Homework #2 is posted

Model Driven Architecture

MDA

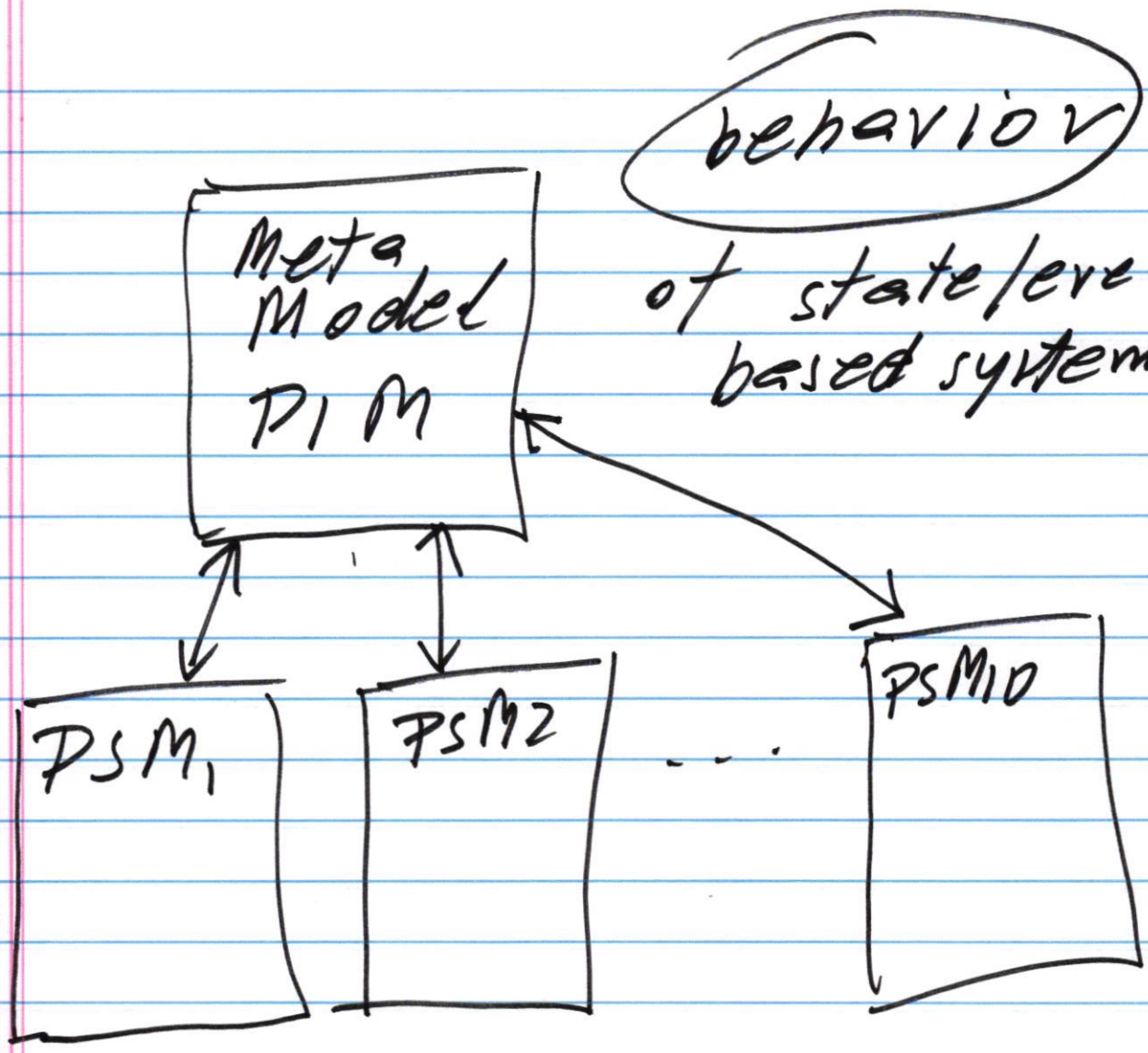
- multi-year version architecture.
- We want to seperate
  - ✖ platform independent aspects of a family of systems

FROM

  - ✔ platform specific aspects of the systems.

Platform

- OS
- DB
- network
- hardware
- data structures / types
- algorithms
- . . .

