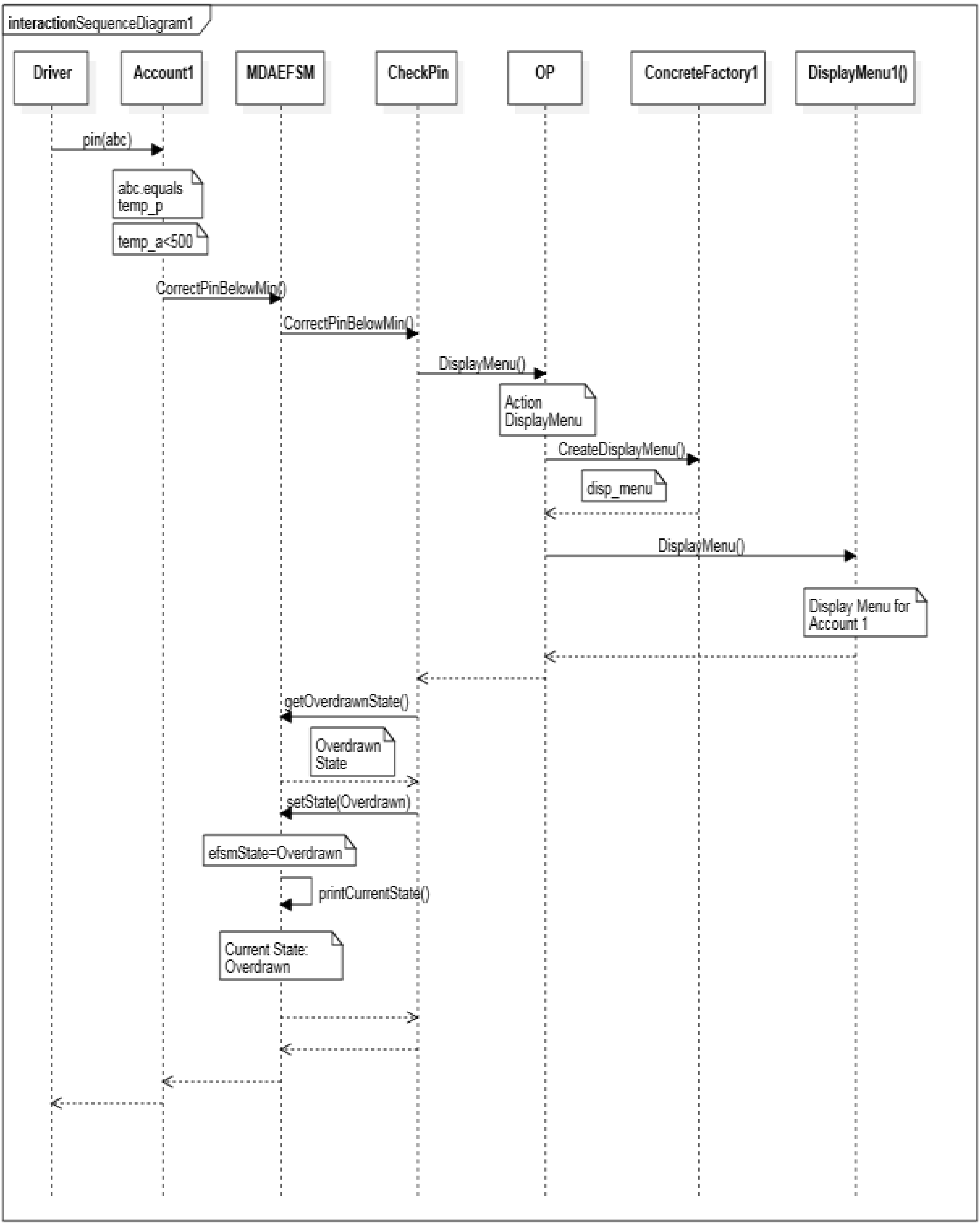
Scenario 1: Account 1
Operation: open(abc,xyz,100.5)



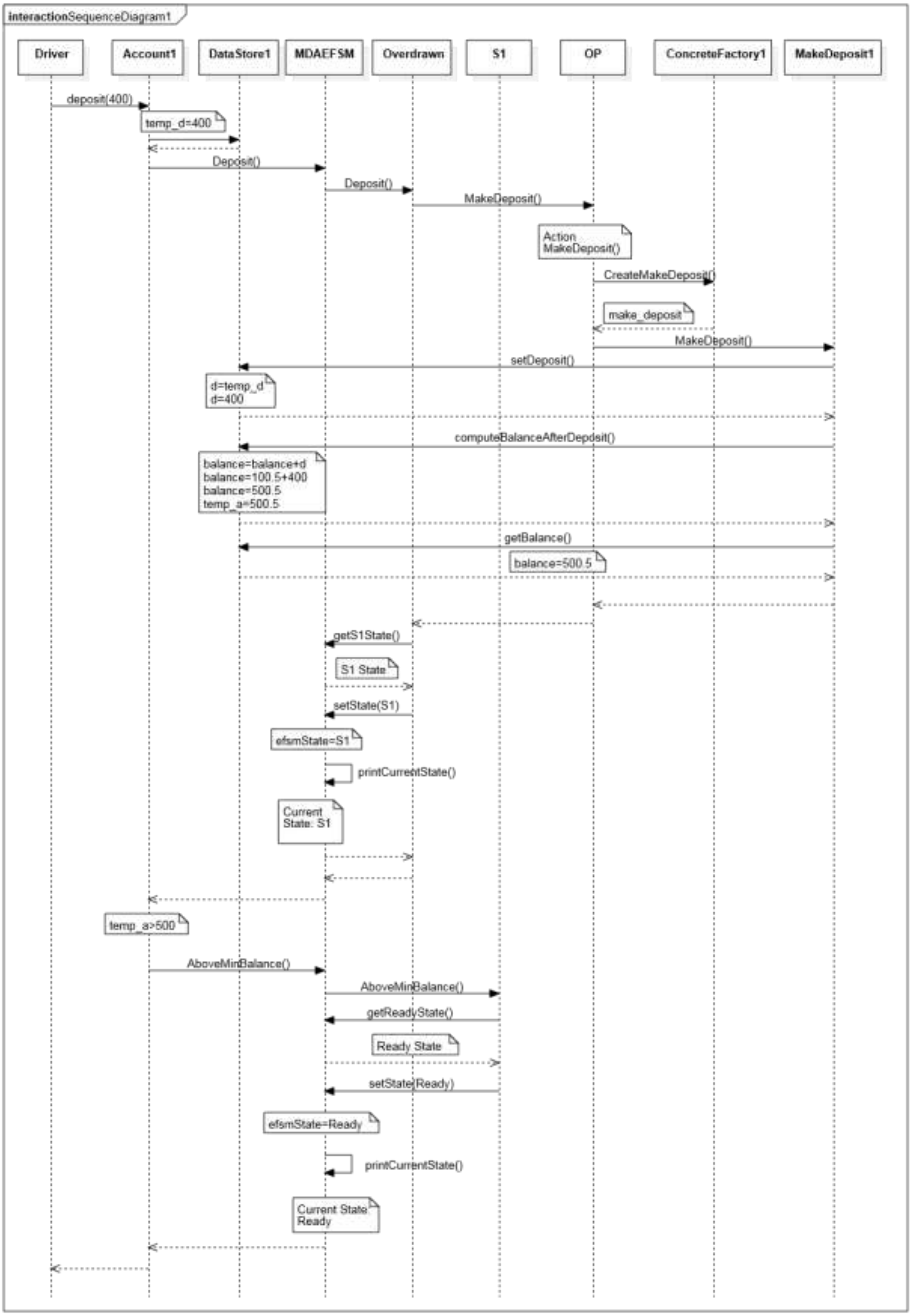
Operation: login(xyz)



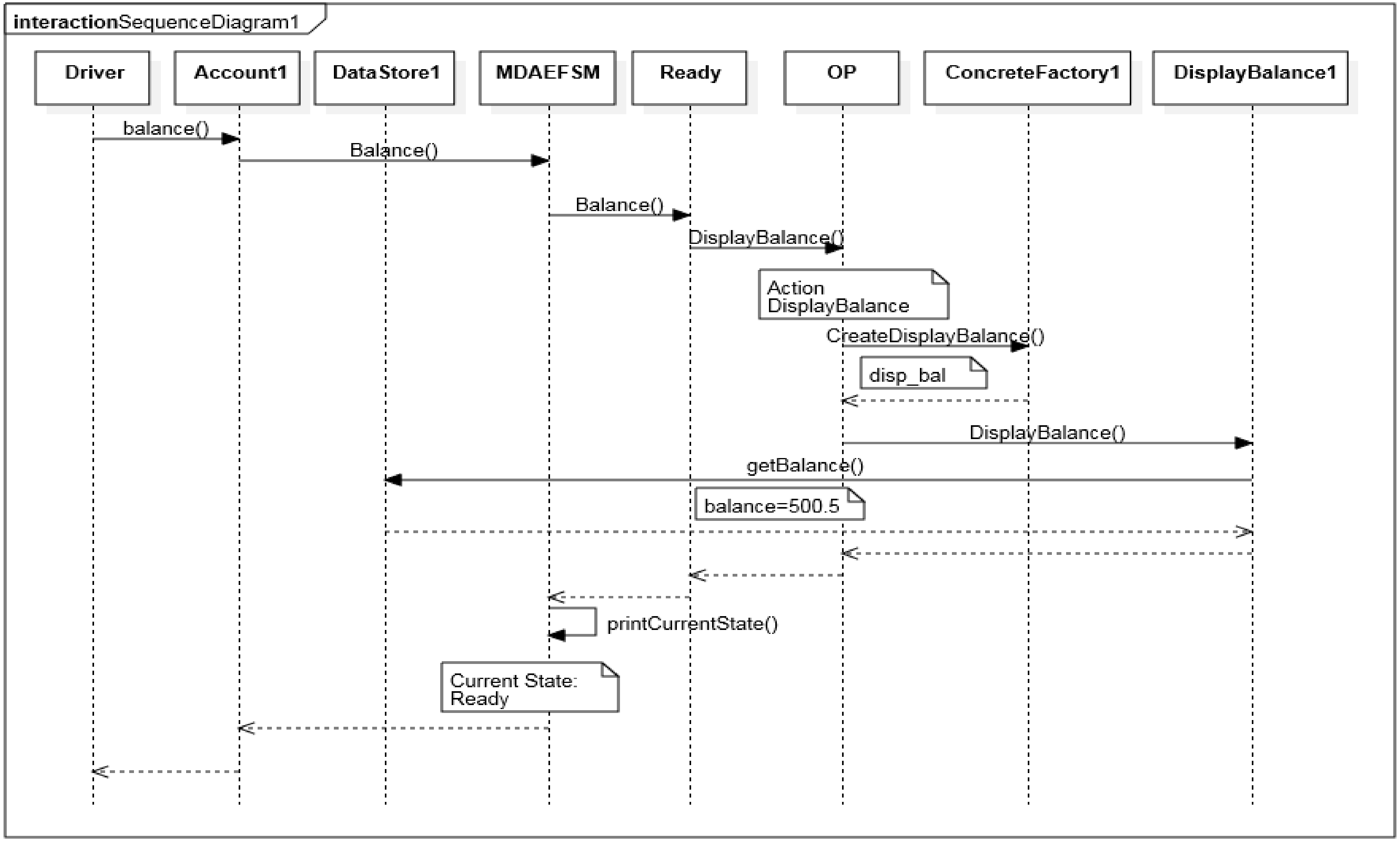
Operation: pin(abc)



