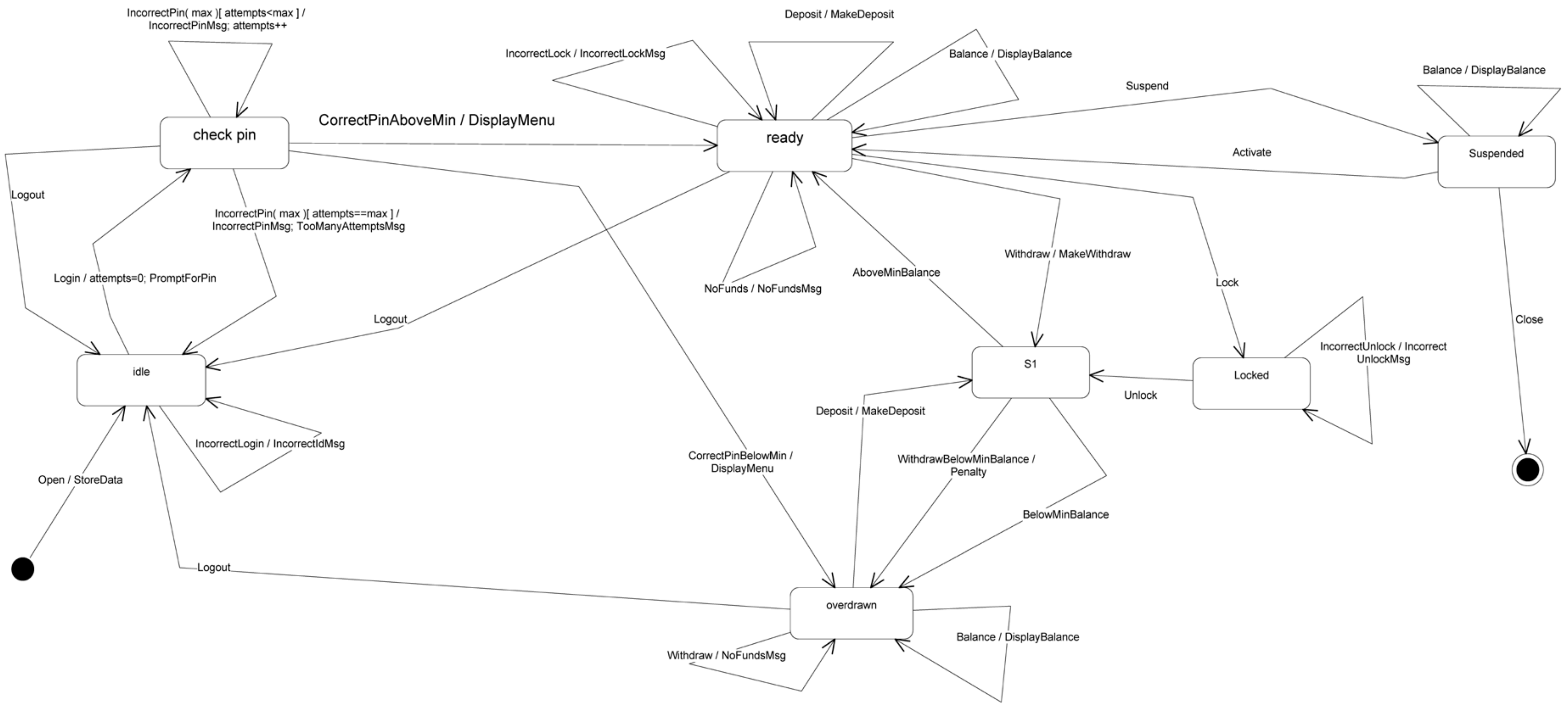
Jiaqi Yan
jyan31@hawk.iit.edu
A20321362
Spring 2016

CS 586 MDABankAccount Project Report

"A BANK IS A PLACE WHERE THEY LEND YOU AN UMBRELLA IN FAIR WEATHER AND ASK FOR IT BACK WHEN IT BEGINS TO RAIN."
--- ROBERT FROST

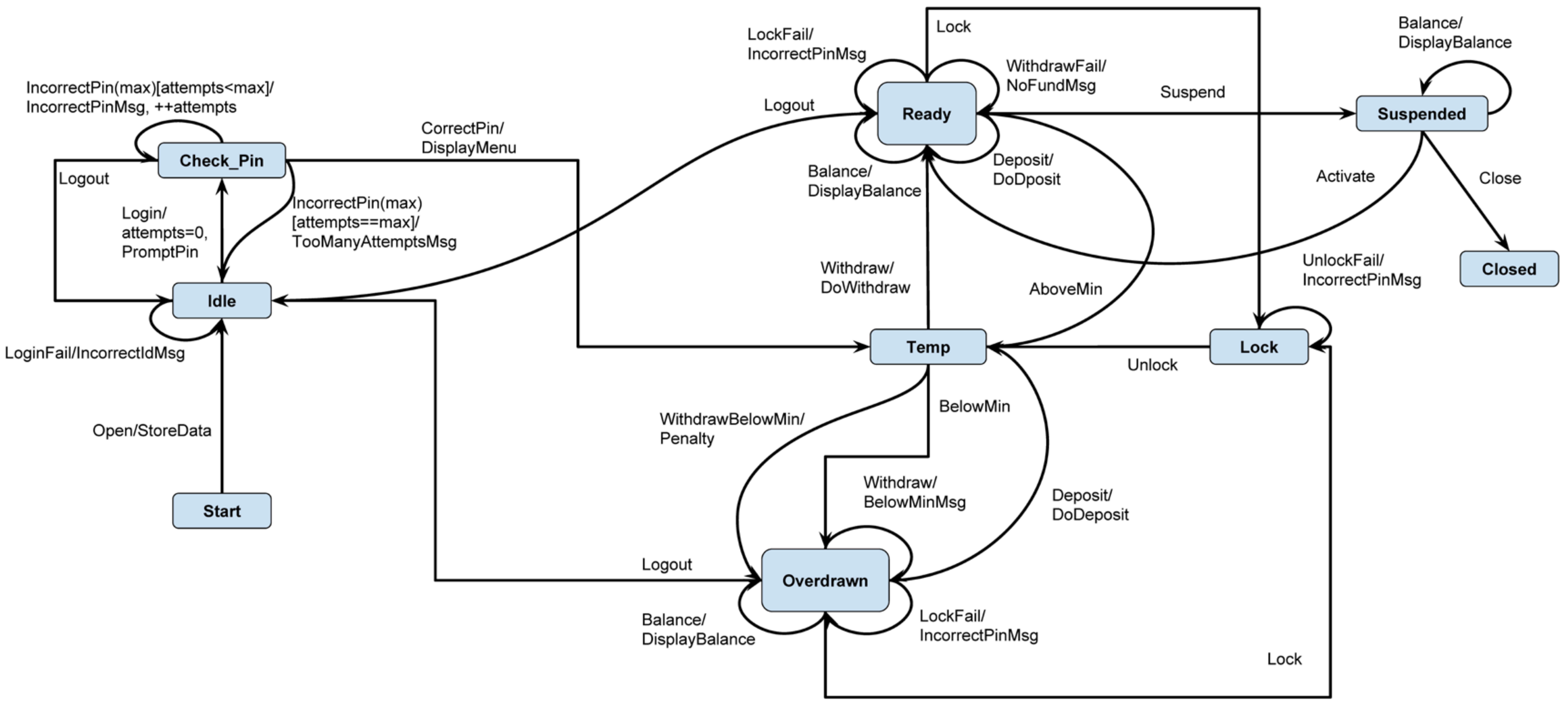
Table of Contents

1. MDA-EFSM Model for the Account Components



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

1. MDA-EFSM Model for the Account Components



EFSM-MDA State Diagram

1. MDA-EFSM Model for the Account Components

As shown in the previous EFSM figure for model driven architecture, I choose to use the sample solution provided at Blackboard.

The list of meta events in the MDA-EFSM:
Open()
Login()
LoginFail()
Login()
IncorrectPin(int max)
CorrectPin()
AboveMin()
BelowMin()
Deposit()
Balance()
WithdrawFail()
Withdraw()
WithdrawBelowMin()
LockFail()
Lock()
UnlockFail()
Unlock()
Suspend()
Activate()
Close()

The list of meta actions in the MDA-EFSM:
StoreData(): store PIN, ID and balance data into DataStore
IncorrectIdMsg(): show “incorrect ID number” msg to user
IncorrectPinMsg(): show “incorrect PIN number” msg to user
TooManyAttemptMsg(): show “too many attempts” msg to user
DisplayMenu(): display menu on the console
DoDeposit(): deposit money into user’s account
DoWithdrawn(): withdraw money from user’s account
DisplayBalance(): display current account’s balance
PromptPin(): prompt for PIN number
NoFundMsg(): show “no sufficient money” msg to user
Penalty(): deduct penalty from user balance

1. MDA-EFSM Model for the Account Components

Pseudo-code of InputProcessors for Account1

```
void open(string p, string y, float a) {
    data->temp_pin = p;
    data->temp_id = y;
    data->temp_b = a;
    mda->Open()
}
void pin(string x) {
    if (x == data->pin) {
        mda->CorrectPin();
        if (data->b > min_balance) {
            mda->AboveMin();
        } else {
            mda->BelowMin();
        }
    } else {
        mda->IncorrectPin(max_attempts);
    }
}
void deposit(float d) {
    data->temp_d = d;
    mda->Deposit();
    if (data->b > min_balance) {
        mda->AboveMin();
    } else {
        mda->BelowMin();
    }
}
```

```
void withdraw(float w) {
    data->temp_w = w;
    if (data->b <= min_balance) {
        mda->WithdrawFail()
    } else {
        mda->Withdraw();
    }
    if (data->b > min_balance) {
        mda->AboveMin();
    } else {
        mda->BelowMin();
    }
}
void balance() {
    mda->Balance();
}
void logout() {
    mda->Logout();
}
void lock(string x) {
    if (x == data->pin) {
        mda->Lock();
    } else {
        mda->LockFail();
    }
}
```

```
void login(string y) {
    if (y == data->id) {
        mda->Login();
    } else {
        mda->LoginFail();
    }
}
void unlock(string x) {
    if (x == data->pin) {
        mda->Unlock();
        if (data->b > min_balance) {
            mda->AboveMin();
        } else {
            mda->BelowMin();
        }
    } else {
        mda->UnlockFail();
    }
}

max_attempts = 3;
min_balance = 500;
```

1. MDA-EFSM Model for the Account Components

Pseudo-code of InputProcessors for Account2

```
void OPEN(int p, int y, int a) {
    data->temp_pin = p;
    data->temp_id = y;
    data->temp_b = a;
    mda->Open();
}
void Account2::PIN(int x) {
    if (x == data->pin) {
        mda->CorrectPin();
        mda->AboveMin();
    } else {
        mda-
    >IncorrectPin(max_attempts);
    }
}
```

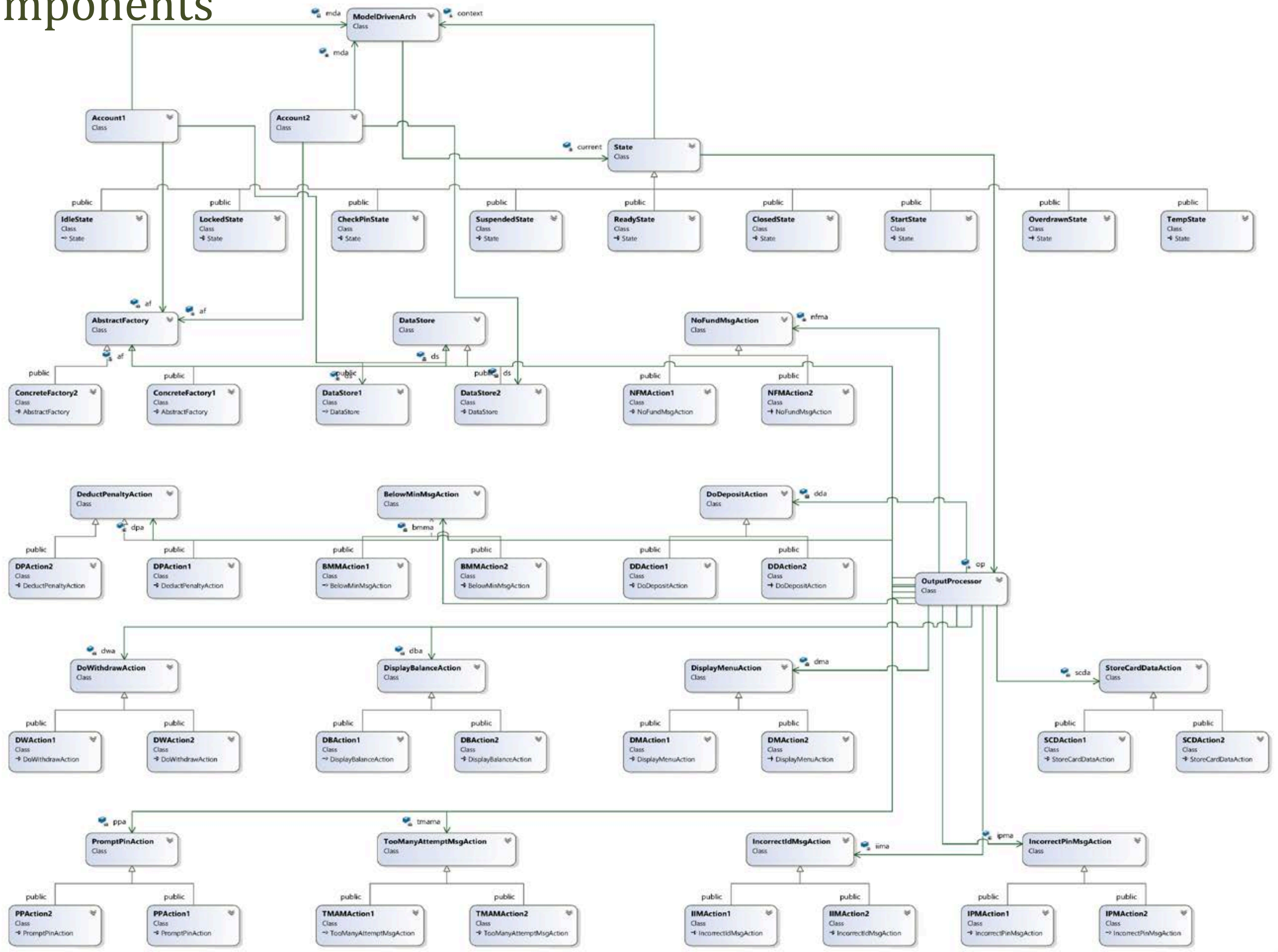
```
void Account2::BALANCE() {
    mda->Balance();
}
void Account2::LOGIN(int y) {
    if (y == data->id) {
        mda->Login();
    } else {
        mda->LoginFail();
    }
}
void Account2::LOGOUT() {
    mda->Logout();
}
void Account2::WITHDRAW(int w) {
    data->temp_w = w;
    if (data->b > min_balance) {
        mda->Withdraw();
        mda->AboveMin();
    } else {
        mda->WithdrawFail();
    }
}
```

```
void Account2::suspend() {
    mda->Suspend();
}
void Account2::activate() {
    mda->Activate();
}
void Account2::close() {
    mda->Close();
}
void Account2::DEPOSIT(int d) {
    data->temp_d = d;
    mda->Deposit();
    mda->AboveMin();
}
```

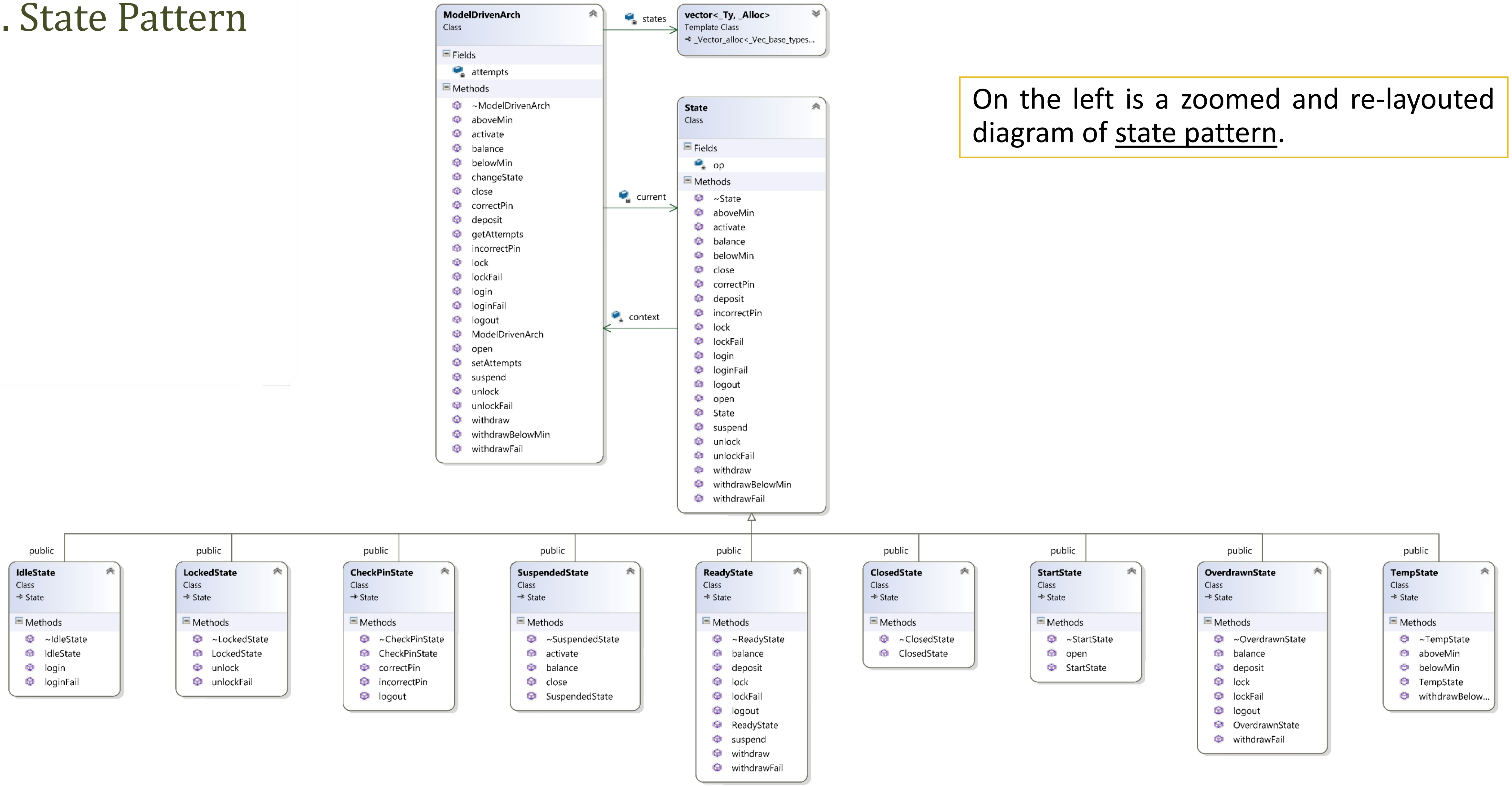
2. Class Diagrams of All Components

The overview architecture is drawn on the right diagram. It just shows the general relationships between various of classes. What missing here is the relationships between concrete factories and their corresponding products, e.g. concrete strategies and concrete DataStore objects, which is presented at Section 5. Abstract Factory Pattern II – Concrete Factories

3 class design patterns group classes by their responsibilities. In the following sections, I will introduce them by group. Classes in each design are described in detail.



3. State Pattern



On the left is a zoomed and re-layouted diagram of state pattern.

