Homework #2 is posted

# Model-Driven Architecture

## MDA

- multi-year architectural paradigm

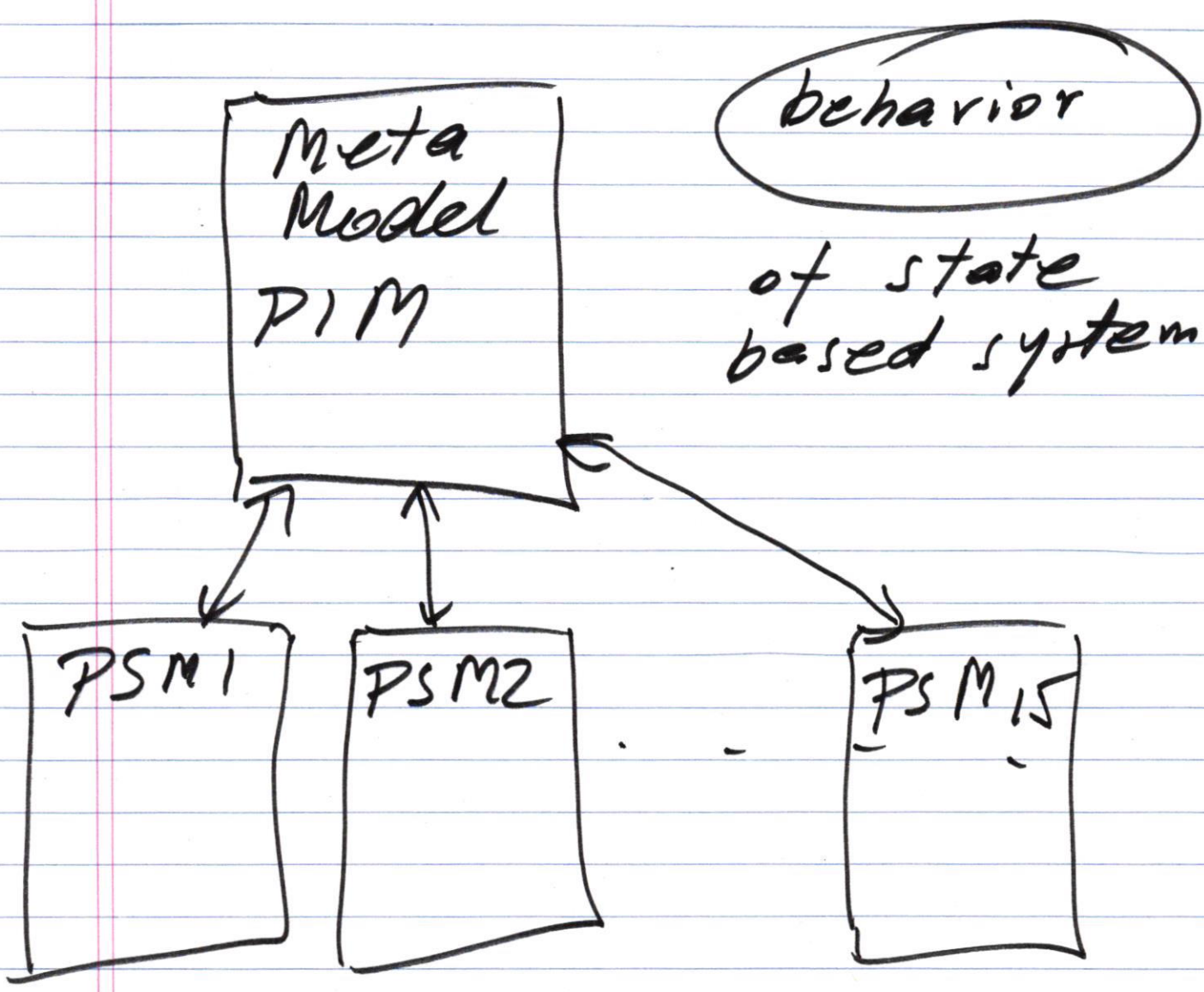
- We want to seperate
  - platform independent aspects of a family of systems

FROM

  - platform specific aspects of the system

# Platform

- operating system
- database
- network
- hardware
- data structures
- algorithms
- . . .

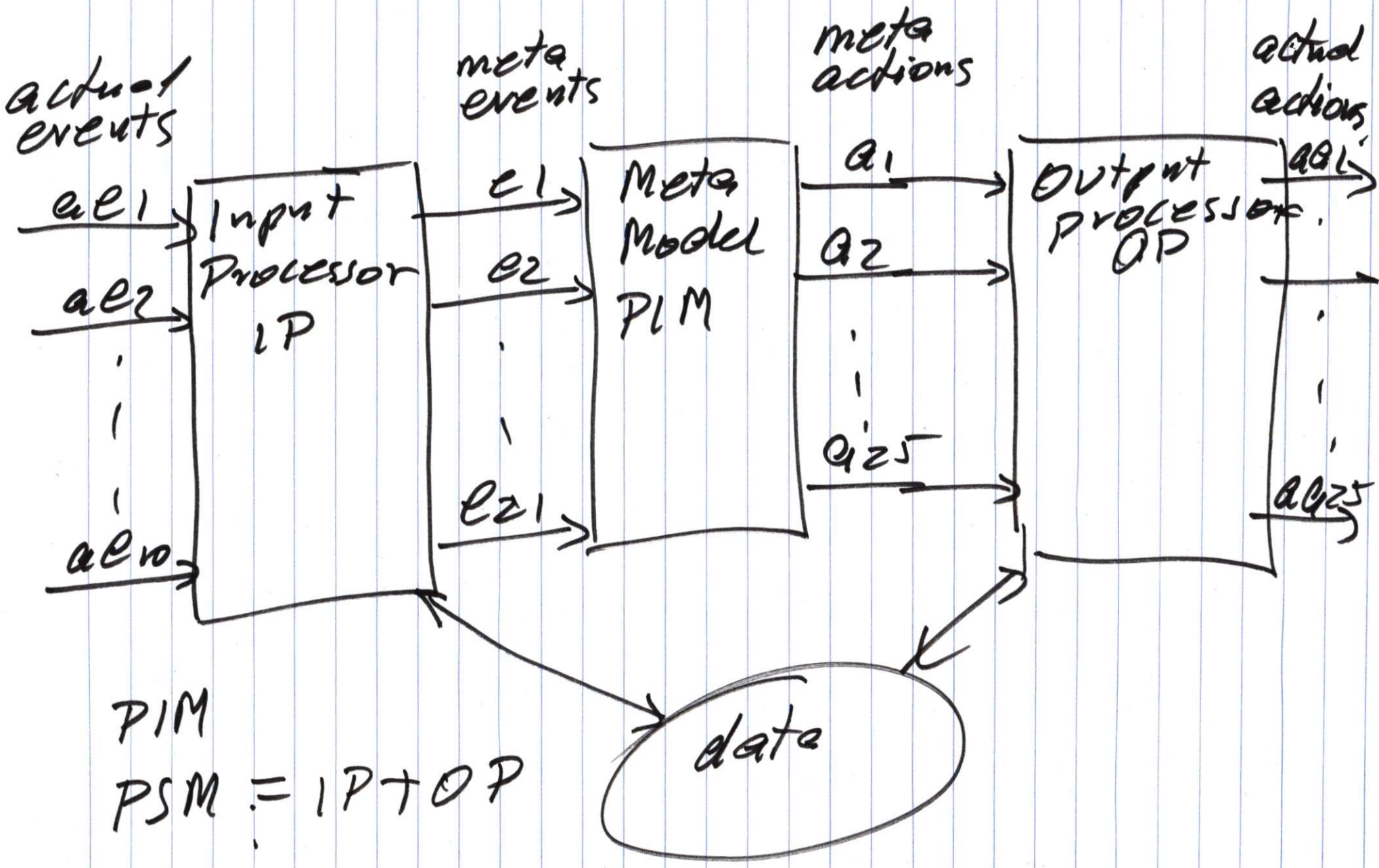

Meta Model PIM
behavior
of state based systems.
PSM1
PSM2
PSM15

# Idea of MDA

PIM: Platform Independent Model.

