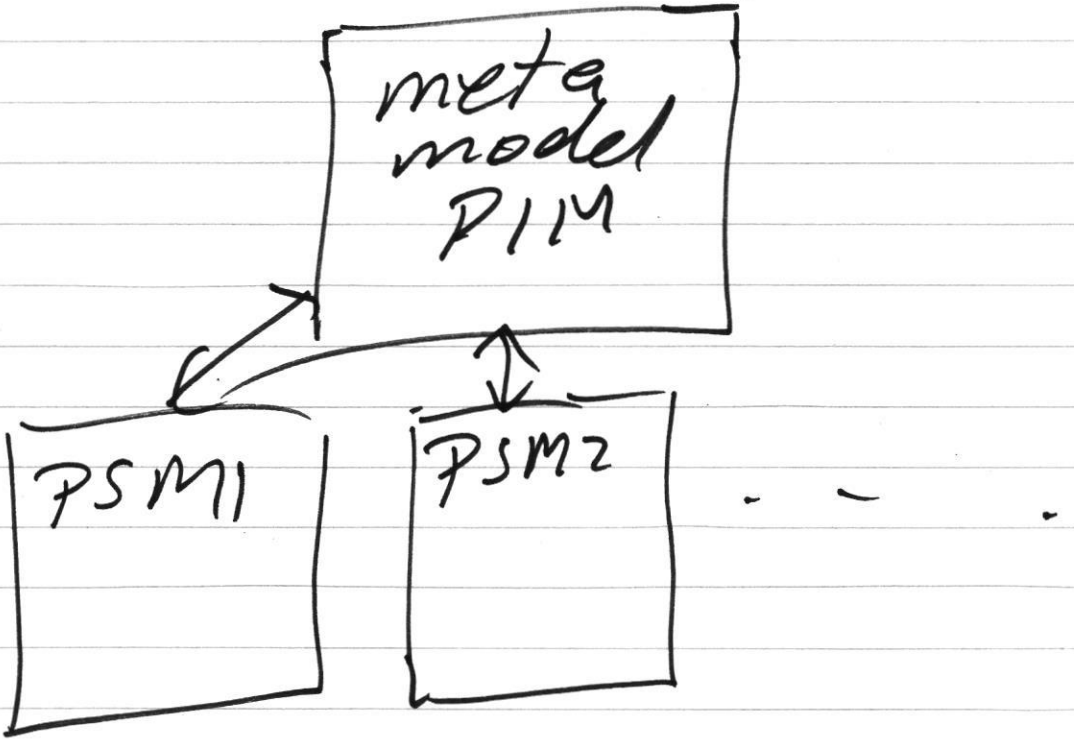
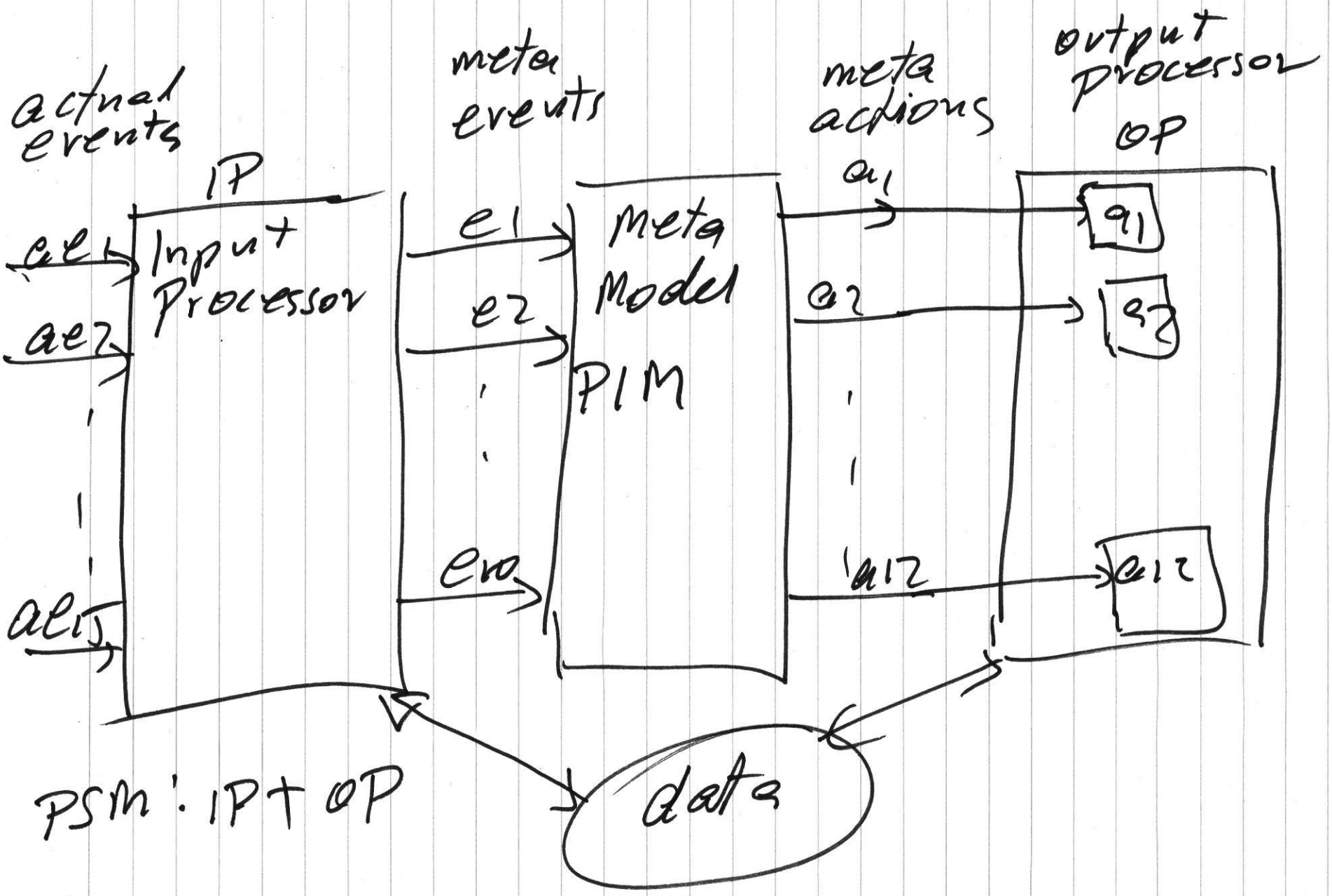
Homework #2 is posted

Model Driven Architecture

Idea:
seperate the platform independent behavior
(PIM)

FROM
platform specific behavior
(PSM)




actual events
IP
Input Processor
ae1
ae2
aeI
meta events
e1
e2
eM
meta Model
PIM
meta actions
a1
a2
aM2
output Processor
OP
q1
q2
qM2
data
PSM: IP + OP