3. State Pattern

Purpose, Attributes and Operations for State Pattern involved classes

Class ModelDrivenArch:			
Purpose	Play the role of context class; keep tracking the current state in the EFSM; forward meta events to concrete states		
Member variables	*current*: pointer to the object of current state *attempts*: number of incorrect PIN attempts *states*: a vector storing all state objects that appear in the state machine		
Constructor	Create objects for each state in the state machine; store them in *states*; set *current* to *StartState* object		
Destructor	Reclaim the objects in *states*		
changeState(StateEnum id)	Used by State objects to change the *current* state of context		
set/getAttempts()	Used by State objects to change/get the value of *attempts*		
open()	Handle open event by calling current->open()	balance()	Handle balance event by calling current->balance()
login()	Handle login event by calling current->login()	withdraw()	Handle withdraw event by calling current->withdraw()
loginFail()	Handle loginFail event by calling current->loginFail()	withdrawFail()	Handle withdrawFail event by calling current->withdrawFail()
logout(int max)	Handle logout event by calling current->logout)	deposit()	Handle deposit event by calling current->deposit()
correctPin()	Handle correctPin event by calling current->correctPin()	lock()	Handle lock event by calling current->lock()
aboveMin()	Handle aboveMin event by calling current->aboveMin()	lockFail()	Handle lockFail event by calling current->lockFail()
belowMin()	Handle belowMin event by calling current->belowMin()	unlock()	Handle unlock event by calling current->unlock()
suspend()	Handle suspend event by calling current->suspend()	unlockFail()	Handle unlockFail event by calling current->unlockFail()
activate()	Handle activate event by calling current->activate()	close()	Handle close event by calling current->close()
incorrectPin(int max)	Handle incorrectPin event by calling current->incorrectPin(max)	withdrawBelowMin()	Handle withdrawBelowMin event by calling current->withdrawBelowMin()

3. State Pattern

Purpose, Attributes and Operations for State Pattern involved classes

Class: State			
Purpose	Group all states appeared in the EFSM; define and provide default implementation of all event handler, e.g. do **NOTHING**		
Member variables	*context*: pointer to the EFSM's context, e.g. a ModelDrivenArch instance *op*: pointer to OutputProcessor instance for issuing action		
Constructor	Initilize *context* and *op* with provided parameters		
Destructor	No need to do anything		
open()	Virtual function overridden by subclasses, empty operation	balance()	Virtual function overridden by subclasses, empty operation
login()	Virtual function overridden by subclasses, empty operation	withdraw()	Virtual function overridden by subclasses, empty operation
loginFail()	Virtual function overridden by subclasses, empty operation	withdrawFail()	Virtual function overridden by subclasses, empty operation
logout(int max)	Virtual function overridden by subclasses, empty operation	deposit()	Virtual function overridden by subclasses, empty operation
correctPin()	Virtual function overridden by subclasses, empty operation	lock()	Virtual function overridden by subclasses, empty operation
aboveMin()	Virtual function overridden by subclasses, empty operation	lockFail()	Virtual function overridden by subclasses, empty operation
belowMin()	Virtual function overridden by subclasses, empty operation	unlock()	Virtual function overridden by subclasses, empty operation
suspend()	Virtual function overridden by subclasses, empty operation	unlockFail()	Virtual function overridden by subclasses, empty operation
activate()	Virtual function overridden by subclasses, empty operation	close()	Virtual function overridden by subclasses, empty operation
incorrectPin(int max)	Virtual function overridden by subclasses, empty operation	withdrawBelowMin()	Virtual function overridden by subclasses, empty operation

3. State Pattern

Purpose, Attributes and Operations for State Pattern involved classes

Class: StartState	
Purpose	Represent the Start state in EFSM; handle open event when current state is Start
Member variables	Inherit *context* and *op* from base class State
open()	Change *current* to Idle state, issue StoreData action to OutputProcessor *op*

Class: IdelState	
Purpose	Represent the Idle state in EFSM; handle login and loginFail events when current state is Idle
Member variables	Inherit *context* and *op* from base class State
login()	Change *current* to CheckPin state, set context's *attempts* to 0, issue PromptPin action to OutputProcessor *op*
loginFail()	Issue IncorrectIdMsg to OutputProcessor *op*

Class: CheckPinState	
Purpose	Represent the CheckPin state in EFSM; handle correctPin, incorrectPin and logout events when current state is CheckPin
Member variables	Inherit *context* and *op* from base class State
correctPin()	Change *current* to Temp state, issue DisplayMenu action to OutputProcessor *op*
incorrectPin(int max)	If *attempts* >= *max*, reduce to Idle state and issue TooManyAttemptMsg action to OutputProcessor *op*; otherwise, increase *context*'s *attempts* by one and issue IncorrectPinMsg action to OutputProcessor *op*
logout()	Change current to Idle state

3. State Pattern

Purpose, Attributes and Operations for State Pattern involved classes

Class: ReadyState	
Purpose	Represent the Ready state in EFSM; handle events illustrated in the EFSM figure
Member variables	Inherit *context* and *op* from base class State
balance()	Issue DisplayBalance action to OutputProcessor *op*
lockFail()	Issue IncorrectPinMsg action to OutputProcessor *op*
lock()	Change *current* to Locked state
suspend()	Change *current* to Suspended state
withdrawFail()	Issue NoFundMsg action to OutputProcessor *op*
withdraw()	Change *current* to Temp state, issue DoWithdraw action to OutputProcessor *op*
deposit()	Issue DoDeposit action to OutputProcessor *op*
logout()	Change *current* to Idle state

3. State Pattern

Purpose, Attributes and Operations for State Pattern involved classes

Class: OverdrawnState	
Purpose	Represent the Overdrawn state in EFSM; handle events illustrated in the EFSM figure
Member variables	Inherit *context* and *op* from base class State
balance()	Issue DisplayBalance action to OutputProcessor *op*
lockFail()	Issue IncorrectPinMsg action to OutputProcessor *op*
lock()	Change *current* to Locked state
withdrawFail()	Issue BelowMinMsg action to OutputProcessor *op*
deposit()	Change *current* to Temp state, issue DoDeposit action to OutputProcessor *op*
logout()	Change *current* to Idle state

Class: LockedState	
Purpose	Represent the Locked state in EFSM; handle unlock and unlockFail events when current state is Idle
Member variables	Inherit *context* and *op* from base class State
unlock()	Change *current* to Temp state
unlockFail()	Issue IncorrectPinMsg to OutputProcessor *op*

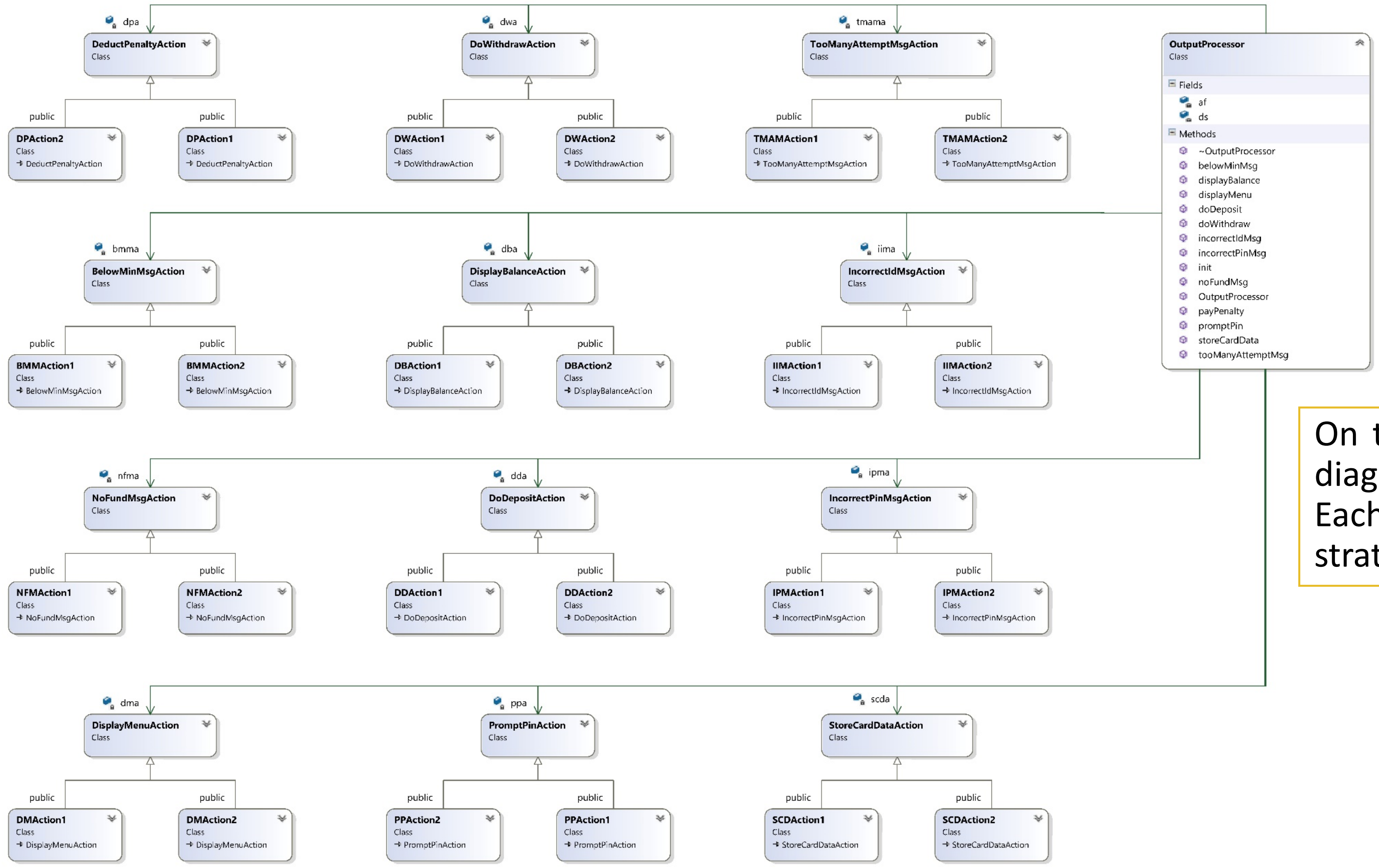
3. State Pattern

Purpose, Attributes and Operations for State Pattern involved classes

Class: SuspendedState	
Purpose	Represent the Suspended state in EFSM; handle activate, balance and close events illustrated in the EFSM figure
Member variables	Inherit *context* and *op* from base class State
balance()	Issue DisplayBalance action to OutputProcessor *op*
activate()	Change *current* to Ready state
close()	Change *current* to Closed state

Class: TempState	
Purpose	Represent the Temp state in EFSM; handle aboveMin, belowMin and withdrawBelowMin events when current state is Idle
Member variables	Inherit *context* and *op* from base class State
aboveMin()	Change *current* to Ready state
belowMin()	Change *current* to Overdrawn state
withdrawBelowMin()	Change *current* to Overdrawn state, issue Penalty action to OutputProcessor *op*

4. Strategy Pattern



On the left is a zoomed out diagram of strategy pattern. Each action corresponds to a strategy.

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: OutputProcessor			
Purpose	Play the role of output processor in MDA architecture; as client of various action strategies, issue actions on each strategy's base class		
Member variables	*ds*: point to a DataStore object; some actions need a DataStore object to access data	*dpa*: pinter to DeductPenaltyAction strategy	
	sda: pointer to StoreDataAction strategy	*ipma*: pointer to IncorrectPinMsgAction strategy	
	iimp: pointer to IncorrectIdMsgAction strategy	*tmama*: pointer to TooManyAttmeptAction strategy	
	ppa: pointer to PromptPinAction strategy	*dma*: pointer to DisplayMenuAction strategy	
	dda: pointer to DoDepositAction strategy	*nfma*: pointer to NoFundMsaAction strategy	
	dba: pointer to DoWithdrawAction strategy	*bmma*: pointer to BelowMinMsgAction strategy	
	af: Abstract factory instance used to create/configure all the previous pointers		
Constructor(AbstractFacotry *af)	Initialize *af* with a concrete factory	Destructor()	Do nothing
init()	Create/configure DataStore object and action strategies using *af*, see Abstract Factory Pattern for details		
storeData()	Forward action by invoking sda->storeData()	doDeposit()	Forward action by invoking dda->doDeposit()
incorrecePinMsg()	Forward action by invoking ipma>incorrecePinMsg()	noFundMsg()	Forward action by invoking nfma->noFundMsg()
incorrectIdMsg()	Forward action by invoking iima->incorrecePinMsg()	displayBalance()	Forward action by invoking dba->displayBalance()
tooManyAttemptMsg()	Forward action by invoking tmama->tooManyAttemptMsg()	doWithdraw()	Forward action by invoking dwm->doWithdraw()
promptPin()	Forward action by invoking ppa->promptPin()	belowMinMsg()	Forward action by invoking bmma->belowMinMsg()
displayMenu()	Forward action by invoking dma->displayMenu()	payPenalty()	Forward action by invoking dpa->payPenalty()

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: StoreDataAction	
Purpose	Abstract class that groups strategies of StoreData action
Member variables	None
Constructor/Destructor()	Do nothing
storeData(DataStore *ds)	Abstract method; subclass should override it to store temp_pin, temp_id and temp_balance to pin, id and balance respectively; different strategies differs in the type of DataStore

Class: SDAction1	
Purpose	Concrete strategy of StoreData action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
storeData(DataStore *ds)	Cast *ds* to DataStore1 object and execute the store action

Class: SDAction2	
Purpose	Concrete strategy of StoreData action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
storeData(DataStore *ds)	Cast *ds* to DataStore2 object and execute the store action

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: IncorrectPinMsgAction	
Purpose	Abstract class that groups strategies of IncorrectPinMsg action
Member variables	None
Constructor/Destructor()	Do nothing
incorrectPinMsg()	Abstract method; override to print out msg about incorrect PIN number for different accounts

Class: IPMAction1	
Purpose	Concrete strategy of IncorrectPinMsg action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
incorrectPinMsg()	Print out incorrect PIN msg for account1

Class: IPMAction2	
Purpose	Concrete strategy of IncorrectPinMsg action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
incorrectPinMsg()	Print out incorrect PIN msg for account2

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: IncorrectIdMsgAction	
Purpose	Abstract class that groups strategies of IncorrectIdMsg action
Member variables	None
Constructor/Destructor()	Do nothing
incorrectIdMsg()	Abstract method; override to print out msg about incorrect ID number for different accounts

Class: IIMAction1	
Purpose	Concrete strategy of IncorrectIdMsg action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
incorrectIdMsg()	Print out incorrect ID msg for account1

Class: IIMAction2	
Purpose	Concrete strategy of IncorrectIdMsg action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
incorrectIdMsg()	Print out incorrect ID msg for account2

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: TooManyAttemptMsgAction	
Purpose	Abstract class that groups strategies of TooManyAttemptMsg action
Member variables	None
Constructor/Destructor()	Do nothing
tooManyAttemptMsg()	Abstract method; override to print out msg about “too many incorrect PIN number” for different accounts

Class: TMAMAction1	
Purpose	Concrete strategy of TooManyAttemptMsg action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
tooManyAttemptMsg()	Print out too many incorrect PIN number msg for account1

Class: TMAMAction2	
Purpose	Concrete strategy of TooManyAttemptMsg action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
tooManyAttemptMsg()	Print out too many incorrect PIN number msg for account2

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: PromptPinAction	
Purpose	Abstract class that groups strategies of PromptPin action
Member variables	None
Constructor/Destructor()	Do nothing
promptPin()	Abstract method; override to print out msg about “please input PIN number” for different accounts

Class: PPAction1	
Purpose	Concrete strategy of PromptPin action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
promptPin()	Print out “please input PIN number” msg for account1

Class: PPAction2	
Purpose	Concrete strategy of PromptPin action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
promptPin()	Print out “please input PIN number” msg for account2

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: DisplayMenuAction	
Purpose	Abstract class that groups strategies of DisplayMenu action
Member variables	None
Constructor/Destructor()	Do nothing
displayMenu()	Abstract method; override to print menu for different accounts

Class: DMAction1	
Purpose	Concrete strategy of DisplayMenu action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
displayMenu()	Print out menu for account1

Class: DMAction2	
Purpose	Concrete strategy of DisplayMenu action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
displayMenu()	Print out menu for account2

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: DoDepositAction	
Purpose	Abstract class that groups strategies of DoDeposit action
Member variables	None
Constructor/Destructor()	Do nothing
doDeposit(DataStore *ds)	Abstract method; override to make deposit with *temp_d* and *balance* in DataStore

Class: DDAction1	
Purpose	Concrete strategy of DoDeposit action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
doDeposit(DataStore *ds)	Make deposit by adding *temp_d* and *balance* in DataStore

Class: DDAction2	
Purpose	Concrete strategy of DoDeposit action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
doDeposit(DataStore *ds)	Make deposit by adding *temp_d* and *balance* in DataStore

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: NoFundMsgAction	
Purpose	Abstract class that groups strategies of NoFundMsg action
Member variables	None
Constructor/Destructor()	Do nothing
noFundMsg()	Abstract method; override to prompt msg about insufficient fund

Class: NFMAction1	
Purpose	Concrete strategy of NoFundMsg action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
doDeposit()	Prompt msg about insufficient fund

Class: NFMAction2	
Purpose	Concrete strategy of NoFundMsg action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
doDeposit()	Prompt msg about insufficient fund

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: DisplayBalanceAction	
Purpose	Abstract class that groups strategies of DisplayBalance action
Member variables	None
Constructor/Destructor()	Do nothing
displayBalance(DataStore *ds)	Abstract method; override to display current balance for different accounts

Class: DBAction1	
Purpose	Concrete strategy of DisplayBalance action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
displayBalance(DataStore *ds)	display current balance for account 1

Class: DBAction2	
Purpose	Concrete strategy of DisplayBalance action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
displayBalance(DataStore *ds)	display current balance for account 2

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: DoWithdrawAction	
Purpose	Abstract class that groups strategies of DoWithdraw action
Member variables	None
Constructor/Destructor()	Do nothing
doWithdraw(DataStore *ds)	Abstract method; override to make withdraw from *temp_w* from *balance*

Class: DWAction1	
Purpose	Concrete strategy of DoWithdraw action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
doWithdraw(DataStore *ds)	Make withdraw from *temp_w* from *balance*

Class: DWAction2	
Purpose	Concrete strategy of DoWithdraw action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
doWithdraw(DataStore *ds)	Make withdraw from *temp_w* from *balance*

4. Strategy Pattern

Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: BelowMinMsgAction	
Purpose	Abstract class that groups strategies of BelowMinMsg action
Member variables	None
Constructor/Destructor()	Do nothing
belowMinMsg()	Abstract method; override to print out msg about below minimum balance

Class: DWAction1	
Purpose	Concrete strategy of BelowMinMsg action for Account1
Member variables	None
Constructor/Destructor()	Do nothing
belowMinMsg()	Print out msg about below minimum balance

Class: DWAction2	
Purpose	Concrete strategy of BelowMinMsg action for Account2
Member variables	None
Constructor/Destructor()	Do nothing
belowMinMsg()	Print out msg about below minimum balance

4. Strategy Pattern

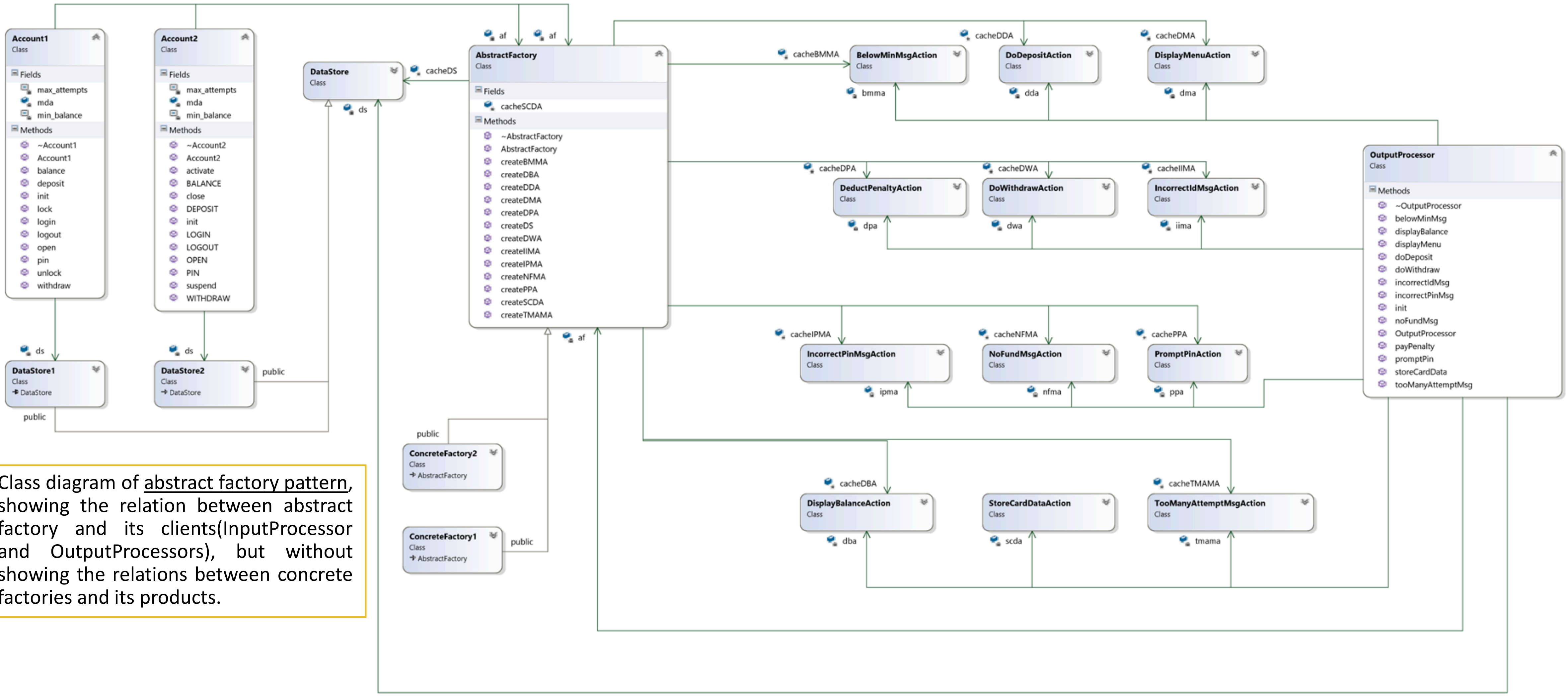
Purpose, Attributes and Operations for Strategy Pattern involved classes

