CS 586 PROJECT

Spring 2009

Prepared by Oleksandr Shashkov

[redacted]

1. Introduction

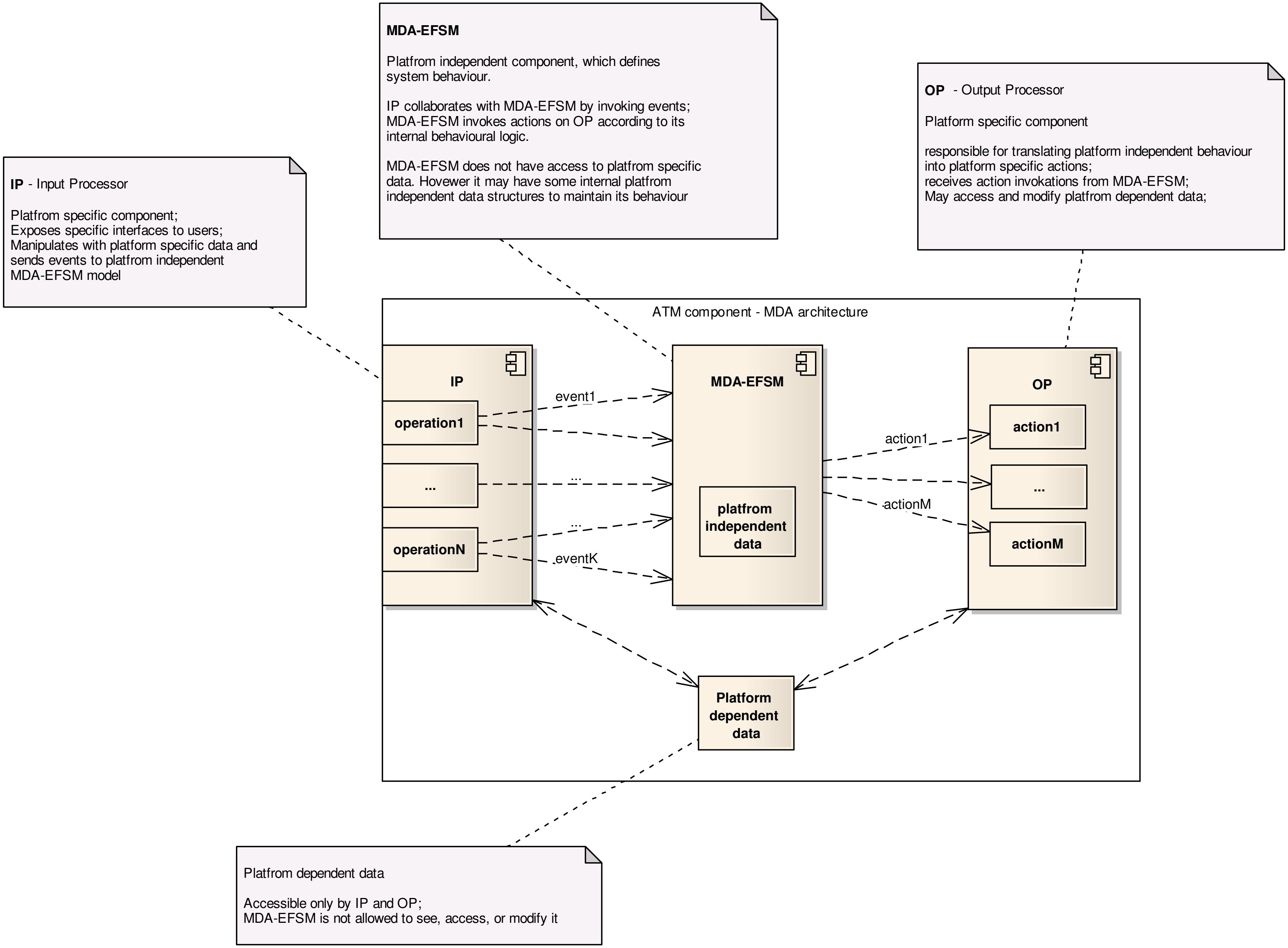
The goal of the project is to design and implement two ATM components using a Model-Driven Architecture (MDA) covered in the course. An executable meta-model, referred to as MDA-EFSM, of ATM components should capture the "generic behavior" of both ATM components and should be decoupled from data and implementation details.

