

## Exam # 2

Monday, March 25 at 5:00 pm

Closed books and notes.

The coverage is posted

---

Project Part # 1  
is posted

## COVERAGE FOR EXAM #2

CS 586; Spring 2024

Exam #2 will be held on **Monday, March 25, 2024**, at **5:00 p.m.**

Location: 104 Stuart Building

The exam is a **CLOSED books and notes** exam.

Coverage for the exam:

- OO design patterns: proxy, adapter, strategy, and abstract factory patterns.  
[Textbook: Section 3.4 (pp. 263-275); Handout #1, class notes]
- Interactive systems. Model-View-Controller architectural pattern [Textbook: Section 2.4, pp. 123-143]
- Client-Server Architecture
  - Client-Dispatcher-Server [Textbook: Section 3.6: pp. 323-337]
  - Client-Broker-Server Architecture [Textbook: Section 2.3; pp. 99-122]

### Sources:

- Textbook: F. Buschmann, et. al., Pattern-oriented software architecture, vol. I, John Wiley & Sons.
- Class notes
- Handout #1

# Pipes and Filters Architecture

---

software architecture  
↓

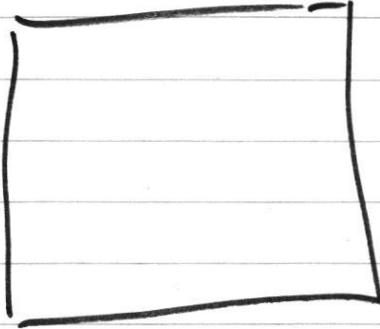
a set of components  
+  
relationships between  
components

---

components

Filters : processing  
components

Pipes : "transmit" data  
between filters.