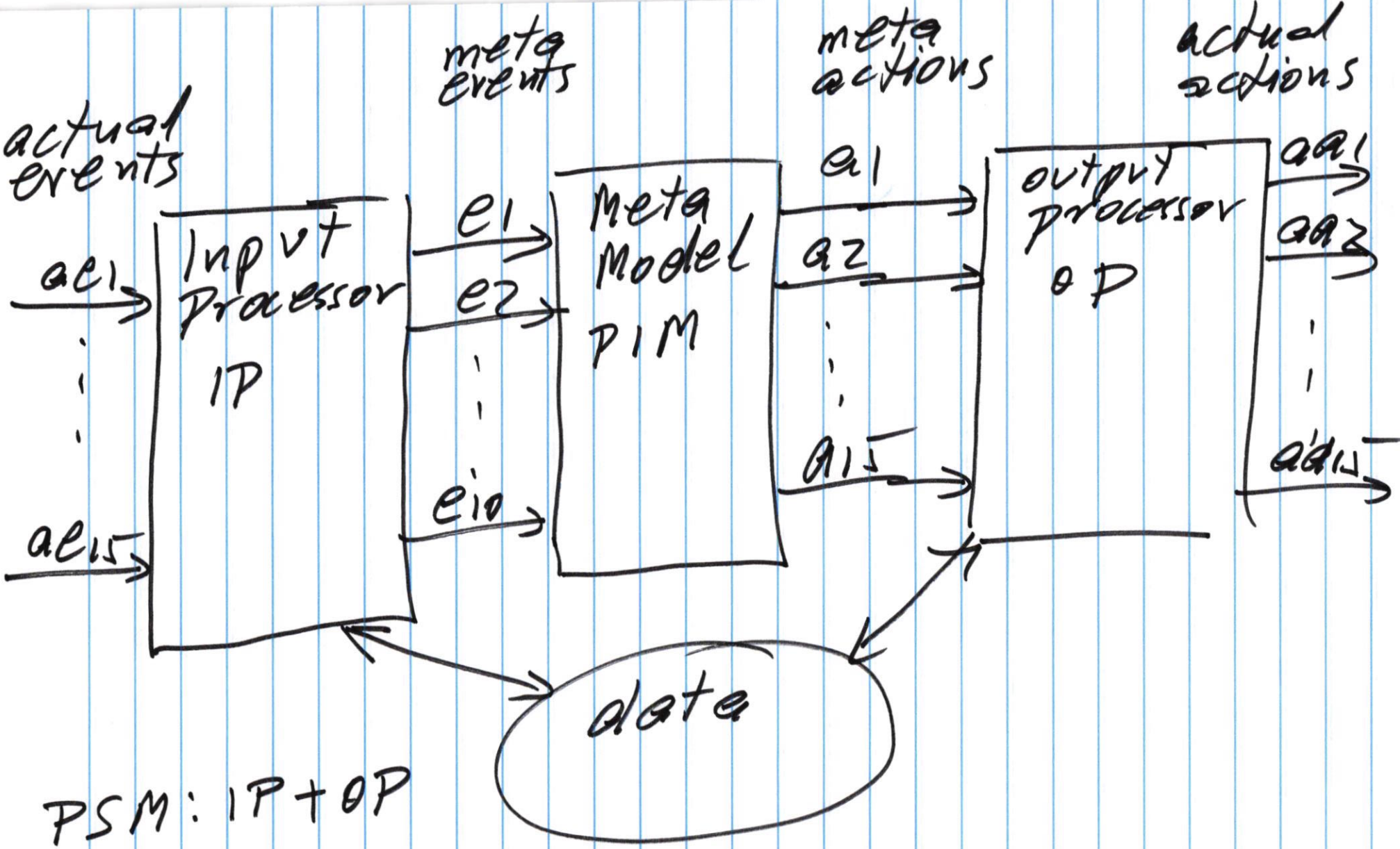

behavior
Meta Model PIM
of state/event based systems.
PSM1
PSM2
PSM10

MDA

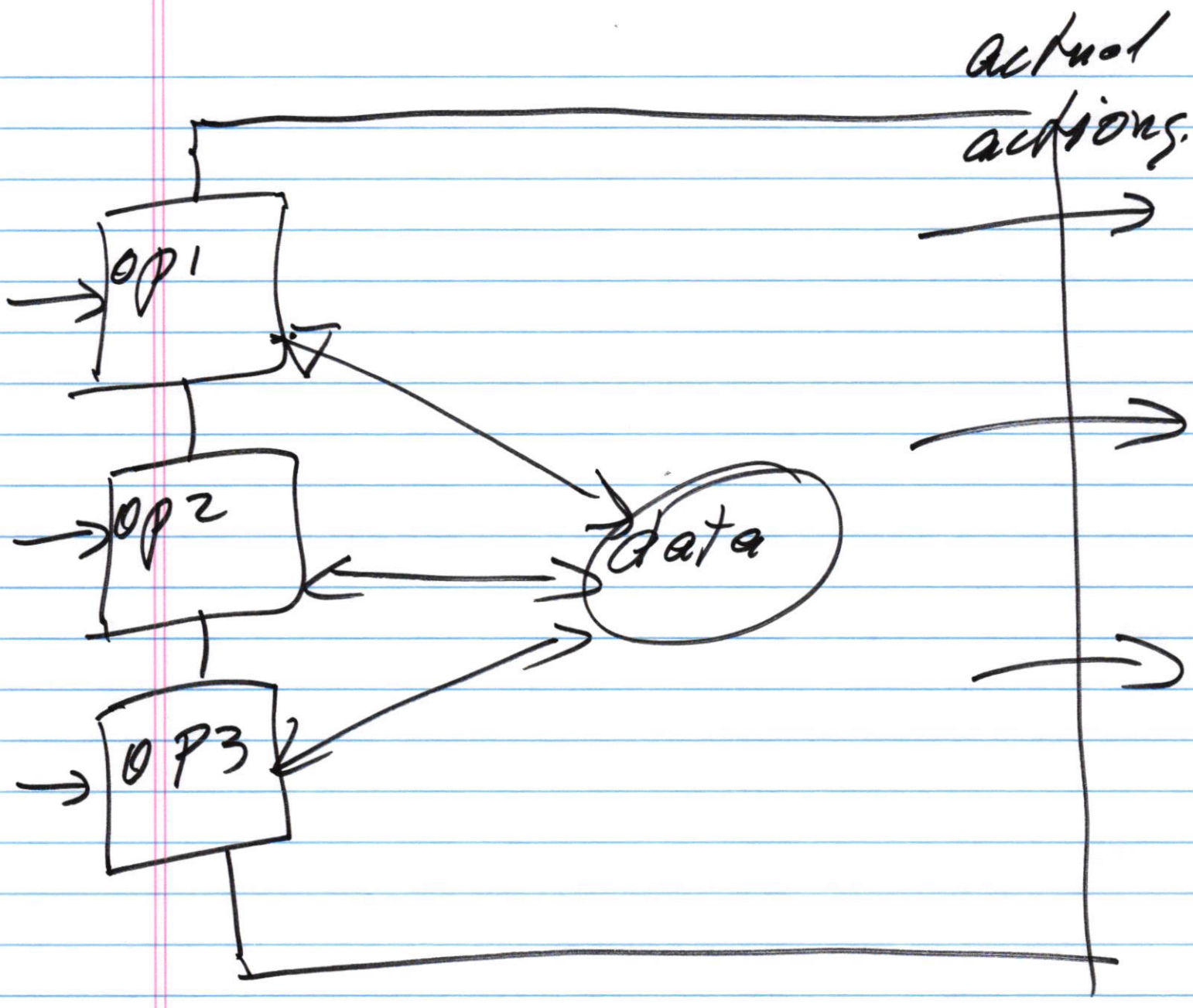
PIM: Platform Independent Model

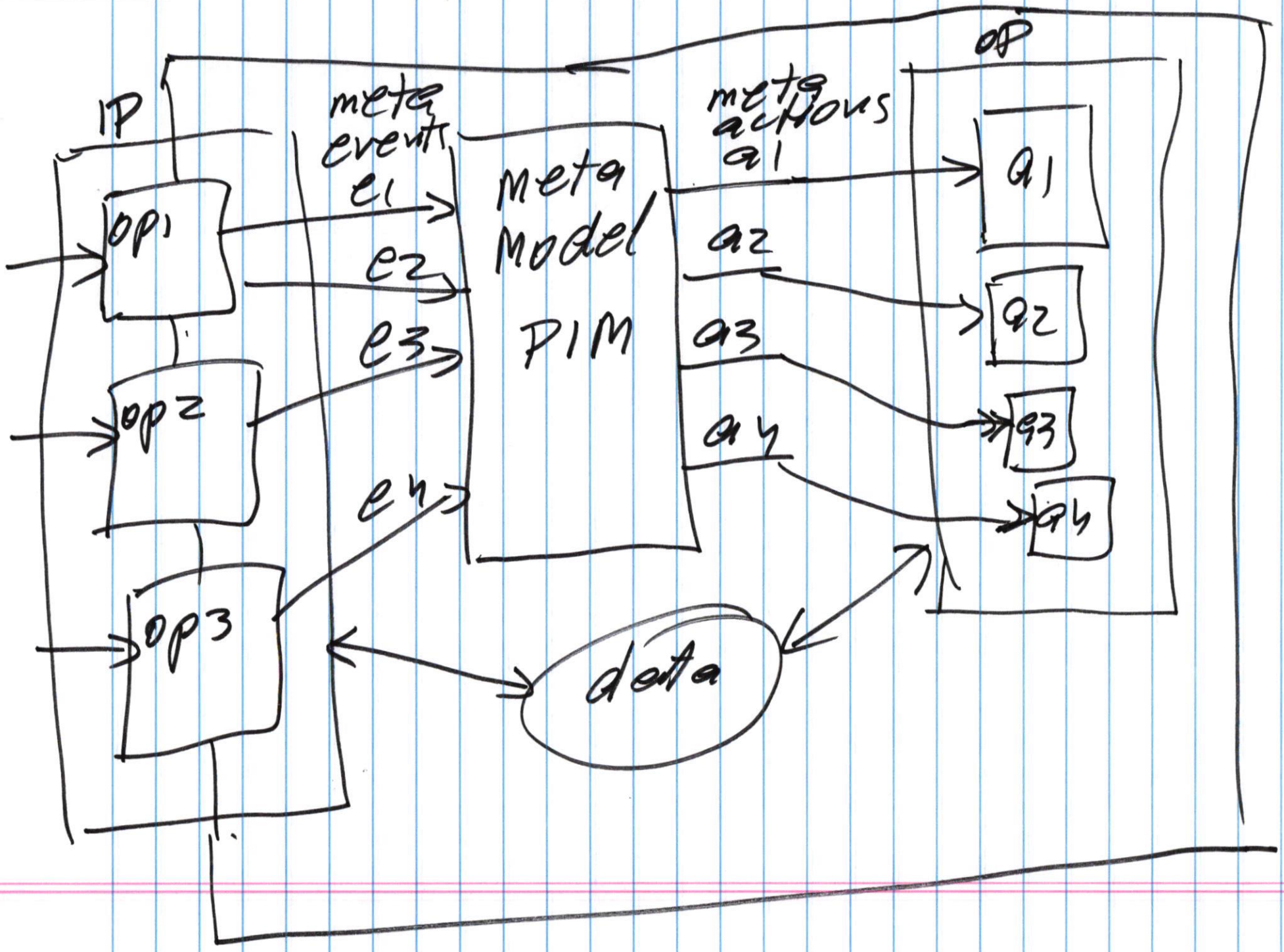
Model will be "stable"

PSM: Platform Specific Model


actual events
meta events
meta actions
actual actions
ae1
ae15
Input Processor
IP
e1
e2
e10
Meta Model
PIM
a1
a2
a15
output processor
OP
aa1
aa2
aa15
data

PSM: IP + OP


actual
actions
op1
op2
op3
data


IP
op1
op2
op3
meta events
e1
e2
e3
e4
Meta Model PIM
meta actions
a1
a2
a3
a4
OP
q1
q2
q3
q4
data

event/state based systems

Meta model that should be platform independent

⇓

meta behavior.