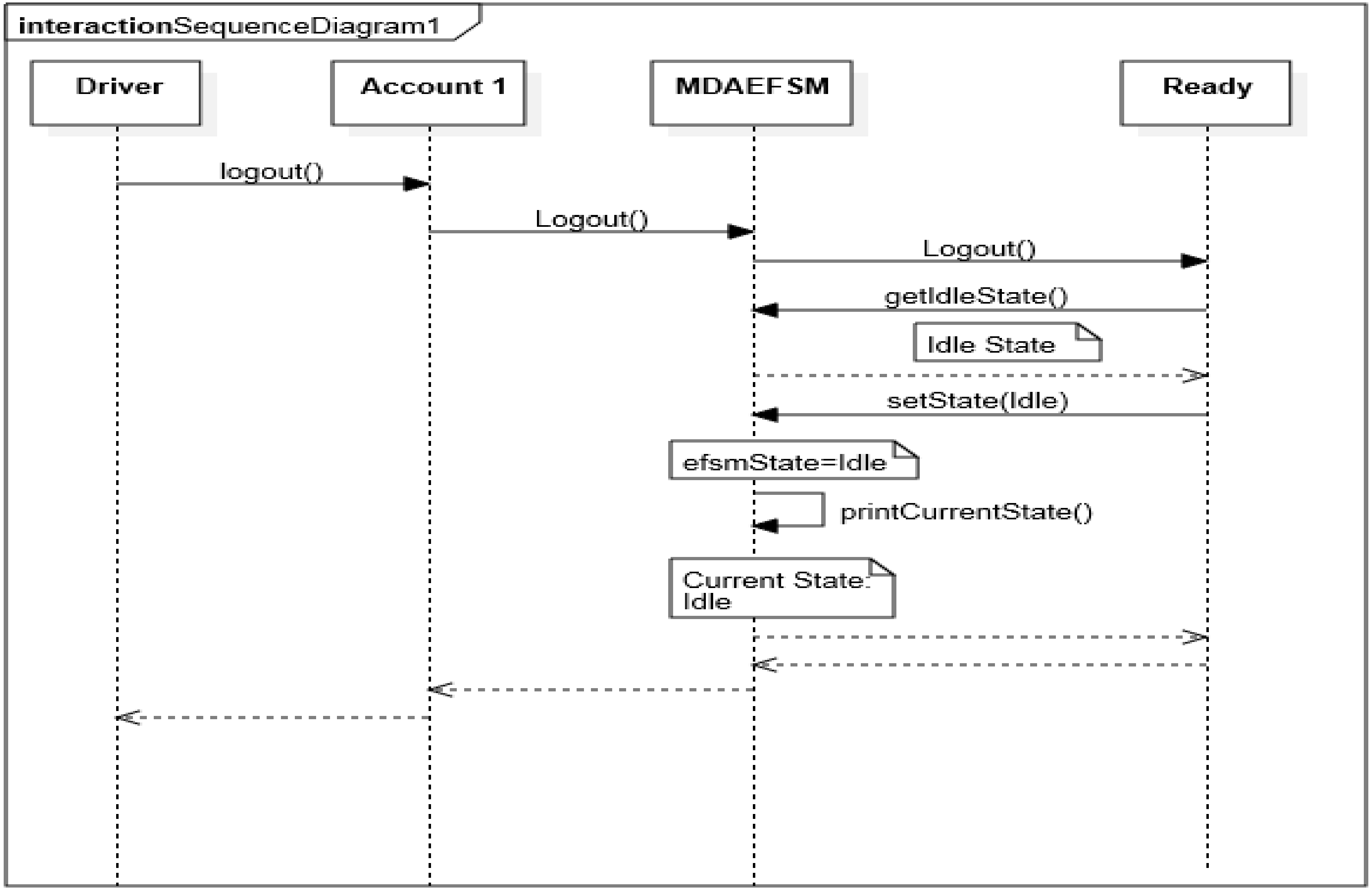
Operation: deposit(400)



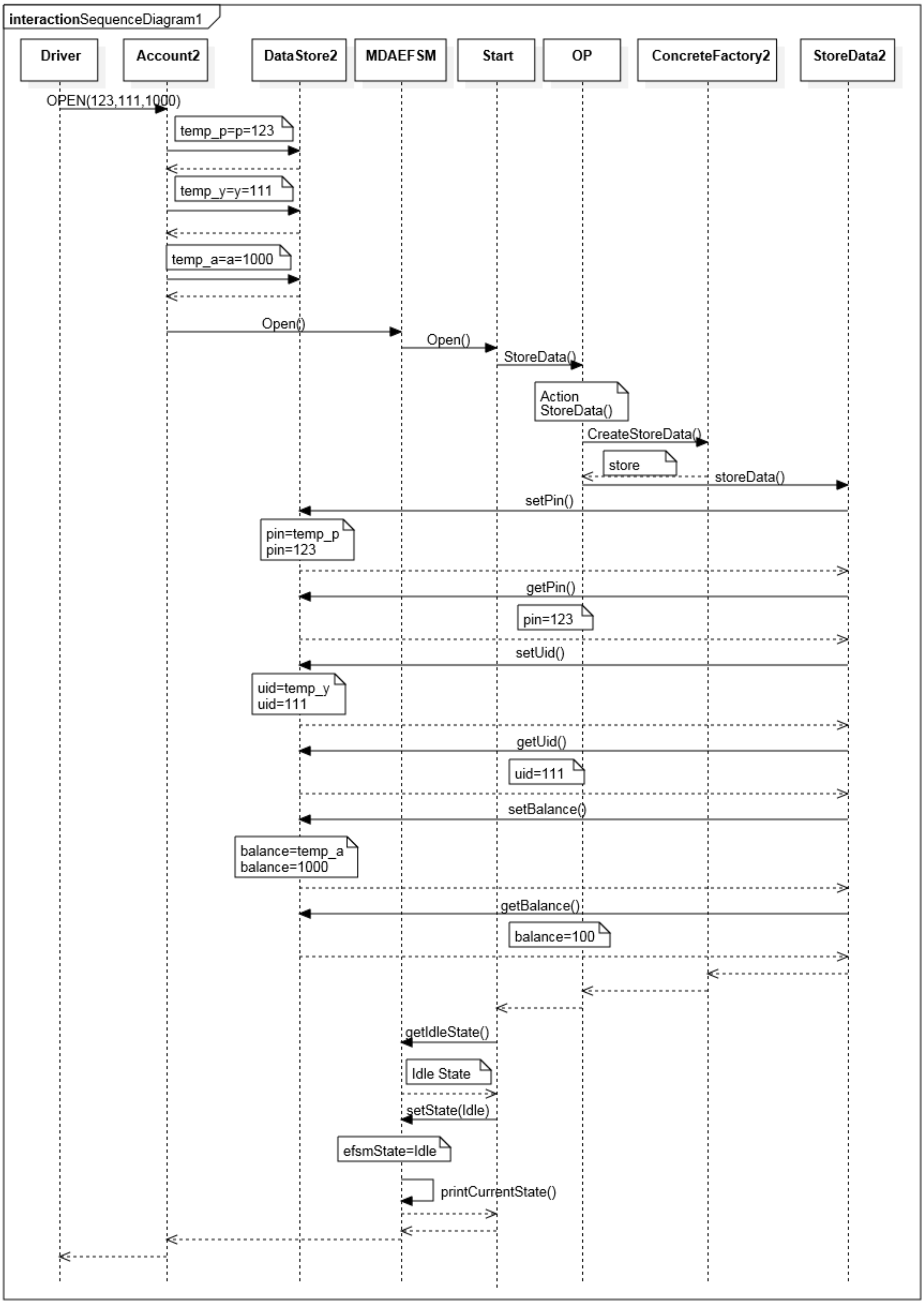
Operation: balance()



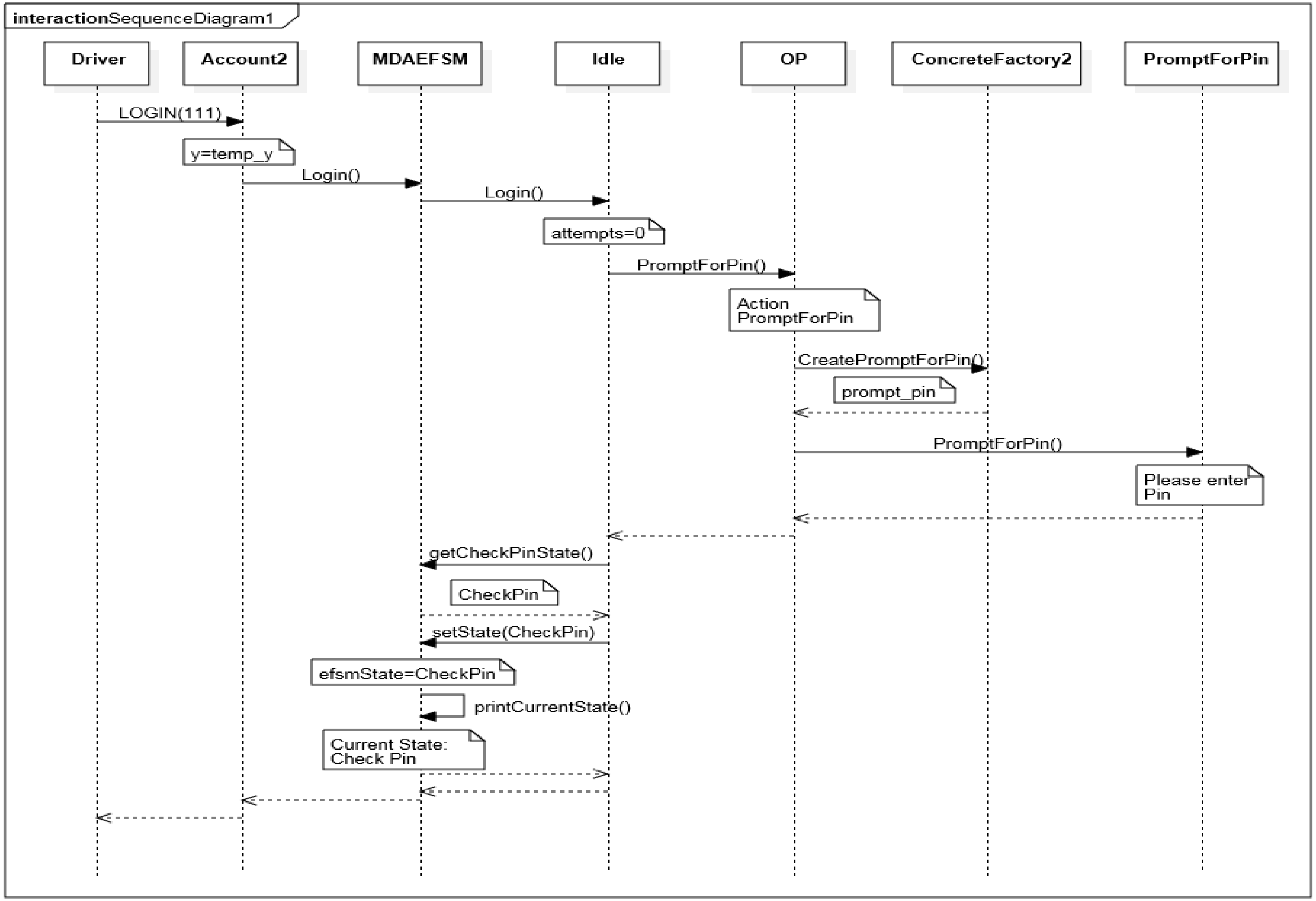
Operation: logout()