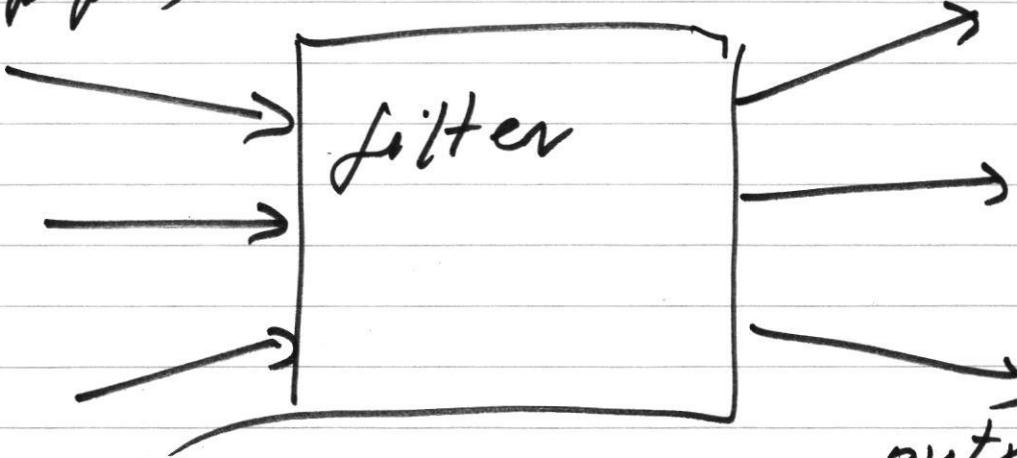
filter



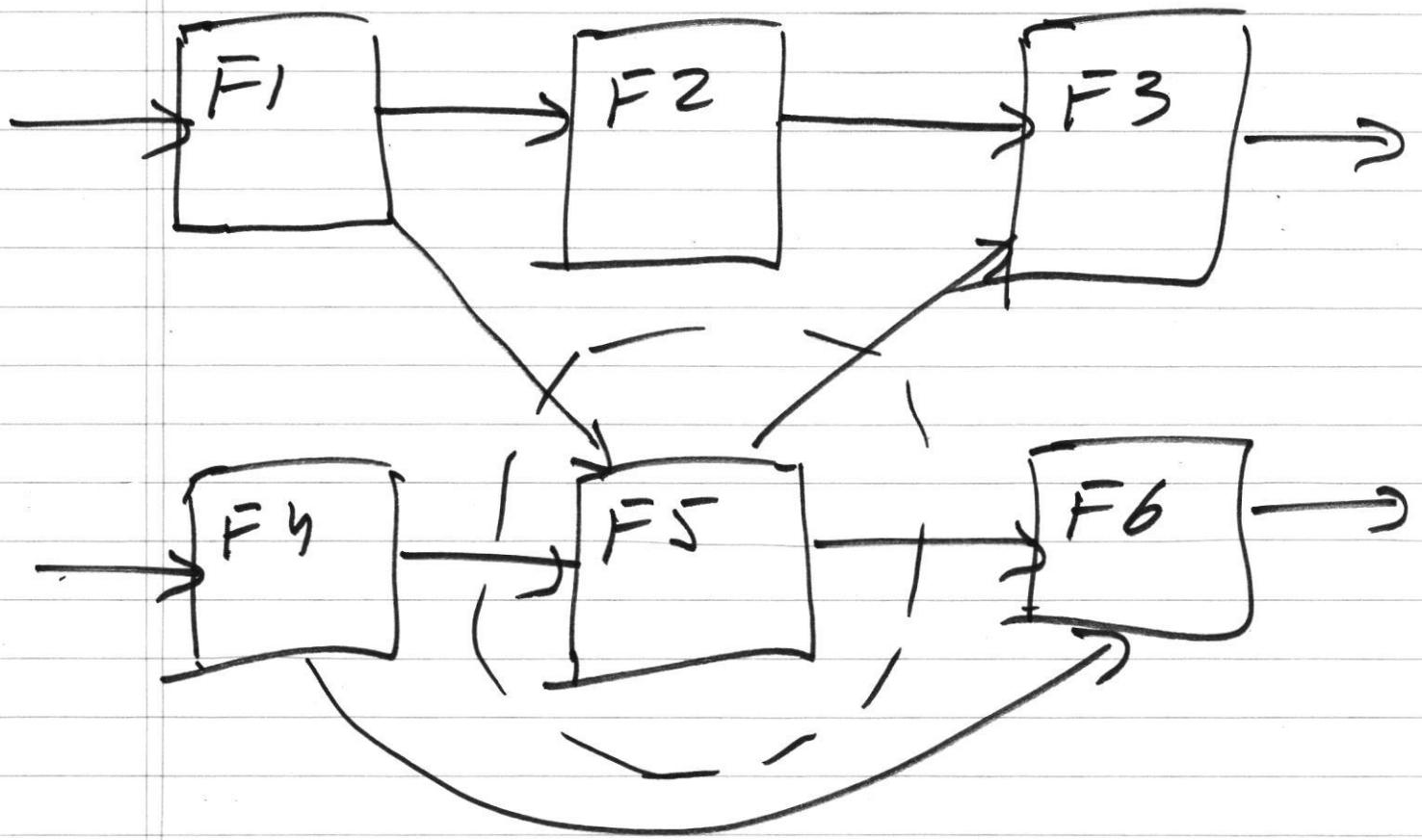
pipe



input  
pipes

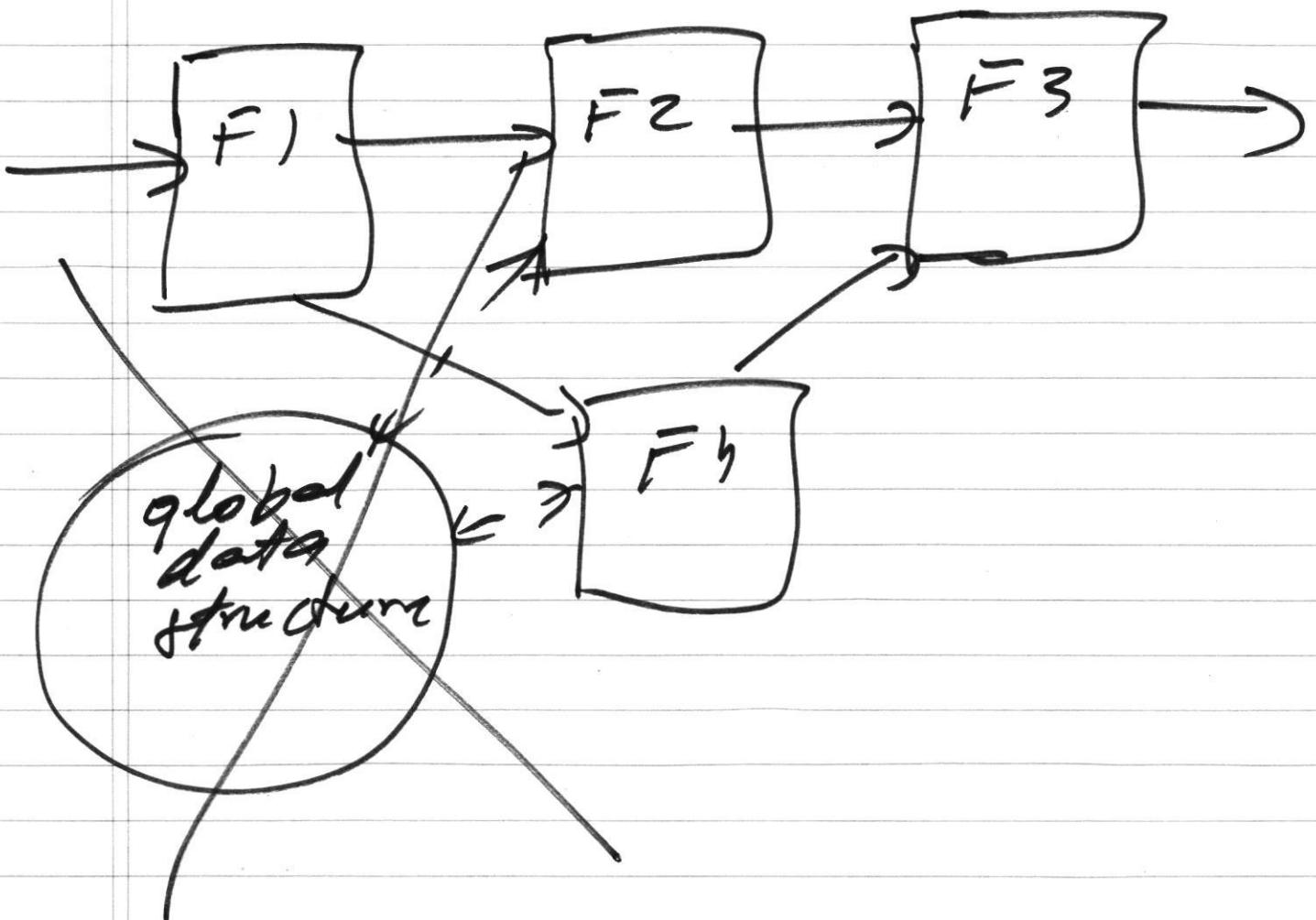


output  
pipes

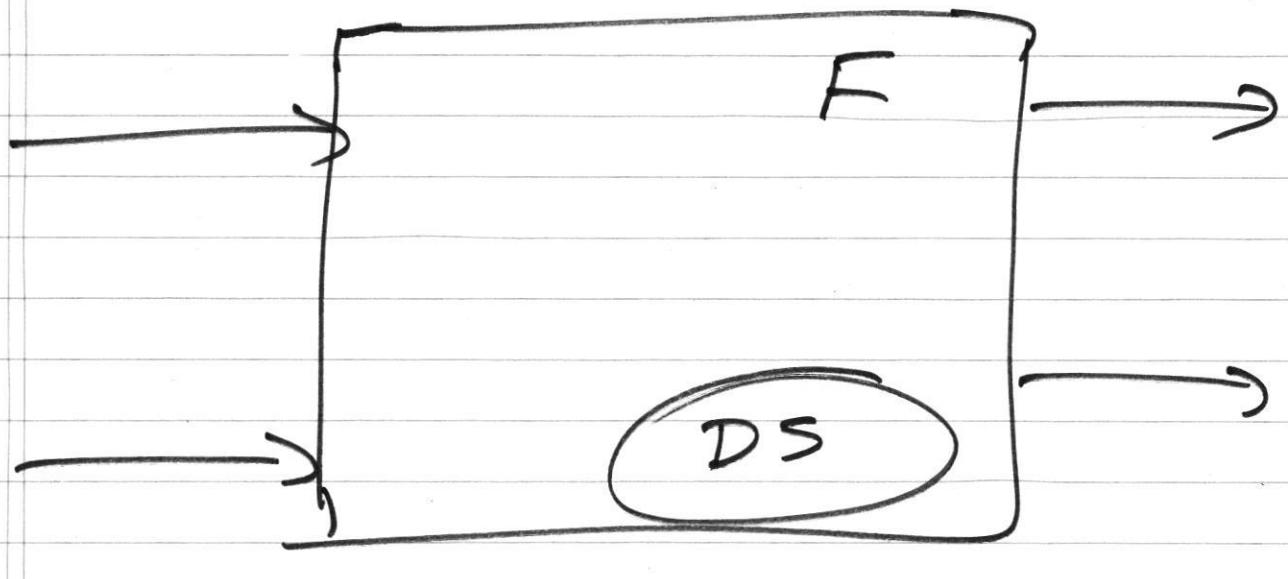


## Properties

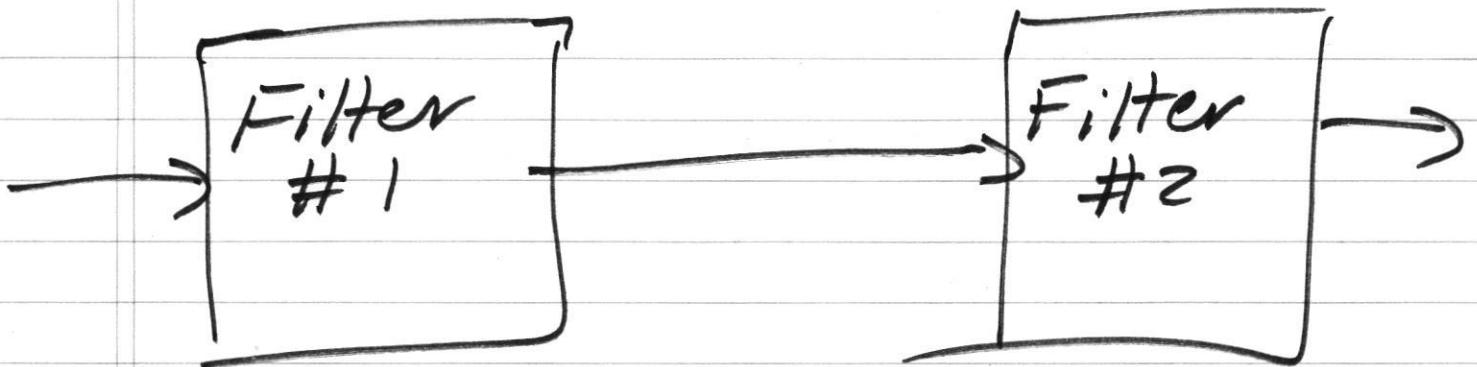
1. Filters are independent entities
2. Filters do not have access to global data structures.



3. Filters do not share  
"internal states/date"  
with other filters



## Design issues

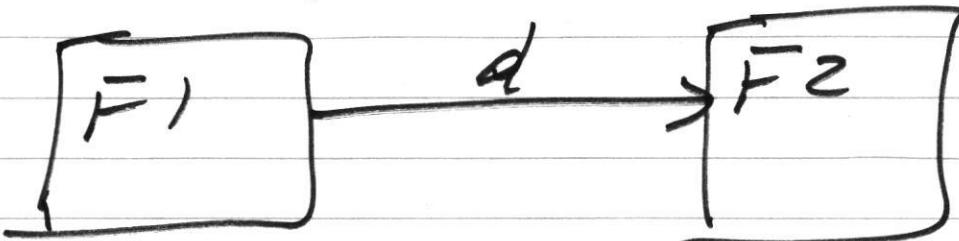


### Filters

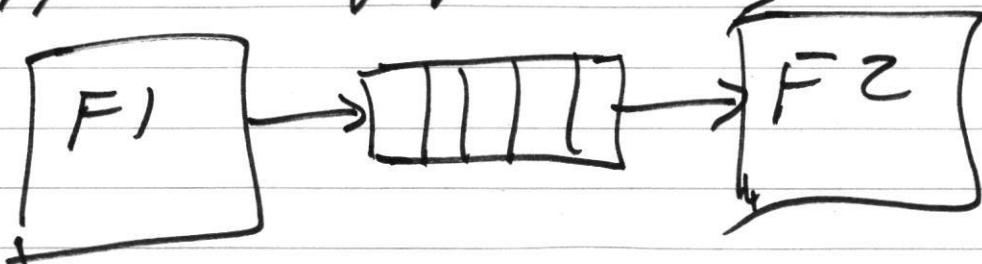
- \* passive filters
- \* active filters .

### Pipes

#### 1. unbuffered pipes

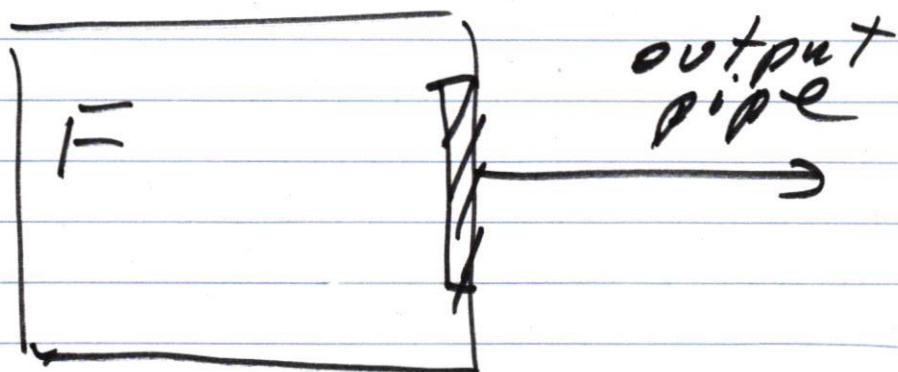


#### 2. buffered pipes



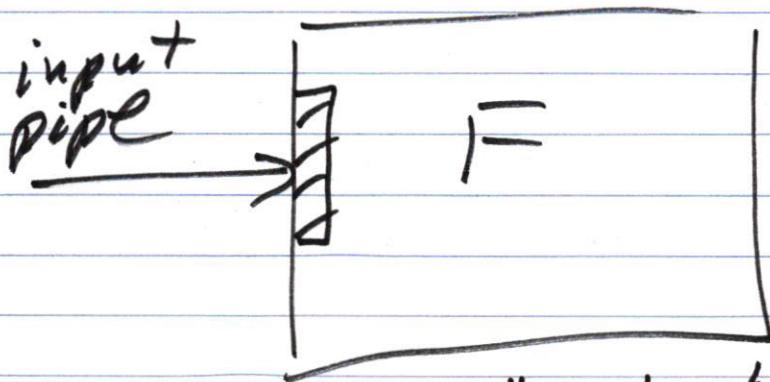
## 1. passive filters

(a) with pull-out pipe



output pipe "pulls" data  
out of filter F

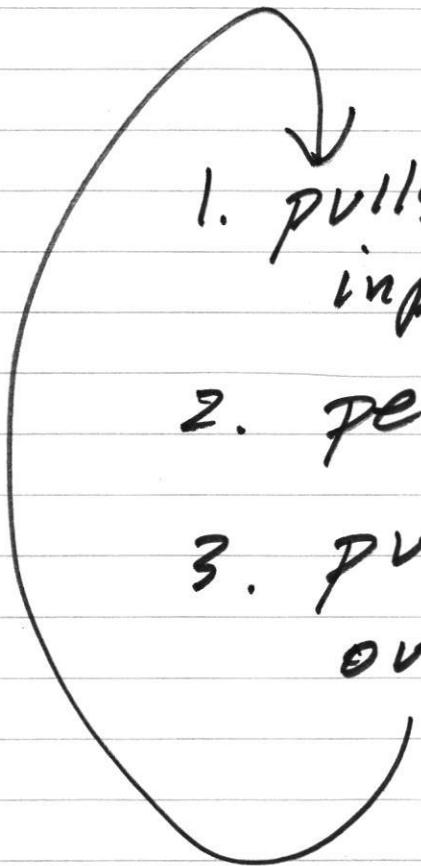
(b) with push pipe .



input pipe "pushes" data into  
Filter F .