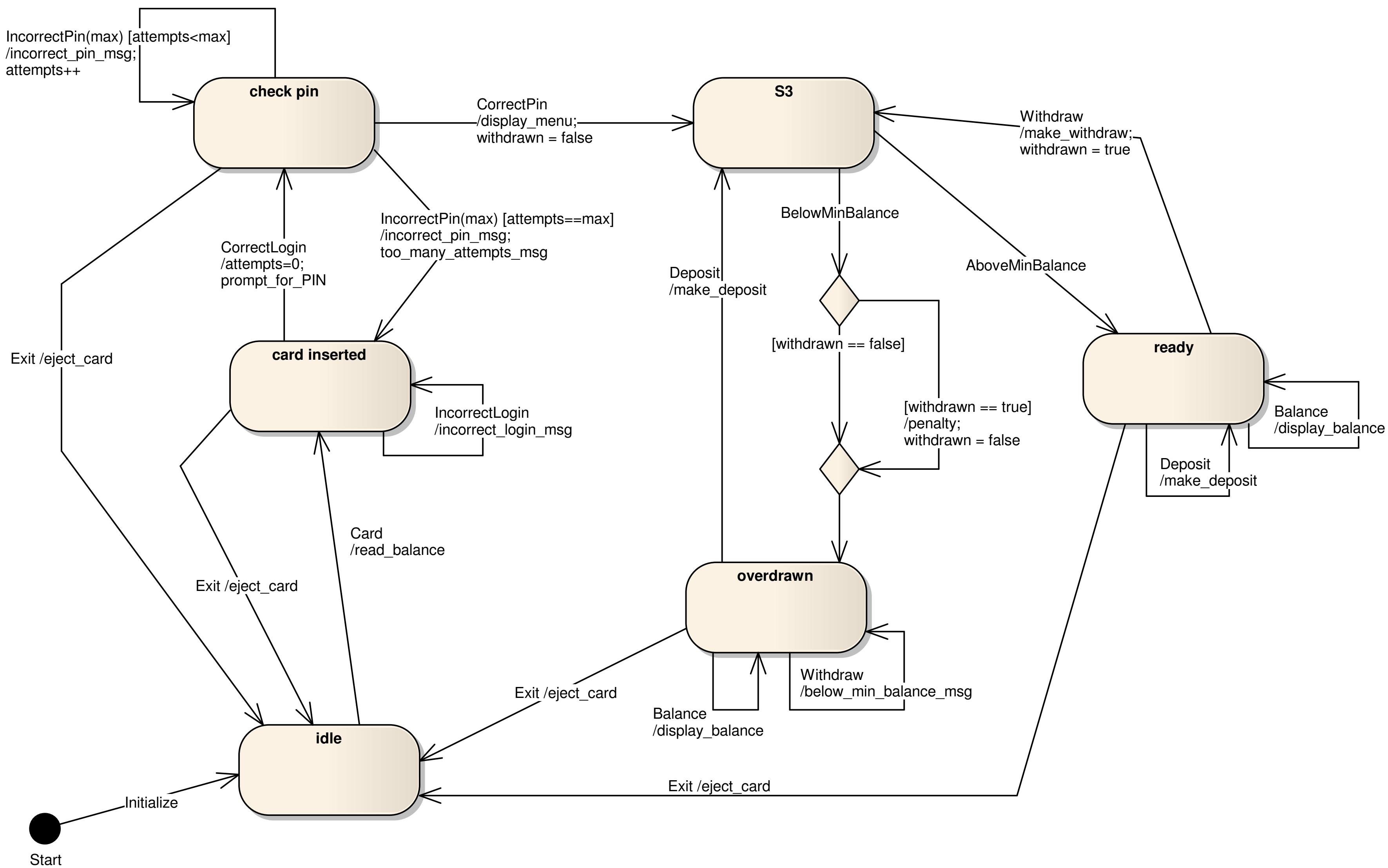
Both ATM components are state based components. The set of operations supported by ATMs as well as their state diagrams are given as initial requirements. These given models are expressed in platform dependent form. Our goal is to separate platform dependent data and behavior from "generic" (platform independent) behavior and realize them in a form of independent components.

2. Model-Driven Architecture

Model Driven Architecture (MDA) is used whenever it becomes necessary to separate platform independent aspects of the system from platform specific ones. Platform independent aspects in such systems are represented by Meta Model or Platform Independent Model (PIM) which captures system behavior that should be stable (not changing often) from platform to platform or from version to version. Thus PIM plays core role in MDA. Platform dependent aspects in MDA are represented by Input Processor (IP), Output Processor (OP) and Platform Dependent Data. In the project the given sets of the operations represent interface of the IPs (ATM1 and ATM2 operations). The specific output actions of the ATMs are represented by OP. General overview of MDA is shown on the next page. The core of the system is MDA EFSM.

The behavior of core meta model is represented by a state diagram for MDA EFSM combined with the lists of events and actions those are independent from specific ATM realization.


MDA-EFSM
Platfrom independent component, which defines system behaviour.
IP collaborates with MDA-EFSM by invoking events; MDA-EFSM invokes actions on OP according to its internal behavioural logic.
MDA-EFSM does not have access to platfrom specific data. Hovewer it may have some internal platfrom independent data structures to maintain its behaviour
OP - Output Processor
Platform specific component
responsible for translating platform independent behaviour into platform specific actions;
receives action invokations from MDA-EFSM;
May access and modify platfrom dependent data;
IP - Input Processor
Platfrom specific component;
Exposes specific interfaces to users;
Manipulates with platform specific data and sends events to platfrom independent MDA-EFSM model
ATM component - MDA architecture
IP
operation1
...
operationN
event1
...
...
eventK
MDA-EFSM
platfrom independent data
action1
actionM
OP
action1
...
actionM
Platform dependent data
Platfrom dependent data
Accessible only by IP and OP;
MDA-EFSM is not allowed to see, access, or modify it


IncorrectPin(max) [attempts<max]
/incorrect_pin_msg;
attempts++
check pin
CorrectPin
/display_menu;
withdrawn = false
S3
Withdraw
/make_withdraw;
withdrawn = true
IncorrectPin(max) [attempts==max]
/incorrect_pin_msg;
too_many_attempts_msg
CorrectLogin
/attempts=0;
prompt_for_PIN
BelowMinBalance
AboveMinBalance
Deposit
/make_deposit
[withdrawn == false]
Exit /eject_card
card inserted
ready
IncorrectLogin
/incorrect_login_msg
[withdrawn == true]
/penalty;
withdrawn = false
Balance
/display_balance
Deposit
/make_deposit
Card
/read_balance
Exit /eject_card
overdrawn
Exit /eject_card
Withdraw
/below_min_balance_msg
Balance
/display_balance
idle
Exit /eject_card
Initialize
Start

