Project description
part #1
is posted

A sample MDA-EFSM for
ATMs components
is posted

Pipes and Filters architecture

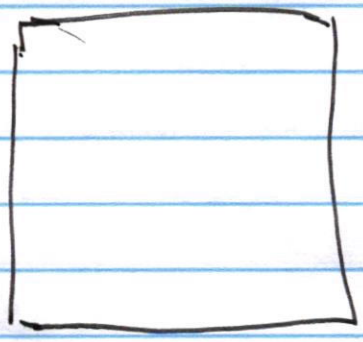
software architecture

a set of components
+
relationships between components

Components:

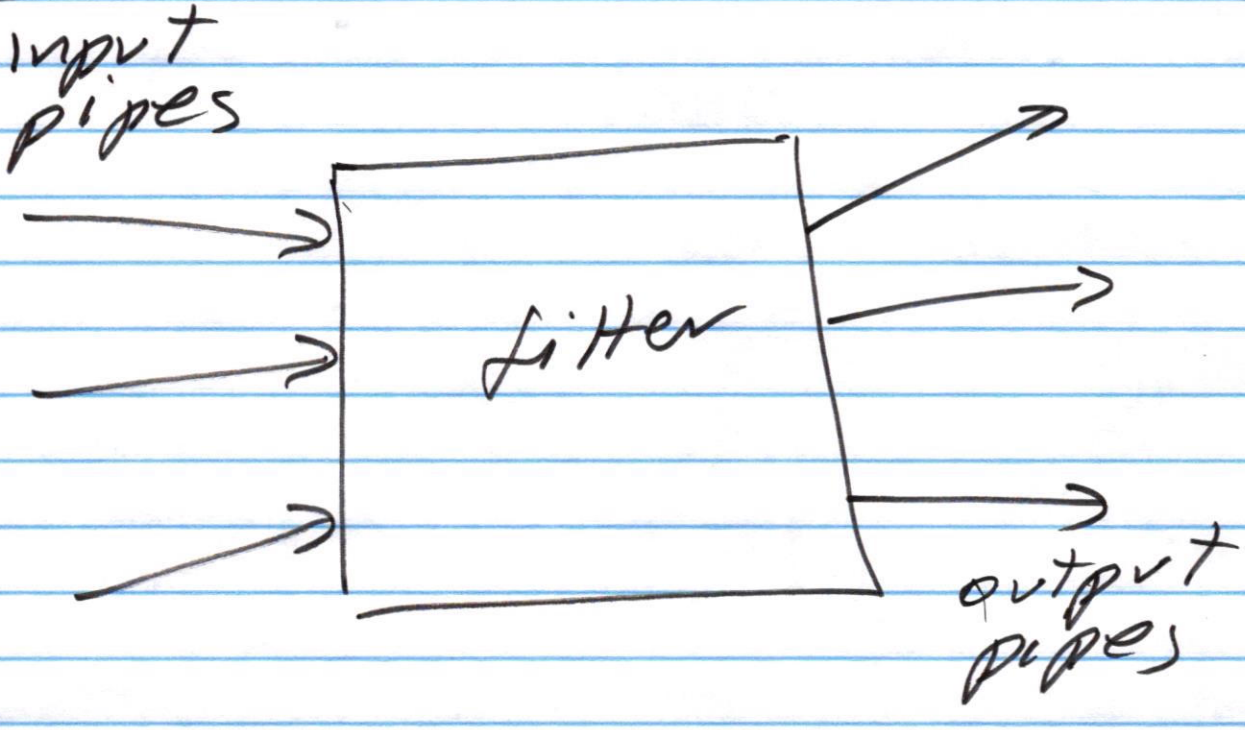
Filter: processing component

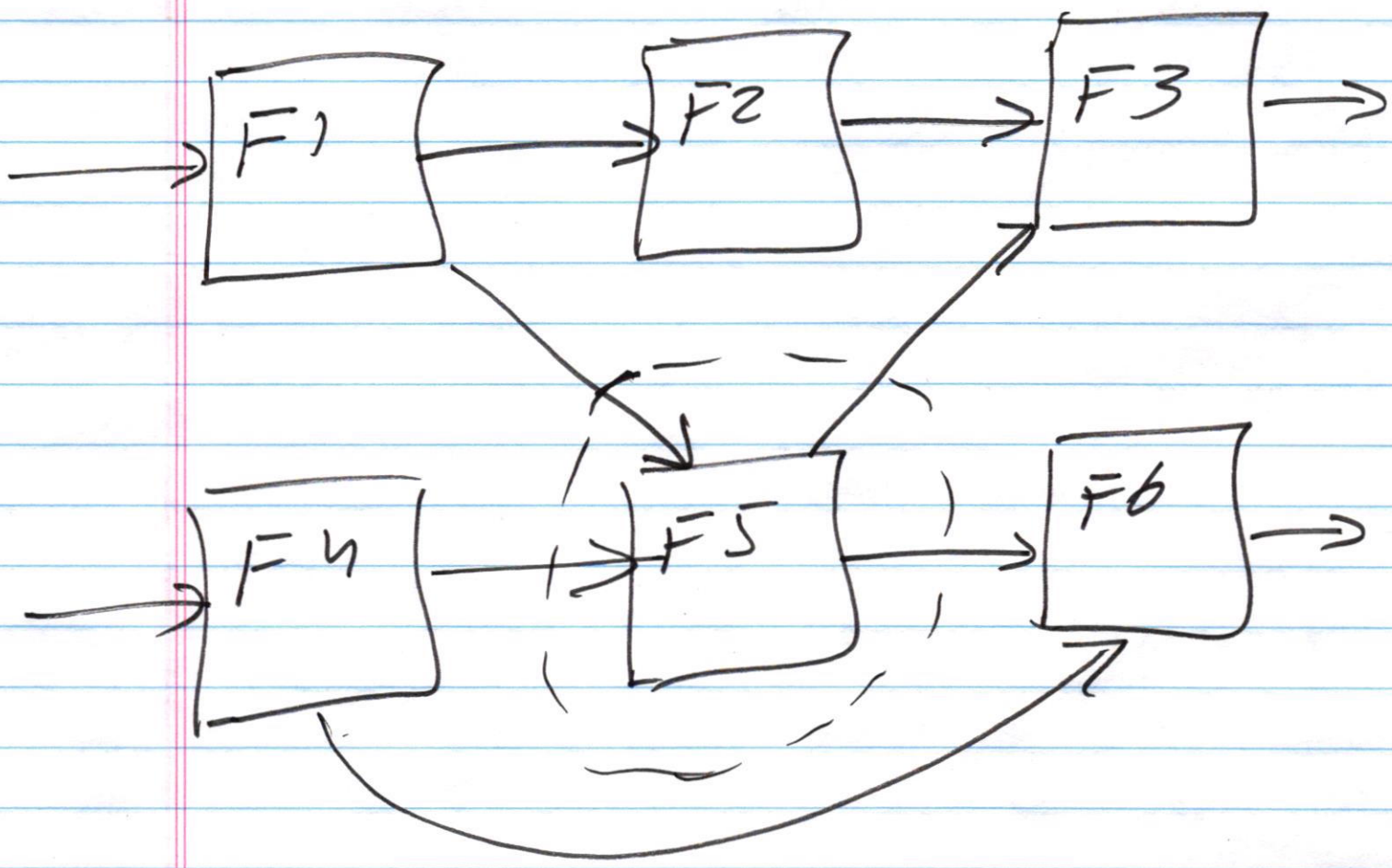
pipes: "transmit" data between filters



filter

pipe

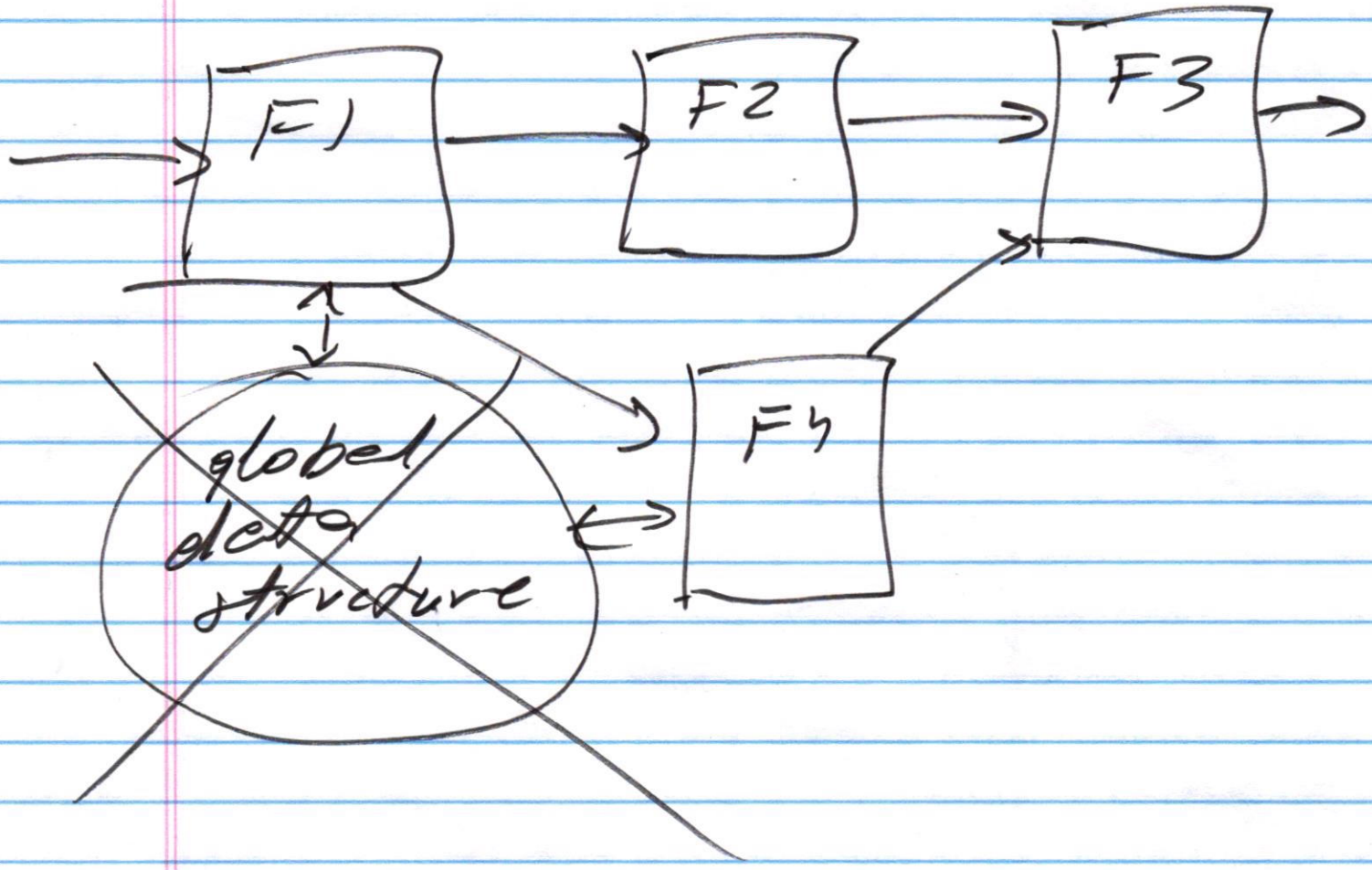



F1
F2
F3
F4
F5
F6

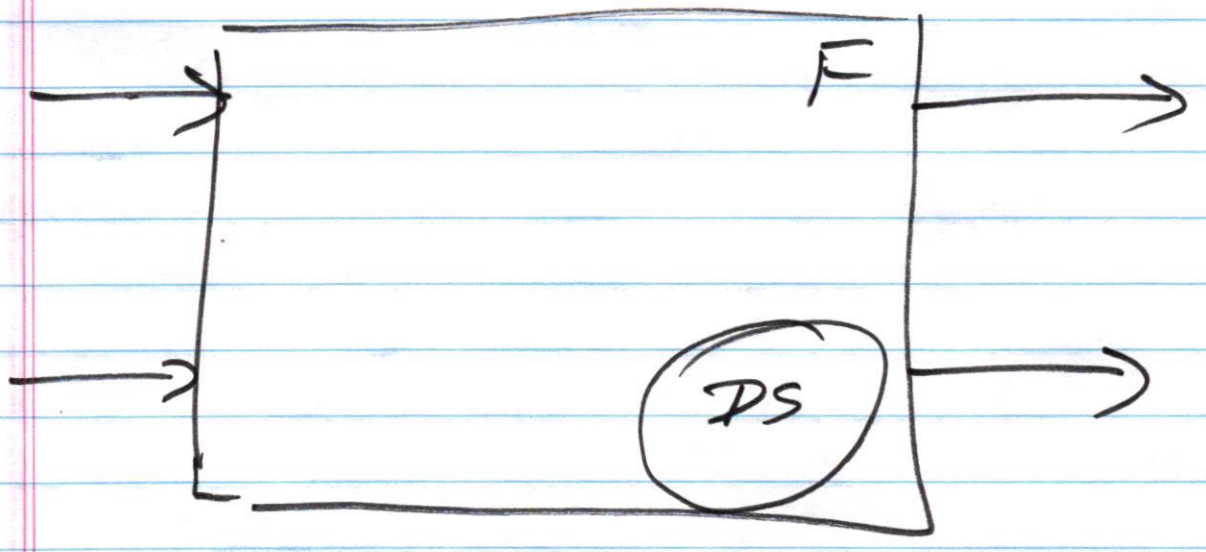
Properties

1. Filters are independent entities.

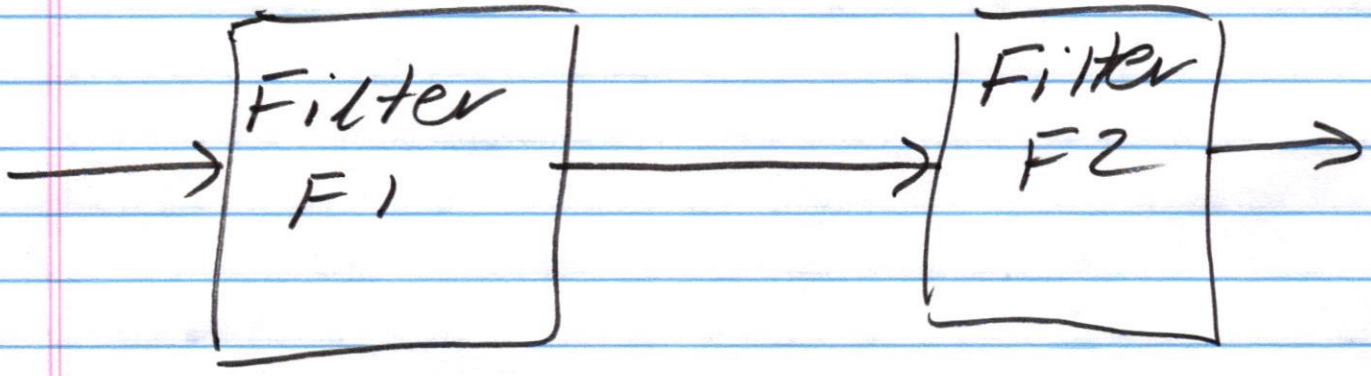
2. Filters do not have access to global data structures.



3. Filters do not share "internal states/data" of of other filters



Design issues

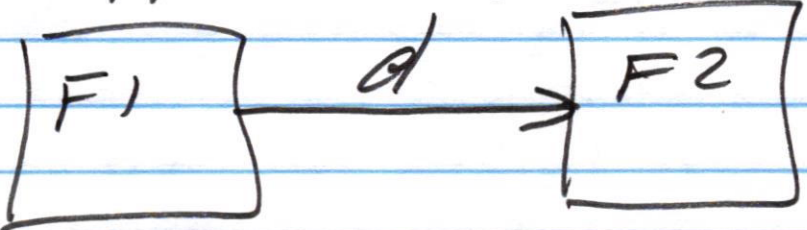


Filters

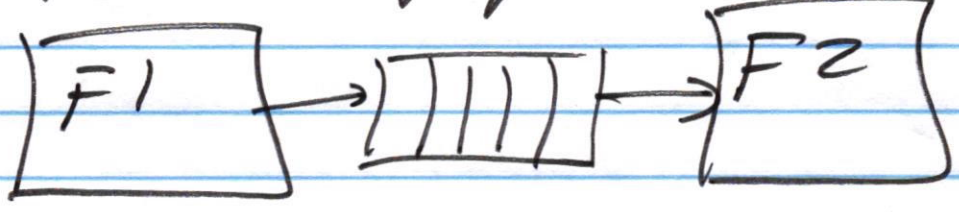
* passive filters
* active filters

Pipes

1. unbuffered pipes.