Modeling Languages

- 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

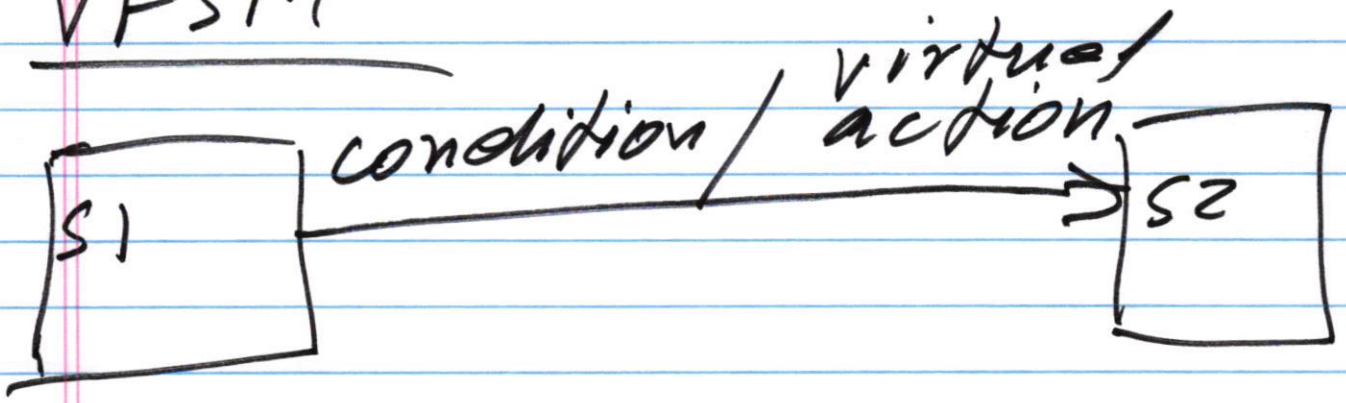
ATM component

EFSM

actual events { card(int pin, int bal)
card(float bal, string pin, string vid)
.
.
.

VFSM: card

VFSM

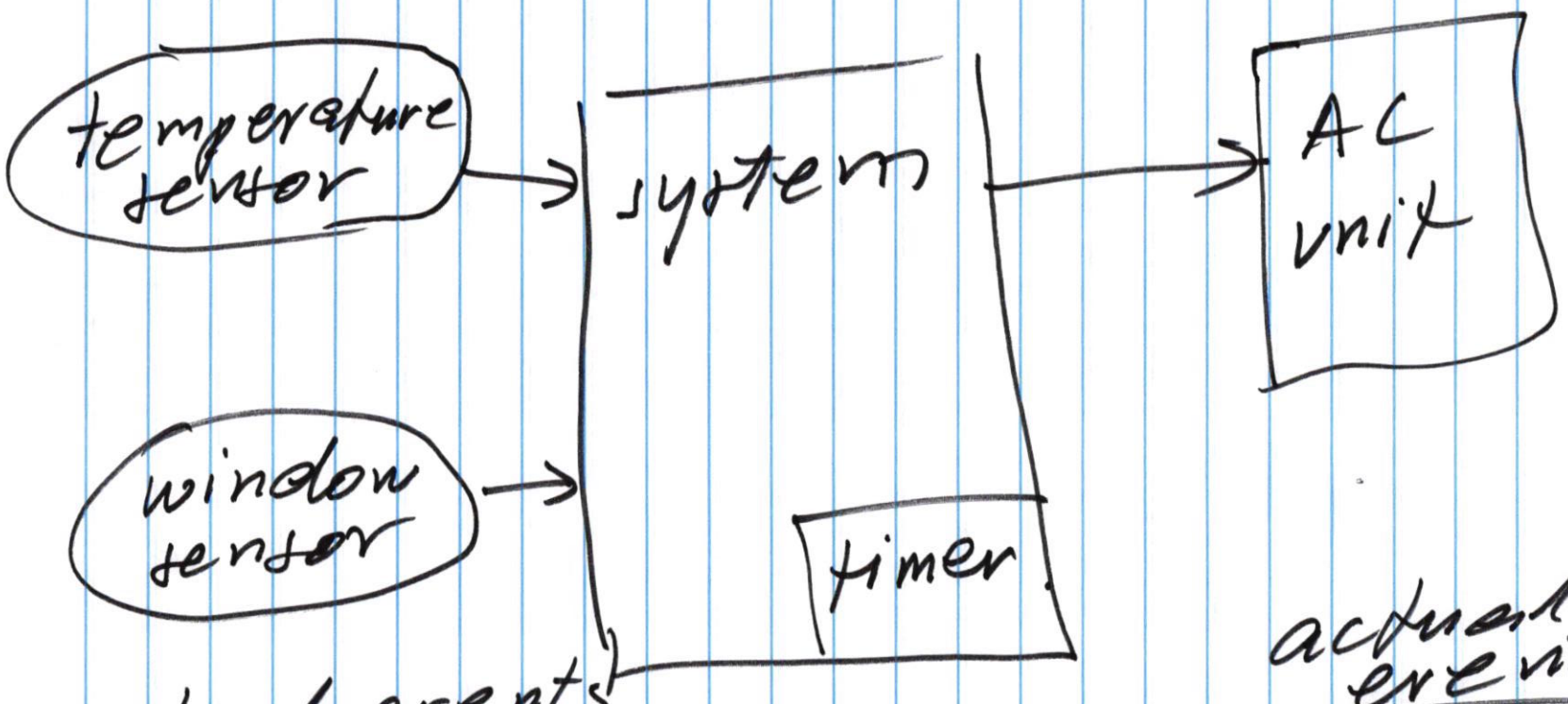


condition: Boolean expression consists of virtual events.

and, or, not

1. Model is in S1
2. virtual event e is invoked
   e = True
3. if condition evaluates to True
   (e or eS)
4. Transition from S1 to S2
   virtual action is invoked