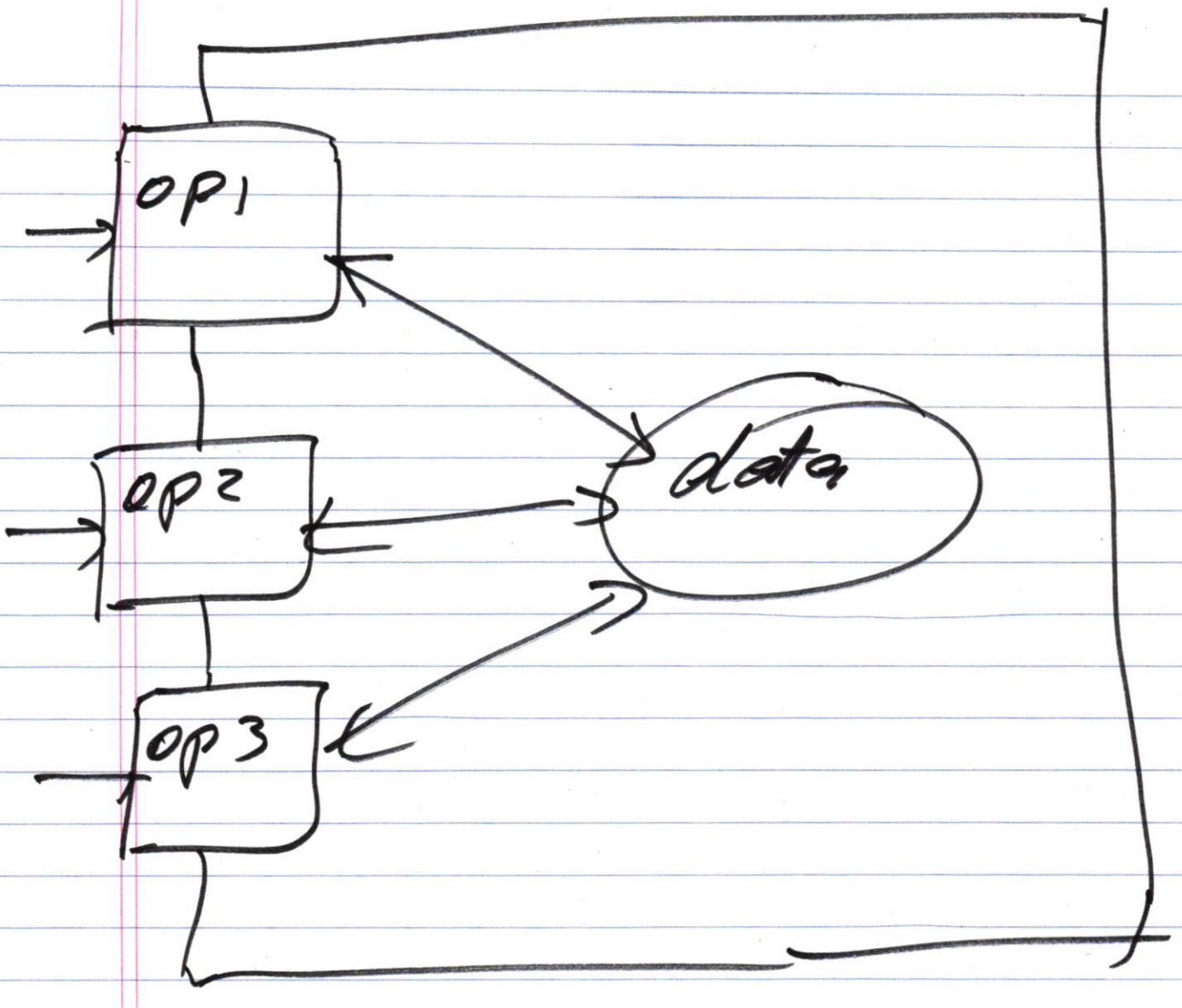
Model will be "stable"

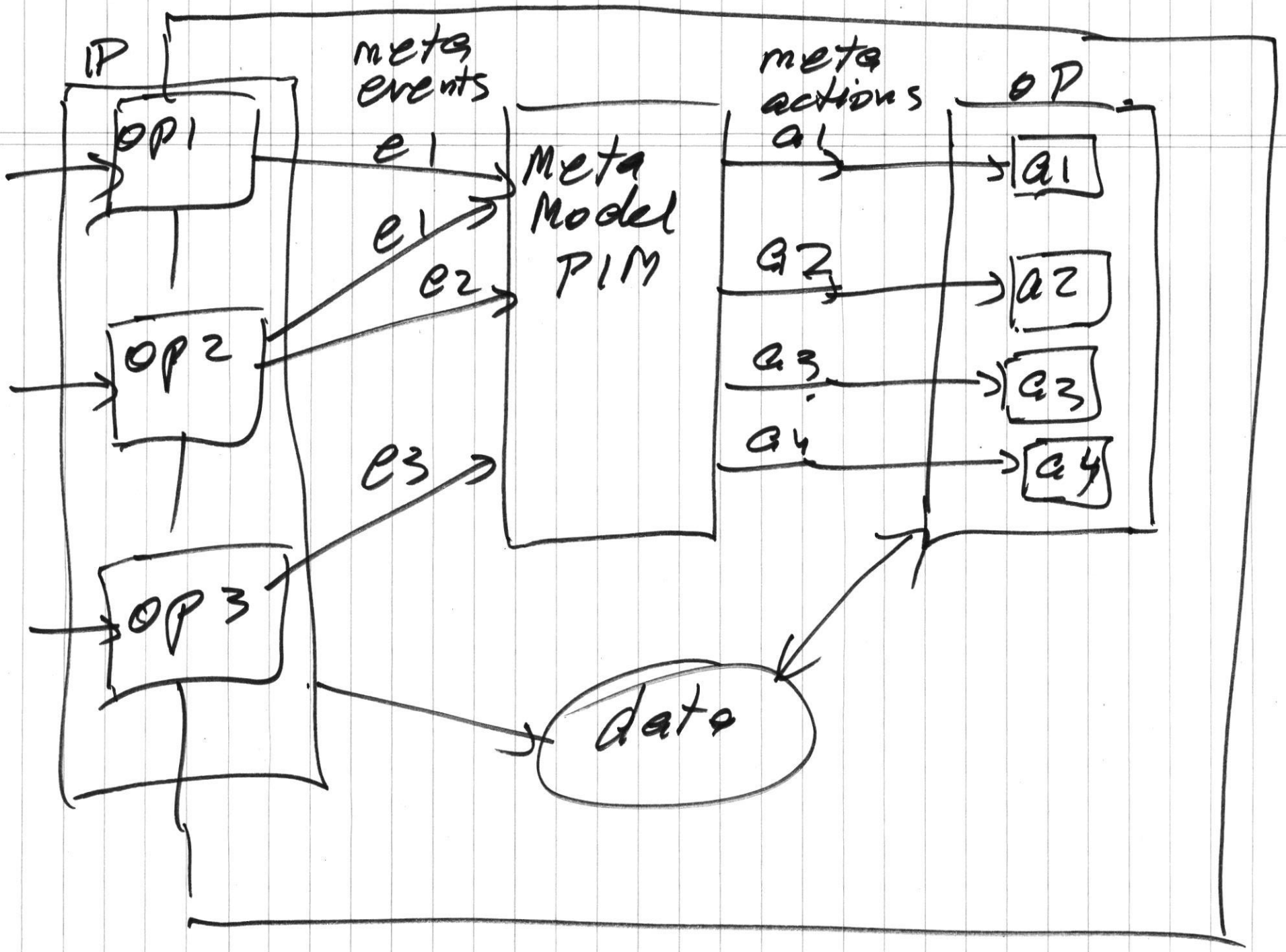
PSM: Platform Specific Model.


actual events
ae1
ae2
aen
Input Processor IP
meta events
e1
e2
ez1
Meta Model PIM
meta actions
a1
a2
az5
Output processor OP
actual actions
aa1
aaz5
data

PIM

PSM = IP + OP

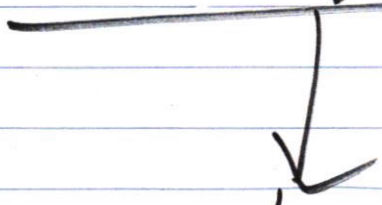

op1
op2
op3
data


IP
op1
op2
op3
meta events
e1
e1
e2
e3
Meta Model PIM
meta actions
a1
a2
a3
a4
OP
a1
a2
a3
a4
data

Meta Model should capture platform independent behavior.