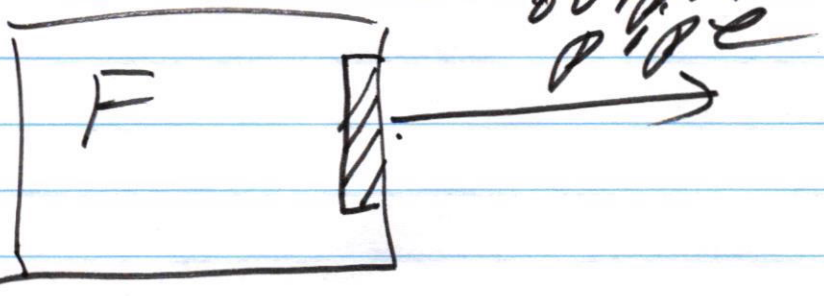


2. buffered pipes.



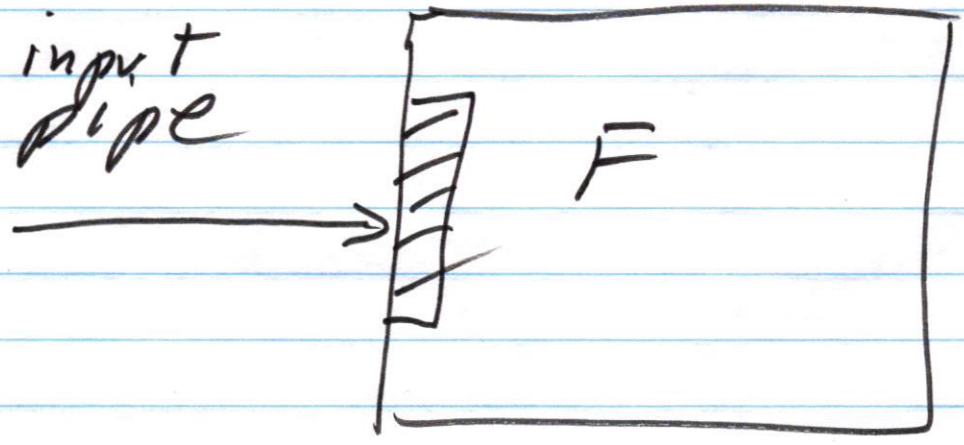
1. passive filters.

(a) with pull-out pipes



output pipe "pulls" data out of the filter F

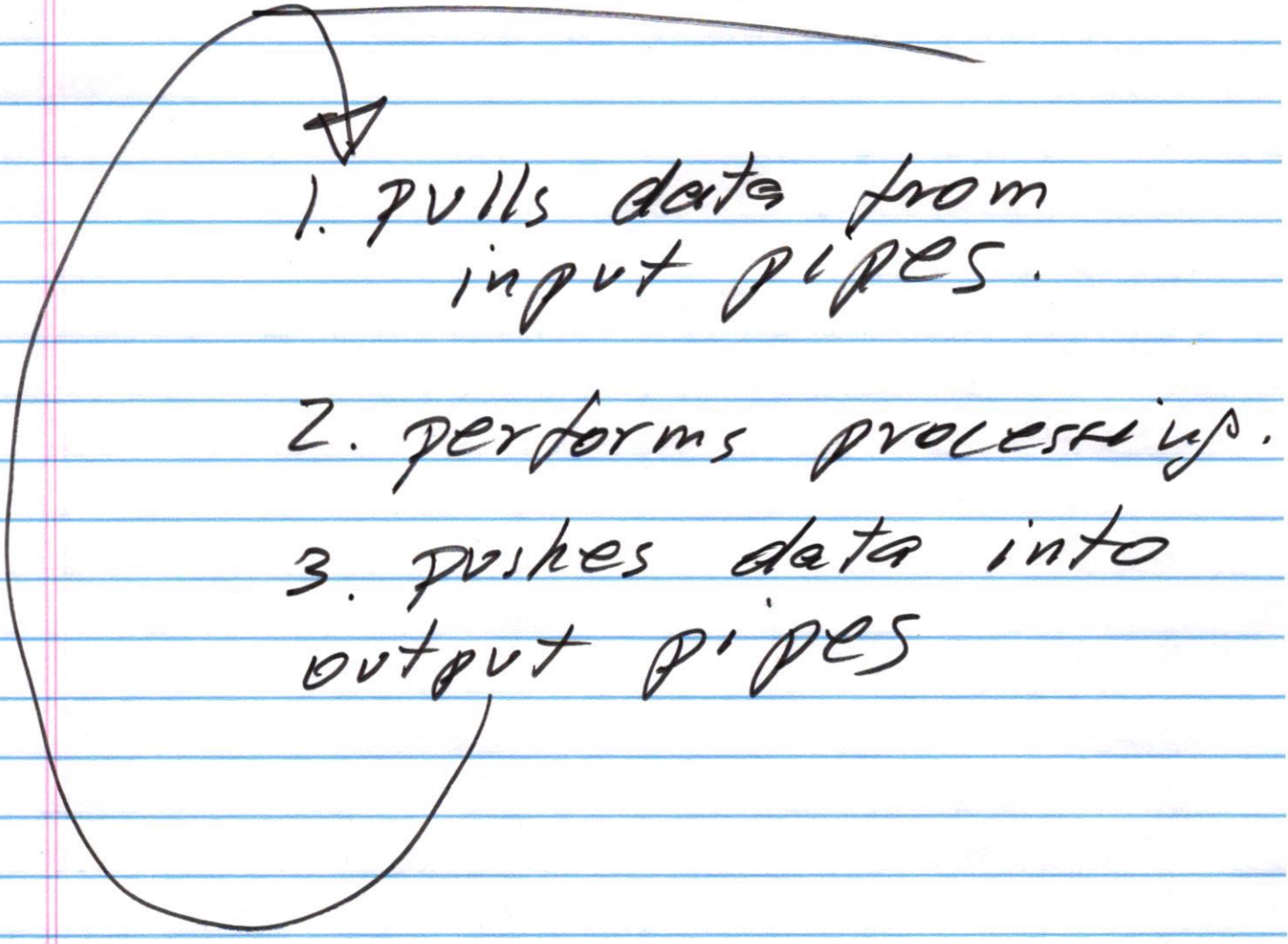
(b) with push pipes.



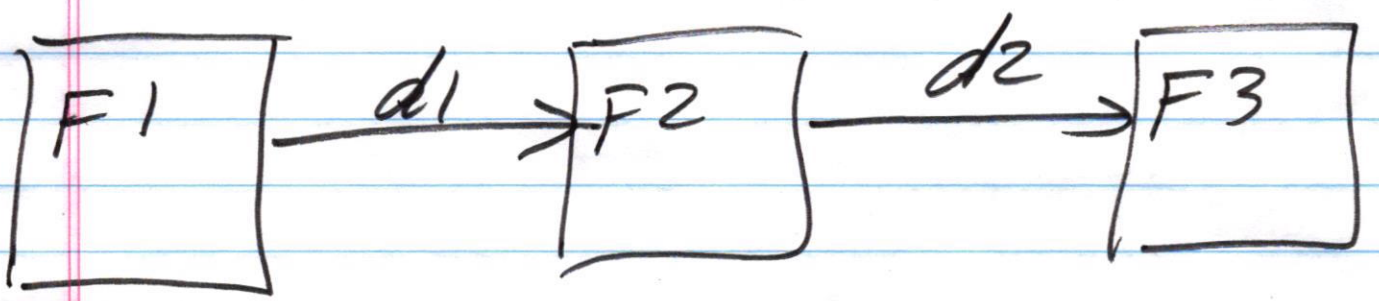
Input pipe "pushes" data into filter F

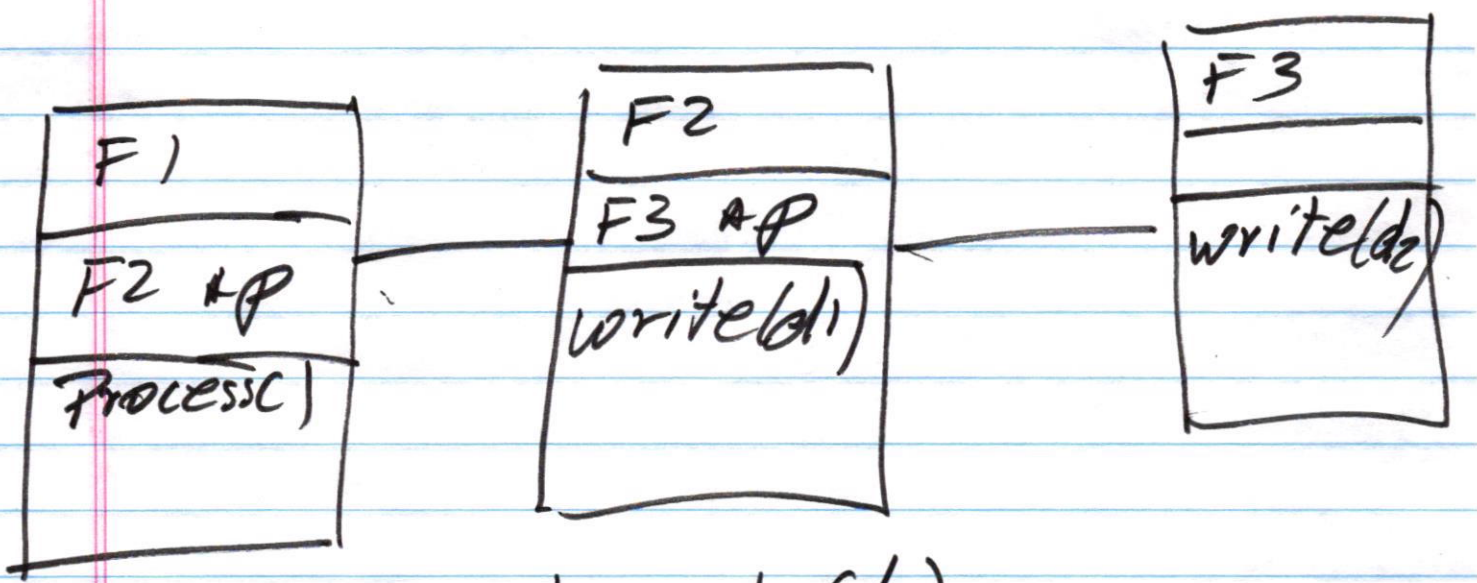
Active filter

Active loop



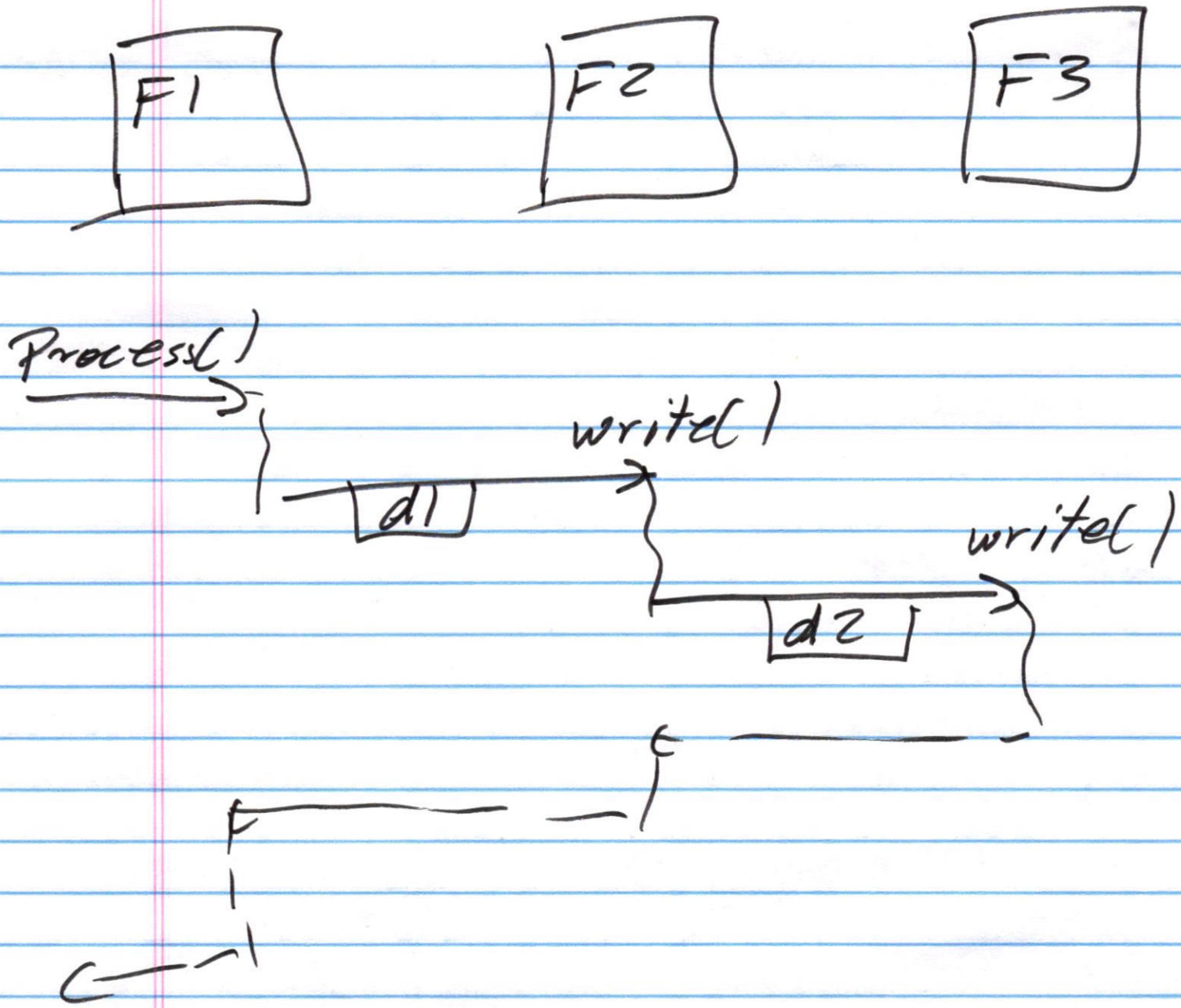
I. Passive filters
unbuffered pipes
push pipes