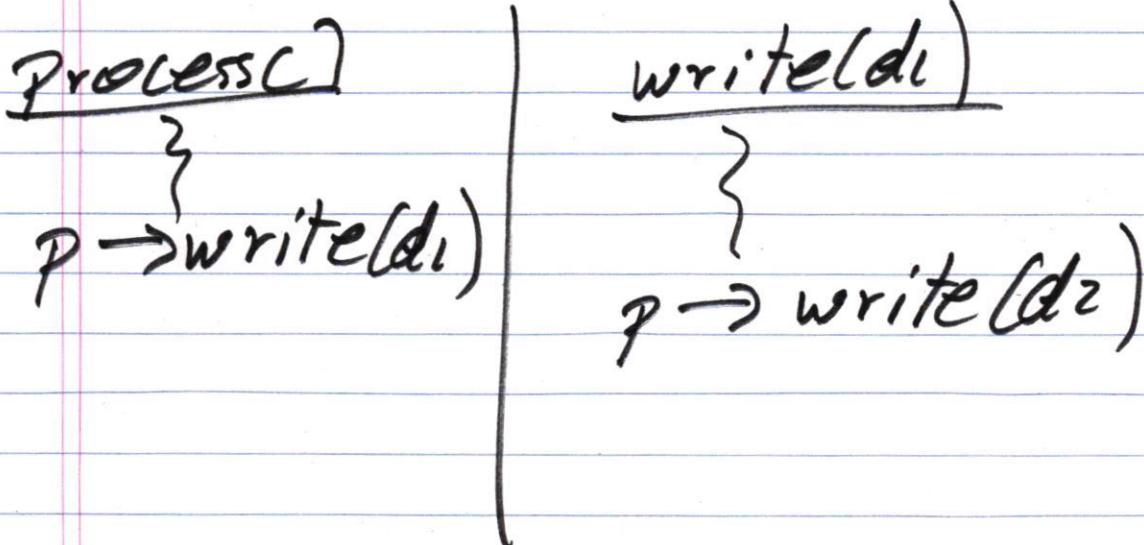
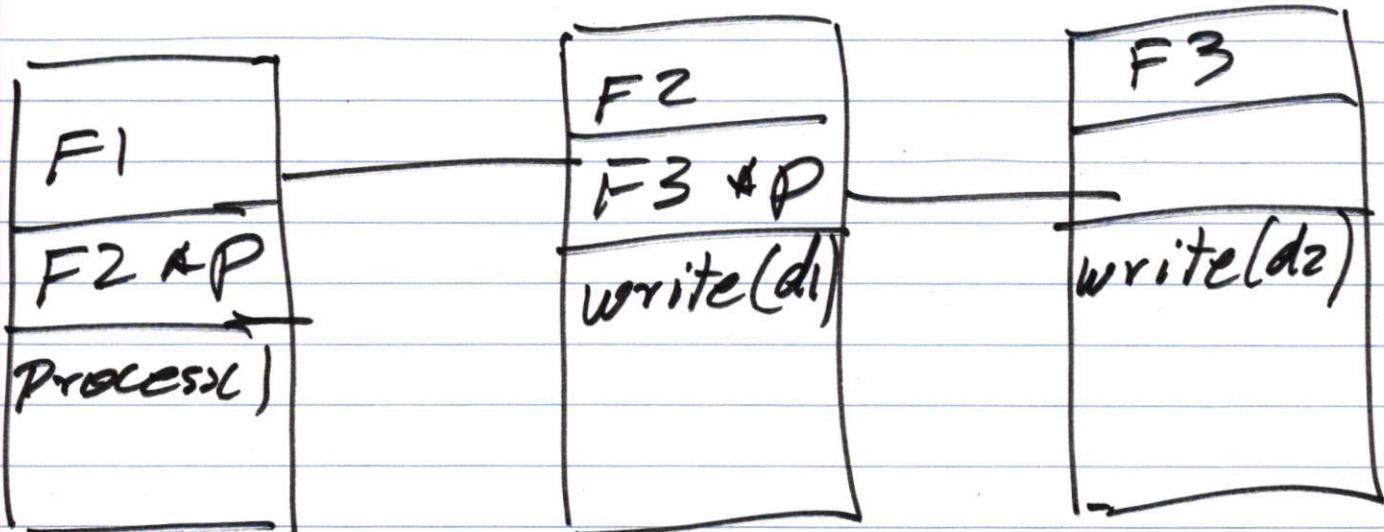
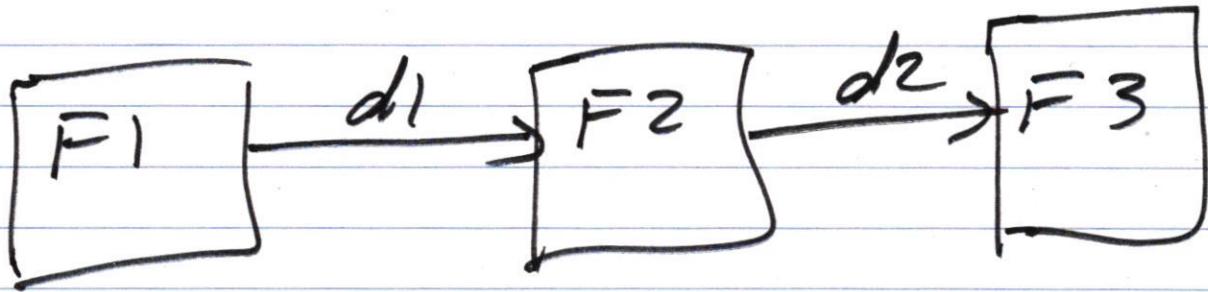
# Active filter

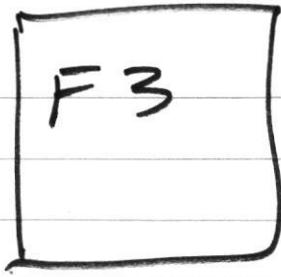
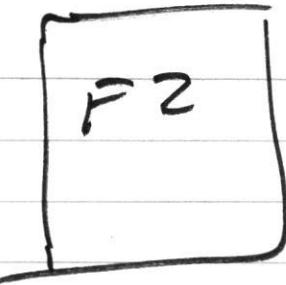
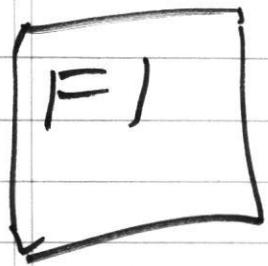
## Active loop



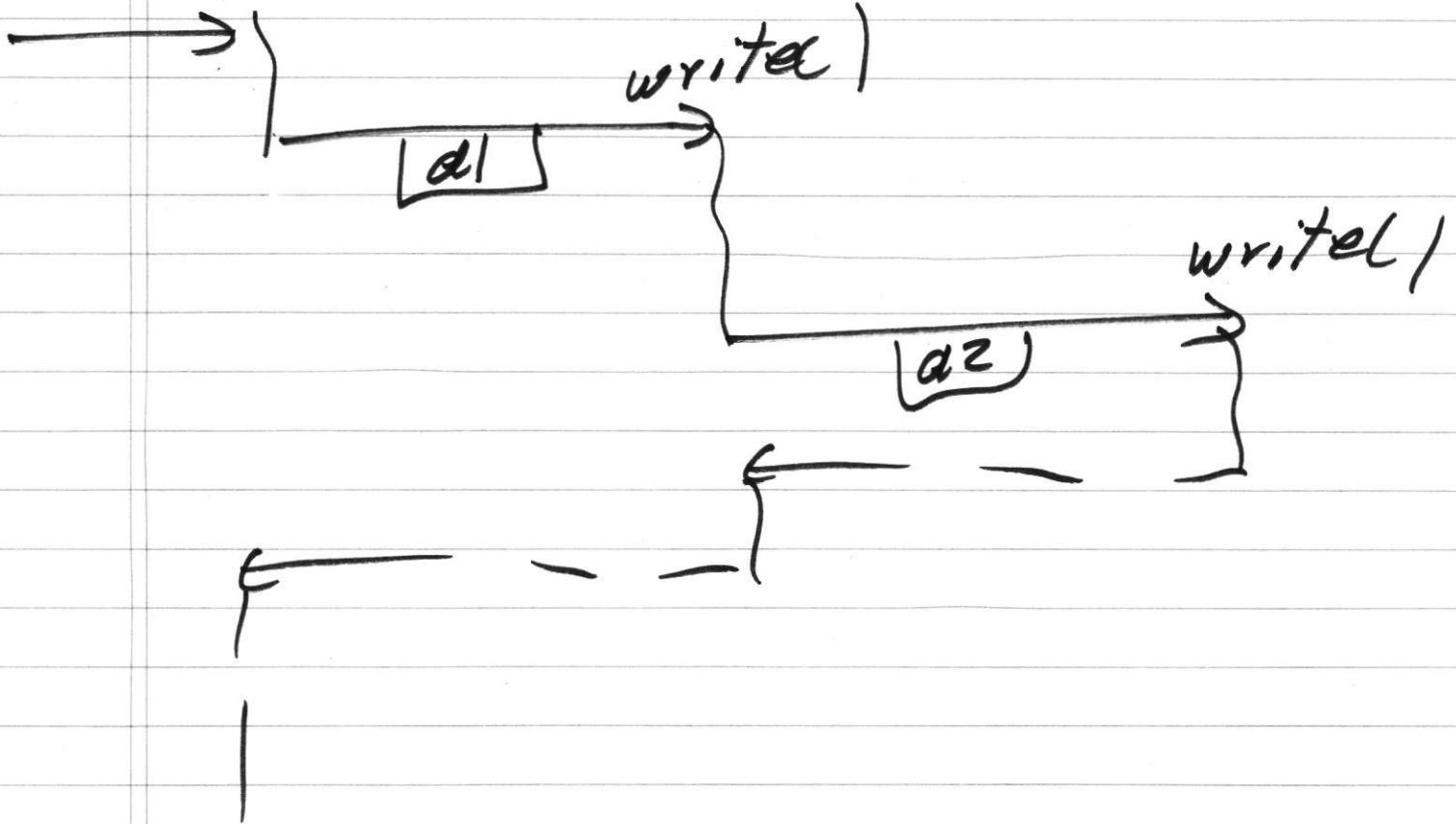
1. pulls data from input pipes
2. performs processing .
3. pushes data into output pipes