List of events for MDA-EFSM

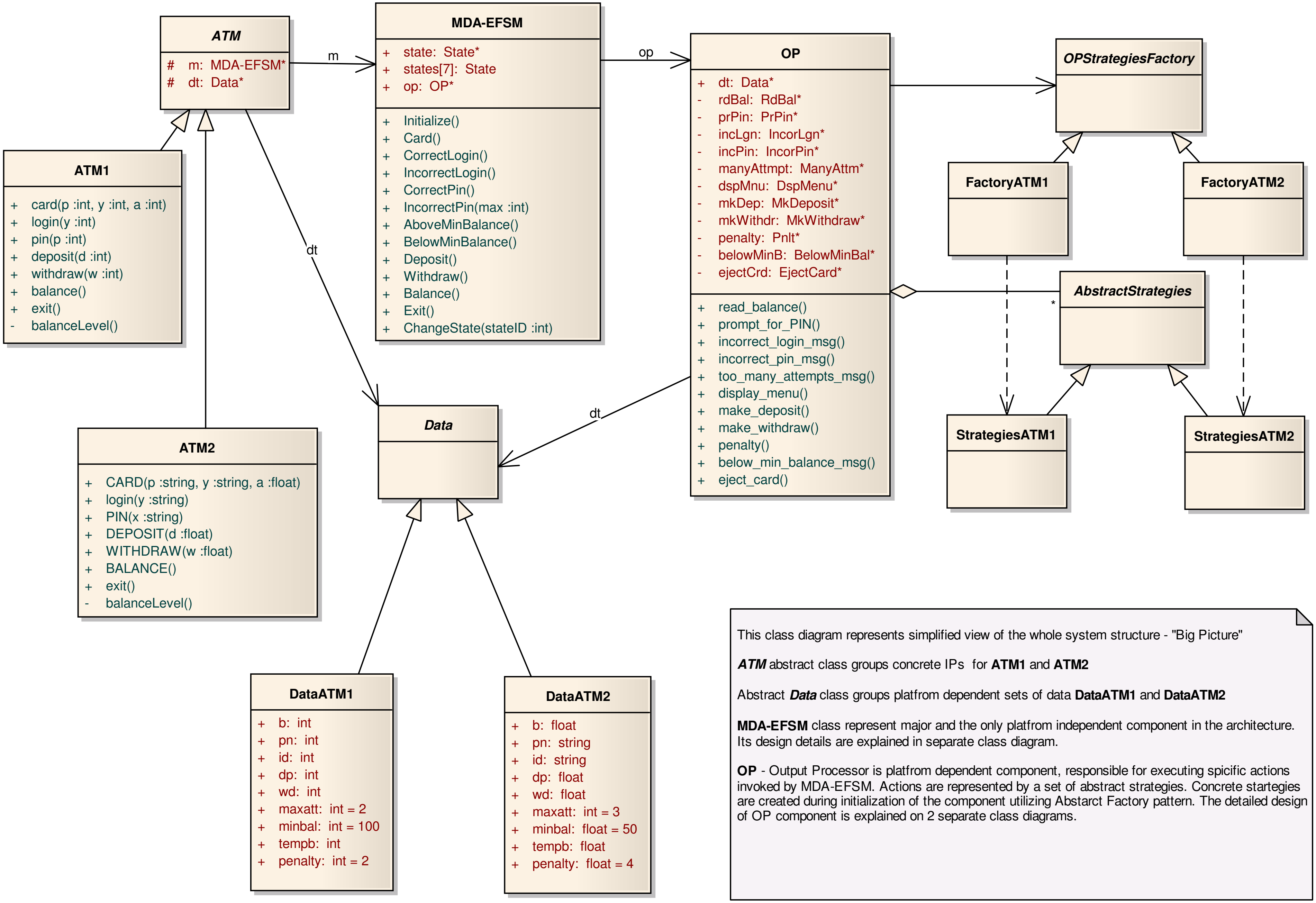
Initialize()
Card()
CorrectLogin()
IncorrectLogin()
Exit()
CorrectPin()
IncorrectPin(int max)
Deposit()
Withdraw()
Balance()
AboveMinBalance()
BelowMinBalance()