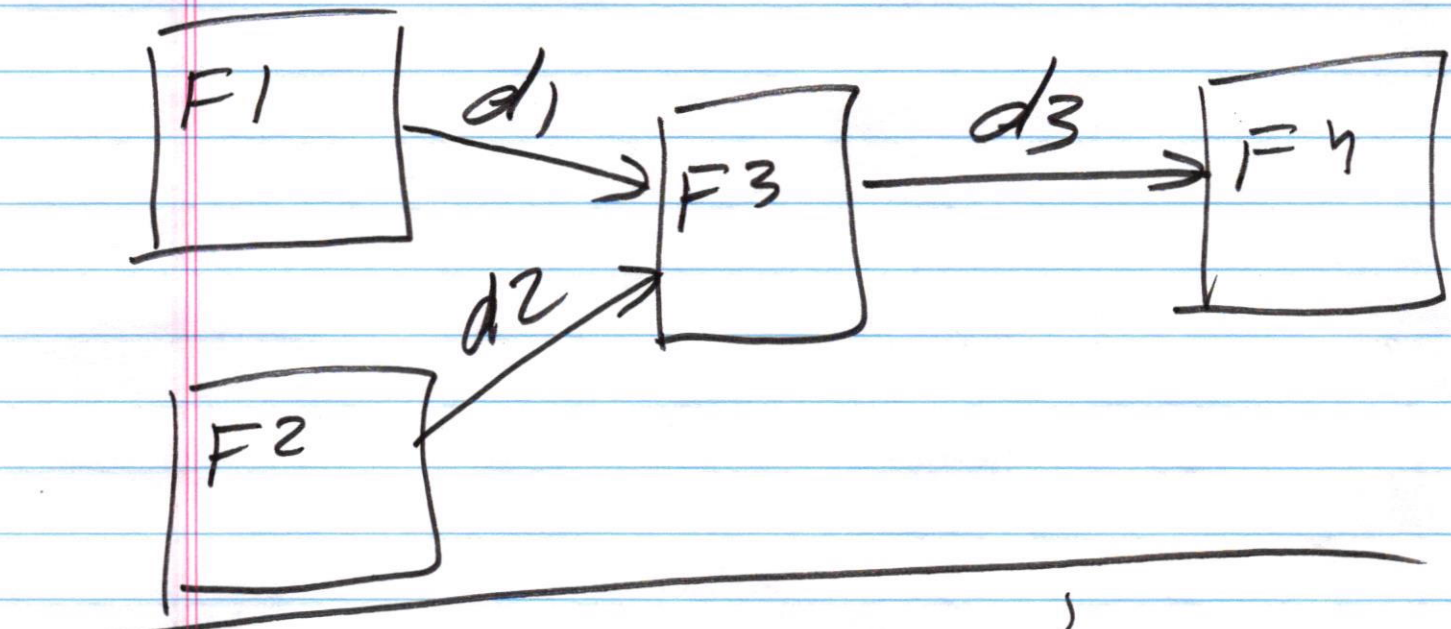


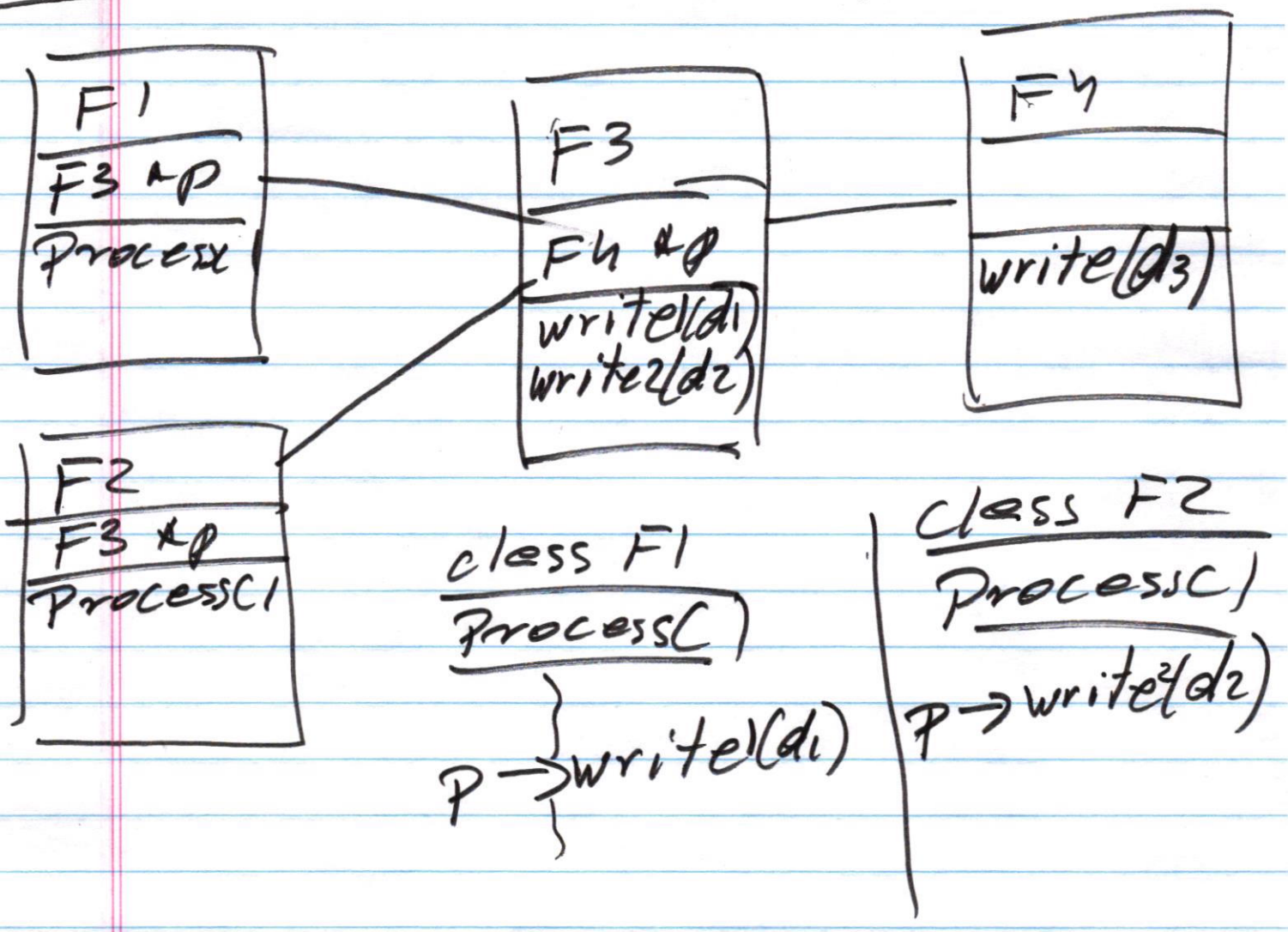


Process()
computes d1
p → write(d1)

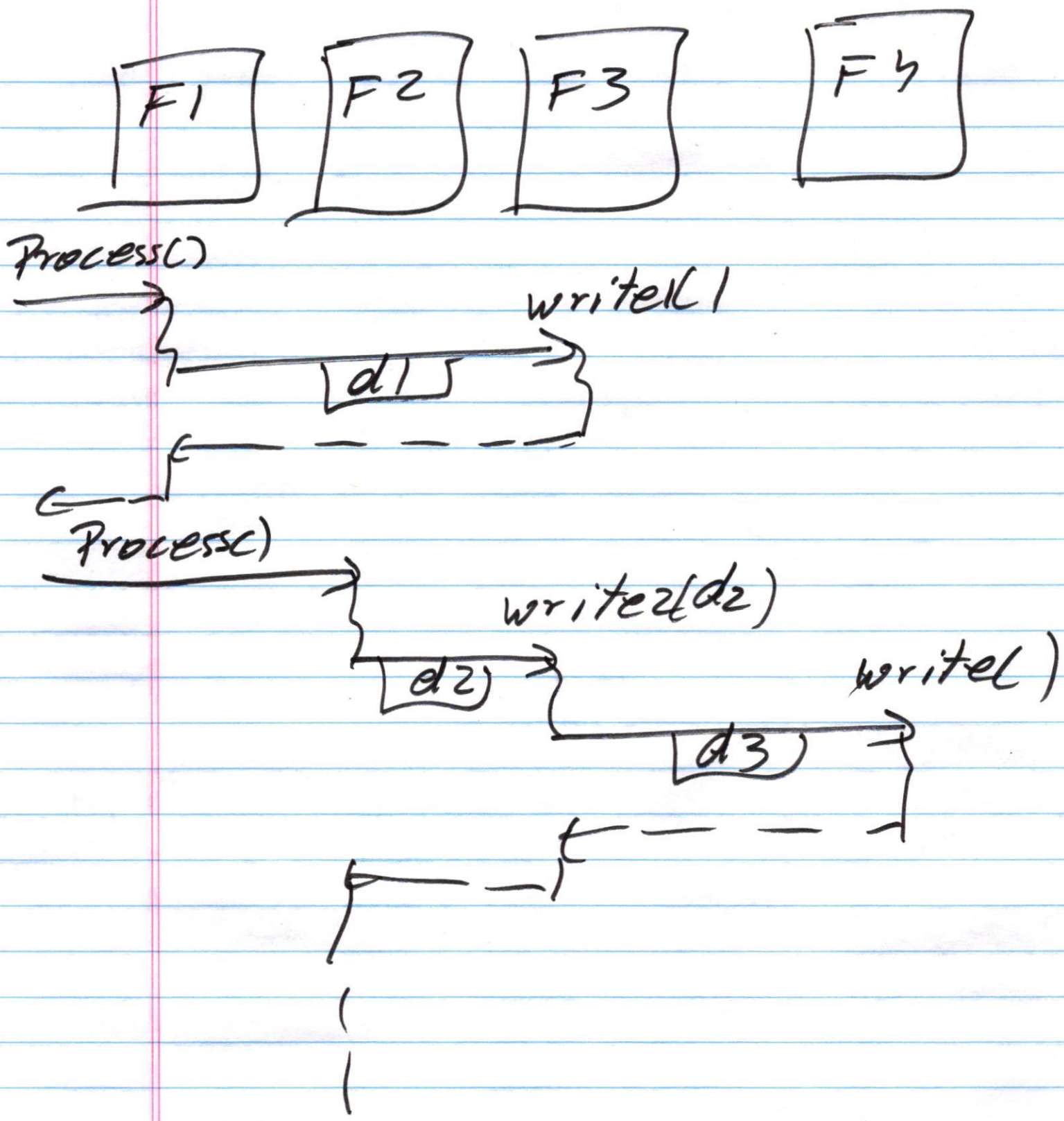
write(d1)
computes d2
p → write(d2)