Modeling Languages

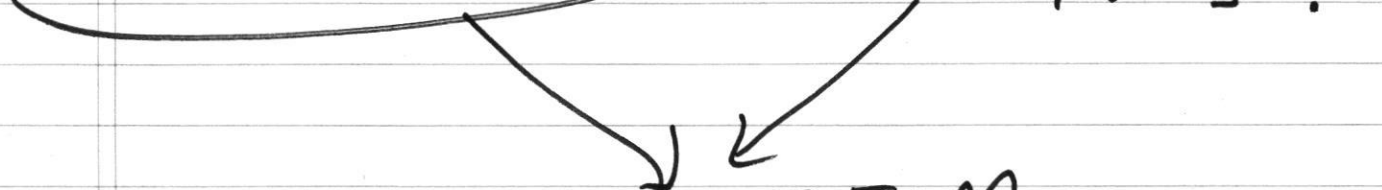
EFSM
VFSM

EFSM

data
easy to understand

VFSM

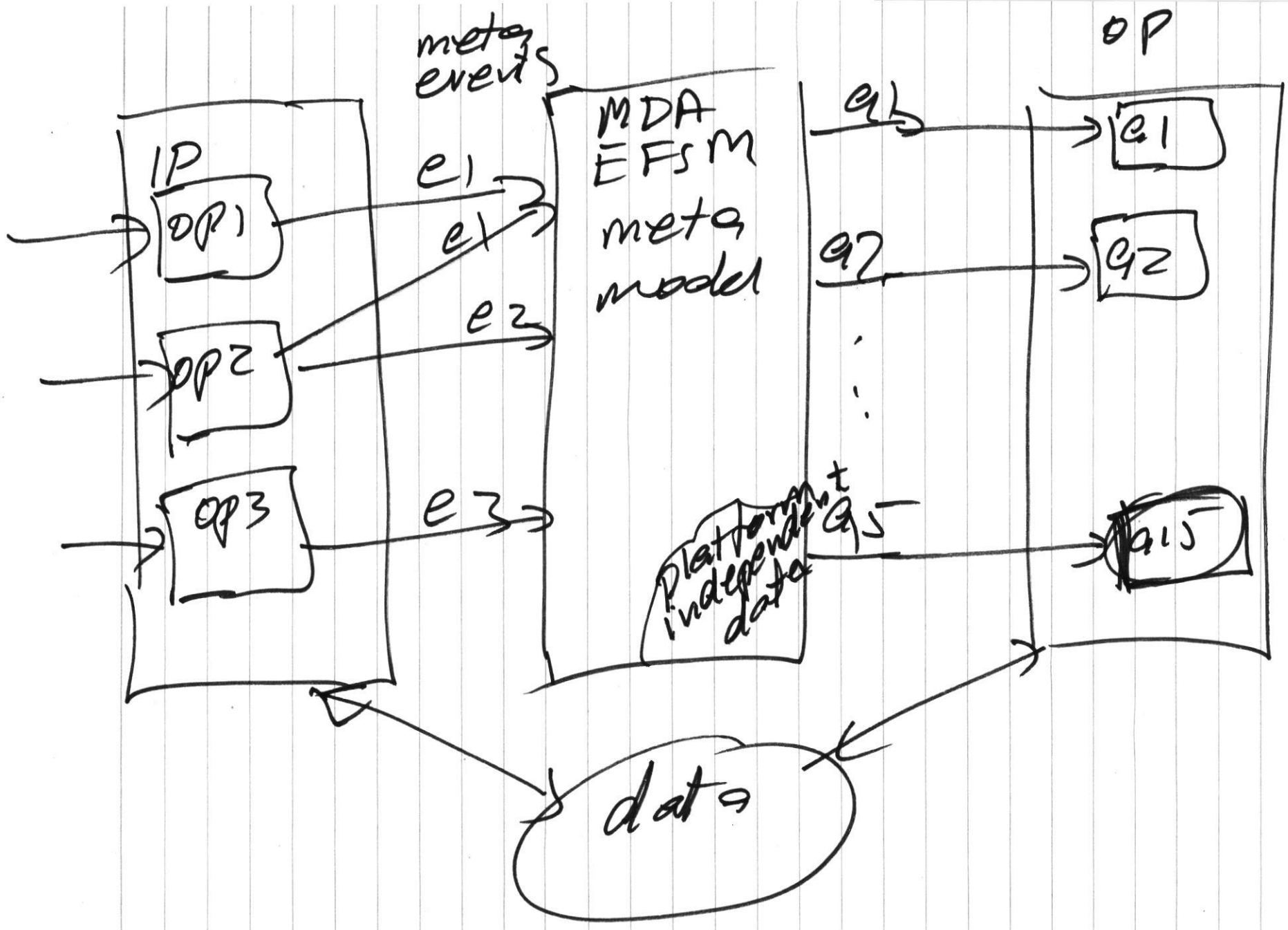
no data
complex execution rules.