Air Conditioning System



actual events

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

actual events

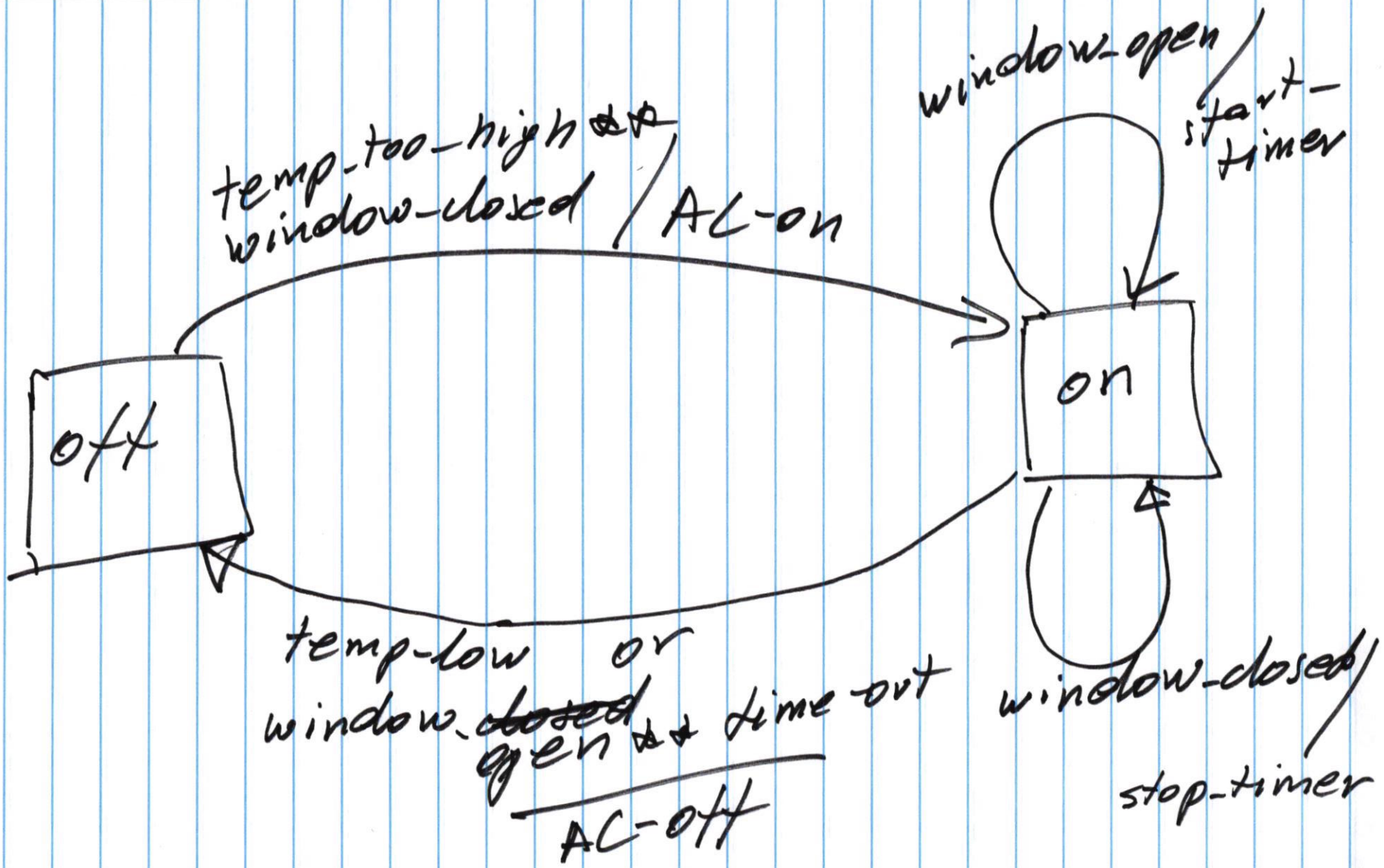
1. turn on AC
2. turn off AC
3. start timer
4. stop timer

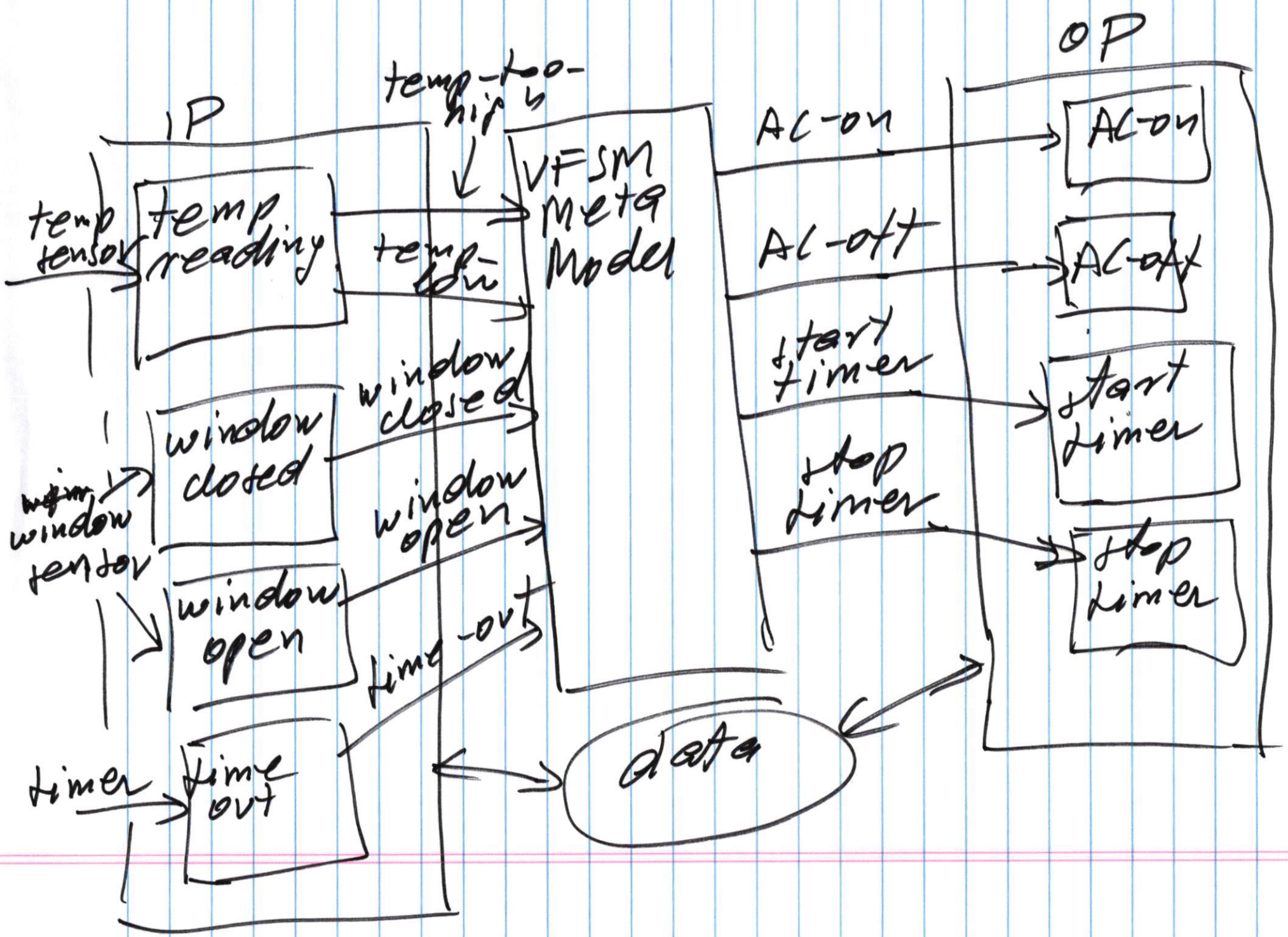
virtual events

1. temp_too_high
2. window_closed
3. window_open
4. temp_low
5. time-out

virtual actions

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


window_open / start_timer
temp_too_high ** / window_closed / AC_on
off
on
temp_low or window_closed ** time_out / AC_off
window_closed / stop_timer


IP
temp-too-high
OP
temp tensor
temp reading
temp-too-high
VFSM Meta Model
AC-on
AC-off
start timer
stop timer
AC-on
AC-off
start timer
stop timer
window closed
window open
window tensor
window closed
window open
time-out
timer
time out
data

VFSM

advantages

no data

platform independent

disadvantages

rules of execution are very complex.