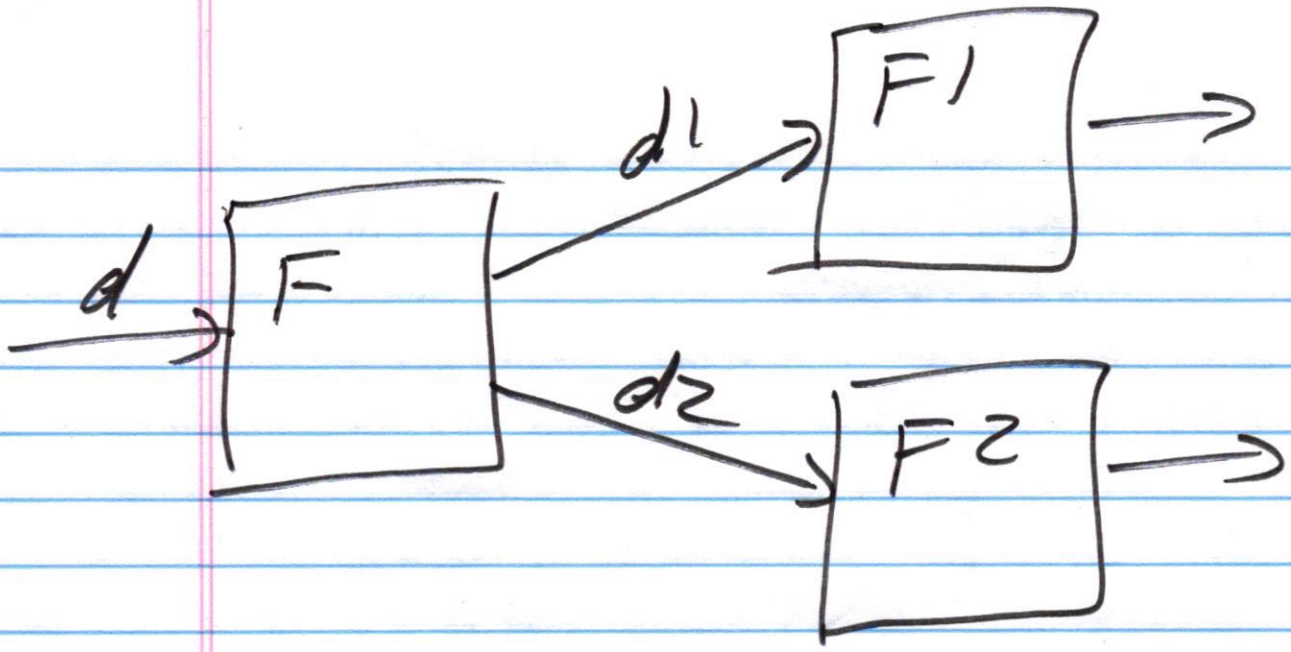

F1
F2
F3
Process()
write()
d1
write()
d2

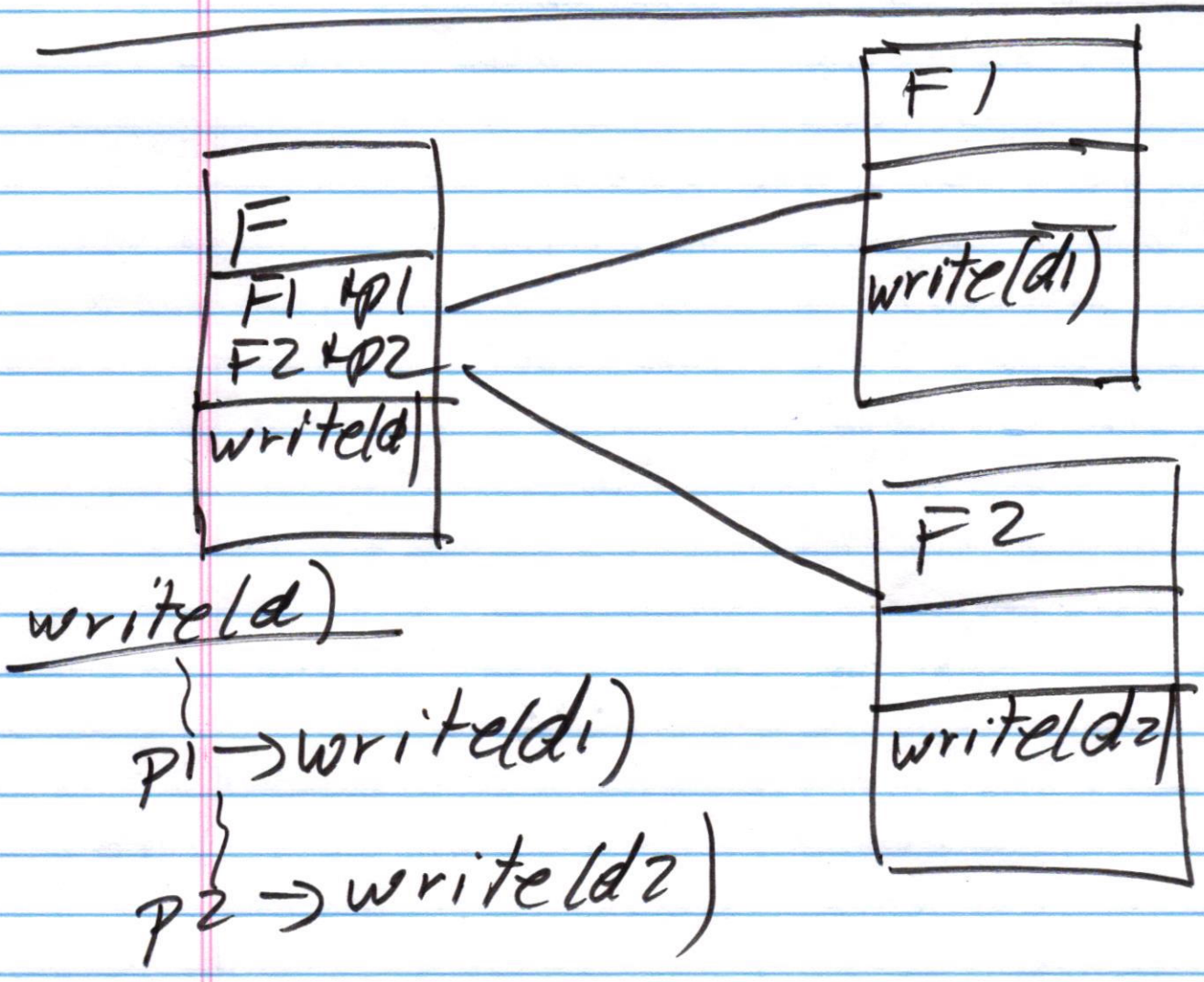

F1
d1
F3
d3
F4
d2
F2

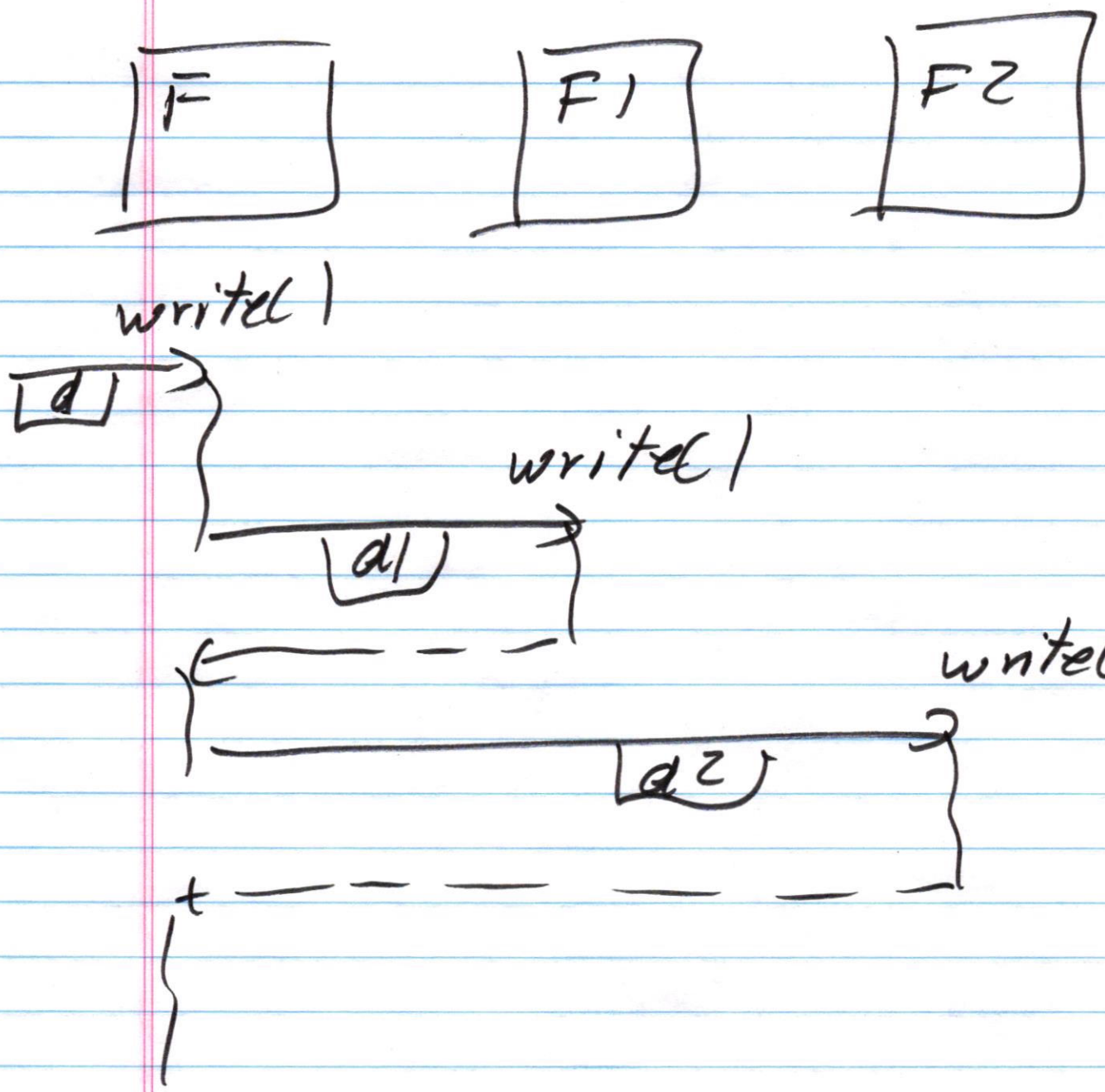

F1
F3 *p
Process
F3
F4 *p
write1(d1)
write2(d2)
F4
write(d3)
F2
F3 *p
Process1
class F1
Process()
P → write1(d1)
class F2
Process1()
P → write2(d2)

```
class F3
---------------------------------------
write1 (d1)
{ if (d2 is present)
  { do processing
      using d1 and d2
      to compute d3
      P -> write(d3)
  }
  else store d1
}
---------------------------------------
write2 (d2)
{ if (d1 is present) {
    do processing using
       d1 and d2
       to compute d3
       P -> write(d3)
  }else store d2
}
```

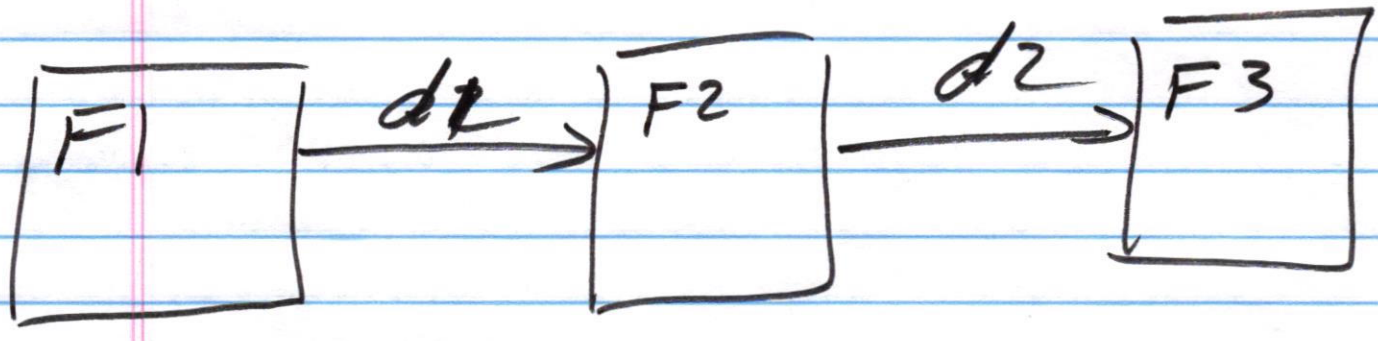

F1
F2
F3
F4
Process()
write1(d1)
d1
Process()
write2(d2)
d2
write1(d3)
d3

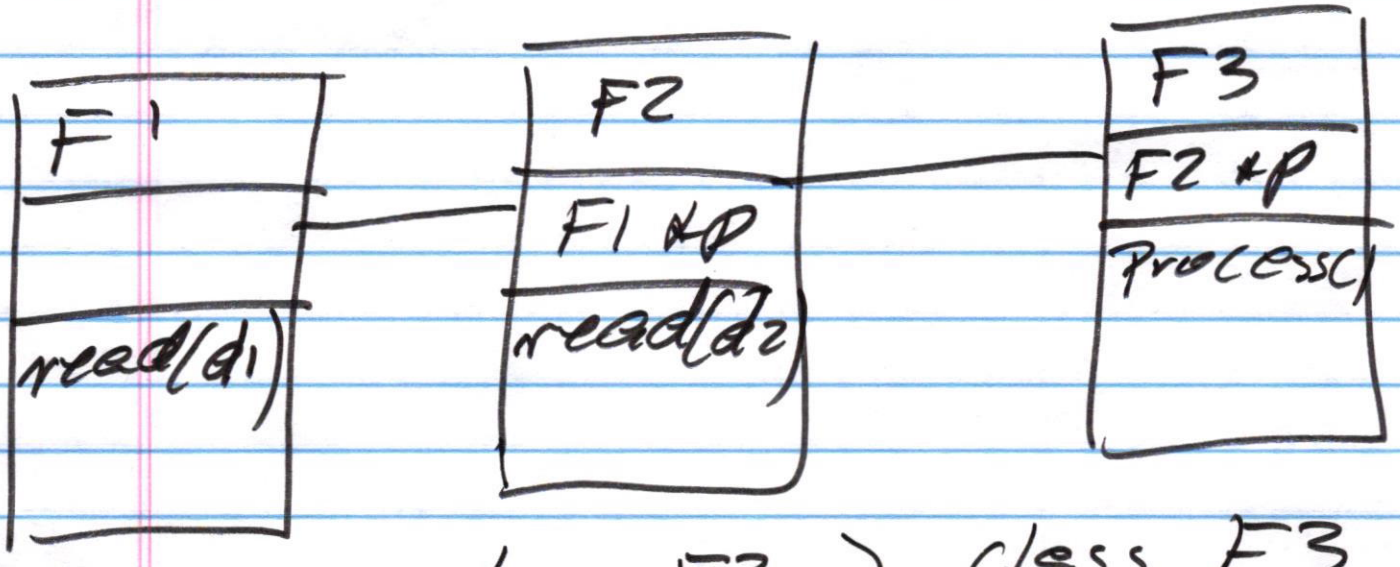

d
F
d1
F1
d2
F2


F
F1 *p1
F2 *p2
writeld
F1
write(d1)
F2
writeld2
write(d)
p1 → write(d1)
p2 → write(d2)


F
F1
F2
write(1
d
write(1
d1
write(1
d2
t

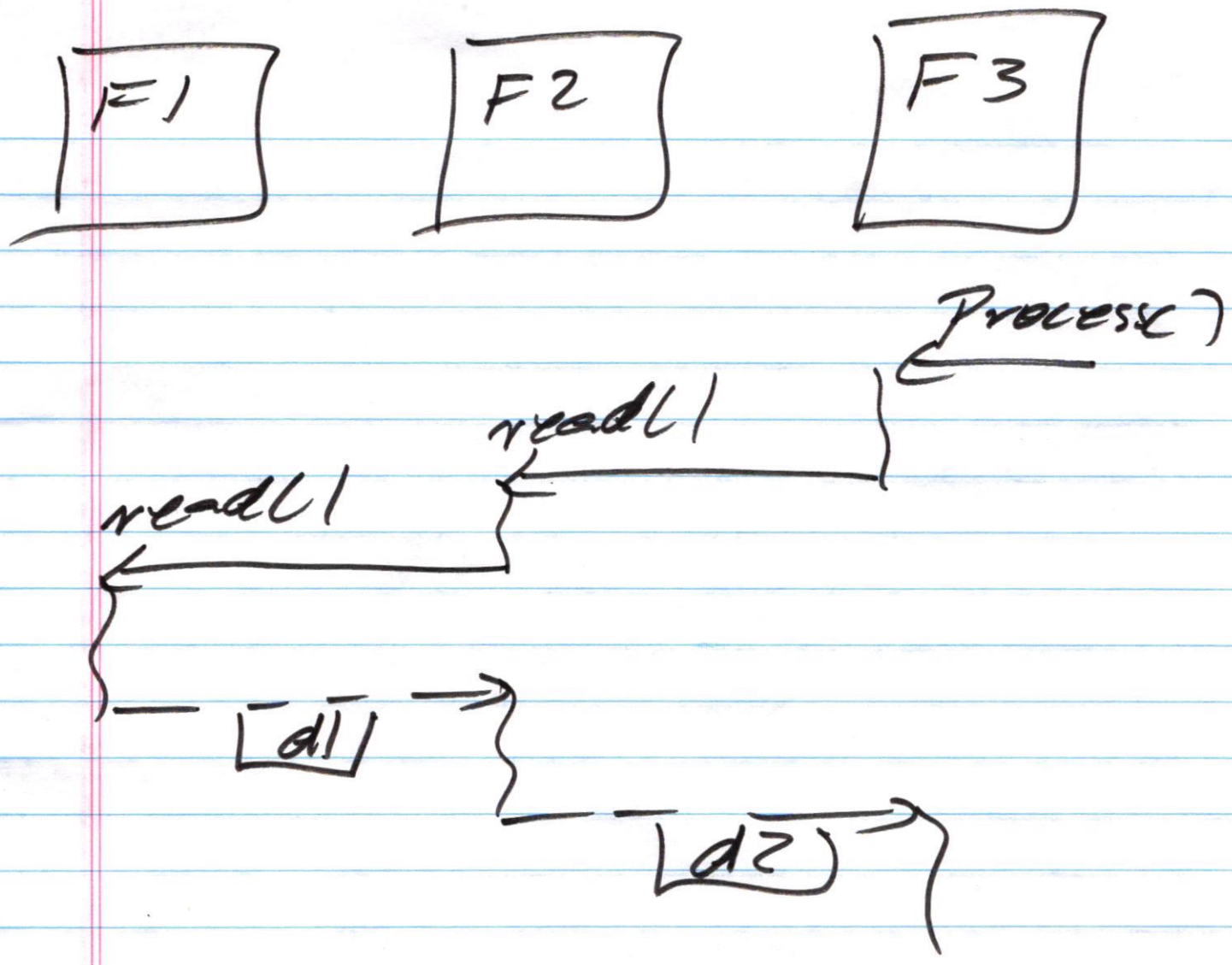
II passive filters
unbuffered pipes
pull-out pipes