---

State based systems



meta behavior.

## Modeling Language

- FSM
- EFSM
- VFSM

# EFSM

- states
- transitions.
- variables/data
- actual events
- actual actions
- conditions

} platform dependent.

EFSM: platform dependent.

# VFSM: Virtual Finite State Machine

- states
- transitions
- virtual events
- virtual actions
- conditions: Boolean expression.

no data

virtual event: name
(Boolean variable)

virtual action: name

EFSM

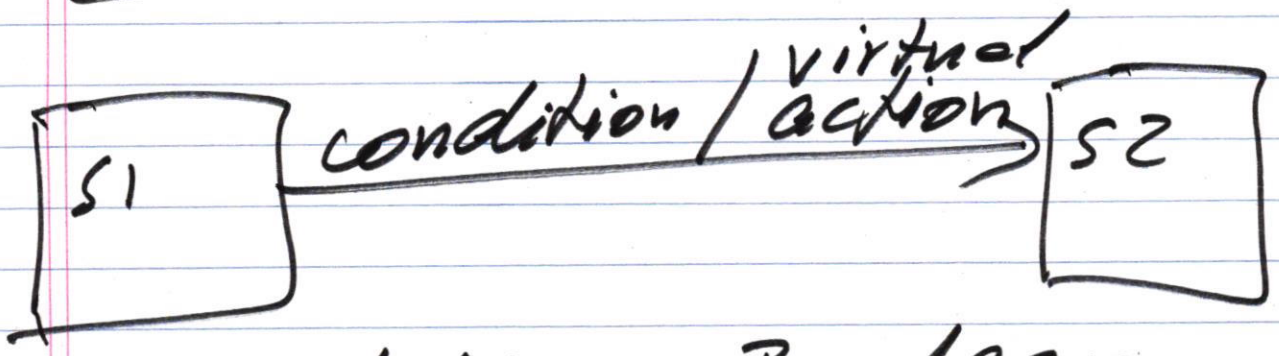
ATM component

actual events

- card(int pin, int bal)
- card(float bal, string pin, string vid)

VFSM: card

# VFSM

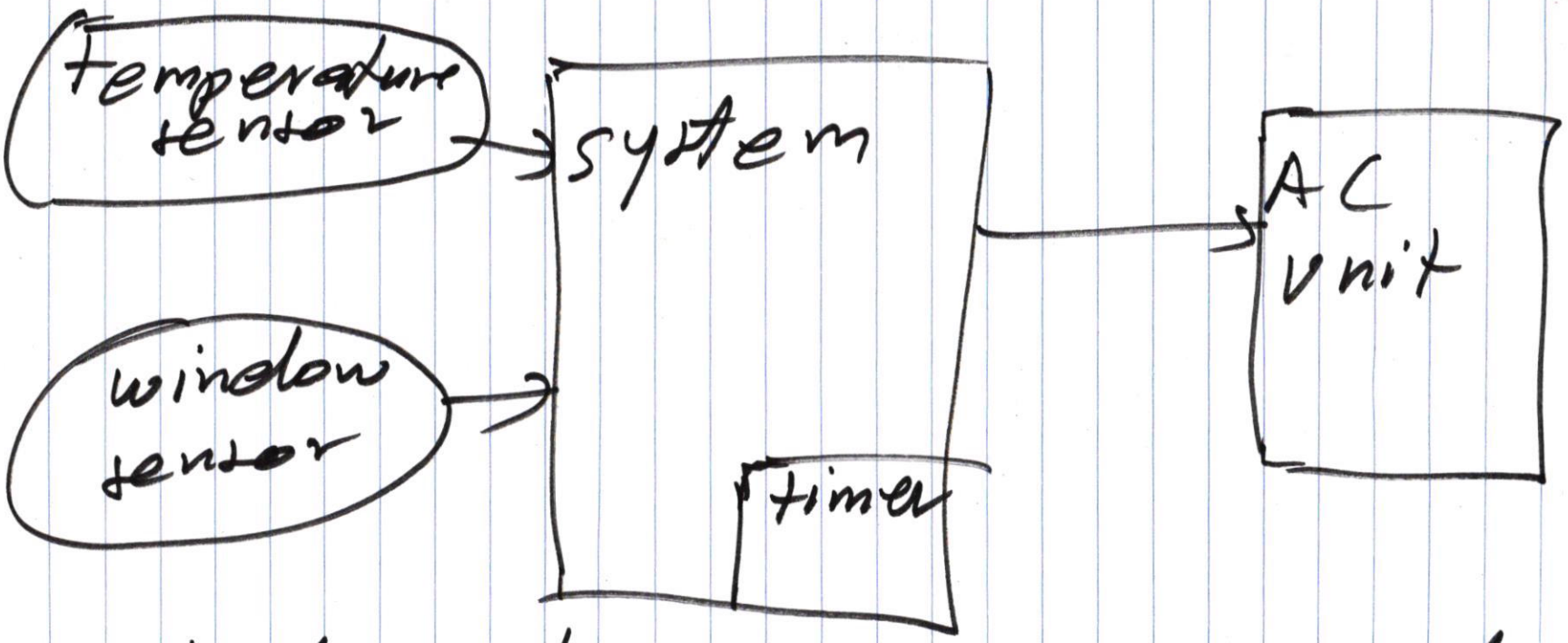



condition: Boolean expression
* virtual events (Boolean variables)
and, or, not

---

1. Model is in S1
2. virtual event e is invoked.
   e has now value "True"
3. if condition evaluates to "True"
   (e or e5)
4. transion from S1 to S2 and virtual action is invoked