EFSM

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

We want to use
EFSM

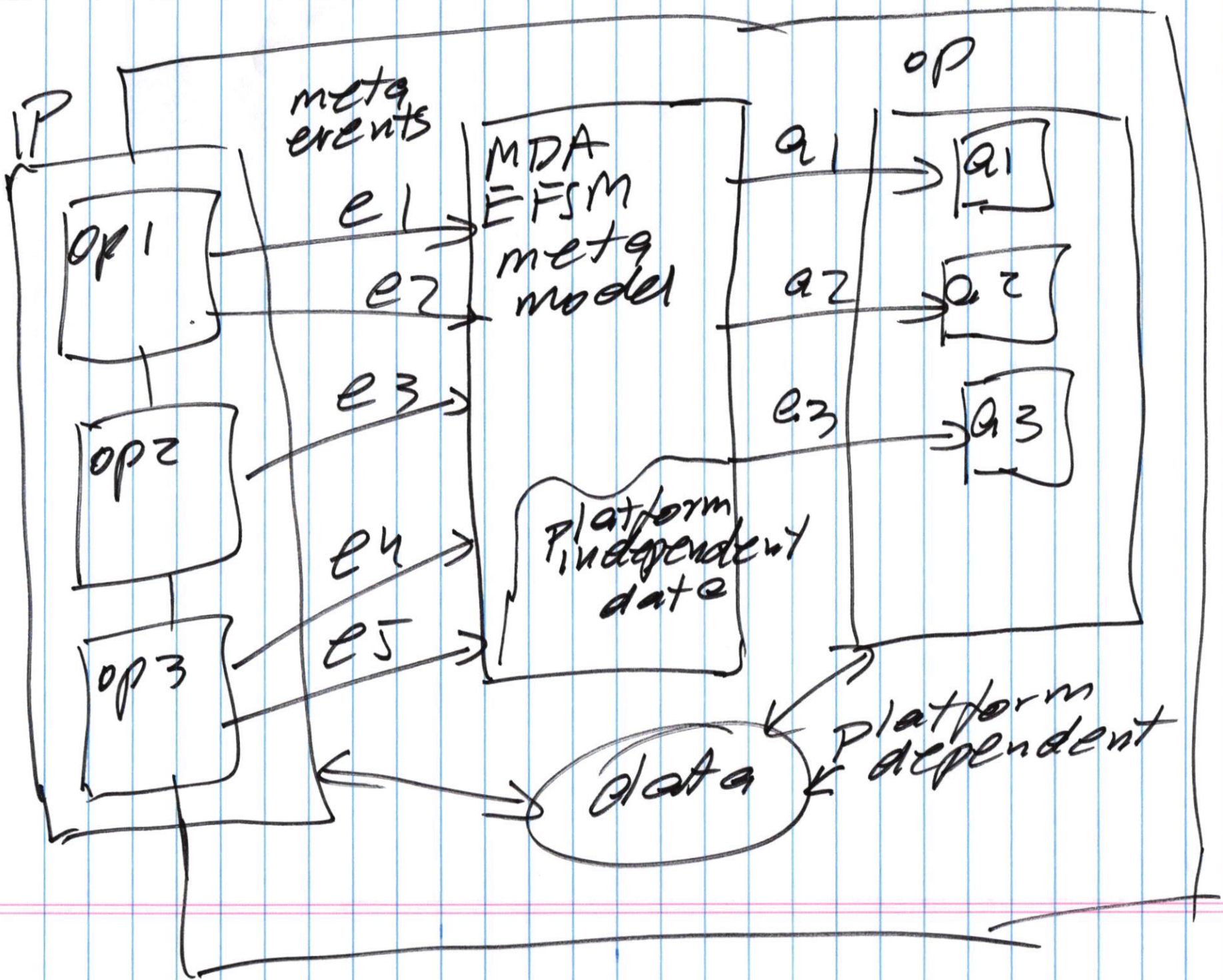
but with restrictions

data
events
actions
} must be platform independent

MDA - EFSM

(1) EFSM with no data
or
platform independent data

(2) platform independent events and actions

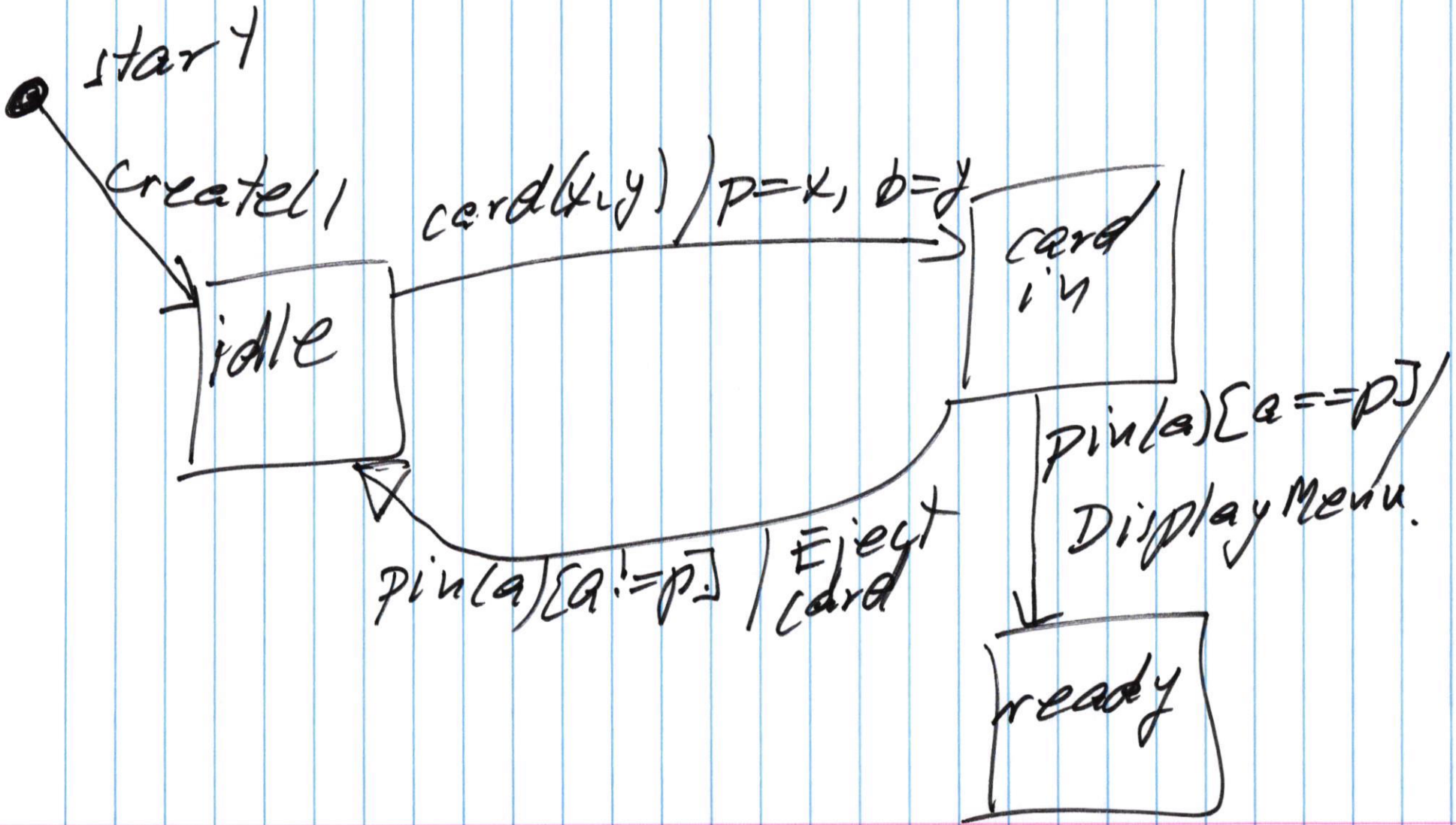

IP
meta events
op
MDA EFSM meta model
op1
op2
op3
e1
e2
e3
e4
e5
q1
q2
q3
Q1
Q2
Q3
Platform Pindependent data
data
Platform dependent

ATM-1 component

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

} actual events.