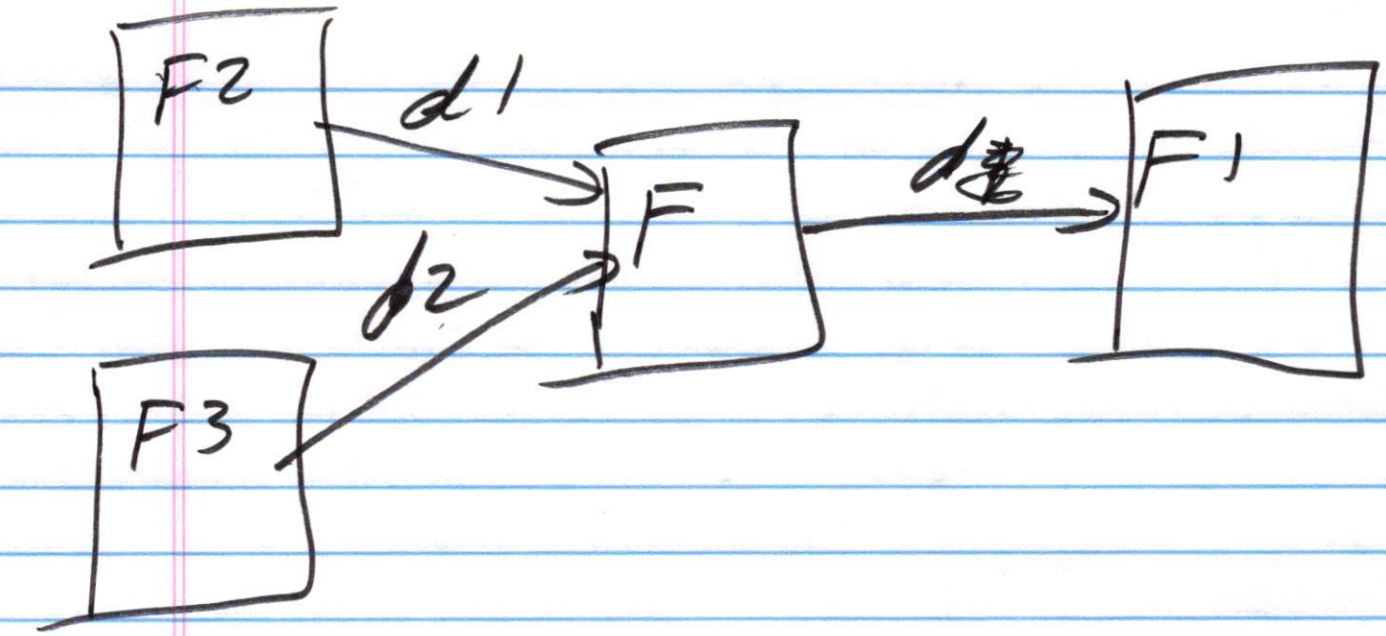


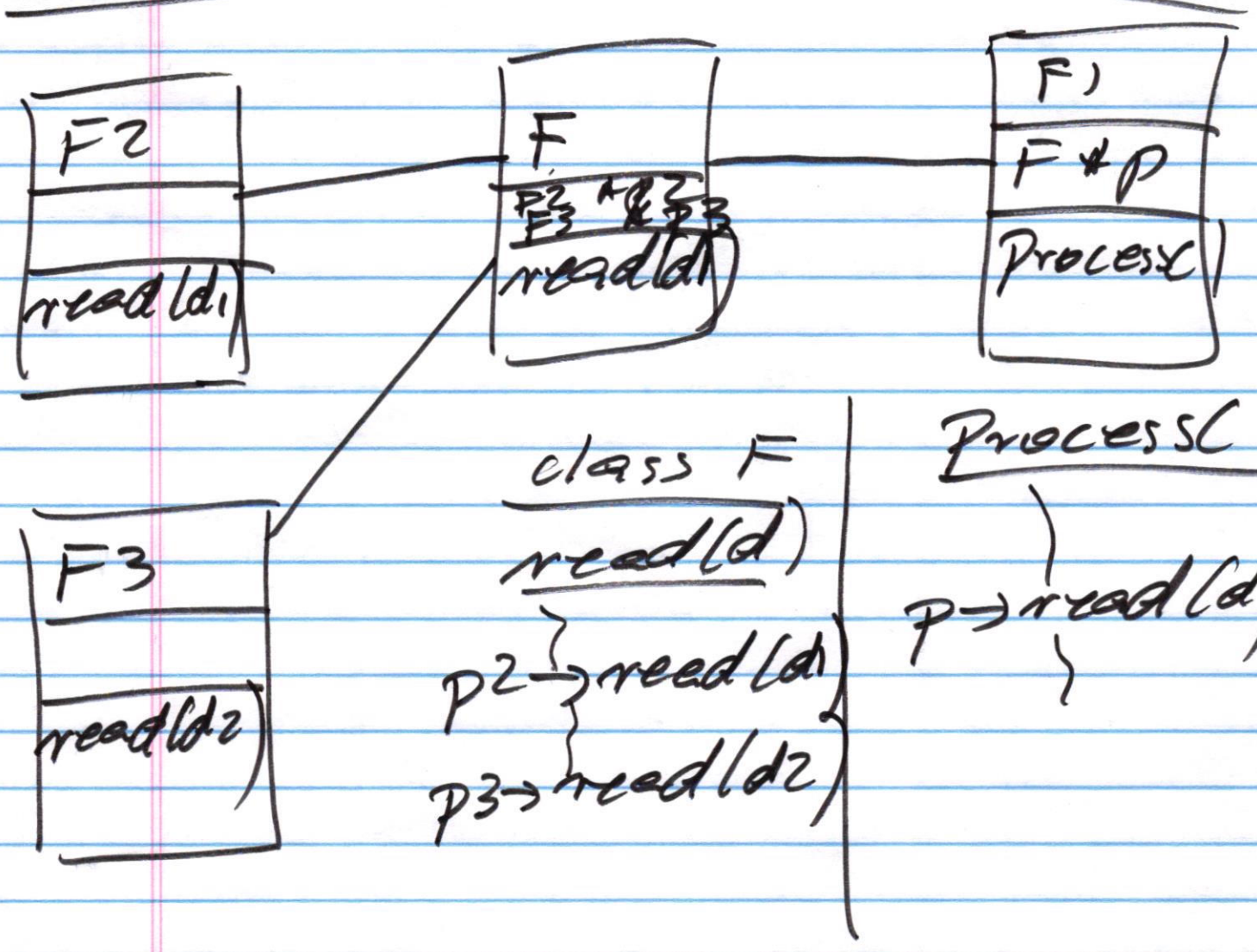


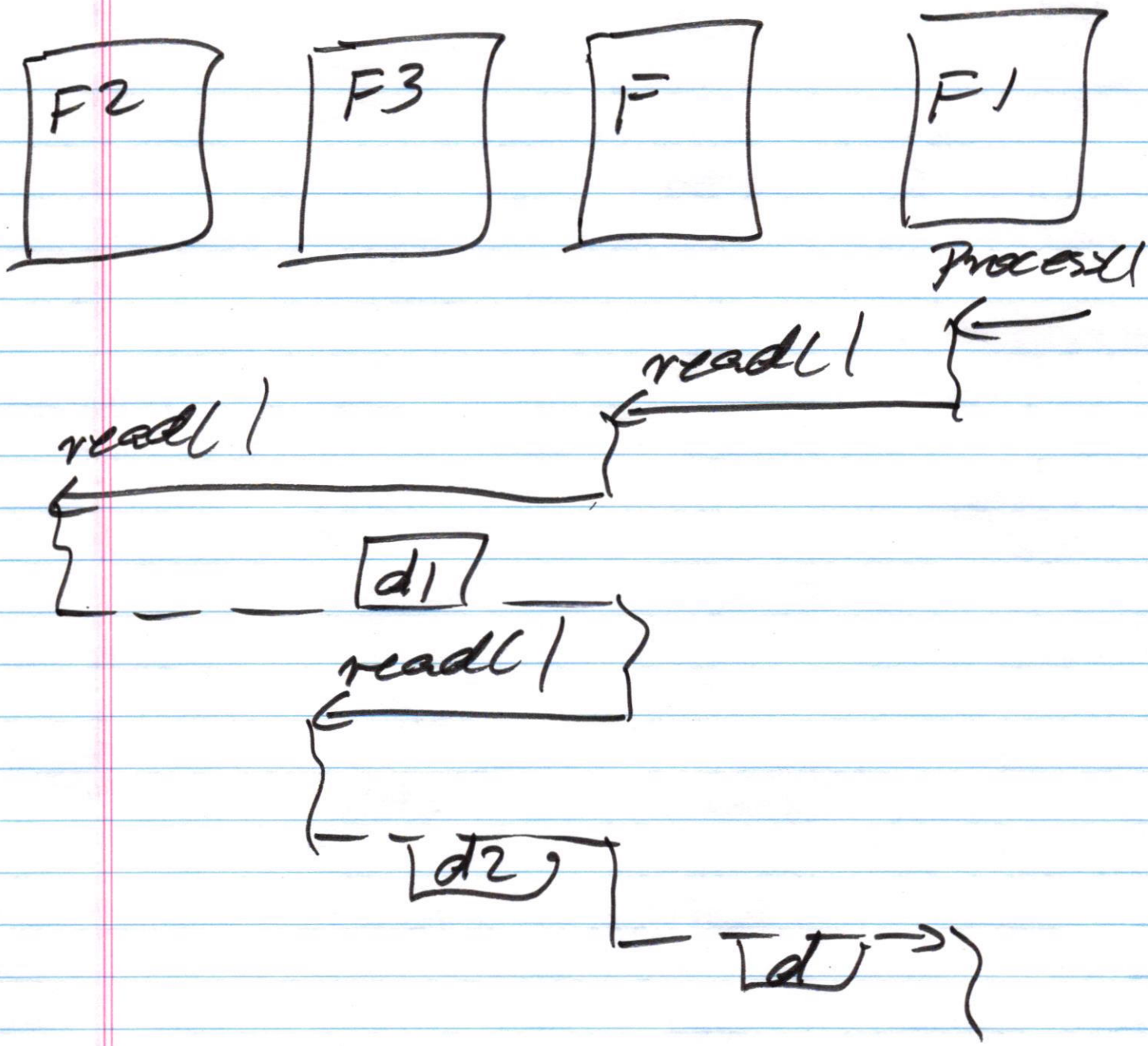
class F2
read(d2)
p->read(d1)

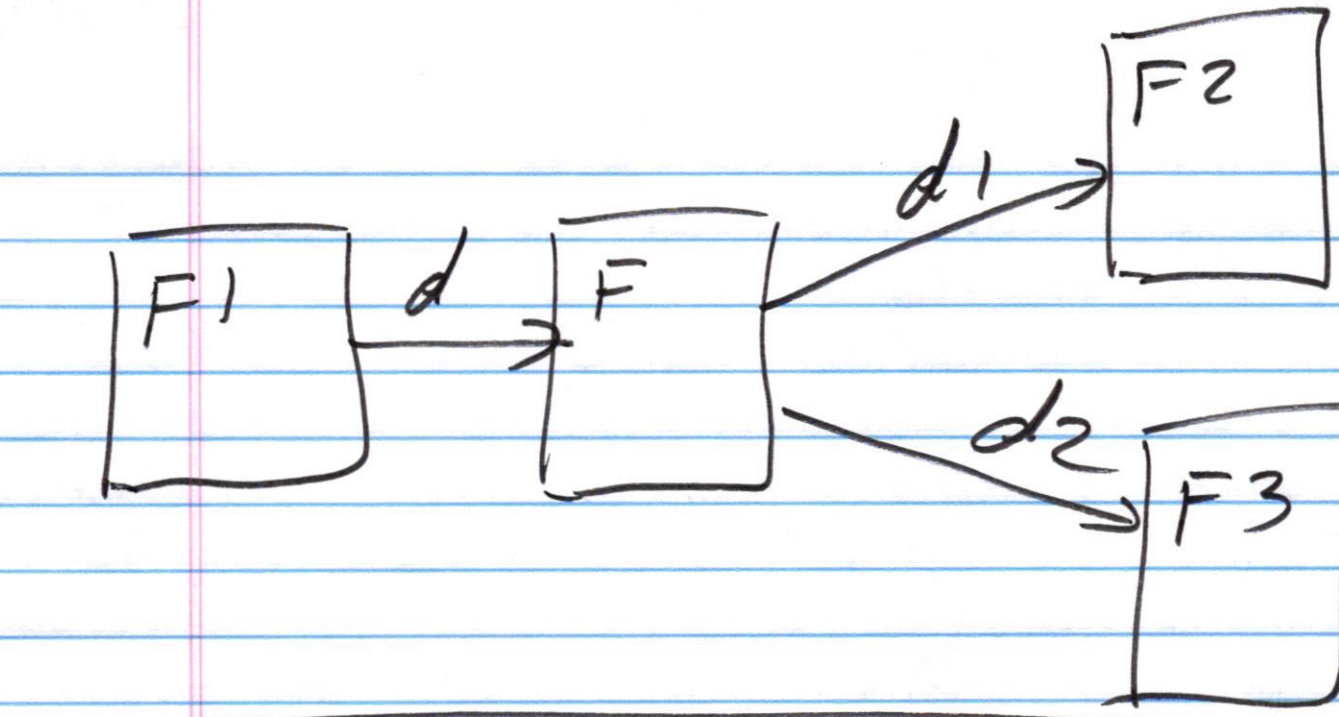
class F3
Process()
p->read(d2)

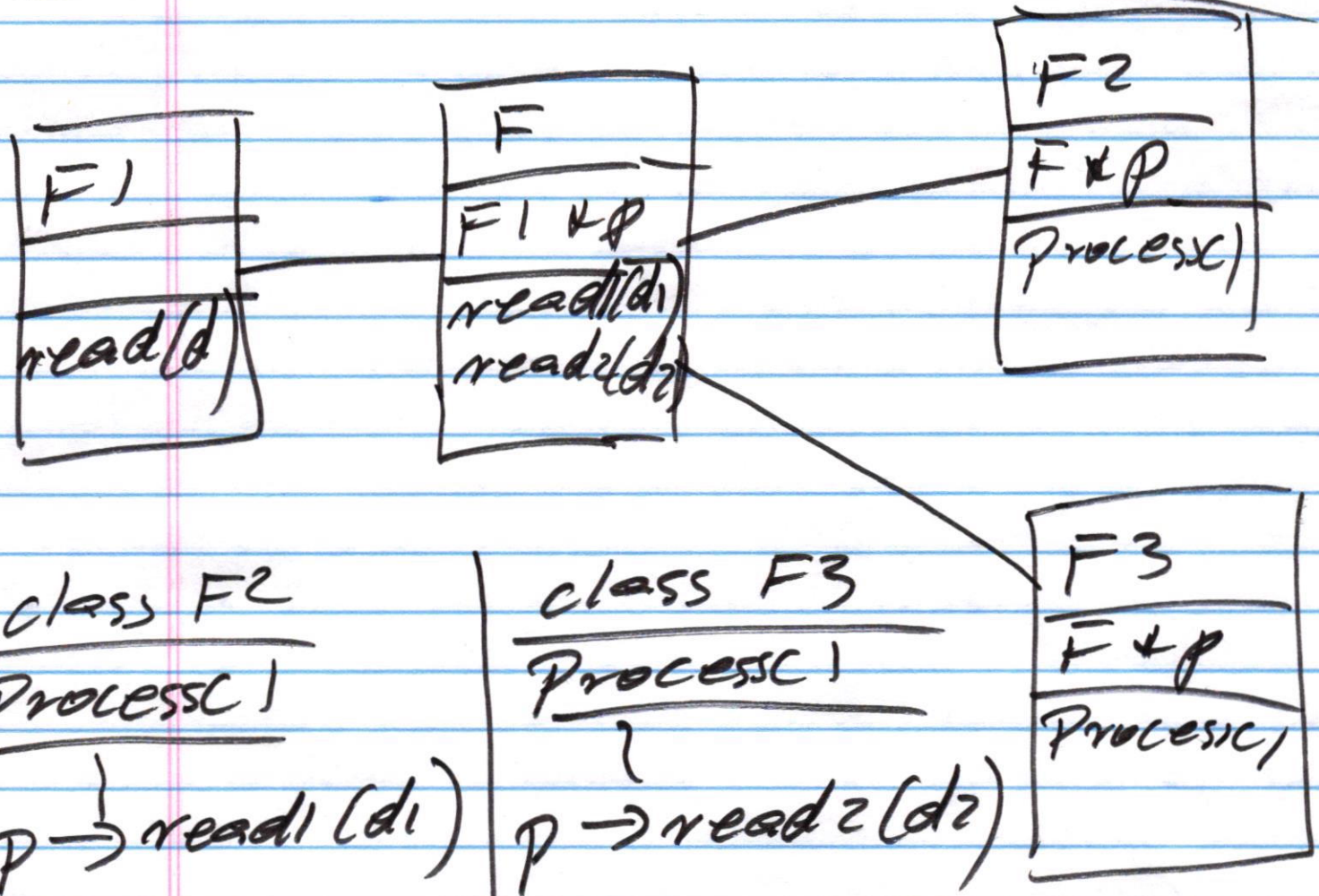

F1
F2
F3
Process()
read()
read()
d1
d2


F2
d1
F
d*
F1
d2
F3


F2
read(d1)
F
F3 *&F3
read(d1)
F1
F*p
Process(1)
F3
read(d2)
class F
read(d)
p2→read(d)
p3→read(d2)
Process(1)
p→read(d)


F2
F3
F
F1
Process1
readL1
readL1
d1
readL1
d2
d





class F2

Process()

p → read1(d1)

class F3

Process()

p → read2(d2)