# Air Conditioning System




## actual events

1. temperature reading
2. window closed
3. window open
4. time-out

## actual actions

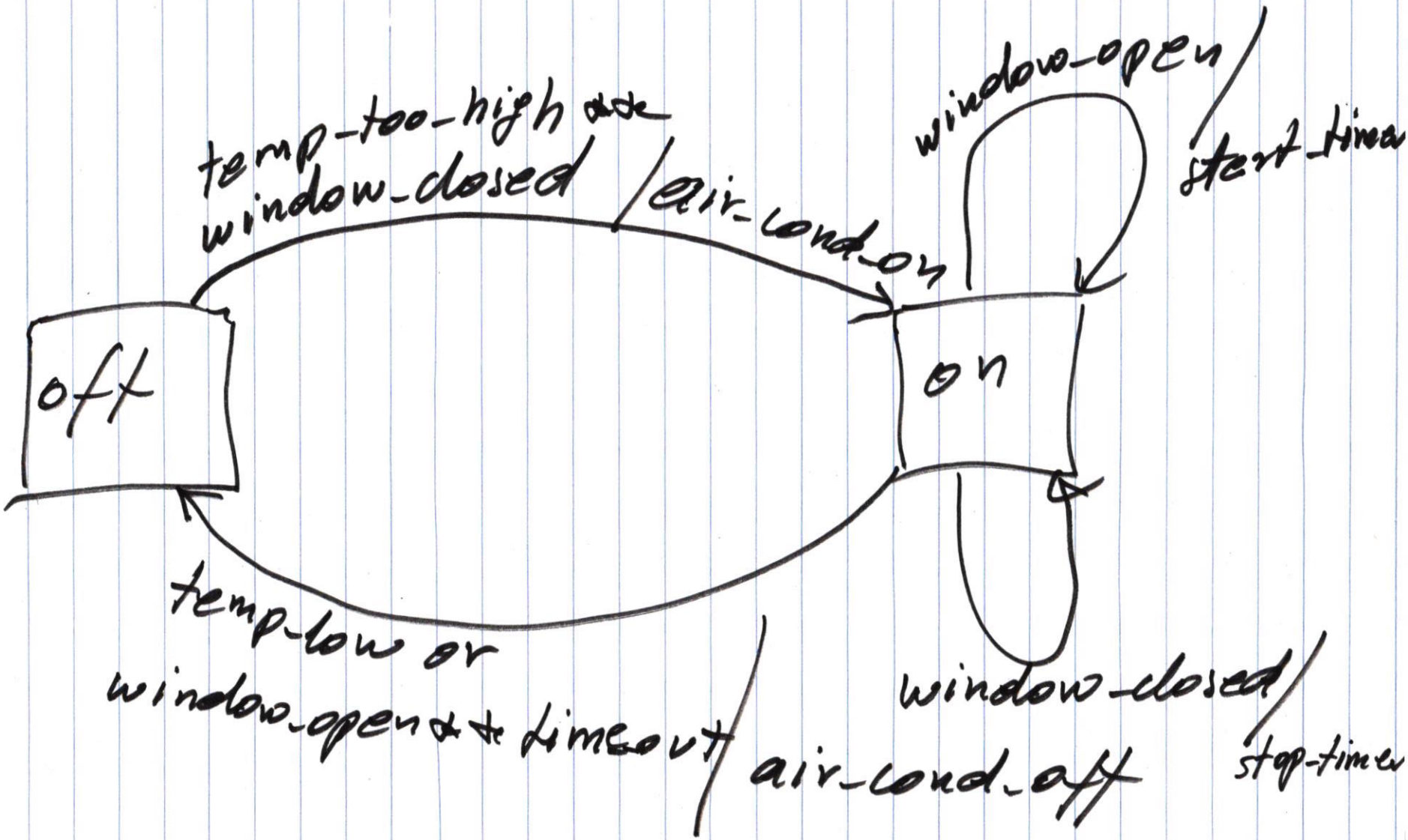
1. turn on AC
2. turn off AC
3. start timer
4. stop timer

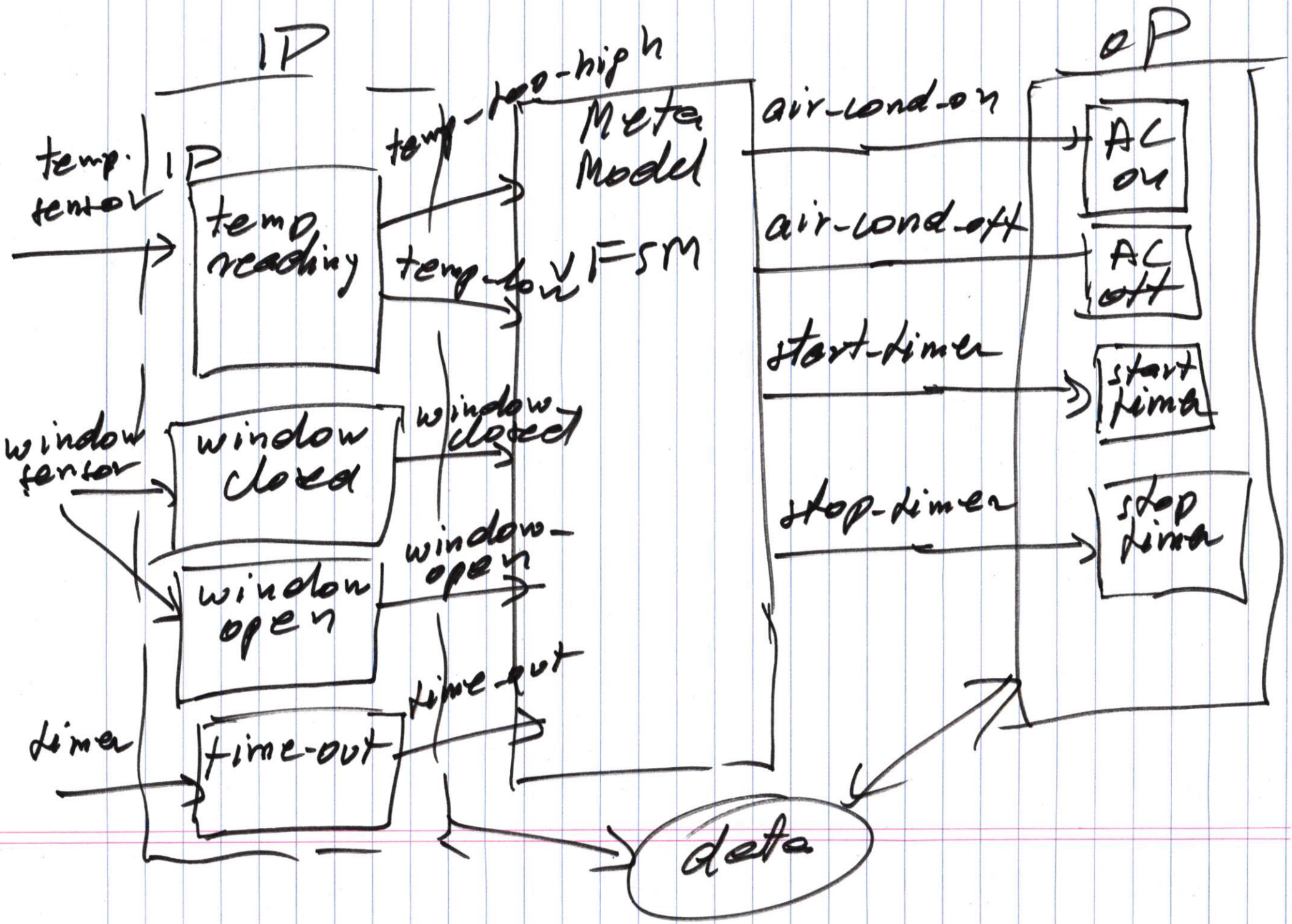
## virtual events

1. temp_too_high
2. window_closed
3. window_open
4. temp_low
5. time_out

## virtual action

1. air_cond_on
2. start_timer
3. stop_timer
4. air_cond_off


temp-too-high and
window_closed / air_cond_on
window_open /
start_timer
off
on
temp_low or
window_open & timeout /
air_cond_off
window_closed /
stop_timer


IP
temp. sensor
IP
temp reading
temp-too-high
temp-low
Meta Model
FSM
window sensor
window closed
window closed
window open
window-open
time
time-out
time-out
data
oP
air-cond-on
air-cond-off
start-timer
stop-timer
AC on
AC off
start timer
stop timer

# VFSM

advantages

no data

platform independent

disadvantages

rules of execution are complicated!!

# EFSM

* advantages.
  * very easy to understand
  * rules of execution are very simple.
* disadvantages.
  * containing data.
  * platform dependent events/actions.

We want to use

EFSM

but with

restrictions.