Class: DeductPenaltyAction	
Purpose	Abstract class that groups strategies of DeductPenalty action
Member variables	*penalty*: amount of penalty to pay for different accounts
Constructor/Destructor()	Do nothing
payPenalty(DataStore *ds)	Abstract method; override to deduct penalty from *balance*

Class: DPAction1	
Purpose	Concrete strategy of BelowMinMsg action for Account1
Member variables	Inherit *penalty* from base class
Constructor/Destructor()	Do nothing
payPenalty(DataStore *ds)	Deduct penalty from *balance,* penalty for Account1 is 20

Class: DPAction2	
Purpose	Concrete strategy of BelowMinMsg action for Account2
Member variables	Inherit *penalty* from base class
Constructor/Destructor()	Do nothing
payPenalty(DataStore *ds)	Deduct penalty from *balance*, penalty for Account2 is zero

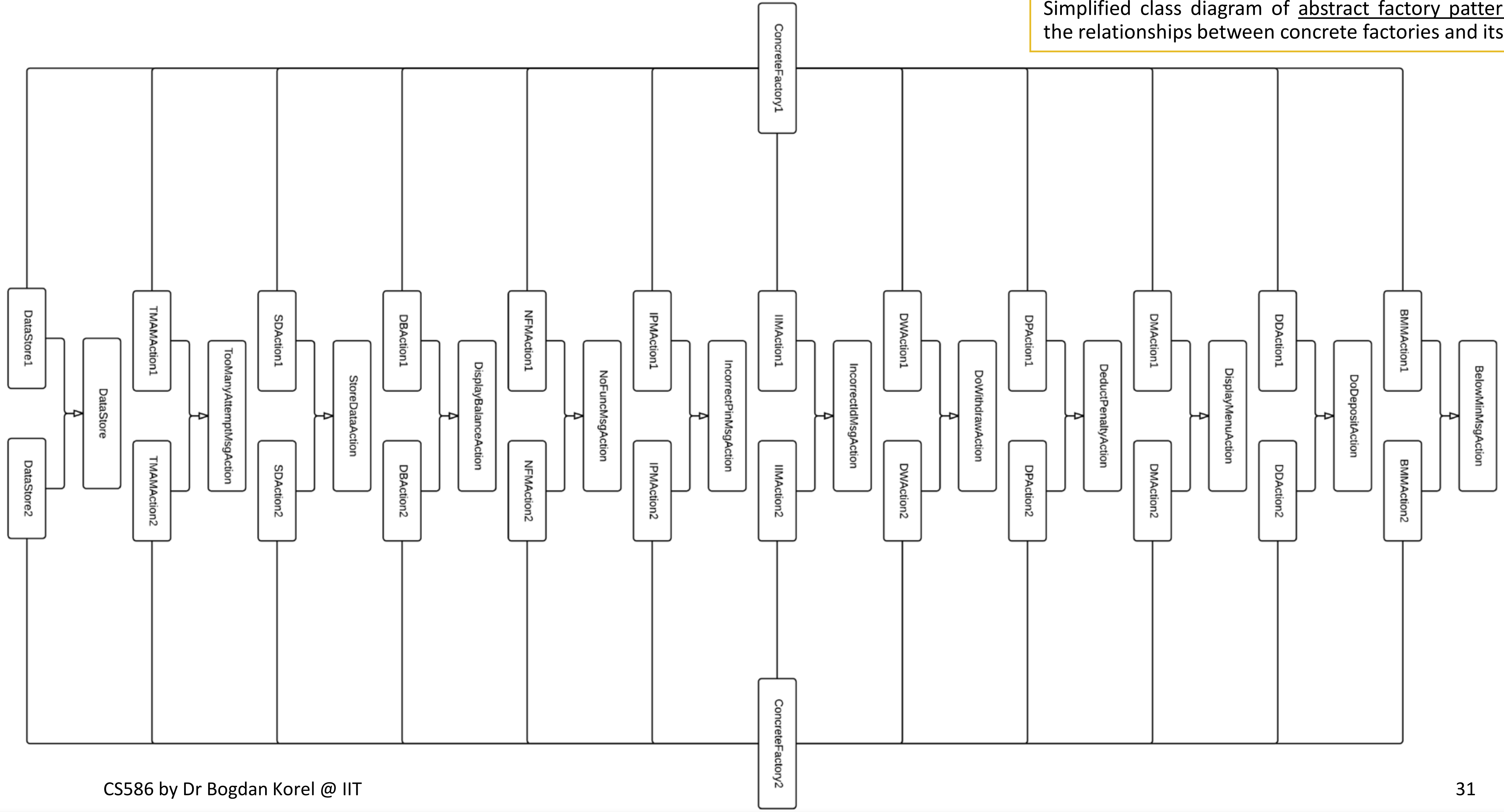
5. Abstract Factory Pattern I – AbstractFactory



Class diagram of abstract factory pattern, showing the relation between abstract factory and its clients(InputProcessor and OutputProcessors), but without showing the relations between concrete factories and its products.

5. Abstract Factory Pattern II – Concrete Factories

Simplified class diagram of abstract factory pattern showing the relationships between concrete factories and its products.



5. Abstract Factory Pattern

Purpose, Attributes and Operations for Abstract Factory Pattern involved classes

Class: AbstractFactory			
Purpose	Groups various of concrete factories for different clients (InputProcessors and OutpurProcessor)		
Member variables	*cacheDS*: pointer to a DataStore object;	*cacheDPA*: pinter to DeductPenaltyAction strategy	
Notice: since InputProcessor and OutputProcessor should operate on the same DataStore instance, AbstractFactory should only create 1 such object and cache it for later usage.	*cacheSDA*: pointer to StoreDataAction strategy	*cacheIPMA*: pointer to IncorrectPinMsgAction strategy	
	cacheIIMP: pointer to IncorrectIdMsgAction strategy	*cacheTMAMA*: pointer to TooManyAttmeptAction strategy	
	cachePPA: pointer to PromptPinAction strategy	*cacheDMA*: pointer to DisplayMenuAction strategy	
	cacheDDA: pointer to DoDepositAction strategy	*cacheNFMA*: pointer to NoFundMsaAction strategy	
	cacheDWA: pointer to DoWithdrawAction strategy	*cacheBMMA*: pointer to BelowMinMsgAction strategy	
Constructor()	Initialize *all pointers above* to *NULL*	Destructor()	Do nothing
createDS()	Abstract method returning DataStore instance to client		
createSDA()	Abstract method returning StoreCardAction strategy	createDDA()	Abstract method returning DoDepositAction strategy
createIPMA()	Abstract method returning IncorrectPinMsgAction strategy	createNFMA()	Abstract method returning NoFundMsgAction strategy
createIIMA()	Abstract method returning IncorrectIdMsgAction strategy	createDBA()	Abstract method returning DisplayBalanceAciton strategy
createTMAMA()	Abstract method returning IncorrectIdMsgAction strategy	createDWA()	Abstract method returning DoWithdrawAction strategy
createPPA()	Abstract method returning PromptPinAciton strategy	createBMMA()	Abstract method returning BelowMinMsgAction strategy
createDMA()	Abstract method returning DisplayMenuAction strategy	createDPA()	Abstract method returning DeductPenaltyAction strategy

5. Abstract Factory Pattern

Purpose, Attributes and Operations for Abstract Factory Pattern involved classes

Class: ConcreteFactory1			
Purpose	Concrete factory who create strategies and DataStore1 instance for Account1 and its paired OutputProcessor		
Member variables	*Cache* pointers inherited from AbstractFactory base class		
Constructor()	Do nothing (use base class's constructor)	Destructor()	Do nothing
createDS()	Create DataStore1 instance, cache it before return it to client; thereafter return the cached instance		
createSDA()	Create SDAction1 instance, cache it before return it to client; thereafter return the cached instance		
createIPMA()	Create IPMAction1 instance, cache it before return it to client; thereafter return the cached instance		
createIIMA()	Create IIMAction1 instance, cache it before return it to client; thereafter return the cached instance		
createTMAMA()	Create TMAMAction1 instance, cache it before return it to client; thereafter return the cached instance		
createPPA()	Create PPAction1 instance, cache it before return it to client; thereafter return the cached instance		
createDMA()	Create DMAction1 instance, cache it before return it to client; thereafter return the cached instance		
createDDA()	Create DDAction1 instance, cache it before return it to client; thereafter return the cached instance		
createNFMA()	Create NFMAction1 instance, cache it before return it to client; thereafter return the cached instance		
createDBA()	Create DBAction1 instance, cache it before return it to client; thereafter return the cached instance		
createDWA()	Create DWAction1 instance, cache it before return it to client; thereafter return the cached instance		
createBMMA()	Create BMMAction1 instance, cache it before return it to client; thereafter return the cached instance		
createDPA()	Create DPAction1 instance, cache it before return it to client; thereafter return the cached instance		

5. Abstract Factory Pattern

Purpose, Attributes and Operations for Abstract Factory Pattern involved classes

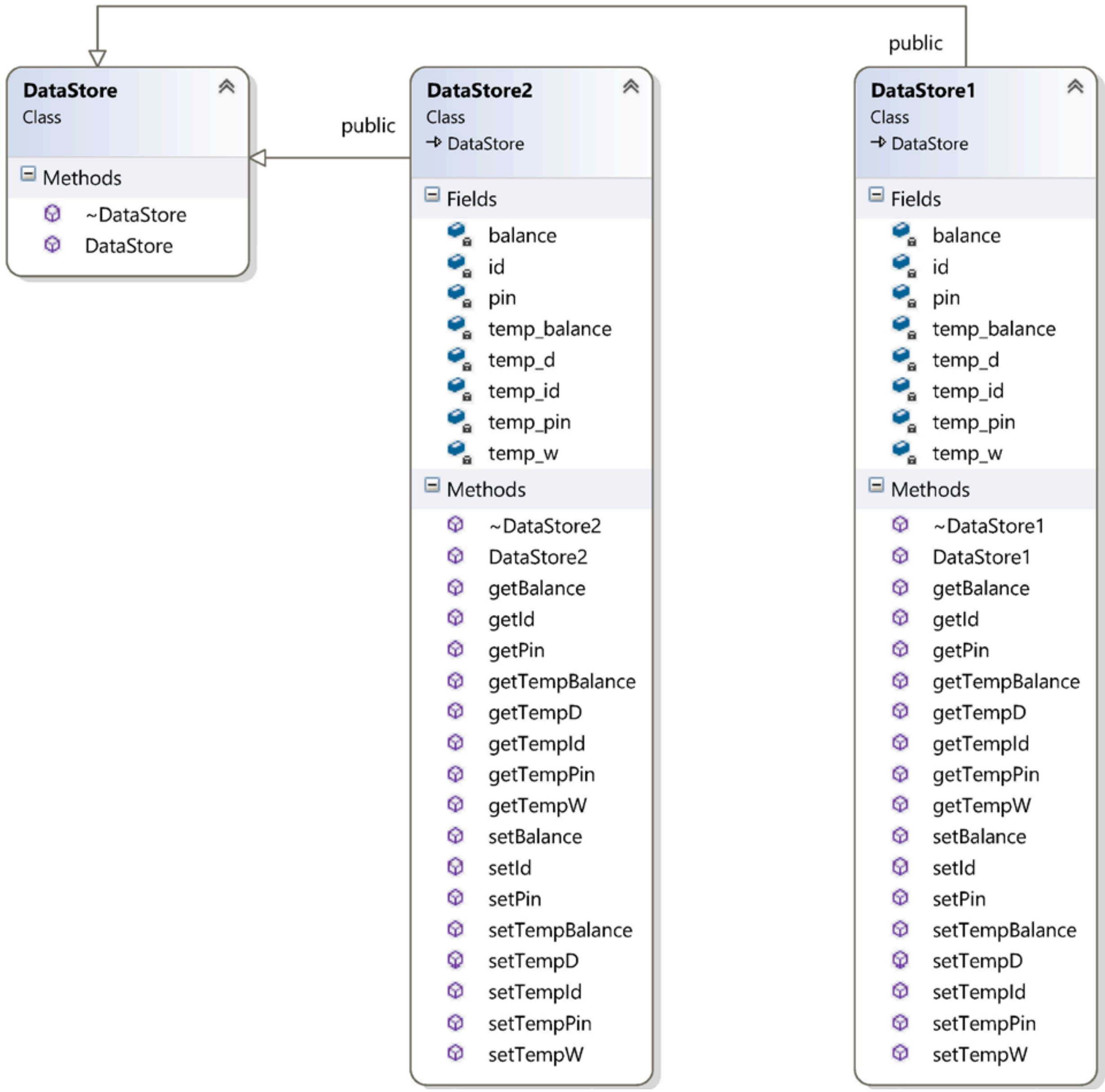
Class: ConcreteFactory1			
Purpose	Concrete factory who create strategies and DataStore2 instance for Account2 and its paired OutputProcessor		
Member variables	*Cache* pointers inherited from AbstractFactory base class		
Constructor()	Do nothing (use base class's constructor)	Destructor()	Do nothing
createDS()	Create DataStore2 instance, cache it before return it to client; thereafter return the cached instance		
createSDA()	Create SDAction2 instance, cache it before return it to client; thereafter return the cached instance		
createIPMA()	Create IPMAction2 instance, cache it before return it to client; thereafter return the cached instance		
createIIMA()	Create IIMAction2 instance, cache it before return it to client; thereafter return the cached instance		
createTMAMA()	Create TMAMAction2 instance, cache it before return it to client; thereafter return the cached instance		
createPPA()	Create PPAction2 instance, cache it before return it to client; thereafter return the cached instance		
createDMA()	Create DMAction2 instance, cache it before return it to client; thereafter return the cached instance		
createDDA()	Create DDAction2 instance, cache it before return it to client; thereafter return the cached instance		
createNFMA()	Create NFMAction2 instance, cache it before return it to client; thereafter return the cached instance		
createDBA()	Create DBAction2 instance, cache it before return it to client; thereafter return the cached instance		
createDWA()	Create DWAction2 instance, cache it before return it to client; thereafter return the cached instance		
createBMMA()	Create BMMAction2 instance, cache it before return it to client; thereafter return the cached instance		
createDPA()	Create DPAction2 instance, cache it before return it to client; thereafter return the cached instance		

6. Details for Other Classes: DataStore

Class: DataStore	
Purpose	Groups different accounts' data

Class: DataStore1	
Purpose	Data store used in implementing Account1's logic
Member variables	balance: float type, represent account balance id: string type, represent account id pin: string type, represent PIN number temp_balance: float type, temporal data used for open() temp_pin: string type, temporal data used for open() temp_id: string type, temporal data used for open() temp_d: float type, temporal data used for deposit() temp_w: float type, temporal data used for withdraw()
Operations	Various getter and setter for member variables

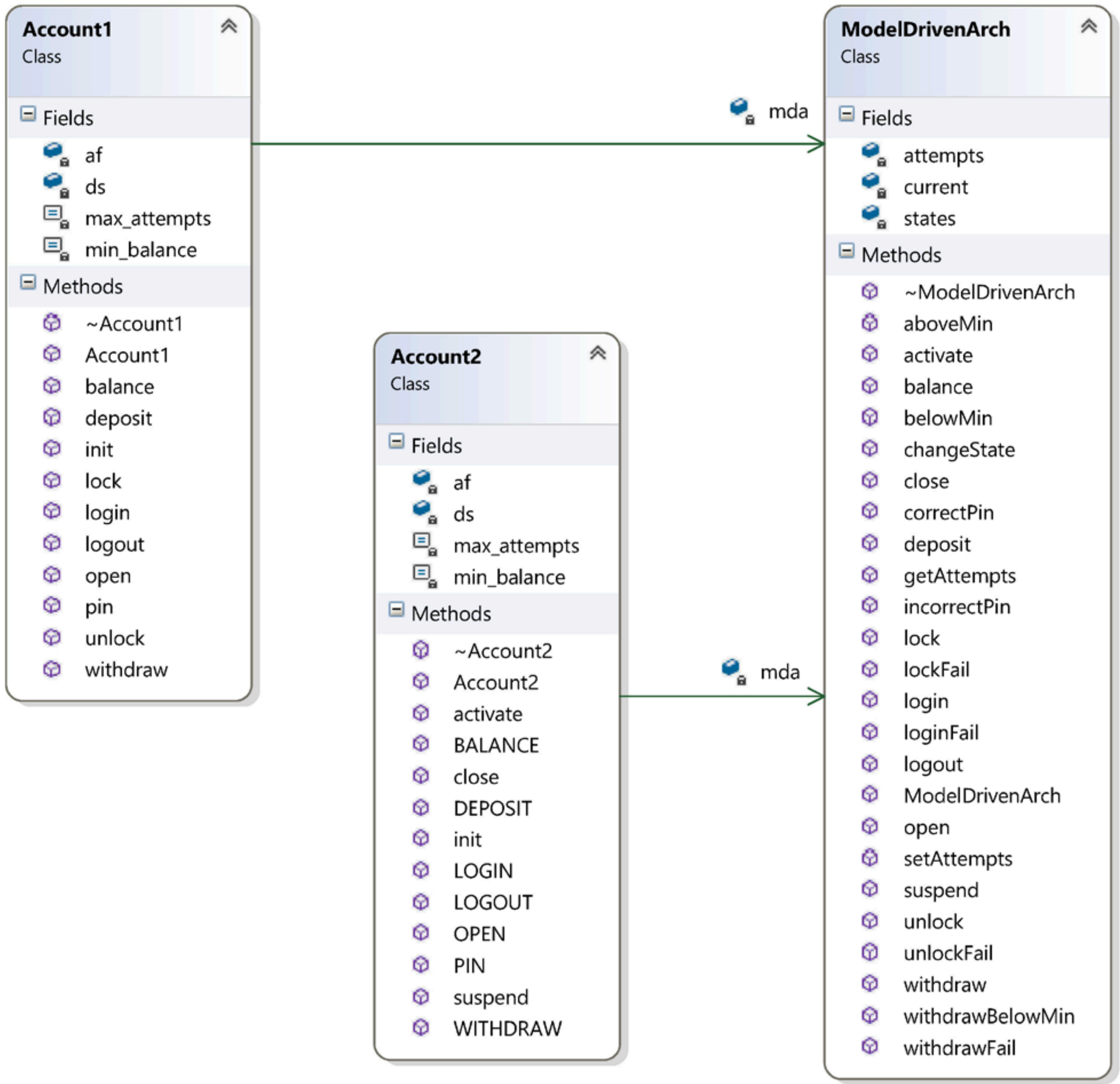
Class: DataStore1	
Purpose	Data store used in implementing Account2's logic
Member variables	balance: int type, represent account balance id: int type, represent account id pin: int type, represent PIN number temp_balance: int type, temporal data used for open() temp_pin: int type, temporal data used for open() temp_id: int type, temporal data used for open() temp_d: int type, temporal data used for deposit() temp_w: int type, temporal data used for withdraw()
Operations	Various getter and setter for member variables



6. Details for Other Classes: Accounts

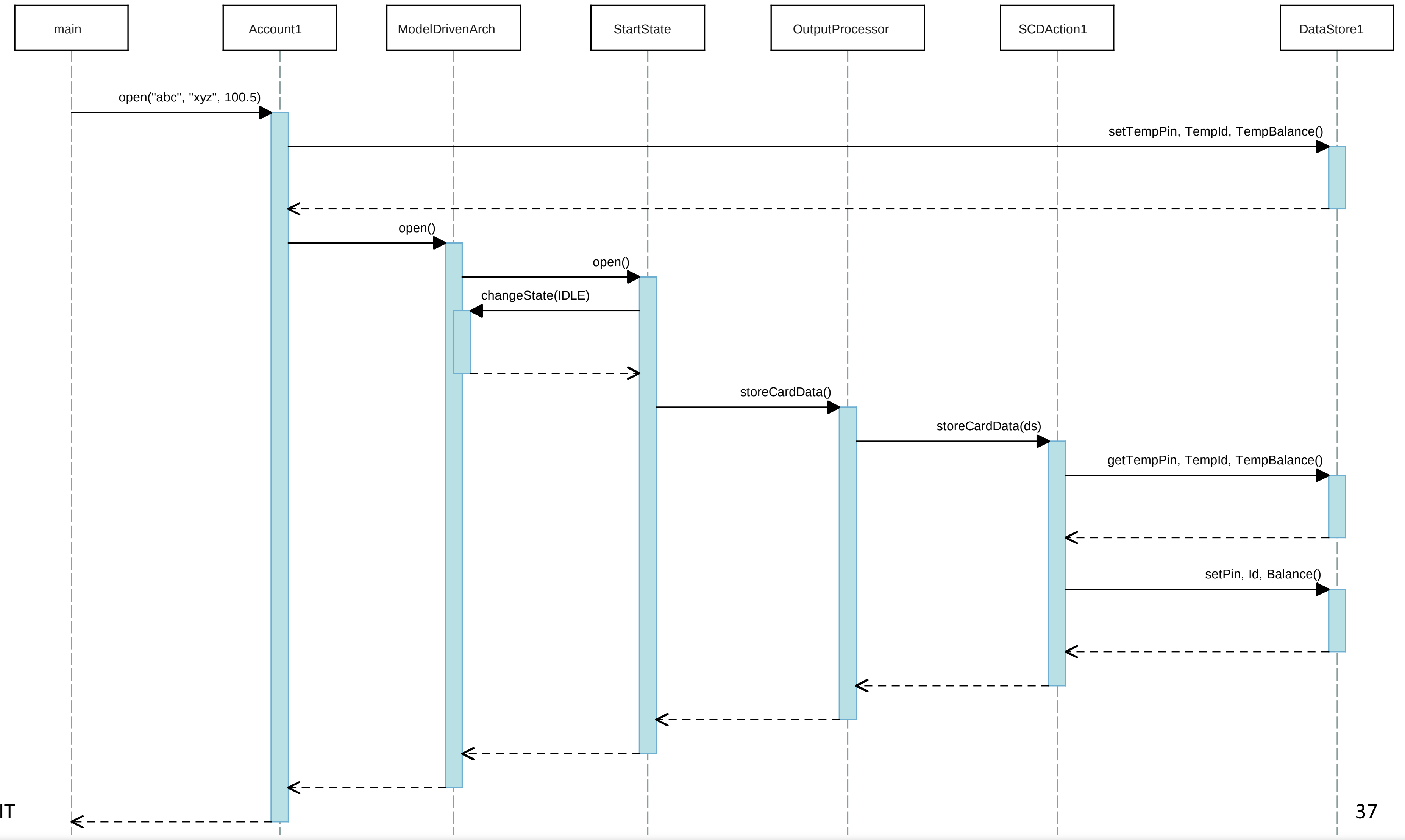
Class: Account1	
Purpose	Play the role of InputProcessor in Model-driven architecture
Member variables	*mda*: pointer to ModelDrivenArch instance *af*: pointer to an AbstractFactory instance *ds*: pointer to its DataStore instance *max_attempts*: maximum number of incorrect PIN allowed *min_balance*: minimum balance amount
Constructor()	Initialize *mda, af, max_attempts* and *min_balance*
Destructor()	Reclaim/free DataStore1 object *ds*
Operations	Events in Account1-EFSM; each operation's pseudo code is shown in section 1, page 5.