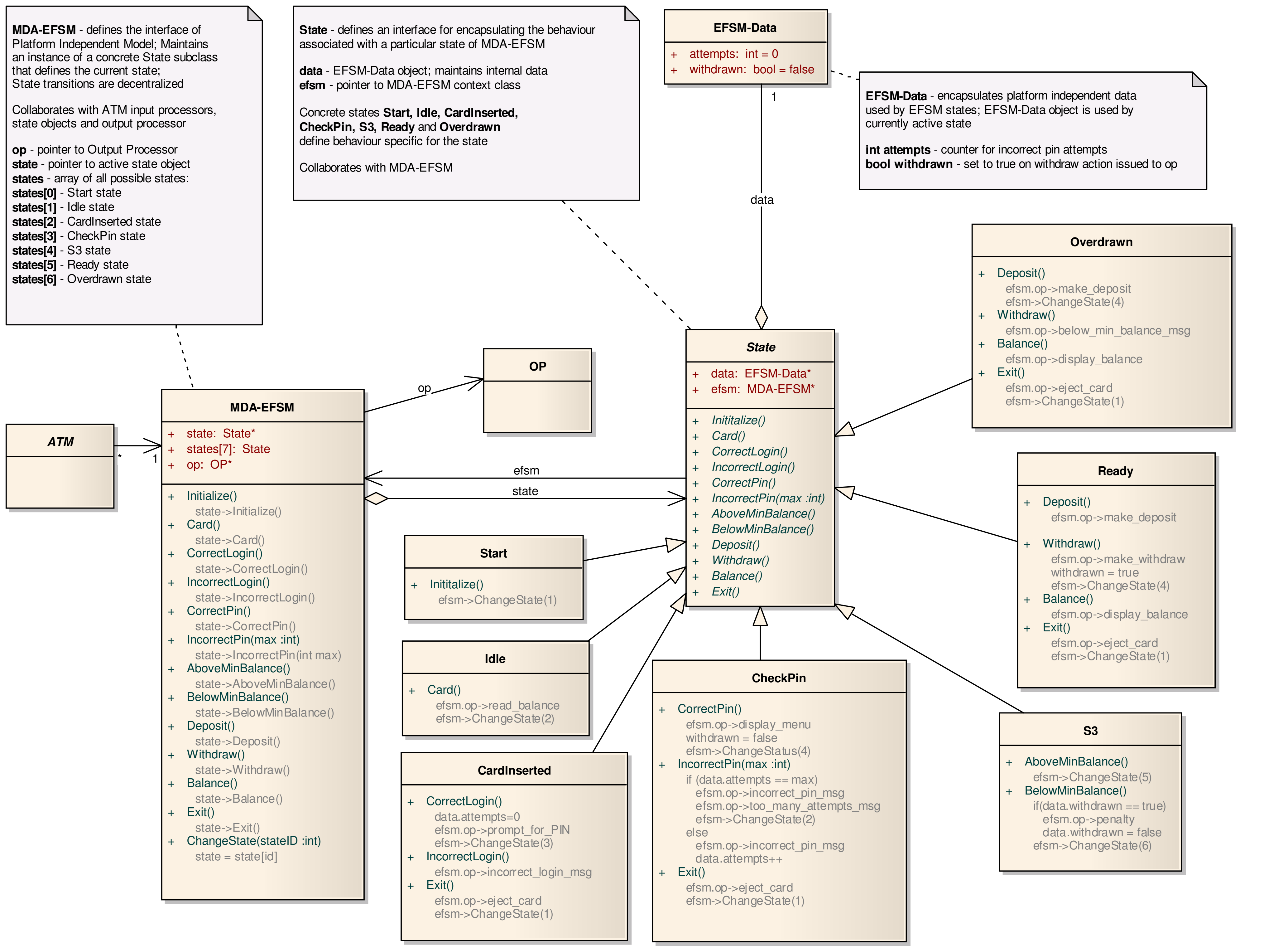
List of actions for the MDA-EFSM

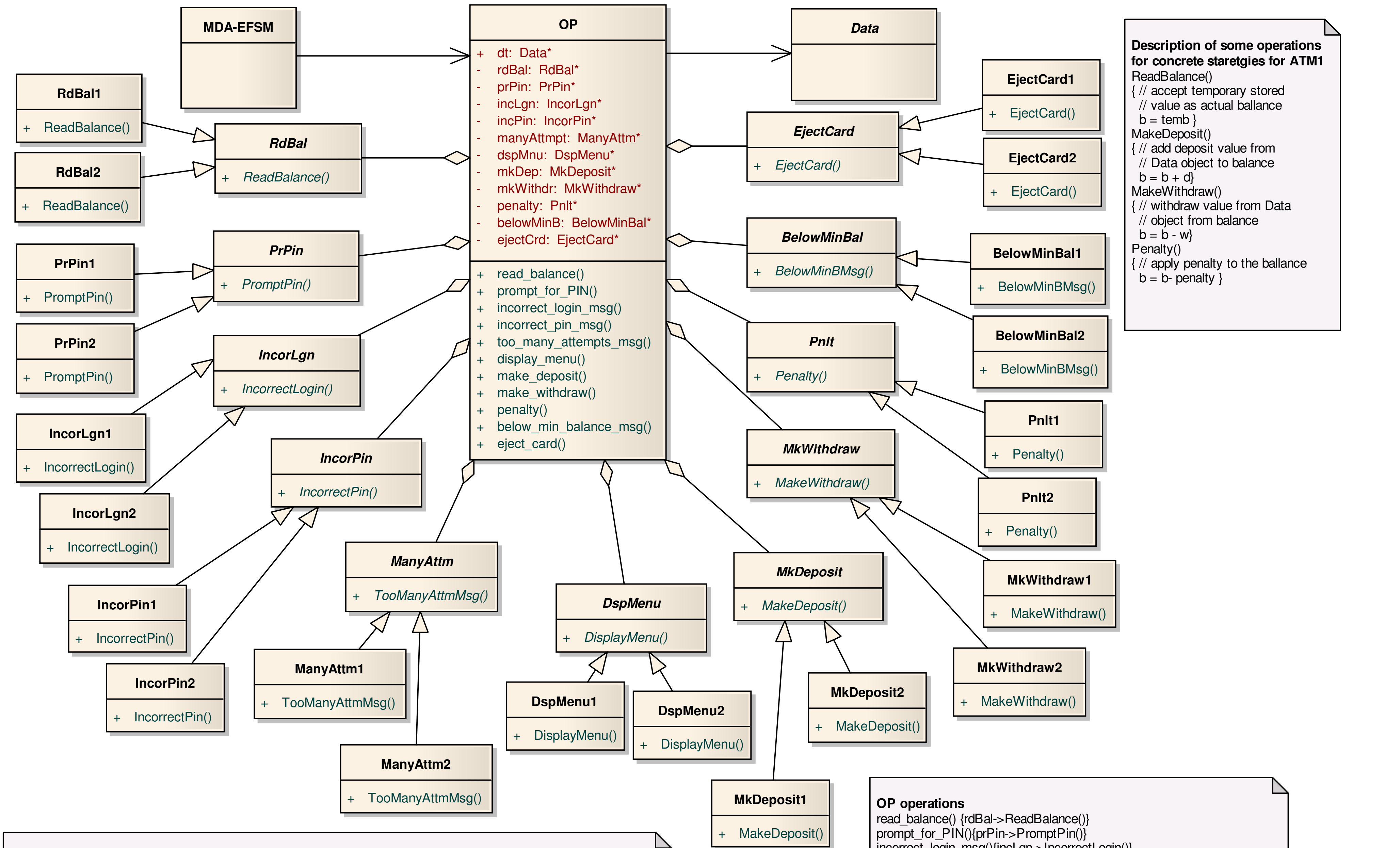
read_balance
promt_for_PIN
incorrect_login_msg
incorrect_pin_msg
too_many_attempts_msg
display_menu
make_deposit
make_withdraw
penalty
below_min_balance_msg
eject_card

3. Static structures. Class diagrams for MDA

The static structure (Class Diagram) of the entire system is shown on the next page. The refined diagrams for the design solutions of the major components are following "Big Picture". It has been decided to utilize Strategy pattern combined with Abstract Factory pattern in order to add flexibility into OP component as well as for entire system. MDA-EFSM component has been translated into State Pattern design solution based on previously shown state diagram. The description of Operations and Attributes for IPs ATM1 and ATM2 are closing this section.