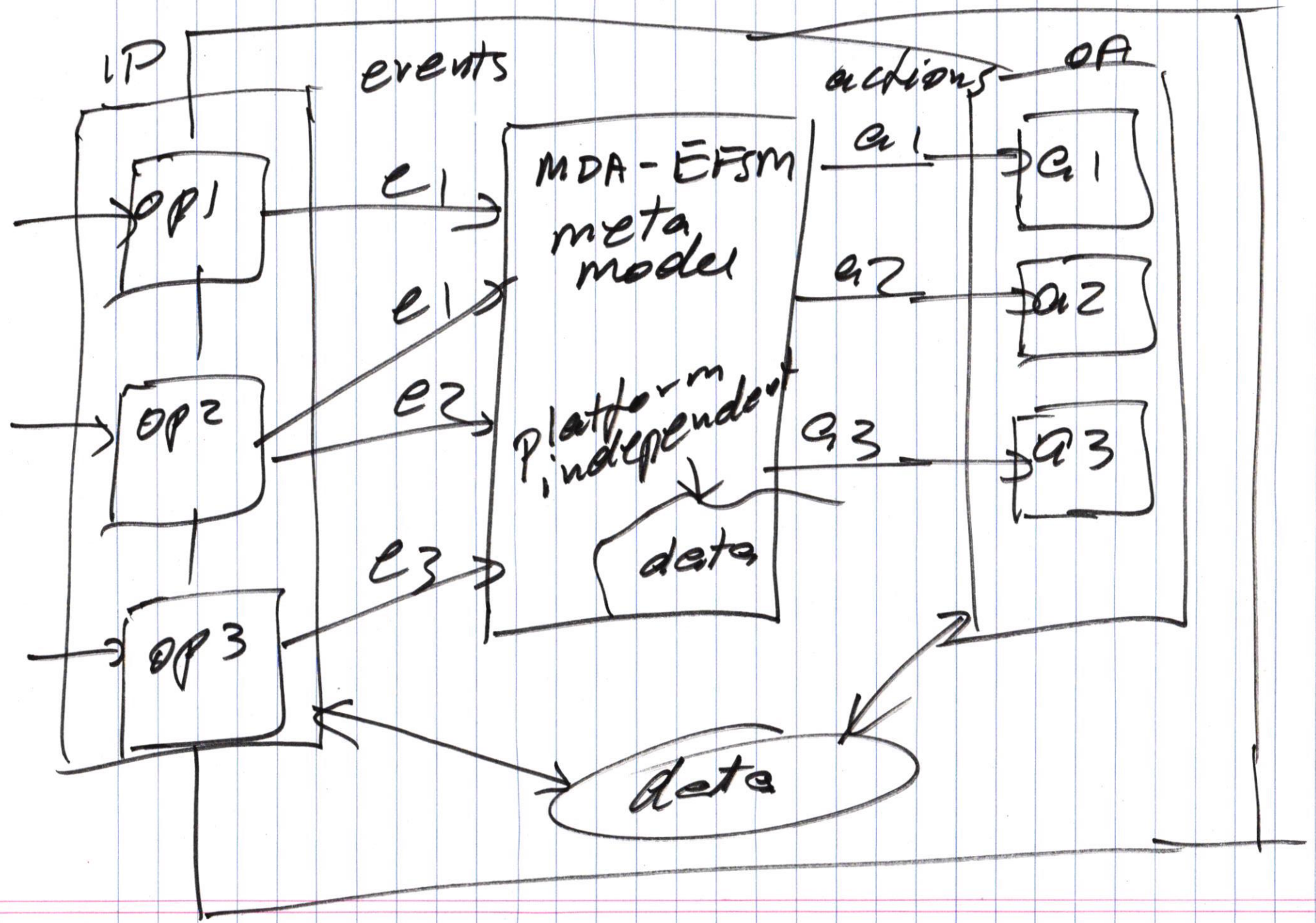
data, events, actions } must be platform independent

⇓

**MDA-EFSM**

(1) EFSM with no data or platform independent data

(2) platform independent events and actions.

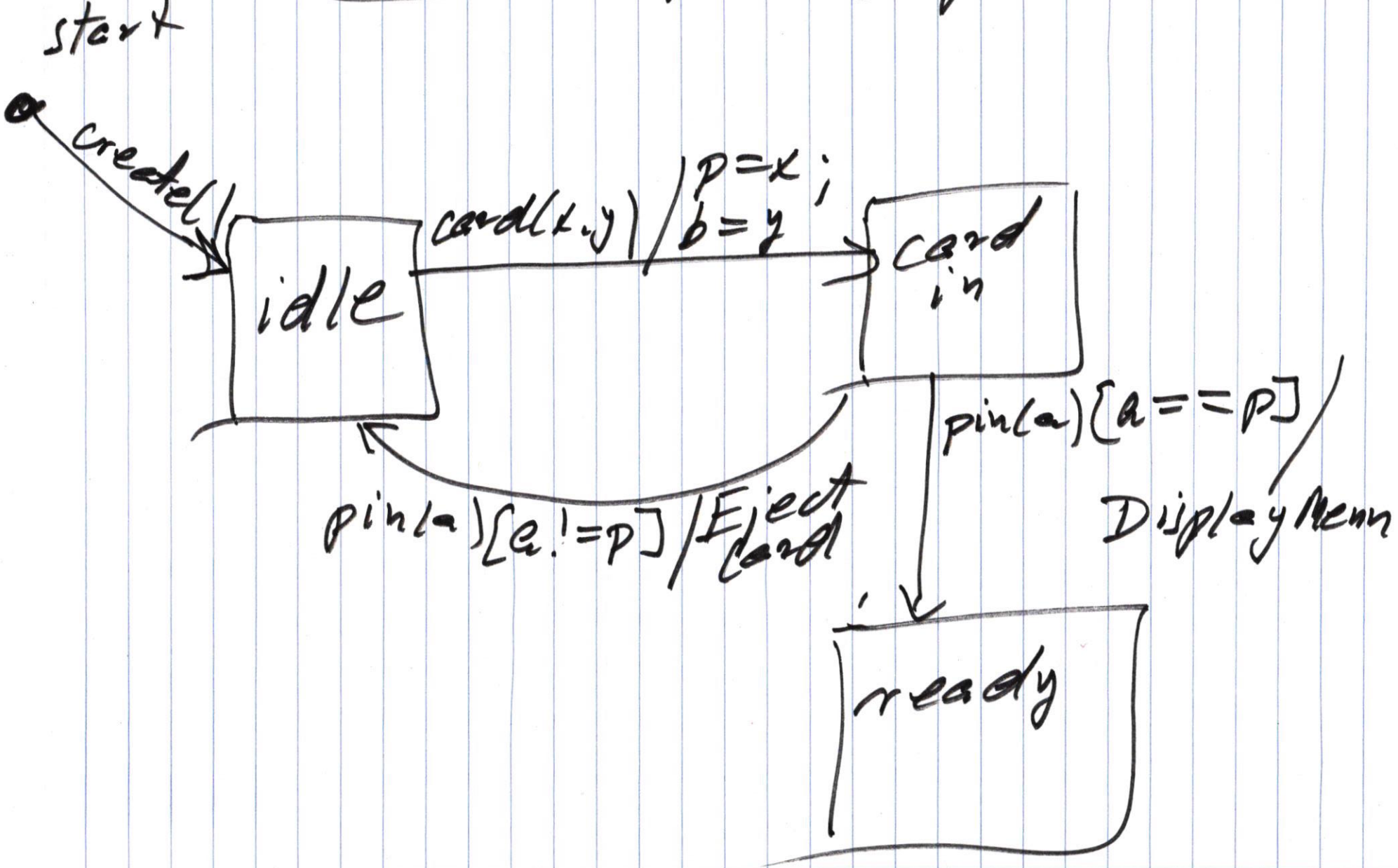

IP
events
actions
OA
MDA - EFSM
meta model
platform independent
data
op1
op2
op3
e1
e1
e2
e3
a1
a2
a3
a1
a2
a3
data

ATM-1 component

createI()
card(int x, int y)
pin(int a)

} actual events.