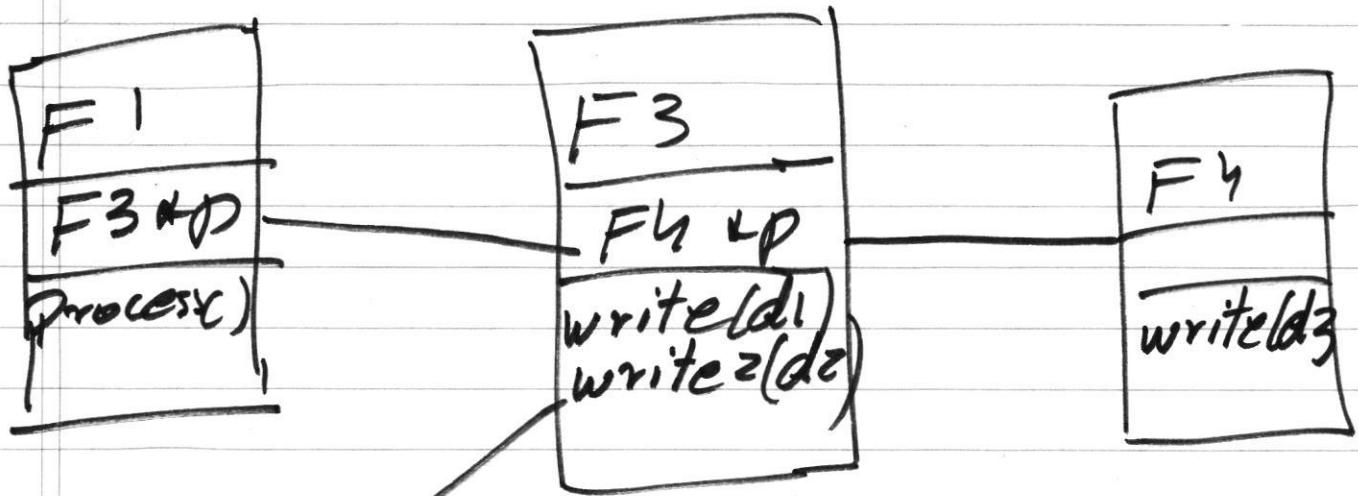
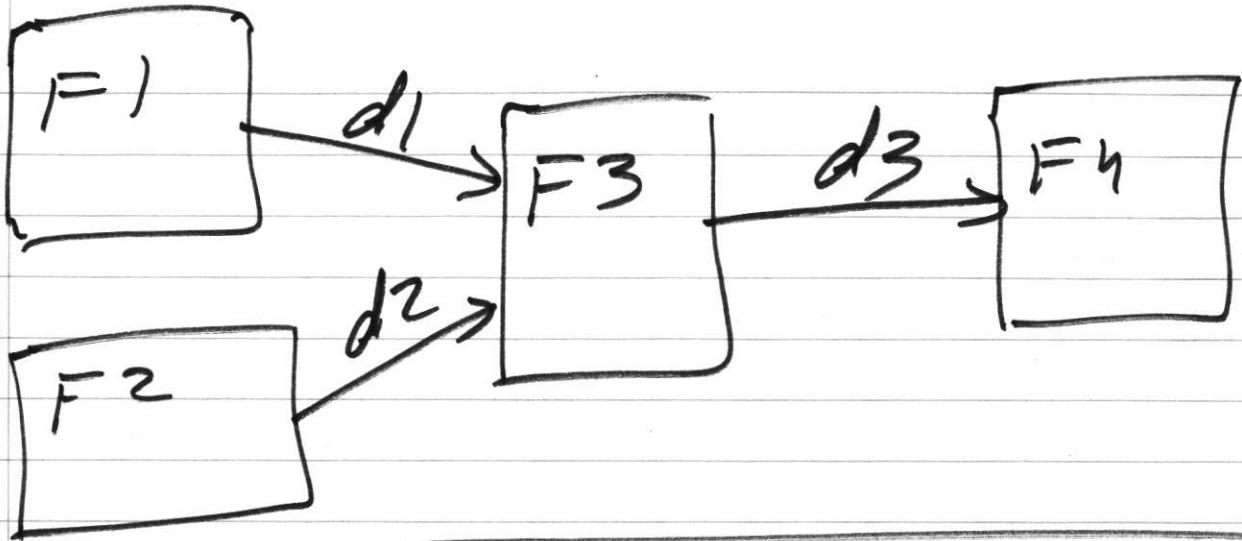
I. passive filters.  
unbuffered pipes  
push pipes.





Process()





class F1  
Process)

}  
P → write(d1)

class F2  
Process)

}  
P → write2(d2)

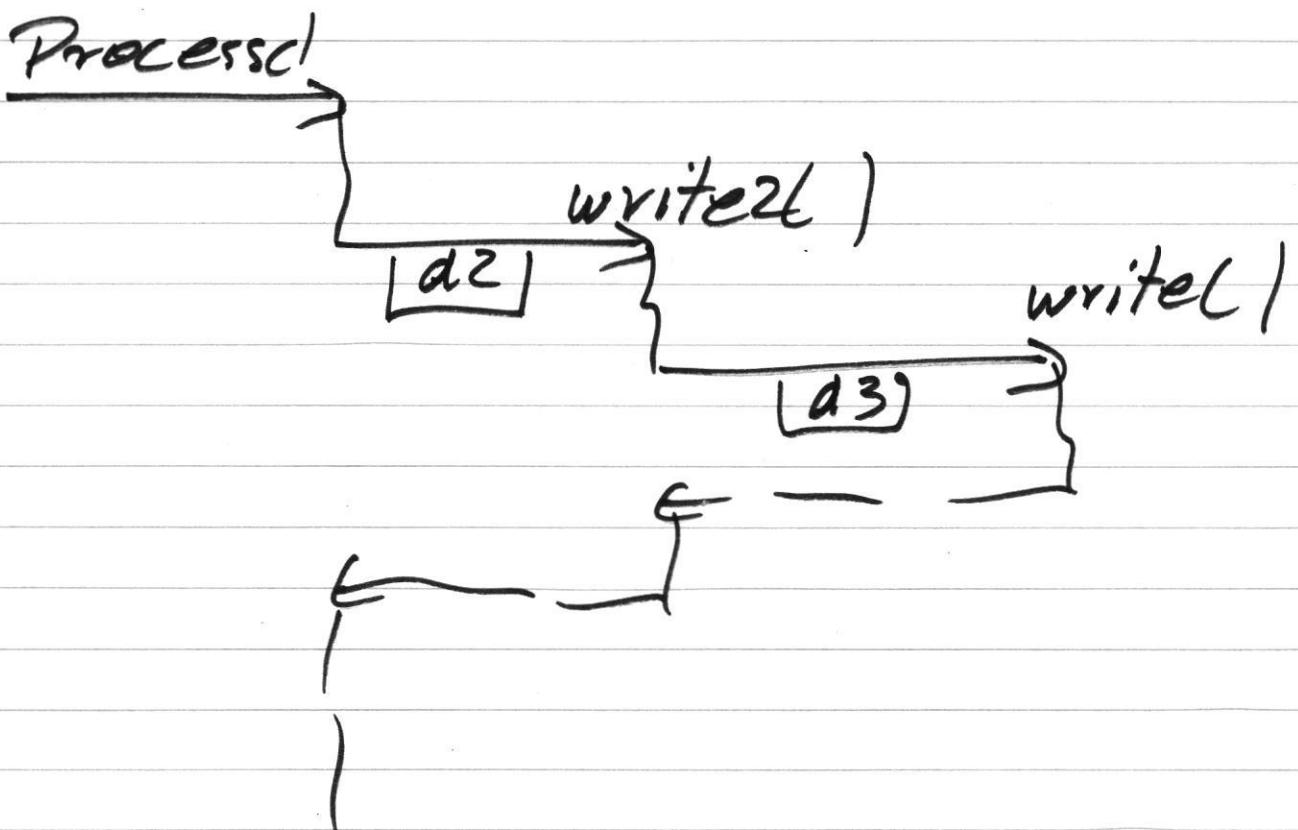
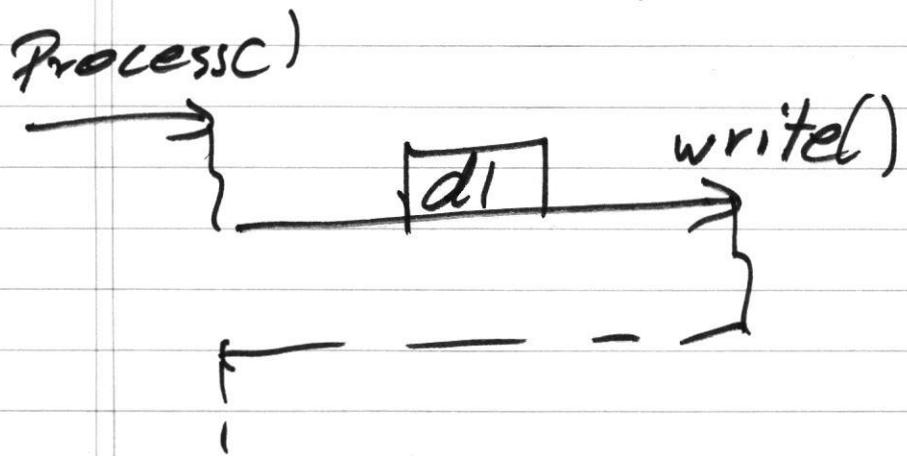
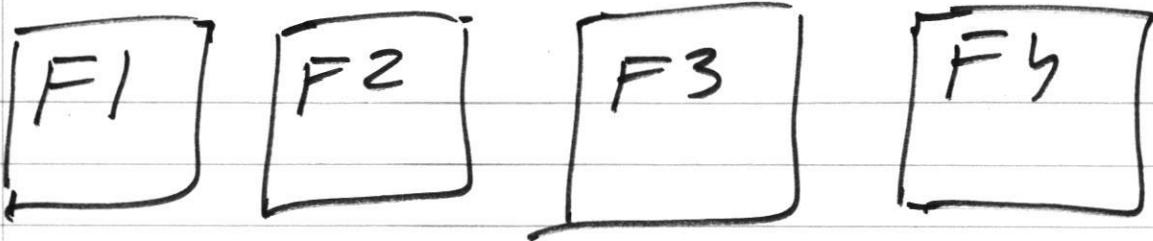
## class F3