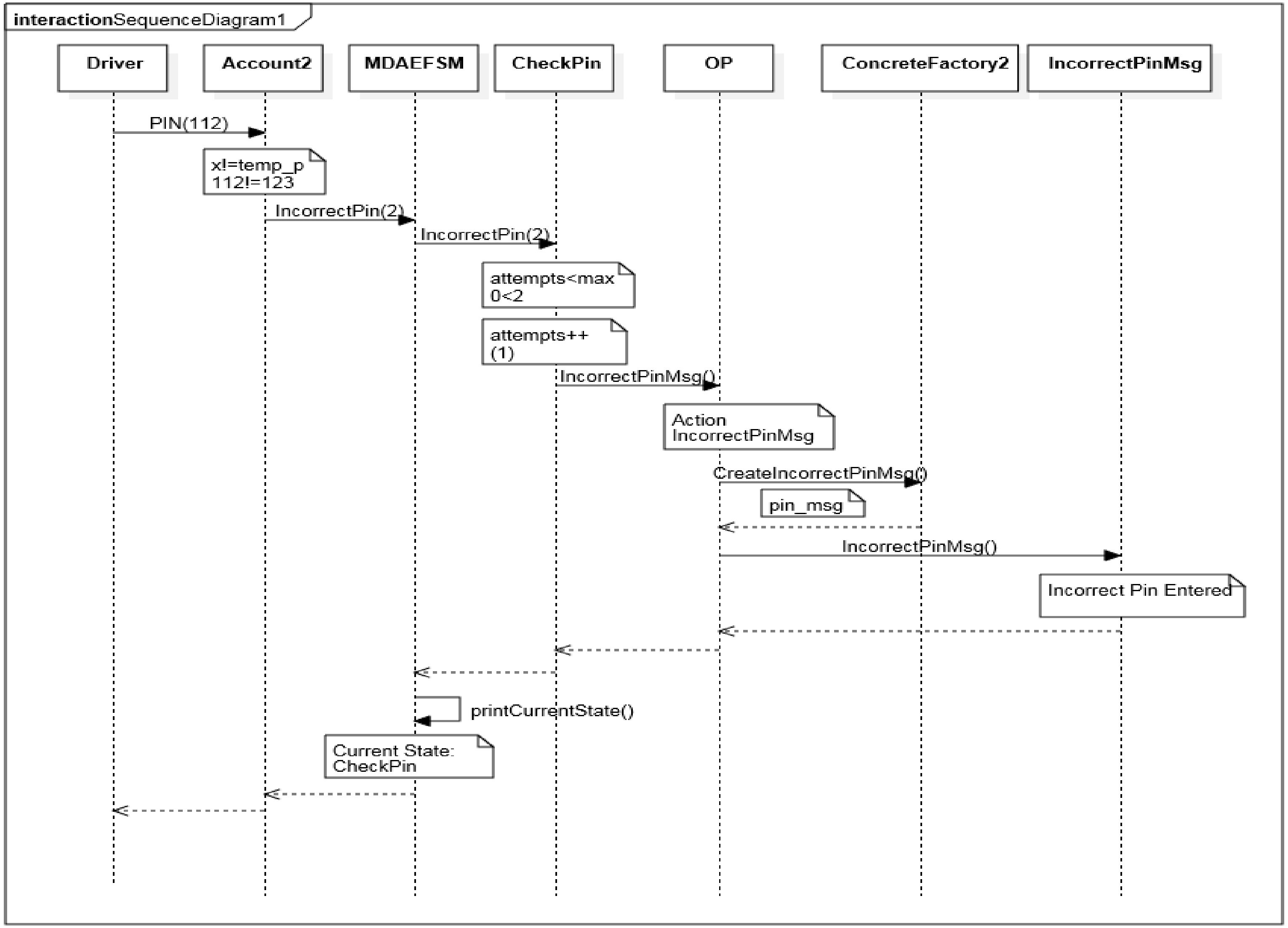
Scenario 2: Account 2
Operation: OPEN(123,111,1000)



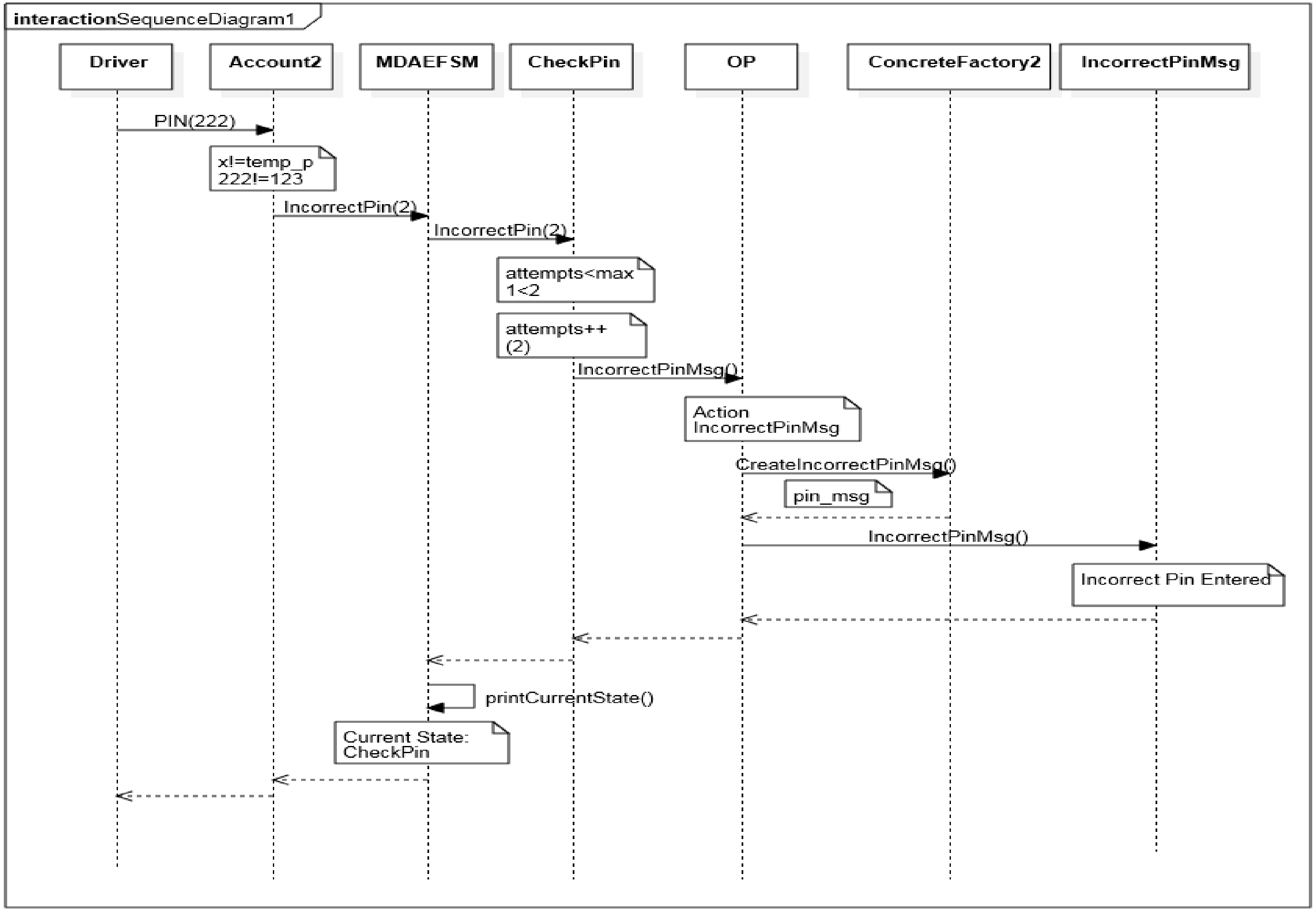
Operation: LOGIN(111)