# EFSM – platform dependent



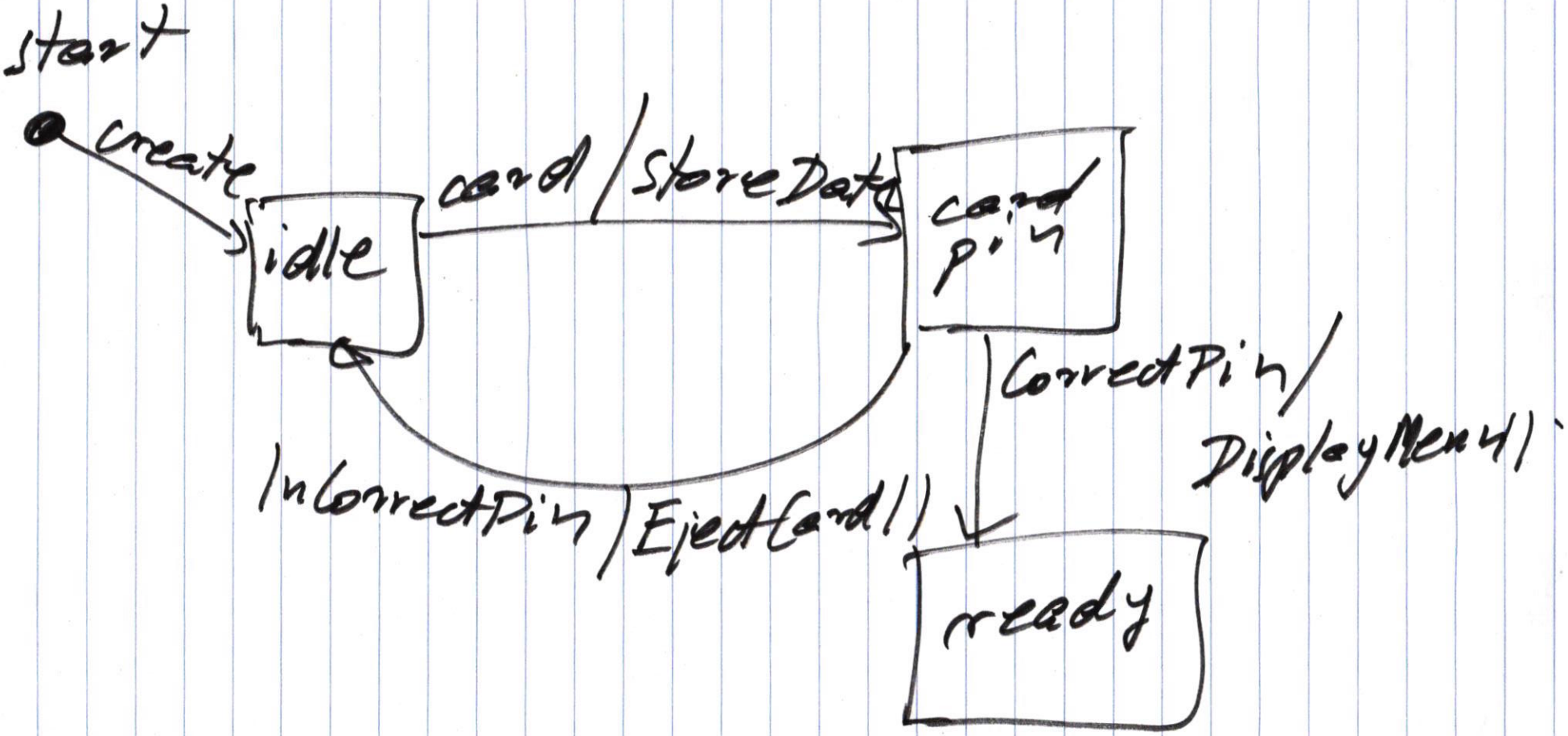
meta events

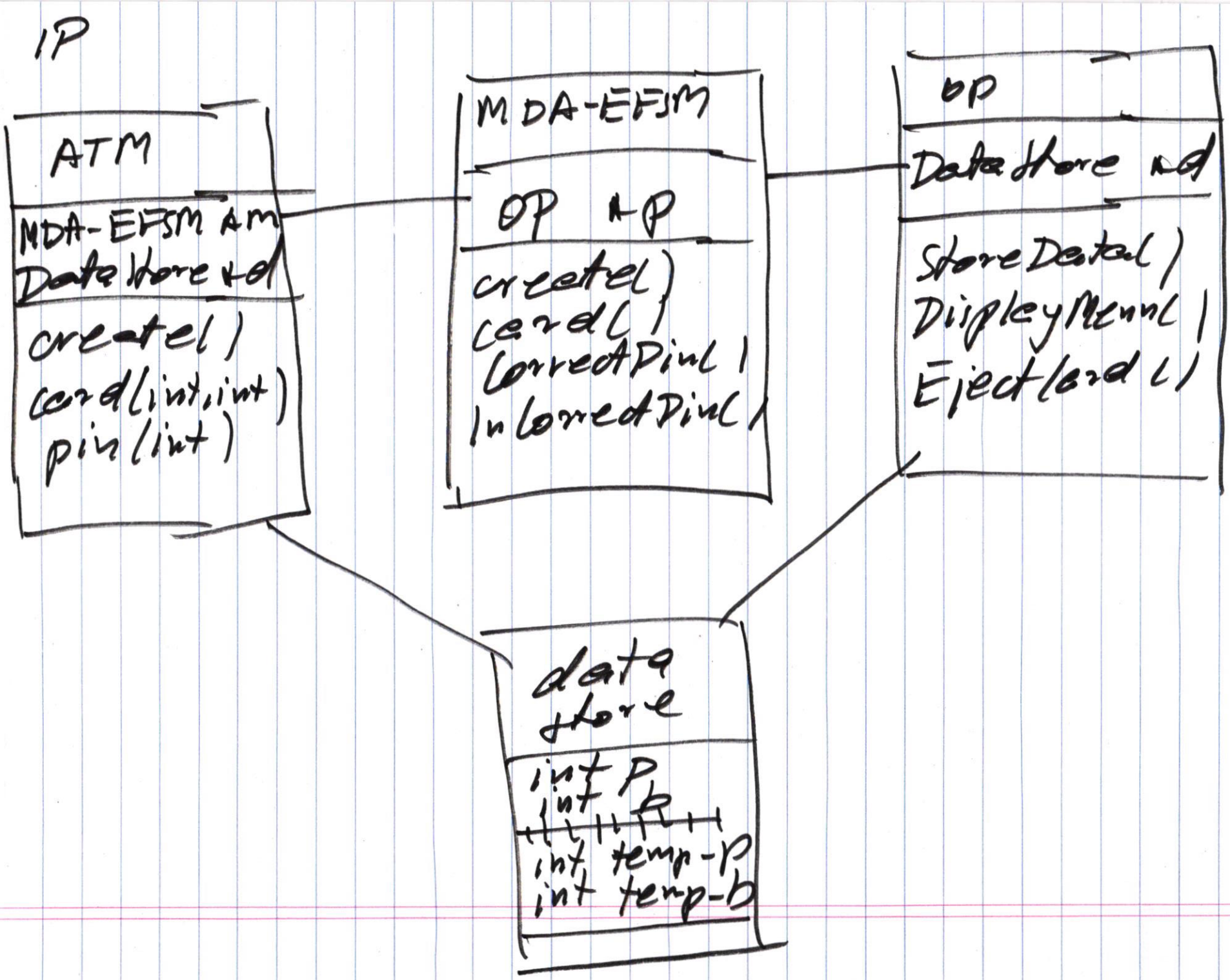
createCard()
cardIn()
CorrectPin()
IncorrectPin()

meta actions

StoreData()
DisplayMenu()
EjectCard()

# MDA-EFSM


start
create
idle
card / StoreData
card
pin
CorrectPin /
DisplayMenu()
IncorrectPin / EjectCard()
ready


IP
ATM
MDA-EFSM ATM
DataStore ref
create()
card(int,int)
pin(int)
MDA-EFSM
OP ATP
create()
card()
CorrectPin()
IncorrectPin()
OP
DataStore ref
StoreData()
DisplayMenu()
EjectCard()
data store
int P
int B
int temp-P
int temp-b