ATM
m: MDA-EFSM*
dt: Data*
m
MDA-EFSM
+ state: State*
+ states[7]: State
+ op: OP*
+ Initialize()
+ Card()
+ CorrectLogin()
+ IncorrectLogin()
+ CorrectPin()
+ IncorrectPin(max :int)
+ AboveMinBalance()
+ BelowMinBalance()
+ Deposit()
+ Withdraw()
+ Balance()
+ Exit()
+ ChangeState(stateID :int)
op
OP
+ dt: Data*
- rdBal: RdBal*
- prPin: PrPin*
- incLgn: IncorLgn*
- incPin: IncorPin*
- manyAttmpt: ManyAttm*
- dspMnu: DspMenu*
- mkDep: MkDeposit*
- mkWithdr: MkWithdraw*
- penalty: Pnlt*
- belowMinB: BelowMinBal*
- ejectCrd: EjectCard*
+ read_balance()
+ prompt_for_PIN()
+ incorrect_login_msg()
+ incorrect_pin_msg()
+ too_many_attempts_msg()
+ display_menu()
+ make_deposit()
+ make_withdraw()
+ penalty()
+ below_min_balance_msg()
+ eject_card()
OPStrategiesFactory
FactoryATM1
FactoryATM2
AbstractStrategies
*
StrategiesATM1
StrategiesATM2
ATM1
+ card(p :int, y :int, a :int)
+ login(y :int)
+ pin(p :int)
+ deposit(d :int)
+ withdraw(w :int)
+ balance()
+ exit()
- balanceLevel()
dt
Data
dt
ATM2
+ CARD(p :string, y :string, a :float)
+ login(y :string)
+ PIN(x :string)
+ DEPOSIT(d :float)
+ WITHDRAW(w :float)
+ BALANCE()
+ exit()
- balanceLevel()
DataATM1
+ b: int
+ pn: int
+ id: int
+ dp: int
+ wd: int
+ maxatt: int = 2
+ minbal: int = 100
+ tempb: int
+ penalty: int = 2
DataATM2
+ b: float
+ pn: string
+ id: string
+ dp: float
+ wd: float
+ maxatt: int = 3
+ minbal: float = 50
+ tempb: float
+ penalty: float = 4
This class diagram represents simplified view of the whole system structure - "Big Picture"
ATM abstract class groups concrete IPs for ATM1 and ATM2
Abstract Data class groups platfrom dependent sets of data DataATM1 and DataATM2
MDA-EFSM class represent major and the only platfrom independent component in the architecture. Its design details are explained in separate class diagram.
OP - Output Processor is platfrom dependent component, responsible for executing spicific actions invoked by MDA-EFSM. Actions are represented by a set of abstract strategies. Concrete startegies are created during initialization of the component utilizing Abstarct Factory pattern. The detailed design of OP component is explained on 2 separate class diagrams.


MDA-EFSM - defines the interface of Platform Independent Model; Maintains an instance of a concrete State subclass that defines the current state; State transitions are decentralized
Collaborates with ATM input processors, state objects and output processor
op - pointer to Output Processor
state - pointer to active state object
states - array of all possible states:
states[0] - Start state
states[1] - Idle state
states[2] - CardInserted state
states[3] - CheckPin state
states[4] - S3 state
states[5] - Ready state
states[6] - Overdrawn state
State - defines an interface for encapsulating the behaviour associated with a particular state of MDA-EFSM
data - EFSM-Data object; maintains internal data
efsm - pointer to MDA-EFSM context class
Concrete states Start, Idle, CardInserted, CheckPin, S3, Ready and Overdrawn define behaviour specific for the state
Collaborates with MDA-EFSM
EFSM-Data
+ attempts: int = 0
+ withdrawn: bool = false
1
data
EFSM-Data - encapsulates platform independent data used by EFSM states; EFSM-Data object is used by currently active state
int attempts - counter for incorrect pin attempts
bool withdrawn - set to true on withdraw action issued to op
Overdrawn
+ Deposit()
efsm.op->make_deposit
efsm->ChangeState(4)
+ Withdraw()
efsm.op->below_min_balance_msg
+ Balance()
efsm.op->display_balance
+ Exit()
efsm.op->eject_card
efsm->ChangeState(1)
OP
op
State
+ data: EFSM-Data*
+ efsm: MDA-EFSM*
+ Inititalize()
+ Card()
+ CorrectLogin()
+ IncorrectLogin()
+ CorrectPin()
+ IncorrectPin(max :int)
+ AboveMinBalance()
+ BelowMinBalance()
+ Deposit()
+ Withdraw()
+ Balance()
+ Exit()
ATM
*
1
MDA-EFSM
+ state: State*
+ states[7]: State
+ op: OP*
+ Initialize()
state->Initialize()
+ Card()
state->Card()
+ CorrectLogin()
state->CorrectLogin()
+ IncorrectLogin()
state->IncorrectLogin()
+ CorrectPin()
state->CorrectPin()
+ IncorrectPin(max :int)
state->IncorrectPin(int max)
+ AboveMinBalance()
state->AboveMinBalance()
+ BelowMinBalance()
state->BelowMinBalance()
+ Deposit()
state->Deposit()
+ Withdraw()
state->Withdraw()
+ Balance()
state->Balance()
+ Exit()
state->Exit()
+ ChangeState(stateID :int)
state = state[id]
efsm
state
Ready
+ Deposit()
efsm.op->make_deposit
+ Withdraw()
efsm.op->make_withdraw
withdrawn = true
efsm->ChangeState(4)
+ Balance()
efsm.op->display_balance
+ Exit()
efsm.op->eject_card
efsm->ChangeState(1)
Start
+ Inititalize()
efsm->ChangeState(1)
Idle
+ Card()
efsm.op->read_balance
efsm->ChangeState(2)
CheckPin
+ CorrectPin()
efsm.op->display_menu
withdrawn = false
efsm->ChangeStatus(4)
+ IncorrectPin(max :int)
if (data.attempts == max)
efsm.op->incorrect_pin_msg
efsm.op->too_many_attempts_msg
efsm->ChangeState(2)
else
efsm.op->incorrect_pin_msg
data.attempts++
+ Exit()
efsm.op->eject_card
efsm->ChangeState(1)
S3
+ AboveMinBalance()
efsm->ChangeState(5)
+ BelowMinBalance()
if(data.withdrawn == true)
efsm.op->penalty
data.withdrawn = false
efsm->ChangeState(6)
CardInserted
+ CorrectLogin()
data.attempts=0
efsm.op->prompt_for_PIN
efsm->ChangeState(3)
+ IncorrectLogin()
efsm.op->incorrect_login_msg
+ Exit()
efsm.op->eject_card
efsm->ChangeState(1)