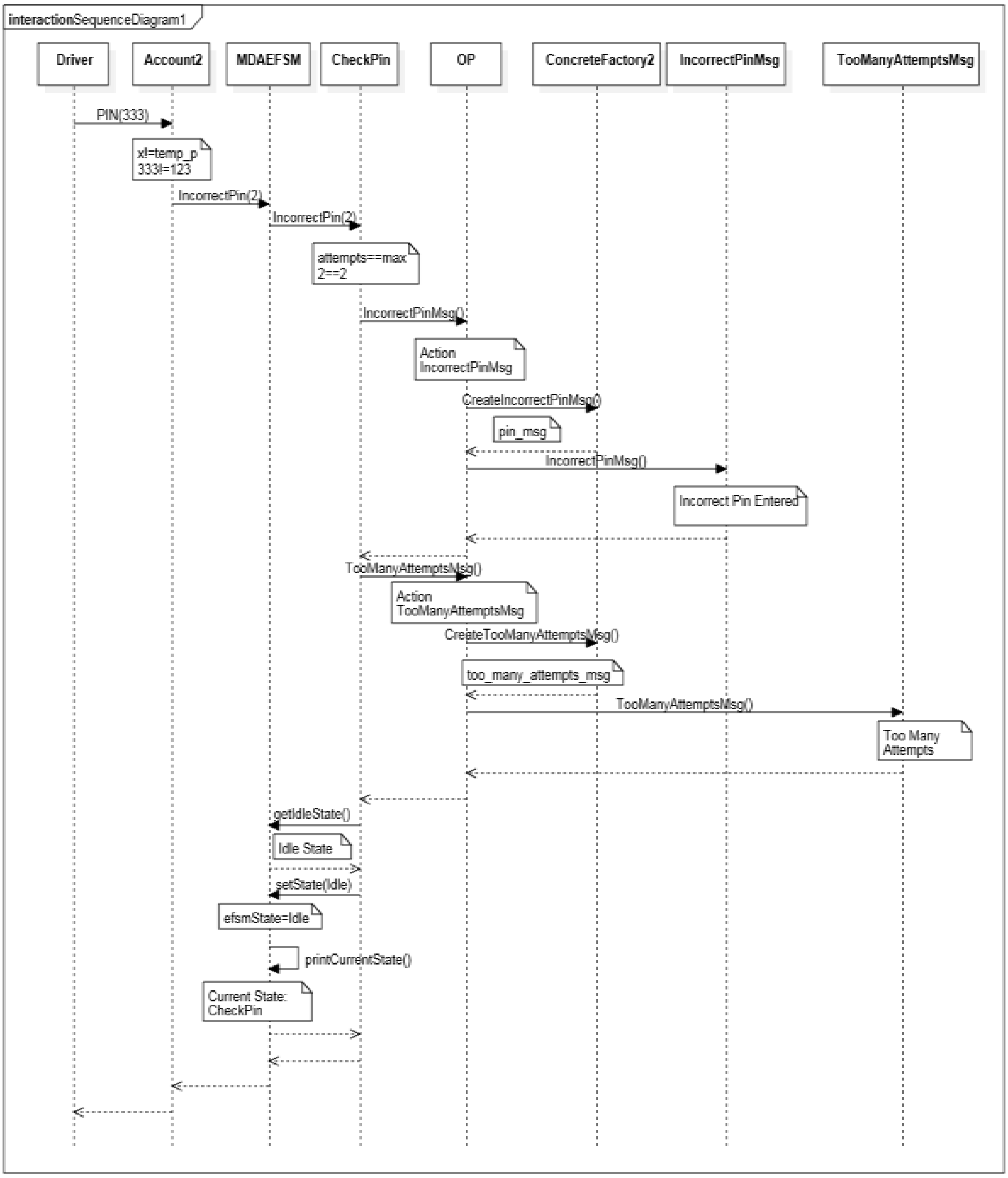
Operation: PIN(112)



Operation: PIN(222)



Operation: PIN(333)



5. Source Code and Patterns

The class are order in the following manner:

5.1 Driver – 1 Class

5.2 Accounts- 2 Classes

1. Account1
2. Account2

5.3 Data Store- 3 Classes

1. DataStore
2. DataStore1
3. DataStore2

5.4 Output Processor-1 Class

1. OP

5.5 State Pattern-10 Classes

1. MDAEFSM
2. State
3. Start
4. Idle
5. CheckPin
6. Ready
7. Locked
8. Suspended
9. S1
10. Overdrawn

5.6 Strategy Pattern-23 Classes

1. StoreData
2. StoreData1
3. StoreData2
4. IncorrectIdMsg
5. IncorrectPinMsg
6. TooManyAttemptsMsg
7. DisplayMenu
8. DisplayMenu1
9. DisplayMenu2
10. MakeDeposit
11. MakeDeposit1
12. Make Deposit2
13. DisplayBalance
14. DisplayBalance1
15. DisplayBalance2
16. PromptForPin
17. MakeWithdraw
18. MakeWithdraw1
19. MakeWithdraw2
20. Penalty
21. IncorrectLockMsg
22. IncorrectUnlockMsg
23. NoFundsMsg

5.7 Abstract Factory-3 Classes

1. AbstractFactory
2. ConcreteFactory1
3. ConcreteFactory2

5.1 Driver.java

```
/*
 Driver used for user input and is the main
 */
package project;
import java.io.*;
import MDAE.MDAEFSM;
import Output.OP;
import Accounts.*;
import Abstract_Factory.*;

public class Driver
{
   public static void main(String[] args) throws IOException
   {
          BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
          int input=0;
          int choice = 1;
          System.out.println("Select Account" );
          System.out.println("1. Account-1" );
          System.out.println("2. Account-2" );
          input=Integer.parseInt(br.readLine());
          if(input==1)
          {
                 ConcreteFactory1 factory = new ConcreteFactory1();
                 OP output = new OP(factory,factory.GetDataStore());
                 MDAEFSM mdaefsm = new MDAEFSM(factory,output);
                 Account1 ac1 = new Account1(mdaefsm,factory.GetDataStore());
                 String x,p,y;
                 float a,d,w;
                 System.out.println("ACCOUNT 1" );
                 System.out.println("Menu Of Operations" );
                 System.out.println("1. open(String p,String y,float a)" );
                 System.out.println("2. login(String y)" );
                 System.out.println("3. pin(String x)" );
                 System.out.println("4. logout()" );
                 System.out.println("5. balance()" );
                 System.out.println("6. deposit(float d)" );
                 System.out.println("7. withdraw(float w)" );
                 System.out.println("8. lock(String x)" );
                 System.out.println("9. unlock(String x)" );
                 System.out.println("10. Quit" );
                 while (true)
                 {
                        System.out.println("  Select Operation: ");
                        System.out.println("1-open,2-login,3-pin,4-logout,5-balance,6-deposit,7-
                        withdraw,8-lock,9-unlock");
                        input = Integer.parseInt(br.readLine());
                        if(input==0)
                        continue;
```

```
if(input==10)
break;
choice = input;
switch(choice)
{
        case 1:   //open
                System.out.println("Operation:  open(String p, String y, float
                a)");
                System.out.println("Enter value of the parameter p(PIN):");
                p =br.readLine();
                System.out.println("Enter value of the parameter y(User
                ID:");
                y = br.readLine();
                System.out.println("Enter value of the parameter
                a(Balance:");
                a = Float.parseFloat(br.readLine());
                ac1.open(p,y,a);
                break;

        case 2:  //login
                System.out.println("  Operation:  login(String y)");
                System.out.println("  Enter value of parameter y(User ID):");
                y = br.readLine();
                ac1.login(y);
                break;

        case 3:  //pin
                System.out.println("  Operation:  pin(String x)");
                System.out.println("  Enter value of parameter x(Pin):");
                x = br.readLine();
                ac1.pin(x);
                break;

        case 4:  //logout
                System.out.println("  Operation:  logout()");
                ac1.logout();
                break;

        case 5:  //balance
                System.out.println("  Operation:  balance()");
                ac1.balance();
                break;

        case 6:  //deposit
                System.out.println("  Operation:  deposit(float d)");
                System.out.println("  Enter value of the parameter
                d(deposit):");
                d = Float.parseFloat(br.readLine());
                ac1.deposit(d);
                break;
```

```
                    case 7:  // withdraw
                            System.out.println("  Operation:  withdraw(float w)");
                            System.out.println("  Enter value of the parameter
                            w(withdraw):");
                            w = Float.parseFloat(br.readLine());
                            ac1.withdraw(w);
                            break;

                    case 8:  // lock
                            System.out.println("  Operation:  lock(String x)");
                            System.out.println("  Enter value of the parameter x(Pin):");
                            x = br.readLine();
                            ac1.lock(x);
                            break;

                    case 9:  // unlock
                            System.out.println("  Operation:  unlock(String x)");
                            System.out.println("  Enter value of the parameter x(Pin):");
                            x = br.readLine();
                            ac1.unlock(x);
                            break;

                    case 10: //quit
                            System.out.println("Quiiting Account 1");

                    default:
                            System.out.println("Invalid Choice");
                            break;
            }
    }
    System.out.println("Thank you for using Account 1" );
    }
    else if(input==2)
    {
             ConcreteFactory2 factory = new ConcreteFactory2();
            OP output = new OP(factory,factory.GetDataStore());
            MDAEFSM mdaefsm = new MDAEFSM(factory,output);
            Account2 ac2 = new Account2(mdaefsm,factory.GetDataStore());
            System.out.println("Account 2" );
            int p,y,a,d,w,x;
            System.out.println("MENU of Operations" );
            System.out.println("1. OPEN(int p, int y,int a)" );
            System.out.println("2. LOGIN()" );
            System.out.println("3. PIN(int x)" );
            System.out.println("4. LOGOUT()" );
            System.out.println("5. BALANCE()" );
            System.out.println("6. DEPOSIT(int d)" );
            System.out.println("7. WITHDRAW(int w)" );
            System.out.println("8. suspend()" );
            System.out.println("9. activate()" );
            System.out.println("10. close()" );
```

```
System.out.println("11. Quit " );
while (true)
{
        System.out.println("  Select Operation: ");
        System.out.println("1-OPEN,2-LOGIN,3-PIN,4-LOGOUT,5-
        BALANCE,6-DEPOSIT,7-WITHDRAW,8-suspend,9-activate,10-close");
        input = Integer.parseInt(br.readLine());
        if(input==0)
                 continue;
        if(input==11)
                break;
        choice = input;
        switch(choice)
        {
                case 1:  //OPEN
                        System.out.println("\n  Operation:  OPEN(int p, int
                        y, int a)");
                        System.out.println("   Enter value of the parameter
                        p(pin):");
                        p = Integer.parseInt(br.readLine());
                        System.out.println("   Enter value of the parameter
                        y(user ID):");
                        y = Integer.parseInt(br.readLine());
                        System.out.println("   Enter value of the parameter
                        a(balance):");
                        a = Integer.parseInt(br.readLine());
                        ac2.OPEN(p,y,a);
                        break;

                case 2:  //LOGIN
                        System.out.println("\n  Operation:  LOGIN(int y)");
                        System.out.println("   Enter value of the parameter
                        y(user ID):");
                        y = Integer.parseInt(br.readLine());
                        ac2.LOGIN(y);
                        break;

                case 3: //PIN
                        System.out.println("  Operation:  PIN(int x)");
                        System.out.println("  Enter value of x(Pin):");
                        x = Integer.parseInt(br.readLine());
                        ac2.PIN(x);
                        break;

                case 4:  //LOGOUT
                        System.out.println("\n  Operation:  LOGOUT");
                        ac2.LOGOUT();
                        break;

                case 5: // BALANCE
                        System.out.println("  Operation:  BALANCE()");
                        ac2.BALANCE();
                        break;

                    case 6:  //DEPOSIT
                        System.out.println("  Operation:  DEPOSIT(int d)");
                        System.out.println("  Enter value of the parameter
                        d(Deposit):");
                        d = Integer.parseInt(br.readLine());
                        ac2.DEPOSIT(d);
                        break;

                    case 7:  // WITHDRAW
                        System.out.println("  Operation:  WITHDRAW(int
                        w)");
                        System.out.println("  Enter value of the parameter
                        w(Withdraw):");
                        w = Integer.parseInt(br.readLine());
                        ac2.WITHDRAW(w);
                        break;

                    case 8:  // suspend
                        System.out.println("  Operation:  suspend()");
                        ac2.suspend();
                        break;

                    case 9:  // activate
                        System.out.println("  Operation:  activate()");
                        ac2.activate();
                        break;

                    case 10:  // close
                        System.out.println("  Operation:  close()");
                        ac2.close();
                        break;

                    default:
                    System.out.println("Invalid Choice");
                    break;
            }
        }
        System.out.println("Thanks for using Account-2" );
    }
 }
}
```