ATM class

```
create( ){
    m->create();
}
card (int x, int y)
{      d->temp_p = x;
       d->tem_b = y;
       m->card();
}

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

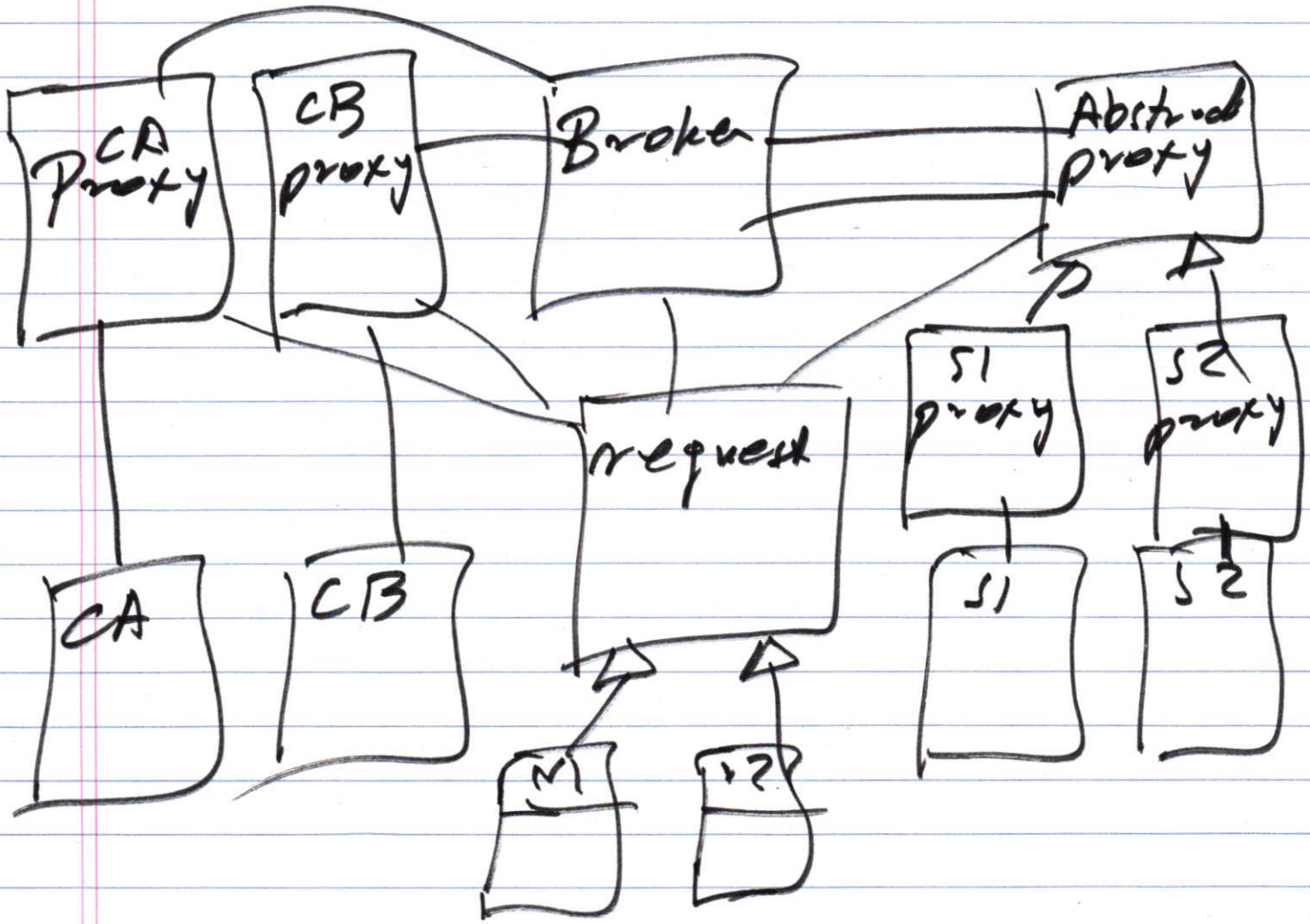
@P class