Class: Account2	
Purpose	Play the role of InputProcessor in Model-driven architecture
Member variables	*mda*: pointer to ModelDrivenArch instance *af*: pointer to an AbstractFactory instance *ds*: pointer to its DataStore instance *max_attempts*: maximum number of incorrect PIN allowed *min_balance*: minimum balance amount
Constructor()	Initialize *mda, af, max_attempts* and *min_balance*
Destructor()	Reclaim/free DataStore2 object *ds*
Operations	Events in Account2-EFSM; each operation's pseudo code is shown in section 1, page 6.



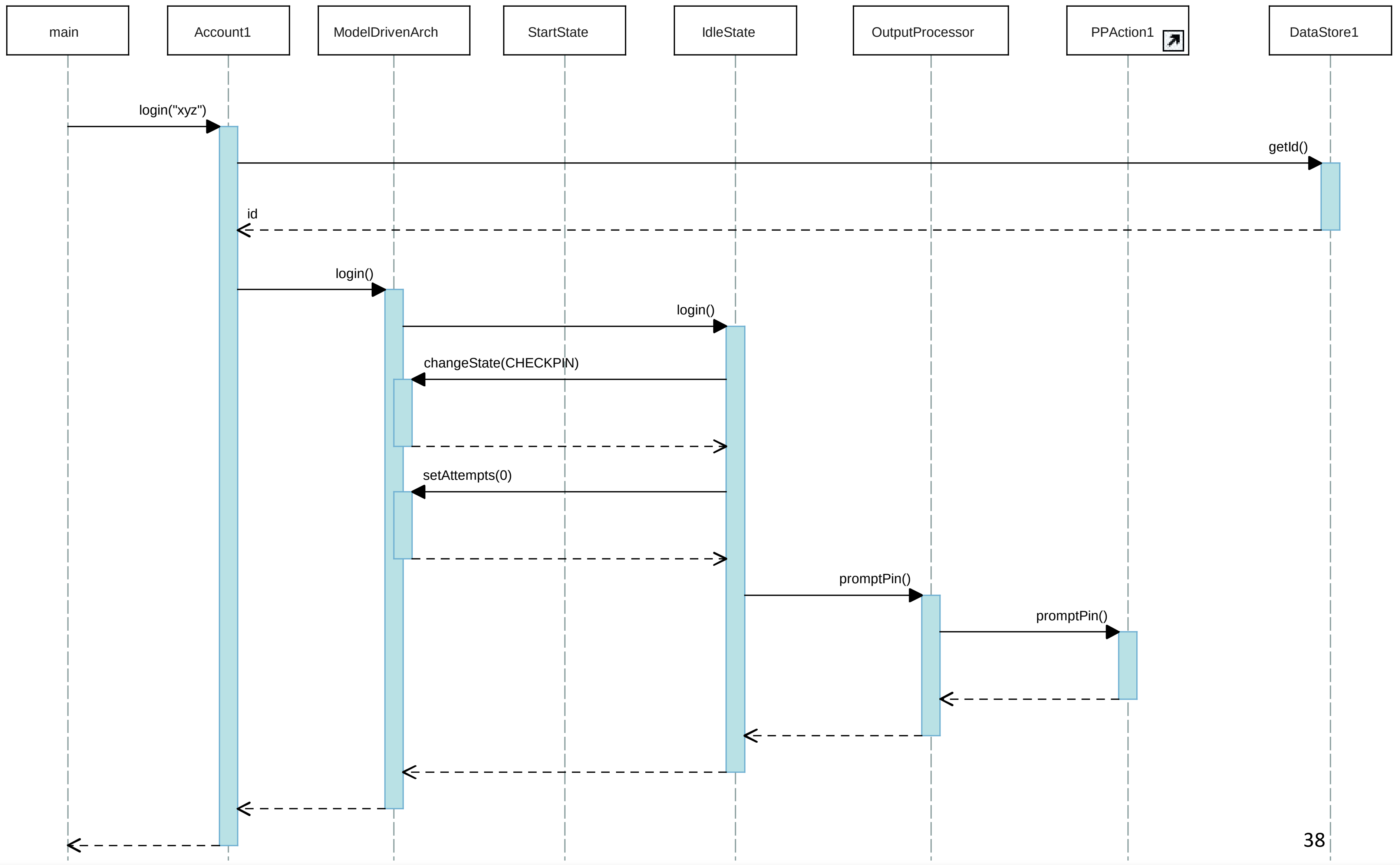
7. Dynamics for Two Scenarios

Scenario 1, *open(abc, xyz, 100.5)*



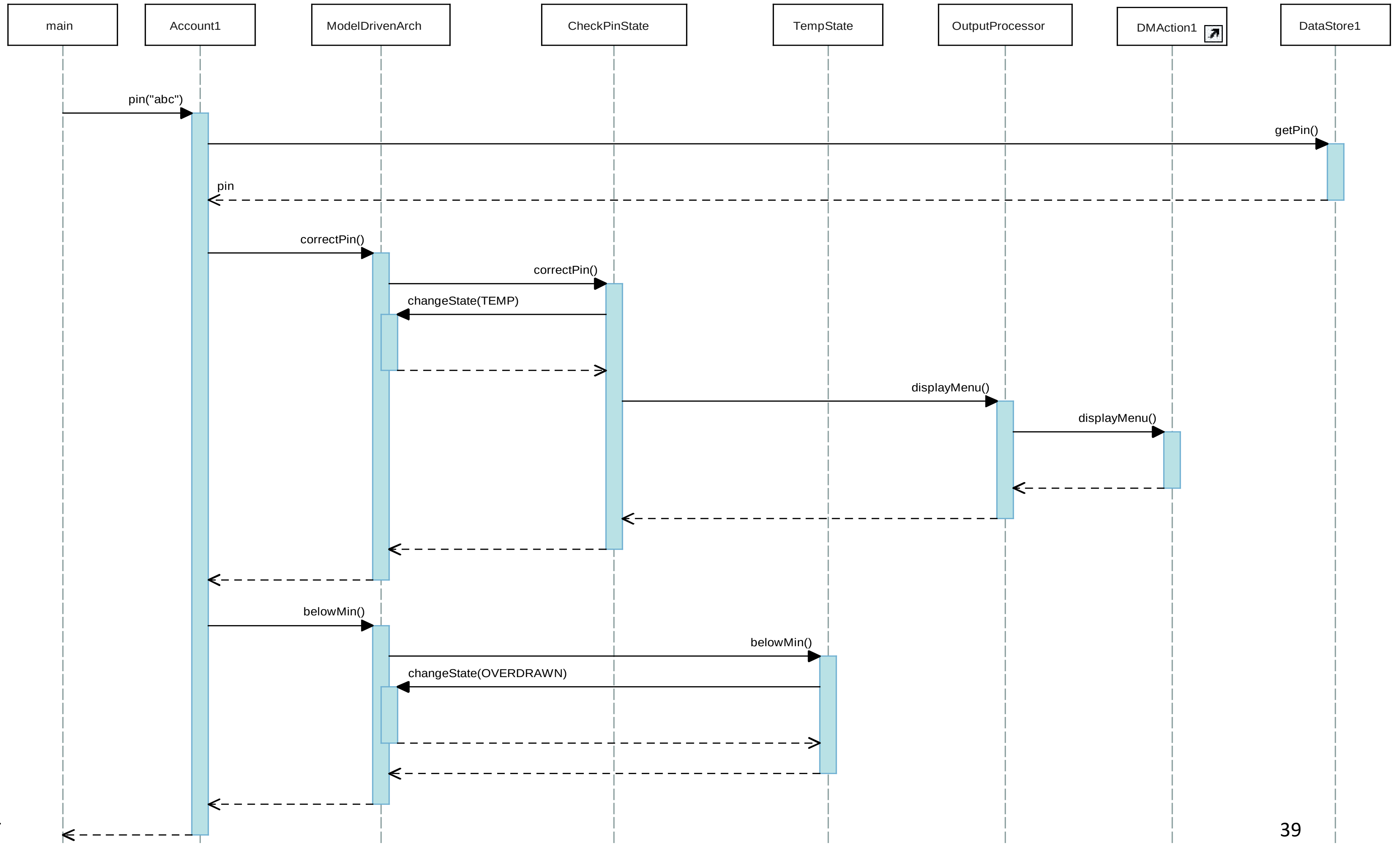
7. Dynamics for Two Scenarios

Scenario 1, *login(xyz)*



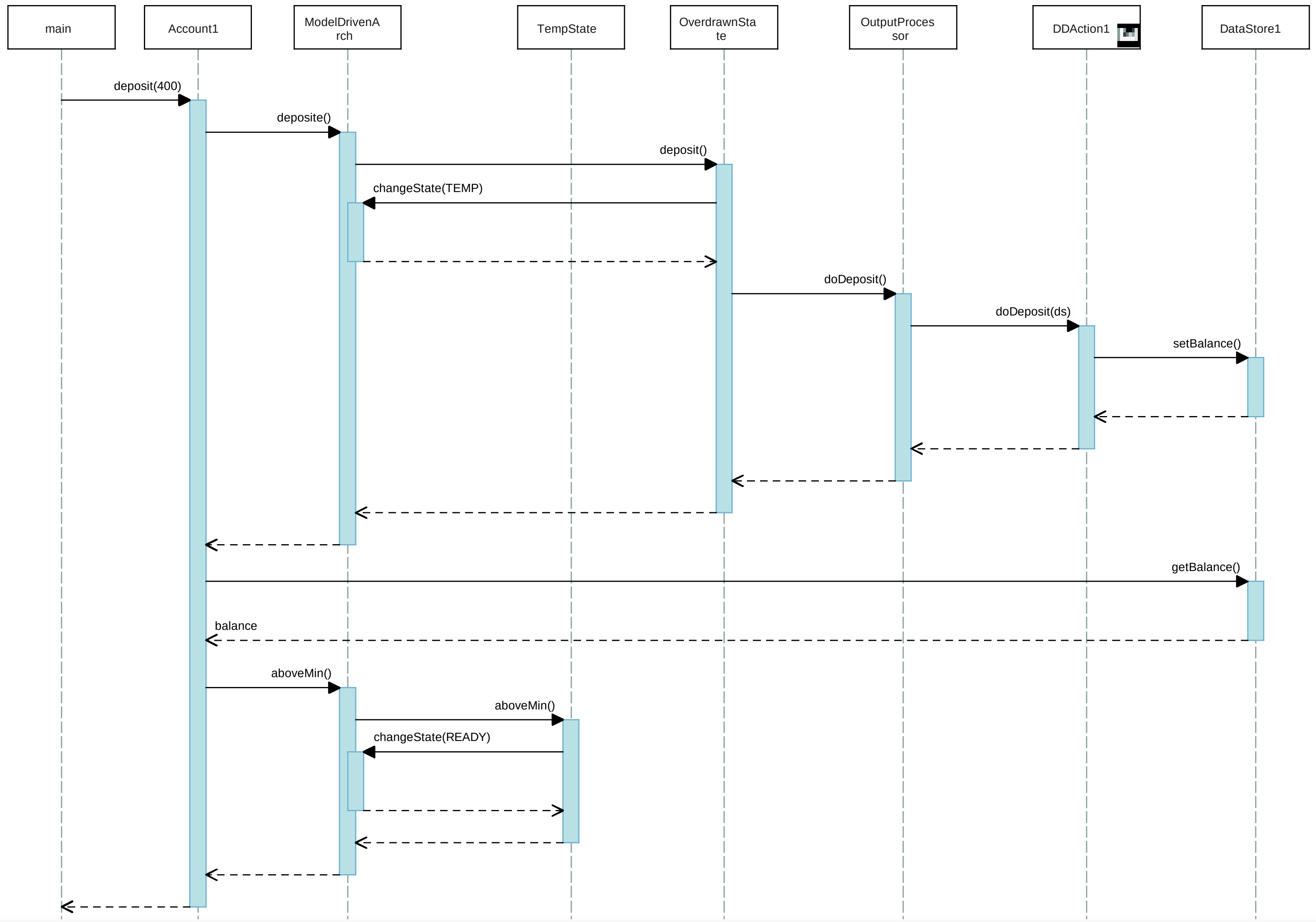
7. Dynamics for Two Scenarios

<u>Scenario 1</u>, *pin(abc)*



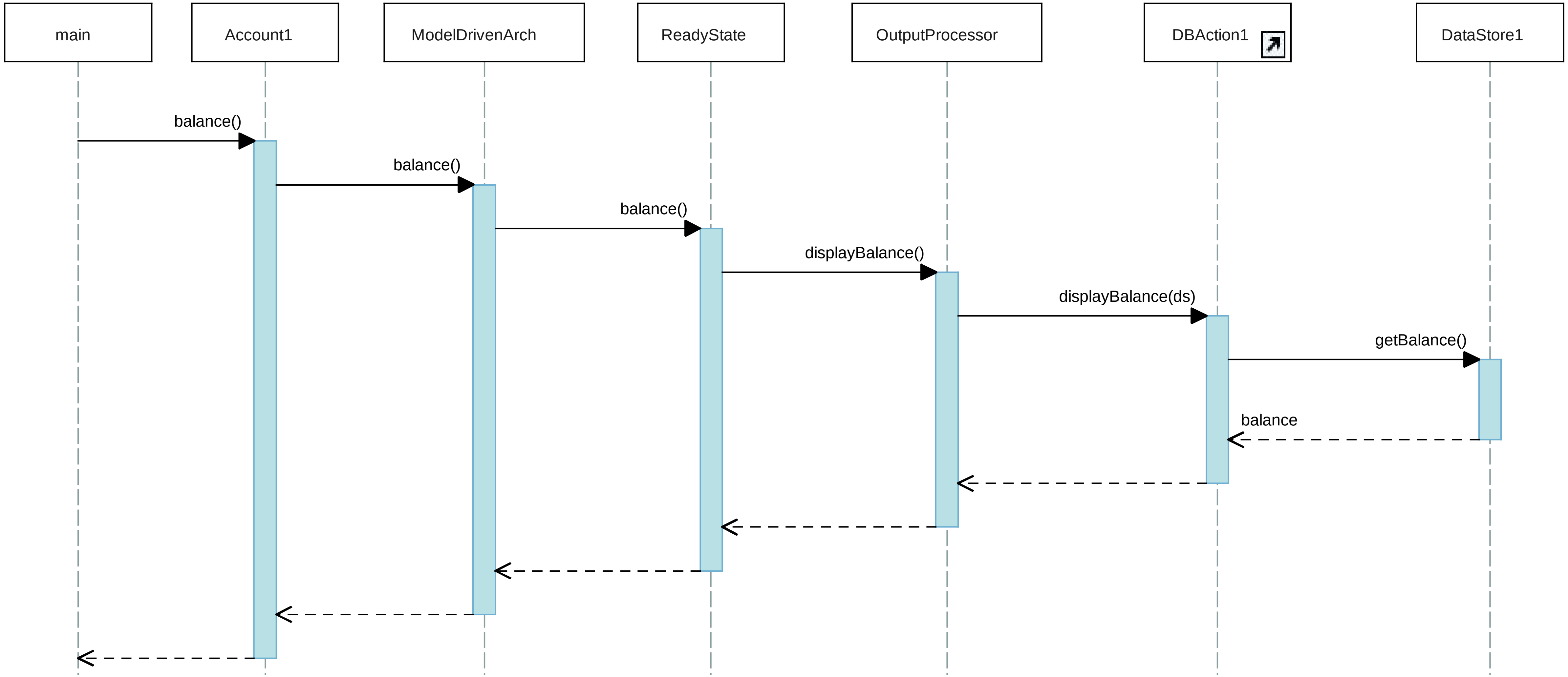
7. Dynamics for Two Scenarios

Scenario 1, *deposit(400)*



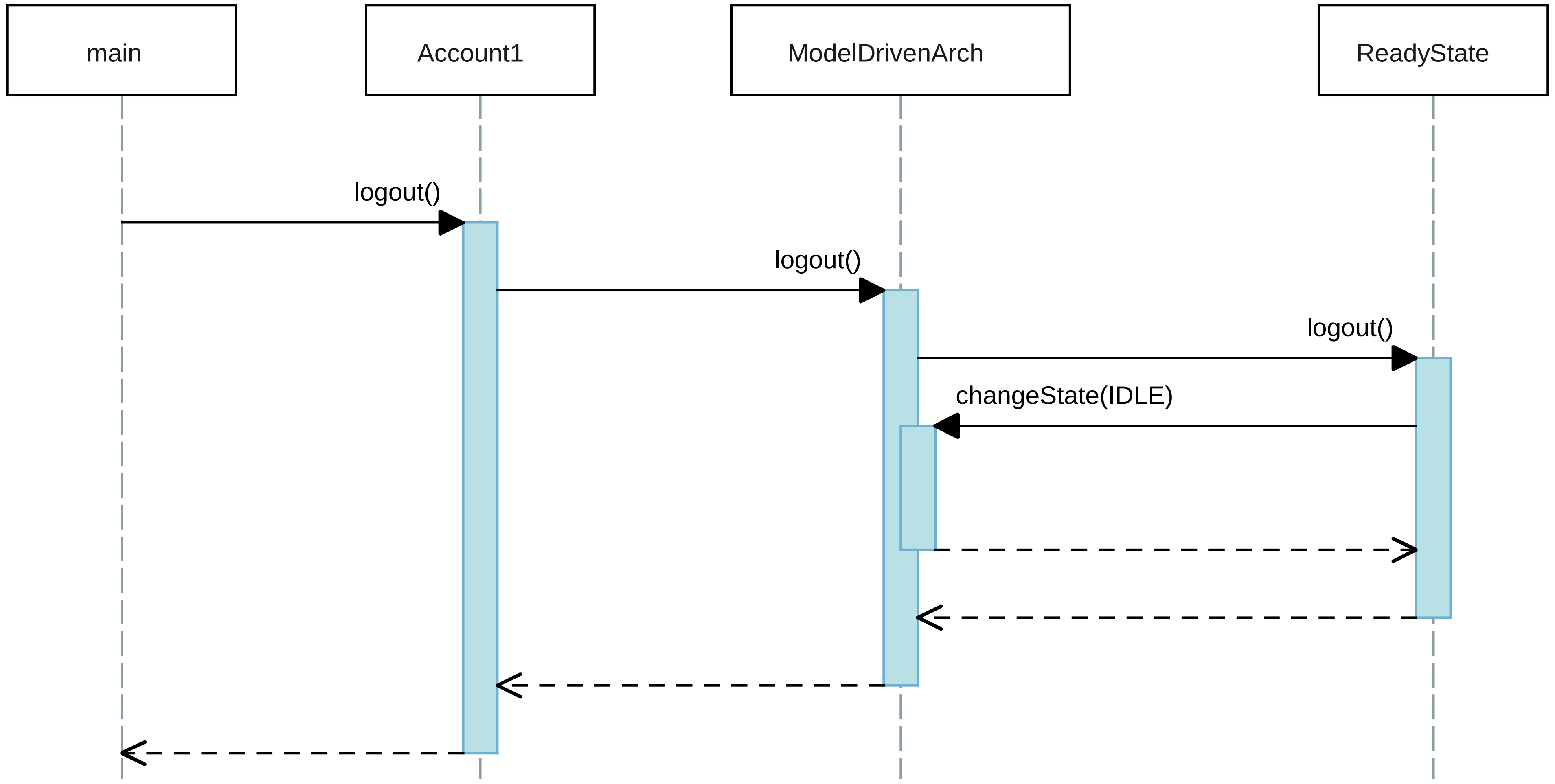