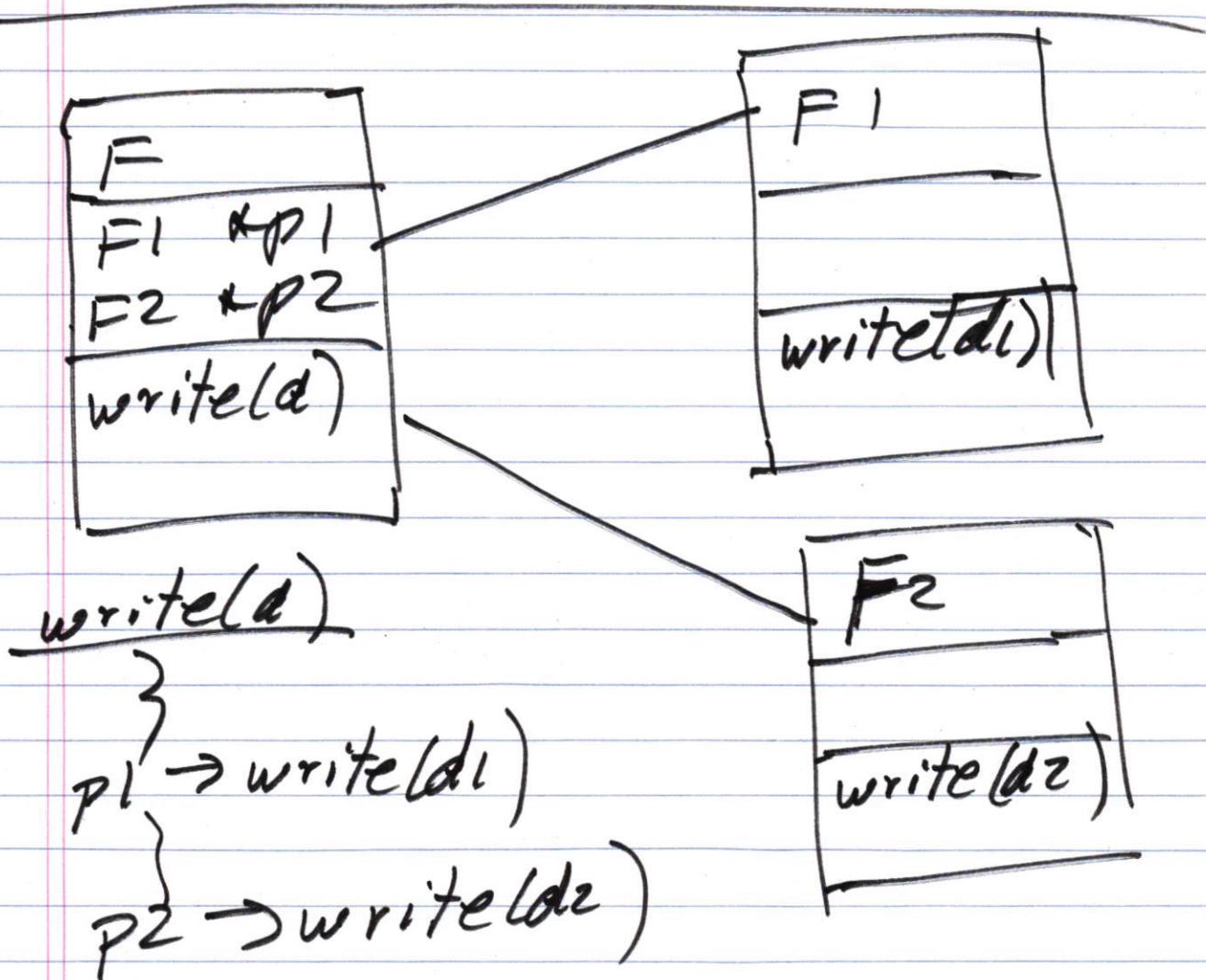
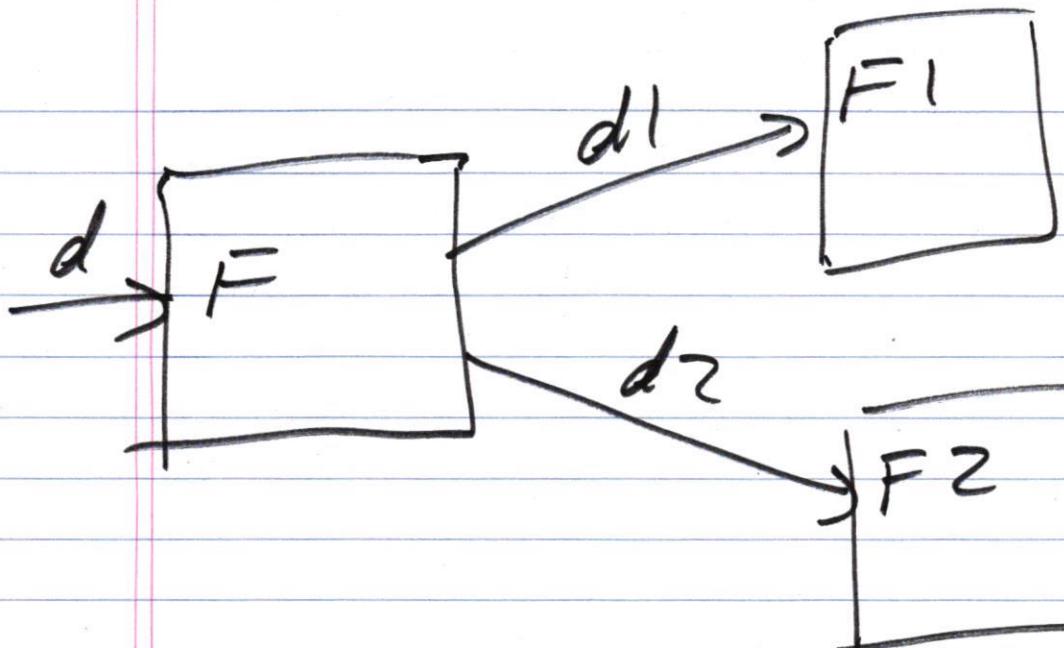
```
writel(d1)
} if (d2 is present) {
    do processing using
    d1 and d2 resulting
    in d3
    p → writeld3)
}
} else store d1
```

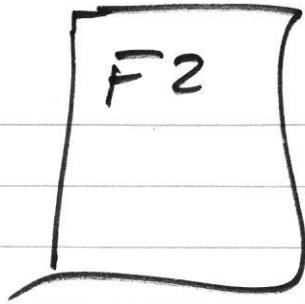
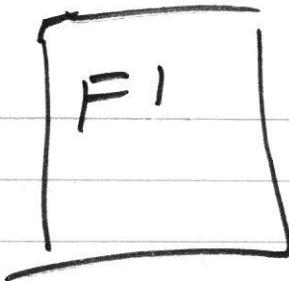
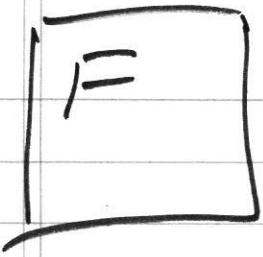
## writel(d2)

```
} if (d1 is present) {
    do processing using
    d1 and d2 resulting
    in d3
    p → writeld3)
}
} else store d2
```

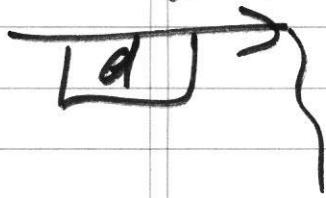
4



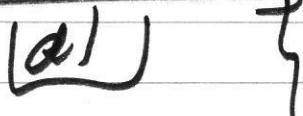




writel )



writel )