```
storeData()
{
    d->p = d->temp_p;
    d->b = d->temp_b
}
```

# Homework #2

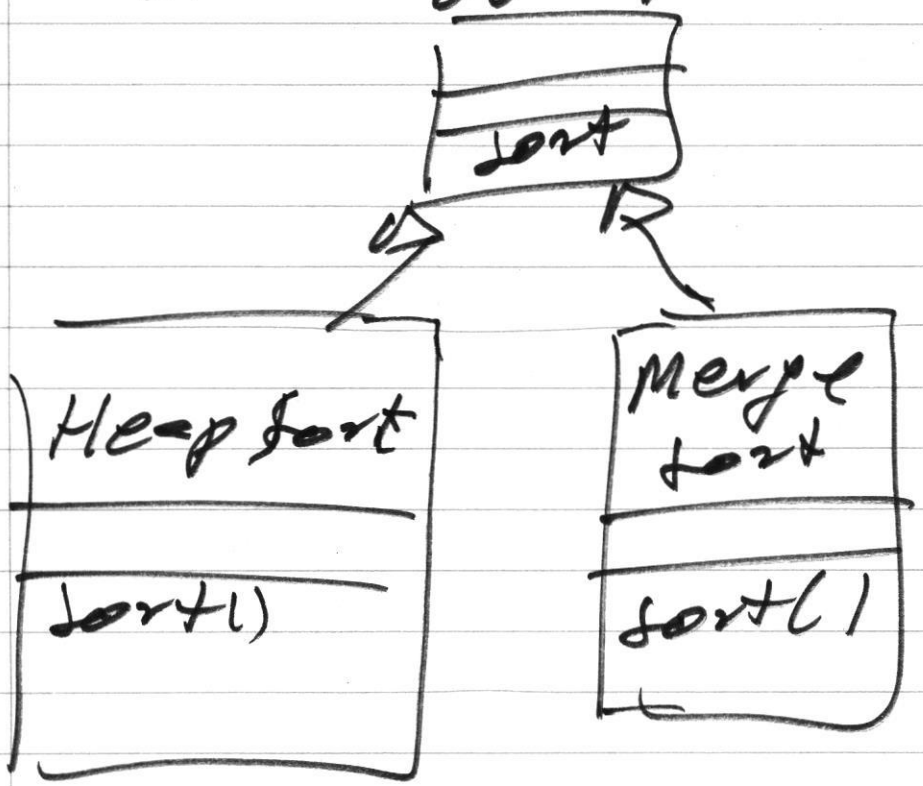
## Problem #1

Client-Broker-Server

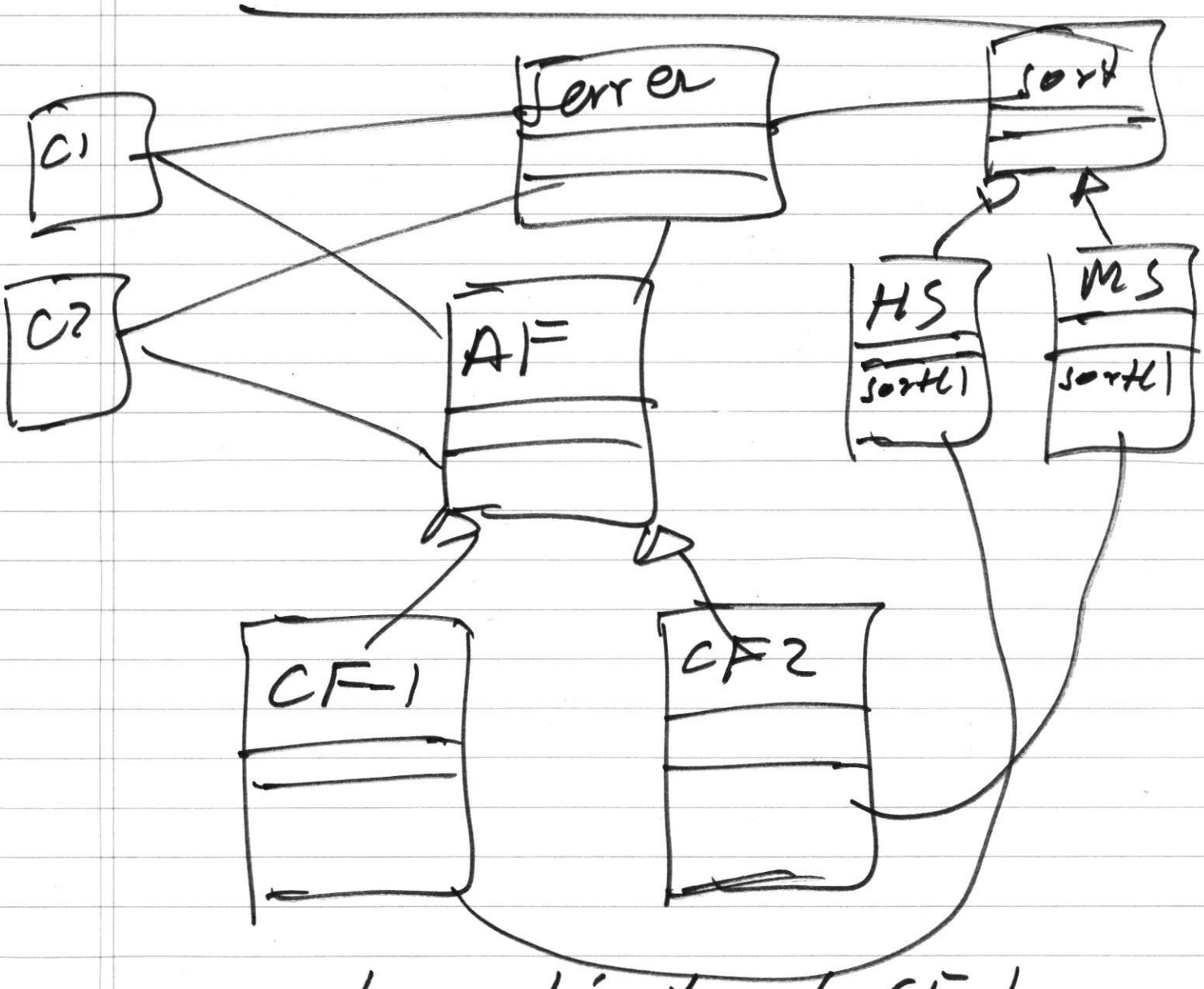


# Problem #3

Strategy pattern.



# abstract factory.




C1 creates object of CF-1
C1 passes CF-1 to server

# 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)

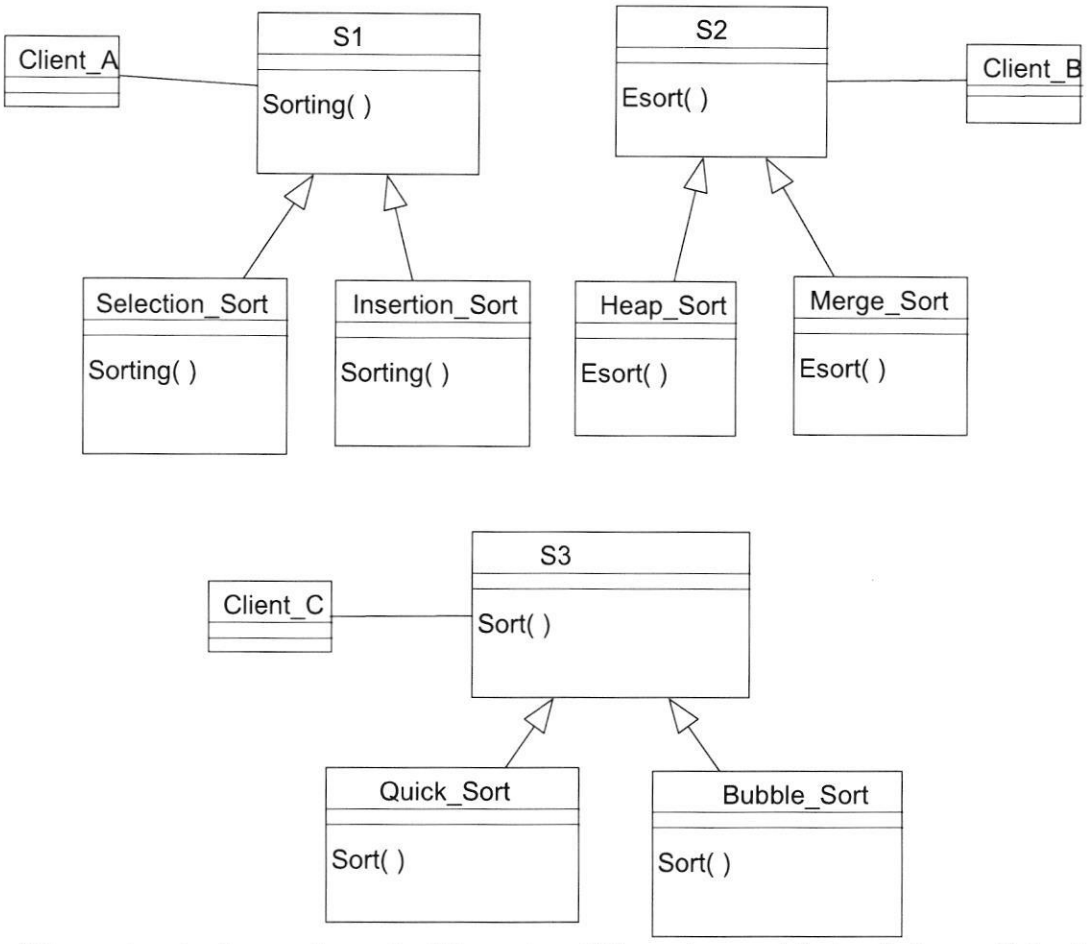
There exist two client processes and they request the following services:

| **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.

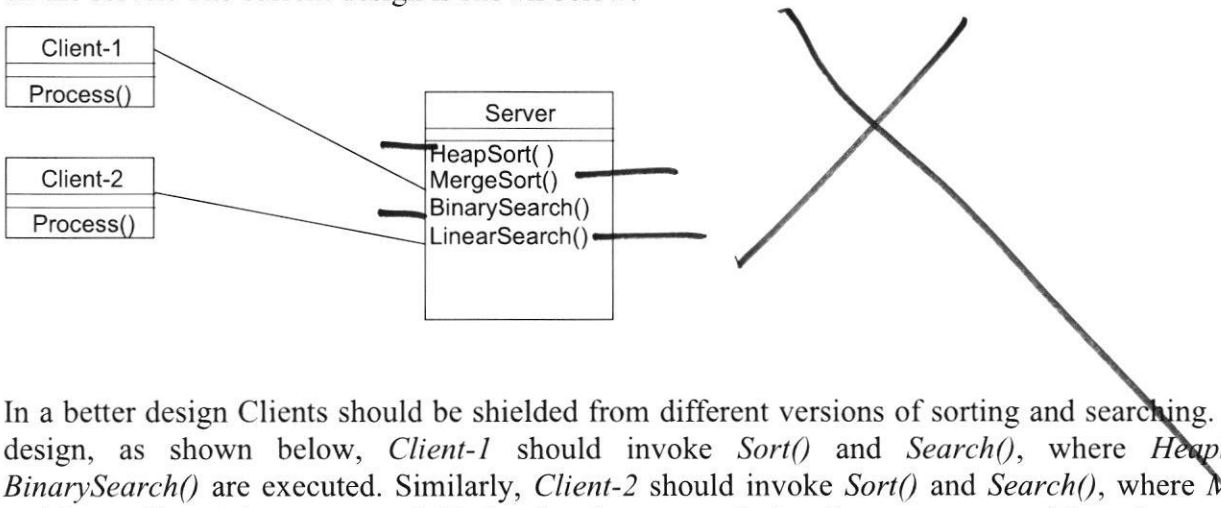
Provide two solutions that are based on:

1. An **association-based** version of the Adapter pattern
2. An **inheritance-based** version of the Adapter pattern

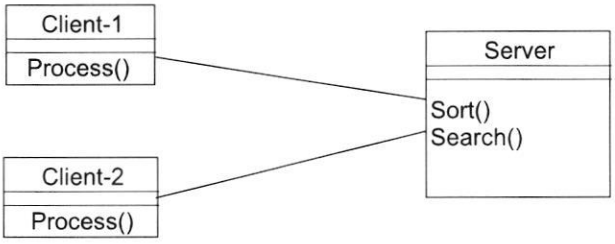
Provide a class diagram for each solution. You do not have to provide any description for classes/operations of the above class diagram (only new classes/operations should be described using **pseudo-code**).

**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*.