7. Dynamics for Two Scenarios

Scenario 1, *balance()*



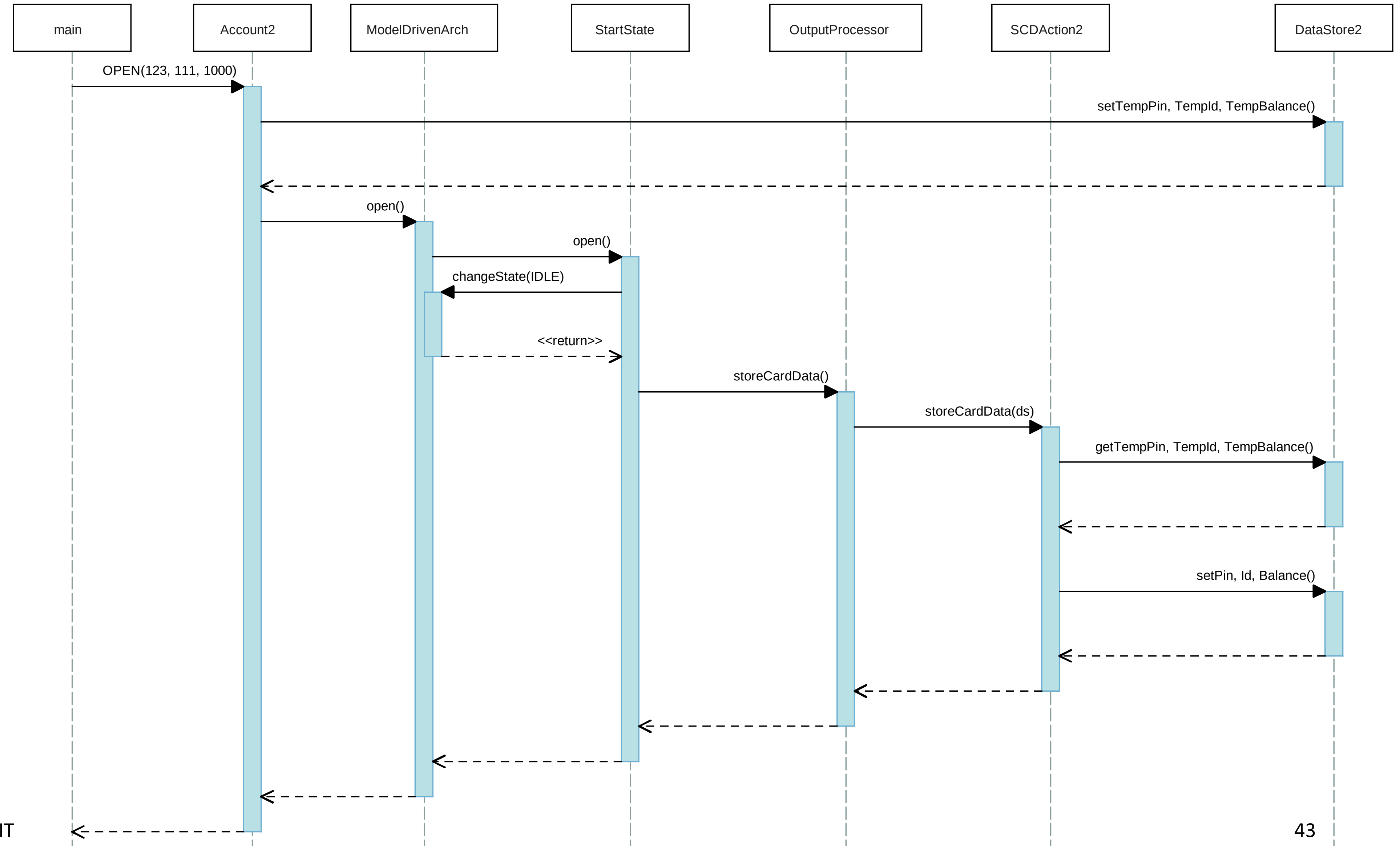
7. Dynamics for Two Scenarios

Scenario 1, *logout()*



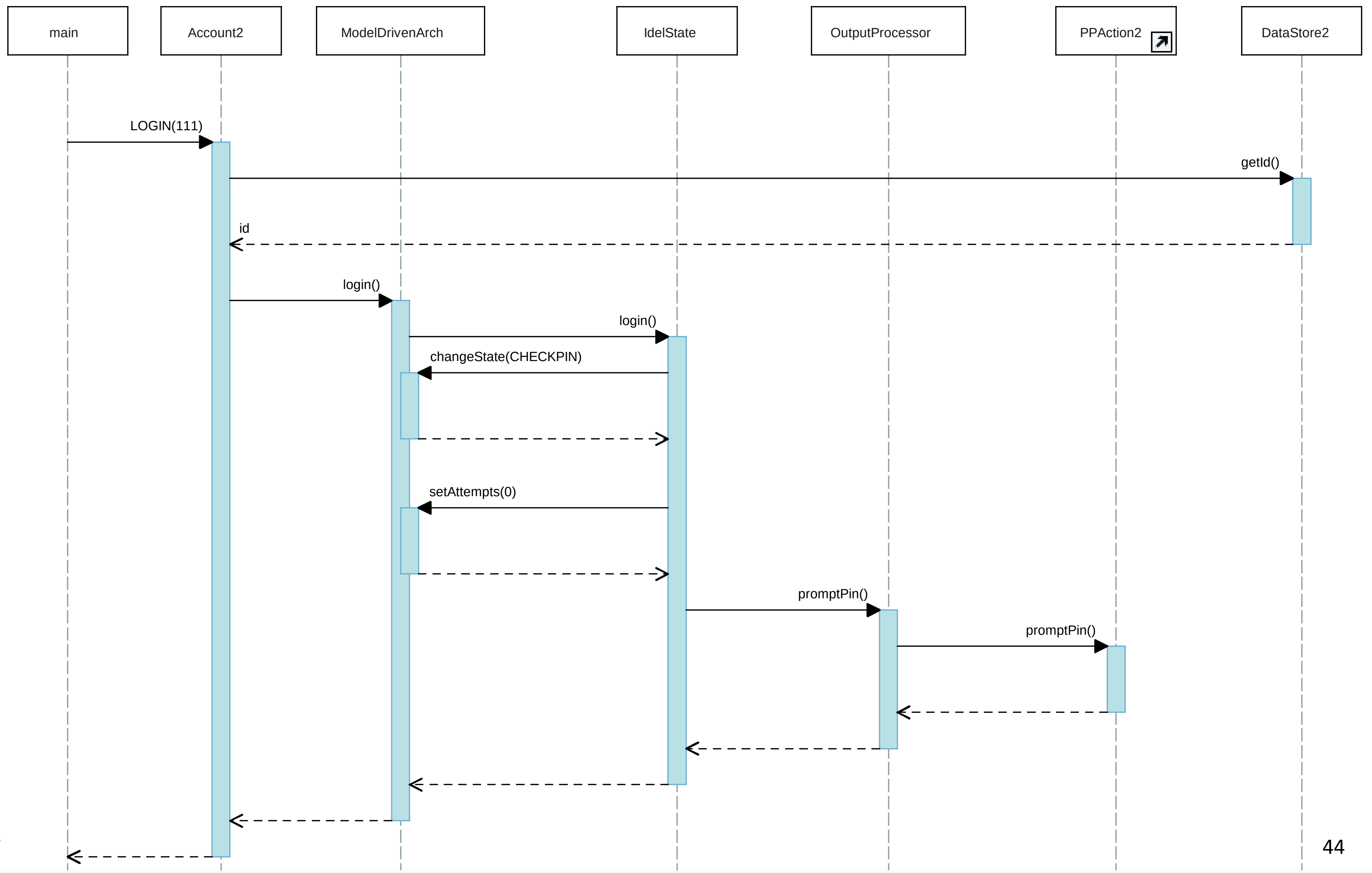
7. Dynamics for Two Scenarios

Scenario 2, *OPEN()*



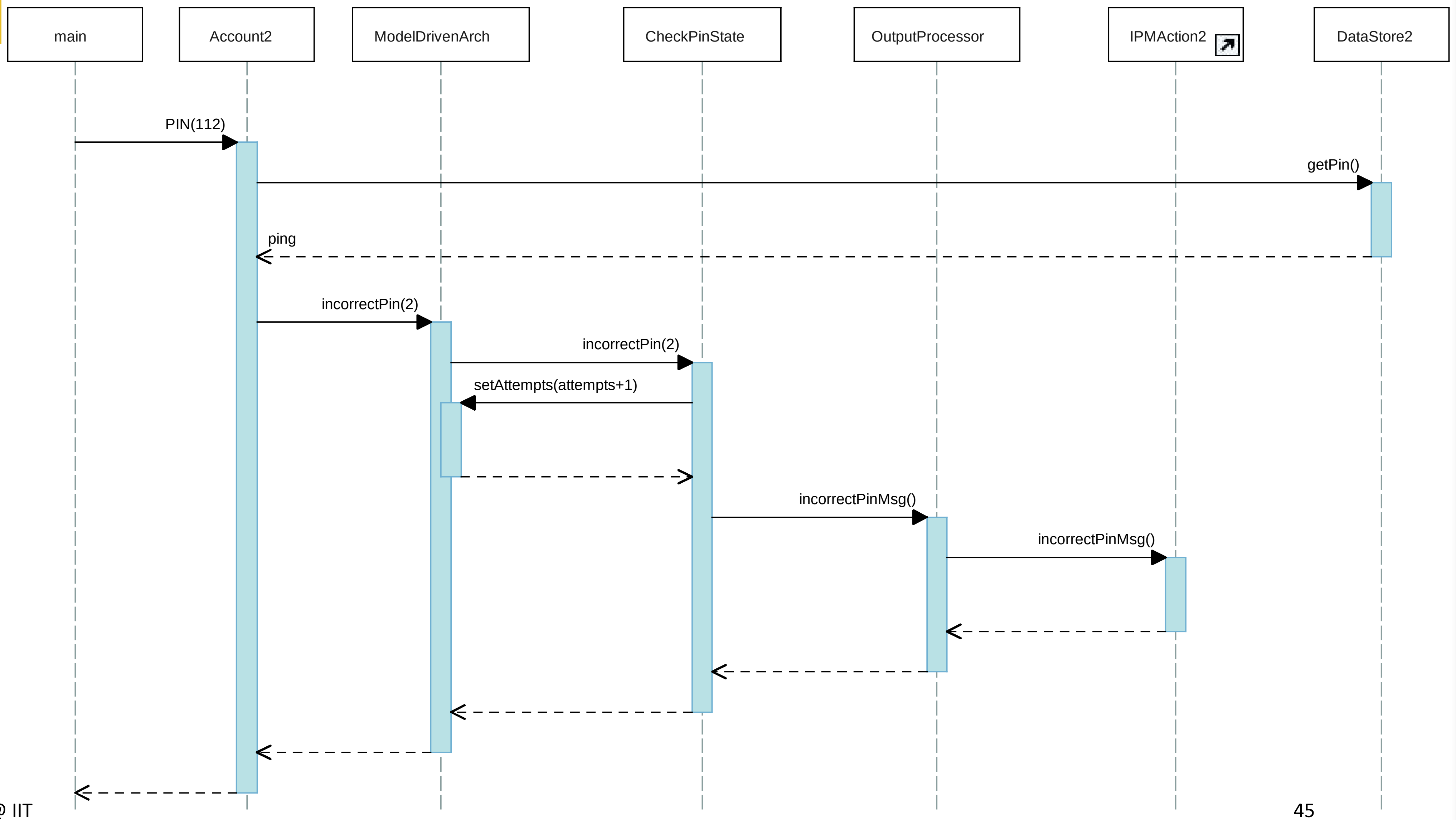
7. Dynamics for Two Scenarios

Scenario 2, *LOGIN()*



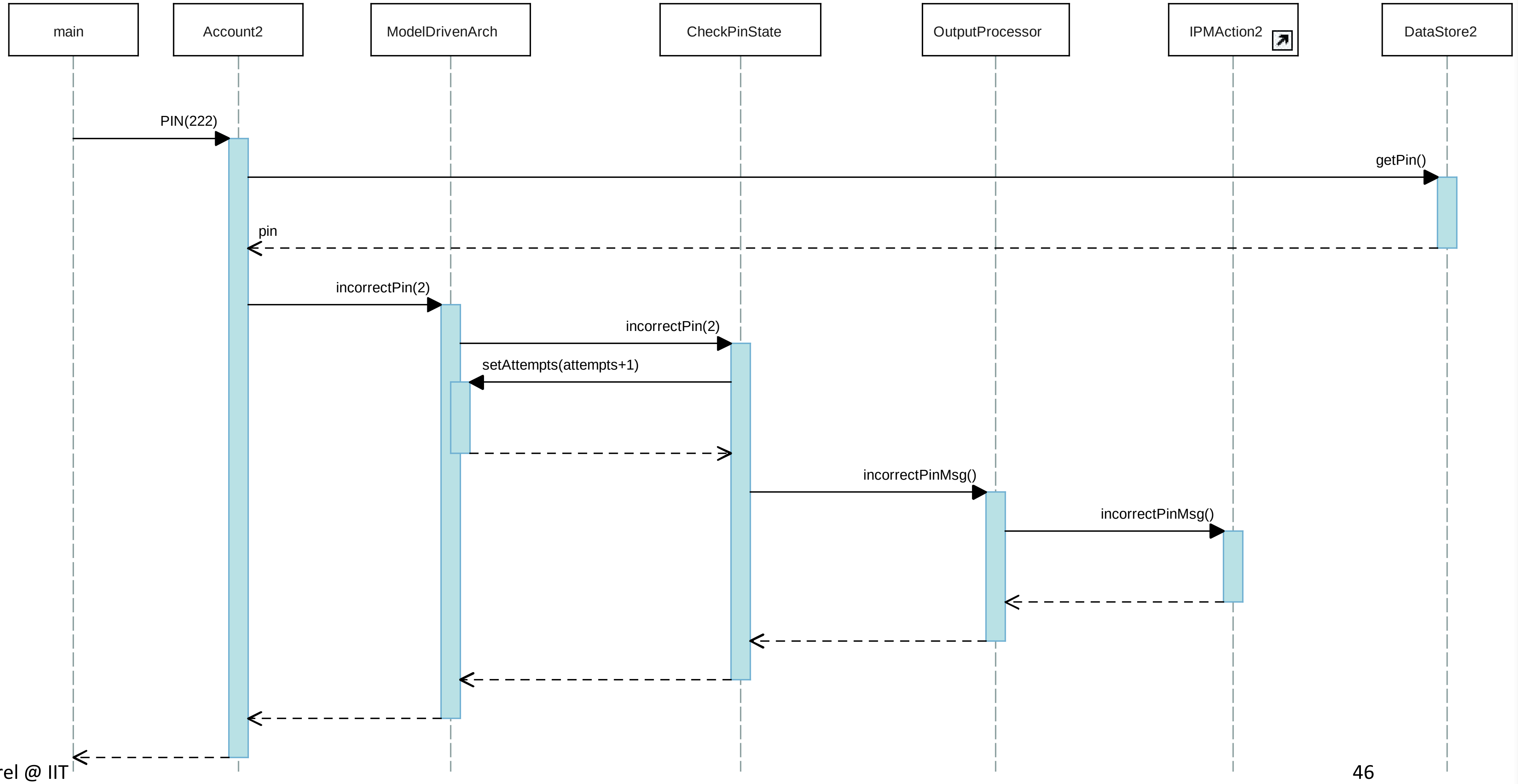
7. Dynamics for Two Scenarios

Scenario 2, *PIN(112)*



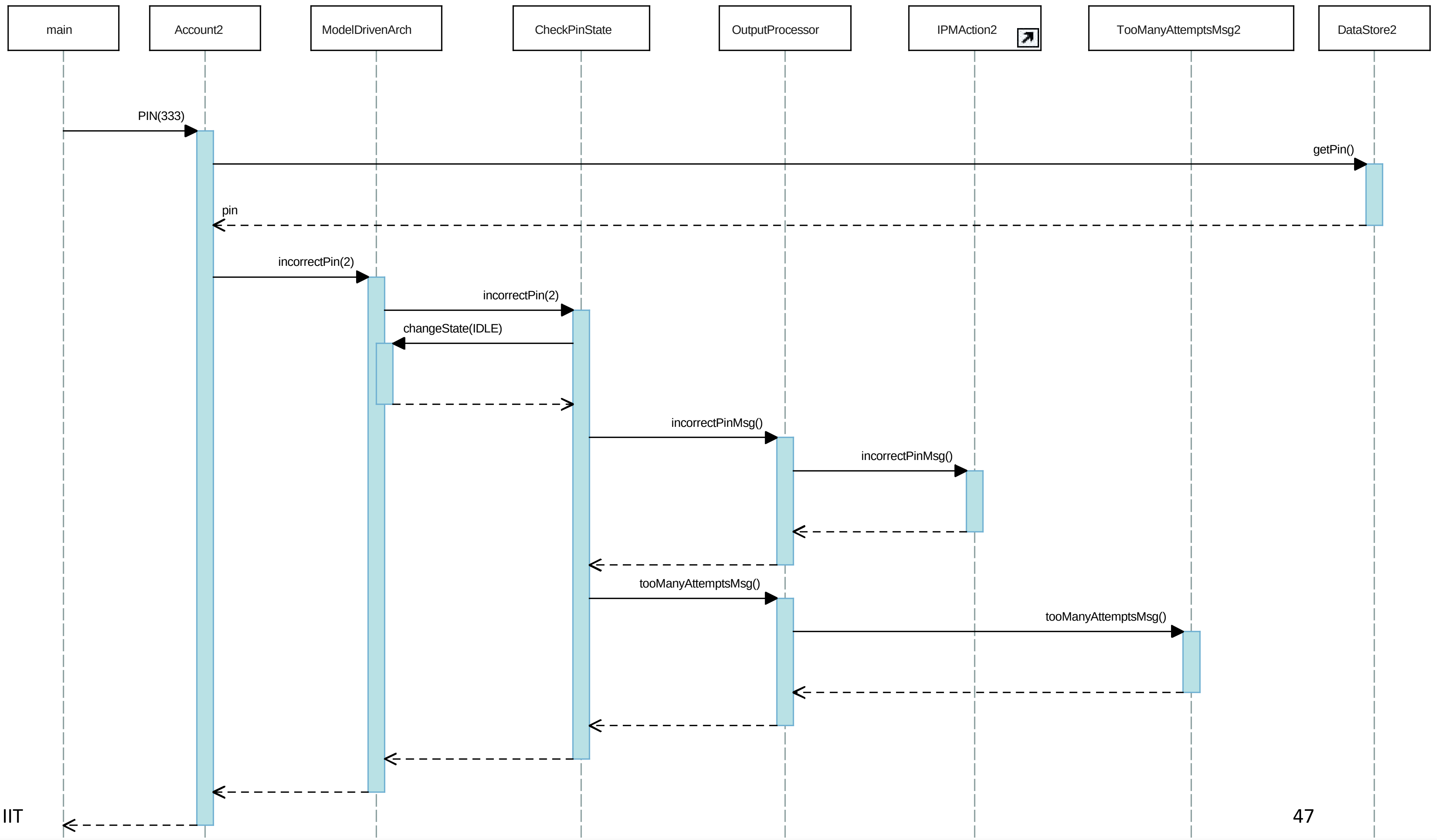
7. Dynamics for Two Scenarios

Scenario 2, *PIN(222)*



7. Dynamics for Two Scenarios

Scenario 2, *PIN(333)*



8. Source Code and Patterns

- State Pattern

State pattern is used for maintaining the state transition of MDA-EFSM. Its implementation locates at include/ModelDriveArch.hpp and src/ModelDriveArch.cpp. The classes involved in state pattern are:

ModelDrivenArch class: state machine context. since decentralized state pattern is used, this class delegate all events handling to the state classes.