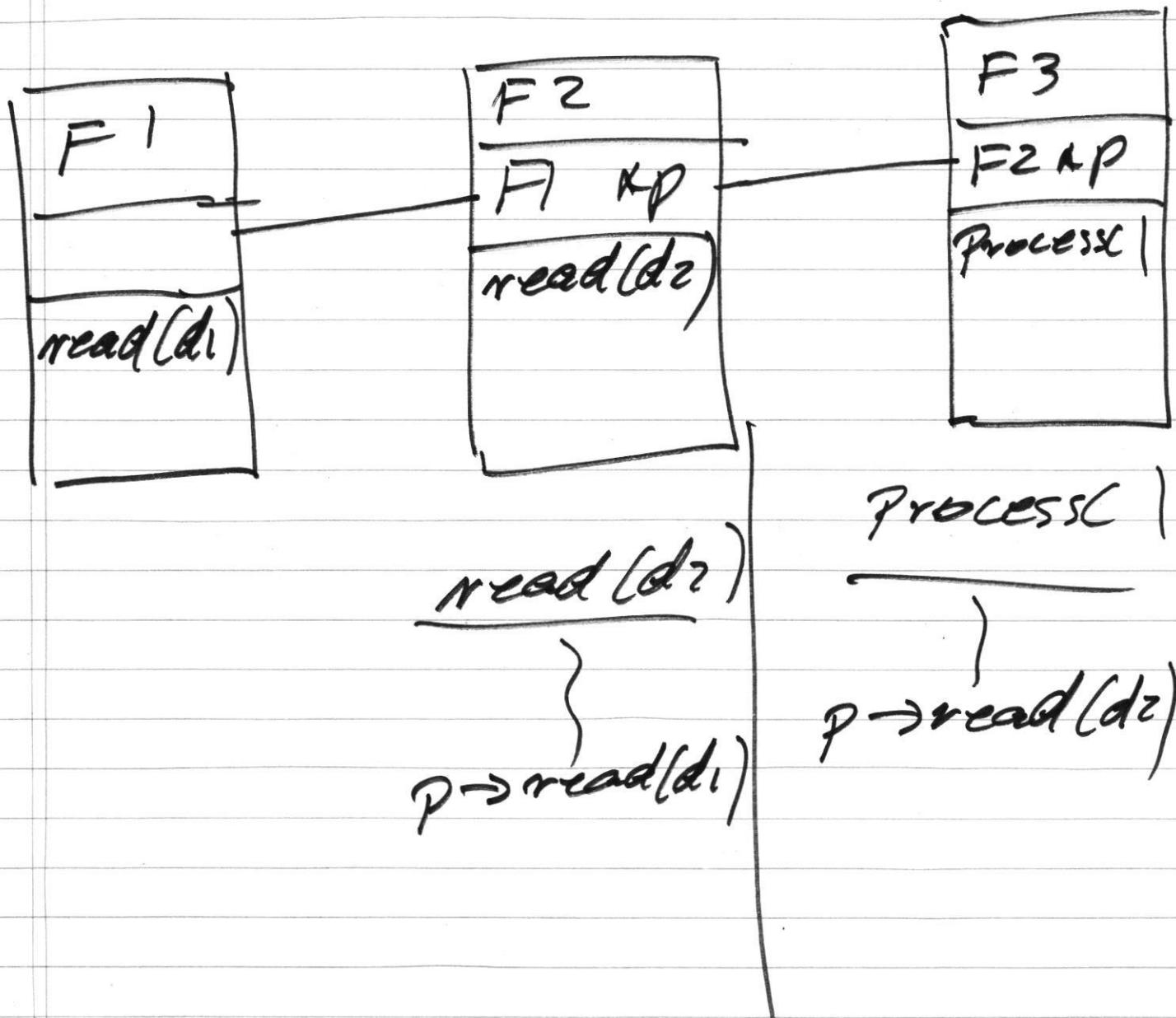
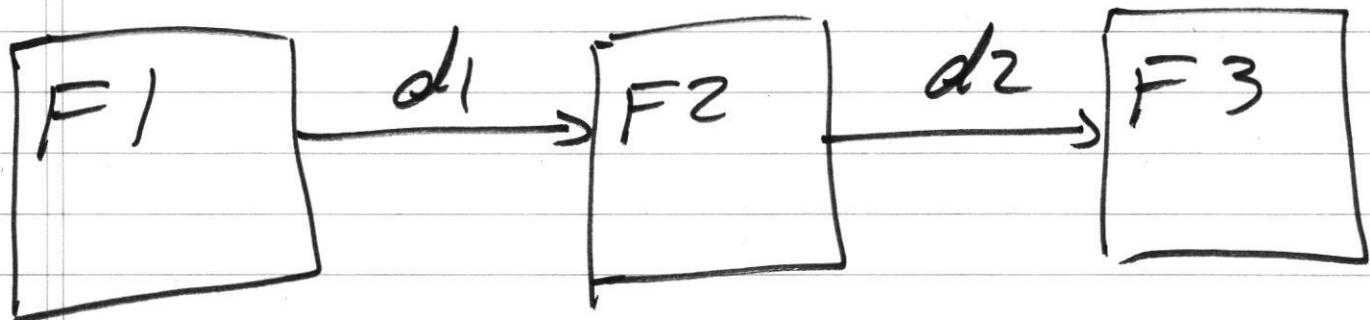


writel )



|  
|  
|

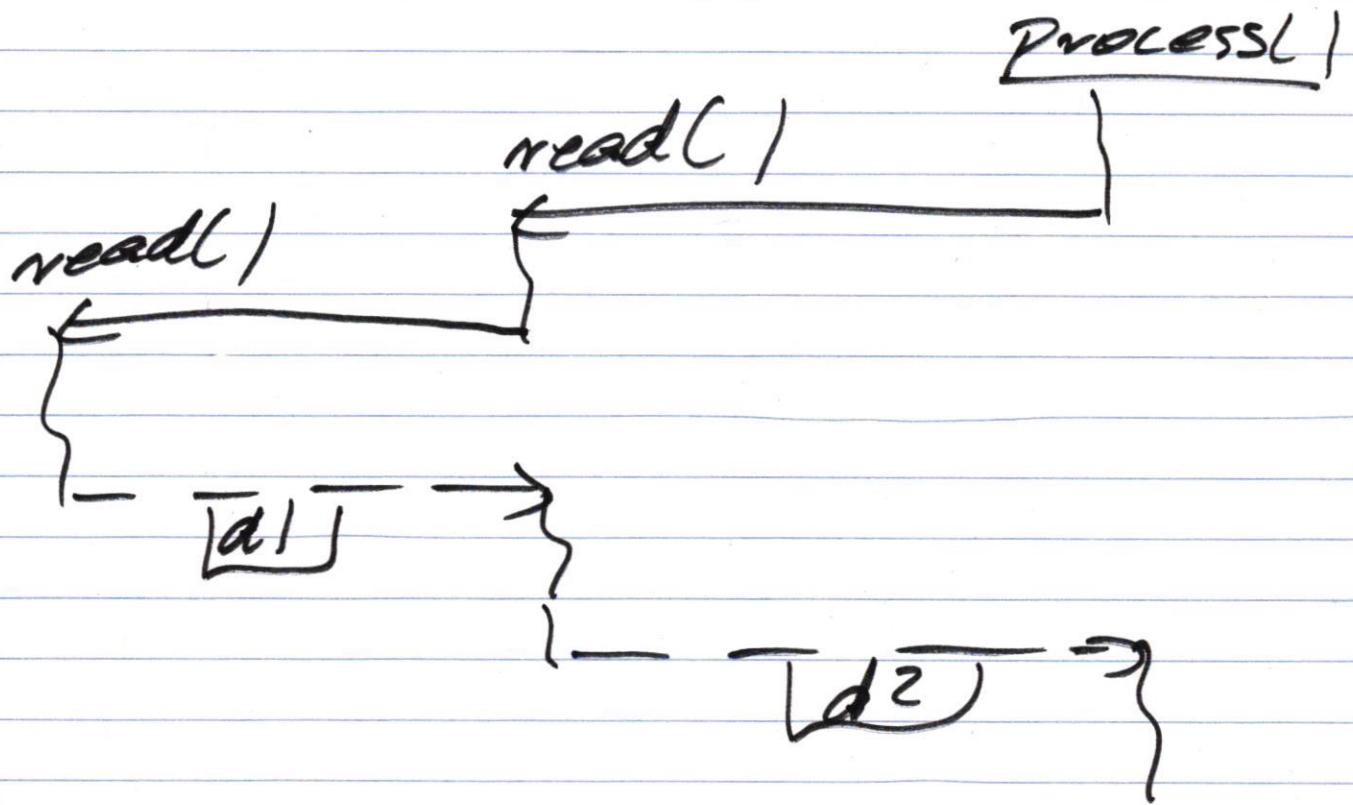
## II passive filters unbuffered pipes pull-out pipes