5.2.1 Account1.java

```
/*
 Implementation for Account 1
 */
package Accounts;
import MDAE.MDAEFSM;
import Data_Store.DataStore;
import Data_Store.DataStore1;
public class Account1
{
        /* MDAEFSM object (Pointer to MDAEFSM) */
        MDAEFSM m = null;
        /* DataStore object (Pointer to DataStore) */
        DataStore ds = null;

        public Account1(MDAEFSM m,DataStore ds)
        {
                this.m = m;
                this.ds = ds;
        }
        /*implementations for the functions belonging to Account 1*/
        public void open(String p, String y, float a)
        {
                ((DataStore1)ds).temp_p = p; //stores the value for pin
                ((DataStore1)ds).temp_y = y; //stores the value for user id
                ((DataStore1)ds).temp_a = a; //stores the value for balance
                m.Open();
        }
        public void pin( String x )
        {
                /*check if the entered pin matches the stored value for pin*/
                if( x.equals(((DataStore1)ds).temp_p ) )
                {
                        /*check if the stored balance is greater than 500 and call either of the 2
                        methods*/
                        if( ((DataStore1)ds).temp_a > 500 )
                                m.CorrectPinAboveMin();
                        else
                                m.CorrectPinBelowMin();
                }
                else
                        m.IncorrectPin(3);
        }
        public void deposit(float d)
        {
                ((DataStore1)ds).temp_d = d;//store the input in temp_d
                m.Deposit();
                /*check the condition and call either operation*/
                if( ((DataStore1)ds).temp_a > 500 )
                        m.AboveMinBalance();
                else
                        m.BelowMinBalance();
        }
        public void withdraw(float w)
        {
                ((DataStore1)ds).temp_w = w;//store the input in temp_d
                m.Withdraw();
        /*check the condition and call either operation*/
        if( ((DataStore1)ds).temp_a > 500 )
            m.AboveMinBalance();
        else
            m.WithdrawBelowMinBalance();
    }
    public void balance()
    {
        m.Balance();
    }
    public void login(String y)
    {
        /*call Login() only if the entered pin and stored pin match*/
        if(y.equals(((DataStore1)ds).temp_y))
        m.Login();
        else
        m.IncorrectLogin();
    }
    public void logout()
    {
        m.Logout();
    }
    public void lock(String x)
    {
        /*calls Lock() only if the enter pin matches x*/
        if(x.equals(((DataStore1)ds).temp_p))
            m.Lock();
        else
            m.IncorrectLock();
    }
    public void unlock(String x)
    {
        /*calls Unlock() only if the enter pin matches x*/
        if(x.equals(((DataStore1)ds).temp_p))
        {
            m.Unlock();
            if(((DataStore1)ds).balance > 500)
                m.AboveMinBalance();
            else
                m.BelowMinBalance();
        }
        else
            m.IncorrectUnlock();
    }
}
```

5.2.2 Account2.java

```
/*
 Implementation for Account 2
 */
package Accounts;
import MDAE.MDAEFSM;
import Data_Store.DataStore;
import Data_Store.DataStore2;
public class Account2
{
   /* MDAEFSM object (Pointer to MDAEFSM) */
   MDAEFSM m = null;
   /* Pointer to DataStore */
   DataStore ds = null;
   public Account2(MDAEFSM m,DataStore ds)
   {
          this.m = m;
          this.ds = ds;
   }
   /*implementations for the functions belonging to Account 2*/
   public void OPEN(int p, int y, int a)
   {
          ((DataStore2)ds).temp_p = p;//stores the value for pin
          ((DataStore2)ds).temp_y = y;//stores the value for user id
          ((DataStore2)ds).temp_a = a;//stores the value for balance
          m.Open();
   }
   public void PIN(int x)
   {
          /*check if the entered pin matches the stored value for pin*/
          if(x == ((DataStore2)ds).temp_p)
                 m.CorrectPinAboveMin();
          else
                 m.IncorrectPin(2);
   }
   public void DEPOSIT(int d)
   {
          ((DataStore2)ds).temp_d = d;//store the input in temp_d
          m.Deposit();
   }
   public void WITHDRAW(int w)
   {
          ((DataStore2)ds).temp_w = w;//store the input in temp_w
          if(((DataStore2)ds).temp_a > 0)//check if balance>0
          {
                 m.Withdraw();
                 m.AboveMinBalance();
          }
          else
                  m.NoFunds();
   }
   public void BALANCE()
   {
          m.Balance();
   }
   public void LOGIN(int y)
   {
```

```
            /*call Login() only if the entered pin and stored pin match*/
            if(y==((DataStore2)ds).temp_y)
            m.Login();
            else
            m.IncorrectLogin();
   }
   public void LOGOUT()
   {
             m.Logout();
   }
   public void suspend()
   {
            m.Suspend();
   }
   public void activate()
   {
            m.Activate();
   }
   public void close()
   {
            m.Close();
   }
}
```

5.3.1 DataStore.java

```
/*
Abstract Class for DataStore
 */
package Data_Store;
public class DataStore
{
   //Abstract Class
}
```

5.3.2 DataStore1.java

```
/*
DataStore for maintaining values of Account 1
*/
package Data_Store;
public class DataStore1 extends DataStore
{
       /*Temporary variables*/
            public String temp_p;//for Pin
            public String temp_y;//for Uid
            public float temp_a;//for balance
            public float temp_d;//for deposit
            public float temp_w;//for withdraw

            /*Permanent Variables */
            public String pin;  // for Pin
            public String uid; //for Uid
            public float balance;//for balance
            public float d;   // for Deposit
            public float w;   // for Withdraw

            /*Methods to set the pin value and return it*/
            public String setPin()
            {
                    return this.pin = this.temp_p;
            }
            public String getPin()
            {
                    return this.pin;
            }
            /*Methods to set the uid value and return it*/
            public String setUid()
            {
                    return this.uid = this.temp_y;
            }
            public String getUid()
            {
                    return this.uid;
            }
            /*Methods to set the balance value and return it*/
            public float setBalance()
            {
                    return this.balance = this.temp_a;
            }
            public float getBalance()
            {
                    return this.balance;
            }
            /*Methods to set the deposit value and return it*/
            public void setDeposit()
            {
                    this.d = this.temp_d;
            }
            public float getDeposit()
            {
                    return this.d;
            }
            /*Methods to set the withdraw value and return it*/
            public void setWithdraw()
            {
                    this.w = this.temp_w;
            }
            public float getWithdraw()
            {
                    return this.w;
            }
            /*Methods to set the penalty value and subtract penalty from the balance*/
```

```
	public float setPenalty()
	{
		this.balance = this.balance - 20;
		this.temp_a = balance;
		return this.balance;
	}
	/*Methods to compute the balance value after deposit*/
	public void computeBalanceAfterDeposit()
	{
		this.balance = this.balance + this.d;
		this.temp_a = this.balance;
	}
	/*Methods to compute the balance value after withdraw*/
	public void computeBalanceAfterWithdraw()
	{
		this.balance = this.balance - this.w;
		this.temp_a = this.balance;
	}
}
```

5.3.3 DataStore2.java

```
/*
DataStore for maintaining values of Account 2
 */
package Data_Store;
public class DataStore2 extends DataStore
{
	/* Temporary variables */
	public int temp_p;//for Pin
	public int temp_y;//for User ID
	public int temp_a;//for balance
	public int temp_d;//for deposit
	public int temp_w;//for withdraw

	/* Permanent Variables */
	public int pin;  // for Pin
	public int uid; //for Uid
	public int balance;// for balance
	public int d; // for deposit
	public int w; // for withdraw

	/*Methods to set the pin value and return it*/
	public int setPin()
	{
		return this.pin = this.temp_p;
	}
	public int getPin()
	{
		return this.pin;
	}
```

```
        /*Methods to set the uid value and return it*/
        public int setUid()
        {
                return this.uid = this.temp_y;
        }
        public int getUid()
        {
                return this.uid;
        }
        /*Methods to set the balance value and return it*/
        public int setBalance()
        {
                return this.balance = this.temp_a;
        }
        public int getBalance()
        {
                return this.balance;
        }
        /*Methods to set the deposit value and return it*/
        public void setDeposit()
        {
                this.d = this.temp_d;
        }
        public int getDeposit()
        {
                return this.d;
        }
        /*Methods to set the withdraw value and return it*/
        public void setWithdraw()
        {
                this.w = this.temp_w;
        }
        public int getWithdraw()
        {
                return this.w;
        }
        /*Methods to compute the balance value after deposit*/
        public void computeBalanceAfterDeposit()
        {
                this.balance = this.balance + this.d;
                this.temp_a = this.balance;
        }
        /*Methods to compute the balance value after withdraw*/
        public void computeBalanceAfterWithdraw()
        {
                this.balance = this.balance - this.w;
                this.temp_a = this.balance;
        }
}
```

5.4.1 OP.java

```
/*
 The output processor class for the actions
 */
package Output;
import Data_Store.*;
import Abstract_Factory.*;
import Strategy.*;
public class OP
{
   AbstractFactory af =null;
   DataStore ds = null;
   public OP(AbstractFactory af,DataStore ds)
   {
          this.af = af;
          this.ds = ds;
   }
   public void StoreData()
   {
          System.out.println("Action StoreData");
          StoreData store_d = af.CreateStoreData();
          store_d.StoreData(ds);
   }
   public void IncorrectIdMsg()
   {
          System.out.println("Action IncorrectIdMsg");
          IncorrectIdMsg id_msg = af.CreateIncorrectIdMsg();
          id_msg.IncorrectIdMsg();
   }
   public void IncorrectPinMsg()
   {
          System.out.println("Action IncorrectPinMsg");
          IncorrectPinMsg pin_msg = af.CreateIncorrectPinMsg();
          pin_msg.IncorrectPinMsg();
   }
   public void TooManyAttemptsMsg()
   {
          System.out.println("Action TooManyAttemptsMsg");
          TooManyAttemptsMsg too_many_attempts = af.CreateTooManyAttemptsMsg();
          too_many_attempts.TooManyAttemptsMsg();
   }


   public void DisplayMenu()
   {
          System.out.println("Action DisplayMenu");
          DisplayMenu disp_menu = af.CreateDisplayMenu();
          disp_menu.DisplayMenu();
   }
   public void MakeDeposit()
```

```
    {
            System.out.println("Action MakeDeposit ");
            MakeDeposit make_deposit = af.CreateMakeDeposit();
            make_deposit.MakeDeposit(ds);
    }
    public void DisplayBalance()
    {
            System.out.println("Action DisplayBalance ");
            DisplayBalance disp_bal = af.CreateDisplayBalance();
            disp_bal.DisplayBalance(ds);
    }
    public void PromptForPin()
    {
            System.out.println("Action PromptForPin ");
            PromptForPin prompt_pin = af.CreatePromptForPin();
            prompt_pin.PromptForPin();
    }
    public void MakeWithdraw()
    {
            System.out.println("Action MakeWithdraw  ");
            MakeWithdraw make_withdraw = af.CreateMakeWithdraw();
            make_withdraw.MakeWithdraw(ds);
    }
    public void Penalty()
    {
            System.out.println("Action Penalty  ");
            Penalty penalty = af.CreatePenalty();
            penalty.Penalty(ds);
    }
    public void IncorrectLockMsg()
    {
            System.out.println("Action IncorrectLockMsg");
            IncorrectLockMsg lock_msg = af.CreateIncorrectLockMsg();
            lock_msg.IncorrectLockMsg();
    }
    public void IncorrectUnlockMsg()
    {
            System.out.println("Action IncorrectUnlockMsg");
            IncorrectUnlockMsg unlcok_msg = af.CreateIncorrectUnlockMsg();
            unlcok_msg.IncorrectUnlockMsg();
    }
    public void NoFundsMsg()
    {
            System.out.println("Action NoFundsMsg");
            NoFundsMsg no_funds = af.CreateNoFundsMsg();
            no_funds.NoFundsMsg();
    }
}
```