Platform dependent FSM



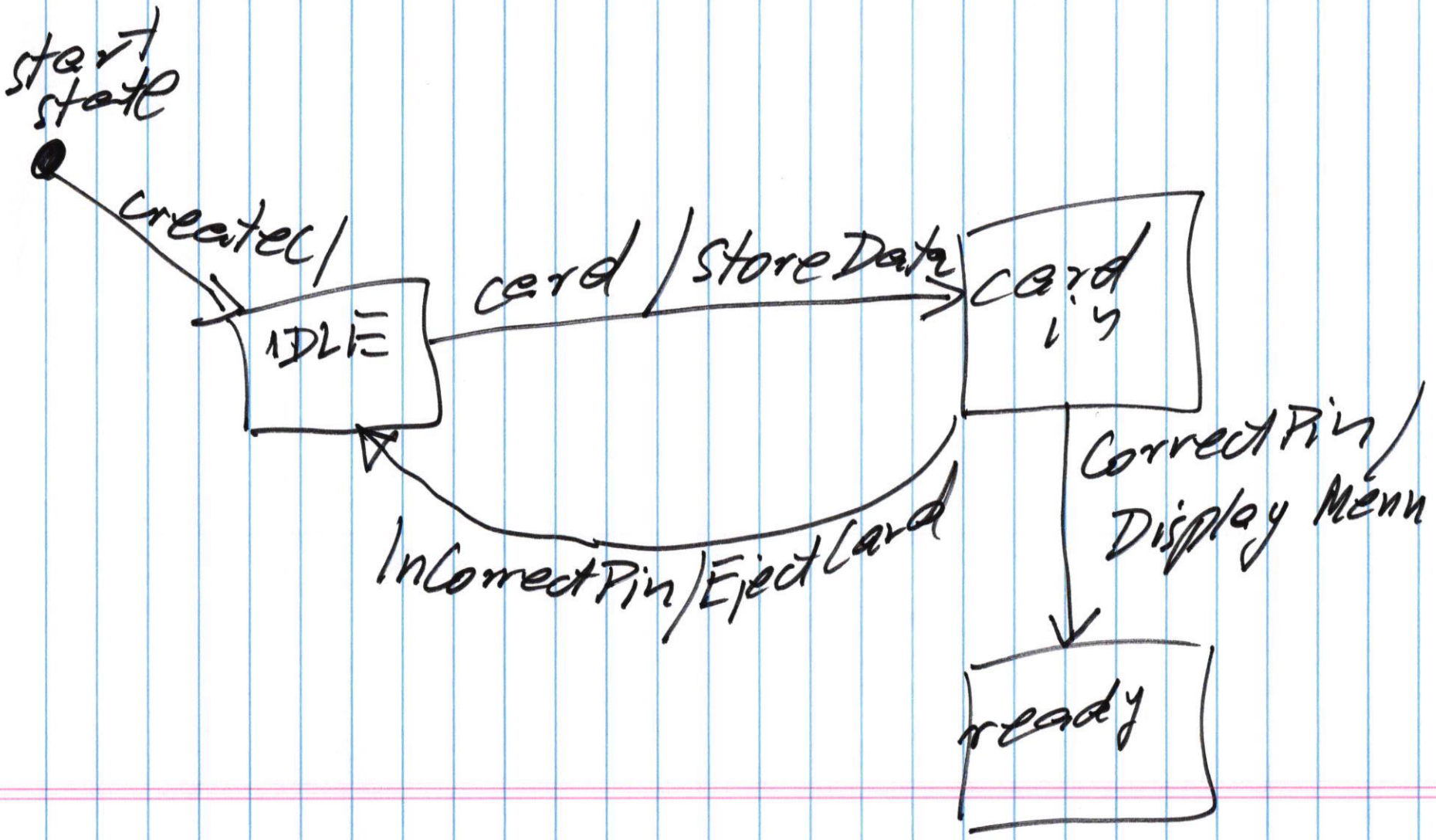
meta events

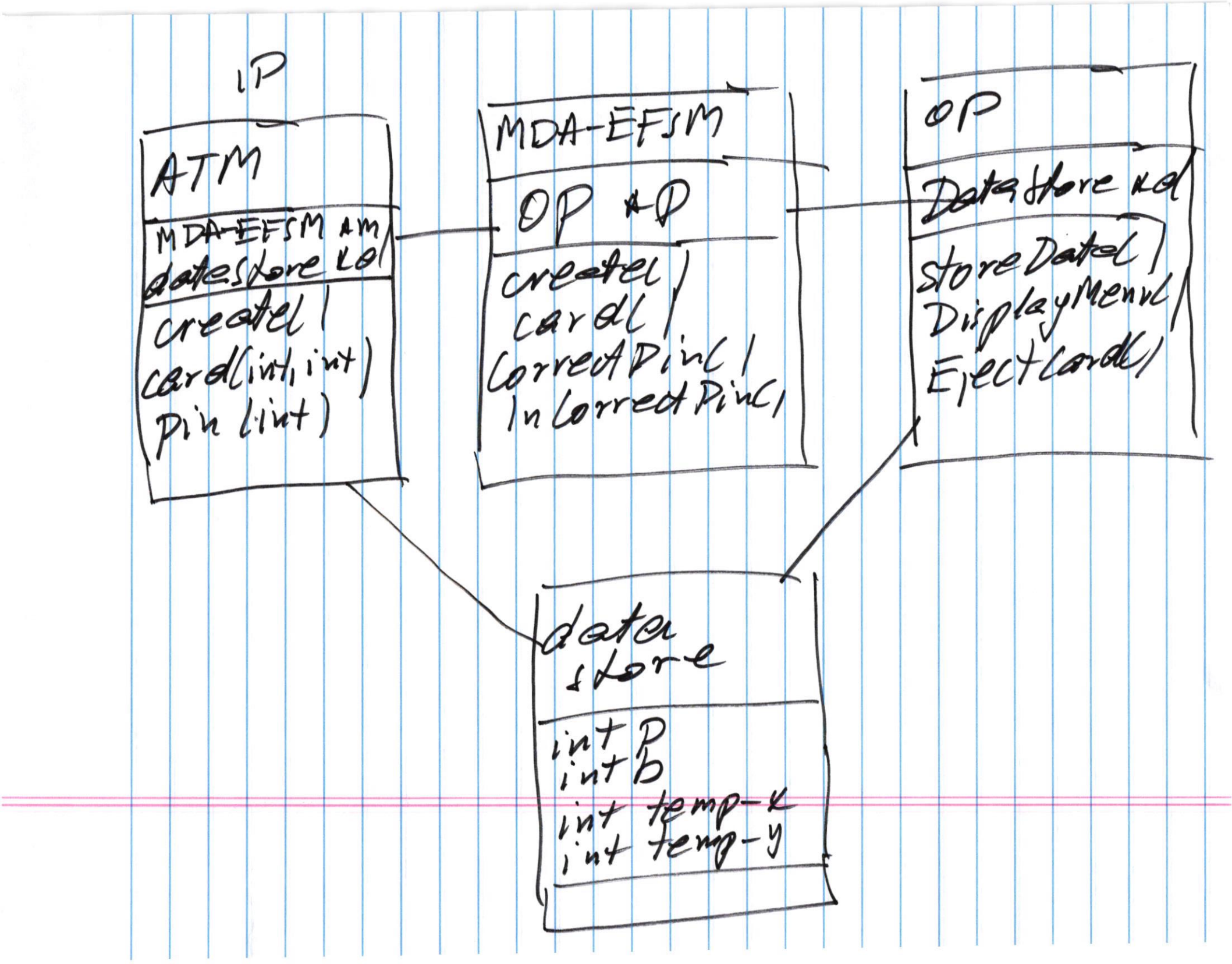
createC)
cardC)
CorrectPinC)
IncorrectPinC)

meta actions

StoreData
EjectCard
DisplayMenu

MDA-EFSM


start state
created/
IDLE
card / StoreData
card in
CorrectPin/
Display Menu
InCorrectPin/EjectCard
ready


UP
ATM
MDA-EFSM am
dataStore kd
createl()
cardl(int, int)
Pin(int)
MDA-EFSM
OP *p
createl()
cardl()
CorrectPinl()
InCorrectPinl()
OP
DataStore kd
StoreDatel()
DisplayMenul()
Ejectcardl()
data
store
int P
int b
int temp-x
int temp-y