State class: parent of all states class; it provides default event handling, e.g. do nothing.

Concrete state classes, including **StartState, IdleState, CheckPinState, ReadyState, OverdrawnState, LockedState, SuspendedState, CloseState** and **TempState**

8. Source Code and Patterns

- Strategy Pattern

  Strategy pattern groups the actions of different accounts. Its implementation locates at include/Actions.hpp and src/Actions.cpp. The classes involved in strategy pattern are:

  Each meta action corresponds to an abstract strategy class, including **StoreCardDataAction, IncorrectPinMsgAction, IncorrectIdMsgAction, TooManyAttemptMsgAction, PromptPinAction, DisplayMenuAction, DoDepositAction, NoFundMsgAction, DisplayBalanceAction, DoWithdrawAction, BelowMinMsgAction, DeductPenaltyAction**

  For each strategy, 2 concrete strategies are provided for either Account1 or Account2, including **SCDAction1, SCDAction2, IPMAction1, IPMAction2, IIMAction1, IIMAction2, TMAMAction1, TMAMAction2, PPAction1, PPAction2, DMAction1, DMAction2, DDAction1, DDAction2, NFMAction1, NFMAction2, DBAction1, DBAction2, DWAction1, DWAction2, BMMAction1, BMMAction2, DPAction1, DPAction2**

  OutputProcessor: the client class of strategies

8. Source Code and Patterns

- Abstract Factory Pattern

Abstract factory pattern eases the burden of create and configure strategies and data store for clients. Its implementation locates at include/AbstractFactory.hpp and src/AbstractFactory.cpp. Client of abstract factory are InputProcessors and Outprocessor, located at include/Accounts.hpp, includes/Actions.hpp, src/Accounts.cpp and src/Actions.cpp. The classes involved in strategy pattern are:

AbstractFactory class: defines the possible products could be created for its clients

ConcreteFactory1 and **ConcreteFactory2** classes: concrete classes for Account1 and Account2 respectively.

Account1, **Account2** and **OutputProcessor** classes: client of AbstractFactory

Concrete products created by concrete factories:

For Account1 and OutputProcessor: **DataStore1, SCDAction1, IPMAction1, IIMAction1, TMAMAction1, PPAction1, DMAction1, DDAction1, NFMAction1, DBAction1, DWAction1, BMMAction1, DPAction1**

For Account2 and OutputProcessor**: DataStore2, SCDAction2, IPMAction2, IIMAction2, TMAMAction2, PPAction2, DMAction2, DDAction2, NFMAction2, DBAction2, DWAction2, BMMAction2, DPAction2**

9. Source Code

This section presents the C++ source code. The project is developed under CLion and Vim. To make it be able to compile under various OS platform, I choose to use CMake. The CMakeList.txt file is used to generate Makefile. There are 3 major directory. Class declaration files (header files) are in *include*; implementation of all classes are in .cpp files under *src* directory. Built binary executable are in *build*, which also contains generated Makefile. To rebuild the project, simple use the following commands:

```
cd build            # change to build directory
cmake ..            # generate build scripts
make                # complie, link and build executable main
```

There are 5 files under include: Accounts.hpp, AbstractFactory.hpp, Actions.hpp, ModelDrivenArch.hpp, DataStore.hpp. Correspondingly, there are 5 files under src: Accounts.cpp, AbstractFactory.cpp, Actions.cpp, ModelDrivenArch.cpp, main.cpp. Source code are presented in pair of (xxxx.hpp, xxxx.cpp).

There is also a README.md file explaining how to build and run the program. Basically, I adopt the sample driver to run Account1 and Account2. You should be able to run the program as the professor runs the in-class demo.

9. Source Code: Demo

Screenshot of running scenarios 1

```
BankAccount Version 1.0
Please choose the type of ACCOUNT
1. ACCOUNT-1
2. ACCOUNT-2
q. Quit the demo program
1
                    ACCOUNT-1
              MENU of Operations
        0. open(string, string, float)
        1. login(string)
        2. pin(string)
        3. deposit(float)
        4. withdraw(float)
        5. balance()
        6. logout()
        7. lock(string)
        8. unlock(string)
        q. Quit ACCOUNT-1
        ACCOUNT-1 Execution
 Select Operation:
0-open, 1-login, 2-pin, 3-deposit, 4-withdraw, 5-balance, 6-logout, 7-lock, 8-unlock
0
 Operation:  open(string p, string y, float a)
 Enter value of the parameter p:
abc
 Enter value of the parameter y:
xyz
 Enter value of the parameter a:
100.5
 Select Operation:
0-open, 1-login, 2-pin, 3-deposit, 4-withdraw, 5-balance, 6-logout, 7-lock, 8-unlock
1
 Operation:  login(string y)
 Enter value of the parameter y:
xyz
                            Please Input PIN to Proceed at Account1
 Select Operation:
0-open, 1-login, 2-pin, 3-deposit, 4-withdraw, 5-balance, 6-logout, 7-lock, 8-unlock
2
 Operation:  pin(string x)
 Enter value of the parameter x
abc
                            Menu at Account1
                                                    1. Display Balance
                                                    2. Make Deposit
                                                    3. Withdraw
                                                    4. Lock Account
                                                    5. Unlock Account
                                                    6. Logout
 Select Operation:
0-open, 1-login, 2-pin, 3-deposit, 4-withdraw, 5-balance, 6-logout, 7-lock, 8-unlock
3
 Operation:  deposit(float d)
 Enter value of the parameter d:
400
 Select Operation:
0-open, 1-login, 2-pin, 3-deposit, 4-withdraw, 5-balance, 6-logout, 7-lock, 8-unlock
5
 Operation:  balance()
                            Current Balance = $500.5 at Account1
 Select Operation:
0-open, 1-login, 2-pin, 3-deposit, 4-withdraw, 5-balance, 6-logout, 7-lock, 8-unlock
6
 Operation:  logout()
 Select Operation:
0-open, 1-login, 2-pin, 3-deposit, 4-withdraw, 5-balance, 6-logout, 7-lock, 8-unlock
q
Please choose the type of ACCOUNT
1. ACCOUNT-1
2. ACCOUNT-2
q. Quit the demo program
```

9. Source Code: Demo

Screenshot of running scenarios 2

```
BankAccount Version 1.0
Please choose the type of ACCOUNT
1. ACCOUNT-1
2. ACCOUNT-2
q. Quit the demo program
2
                    ACCOUNT-2
              MENU of Operations
        0. OPEN(int,int,int)
        1. LOGIN(int)
        2. PIN(int)
        3. DEPOSIT(int)
        4. WITHDRAW(int)
        5. BALANCE()
        6. LOGOUT()
        7. suspend()
        8. activate()
        9. close()
        q. Quit ACCOUNT-2
        ACCOUNT-2 Execution
 Select Operation:
0-OPEN, 1-LOGIN, 2-PIN, 3-DEPOSIT, 4-WITHDRAW, 5-BALANCE, 6-LOGOUT, 7-suspend, 8-activate, 9-close
0
 Operation:  OPEN(int p, int y, int a)
 Enter value of the parameter p:
123
 Enter value of the parameter y:
111
 Enter value of the parameter a:
1000
 Select Operation:
0-OPEN, 1-LOGIN, 2-PIN, 3-DEPOSIT, 4-WITHDRAW, 5-BALANCE, 6-LOGOUT, 7-suspend, 8-activate, 9-close
1
 Operation:  LOGIN(int y)
 Enter value of the parameter y:
111
                            Please Input PIN to Proceed at Account2
 Select Operation:
0-OPEN, 1-LOGIN, 2-PIN, 3-DEPOSIT, 4-WITHDRAW, 5-BALANCE, 6-LOGOUT, 7-suspend, 8-activate, 9-close
2
 Operation:  PIN(int x)
 Enter value of the parameter x
112
                            Incorrect Pin Input at Account 2
 Select Operation:
0-OPEN, 1-LOGIN, 2-PIN, 3-DEPOSIT, 4-WITHDRAW, 5-BALANCE, 6-LOGOUT, 7-suspend, 8-activate, 9-close
2
 Operation:  PIN(int x)
 Enter value of the parameter x
222
                            Incorrect Pin Input at Account 2
 Select Operation:
0-OPEN, 1-LOGIN, 2-PIN, 3-DEPOSIT, 4-WITHDRAW, 5-BALANCE, 6-LOGOUT, 7-suspend, 8-activate, 9-close
2
 Operation:  PIN(int x)
 Enter value of the parameter x
333
                            #Attempts Exceed Maximum Allowed at Account2
 Select Operation:
0-OPEN, 1-LOGIN, 2-PIN, 3-DEPOSIT, 4-WITHDRAW, 5-BALANCE, 6-LOGOUT, 7-suspend, 8-activate, 9-close
q
Please choose the type of ACCOUNT
1. ACCOUNT-1
2. ACCOUNT-2
q. Quit the demo program
q
```

9. Source Code: Accounts.hpp

```
/**
 * Declaration of two different logic/implementation:
 * Account1 and Account2, as Input Processor in Model-
 * driven Architecture
 */

#ifndef _ACCOUNTS_HPP
#define _ACCOUNTS_HPP

#include "ModelDrivenArch.hpp"
#include "DataStore.hpp"

class AbstractFactory;

class Account1 {
        private:
                ModelDrivenArch *mda;
                DataStore1 *ds;
                AbstractFactory *af;
                const int min_balance; /* minimum balance */
                const int max_attempts; /* max number of PIN attempts */
        public:
                Account1(ModelDrivenArch *m, AbstractFactory *a, int mb = 500, int ma =
3):
                        mda(m), af(a), min_balance(mb), max_attempts(ma) {};
                virtual ~Account1() {
                        delete ds;
                }

                /**
                 * Init DataStore with AbstractFactory
                 */
                void init();

                /**
                 * Possible events happening to Account1
                 */
                void open(string p, string y, float a);
                void pin(string x);
                void deposit(float d);
                void withdraw(float w);
                void balance();
                void login(string y);
                void logout();
                void lock(string x);
                void unlock(string x);
};

class Account2 {
        private:
                ModelDrivenArch *mda;
                DataStore2 *ds;
                AbstractFactory *af;
                const int min_balance;
                const int max_attempts;
        public:
                Account2(ModelDrivenArch *m, AbstractFactory *a, int mb = 0, int ma = 2):
                        mda(m), af(a), min_balance(mb), max_attempts(ma) {};
                virtual ~Account2() {
                        delete ds;
                }

                /**
                 * Init DataStore with AbstractFactory
                 */
                void init();

                /**
                 * Possible events happening to Account2
                 */
                void OPEN(int p, int y, int a);
                void PIN(int x);
                void DEPOSIT(int d);
                void WITHDRAW(int w);
                void BALANCE();
                void LOGIN(int y);
                void LOGOUT();
                void suspend();
                void activate();
                void close();
};

#endif
```

9. Source Code: Accounts.cpp

```cpp
/**
 * An implementation of the pseudo-code
 * of Input Processors, account1 and account2
 */

#include "Accounts.hpp"
#include "AbstractFactory.hpp"

/**
 * To use ds properly, still need to cast it
to DataStore2
 */
void Account1::init() {
        ds = (DataStore1 *)(af->createDS());
}

void Account1::open(string p, string y, float
a) {
        ds->setTempPin(p);
        ds->setTempId(y);
        ds->setTempBalance(a);
        mda->open();
}

void Account1::pin(string x) {
        if (x == ds->getPin()) {
                mda->correctPin();
                if (ds->getBalance() >
min_balance) {
                        mda->aboveMin();
                } else {
                        mda->belowMin();
                }
        } else {
                mda-
>incorrectPin(max_attempts);
        }
}

void Account1::deposit(float d) {
        ds->setTempD(d);
        mda->deposit();
        if (ds->getBalance() > min_balance) {
                mda->aboveMin();
        } else {
                mda->belowMin();
        }
}


void Account1::withdraw(float w) {
        ds->setTempW(w);
        if (ds->getBalance() <= min_balance) {
                mda->withdrawFail();
        } else {
                mda->withdraw();
        }
        if (ds->getBalance() > min_balance) {
                mda->aboveMin();
        } else {
                mda->withdrawBelowMin();
        }
}

void Account1::balance() {
        mda->balance();
}

void Account1::login(string y) {
        if (y == ds->getId()) {
                mda->login();
        } else {
                mda->loginFail();
        }
}

void Account1::logout() {
        mda->logout();
}

void Account1::lock(string x) {
        if (x == ds->getPin()) {
                mda->lock();
        } else {
                mda->lockFail();
        }
}

void Account1::unlock(string x) {
        if (x == ds->getPin()) {
                mda->unlock();
                if (ds->getBalance() >
min_balance) {
                        mda->aboveMin();
                } else {
                        mda->belowMin();
                }
        } else {
                mda->unlockFail();
        }
}

/**
 * To use ds properly, cast it to DataStore2
 */
void Account2::init() {
        ds = (DataStore2 *)af->createDS();
}

void Account2::OPEN(int p, int y, int a) {
        ds->setTempPin(p);
        ds->setTempId(y);
        ds->setTempBalance(a);
        mda->open();
}

void Account2::PIN(int x) {
        if (x == ds->getPin()) {
                mda->correctPin();
                mda->aboveMin();
        } else {
                mda-
>incorrectPin(max_attempts);
        }
}

void Account2::DEPOSIT(int d) {
        ds->setTempD(d);
        mda->deposit();
        if (ds->getBalance() > min_balance) {
                mda->aboveMin();
        } else {
                mda->belowMin();
        }
}

void Account2::WITHDRAW(int w) {
        ds->setTempW(w);
        if (ds->getBalance() > min_balance) {
                mda->withdraw();
                mda->aboveMin();
        } else {
                mda->withdrawFail();
        }
}

void Account2::BALANCE() {
        mda->balance();
}

void Account2::LOGIN(int y) {
        if (y == ds->getId()) {
                mda->login();
        } else {
                mda->loginFail();
        }
}

void Account2::LOGOUT() {
        mda->logout();
}

void Account2::suspend() {
        mda->suspend();
}

void Account2::activate() {
        mda->activate();
}

void Account2::close() {
        mda->close();
}
```

9. Source Code: AbstractFactory.hpp

```cpp
/**
 * Declaration of abstract and concrete factories,
 * to form Abstract Factory Pattern
 */

#ifndef _ABSTRACTFACTORY_HPP
#define _ABSTRACTFACTORY_HPP

#include "DataStore.hpp"
#include "Actions.hpp"

/**
 * Abstract factory is ABSTRACT class, declaring
 * various of pure virtual functions that helps client
 * create/get different products
 */
class AbstractFactory {
        protected:
                /**
                 * Various objects created by this factory.
                 */
                DataStore *cacheDS;
                StoreDataAction *cacheSCDA;
                IncorrectPinMsgAction *cacheIPMA;
                IncorrectIdMsgAction *cacheIIMA;
                TooManyAttemptMsgAction *cacheTMAMA;
                PromptPinAction *cachePPA;
                DisplayMenuAction *cacheDMA;
                DoDepositAction *cacheDDA;
                NoFundMsgAction *cacheNFMA;
                DisplayBalanceAction *cacheDBA;
                DoWithdrawAction *cacheDWA;
                BelowMinMsgAction *cacheBMMA;
                DeductPenaltyAction *cacheDPA;
        public:
                AbstractFactory();
                virtual ~AbstractFactory();

                /**
                 * Create instances of various classes
                 */
                virtual DataStore *createDS() = 0;
                virtual StoreDataAction *createSDA() = 0;
                virtual IncorrectPinMsgAction *createIPMA() =
0;
                virtual IncorrectIdMsgAction *createIIMA() =
0;
                virtual TooManyAttemptMsgAction *createTMAMA()
= 0;
                virtual PromptPinAction *createPPA() = 0;
                virtual DisplayMenuAction *createDMA() = 0;
                virtual DoDepositAction *createDDA() = 0;
                virtual NoFundMsgAction *createNFMA() = 0;
                virtual DisplayBalanceAction *createDBA() = 0;
                virtual DoWithdrawAction *createDWA() = 0;
                virtual BelowMinMsgAction *createBMMA() = 0;
                virtual DeductPenaltyAction *createDPA() = 0;
};

/**
 * Concrete factory used specially by Account1 instance.
 * MUST override pure virtual functions in AbstractFactory.
 */
class ConcreteFactory1: public AbstractFactory {
        public:
                ConcreteFactory1() {};
                virtual ~ConcreteFactory1() {};

                virtual DataStore *createDS();
                virtual StoreDataAction *createSDA();
                virtual IncorrectPinMsgAction *createIPMA();
                virtual IncorrectIdMsgAction *createIIMA();
                virtual TooManyAttemptMsgAction
*createTMAMA();
                virtual PromptPinAction *createPPA();
                virtual DisplayMenuAction *createDMA();
                virtual DoDepositAction *createDDA();
                virtual NoFundMsgAction *createNFMA();
                virtual DisplayBalanceAction *createDBA();
                virtual DoWithdrawAction *createDWA();
                virtual BelowMinMsgAction *createBMMA();
                virtual DeductPenaltyAction *createDPA();
};

/**
 * Concrete factory used specially by Account2 instance
 * MUST override pure virtual functions in AbstractFactory.
 */
class ConcreteFactory2: public AbstractFactory {
        public:
                ConcreteFactory2() {};
                virtual ~ConcreteFactory2() {};

                virtual DataStore *createDS();
                virtual StoreDataAction *createSDA();
                virtual IncorrectPinMsgAction *createIPMA();
                virtual IncorrectIdMsgAction *createIIMA();
                virtual TooManyAttemptMsgAction
*createTMAMA();
                virtual PromptPinAction *createPPA();
                virtual DisplayMenuAction *createDMA();
                virtual DoDepositAction *createDDA();
                virtual NoFundMsgAction *createNFMA();
                virtual DisplayBalanceAction *createDBA();
                virtual DoWithdrawAction *createDWA();
                virtual BelowMinMsgAction *createBMMA();
                virtual DeductPenaltyAction *createDPA();
};

#endif
```

9. Source Code: AbstractFactory.cpp, part 1

```cpp
#include "AbstractFactory.hpp"

/**
 * Explicitly initialize cacheX to NULL
 */
AbstractFactory::AbstractFactory():
	cacheDS(NULL), cacheSCDA(NULL),
	cacheIPMA(NULL), cacheIIMA(NULL),
	cacheTMAMA(NULL), cachePPA(NULL),
	cacheDMA(NULL), cacheDDA(NULL),
	cacheNFMA(NULL), cacheDBA(NULL),
	cacheDWA(NULL), cacheBMMA(NULL),
	cacheDPA(NULL) {
}

/**
 * Abstract Factory is NOT responsible for reclaim
 * created/allocated objects.
 * For various actions, OutputProcessor reclaims them
 * For data stores, Account reclaims them
 */
AbstractFactory::~AbstractFactory() {
}

/**
 * Account class and OutputProcessor class
 * should operate on the same instance of DataStore.
 */
DataStore *ConcreteFactory1::createDS() {
	if (!cacheDS) {
		cacheDS = new DataStore1();
	}
	return cacheDS;
}

DataStore *ConcreteFactory2::createDS() {
	if (!cacheDS) {
		cacheDS = new DataStore2();
	}
	return cacheDS;
}
```

```cpp
/**
 * We can let factory return new instance of strategy/action
 * every time OutputProcessor requests it.
 * HOWEVER, in this project, one instance is enough.
 */

/**
 * Create concrete instance of StoreCardDataAction
 */
StoreDataAction *ConcreteFactory1::createSDA() {
	if (!cacheSCDA) {
		cacheSCDA = new SCDAction1();
	}
	return cacheSCDA;
}

StoreDataAction *ConcreteFactory2::createSDA() {
	if (!cacheSCDA) {
		cacheSCDA = new SCDAction2();
	}
	return cacheSCDA;
}

/**
 * Create concrete instance of IncorrectPinMsgAction
 */
IncorrectPinMsgAction *ConcreteFactory1::createIPMA() {
	if (!cacheIPMA) {
		cacheIPMA = new IPMAction1();
	}
	return cacheIPMA;
}

IncorrectPinMsgAction *ConcreteFactory2::createIPMA() {
	if (!cacheIPMA) {
		cacheIPMA = new IPMAction2();
	}
	return cacheIPMA;
}
```