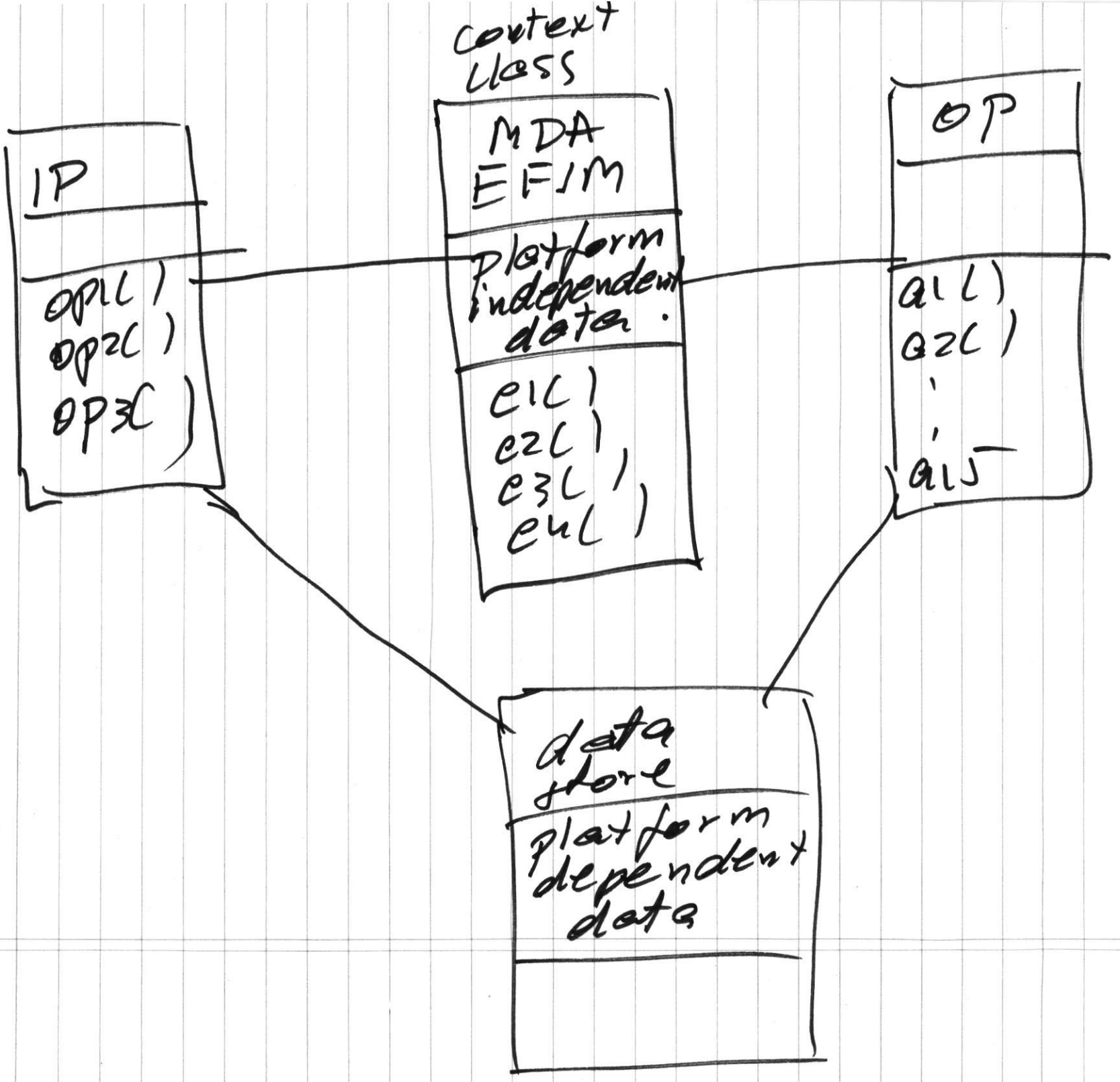


MDA-EFSM

EFSM with restrictions

1. no data
   or
   platform independent data
2. events/operations/actions must be platform independent


meta events
IP
op1
op2
op3
e1
e1
e2
e3
MDA EFSM meta model
platform independent data
OP
e1
e2
e3
data


Context class
MDA EFIM
Platform independent data
IP
op1()
op2()
op3()
OP
q1()
q2()
qlJ
data store
Platform dependent data
e1()
e2()
e3()
e4()

MDA-EFSM

We need to identify:

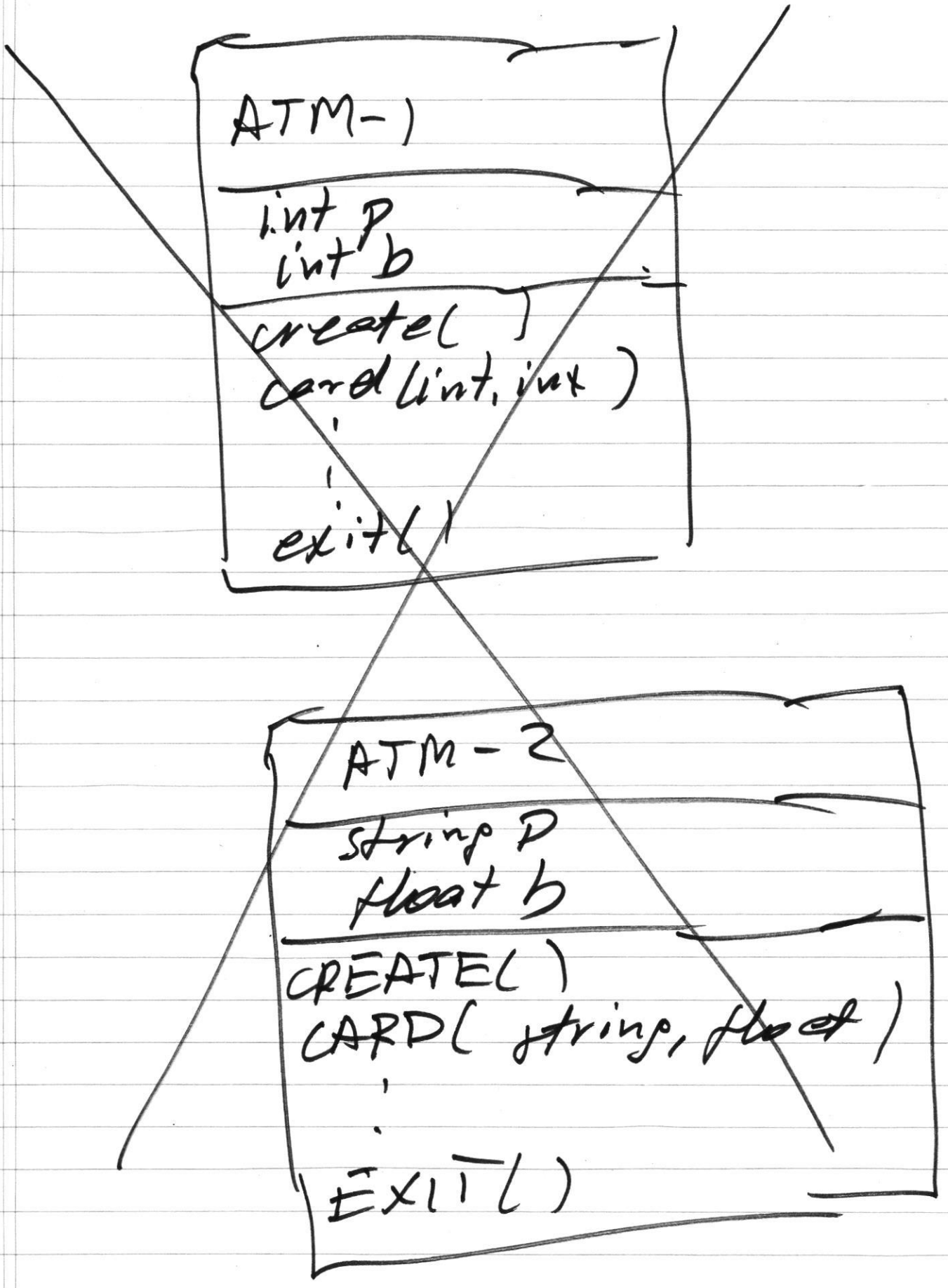
1. meta events: e1, e2 ...
2. meta actions: a1, a2 ...
3. state diagram (MDA-EFSM)

ATM-1 component

```
create()
card(int x, int x)
pin(int a)
deposit(int d)
withdraw(int w)
balance()
exit()
```