ATM class

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

```
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()
}
```

class OP

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

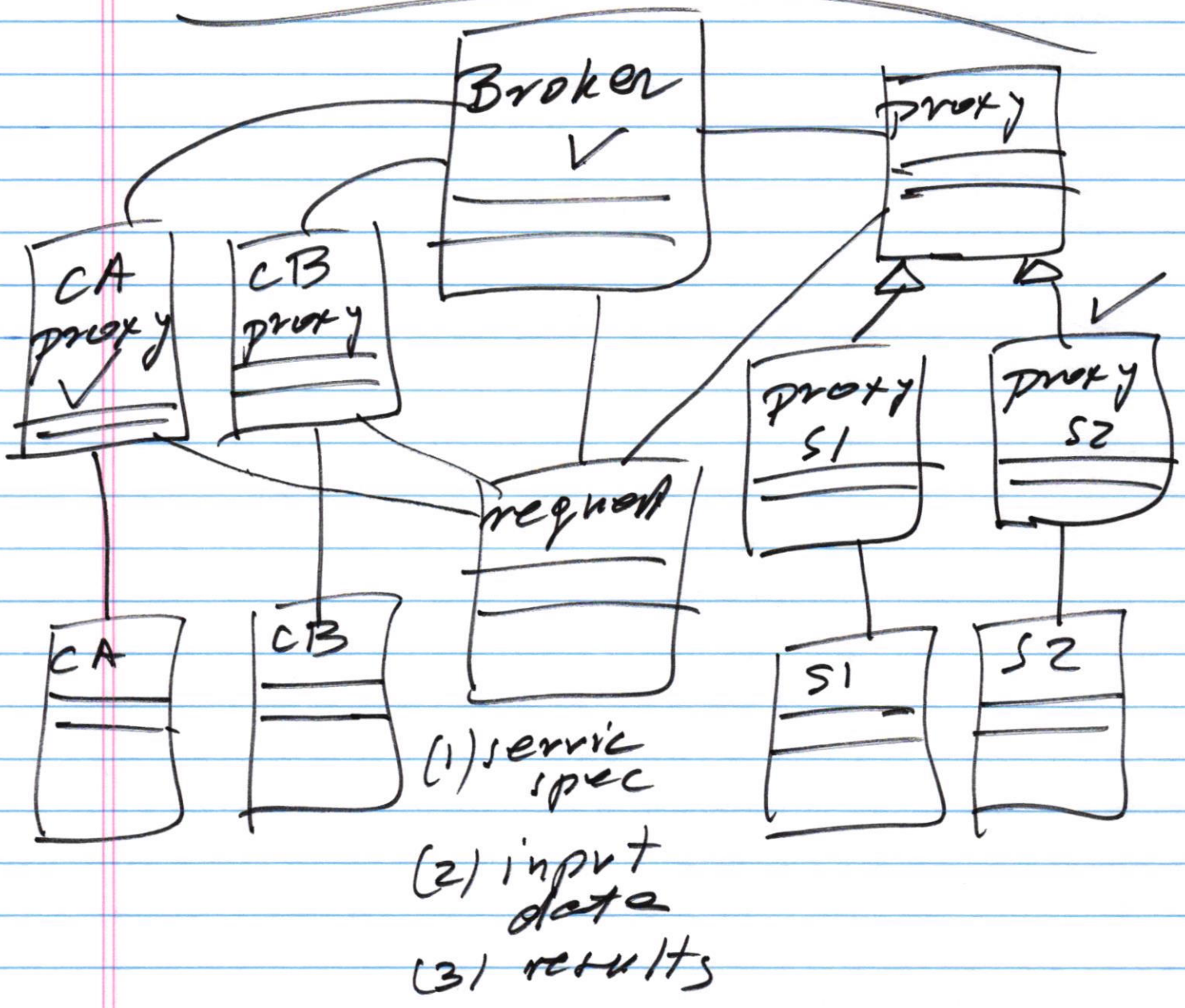
```
DisplayMenu()
{ display menu
     implementation
}
```

```
EjectCard()
{ implementation
     of EjectCard
}.
```

Homework #2

Problem #1

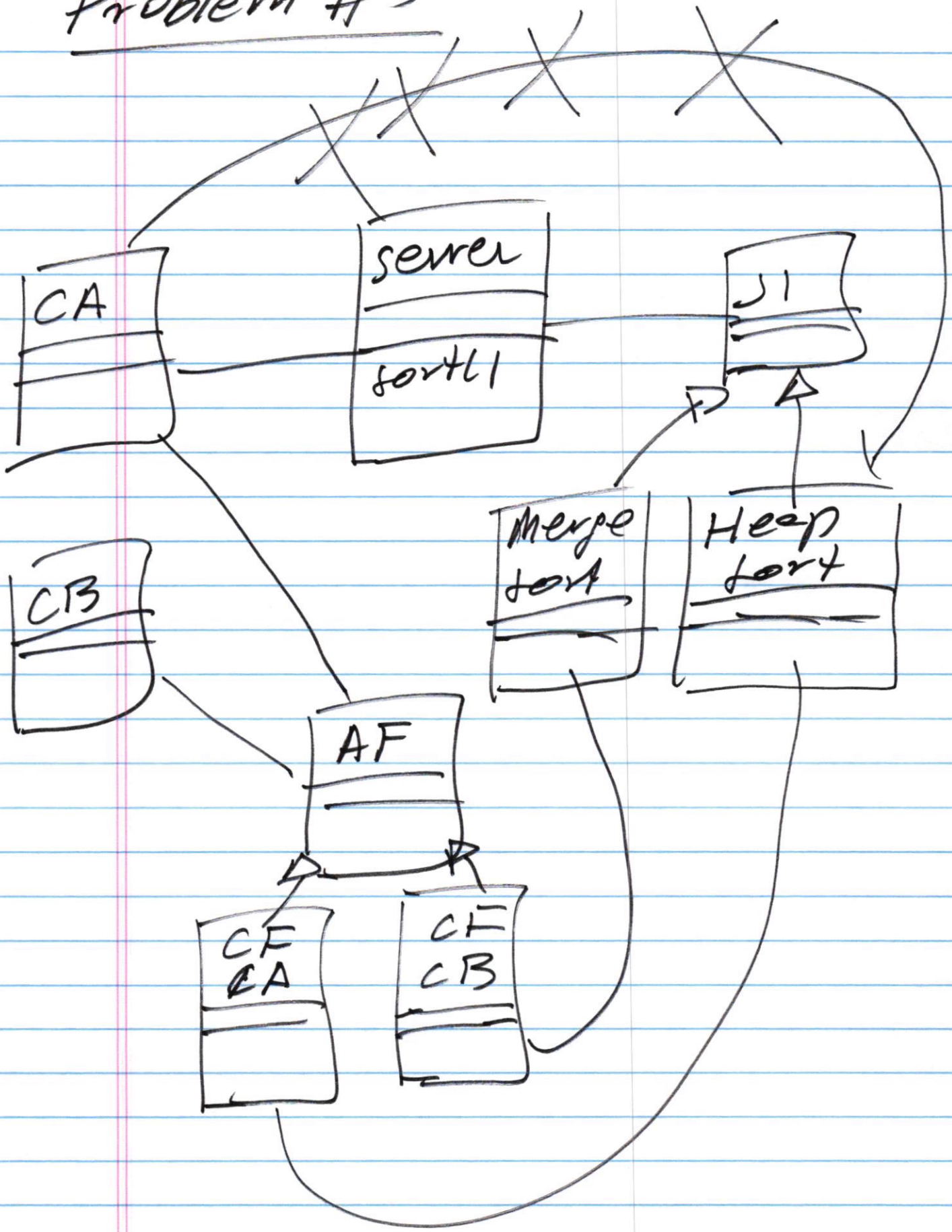
Client - Broker - Server



Problem #2

Adapter pattern.

Problem #3



HOMEWORK ASSIGNMENT #2

CS 586; Fall 2025
Due Date: **October 16, 2025**
Late homework: 50% off
After **October 21,** 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 (40 points)
There exist two servers **S1** and **S2**. Both servers support the following services:

Services supported by **server-S1**:
void Service1(string, int, int)
void Service2(string, int, int)
int Service3(string)
float Service4(string)

Services supported by **server-S2**:
void Service1(string, int)
void Service2(string, int)
int Service3(string)
float Service4(string)