9. Source Code: AbstractFactory.cpp, part 2

```
/**
 * Create concrete instance of IncorrectIdMsgAction
 */
IncorrectIdMsgAction *ConcreteFactory1::createIIMA() {
        if (!cacheIIMA) {
                cacheIIMA = new IIMAction1();
        }
        return cacheIIMA;
}

IncorrectIdMsgAction *ConcreteFactory2::createIIMA() {
        if (!cacheIIMA) {
                cacheIIMA = new IIMAction2();
        }
        return cacheIIMA;
}

/**
 * Create concrete instance of TooManyAttemptMsgAction
 */
TooManyAttemptMsgAction *ConcreteFactory1::createTMAMA() {
        if (!cacheTMAMA) {
                cacheTMAMA = new TMAMAction1();
        }
        return cacheTMAMA;
}

TooManyAttemptMsgAction *ConcreteFactory2::createTMAMA() {
        if (!cacheTMAMA) {
                cacheTMAMA = new TMAMAction2();
        }
        return cacheTMAMA;
}

/**
 * Create concrete instance of PromptPinAction
 */
PromptPinAction *ConcreteFactory1::createPPA() {
        if (!cachePPA) {
                cachePPA = new PPAction1();
        }
        return cachePPA;
}

PromptPinActio[illegible]A() {
```

```
        if (!cachePPA) {
                cachePPA = new PPAction2();
        }
        return cachePPA;
}
/**
 * Create concrete instance of DisplayMenuAction
 */
DisplayMenuAction *ConcreteFactory1::createDMA() {
        if (!cacheDMA) {
                cacheDMA = new DMAction1();
        }
        return cacheDMA;
}

DisplayMenuAction *ConcreteFactory2::createDMA() {
        if (!cacheDMA) {
                cacheDMA = new DMAction2();
        }
        return cacheDMA;
}

/**
 * Create concrete instance of DoDepositAction
 */
DoDepositAction *ConcreteFactory1::createDDA() {
        if (!cacheDDA) {
                cacheDDA = new DDAction1();
        }
        return cacheDDA;
}

DoDepositAction *ConcreteFactory2::createDDA() {
        if (!cacheDDA) {
                cacheDDA = new DDAction2();
        }
        return cacheDDA;
}
```

9. Source Code: AbstractFactory.cpp, part 3

```cpp
/**
 * Create concrete instance of NoFundMsgAction
 */
NoFundMsgAction *ConcreteFactory1::createNFMA() {
        if (!cacheNFMA) {
                cacheNFMA = new NFMAction1();
        }
        return cacheNFMA;
}

NoFundMsgAction *ConcreteFactory2::createNFMA() {
        if (!cacheNFMA) {
                cacheNFMA = new NFMAction2();
        }
        return cacheNFMA;
}

/**
 * Create concrete instance of DisplayBalanceAction
 */
DisplayBalanceAction *ConcreteFactory1::createDBA() {
        if (!cacheDBA) {
                cacheDBA = new DBAction1();
        }
        return cacheDBA;
}

DisplayBalanceAction *ConcreteFactory2::createDBA() {
        if (!cacheDBA) {
                cacheDBA = new DBAction2();
        }
        return cacheDBA;
}

/**
 * Create concrete instance of DoWithdrawAction
 */
DoWithdrawAction *ConcreteFactory1::createDWA() {
        if (!cacheDWA) {
                cacheDWA = new DWAction1();
        }
        return cacheDWA;
}

DoWithdrawAction *ConcreteFactory2::createDWA() {
        if (!cacheDWA) {
                cacheDWA = new DWAction2();
        }
        return cacheDWA;
}

/**
 * Create concrete instance of BelowMinMsgAction
 */
BelowMinMsgAction *ConcreteFactory1::createBMMA() {
        if (!cacheBMMA) {
                cacheBMMA = new BMMAction1();
        }
        return cacheBMMA;
}

BelowMinMsgAction *ConcreteFactory2::createBMMA() {
        if (!cacheBMMA) {
                cacheBMMA = new BMMAction2();
        }
        return cacheBMMA;
}

/**
 * Create concrete instance of DeductPenaltyAction
 */
DeductPenaltyAction *ConcreteFactory1::createDPA() {
        if (!cacheDPA) {
                cacheDPA = new DPAction1();
        }
        return cacheDPA;
}

DeductPenaltyAction *ConcreteFactory2::createDPA() {
        if (!cacheDPA) {
                cacheDPA = new DPAction2();
        }
        return cacheDPA;
}
```

9. Source Code: Actions.hpp, part 1

```
/**
 * Declaration of strategies and context class
 * in Strategy Pattern.
 * Each strategy class corresponds to a meta action.
 * 2 concrete strategies are subclasses from each strategy class
 * to meet the needs of 2 account logic.
 */

#ifndef _ACTIONS_HPP
#define _ACTIONS_HPP

#include "DataStore.hpp"

class AbstractFactory;

/**
 * Store card's PIN, ID and balance information
 */
class StoreDataAction {
        public:
                StoreDataAction() {};
                virtual ~StoreDataAction() {};
                virtual void storeData(DataStore *ds) = 0;
};

class SCDAction1: public StoreDataAction {
        public:
                virtual void storeData(DataStore *ds);
};

class SCDAction2: public StoreDataAction {
        public:
                virtual void storeData(DataStore *ds);
};
```

```
/**
 * Emit incorrect PIN number message
 */
class IncorrectPinMsgAction {
        public:
                IncorrectPinMsgAction() {};
                virtual ~IncorrectPinMsgAction() {};
                virtual void incorrectPinMsg() = 0;
};

class IPMAction1: public IncorrectPinMsgAction {
        public:
                virtual void incorrectPinMsg();
};

class IPMAction2: public IncorrectPinMsgAction {
        public:
                virtual void incorrectPinMsg();
};

/**
 * Emit incorrect ID message
 */
class IncorrectIdMsgAction {
        public:
                IncorrectIdMsgAction() {};
                virtual ~IncorrectIdMsgAction() {};
                virtual void incorrectIdMsg() = 0;
};

class IIMAction1: public IncorrectIdMsgAction {
        public:
                virtual void incorrectIdMsg();
};

class IIMAction2: public IncorrectIdMsgAction {
        public:
                virtual void incorrectIdMsg();
};
```

9. Source Code: Actions.hpp, part 2

```
/**
 * Emit too many attempts message after input too many incorrect PIN
 */
class TooManyAttemptMsgAction {
        public:
                TooManyAttemptMsgAction() {};
                virtual ~TooManyAttemptMsgAction() {};
                virtual void tooManyAttemptMsg() = 0;
};

class TMAMAction1: public TooManyAttemptMsgAction {
        public:
                virtual void tooManyAttemptMsg();
};

class TMAMAction2: public TooManyAttemptMsgAction {
        public:
                virtual void tooManyAttemptMsg();
};

/**
 * Prompt for PIN
 */
class PromptPinAction {
        public:
                PromptPinAction() {};
                virtual ~PromptPinAction() {};
                virtual void promptPin() = 0;
};

class PPAction1: public PromptPinAction {
        public:
                virtual void promptPin();
};

class PPAction2: public PromptPinAction {
        public:
                virtual void promptPin();
};
```

```
/**
 * Display menu after input correct PIN
 */
class DisplayMenuAction {
        public:
                DisplayMenuAction() {};
                virtual ~DisplayMenuAction() {};
                virtual void displayMenu() = 0;
};

class DMAction1: public DisplayMenuAction {
        public:
                virtual void displayMenu();
};

class DMAction2: public DisplayMenuAction {
        public:
                virtual void displayMenu();
};
/**
 * Make deposit
 */
class DoDepositAction {
        public:
                DoDepositAction() {};
                virtual ~DoDepositAction() {};
                virtual void doDeposit(DataStore *ds) = 0;
};

class DDAction1: public DoDepositAction {
        public:
                virtual void doDeposit(DataStore *ds);
};

class DDAction2: public DoDepositAction {
        public:
                virtual void doDeposit(DataStore *ds);
};
```

9. Source Code: Actions.hpp, part 3

```
/**
 * Show no fund message when withdraw under minimum balance
 */
class NoFundMsgAction {
        public:
                NoFundMsgAction () {};
                virtual ~NoFundMsgAction() {};
                virtual void noFundMsg() = 0;
};

class NFMAction1: public NoFundMsgAction {
        public:
                virtual void noFundMsg();
};

class NFMAction2: public NoFundMsgAction {
        public:
                virtual void noFundMsg();
};

/**
 * Show current balance
 */
class DisplayBalanceAction {
        public:
                DisplayBalanceAction() {};
                virtual ~DisplayBalanceAction() {};
                virtual void displayBalance(DataStore *ds) = 0;
};

class DBAction1: public DisplayBalanceAction {
        public:
                virtual void displayBalance(DataStore *ds);
};

class DBAction2: public DisplayBalanceAction {
        public:
                virtual void displayBalance(DataStore *ds);
};
```

```
/**
 * Make withdraw
 */
class DoWithdrawAction {
        public:
                DoWithdrawAction () {};
                virtual ~DoWithdrawAction() {};
                virtual void doWithdraw(DataStore *ds) = 0;
};

class DWAction1: public DoWithdrawAction {
        public:
                virtual void doWithdraw(DataStore *ds);
};

class DWAction2: public DoWithdrawAction {
        public:
                virtual void doWithdraw(DataStore *ds);
};

/**
 * Display current balance is below minimum balance
 */
class BelowMinMsgAction {
        public:
                BelowMinMsgAction () {};
                virtual ~BelowMinMsgAction() {};
                virtual void belowMinMsg() = 0;
};

class BMMAction1: public BelowMinMsgAction {
        public:
                virtual void belowMinMsg();
};

class BMMAction2: public BelowMinMsgAction {
        public:
                virtual void belowMinMsg();
};
```

9. Source Code: Actions.hpp, part 4

```
/**
 * Deduct penalty when overdrawn
 */
class DeductPenaltyAction {
        protected:
                const float penalty;
        public:
                DeductPenaltyAction(float p = 0.0f): penalty(p) {};
                virtual ~DeductPenaltyAction() {};
                virtual void payPenalty(DataStore *ds) = 0;
};

class DPAction1: public DeductPenaltyAction {
        public:
                DPAction1(float p = 20.0f): DeductPenaltyAction(p) {};
                virtual void payPenalty(DataStore *ds);
};

class DPAction2: public DeductPenaltyAction {
        public:
                virtual void payPenalty(DataStore *ds);
};
```

```
/**
 * Context of different strategies/actions
 */
class OutputProcessor {
        private:
                DataStore *ds;
                StoreDataAction *sda;
                IncorrectPinMsgAction *ipma;
                IncorrectIdMsgAction *iima;
                TooManyAttemptMsgAction *tmama;
                PromptPinAction *ppa;
                DisplayMenuAction *dma;
                DoDepositAction *dda;
                NoFundMsgAction *nfma;
                DisplayBalanceAction *dba;
                DoWithdrawAction *dwa;
                BelowMinMsgAction *bmma;
                DeductPenaltyAction *dpa;
                AbstractFactory *af;
        public:
                OutputProcessor(AbstractFactory *a): af(a) {};
                virtual ~OutputProcessor();

                void init();
                void storeData();
                void incorrectPinMsg();
                void incorrectIdMsg();
                void tooManyAttemptMsg();
                void promptPin();
                void displayMenu();
                void doDeposit();
                void noFundMsg();
                void displayBalance();
                void doWithdraw();
                void belowMinMsg();
                void payPenalty();
};

#endif
```

9. Source Code: Actions.cpp, part 1

```cpp
#include <iostream>
#include "Actions.hpp"
#include "AbstractFactory.hpp"

using namespace std;

/**
 * Typically, actions for Account1 will use DataStore1 instance,
 * which is casted from ds
 */
void SCDAction1::storeData(DataStore* ds) {
        DataStore1 *ds1 = (DataStore1 *)ds;
        ds1->setPin(ds1->getTempPin());
        ds1->setId(ds1->getTempId());
        ds1->setBalance(ds1->getTempBalance());
}


/**
 * Similarly, actions for Account2 will use DataStore1 instance,
 * which is casted from ds
 */
void SCDAction2::storeData(DataStore* ds) {
        DataStore2 *ds2 = (DataStore2 *)ds;
        ds2->setPin(ds2->getTempPin());
        ds2->setId(ds2->getTempId());
        ds2->setBalance(ds2->getTempBalance());
}



/**
 * Incorrect PIN message
 */
void IPMAction1::incorrectPinMsg() {
        cout<<"\tIncorrect Pin Input at Account 1\n";
}

void IPMAction2::incorrectPinMsg() {
        cout<<"\tIncorrect Pin Input at Account 2\n";
}
```

```cpp
/**
 * Incorrect ID message
 */
void IIMAction1::incorrectIdMsg() {
        cout<<"\tIncorrect ID Input at Account 1\n";
}

void IIMAction2::incorrectIdMsg() {
        cout<<"\tIncorrect ID Input at Account 2\n";
}

/**
 * Too many attempts message
 */
void TMAMAction1::tooManyAttemptMsg() {
        cout<<"\t#Attempts Exceed Maximum Allowed at Account1\n";
}

void TMAMAction2::tooManyAttemptMsg() {
        cout<<"\t#Attempts Exceed Maximum Allowed at Account2\n";
}

/**
 * Prompt for PIN message
 */
void PPAction1::promptPin() {
        cout<<"\tPlease Input PIN to Proceed at Account1\n";
}

void PPAction2::promptPin() {
        cout<<"\tPlease Input PIN to Proceed at Account2\n";
}

/**
 * Deduct penalty message
 */
void DPAction1::payPenalty(DataStore *ds) {
        DataStore1 *ds1 = (DataStore1 *)ds;
        ds1->setBalance(ds1->getBalance() - penalty);
}

void DPAction2::payPenalty(DataStore *ds) {
        DataStore2 *ds2 = (DataStore2 *)ds;
        ds2->setBalance(ds2->getBalance() - penalty);
}
```

9. Source Code: Actions.cpp, part 2

```
/**
 * Display menu on ATM
 */
void DMAction1::displayMenu() {
        cout<<"\tMenu at Account1\n";
        cout<<"\t\t1. Display Balance\n";
        cout<<"\t\t2. Make Deposit\n";
        cout<<"\t\t3. Withdraw\n";
        cout<<"\t\t4. Lock Account\n";
        cout<<"\t\t5. Unlock Account\n";
        cout<<"\t\t6. Logout\n";
}

void DMAction2::displayMenu() {
        cout<<"\tMenu at Account2\n";
        cout<<"\t\t1. Display Balance\n";
        cout<<"\t\t2. Make Deposit\n";
        cout<<"\t\t3. Withdraw\n";
        cout<<"\t\t4. Suspend Account\n";
        cout<<"\t\t5. Activate Account\n";
        cout<<"\t\t6. Logout\n";
        cout<<"\t\t7. Close\n";
}

/**
 * Make deposit to account
 */
void DDAction1::doDeposit(DataStore *ds) {
        DataStore1 *ds1 = (DataStore1 *)ds;
        ds1->setBalance(ds1->getBalance() + ds1->getTempD());
}

void DDAction2::doDeposit(DataStore *ds) {
        DataStore2 *ds2 = (DataStore2 *)ds;
        ds2->setBalance(ds2->getBalance() + ds2->getTempD());
}


/**
 * No fund message
 */
void NFMAction1::noFundMsg() {
        cout<<"\tZero Balance at Account1\n";
}
```

```
void NFMAction2::noFundMsg() {
        cout<<"\tZero Balance at Account2\n";
}

/**
 * Display current balance
 */
void DBAction1::displayBalance(DataStore *ds) {
        DataStore1 *ds1 = (DataStore1 *)ds;
        cout<<"\tCurrent Balance = $"<<ds1->getBalance()<<" at Account1\n";
}

void DBAction2::displayBalance(DataStore *ds) {
        DataStore2 *ds2 = (DataStore2 *)ds;
        cout<<"\tCurrent Balance = $"<<ds2->getBalance()<<" at Account2\n";
}


/**
 * Make withdraw
 */
void DWAction1::doWithdraw(DataStore *ds) {
        DataStore1 *ds1 = (DataStore1 *)ds;
        ds1->setBalance(ds1->getBalance() - ds1->getTempW());
}

void DWAction2::doWithdraw(DataStore *ds) {
        DataStore2 *ds2 = (DataStore2 *)ds;
        ds2->setBalance(ds2->getBalance() - ds2->getTempW());
}


/**
 * Below minimum balance message
 */
void BMMAction1::belowMinMsg() {
        cout<<"\tBalance Below Minimum Allowed at Account1\n";
}

void BMMAction2::belowMinMsg() {
        cout<<"\tBalance Below Minimum Allowed at Account2\n";
}
```

9. Source Code: Actions.cpp, part 3

```cpp
/**
 * Configure all strategies with Abstract Factory Pattern
 */
void OutputProcessor::init() {
        ds = af->createDS();
        sda = af->createSDA();
        ipma = af->createIPMA();
        iima = af->createIIMA();
        tmama = af->createTMAMA();
        ppa = af->createPPA();
        dma = af->createDMA();
        dda = af->createDDA();
        nfma = af->createNFMA();
        dba = af->createDBA();
        dwa = af->createDWA();
        bmma = af->createBMMA();
        dpa = af->createDPA();
}

/**
 * Reclaim strategy objects
 */
OutputProcessor::~OutputProcessor() {
        delete sda;
        delete ipma;
        delete iima;
        delete tmama;
        delete ppa;
        delete dma;
        delete dda;
        delete nfma;
        delete dba;
        delete dwa;
        delete bmma;
        delete dpa;
};

void OutputProcessor::storeData() {
        sda->storeData(ds);
}


/**
 * As context class in Strategy Pattern,
 * OutputProcessor forwards actions to corresponding strategies.
 */
void OutputProcessor::incorrectPinMsg() {
        ipma->incorrectPinMsg();
}

void OutputProcessor::incorrectIdMsg() {
        iima->incorrectIdMsg();
}

void OutputProcessor::tooManyAttemptMsg() {
        tmama->tooManyAttemptMsg();
}

void OutputProcessor::promptPin() {
        ppa->promptPin();
}

void OutputProcessor::displayMenu() {
        dma->displayMenu();
}

void OutputProcessor::doDeposit() {
        dda->doDeposit(ds);
}

void OutputProcessor::noFundMsg() {
        nfma->noFundMsg();
}

void OutputProcessor::displayBalance() {
        dba->displayBalance(ds);
}

void OutputProcessor::doWithdraw() {
        dwa->doWithdraw(ds);
}

void OutputProcessor::belowMinMsg() {
        bmma->belowMinMsg();
}

void OutputProcessor::payPenalty() {
        dpa->payPenalty(ds);
}
```

9. Source Code: DataStore.hpp, part 1

```
/**
 * Declaration of DataStore.
 */

#ifndef _DATASTORE_HPP
#define _DATASTORE_HPP

#include <string>

using namespace std;


/**
 * Act as an unified stub/interface used by
 * Account1/2 and OutputProcessor
 */
class DataStore {
        public:
                DataStore() {};
                virtual ~DataStore() {};
};


/**
 * Data used in Account1. Notice the type of various
 * member variables.
 */
class DataStore1: public DataStore {
        private:
                /**
                 * Account1 information: PIN number,
                 * ID and balance amount
                 */
                string pin;
                string id;
                float balance;
                /**
                 * Temporal storage for events' parameters
                 */
                float temp_d;
                float temp_w;
                string temp_pin;
                string temp_id;
                float temp_balance;
```

```
public:
        DataStore1() {};
        virtual ~DataStore1() {};

        /**
         * Setters for all fields
         */
        void setPin(string p) {
                pin = p;
        }
        void setId(string i) {
                id = i;
        }
        void setBalance(float b) {
                balance = b;
        }
        void setTempD(float d) {
                temp_d = d;
        }
        void setTempW(float w) {
                temp_w = w;
        }
        void setTempPin(string p) {
                temp_pin = p;
        }
        void setTempId(string i) {
                temp_id = i;
        }
        void setTempBalance(float b) {
                temp_balance = b;

        }
        /**
         * Getters for all fields
         */
        string getPin() {
                return pin;
        }
        string getId() {
                return id;
        }
```

9. Source Code: DataStore.hpp, part 2

```
                float getBalance() {
                        return balance;
                }
                float getTempD() {
                        return temp_d;
                }
                float getTempW() {
                        return temp_w;
                }
                string getTempPin() {
                        return temp_pin;
                }
                string getTempId() {
                        return temp_id;
                }
                float getTempBalance() {
                        return temp_balance;
                }
};

/**
 * Data used in Account2
 */
class DataStore2: public DataStore {
        private:
                /**
                 * Account1 information: PIN number,
                 * ID and balance amount
                 */
                int pin;
                int id;
                int balance;
                /**
                 * Temporal storage for events' parameters
                 */
                int temp_d;
                int temp_w;
                int temp_pin;
                int temp_id;
                int temp_balance;
        public:
                DataStore2() {};
                virtual ~DataStore2() {};
```

```
 * Setters for all fields
 */
virtual void setPin(int p) {
        pin = p;
};
virtual void setId(int i) {
        id = i;
};
virtual void setBalance(int b) {
        balance = b;
};
virtual void setTempD(int d) {
        temp_d = d;
};
virtual void setTempW(int w) {
        temp_w = w;
};
virtual void setTempPin(int p) {
        temp_pin = p;
};
virtual void setTempId(int i) {
        temp_id = i;
};
virtual void setTempBalance(int b) {
        temp_balance = b;
};

/**
 * Getters for all fields
 */
virtual int getPin() {
        return pin;
};
virtual int getId() {
        return id;
};
virtual int getBalance() {
        return balance;
};
virtual int getTempD() {
        return temp_d;
};
virtual int getTempW() {
        return temp_w;
};
```

```
        virtual int getTempPin() {
                return temp_pin;
        };
        virtual int getTempId() {
                return temp_id;
        };
        virtual int getTempBalance() {
                return temp_balance;
        };
};

#endif
```