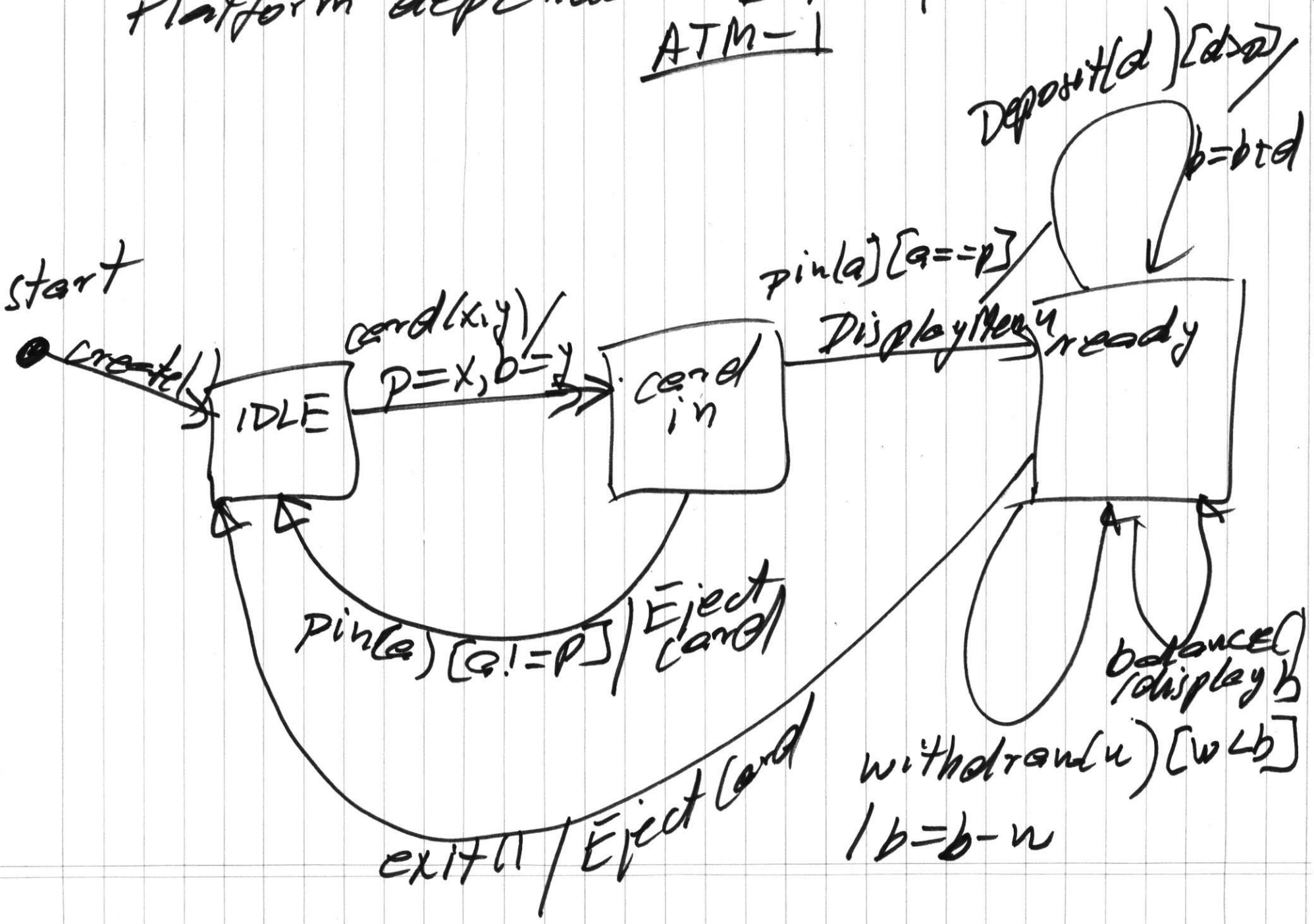
ATM-2 component

```
CREATE()
CARD(string x, float y)
PIN(string a)
DEPOSIT(float d)
WITHDRAW(float w)
BALANCE()
EXIT()
```


ATM-1
int P
int b
create( )
card(int, int)
exit( )
ATM-2
string P
float b
CREATE( )
CARD(string, float)
EXIT( )