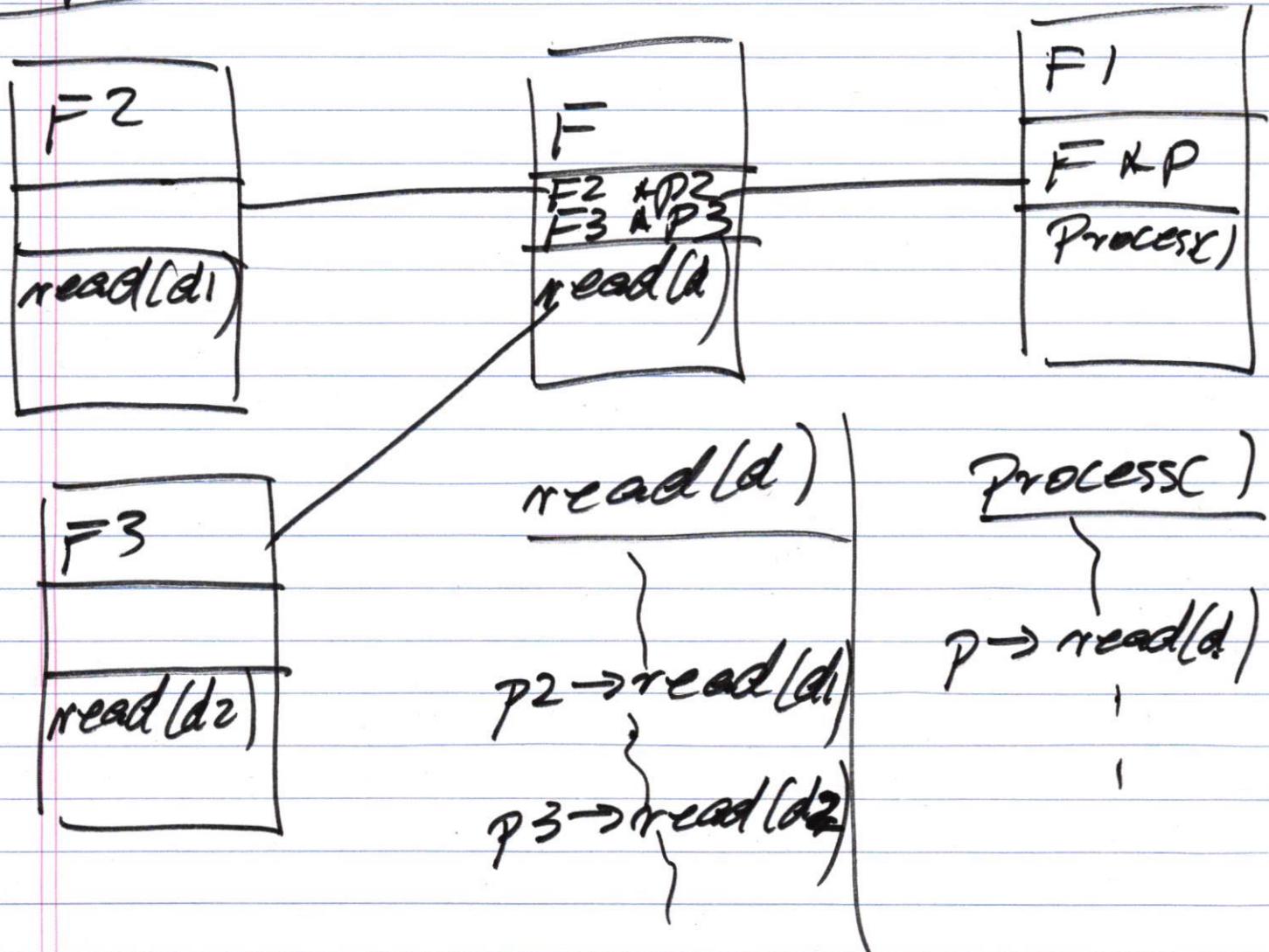
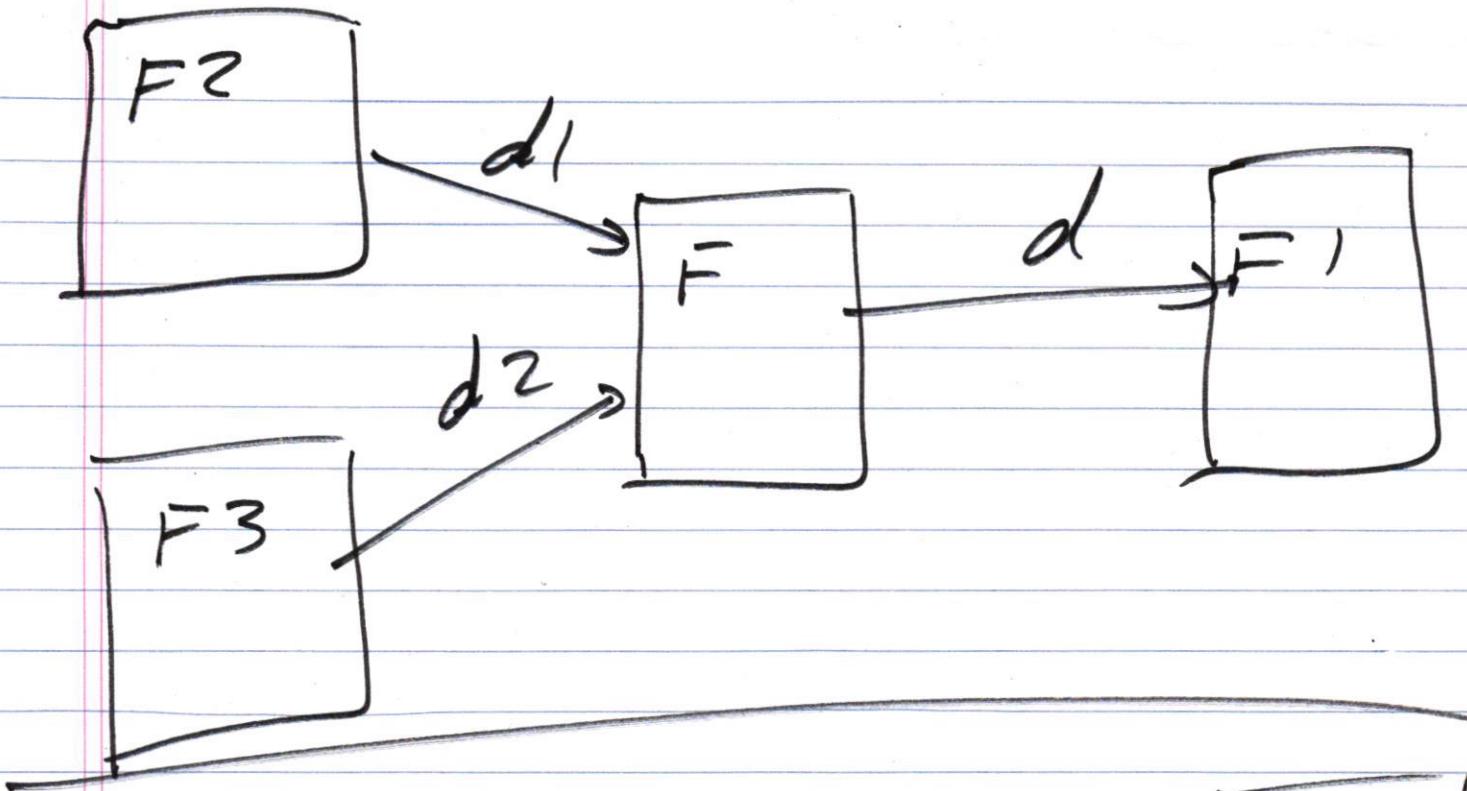