5.5.1 MDAEFSM.java (State Pattern)

```
/*
 MDAEFSM maintains all the events used for performing actions
 */
package MDAE;
import Abstract_Factory.*;
import Output.*;
public class MDAEFSM
{
	/*State Objects*/
	State startState = new Start(this);
	State idleState = new Idle(this);
	State checkpinState = new CheckPin(this);
	State readyState = new Ready(this);
	State lockedState = new Locked(this);
	State overdrawnState = new Overdrawn(this);
	State suspenedState = new Suspended(this);
	State s1State = new S1(this);
	State efsmState = null;
	public int attempts;
	AbstractFactory af =null;
	OP op = null;

	public MDAEFSM(AbstractFactory af,OP op)
	{
		efsmState = startState;
		attempts = 0;
		this.af = af;
		this.op = op;
	}
	public void Open()
	{
		efsmState.Open();
		printCurrentState();
	}
	public void Login()
	{
		efsmState.Login();
		attempts = 0;
		printCurrentState();
	}
	public void IncorrectLogin()
	{
		efsmState.IncorrectLogin();
		printCurrentState();
	}
	public void IncorrectPin(int max)
	{
		efsmState.IncorrectPin(max);
		printCurrentState();
}
public void CorrectPinBelowMin()
{
        efsmState.CorrectPinBelowMin();
        printCurrentState();
}
public void CorrectPinAboveMin()
{
        efsmState.CorrectPinAboveMin();
        printCurrentState();
}
public void Deposit()
{
        efsmState.Deposit();
        printCurrentState();
}
public void BelowMinBalance()
{
        efsmState.BelowMinBalance();
        printCurrentState();
}
public void AboveMinBalance()
{
        efsmState.AboveMinBalance();
        printCurrentState();
}
public void Logout()
{
        efsmState.Logout();
        printCurrentState();
}
public void Balance()
{
        efsmState.Balance();
        printCurrentState();
}
public void Withdraw()
{
        efsmState.Withdraw();
        printCurrentState();
}
public void WithdrawBelowMinBalance()
{
        efsmState.WithdrawBelowMinBalance();
        printCurrentState();
}
public void NoFunds()
{
        efsmState.NoFunds();
        printCurrentState();
}
```

```
public void Lock()
{
            efsmState.Lock();
            printCurrentState();
}
public void IncorrectLock()
{
            efsmState.IncorrectLock();
            printCurrentState();
}
public void Unlock()
{
            efsmState.Unlock();
            printCurrentState();
}
public void IncorrectUnlock()
{
            efsmState.IncorrectUnlock();
            printCurrentState();
}
public void Suspend()
{
            efsmState.Suspend();
            printCurrentState();
}
public void Activate()
 {
            efsmState.Activate();
            printCurrentState();
}
public void Close()
{
            efsmState.Close();
            printCurrentState();
}
/*set the current state*/
public void setState(State efsmState)
{
            this.efsmState = efsmState;
}
public State getStartState()
{
            return startState;
}
public State getIdleState()
{
            return idleState;
}
public State getCheckPinState()
{
            return checkpinState;
```

```
		}
		public State getReadyState()
		{
				return readyState;
		}
		public State getS1State()
		{
				return s1State;
		}
		public State getLockedState()
		{
				return lockedState;
		}
		public State getOverdrawnState()
		{
				return overdrawnState;
		}
		public State getSuspendedState()
		{
				return suspenedState;
		}
		public void printCurrentState()
		{
				System.out.println("Current State : "+ efsmState.getClass().getName());
		}
}
```

5.5.2 State.java (State Pattern)

```
/*
 Interface class for state
 */
package MDAE;
public interface State
{
   public void Open();
   public void Login();
   public void IncorrectLogin();
   public void IncorrectPin(int max);
   public void CorrectPinBelowMin();
   public void CorrectPinAboveMin();
   public void Deposit();
   public void BelowMinBalance();
   public void AboveMinBalance();
   public void Logout();
   public void Balance();
   public void Withdraw();
   public void WithdrawBelowMinBalance();
   public void NoFunds();
   public void Lock();
   public void IncorrectLock();
   public void Unlock();
   public void IncorrectUnlock();
   public void Suspend();
   public void Activate();
   public void Close();
}
```

5.5.3 Start.java(State Pattern)

```
/*
 To perform operation when the Machine is in CheckPin State
 */
package MDAE;
public class Start implements State
{
            MDAEFSM m=null;
            public Start(MDAEFSM m)
            {
            this.m =  m;
            }
            public void Open()
            {
                        m.op.StoreData();
                        m.setState(m.getIdleState());
            }
            public void Login()
            {
            }
            public void IncorrectLogin()
            {
            }
            public void IncorrectPin(int max)
            {
            }
            public void CorrectPinBelowMin()
            {
            }
            public void CorrectPinAboveMin()
            {
            }
            public void Deposit()
            {
            }
            public void BelowMinBalance()
            {
            }
            public void AboveMinBalance()
            {
            }
            public void Logout()
            {
            }
            public void Balance()
            {
            }
            public void Withdraw()
            {
            }
            public void WithdrawBelowMinBalance()
            {
            }
            public void NoFunds()
            {
            }
            public void Lock()
            {
            }
            public void IncorrectLock()
            {
            }
            public void Unlock()
            {
            }
            public void IncorrectUnlock()
            {
            }
            public void Suspend()
            {
            }
            public void Activate()
            {
            }
            public void Close()
            {
            }
}
```

5.5.4 Idle.java(State Pattern)

```
/*
 To perform operation when the Machine is in Idle State
 */
package MDAE;
public class Idle implements State
{
        MDAEFSM m=null;
        public Idle(MDAEFSM m)
        {
                this.m =  m;
        }
        public void Open()
        {
        }
        public void Login()
        {
                m.attempts = 0;
                m.op.PromptForPin();
                m.setState(m.getCheckPinState());
        }
        public void IncorrectLogin()
        {
                m.op.IncorrectIdMsg();
        }
        public void IncorrectPin(int max)
        {
        }
        public void CorrectPinBelowMin()
        {
        }
        public void CorrectPinAboveMin()
        {
        }
        public void Deposit()
        {
        }
        public void BelowMinBalance()
        {
        }
        public void AboveMinBalance()
        {
        }
        public void Logout()
        {
        }
        public void Balance()
        {
        }
        public void Withdraw()
        {
        }
        public void WithdrawBelowMinBalance()
        {
        }
        public void NoFunds()
        {
        }
        public void Lock()
        {
        }
        public void IncorrectLock()
        {
        }
        public void Unlock()
```

```
    {
    }
    public void IncorrectUnlock()
    {
    }
    public void Suspend()
    {
    }
    public void Activate()
    {
    }
    public void Close()
    {
    }
}
```

5.5.5 CheckPinState.java (State Pattern)

```
/*
 To perform operation when the Machine is in CheckPin State
 */
public class CheckPin implements State
{
    MDAEFSM m=null;
    public CheckPin(MDAEFSM m)
    {
        this.m =  m;
    }
    public void Open()
    {
    }
    public void Login()
    {
    }
    public void IncorrectLogin()
    {
    }
    public void IncorrectPin(int max)
    {
        //check conditions provided in the EFSM
        if( m.attempts < max )
        {
            m.attempts++;
            m.op.IncorrectPinMsg();
        }
        else if( m.attempts == max )
        {
            m.op.IncorrectPinMsg();
            m.op.TooManyAttemptsMsg();
            m.setState(m.getIdleState());//change the state and set it
        }
}
public void CorrectPinBelowMin()
{
                        m.op.DisplayMenu();
                        m.setState(m.getOverdrawnState());
}
public void CorrectPinAboveMin()
{
            m.op.DisplayMenu();
            m.setState(m.getReadyState());
}
public void Deposit()
{
}
public void BelowMinBalance()
{
}
public void AboveMinBalance()
{
}
public void Logout()
{
            m.setState(m.getIdleState());
}
public void Balance()
{
}
public void Withdraw()
{
}
public void WithdrawBelowMinBalance()
{
}
public void NoFunds()
{
}
public void Lock()
{
}
public void IncorrectLock()
{
}
public void Unlock()
{
}
public void IncorrectUnlock()
{
}
public void Suspend()
{
}
```

```
	public void Activate()
	{
	}
	public void Close()
	{
	}
}
```

5.5.6 Ready.java (State Pattern)

```
/*
 To perform operation when the Machine is in Ready State
 */
package MDAE;
public class Ready implements State
{
	MDAEFSM m=null;
	public Ready(MDAEFSM m)
	{
		this.m =  m;
	}
	public void Open()
	{
	}
	public void Login()
	{
	}
	public void IncorrectLogin()
	{
	}
	public void IncorrectPin(int max)
	{
	}
	public void CorrectPinBelowMin()
	{
	}
	public void CorrectPinAboveMin()
	{
	}
	public void Deposit()
	{
		m.op.MakeDeposit();
	}
	public void BelowMinBalance()
	{
	}
	public void AboveMinBalance()
	{
	}
	public void Logout()
	{
        m.setState(m.getIdleState());
    }
    public void Balance()
    {
        m.op.DisplayBalance();
    }
    public void Withdraw()
    {
        m.op.MakeWithdraw();
        m.setState(m.getS1State());
    }
    public void WithdrawBelowMinBalance()
    {
    }
    public void NoFunds()
    {
        m.op.NoFundsMsg();
    }
    public void Lock()
    {
        m.setState(m.getLockedState());
    }
    public void IncorrectLock()
    {
        m.op.IncorrectLockMsg();
    }
    public void Unlock()
    {
    }
    public void IncorrectUnlock()
    {
    }
    public void Suspend()
    {
        m.setState(m.getSuspendedState());
    }
    public void Activate()
    {
    }
    public void Close()
    {
    }
}
```