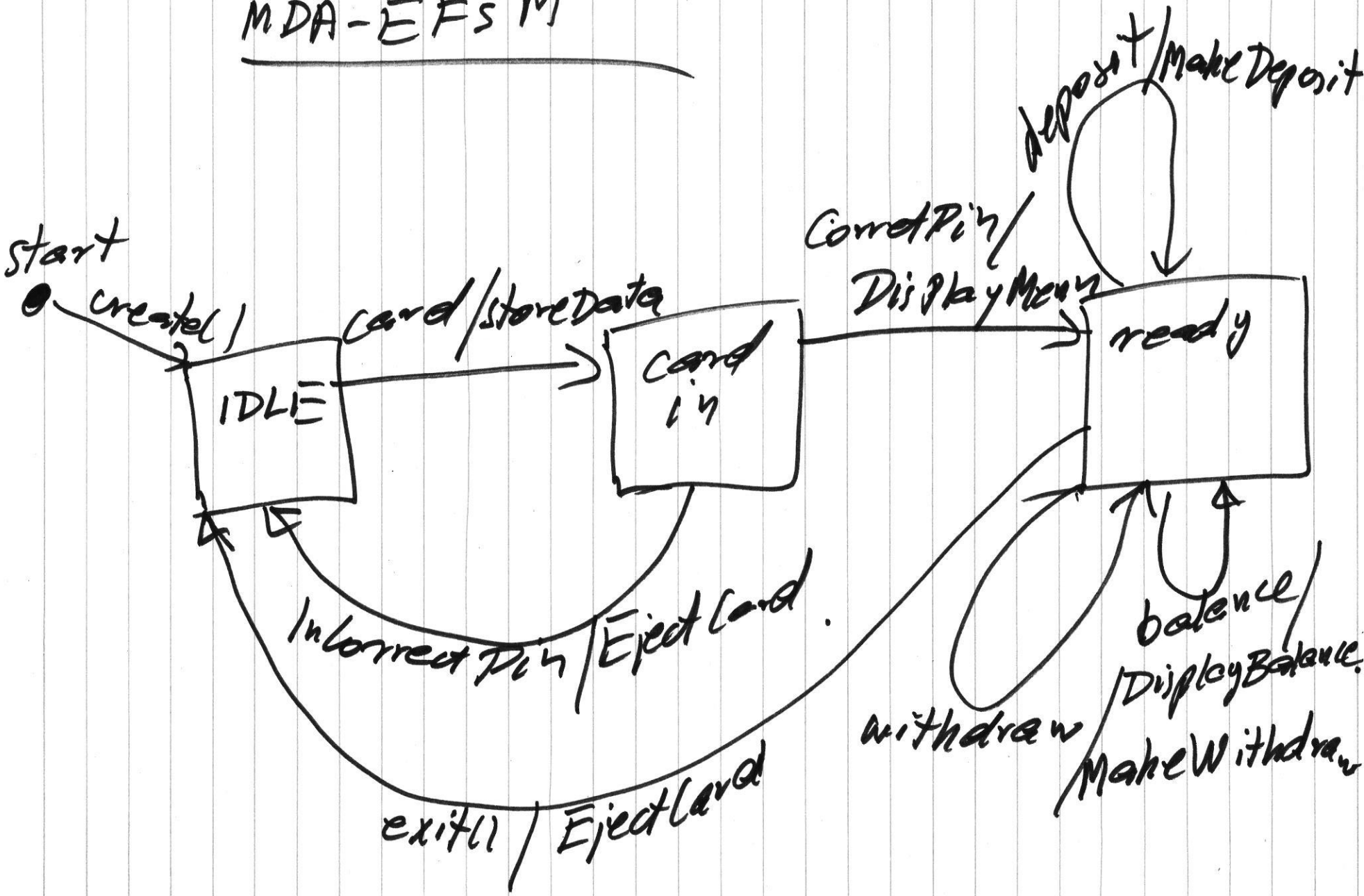
Platform dependent EFSM

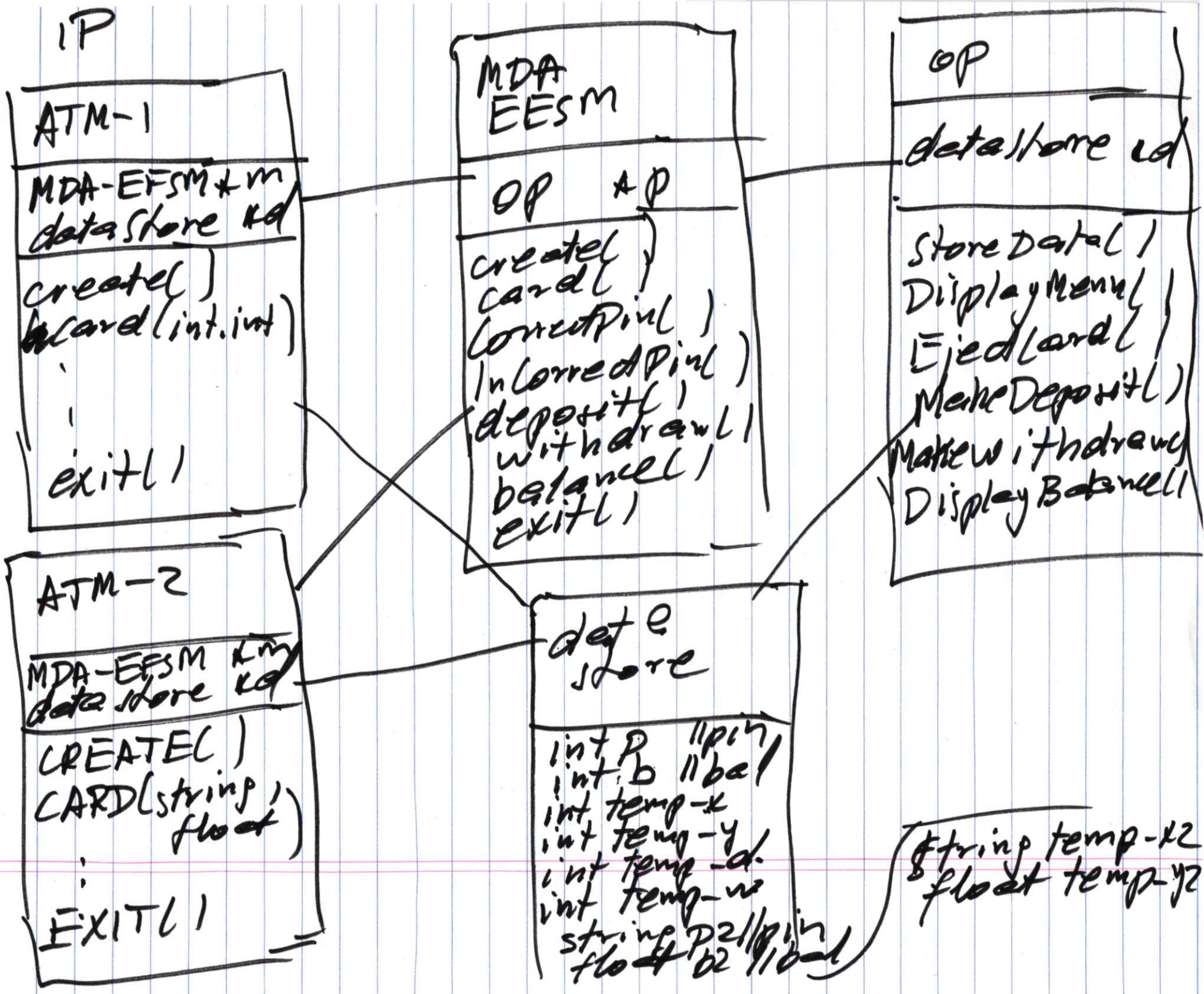
ATM-1



meta events	meta actions
create()	StoreData()
card()	DisplayMenu()
CorrectPin()	EjectCard()
IncorrectPin()	MakeDeposit()
deposit()	MakeWithdraw()
withdraw()	DisplayBalance()
balance()	
exit()	

MDA-EFSM


start
create()
IDLE
card()/storeData
card in
CorrectPin/
DisplayMenu
ready
deposit/MakeDeposit
IncorrectPin/EjectCard
exit()/EjectCard
withdraw/
MakeWithdraw
balance/
DisplayBalance


IP
ATM-1
MDA-EFSM *m
dataStore *d
create()
Card(int,int)
exit()
ATM-2
MDA-EFSM *m
dataStore *d
CREATEC()
CARD(string, float)
EXITC()
MDA EESM
OP AP
create()
card()
CorrectPin()
InCorredPin()
deposit()
withdrawl()
balance()
exit()
OP
dataStore *d
StoreData()
DisplayMenu()
EjectCard()
MakeDeposit()
MakeWithdrawl()
DisplayBalance()
date Store
int P //pin
int b //bal
int temp-x
int temp-y
int temp-d
int temp-w
string P2 //pin
float b2 //bal
string temp-x2
float temp-y2

ATM-1 class

```
created()
{ m -> created() }
```

```
card(int x, int y)
{  d -> temp_x = x
   d -> temp_y = y
   m -> card()
}
```

```
pin( int a )
{  if (a == d->p)  m -> correctPin()
   else m -> IncorrectPin()
}
```

```
deposit( int d)
{   if [d>0] {
        d -> temp_d = d
        m -> deposit()
    }
}
```

ATM - 1

```
withdraw (int w)
{ if (w < d->b) {
        d->temp_w = w
        m->withdraw()
  }
}
```

```
balance()
{ m->balance()
}
```

```
exit()
{ m->exit()
}
```

OP class

```
storeDate()
{
    d->p = d->temp-x;
    d->b = d->temp-y;
}
```

```
DisplayMenu()
{ Implementation of
        display menu
}
```

```
EjectCard()
{ Implementation of
        ejecting card
}
```

```
MakeDeposit()
{  d->b = d->b + d->temp_d
}
```

```
MakeWithDraw()
{  d->b = d->b - d->temp-w;
}
```

DisplayBalance()
{ implementation
of displaying d→b }

ATM-2

CREATEC)