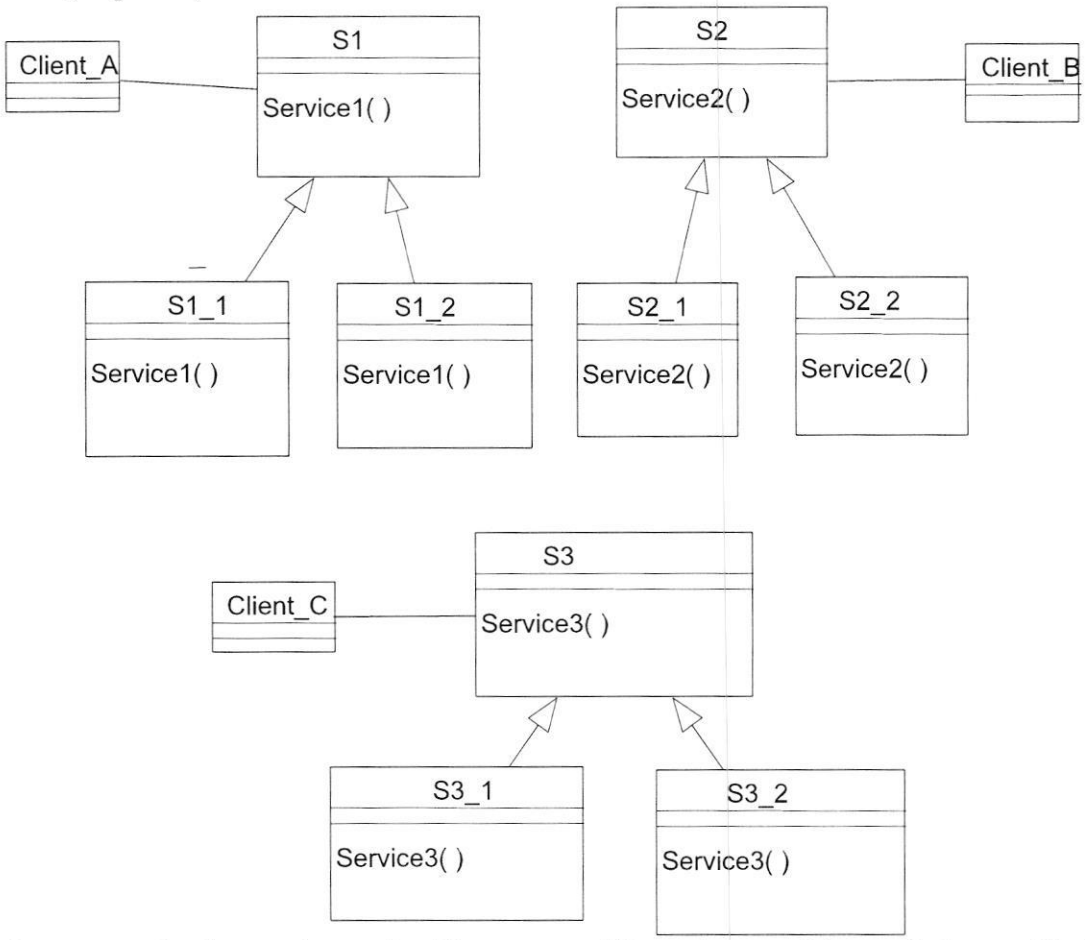
There exist two client processes, and they request the following services:

Client-A	**Client-B**
void Service1(string, int, int)	void Service1(string, int)
void Service2(string, int)	void Service2(string, int, int)
int Service3(string)	int Service3(string)
float Service4(string)	float Service4(string)

The client processes do not know the location (pointer) to 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-2.*
- Provide a sequence diagram to show how *Client-A* gets "void *Service2(string, int)* service.

PROBLEM #2 (25 points)



A design of a system is shown above. In this system, *Client_A* uses objects of classes *S1_1* and *S1_2*, *Client_B* uses objects of classes *S2_1* and *S2_2*, and *Client_C* uses objects of classes *S3_1* and *S3_2*.

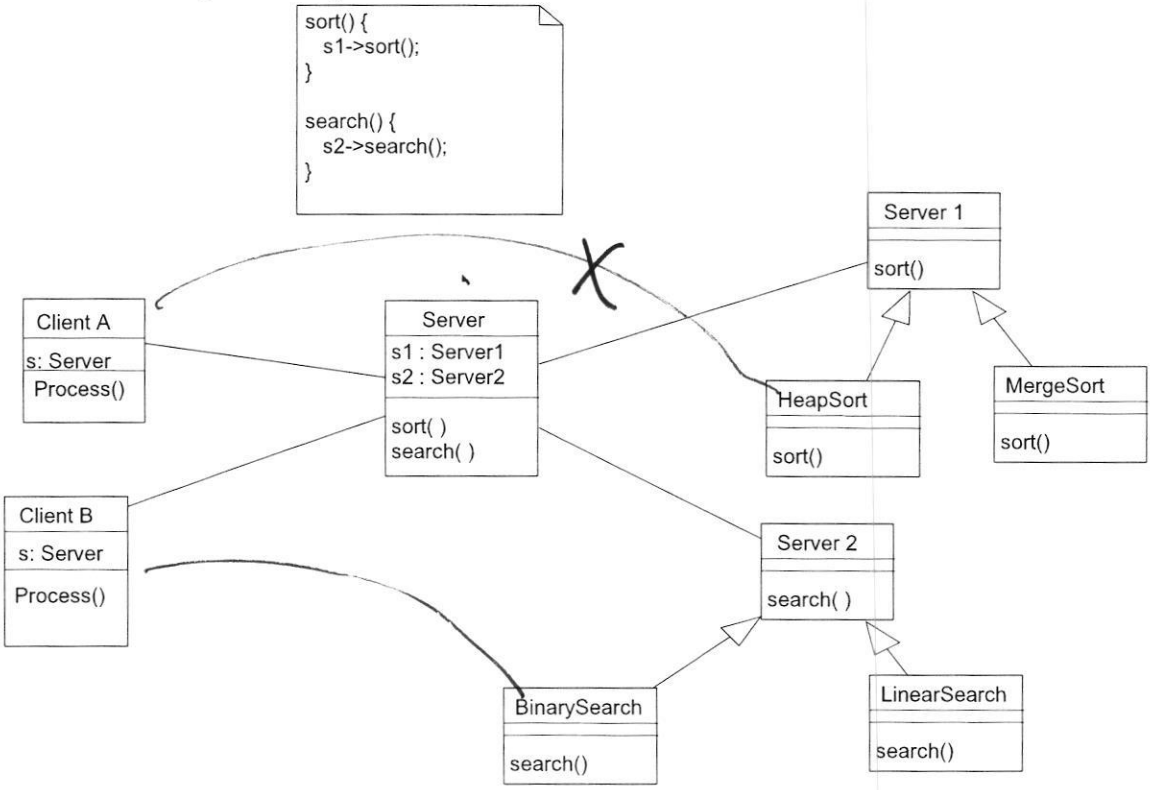
Client_A would like to use objects, operations *Service2()*, of classes *S2_1* and *S2_2* by invoking operation *Service1()*. *Client_C* would like to use objects, operation *Service1(),* of classes *S1_1* and *S1_2* by invoking operation *Service3()*. In addition, *Client_B* would like to use objects, operation *Service3(),* of classes *S3_1* and *S3_2* by invoking operation *Service2()*.

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 *S2_1* and *S2_2* by invoking operations *Service1*(), (2) *Client_B* can use objects of classes *S3_1*, and *S3_2* by invoking operations *Service2(),* and (3) *Client_C* can use objects of classes *S1_1*, and *S1_2* by invoking operations *Service3()*. 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)
Consider a design of the system shown below:



Client A and *Client B* get services, *sort()* and *search()*, directly from the *Server*. However, the *Server* gets the appropriate services from *HeapSort* or *MergeSort* servers for *sort()* service. In addition, the *Server* gets the appropriate services from *BinarySearch* or *LinearSearch* servers for *search()* service.

In this design, clients do not have control where the services are coming from. However, *ClientA,* by invoking *sort()* and *search()* on the *Server,* would like to get *sort()* service from *HeapSort* server and *search()* service from *BinarySearch* server. On the other hand, *ClientB,* by invoking *sort()* and *search()* on the *Server,* would like to get *sort()* from *MergeSort* server and *search()* from *LinearSearch* server. Notice that the current design does not support this option.

Use the **abstract factory** design pattern to solve this problem. In your solution, the *Client* classes should be completely **decoupled** from the issue of invoking services by the *Server* for appropriate versions of *sort()* and *search()*. Notice that none of the classes (and operations) shown in the above class diagram should be modified. However, new operations/classes can be introduced.

- Provide the class diagram and briefly 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. You do not have to provide any description for classes/operations of the above class diagram (only new classes/operations should be described).
- Provide a sequence diagram to show how *ClientB* gets *sort() service* from *MergeSort* server and *search()* service from *LinerSearch* server by invoking *sort()* and *search()* on the *Server*.