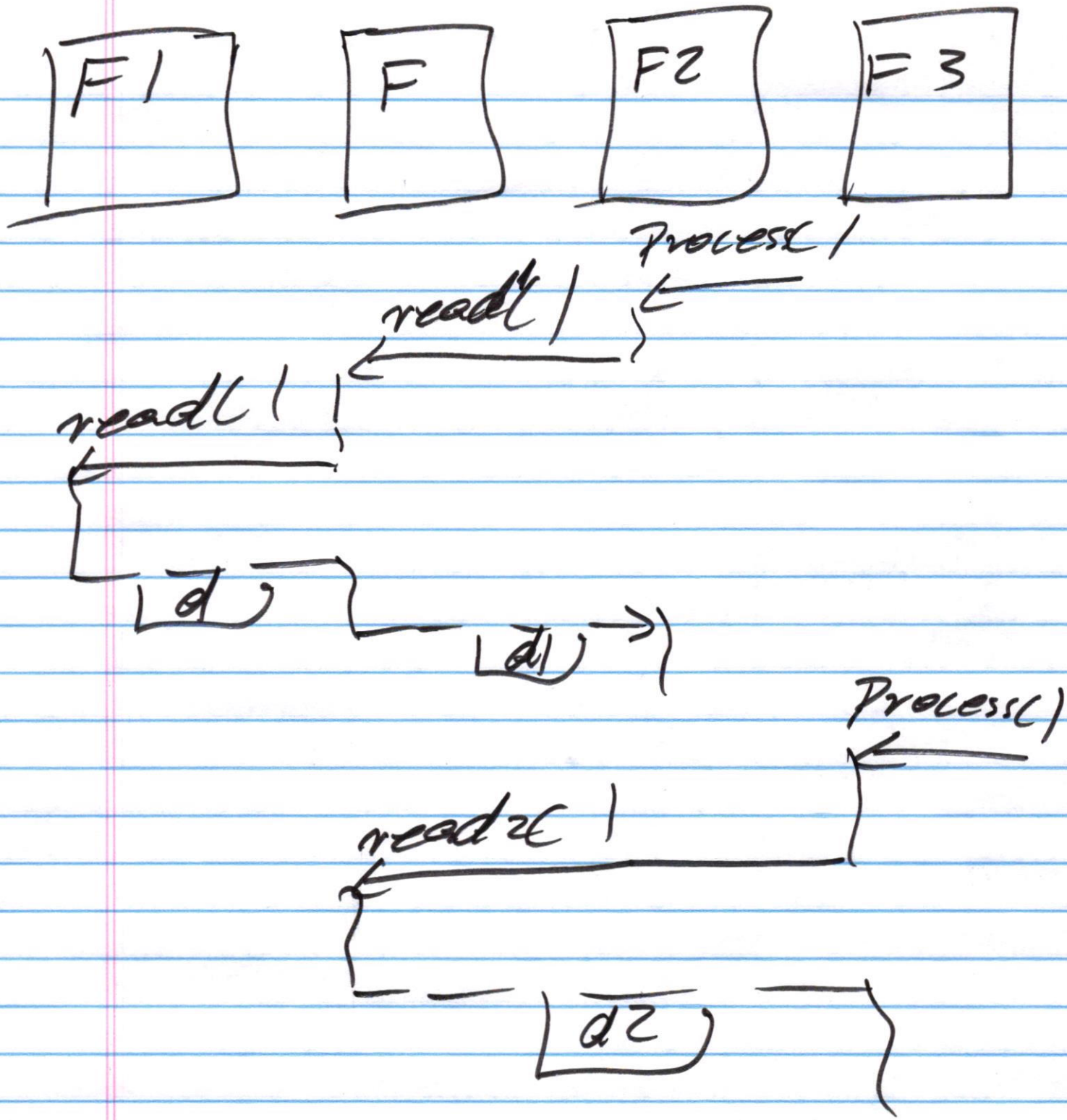
class F

read1(d1)

{ if (d is not present)
    p → read(d)
  }
  process d to compute d1
  return d1
}

read2(d2)

{ if (d is not present)
  { p → read(d)
  }
  process d to compute
      d2
  return d2
}


F1
F
F2
F3
Process()
read()
read()
d
d1
Process()
read2()
d2

Project

create MDA - EFSM
that captures "meta"
behavior of 2 GPs.

deadline: November 7

CS586 PROJECT – General Description

CS 586; Fall 2025

Deadlines:

Part #1: MDA-EFSM (5 points): **Friday, November 7, 2025**

Late submissions: 50% off

After **November 11,** the MDA-EFSM will not be accepted.

This is an **individual** project, not a team project.

Submission: The MDA-EFSM assignment must be submitted on Canvas. Your submission should be a **single PDF file (**otherwise, a 10% penalty will be applied). The hardcopy submissions will not be accepted.

The detailed description of Part #2 of the project will be posted later.

Goal:

The goal of this project is to design two different gas pump components using the Model-Driven Architecture (MDA) and then implement these gas pump components based on this design using an object-oriented programming language.

Description of the Project:

There are two gas pump components: *GP-1* and *GP-2*.

The gas pump **GP-1** component supports the following operations:

Activate (float a)	// the gas pump is activated, where *a* is the price of the gas per liter
Start()	//start the transaction
Cancel()	// cancel the transaction
Approved()	// credit card is approved
StartPump()	// start pumping gas
PumpLiter()	// one liter of gas is dispensed
PayCredit()	// pay for gas by a credit card
Reject()	// credit card is rejected
PayCash(float c)	// pay for gas by cash, where *c* represents prepaid cash
StopPump()	// stop pumping gas

The gas pump **GP-2** component supports the following operations:

Activate (int a, int b)	// the gas pump is activated, where *a* is the price of the Regular gas // *b* is the price of Diesel gas per gallon
Start()	//start the transaction
PayCredit()	// pay for gas by a credit card
Reject()	// credit card is rejected
Approved()	// credit card is approved
Diesel()	// Diesel gas is selected
Regular()	// Regular gas is selected
StartPump()	// start pumping gas
PayDebit(int p)	// pay for gas by a debit card, where *p* is a pin #
Pin(int x)	// pin # is provided, where *x* represents the pin #
Cancel()	// cancel the transaction
PumpGallon()	// one gallon of gas is dispensed
StopPump()	// stop pumping gas
FullTank()	// Tank is full and the pump is stopped

Both gas pump components are state-based and are used to control simple gas pumps. Users can pay by cash, a credit card, or a debit card. The gas pump may dispense different types of gasoline. The price of the gasoline is provided when the gas pump is activated. The detailed behavior of gas pump components is specified using EFSM. The EFSM of Figure 1 shows the detailed behavior of gas pump *GP-1,* and the EFSM of Figure 2 shows the detailed behavior of gas pump *GP-2*. Notice that there are several differences between gas pump components.

Aspects that vary between two gas pump components:

a. Types of gasoline pumped
b. Types of payment
c. Display menu(s)
d. Messages
e. Receipts
f. Operation names and signatures
g. Data types
h. etc.

The goal of this project is to design two gas pump components using the Model-Driven Architecture (MDA) covered in the course. In the first part of the project, you should design an executable meta-model, referred to as MDA-EFSM, for gas pump components. This MDA-EFSM should capture the "generic behavior" of both gas pump components and should be decoupled from data and implementation details. Notice that in your design, there should be **ONLY** one MDA-EFSM for both gas pump components. The meta-model (MDA-EFSM) used in the Model-Driven architecture should be expressed as an EFSM (Extended Finite State Machine) model. Notice that the EFSMs shown in Figure 1 and Figure 2 are **not acceptable** as a meta-model (MDA-EFSM) for this model-driven architecture.

SUBMISSIONS & DEADLINES

Part I: MDA-EFSM

MDA-EFSM model report for the gas pump components should contain:

- A class diagram
- A list of meta events for the MDA-EFSM
- A list of meta actions for the MDA-EFSM, where the responsibility of each action must be described
- A state diagram/model of the MDA-EFSM
- Pseudo-code of all operations of the Input Processors of *GP-1* and *GP-2*

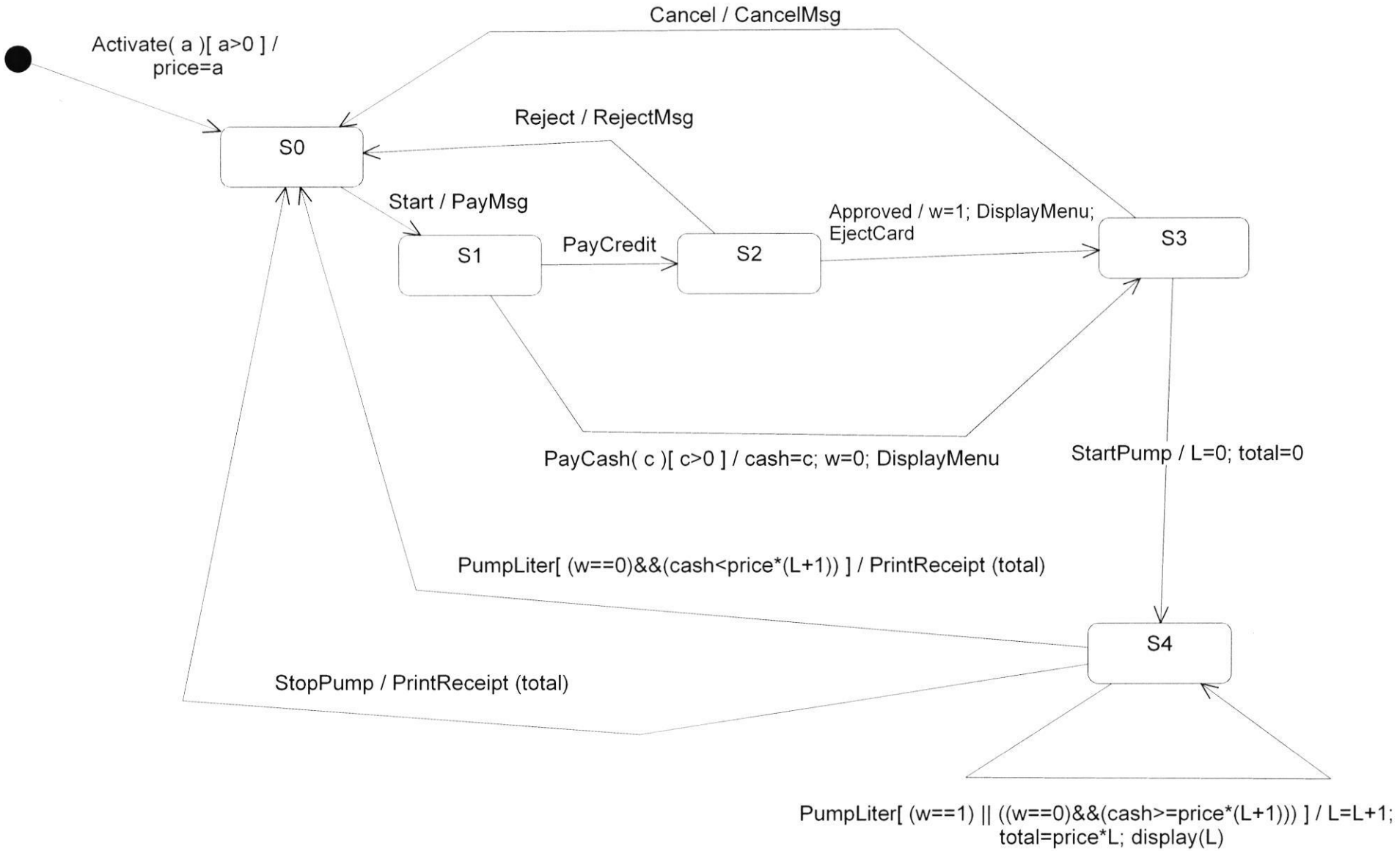


Figure 1: EFSM for gas pump GP-1

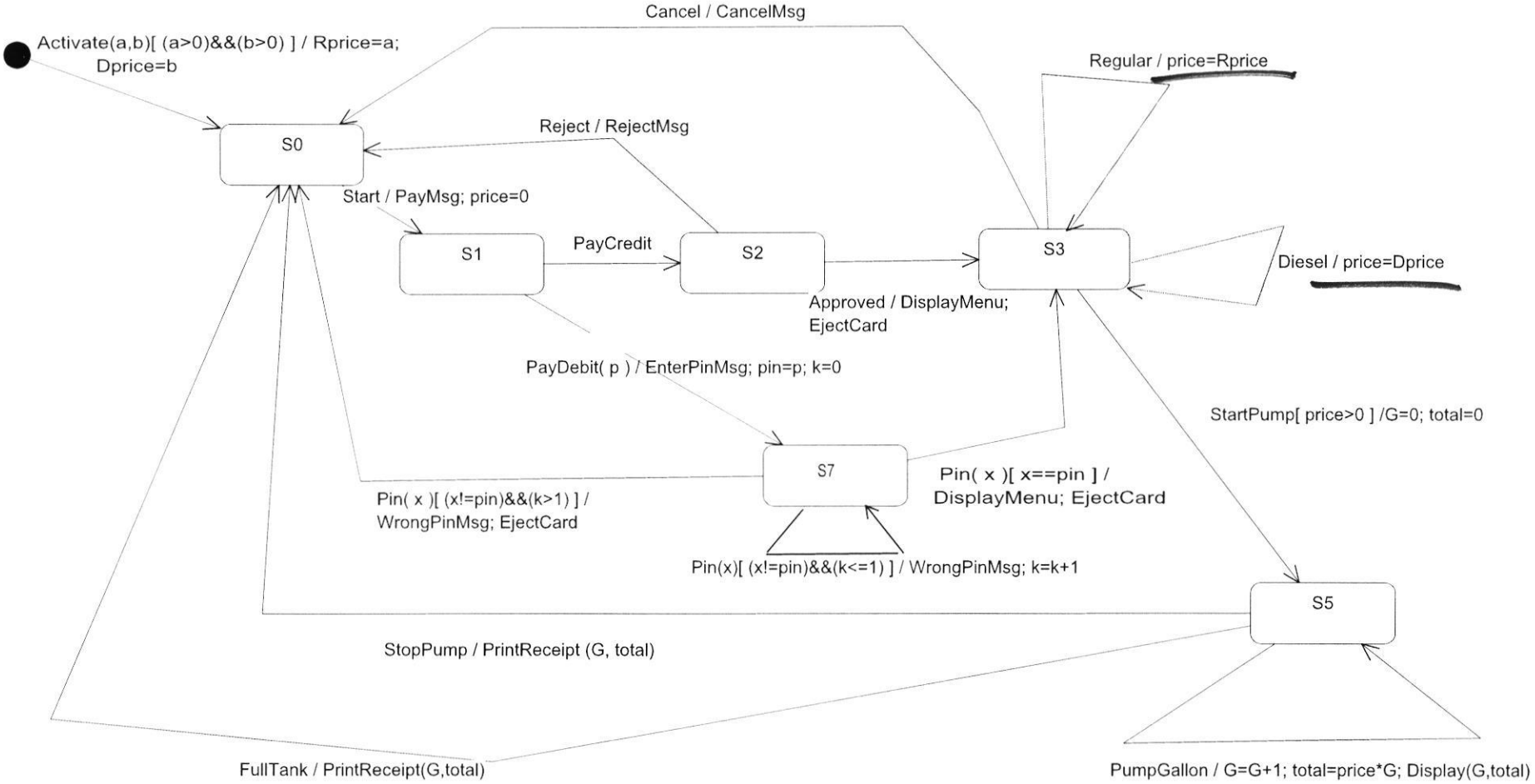
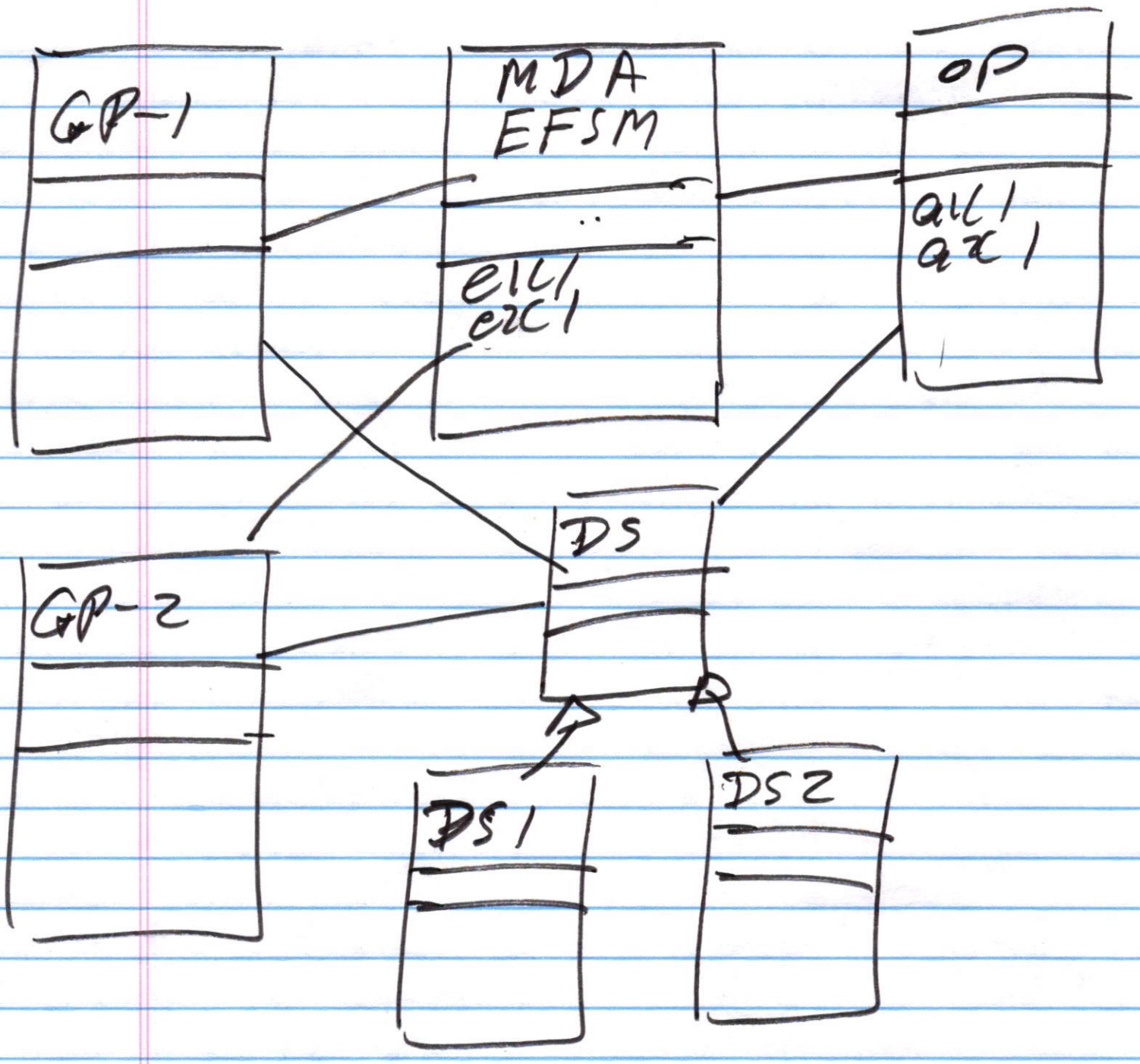


Figure 2: EFSM for gas pump GP-2

IP



GP-1

objects: GP-1, MDA-EFSM, OP, DS1

GP-2

objects: GP-2, MDA-EFSM, OP, DS2

MDA - EFSM

need to capture meta behavior of GP-1 and GP-2.