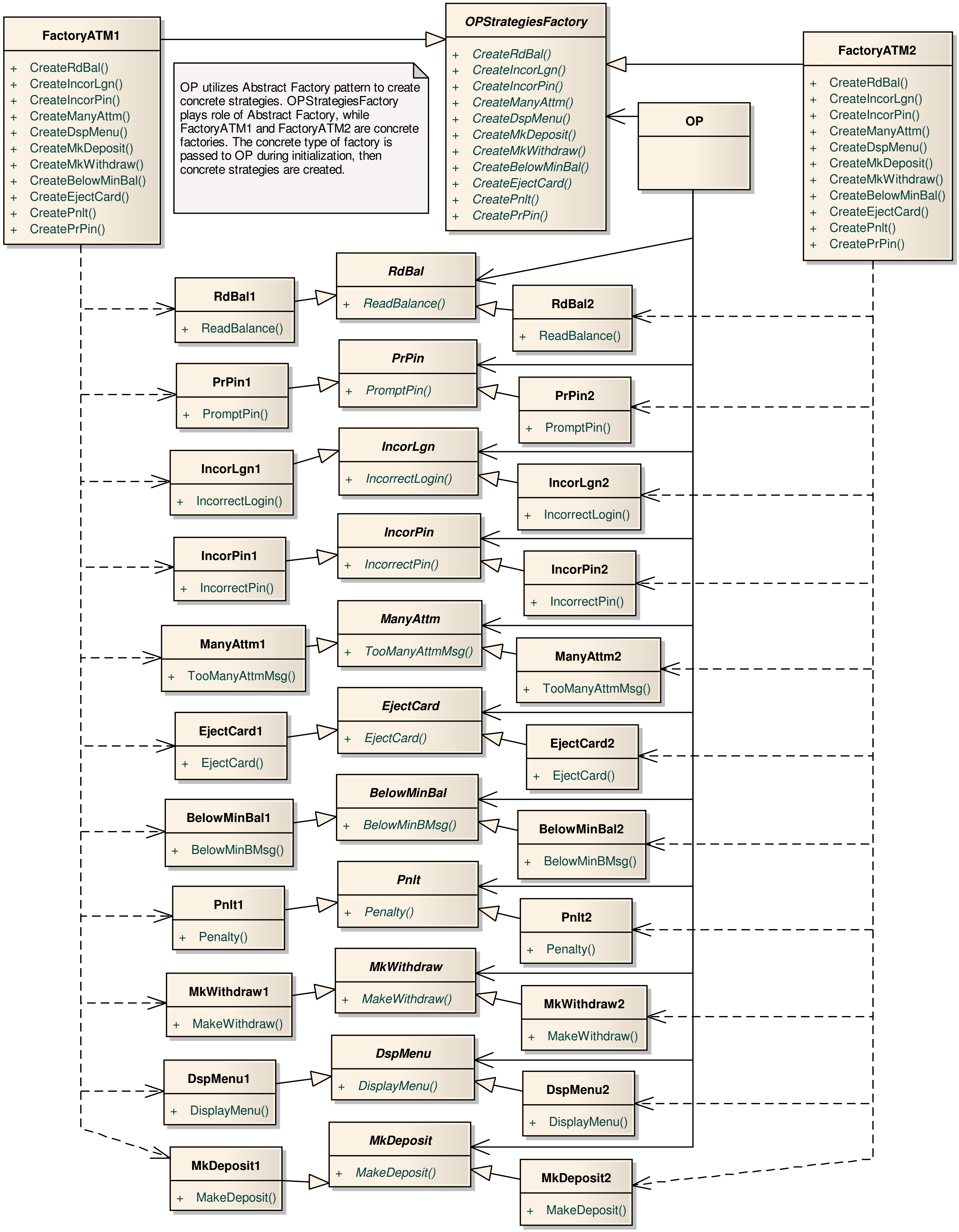


OP - Output Processor

Design solution for this component utilizes strategy pattern.
OP invokes actions on abstract strategies. Pointers are actualy pointing to concrete startegies created during initialization of the system and correspond to platfrom dependent behaviour of ATM1 or ATM2.

The creation of concrete strategies is a responsibility of abstract factory design solution which is also implemented in the system and is shown on other diagram.

OP operations
read_balance() {rdBal->ReadBalance()}
prompt_for_PIN(){prPin->PromptPin()}
incorrect_login_msg(){incLgn->IncorrectLogin()}
incorrect_pin_msg(){incPin->IncorrectPin()}
too_many_attempts_msg(){manyAttm->TooManyAttmMsg()}
display_menu(){dspMnu->DisplayMenu()}
make_deposit(){mkDep->MakeDeposit()}
make_withdraw(){mkWithdr->MakeWithdraw()}
penalty(){penalty->Penalty()}
below_min_balance_msg(){belowMinB->BelowMinBMsg()}
eject_card(){ejectCrd->EjectCard()}



FactoryATM1
+ CreateRdBal()
+ CreateIncorLgn()
+ CreateIncorPin()
+ CreateManyAttm()
+ CreateDspMenu()
+ CreateMkDeposit()
+ CreateMkWithdraw()
+ CreateBelowMinBal()
+ CreateEjectCard()
+ CreatePnlt()
+ CreatePrPin()
OP utilizes Abstract Factory pattern to create concrete strategies. OPStrategiesFactory plays role of Abstract Factory, while FactoryATM1 and FactoryATM2 are concrete factories. The concrete type of factory is passed to OP during initialization, then concrete strategies are created.
OPStrategiesFactory
+ CreateRdBal()
+ CreateIncorLgn()
+ CreateIncorPin()
+ CreateManyAttm()
+ CreateDspMenu()
+ CreateMkDeposit()
+ CreateMkWithdraw()
+ CreateBelowMinBal()
+ CreateEjectCard()
+ CreatePnlt()
+ CreatePrPin()
FactoryATM2
+ CreateRdBal()
+ CreateIncorLgn()
+ CreateIncorPin()
+ CreateManyAttm()
+ CreateDspMenu()
+ CreateMkDeposit()
+ CreateMkWithdraw()
+ CreateBelowMinBal()
+ CreateEjectCard()
+ CreatePnlt()
+ CreatePrPin()
OP
RdBal1
+ ReadBalance()
RdBal
+ ReadBalance()
RdBal2
+ ReadBalance()
PrPin1
+ PromptPin()
PrPin
+ PromptPin()
PrPin2
+ PromptPin()
IncorLgn1
+ IncorrectLogin()
IncorLgn
+ IncorrectLogin()
IncorLgn2
+ IncorrectLogin()
IncorPin1
+ IncorrectPin()
IncorPin
+ IncorrectPin()
IncorPin2
+ IncorrectPin()
ManyAttm1
+ TooManyAttmMsg()
ManyAttm
+ TooManyAttmMsg()
ManyAttm2
+ TooManyAttmMsg()
EjectCard1
+ EjectCard()
EjectCard
+ EjectCard()
EjectCard2
+ EjectCard()
BelowMinBal1
+ BelowMinBMsg()
BelowMinBal
+ BelowMinBMsg()
BelowMinBal2
+ BelowMinBMsg()
Pnlt1
+ Penalty()
Pnlt
+ Penalty()
Pnlt2
+ Penalty()
MkWithdraw1
+ MakeWithdraw()
MkWithdraw
+ MakeWithdraw()
MkWithdraw2
+ MakeWithdraw()
DspMenu1
+ DisplayMenu()
DspMenu
+ DisplayMenu()
DspMenu2
+ DisplayMenu()
MkDeposit1
+ MakeDeposit()
MkDeposit
+ MakeDeposit()
MkDeposit2
+ MakeDeposit()