{ m → created( ) ; }

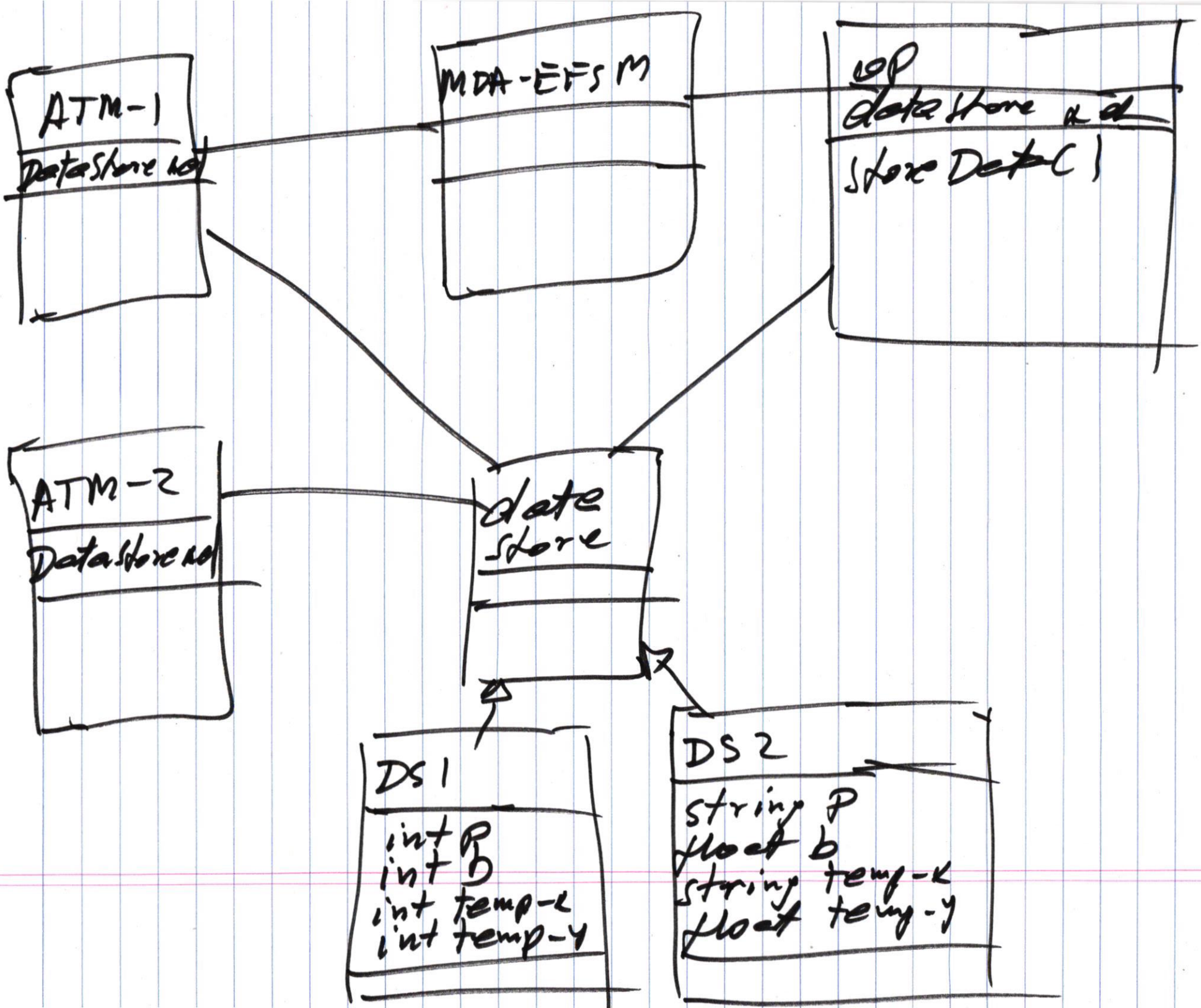
CARD (string x , float y)

{ d → temp_x2 = x ;
d → temp_y2 = y ;
m → card( ) ;
}

CARD (string x , float y)

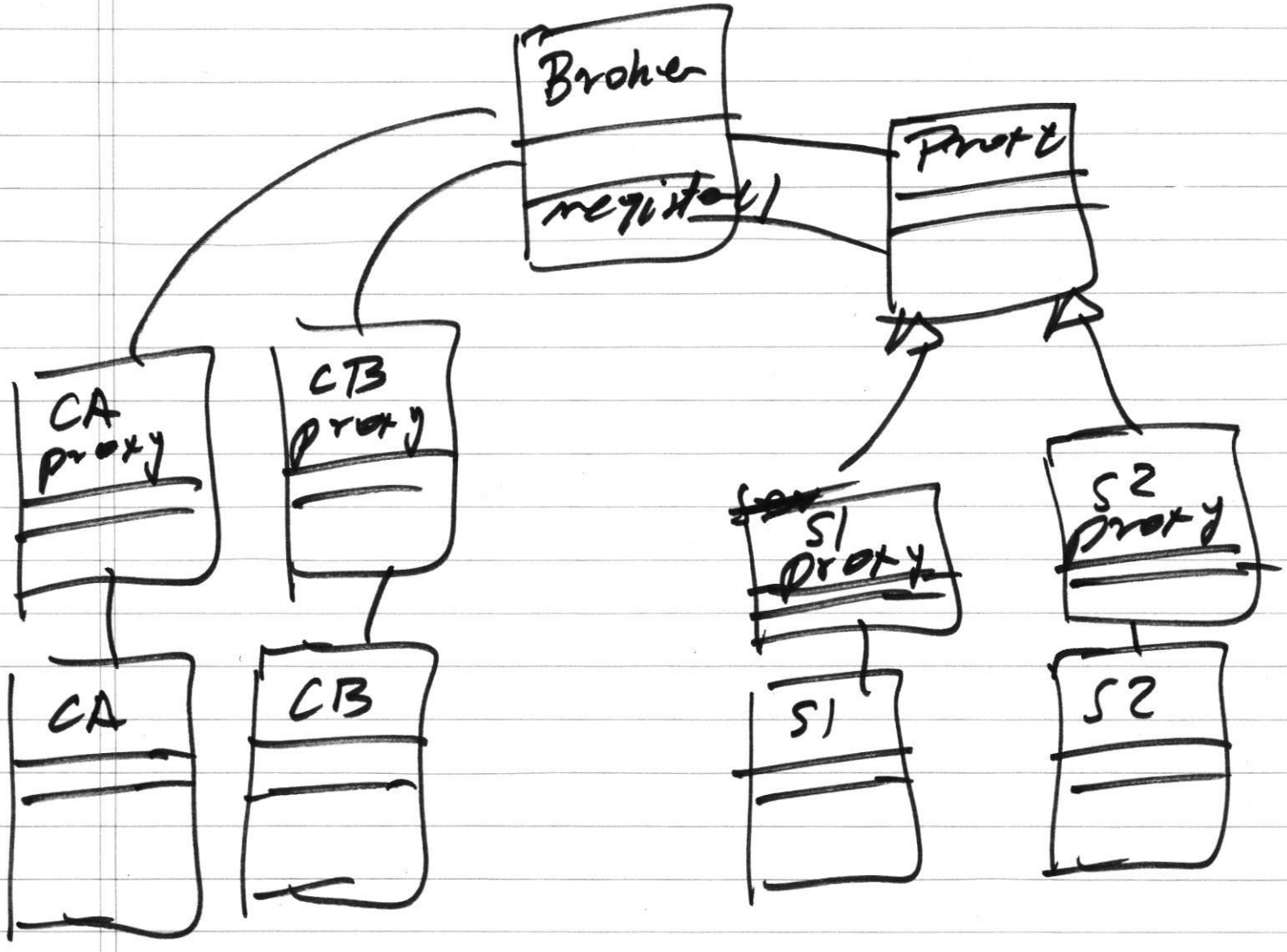
{ d → tempx = x
d → temp_y = y .
m → card( ) ;