* no data or platform independent data

* platform independent meta events.

* platform independent meta actions.

Responsibilities

OP operations
GP1 or GP2 operations

* check conditions.
* store parameters in data store to be used by action operations of OP.
* invoke meta events of MDA-EFSM.

DO NOT ~~data~~ update in data store.

MDA - EFSM

- accept meta events.
- execute MDA-EFSM model.
- invoke meta actions in OP for execution.
- can have access to its own platform independent data.
- DOES NOT have access to data store.

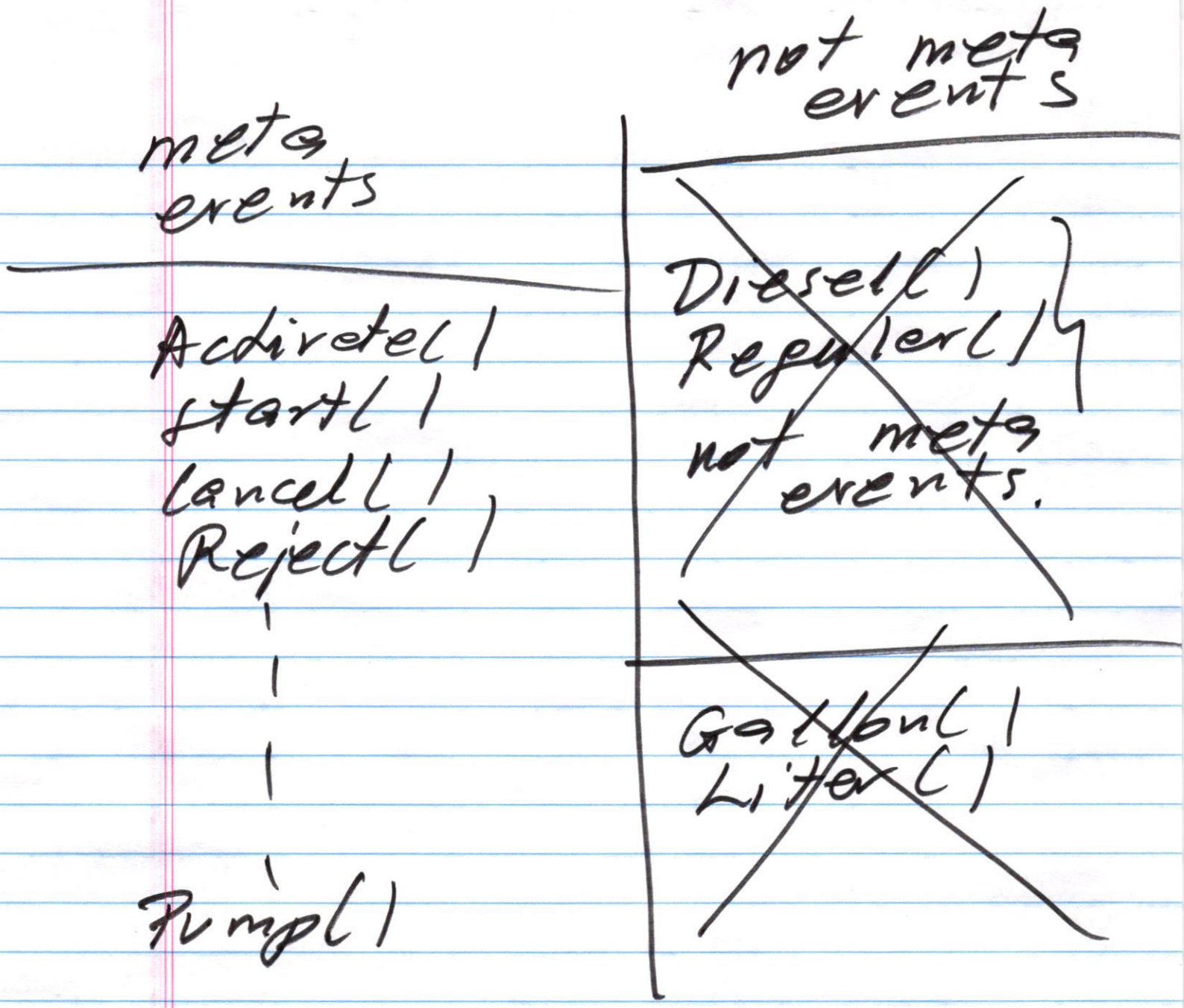
OP

* execute actions invoked by MDA-EFSM.

* manipulates (read/write) data in data store.

submission

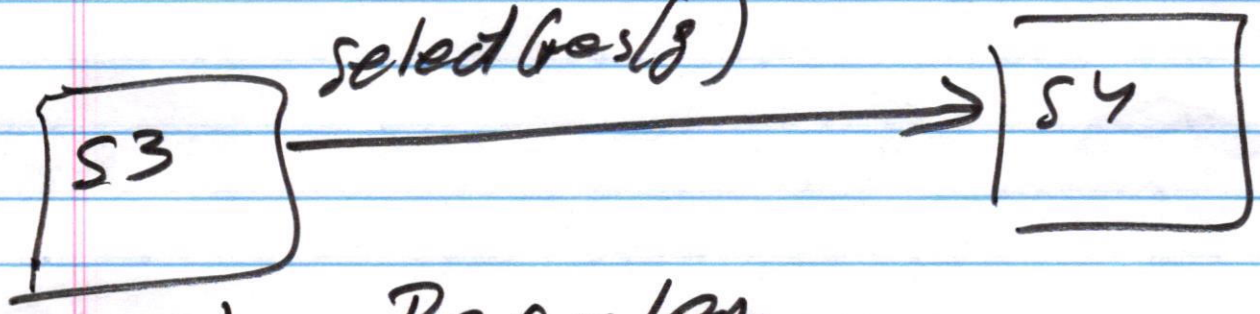
1. class diagram.
2. list of meta events
3. list of meta actions with short description
4. state diagram for MDA-EFSM.
5. pseudo code for all operations of GP-1 and GP-2.


not meta events
meta events
Diesel()
Regular()
not meta events.
Activate()
start()
Cancel()
Reject()
Gallon()
Liter()
Pump()

~~Regular()~~
~~Diesel()~~
~~super()~~
} platform dependent

meta event

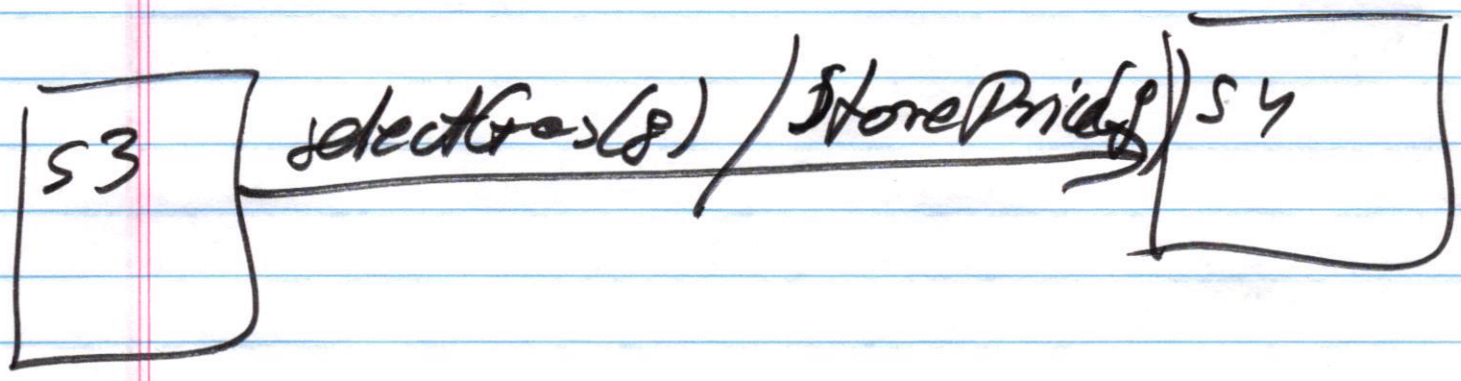
selectGas(int g)



g = 1 Regular
g = 2 Diesel
g = 3 super

meta event — meta action

selectGas(s) — storePrice(s)




```
Regular(1
} m -> selectGas(1)
}
} Diesel(2)
} m -> selectGas(2)
}
```

sample MDA-EFSM

There are two ATM components: ATM-1 and ATM-2.

The **ATM-1** component supports the following operations:

create()	// ATM is created
card (int x, string y)	// ATM card is inserted where *x* is a balance and *y* is a pin #
pin (string x)	// provides pin #
deposit (int d);	// deposit amount *d*
withdraw (int w);	// withdraw amount *w*
balance ();	// display the current balance
lock(string x)	// lock the ATM, where x is a pin #
unlock(string x)	// unlock the ATM, where x is pin #
exit()	// exit from the ATM

The **ATM-2** component supports the following operations:

create()	// ATM is created
CARD (float x, int y)	// ATM card is inserted where *x* is a balance and *y* is a pin #
PIN (int x)	// provides pin #
DEPOSIT (float d);	// deposit amount *d*
WITHDRAW (float w);	// withdraw amount *w*
BALANCE ();	// display the current balance
EXIT()	// exit from the ATM

These ATM components are state-based components and support three types of transactions: withdrawal, deposit, and balance inquiry. Before any transaction can be performed, operation *card*(x, y) (or *CARD*(x, y)) must be issued, where x is an initial balance in the account and y is a pin used to get permission to perform transactions. Before any transaction can be performed, operation *pin(x)* (or *PIN(x)*) must be issued. The *pin(x)* (or *PIN(x)*) operation must contain the valid pin # that must be the same as the pin # provided in *card*(x, y) (or *CARD*(x, y)) operation. There is a limit on the number of attempts with an invalid pin. The account can be overdrawn (below minimum balance), but a penalty may apply. If the balance is below the minimum balance then the withdrawal transaction cannot be performed. In addition, ATM-1 component can be locked by issuing *lock(x)* operation, where x is a pin #. The ATM-1 can be unlocked by *unlock(x)* operation. The detailed behavior of ATM components is specified using EFSM. The EFSM of Figure 1 shows the detail behavior of ATM-1, and the EFSM of Figure 2 shows the detailed behavior of ATM-2. Notice that there are several differences between ATM components.

Aspects that vary between these ATM components:

a. Maximum number of times incorrect pin can be entered
b. Minimum balance
c. Display menu(s)
d. Messages, e.g., error messages, etc.
e. Penalties
f. Operation names and signatures
g. Data types
h. etc.

The goal is to design an executable meta-model, referred to as **MDA-EFSM**, for all ATM components. The MDA-EFSM should capture the "generic behavior" of these two ATM components and should be de-coupled from data and implementation details. Notice that there should be **ONLY** one MDA-EFSM for these two ATM components.