9. Source Code: ModelDriveArch.hpp, part 1

```cpp
/**
 * Declaration of MDA class and possible states
 * in MDA-EFSM, an implementation of decentralized
 * State Pattern.
 */

#ifndef _MODELDRIVENARCH_HPP
#define _MODELDRIVENARCH_HPP

#include <vector>
#include "Actions.hpp"

using namespace std;

/**
 * Enumeration of possible states, used as
 * State ID in changeState() operation
 */
typedef enum {
        START = 0,
        IDLE,
        CHECKPIN,
        READY,
        OVERDRAWN,
        LOCKED,
        SUSPENDED,
        CLOSED,
        TEMP
} StateEnum;

class ModelDrivenArch;


/**
 * Decentralized State Pattern
 */
class State {
        protected:
                ModelDrivenArch *context;
                OutputProcessor *op;
        public:
                State(ModelDrivenArch *ctxt,
                                OutputProcessor *o): context(ctxt), op(o) {};
                virtual ~State() {};
```

```cpp
                /**
                 * Meta events of MDA's EFSM
                 */
                virtual void open() {};
                virtual void login() {};
                virtual void loginFail() {};
                virtual void logout() {};
                virtual void incorrectPin(int max) {};
                virtual void correctPin() {};
                virtual void aboveMin() {};
                virtual void belowMin() {};
                virtual void balance() {};
                virtual void withdraw() {};
                virtual void withdrawFail() {};
                virtual void withdrawBelowMin() {};
                virtual void deposit() {};
                virtual void lock() {};
                virtual void lockFail() {};
                virtual void unlock() {};
                virtual void unlockFail() {};
                virtual void suspend() {};
                virtual void activate() {};
                virtual void close() {};
};


/**
 * Start state
 */
class StartState: public State {
        public:
                StartState(ModelDrivenArch *ctxt,
                                OutputProcessor *o): State(ctxt, o) {};
                virtual ~StartState() {};

                virtual void open();
};
```

9. Source Code: ModelDriveArch.hpp, part 2

```
/**
 * Idle state
 */
class IdleState: public State {
        public:
                IdleState(ModelDrivenArch *ctxt,
                                OutputProcessor *o): State(ctxt, o) {};
                virtual ~IdleState() {};

                virtual void login();
                virtual void loginFail();
};

/**
 * CheckPin state
 */
class CheckPinState: public State {
        public:
                CheckPinState(ModelDrivenArch *ctxt,
                                OutputProcessor *o): State(ctxt, o) {};
                virtual ~CheckPinState() {};

                virtual void correctPin();
                virtual void incorrectPin(int max);
                virtual void logout();
};

/**
 * Ready state
 */
class ReadyState: public State {
        public:
                ReadyState(ModelDrivenArch *ctxt,
                                OutputProcessor *o): State(ctxt, o) {};
                virtual ~ReadyState() {};

                virtual void balance();
                virtual void lockFail();
                virtual void lock();
                virtual void suspend();
                virtual void withdrawFail();
                virtual void withdraw();
                virtual void deposit();
                virtual void logout();
};
```

```
/**
 * Overdrawn state
 */
class OverdrawnState: public State {
        public:
                OverdrawnState(ModelDrivenArch *ctxt,
                                OutputProcessor *o): State(ctxt, o) {};
                virtual ~OverdrawnState() {};

                virtual void logout();
                virtual void balance();
                virtual void lockFail();
                virtual void lock();
                virtual void deposit();
                virtual void withdrawFail();
};

/**
 * Locked state
 */
class LockedState: public State {
        public:
                LockedState(ModelDrivenArch *ctxt,
                                OutputProcessor *o): State(ctxt, o) {};
                virtual ~LockedState() {};

                virtual void unlockFail();
                virtual void unlock();
};
```

9. Source Code: ModelDriveArch.hpp, part 3

```cpp
/**
 * Suspended state
 */
class SuspendedState: public State {
        public:
                SuspendedState(ModelDrivenArch *ctxt,
                                OutputProcessor *o): State(ctxt, o) {};
                virtual ~SuspendedState() {};

                virtual void balance();
                virtual void close();
                virtual void activate();
};

/**
 * Closed state
 */
class ClosedState: public State {
        public:
                ClosedState(ModelDrivenArch *ctxt,
                                OutputProcessor *o): State(ctxt, o) {};
                virtual ~ClosedState() {};
};

/**
 * Temp state after deposit, withdraw, unlock etc.
 */
class TempState: public State {
        public:
                TempState(ModelDrivenArch *ctxt,
                                OutputProcessor *o): State(ctxt, o) {};
                virtual ~TempState() {};

                virtual void aboveMin();
                virtual void belowMin();
                virtual void withdrawBelowMin();
};

/**
 * Context of State Pattern
 */
class ModelDrivenArch {
        private:
                vector<State *> states; /* a list of all states */
                State *current; /* current state of the EFSM */
                int attempts; /* number of incorrect PIN attempts */
        public:
                ModelDrivenArch(OutputProcessor *op);
                virtual ~ModelDrivenArch();

                /**
                 * Used by current state to make transition
                 */
                void changeState(StateEnum stateID);
                /**
                 * Setter and getter for attempts
                 */
                void setAttempts(int a);
                int getAttempts();
                /**
                 * Meta events of MDA
                 */
                void open();
                void login();
                void loginFail();
                void logout();
                void incorrectPin(int max);
                void correctPin();
                void aboveMin();
                void belowMin();
                void balance();
                void withdraw();
                void withdrawFail();
                void withdrawBelowMin();
                void deposit();
                void lock();
                void lockFail();
                void unlock();
                void unlockFail();
                void suspend();
                void activate();
                void close();

};

#endif
```

9. Source Code: ModelDriveArch.cpp, part 1

```cpp
#include "ModelDrivenArch.hpp"
/**
 * Concrete states will override necessary methods,
 */

/**
 * Possible events for Start state:
 * open
 */
void StartState::open() {
        context->changeState(IDLE);
        op->storeData();
}

/**
 * Possible events for Idle state:
 * login, loginFail
 */
void IdleState::login() {
        context->changeState(CHECKPIN);
        context->setAttempts(0);
        op->promptPin();
}

void IdleState::loginFail() {
        op->incorrectIdMsg();
}

/**
 * Possible events for CheckPin state
 * correctPin, incorrectPin, logout
 */
void CheckPinState::correctPin() {
        context->changeState(TEMP);
        op->displayMenu();
}

void CheckPinState::incorrectPin(int max) {
        int attempts = context->getAttempts();
        if (attempts >= max) {
                context->changeState(IDLE);
                op->tooManyAttemptMsg();
        } else if (attempts < max) {
                context->setAttempts(++attempts);
                op->incorrectPinMsg();
        }
}

void CheckPinState::logout() {
        context->changeState(IDLE);
}

/**
 * Possible events for Ready state:
 * balance, lockFail, lock, suspend,
 * withdrawFail, withdraw, deposit, logout
 */
void ReadyState::balance() {
        op->displayBalance();
}

void ReadyState::lockFail() {
        op->incorrectPinMsg();
}

void ReadyState::lock() {
        context->changeState(LOCKED);
}

void ReadyState::suspend() {
        context->changeState(SUSPENDED);
}

void ReadyState::withdrawFail() {
        op->noFundMsg();
}

void ReadyState::withdraw() {
        context->changeState(TEMP);
        op->doWithdraw();
}

void ReadyState::deposit() {
        op->doDeposit();
}

void ReadyState::logout() {
        context->changeState(IDLE);
}
```

9. Source Code: ModelDriveArch.cpp, part 2

```
/**
 * Possible events for Overdrawn state:
 * logout, balance, lockFail, lock,
 * deposit, withdrawFail
 */
void OverdrawnState::logout() {
        context->changeState(IDLE);
}

void OverdrawnState::balance() {
        op->displayBalance();
}

void OverdrawnState::lockFail() {
        op->incorrectPinMsg();
}

void OverdrawnState::lock() {
        context->changeState(LOCKED);
}

void OverdrawnState::deposit() {
        context->changeState(TEMP);
        op->doDeposit();
}

void OverdrawnState::withdrawFail() {
        op->belowMinMsg();
}

/**
 * Possible events for Locked state:
 * unlock, unlockFail
 */
void LockedState::unlock() {
        context->changeState(TEMP);
}

void LockedState::unlockFail() {
        op->incorrectPinMsg();
}
```

```
/**
 * Possible events for Suspended state:
 * activate, balance, close
 */
void SuspendedState::activate() {
        context->changeState(READY);
}

void SuspendedState::balance() {
        op->displayBalance();
}

void SuspendedState::close() {
        context->changeState(CLOSED);
}

/**
 * Possible events for Temp state:
 * aboveMin, belowMin, withdrawBelowMin
 */
void TempState::aboveMin() {
        context->changeState(READY);
}

void TempState::belowMin() {
        context->changeState(OVERDRAWN);
}

void TempState::withdrawBelowMin() {
        context->changeState(OVERDRAWN);
        op->payPenalty();
}
```

9. Source Code: ModelDriveArch.cpp, part 3

```cpp
/**
 * Create the list of all possible states in MDA-EFSM
 */
ModelDrivenArch::ModelDrivenArch(OutputProcessor *op) {
        StartState *ss = new StartState(this, op);
        IdleState *is = new IdleState(this, op);
        CheckPinState *cps = new CheckPinState(this, op);
        ReadyState *rs = new ReadyState(this, op);
        OverdrawnState *os = new OverdrawnState(this, op);
        LockedState *ls = new LockedState(this, op);
        SuspendedState *ss2 = new SuspendedState(this, op);
        ClosedState *cs = new ClosedState(this, op);
        TempState *ts = new TempState(this, op);

        states.push_back(ss);
        states.push_back(is);
        states.push_back(cps);
        states.push_back(rs);
        states.push_back(os);
        states.push_back(ls);
        states.push_back(ss2);
        states.push_back(cs);
        states.push_back(ts);

        current = states[0];
}

/**
 * Reclaim allocated State objects
 */
ModelDrivenArch::~ModelDrivenArch() {
        for (int i = 0; i < states.size(); ++i) {
                if (states[i]) {
                        delete states[i];
                }
        }
}

/**
 * Switch based on enumeration value
 */
void ModelDrivenArch::changeState(StateEnum stateID) {
        switch (stateID) {
                case START:
                        current = states[0];
                        break;
                case IDLE:
                        current = states[1];
                        break;
                case CHECKPIN:
                        current = states[2];
                        break;
                case READY:
                        current = states[3];
                        break;
                case OVERDRAWN:
                        current = states[4];
                        break;
                case LOCKED:
                        current = states[5];
                        break;
                case SUSPENDED:
                        current = states[6];
                        break;
                case CLOSED:
                        current = states[7];
                        break;
                case TEMP:
                        current = states[8];
                        break;
                default:
                        break;
        }
}

/**
 * Setter and getter used by State
 */
void ModelDrivenArch::setAttempts(int a) {
        attempts = a;
}
int ModelDrivenArch::getAttempts() {
        return attempts;
}
```

9. Source Code: ModelDriveArch.cpp, part 4

```
/**
 * In decentralized State Pattern,
 * context just forward events to states
 */
void ModelDrivenArch::open() {
        current->open();
}

void ModelDrivenArch::login() {
        current->login();
}

void ModelDrivenArch::loginFail() {
        current->loginFail();
}

void ModelDrivenArch::logout() {
        current->logout();
}

void ModelDrivenArch::incorrectPin(int max) {
        current->incorrectPin(max);
}

void ModelDrivenArch::correctPin() {
        current->correctPin();
}

void ModelDrivenArch::aboveMin() {
        current->aboveMin();
}

void ModelDrivenArch::belowMin() {
        current->belowMin();
}

void ModelDrivenArch::balance() {
        current->balance();
}

void ModelDrivenArch::withdraw() {
        current->withdraw();
}

void ModelDrivenArch::withdrawFail() {
        current->withdrawFail();
}

void ModelDrivenArch::withdrawBelowMin() {
        current->withdrawBelowMin();
}

void ModelDrivenArch::deposit() {
        current->deposit();
}

void ModelDrivenArch::lock() {
        current->lock();
}

void ModelDrivenArch::lockFail() {
        current->lockFail();
}

void ModelDrivenArch::unlock() {
        current->unlock();
}

void ModelDrivenArch::unlockFail() {
        current->unlockFail();
}

void ModelDrivenArch::suspend() {
        current->suspend();
}

void ModelDrivenArch::activate() {
        current->activate();
}

void ModelDrivenArch::close() {
        current->close();
}
```

9. Source Code: main.cpp, part 1

```cpp
#include <iostream>
#include "MDABankAccountConfig.h"
#include "ModelDrivenArch.hpp"
#include "AbstractFactory.hpp"
#include "Accounts.hpp"

using namespace std;

/**
 * Driver for running Account1
 */
void driverAccount1() {
        ConcreteFactory1* cf1 = new ConcreteFactory1();
        OutputProcessor* op1 = new OutputProcessor(cf1);
        ModelDrivenArch* mda = new ModelDrivenArch(op1);
        Account1* a1 = new Account1(mda, cf1);
        op1->init();
        a1->init();

        cout<< "                          ACCOUNT-1" << endl;
        cout<< "                  MENU of Operations" << endl;
        cout<< "          0. open(string, string, float)" << endl;
        cout<< "          1. login(string)" << endl;
        cout<< "          2. pin(string)" << endl;
        cout<< "          3. deposit(float)" << endl;
        cout<< "          4. withdraw(float)" << endl;
        cout<< "          5. balance()" << endl;
        cout<< "          6. logout()" << endl;
        cout<< "          7. lock(string)" << endl;
        cout<< "          8. unlock(string)" << endl;
        cout<< "          q. Quit the demo program" << endl;
        cout<< "          ACCOUNT-1 Execution" << endl;
        char cmd = '\0';
        while (cmd != 'q') {
                cout<< "  Select Operation: "<<endl;
                cout<< "0-open, 1-login, 2-pin, 3-deposit, 4-withdraw, "
                               "5-balance, 6-logout, 7-lock, 8-unlock"<<endl;
                cin>> cmd;
                string p, y, x;
                float a, w, d;
                switch (cmd) {
                        case '0':  // open
                                cout<<"  Operation:  open(string p, string y, float
a)"<<endl;
                                cout<<"  Enter value of the parameter p:"<<endl;
                                cin>>p;
                                cout<<"  Enter value of the parameter y:"<<endl;
                                cin>>y;
                                cout<<"  Enter value of the parameter a:"<<endl;
                                cin>>a;
                                a1->open(p,y,a);
                                break;
                        case '1':  // login
                                cout<<"  Operation:  login(string y)"<<endl;
                                cout<<"  Enter value of the parameter y:"<<endl;
                                cin>>y;
                                a1->login(y);
                                break;
                        case '2':  // pin
                                cout<<"  Operation:  pin(string x)"<<endl;
                                cout<<"  Enter value of the parameter x"<<endl;
                                cin >> x;
                                a1->pin(x);
                                break;
                        case '3':  // deposit
                                cout<<"  Operation:  deposit(float d)"<<endl;
                                cout<<"  Enter value of the parameter d:"<<endl;
                                cin >> d;
                                a1->deposit(d);
                                break;
                        case '4':  // withdraw
                                cout<<"  Operation:  withdraw(float w)"<<endl;
                                cout<<"  Enter value of the parameter w:"<<endl;
                                cin >> w;
                                a1->withdraw(w);
                                break;
                        case '5':  // balance
                                cout<<"  Operation:  balance()"<<endl;
                                a1->balance();
                                break;
                        case '6':  // logout
                                cout<<"  Operation:  logout()"<<endl;
                                a1->logout();
                                break;
```

9. Source Code: main.cpp, part 2

```cpp
                case '7':  // lock
                        cout<<"  Operation:  lock(string x)"<<endl;
                        cout<<"  Enter value of the parameter x:"<<endl;
                        cin>> x;
                        a1->lock(x);
                        break;
                case '8':  // unlock
                        cout<<"  Operation:  unlock(string x)"<<endl;
                        cout<<"  Enter value of the parameter x:"<<endl;
                        cin>> x;
                        a1->unlock(x);
                        break;

            }
        }

        delete a1;
        delete mda;
        delete op1;
        delete cf1;
}
/**
 * Driver for running Account2
 */
void driverAccount2() {
        ConcreteFactory2* cf2 = new ConcreteFactory2();
        OutputProcessor* op2 = new OutputProcessor(cf2);
        ModelDrivenArch* mda = new ModelDrivenArch(op2);
        Account2* a2 = new Account2(mda, cf2);
        op2->init();
        a2->init();

        cout<< "                          ACCOUNT-2" << endl;
        cout<< "                  MENU of Operations" << endl;
        cout<< "          0. OPEN(int,int,int)" << endl;
        cout<< "          1. LOGIN(int)" << endl;
        cout<< "          2. PIN(int)" << endl;
        cout<< "          3. DEPOSIT(int)" << endl;
        cout<< "          4. WITHDRAW(int)" << endl;
        cout<< "          5. BALANCE()" << endl;
        cout<< "          6. LOGOUT()" << endl;
        cout<< "          7. suspend()" << endl;
        cout<< "          8. activate()" << endl;
        cout<< "          9. close()" << endl;
        cout<< "          q. Quit the demo program" << endl;
        cout<< "          ACCOUNT-2 Execution" << endl;
```

```cpp
char cmd = '\0';
while (cmd != 'q') {
        cout<< "  Select Operation: "<<endl;
        cout<< "0-OPEN, 1-LOGIN, 2-PIN, 3-DEPOSIT, 4-WITHDRAW, "
                "5-BALANCE, 6-LOGOUT, 7-suspend, 8-activate, 9-close"<<endl;
        cin>> cmd;
        int p, y, x, a, w, d;
        switch (cmd) {
                case '0':  // OPEN
                        cout<<"  Operation:  OPEN(int p, int y, int a)"<<endl;
                        cout<<"  Enter value of the parameter p:"<<endl;
                        cin>>p;
                        cout<<"  Enter value of the parameter y:"<<endl;
                        cin>>y;
                        cout<<"  Enter value of the parameter a:"<<endl;
                        cin>>a;
                        a2->OPEN(p,y,a);
                        break;
                case '1':  // LOGIN
                        cout<<"  Operation:  LOGIN(int y)"<<endl;
                        cout<<"  Enter value of the parameter y:"<<endl;
                        cin>>y;
                        a2->LOGIN(y);
                        break;
                case '2':  // PIN
                        cout<<"  Operation:  PIN(int x)"<<endl;
                        cout<<"  Enter value of the parameter x"<<endl;
                        cin >> x;
                        a2->PIN(x);
                        break;
                case '3':  // DEPOSIT
                        cout<<"  Operation:  DEPOSIT(int d)"<<endl;
                        cout<<"  Enter value of the parameter d:"<<endl;
                        cin >> d;
                        a2->DEPOSIT(d);
                        break;
                case '4':  // WITHDRAW
                        cout<<"  Operation:  WITHDRAW(int w)"<<endl;
                        cout<<"  Enter value of the parameter w:"<<endl;
                        cin >> w;
                        a2->WITHDRAW(w);
                        break;
```

9. Source Code: main.cpp, part 3

```
                        case '5':  // BALANCE
                                cout<<"  Operation:  BALANCE()"<<endl;
                                a2->BALANCE();
                                break;
                        case '6':  // LOGOUT
                                cout<<"  Operation:  LOGOUT()"<<endl;
                                a2->LOGOUT();
                                break;
                        case '7':  // suspend
                                cout<<"  Operation:  suspend()"<<endl;
                                a2->suspend();
                                break;
                        case '8':  // activate
                                cout<<"  Operation:  activate()"<<endl;
                                a2->activate();
                                break;
                        case '9':  // close
                                cout<<"  Operation:  close()"<<endl;
                                a2->close();
                                break;

                }
        }

        delete a2;
        delete mda;
        delete op2;
        delete cf2;
}
```

```
int main(int argc, char *argv[]) {
        cout<< "BankAccount" << " Version " <<
                MDABankAccount_VERSION_MAJOR<< "." <<
                MDABankAccount_VERSION_MINOR<< endl;

        char cmd = '\0';
        /**
         * Choose which account program to run
         */
        while (cmd != 'q') {
                cout<< "Please choose the type of ACCOUNT" << endl;
                cout<< "1. ACCOUNT-1" << endl;
                cout<< "2. ACCOUNT-2" << endl;
                cout<< "q. Quit the demo program" << endl;
                cin>> cmd;
                switch (cmd) {
                        case '1':
                                driverAccount1();
                                break;
                        case '2':
                                driverAccount2();
                                break;
                }
        }

        return 0;
}
```