}

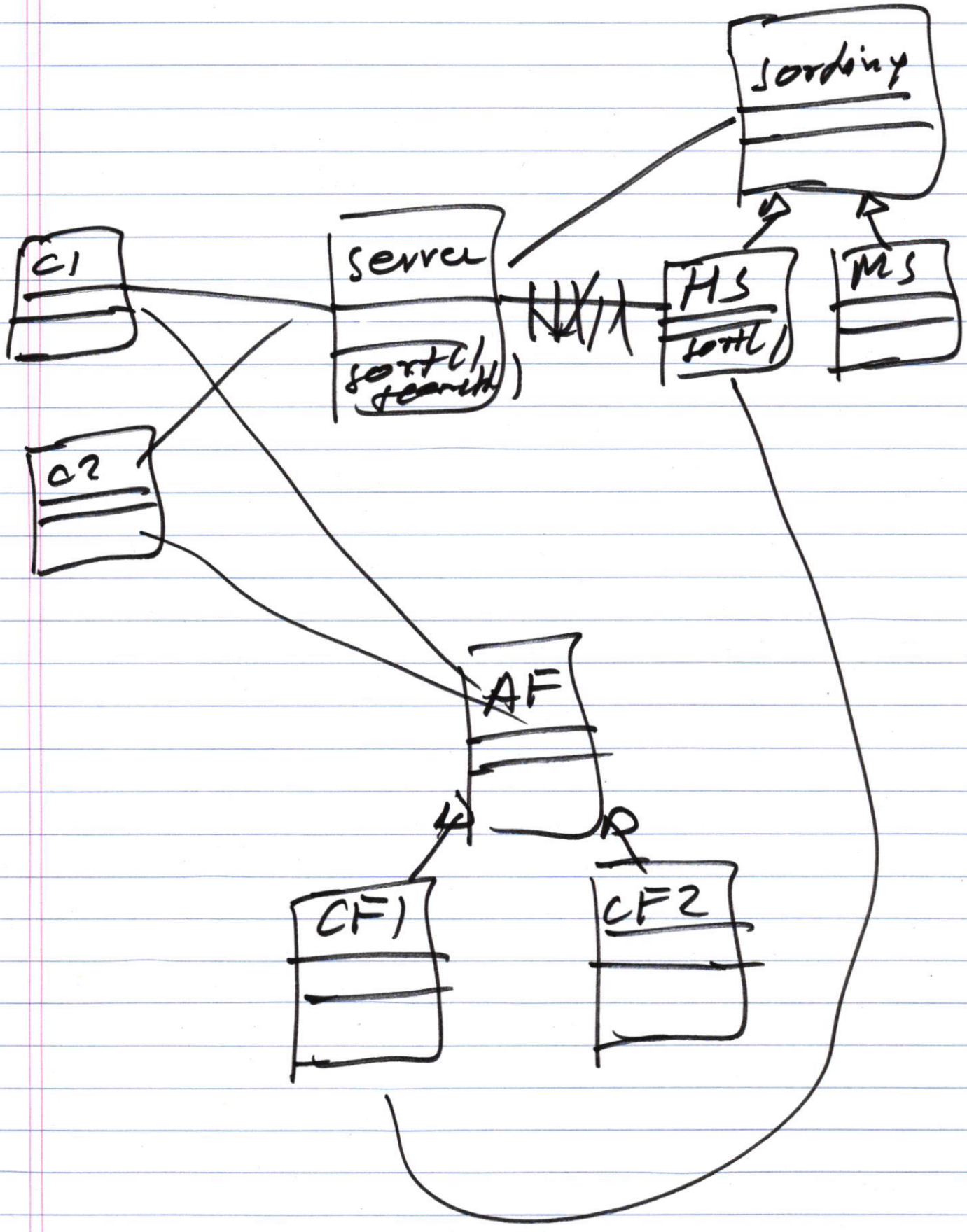

ATM-1
MDA-EFSM
ATM-2
data store
DS1
int P
int D
int temp-c
int temp-y
DS2
string P
float b
string temp-k
float temp-y

Homework #2

Problem #1



Problem #3



HOMEWORK ASSIGNMENT #2

CS 586; Spring 2024
Due Date: **March 5, 2024**
Late homework 50% off
After **March 12,** the homework assignment will not be accepted.

This is an **individual** assignment. **Identical or similar** solutions will be penalized.

Submission: All homework assignments must be submitted on the Blackboard. The submission **must** be as one PDF-file (otherwise, a 10% penalty will be applied).

PROBLEM #1 (35 points)
There exist two servers **S1** and **S2**. Both servers support the following services:

Services supported by server **S1**:
int SetPrice(string, float)
float GetPrice(string)
void BuyStock(string, int)
void SellStock(string, int)

Services supported by server **S2**:
int SetPrice(string, int)
float GetPrice(string)
void BuyStock(int, string)
void SellStock(string, int)
int SetPrice(string, float)

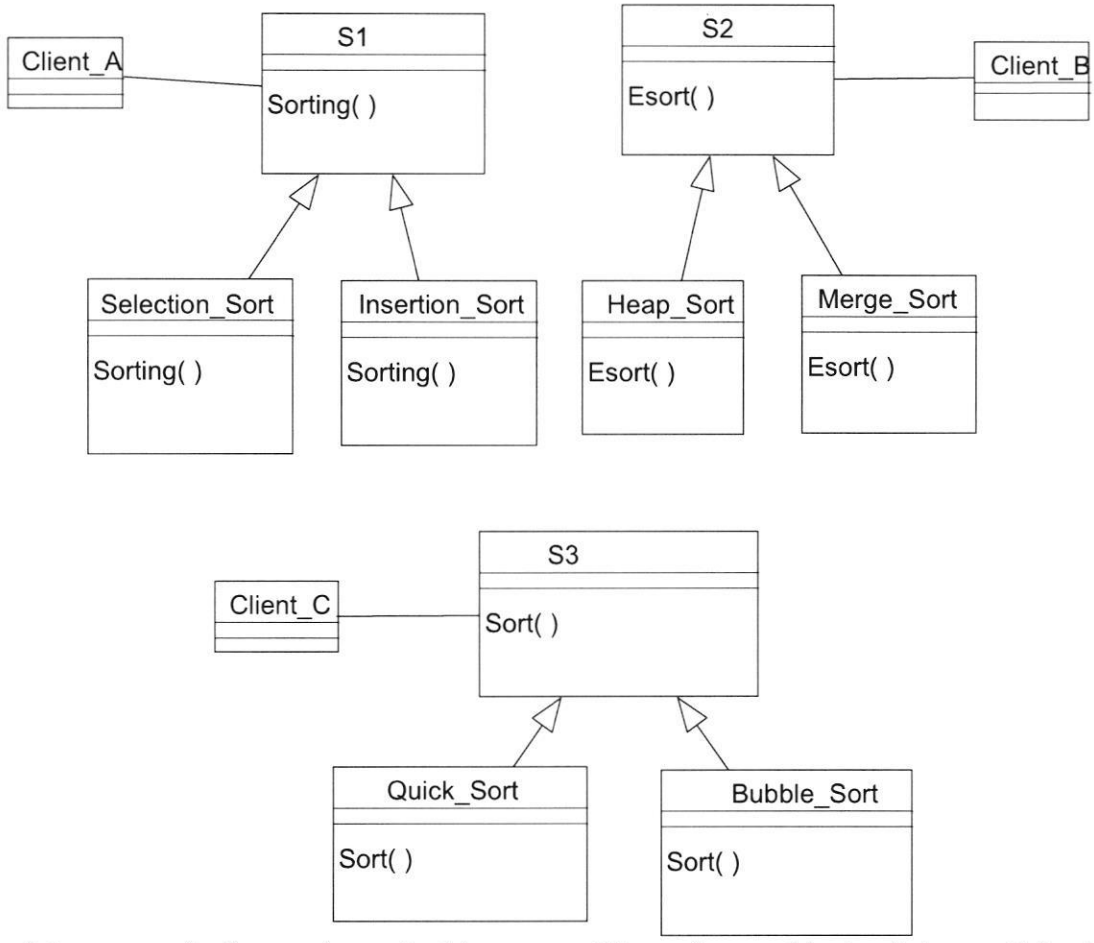
There exist two client processes and they request the following services:

Client-A	**Client-B**
int SetPrice(string, float)	int SetPrice(string, int)
float GetPrice(string)	float GetPrice(string)
void BuyStock(int, string)	void BuyStock(string, int)
void SellStock(string, int)	void SellStock(string, int)

The client processes do not know the location (pointer) of servers that may provide these services. Devise a software architecture using a **Client-Broker-Server** architecture for this problem. In this design, the client processes are not aware of the location of the servers providing these services.

- Provide a class diagram for the proposed architecture. In your design, all components should be **decoupled** as much as possible.
- Provide the pseudocode for all operations of the following components/classes:
  - Broker
  - Client Proxy of *Client-A*
  - Server Proxy of server *S1*.
- Provide a sequence diagram to show how *Client-A* gets "*int SetPrice(string, float"* service.

PROBLEM #2 (30 points)



The design of the system is shown above. In this system *Client_A* uses objects of classes *Selection_Sort* and *Insertion_Sort*, *Client_B* uses objects of classes *Heap_Sort* and *Merge_Sort*, and *Client_C* uses objects of classes *Quick_Sort* and *Bubble_Sort*.

Client_A would like to use objects, operation *Esort(),* of classes *Heap_Sort* and *Merge_Sort* by invoking operation *Sorting()*. *Client_B* would like to use objects, operations *Sort()*, of classes *Quick_Sort* and *Bubble_Sort* by invoking operation *Esort()*. In addition, *Client_C* would like to use objects, operation operation *Esort(),* of classes *Heap_Sort* and *Merge_Sort* by invoking operation *Sort()*.

Provide a design with **minimal** modifications to the existing system using the **Adapter design pattern** in which
(1) *Client_A* can use objects of classes *Heap_Sort* and *Merge_Sort* by invoking operation *Sorting*(),
(2) *Client_B* can use objects of classes *Quick_Sort* and *Bubble_Sort* by invoking operation *Esort(),* and
(3) *Client_C* can use objects of classes *Heap_Sort* and *Merge_Sort* by invoking operation *Sort()*.

Notice that none of the classes shown in the above class diagram should be modified.

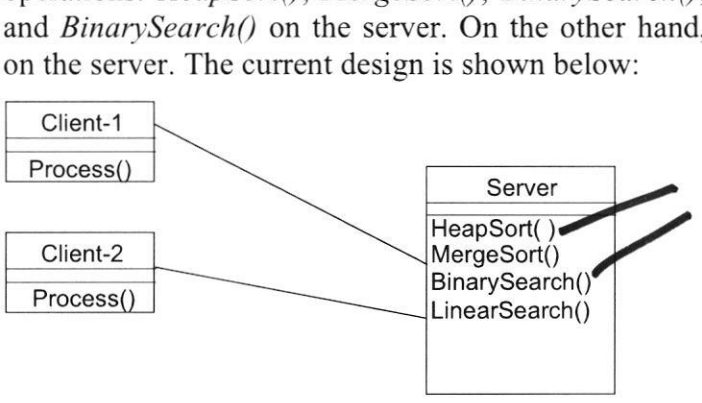
Provide two solutions that are based on:
1. An **association-based** version of the Adapter pattern
2. An **inheritance-based** version of the Adapter pattern

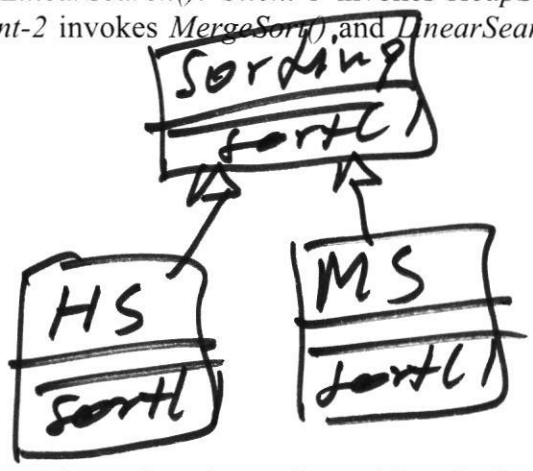
3 adapters. 6 — 11

Provide a class diagram for each solution. You do not have to provide any description for classes/operations of the above class diagram (only new classes/operations should be described using **pseudo-code**).

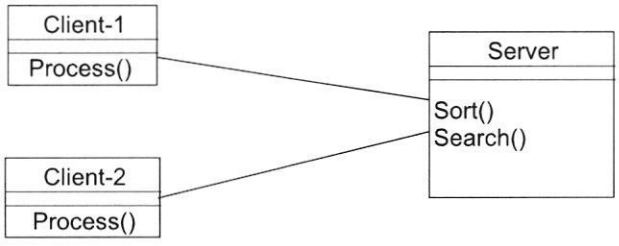
PROBLEM #3 (35 points)

There exist two clients (*Client-1* and *Client-2*) and a *Server* class. The *Server* class supports the following operations: *HeapSort()*, *MergeSort()*, *BinarySearch()*, and *LinearSearch()*. *Client-1* invokes *HeapSort()* and *BinarySearch()* on the server. On the other hand, *Client-2* invokes *MergeSort()* and *LinearSearch()* on the server. The current design is shown below:





In a better design Clients should be shielded from different versions of sorting and searching. In the new design, as shown below, *Client-1* should invoke *Sort()* and *Search()*, where *HeapSort()* and *BinarySearch()* are executed. Similarly, *Client-2* should invoke *Sort()* and *Search()*, where *MergeSort()* and *LinearSearch()* are executed. Notice that the current design does not support this option.



Use the **strategy pattern** and the **abstract factory** design patterns to solve this problem. In your solution, the *Client* classes should be completely **de-coupled** from the issue of invoking appropriate versions of *Sort()* and *Search()*. Notice that in the design new classes/operations should be introduced according to these patterns.

- Provide the class diagram and describe the responsibility of each class and the functionality of each operation using **pseudo-code**. In your design, all components should be **decoupled** as much as possible. In addition, indicate which classes/operations are part of the strategy pattern and which classes/operations are part of the abstract factory pattern.

- Provide a sequence diagram to show how *Client-1* gets services *HeapSort()* and *BinarySearch()* by invoking *Sort()* and *Search()* on the *Server*.