Operations of the Input Processor (ATM-1)

```
// ATM card is inserted where p is a pin, y is an user's identification #, and a is a balance
card (int p, int y, int a) {
        tempb=a;
        pn=p;
        id=y;
        m->Card();
}
// deposit amount d
deposit (int d) {
        dp=d;
        m->Deposit();
        balanceLevel();
}
// withdraw amount w
withdraw (int w) {
        wd=w;
        m->Withdraw();
        balanceLevel();
}
// provides pin #
pin (int p) {
        if (p==pn) {
                m->CorrectPin();
                balanceLevel();
        }
        else m->IncorrectPin(maxatt);
}
// login where y is a client's identification #
login(int y) {
        if (y==id) m->CorrectLogin();
        else m->IncorrectLogin();
}
// display the current balance
balance() {m->Balance();}
// logout from the ATM
exit() {m->Exit();}
// private method; defines whether balance position is below or above min
private balanceLevel() {
        if (b< minbal) m->BelowMinBalance();
        else m->AboveMinBalance();
}
```

Attributes

```
int b              // account balance
int pn             // card PIN
int id             // user's ID
int dp             // deposit amount
int wd             // withdrawal amount
int maxatt = 2     // maximum attempts allowed for entering PIN
int minbal = 100   // min balance below which account considered overdrawn
MDA-EFS* m         // pointer to MetaModel object
int penalty = 2    // penalty applied on overdraw event
int tempb          // variable that temporary holds balance read from the card until IP decides to copy it to b
```

Operations of the Input Processor (ATM-2)

```
// ATM card is inserted where p is a pin, y is an user's identification #, and a is a balance
CARD (string p, string y, float a) {
        tempb=a;
        pn=p;
        id=y;
        m->Card();
}
// provides pin #
PIN (string x) {
        if (x==pn) {
                m->CorrectPin();
                balanceLevel();
        }
        else m->IncorrectPin(maxatt);
}
// deposit amount d
DEPOSIT (float d) {
        dp=d;
        m->Deposit();
        balanceLevel();
}
// withdraw amount w
WITHDRAW (float w) {
        wd=w;
        m->Withdraw();
        balanceLevel();
}
// display the current balance
BALANCE() {m->Balance();}
// login where y is a client's identification #
login (string y) {
        if (y==id) m->CorrectLogin();
        else m->IncorrectLogin();
}
// logout from the ATM
exit() {m->Exit();}
// private method; defines whether balance position is below or above min
private balanceLevel() {
        if (b< minbal) m->BelowMinBalance();
        else m->AboveMinBalance();
}
```

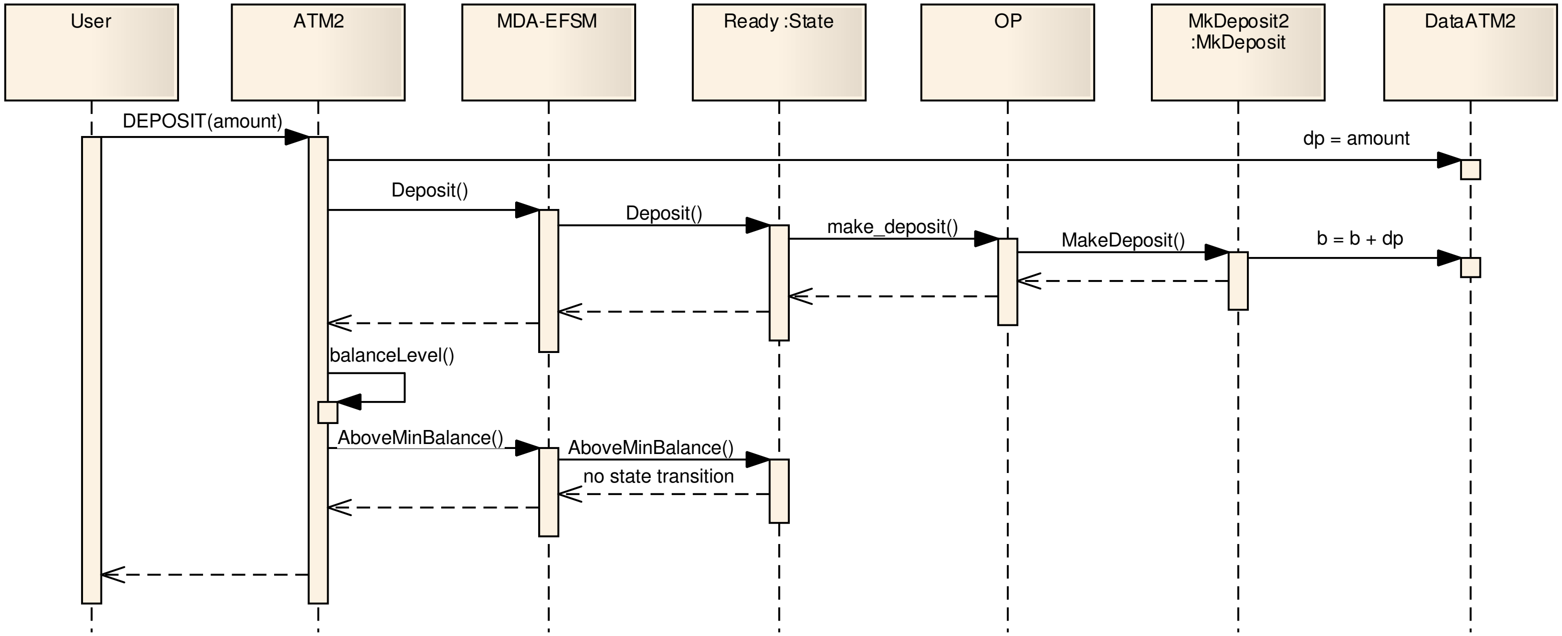
Attributes

```
float b              // account balance
string pn            // card PIN
string id            // user's ID
float dp             // deposit amount
float wd             // withdrawal amount
int maxatt = 3       // maximum attempts allowed for entering PIN
float minbal = 50    // min balance below which account considered overdrawn
MDA-EFS* m           // pointer to MetaModel object
float penalty = 4    // penalty applied on overdraw event
float tempb          // variable that temporary holds balance read from the card until IP decides to copy it to b
```