5.5.7 Locked.java (State Pattern)

```
/*
 To perform operation when the Machine is in Locked State
 */
```

```
package MDAE;
public class Locked implements State
{
        MDAEFSM m=null;
        public Locked(MDAEFSM m)
        {
                this.m =  m;
        }
        public void Open()
        {
        }
        public void Login()
        {
        }
        public void IncorrectLogin()
        {
        }
        public void IncorrectPin(int max)
        {
        }
        public void CorrectPinBelowMin()
        {
        }
        public void CorrectPinAboveMin()
        {
        }
        public void Deposit()
        {
        }
        public void BelowMinBalance()
        {
        }
        public void AboveMinBalance()
        {
        }
        public void Logout()
        {
        }
        public void Balance()
        {
        }
        public void Withdraw()
        {
        }
        public void WithdrawBelowMinBalance()
        {
        }
        public void NoFunds()
        {
        }
        public void Lock()
```

```
    {
    }
    public void IncorrectLock()
    {
    }
    public void Unlock()
    {
        m.setState(m.getS1State());
    }
    public void IncorrectUnlock()
    {
        m.op.IncorrectUnlockMsg();
    }
    public void Suspend()
    {
    }
    public void Activate()
    {
    }
    public void Close()
    {
    }
}
```

5.5.8 Suspended.java (State Pattern)

```
/*
 To perform operation when the Machine is in Suspended State
 */
package MDAE;
public class Suspended implements State
{
    MDAEFSM m=null;
    public Suspended(MDAEFSM m)
    {
        this.m =  m;
    }
    public void Open()
    {
    }
    public void Login()
    {
    }
    public void IncorrectLogin()
    {
    }
    public void IncorrectPin(int max)
    {
    }
    public void CorrectPinBelowMin()
    {
}
public void CorrectPinAboveMin()
{
}
public void Deposit()
{
}
public void BelowMinBalance()
{
}
public void AboveMinBalance()
{
}
public void Logout()
{
}
public void Balance()
{
		m.op.DisplayBalance();
}
public void Withdraw()
{
}
public void WithdrawBelowMinBalance()
{
}
public void NoFunds()
{
}
public void Lock()
{
}
public void IncorrectLock()
{
}
public void Unlock()
{
}
public void IncorrectUnlock()
{
}
public void Suspend()
{
}
public void Activate()
{
		m.setState(m.getReadyState());
}
public void Close()
{
		System.out.println("Account Closed");
```

```
			System.exit(0);
		}
}
```

5.5.9 S1.java (StatePattern)

```
/*
 To perform operation when the Machine is in S1 State
 */
package MDAE;
public class S1 implements State
{
	MDAEFSM m=null;
	public S1(MDAEFSM m)
	{
		this.m =  m;
	}
	public void Open()
	{
	}
	public void Login()
	{
	}
	public void IncorrectLogin()
	{
	}
	public void IncorrectPin(int max)
	{
	}
	public void CorrectPinBelowMin()
	{
	}
	public void CorrectPinAboveMin()
	{
	}
	public void Deposit()
	{
	}
	public void BelowMinBalance()
	{
		m.setState(m.getOverdrawnState());
	}
	public void AboveMinBalance()
	{
		m.setState(m.getReadyState());
	}
	public void Logout()
	{
	}
	public void Balance()
	{
	}
	public void Withdraw()
	{
	}
	public void WithdrawBelowMinBalance()
	{
		m.op.Penalty();
		m.setState(m.getOverdrawnState());
	}
	public void NoFunds()
	{
	}
	public void Lock()
	{
	}
	public void IncorrectLock()
	{
	}
	public void Unlock()
	{
	}
	public void IncorrectUnlock()
	{
	}
	public void Suspend()
	{
	}
	public void Activate()
	{
	}
	public void Close()
	{
	}
}
```

5.5.10 Overdrawn.java (State Pattern)

```
/*
 To perform operation when the Machine is in Overdrawn State
 */
package MDAE;
public class Overdrawn implements State
{
	MDAEFSM m=null;
	public Overdrawn(MDAEFSM m)
	{
		this.m =  m;
	}
	public void Open()
	{
	}
```

```
public void Login()
{
}
public void IncorrectLogin()
{
}
public void IncorrectPin(int max)
{
}
public void CorrectPinBelowMin()
{
}
public void CorrectPinAboveMin()
{
}
public void Deposit()
{
        m.op.MakeDeposit();
        m.setState(m.getS1State());
}
public void BelowMinBalance()
{
}
public void AboveMinBalance()
{
}
public void Logout()
{
        m.setState(m.getIdleState());
}
public void Balance()
{
        m.op.DisplayBalance();
}
public void Withdraw()
{
        m.op.NoFundsMsg();
}
public void WithdrawBelowMinBalance()
{
}
public void NoFunds()
{
}
public void Lock()
{
        m.setState(m.getLockedState());
}
public void IncorrectLock()
{
}
```

```
        public void Unlock()
        {
        }
        public void IncorrectUnlock()
        {
        }
        public void Suspend()
        {
        }
        public void Activate()
        {
        }
        public void Close()
        {
        }
}
```

5.6.1 StoreData.java (Strategy Pattern)

```
/*
 Abstract Class for StoreData action
 */
package Strategy;
import Data_Store.*;
public abstract class StoreData
{
        public abstract void StoreData(DataStore ds);
}
```

5.6.2 StoreData1.java(Strategy Pattern)

```
/*
 StorreData for Account 1
 */
package Strategy;
import Data_Store.*;
public class StoreData1 extends StoreData
{
        public void StoreData(DataStore ds)
        {
                /*store and return values for Pin, User Id and Balance*/
                ((DataStore1)ds).setPin();
                System.out.println("Account 1:The set PIN is " + ((DataStore1)ds).getPin() );
                ((DataStore1)ds).setUid();
                System.out.println("Account 1:The set User ID is " + ((DataStore1)ds).getUid() );
                ((DataStore1)ds).setBalance();
                System.out.println("Account 1:The Balance is " + ((DataStore1)ds).getBalance() );
        }
}
```

5.6.3 StoreData2.java (Strategy Pattern)

```
/*
 StoreData for Account 2
 */
package Strategy;
import Data_Store.*;
public class StoreData2 extends StoreData
{
	public void StoreData(DataStore ds)
	{
		/*store and return values for Pin, User Id and Balance*/
		((DataStore2)ds).setPin();
		System.out.println("Account 2:The set PIN is " + ((DataStore2)ds).getPin() );
		((DataStore2)ds).setUid();
		System.out.println("Account 2:The set User ID is " + ((DataStore2)ds).getUid() );
		((DataStore2)ds).setBalance();
		System.out.println("Account 2:The Balance is " + ((DataStore2)ds).getBalance() );
	}
}
```

5.6.4 IncorrectIdMsg.java (Strategy Pattern)

```
/*
 IncorrectIdMsg for both accounts
 */
package Strategy;
public class IncorrectIdMsg
{
	public void IncorrectIdMsg()
	{
		System.out.println("Incorrect Id entered.");
	}
}
```

5.6.5 IncorrectPinMsg.java (Strategy Pattern)

```
/*
 prints Incorrect Pin Msg
 */
package Strategy;
public class IncorrectPinMsg
{
	public void IncorrectPinMsg()
	{
		System.out.println("Incorrect Pin Entered");
	}
}
```

5.6.6 TooManyAttemptsMsg.java (Strategy Pattern)

```
/*
 Displays TooManyAttemptsMsg
 */
package Strategy;
public class TooManyAttemptsMsg
{
	public void TooManyAttemptsMsg()
	{
		System.out.println("Too Many Attempts");
	}
}
```

5.6.7 DisplayMenu.java (Strategy Pattern)

```
/*
 Abstract class for DisplayMenu action
 */
package Strategy;
public abstract class DisplayMenu
{
	public abstract void DisplayMenu();
}
```

5.6.8 DisplayMenu1.java (Strategy Pattern)

```
/*
 DisplayMenu for Account 1
 */
package Strategy;
public class DisplayMenu1 extends DisplayMenu
{
	public void DisplayMenu()
	{
		System.out.println("Account 1:");
		System.out.println("You can choose any of the following:");
		System.out.println("Deposit");
		System.out.println("Balance");
		System.out.println("Withdraw");
		System.out.println("Lock");
		System.out.println("Logout");
	}
}
```

5.6.9 DisplayMenu2.java (Strategy Pattern)

```
/*
 DisplayMenu for Account 2
 */
package Strategy;
```

```
public class DisplayMenu2 extends DisplayMenu
{
        public void DisplayMenu()
        {
                System.out.println("Account 2:");
                System.out.println("You can choose any of the following:");
                System.out.println("Deposit");
                System.out.println("Balance");
                System.out.println("Withdraw");
                System.out.println("Suspend");
                System.out.println("Logout");
        }
}
```

5.6.10 MakeDeposit.java (Strategy Pattern)

```
/*
 Abstract class for Make Deposit
 */
package Strategy;
import Data_Store.*;
public abstract class MakeDeposit
{
        public abstract void MakeDeposit(DataStore ds);
}
```

5.6.11 MakeDeposit1.java (Strategy Pattern)

```
/*
 MakeDeposit action for Account 1
 */
package Strategy;
import Data_Store.*;
public class MakeDeposit1 extends MakeDeposit
{
        public void MakeDeposit(DataStore ds)
        {
                /*sets the deposit amount in the datastore for Account1, and calculates and returns
                the final balance after deposit*/
                ((DataStore1)ds).setDeposit();
                ((DataStore1)ds).computeBalanceAfterDeposit();
                System.out.println("Account 1: After Deposit, Balance is " +
                ((DataStore1)ds).getBalance() );
        }
}
```

5.6.12 MakeDeposit2.java (Strategy Pattern)

```
/*
MakeDeposit for Account 2
 */
package Strategy;
import Data_Store.*;
public class MakeDeposit2 extends MakeDeposit
{
	public void MakeDeposit(DataStore ds)
	{
		/*sets the deposit amount in the datastore for Account2, and calculates and returns
		the final balance after deposit*/
		((DataStore2)ds).setDeposit();
		((DataStore2)ds).computeBalanceAfterDeposit();
		System.out.println("Account 2: After Deposit, Balance is " +
		((DataStore2)ds).getBalance() );
	}
}
```