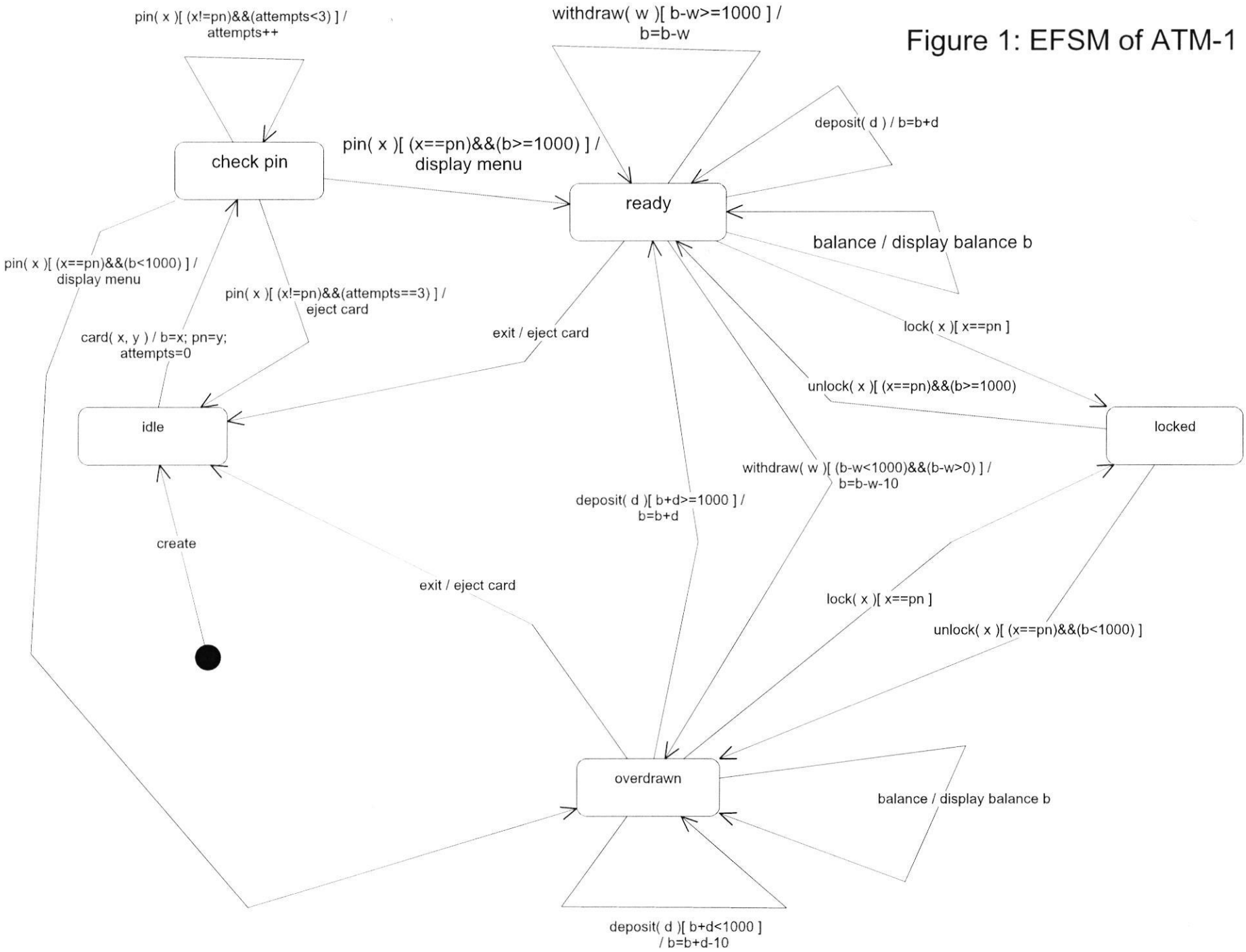


Figure 1: EFSM of ATM-1

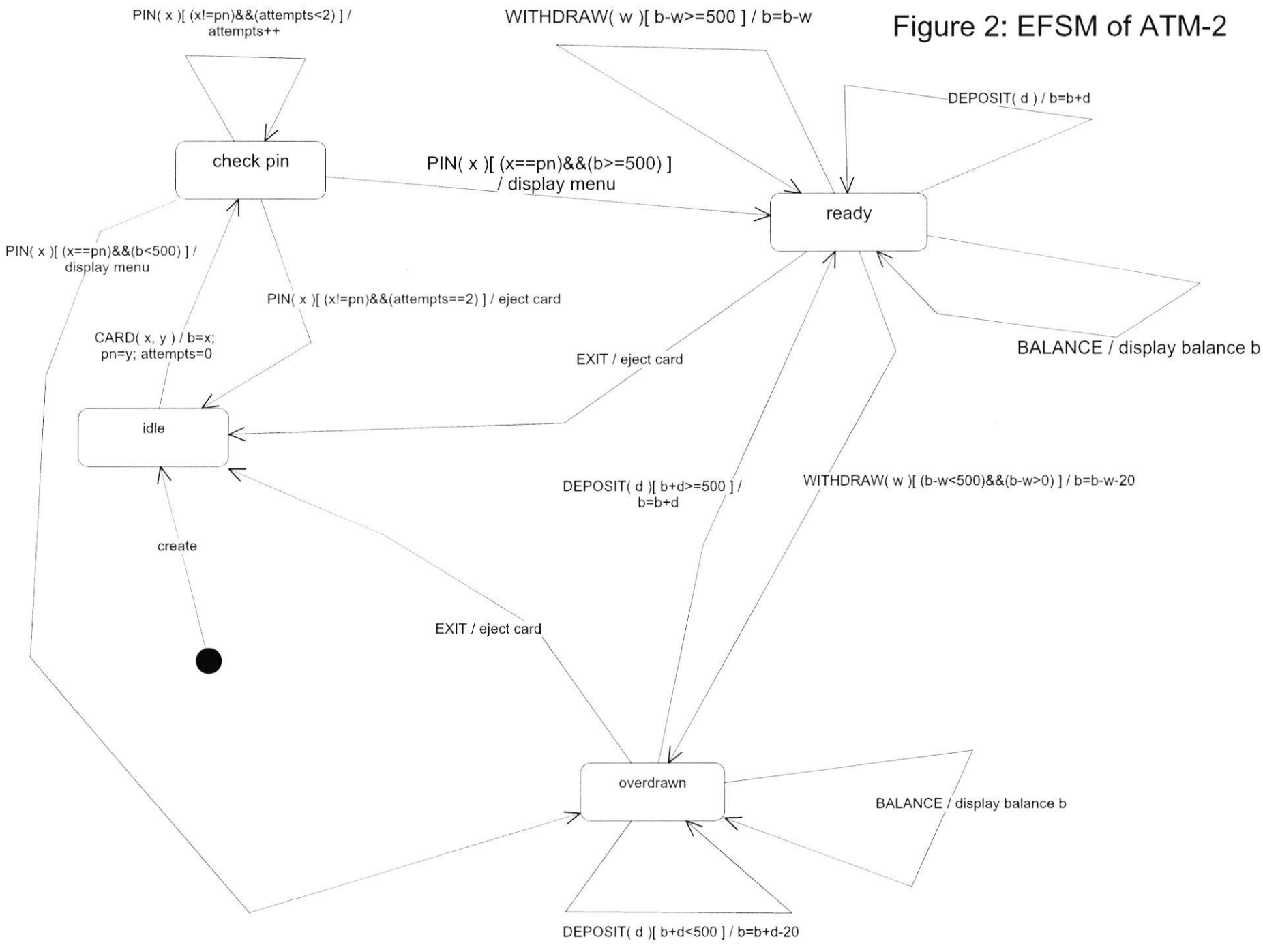
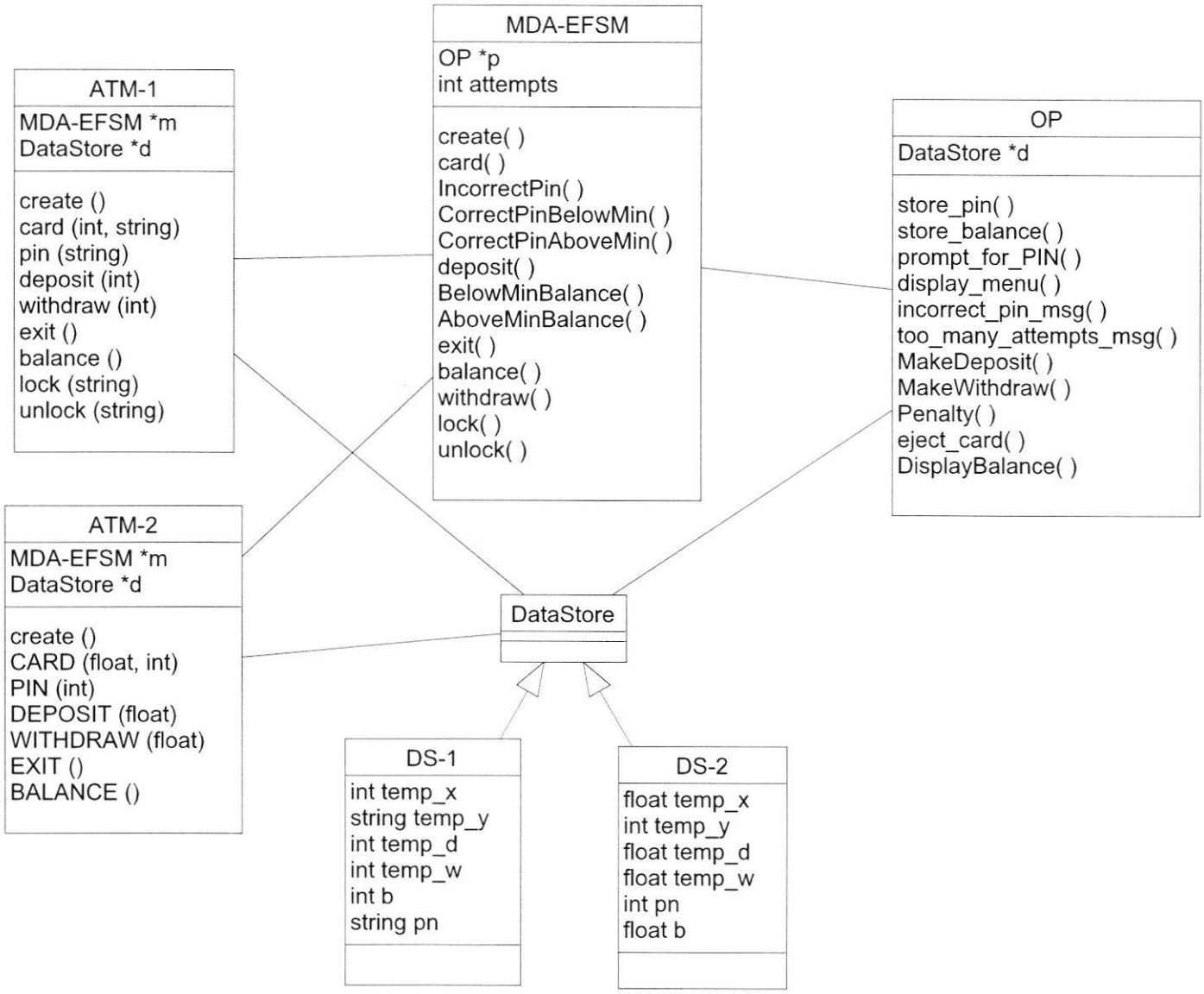
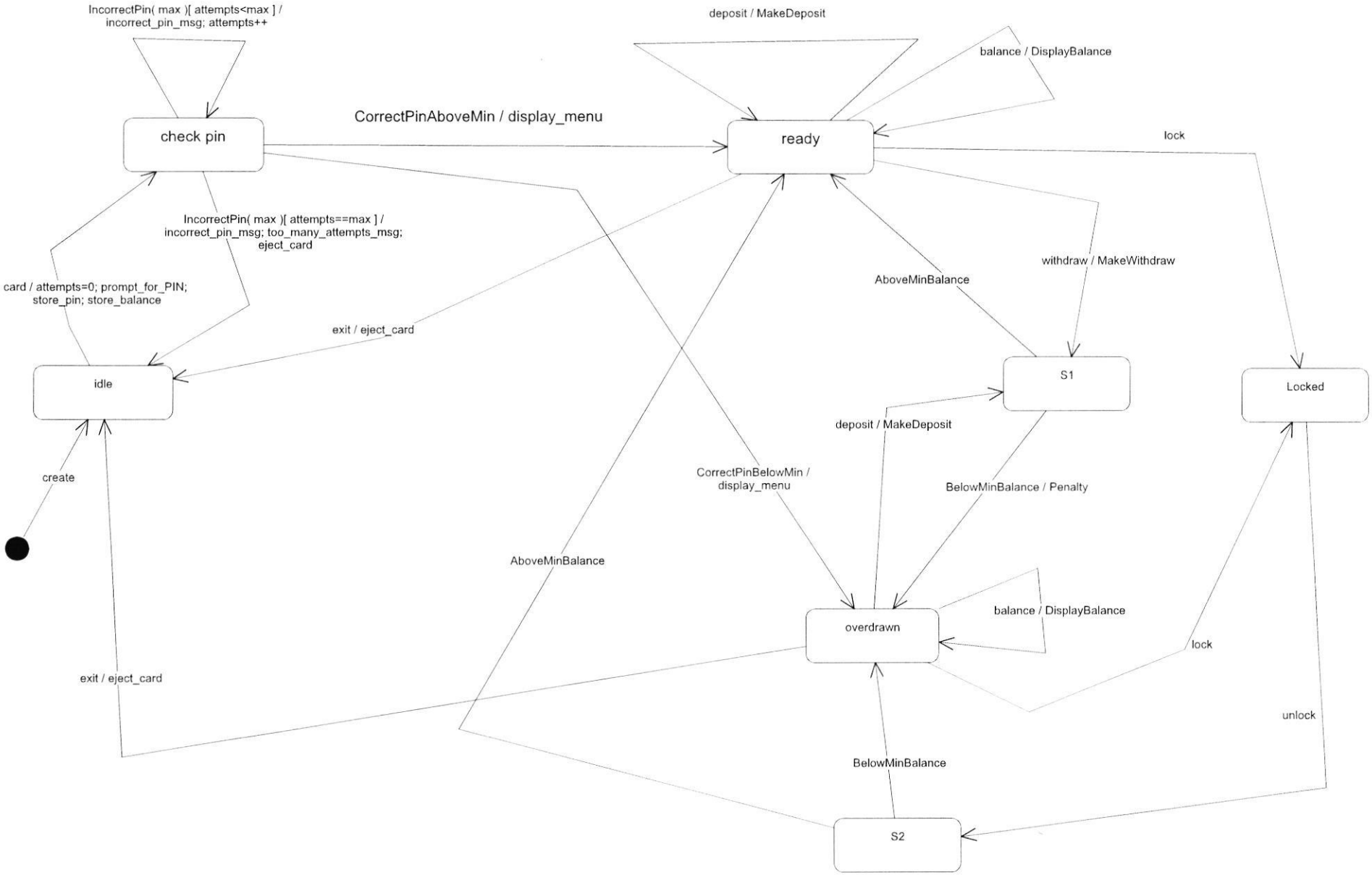


Figure 2: EFSM of ATM-2


MDA-EFSM
OP *p
int attempts
create( )
card( )
IncorrectPin( )
CorrectPinBelowMin( )
CorrectPinAboveMin( )
deposit( )
BelowMinBalance( )
AboveMinBalance( )
exit( )
balance( )
withdraw( )
lock( )
unlock( )
ATM-1
MDA-EFSM *m
DataStore *d
create ()
card (int, string)
pin (string)
deposit (int)
withdraw (int)
exit ()
balance ()
lock (string)
unlock (string)
OP
DataStore *d
store_pin( )
store_balance( )
prompt_for_PIN( )
display_menu( )
incorrect_pin_msg( )
too_many_attempts_msg( )
MakeDeposit( )
MakeWithdraw( )
Penalty( )
eject_card( )
DisplayBalance( )
ATM-2
MDA-EFSM *m
DataStore *d
create ()
CARD (float, int)
PIN (int)
DEPOSIT (float)
WITHDRAW (float)
EXIT ()
BALANCE ()
DataStore
DS-1
int temp_x
string temp_y
int temp_d
int temp_w
int b
string pn
DS-2
float temp_x
int temp_y
float temp_d
float temp_w
int pn
float b


IncorrectPin(max)[attempts<max] /
incorrect_pin_msg; attempts++
deposit / MakeDeposit
balance / DisplayBalance
CorrectPinAboveMin / display_menu
check pin
ready
lock
IncorrectPin(max)[attempts==max] /
incorrect_pin_msg; too_many_attempts_msg;
eject_card
withdraw / MakeWithdraw
AboveMinBalance
card / attempts=0; prompt_for_PIN;
store_pin; store_balance
exit / eject_card
idle
S1
Locked
deposit / MakeDeposit
CorrectPinBelowMin /
display_menu
BelowMinBalance / Penalty
create
AboveMinBalance
balance / DisplayBalance
overdrawn
lock
exit / eject_card
unlock
BelowMinBalance
S2

MDA-EFSM Events:

create()
card()
IncorectPin(int max)
CorrectPinBelowMin()
CorrectPinAboveMin()
deposit()
BelowMinBalance()
AboveMinBalance()
exit()
balance()
withdraw()
lock()
unlock

MDA-EFSM Actions:

store_pin	// stores pin from temporary data store to *pin* in data store
store_balance	// stores pin from temporary data store to *b* in data store
prompt_for_PIN	// prompts to enter pin
display_menu	// display a menu with a list of transactions
incorrect_pin_msg	// displays incorrect pin message
too_many_attempts_msg	// display too many attempts message
MakeDeposit	// makes deposit (increases balance by a value stored in temp. data store)
MakeWithdraw	// makes withdraw (decreases balance by a value stored in temp. data store)
Penalty	// applies penalty (decreases balance by the amount of penalty)
eject_card	// ejects the card
DisplayBalance	// displays the current value of the balance

Operations of the Input Processor
(ATM-1)

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

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

deposit (int d) {
        d->temp_d=d;
        m->deposit();
        if (d->b < 1000)
            m->BelowMinBalance();
        else m->AboveMinBalance();
}

withdraw (int w) {
        d->temp_w=w;
        if ((d->b-w) > 0) m->withdraw();
        if (d->b<1000)
            m->BelowMinBalance();
        else m->AboveMinBalance();
}

pin (string x) {
        if (x==d->pn) {
            if (d->b<1000)
                m->CorrectPinBelowMin ();
            else m->CorrectPinAboveMin();
        }
        else m->IncorrectPin(3)
}
```

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

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

lock (string x) {
        if (d->pn==x) m->lock();
}

unlock (string x) {
        if (x==d->pn) {
            m->unlock();
            if (d->b<1000)
                m->BelowMinBalance ();
            else m->AboveMinBalance();
        }
}
```

Notice:
m: pointer to the MDA-EFSM
d: pointer to the data store
In the data store:
b: contains the current balance
pn: contains the correct pin #

Operations of the Input Processor (ATM-2)

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

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

DEPOSIT (float d) {
        d->temp_d=d;
        m->deposit();
        if (d->b<500)
            m->BelowMinBalance();
        else m->AboveMinBalance();
}

WITHDRAW (float w) {
        d->temp_w=w;
        if ((d->b-w) > 0) m->withdraw();
        if (d->b<500)
            m->BelowMinBalance();
        else m->AboveMinBalance();
}

PIN (int x) {
        if (x==d->pn) {
            if (d->b<500)
                m->CorrectPinBelowMin ();
            else m->CorrectPinAboveMin();
        }
        else m->IncorrectPin(2)
}

EXIT() {m->exit();}

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

Notice:
m: pointer to the MDA-EFSM
d: pointer to the data store
In the data store:
b: contains the current balance
pn: contains the correct pin #