4. Dynamics. Sequence Diagrams

Next two sequence diagrams explain system dynamics in 2 typical scenarios:

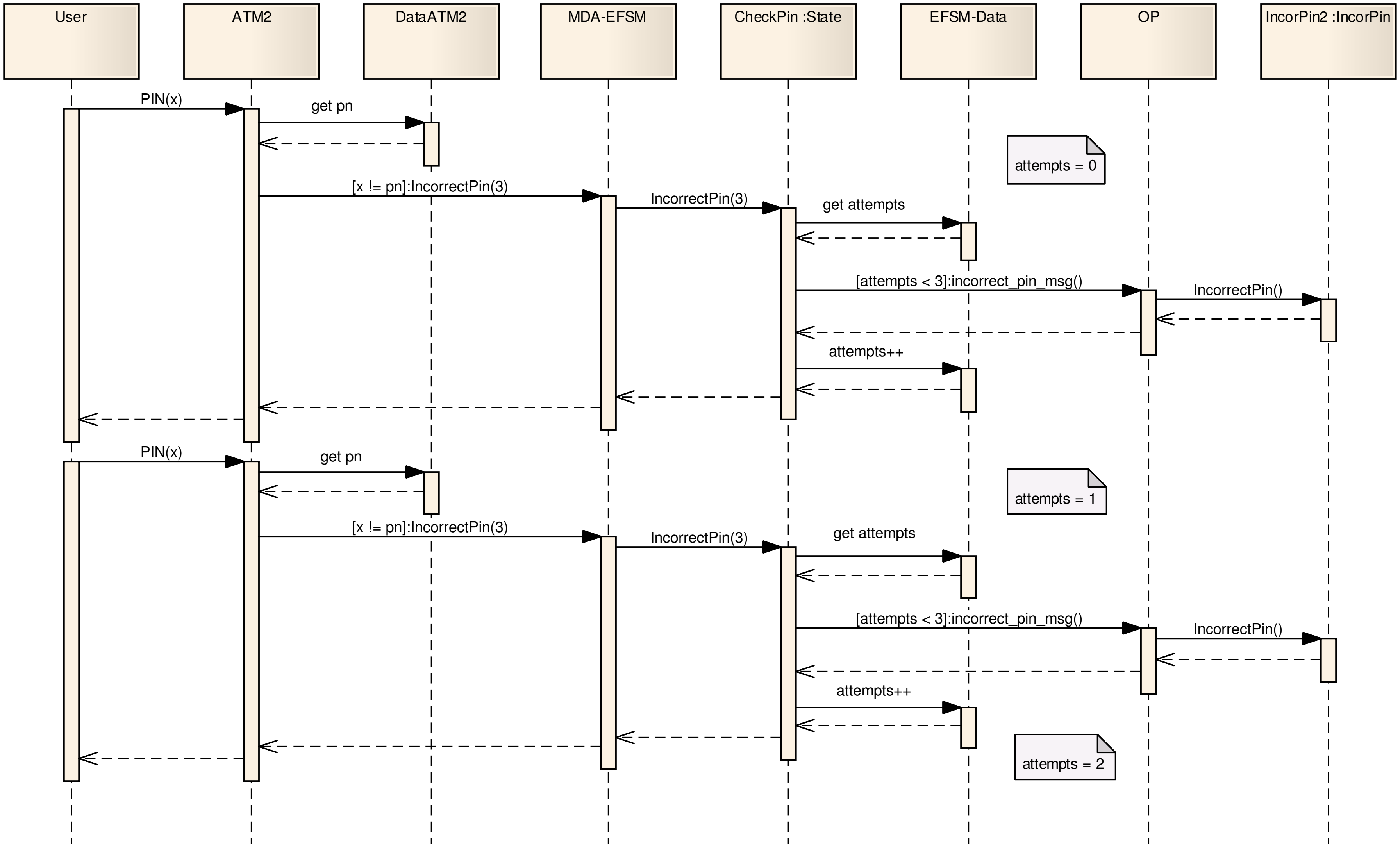
a. Scenario I – deposit in ATM-2 component
b. Scenario II – incorrect pin is entered two times in the ATM-2 component



This sequence diagram represents scenario for Deposit in ATM2

Assumption - the account is above minimum balance
(MDA-EFSM in Ready state)

After the Deposit() operation is returned by MDA-EFSM, ATM2 automaticaly checks the ballance and issues AboveMinBalance() event. Since Ready state does not have implementation for this event, the MDA-EFSM stays in the Ready state (no state transition is performed).



This sequence diagram represents scenario for 2 consecutive attempts to enter incorrect PIN in ATM2

Assumption - the system is in CheckPin state and no PINs has been attempted

The diagram shows events/actions propagation through the system. It's also shown how MDA-EFSM internal data counter "attempts" is modified. There are no state transitions on the diagram since internal counter has not reached maximum allowed attempts (3).

5. Conclusion

This project is an attempt to utilize architecture and design solutions presented in CS586 Software System Architectures course as well as to demonstrate Object Oriented Design approaches and principles along with UML usage. Presented system is based on Model Driven Architecture and its components are designed utilizing 3 widely used patterns – State, Strategy and Abstract Factory.