5.6.13 DisplayBalance.java (Strategy Pattern)

```
/*
 Abstact class for the DisplayBalance Action
 */
package Strategy;
import Data_Store.*;
public abstract class DisplayBalance
{
	public abstract void DisplayBalance(DataStore ds);
}
```

5.6.14 DisplayBalance1.java (Strategy Pattern)

```
/*
 DisplayBalance for Account 1
 */
package Strategy;
import Data_Store.*;
public class DisplayBalance1 extends DisplayBalance
{
	public void DisplayBalance(DataStore ds)
	{
		 /*prints the current balance*/
		System.out.println("Account 1: Balance is " + ((DataStore1)ds).getBalance() );
	}
}
```

5.6.15 DisplayBalance2.java (Strategy Pattern)

```
/*
 DisplayBalance for Account 2
 */

package Strategy;
import Data_Store.*;
public class DisplayBalance2 extends DisplayBalance
{
	public void DisplayBalance(DataStore ds)
	{
		/*prints the current balance*/
		System.out.println("Account 2: Balance is " + ((DataStore2)ds).getBalance() );
	}
}
```

5.6.16 PromptForPin.java (Strategy Pattern)

```
/*
 Displays the action for PromptForPin
 */
package Strategy;
public class PromptForPin
{
	public void PromptForPin()
	{
		System.out.println("Please enter the PIN");
	}
}
```

5.6.17 MakeWithdraw.java (Strategy Pattern)

```
/*
 Abstract class for MakeWithdraw
 */
package Strategy;
import Data_Store.*;
public abstract class MakeWithdraw
{
	public abstract void MakeWithdraw(DataStore ds);
}
```

5.6.18 MakeWithdraw1.java (Strategy Pattern)

```
/*
 MakeWithdraw for Account 1
 */
package Strategy;
import Data_Store.*;
public class MakeWithdraw1 extends MakeWithdraw
```

```
{
        public void MakeWithdraw(DataStore ds)
        {
                /*sets the withdraw amount in the datastore for Account1, and calculates and
                returns the final balance after withdraw*/
                ((DataStore1)ds).setWithdraw();
                ((DataStore1)ds).computeBalanceAfterWithdraw();
                System.out.println("Account 1: After Withdraw, Balance is " +
                ((DataStore1)ds).getBalance() );
        }
}
```

5.6.19 MakeWithdraw2.java (Strategy Pattern)

```
/*
 MakeWithdraw for Account 2
 */
package Strategy;
import Data_Store.*;
public class MakeWithdraw2 extends MakeWithdraw
{
        public void MakeWithdraw(DataStore ds)
        {
                /*sets the withdraw amount in the datastore for Account2, and calculates and
                returns the final balance after withdraw*/
                ((DataStore2)ds).setWithdraw();
                ((DataStore2)ds).computeBalanceAfterWithdraw();
                System.out.println("Account 2: After Withdraw, Balance is " +
                ((DataStore2)ds).getBalance() );
        }
}
```

5.6.20 Penalty.java (Strategy Pattern)

```
/*
 Computes and displays the penalty for account 1
 */
package Strategy;
import Data_Store.*;
public class Penalty
{
        public void Penalty(DataStore ds)
        {
                ((DataStore1)ds).setPenalty();
                System.out.println("Account 1: Minimum required balance is $500. So Penalty is
                applied.");
                System.out.println("After a Penalty of $20, Balance is " + ((DataStore1)ds).balance );
        }
}
```

5.6.21 IncorrectLockMsg.java (Strategy Pattern)

```
/*
 Prints the IncorrectLockMsg
 */
package Strategy;
public class IncorrectLockMsg
{
	public void IncorrectLockMsg()
	{
		System.out.println("Incorrect Lock");
	}
}
```

5.6.22 IncorrectUnlockMsg.java (Strategy Pattern)

```
/*
 prints IncorrectUnlockMsg
 */
package Strategy;
public class IncorrectUnlockMsg
{
	public void IncorrectUnlockMsg()
	{
		System.out.println("Incorrect Unlock");
	}
}
```

5.6.23 NoFundsMsg.java (Strategy Pattern)

```
/*
 Displays NoFundsMsg
 */
package Strategy;
public class NoFundsMsg
{
	public void NoFundsMsg()
	{
		System.out.println("No Funds");
	}
}
```

5.7.1 AbstractFactory.java (Abstract Factory Pattern)

```
/*Abstract Class for Factory of Account1 and Account2*/
package Abstract_Factory;
import Data_Store.*;
import Strategy.*;
public interface AbstractFactory
 {
        public DataStore CreateDataStore();
        public StoreData CreateStoreData();
        public IncorrectIdMsg CreateIncorrectIdMsg();
        public IncorrectPinMsg CreateIncorrectPinMsg();
        public TooManyAttemptsMsg CreateTooManyAttemptsMsg();
        public DisplayMenu CreateDisplayMenu();
        public MakeDeposit CreateMakeDeposit();
        public DisplayBalance CreateDisplayBalance();
        public PromptForPin CreatePromptForPin();
        public MakeWithdraw CreateMakeWithdraw();
        public Penalty CreatePenalty();
        public IncorrectLockMsg CreateIncorrectLockMsg();
        public IncorrectUnlockMsg CreateIncorrectUnlockMsg();
        public NoFundsMsg CreateNoFundsMsg();
}
```

5.7.2 ConcreteFactory1.java (Abstract Factory Pattern)

```
/*
 Concrete Factory class for Account 1
 */
package Abstract_Factory;
import Data_Store.DataStore;
import Data_Store.DataStore1;
import Strategy.*;
public class ConcreteFactory1 implements AbstractFactory
{
        /*creating objects for all actions*/
        DataStore ds = new DataStore1();
        StoreData store=new StoreData1();
        DisplayBalance disp_bal = new DisplayBalance1();
        DisplayMenu disp_menu = new DisplayMenu1();
        IncorrectIdMsg id_msg=new IncorrectIdMsg();
        IncorrectPinMsg pin_msg=new IncorrectPinMsg();
        IncorrectLockMsg lock_msg=new IncorrectLockMsg();
        IncorrectUnlockMsg unlock_msg=new IncorrectUnlockMsg();
        MakeDeposit make_deposit = new MakeDeposit1();
        MakeWithdraw make_withdraw = new MakeWithdraw1();
        NoFundsMsg no_funds=new NoFundsMsg();
        Penalty penalty = new Penalty();
        PromptForPin prompt_pin = new PromptForPin();
        TooManyAttemptsMsg too_many_attempts_msg = new TooManyAttemptsMsg();
```

```
/*returning the creating object for further calls*/
public void ConcreteFactory1()
{
}
public DataStore CreateDataStore()
{
	return(this.ds);
}
public DataStore GetDataStore()
{
	return this.ds;
}
public IncorrectPinMsg CreateIncorrectPinMsg()
{
	return this.pin_msg;
}
public TooManyAttemptsMsg CreateTooManyAttemptsMsg()
{
	return this.too_many_attempts_msg;
}
public DisplayMenu CreateDisplayMenu()
{
	return this.disp_menu;
}
public PromptForPin CreatePromptForPin()
{
	return this.prompt_pin;
}
public DisplayBalance CreateDisplayBalance()
{
	return this.disp_bal;
}
public MakeDeposit CreateMakeDeposit()
{
	return this.make_deposit;
}
public MakeWithdraw CreateMakeWithdraw()
{
	return this.make_withdraw;
}
public Penalty CreatePenalty()
{
	return this.penalty;
}
public StoreData CreateStoreData()
{
	return this.store;
}
public IncorrectIdMsg CreateIncorrectIdMsg()
{
	return this.id_msg;
```

```
        }
        public IncorrectLockMsg CreateIncorrectLockMsg()
        {
                return this.lock_msg;
        }
        public IncorrectUnlockMsg CreateIncorrectUnlockMsg()
        {
                return this.unlock_msg;
        }
        public NoFundsMsg CreateNoFundsMsg()
        {
                return this.no_funds;
        }
}
```

5.7.3 ConcreteFactory2.java (Abstract Factory Pattern)

```
/*
 Concrete Factory class for Account 2
 */
package Abstract_Factory;
import Data_Store.DataStore;
import Data_Store.DataStore2;
import Strategy.*;
public class ConcreteFactory2 implements AbstractFactory
{
        /*creating objects for all actions*/
        DataStore ds = new DataStore2();
        StoreData store=new StoreData2();
        DisplayBalance disp_bal = new DisplayBalance2();
        DisplayMenu disp_menu = new DisplayMenu2();
        IncorrectIdMsg id_msg=new IncorrectIdMsg();
        IncorrectPinMsg pin_msg=new IncorrectPinMsg();
        IncorrectLockMsg lock_msg=new IncorrectLockMsg();
        IncorrectUnlockMsg unlock_msg=new IncorrectUnlockMsg();
        MakeDeposit make_deposit = new MakeDeposit2();
        MakeWithdraw make_withdraw = new MakeWithdraw2();
        NoFundsMsg no_funds=new NoFundsMsg();
        Penalty penalty = new Penalty();
        PromptForPin prompt_pin = new PromptForPin();
        TooManyAttemptsMsg too_many_attempts_msg = new TooManyAttemptsMsg();

        /*returning the creating object for further calls*/
        public void ConcreteFactory2()
        {
        }
        public DataStore CreateDataStore()
        {
                return this.ds;
        }
        public DataStore GetDataStore()
```

```
{
		return this.ds;
}
public IncorrectPinMsg CreateIncorrectPinMsg()
{
		return this.pin_msg;
}
public TooManyAttemptsMsg CreateTooManyAttemptsMsg()
{
		return this.too_many_attempts_msg;
}
public DisplayMenu CreateDisplayMenu()
{
		return this.disp_menu;
}
public PromptForPin CreatePromptForPin()
{
		return this.prompt_pin;
}
public DisplayBalance CreateDisplayBalance()
{
		return this.disp_bal;
}
public MakeDeposit CreateMakeDeposit()
{
		return this.make_deposit;
}
public MakeWithdraw CreateMakeWithdraw()
{
		return this.make_withdraw;
}
public Penalty CreatePenalty()
{
		return this.penalty;
}
public StoreData CreateStoreData()
{
		return this.store;
}
public IncorrectIdMsg CreateIncorrectIdMsg()
{
		return this.id_msg;
}
public IncorrectLockMsg CreateIncorrectLockMsg()
{
		return this.lock_msg;
}
public IncorrectUnlockMsg CreateIncorrectUnlockMsg()
{
		return this.unlock_msg;
}
```

```
		public NoFundsMsg CreateNoFundsMsg()
		{
			return this.no_funds;
		}
}
```

CONCLUSION

The two different Account components have been designed and implements using the State Pattern, Strategy Pattern and Abstract Factory Pattern.
Two Different scenarios, one of Account 1 and the other for Account 2 are mentioned and a sequence diagram clearly shows the operation.
The source code is also documented in section 5, differentiating them according to the pattern they belong to.
A jar file to run the Driver is also submitted.