F1

F2

F3



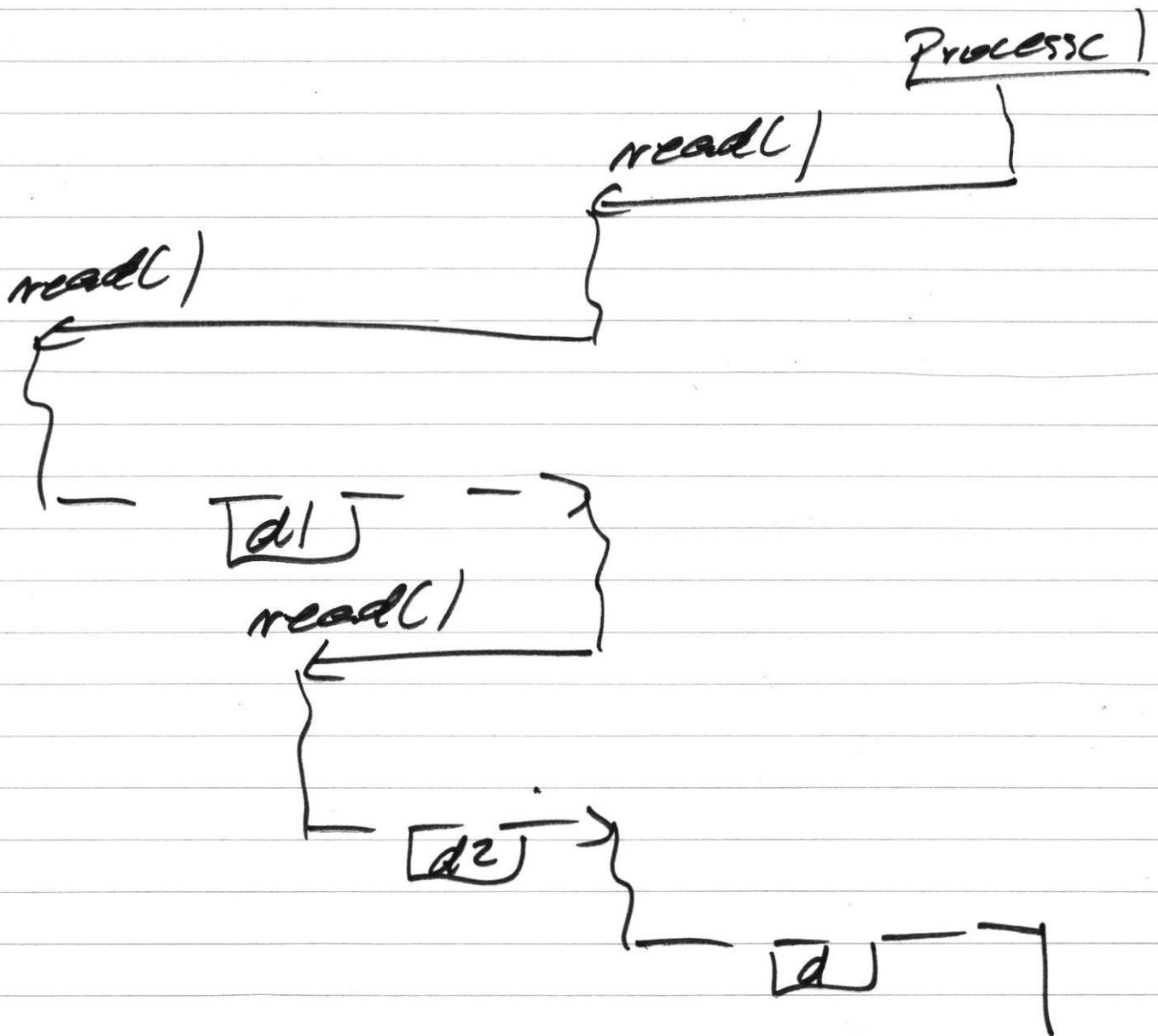


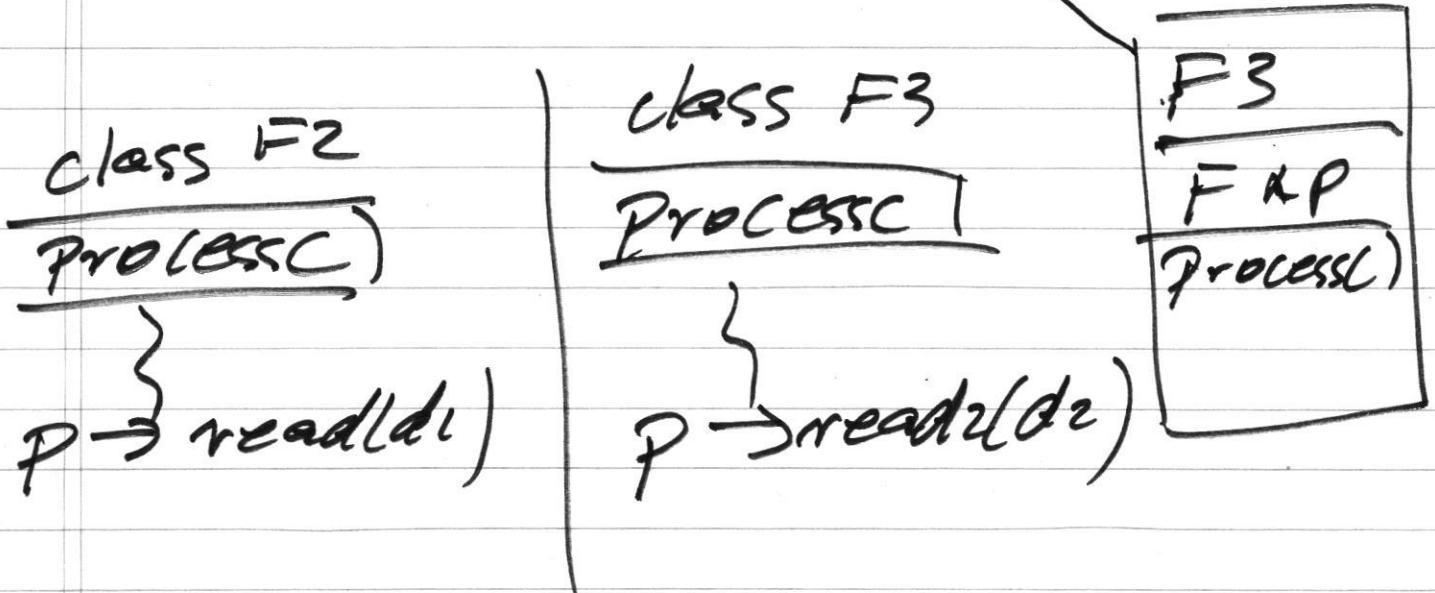
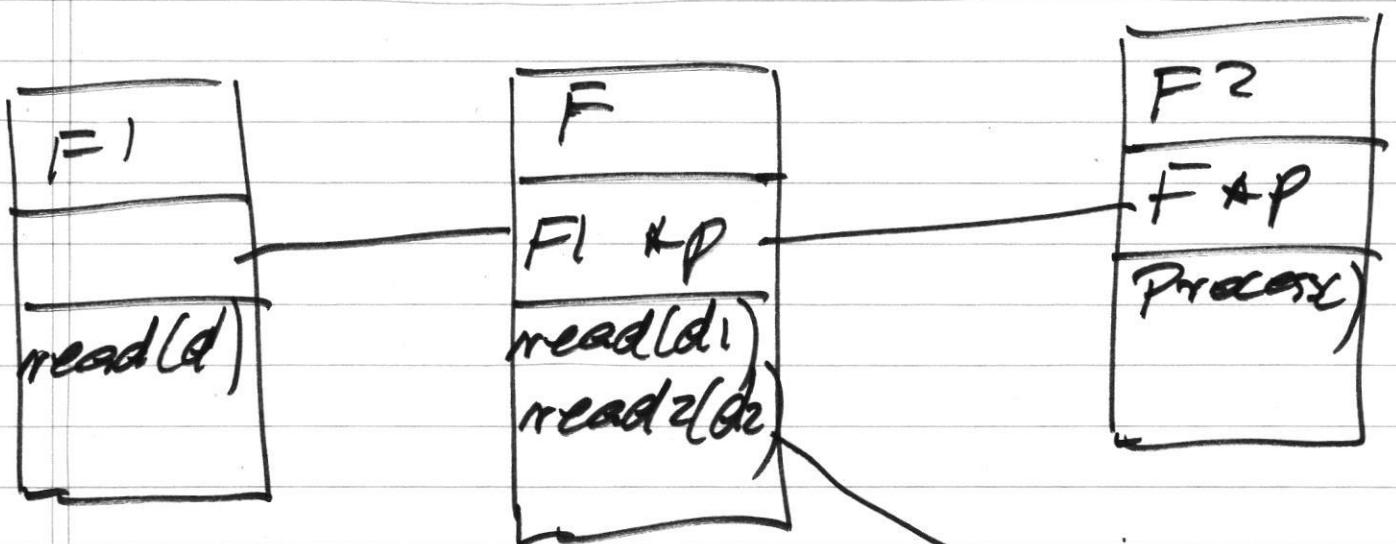
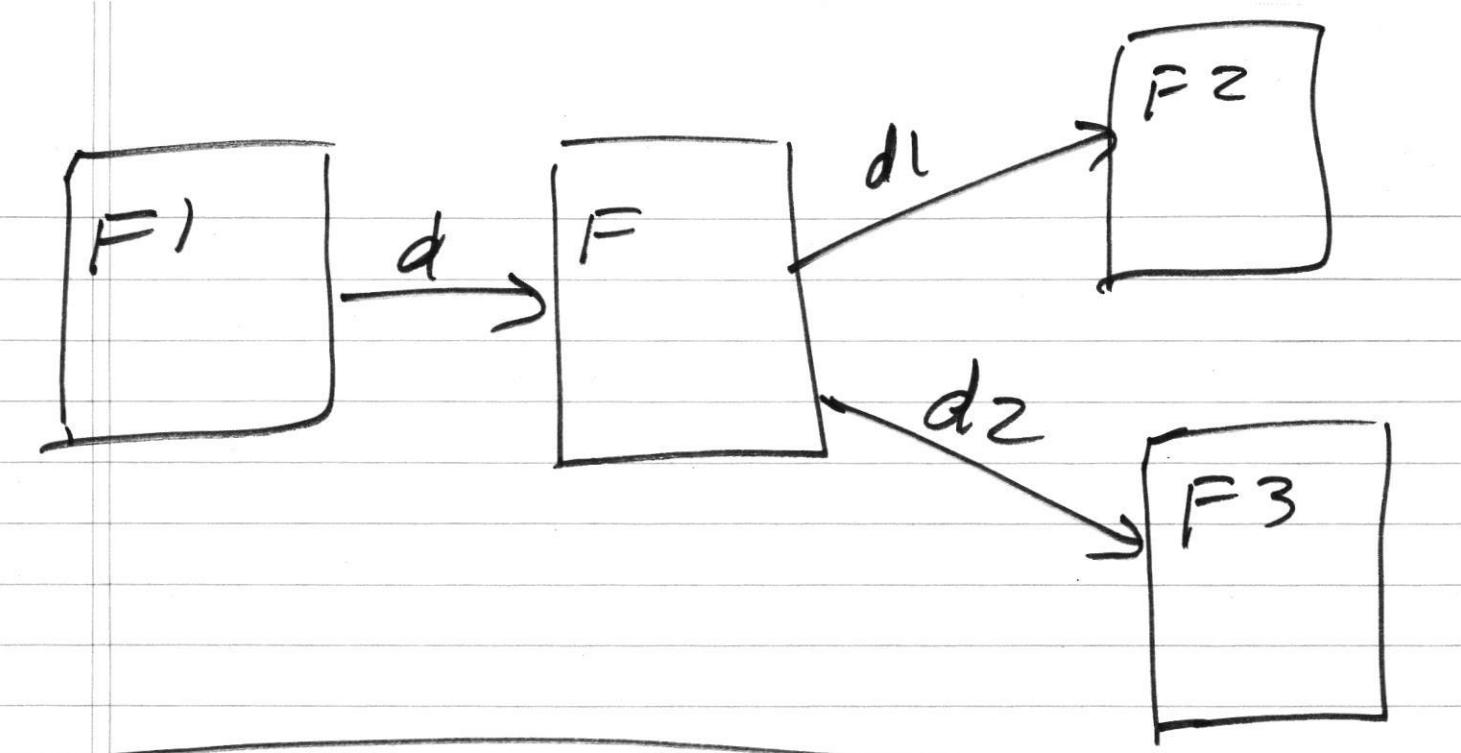
F2

F3

F

F'





## class F

read(d<sub>1</sub>)

{ if (d is not present)

} p → read(d)

} process d to compute d<sub>1</sub>

return d<sub>1</sub>

4  
read2(d<sub>2</sub>)

{ if (d is not present)

} p → read(d)

} process d to compute d<sub>2</sub>

return d<sub>2</sub>

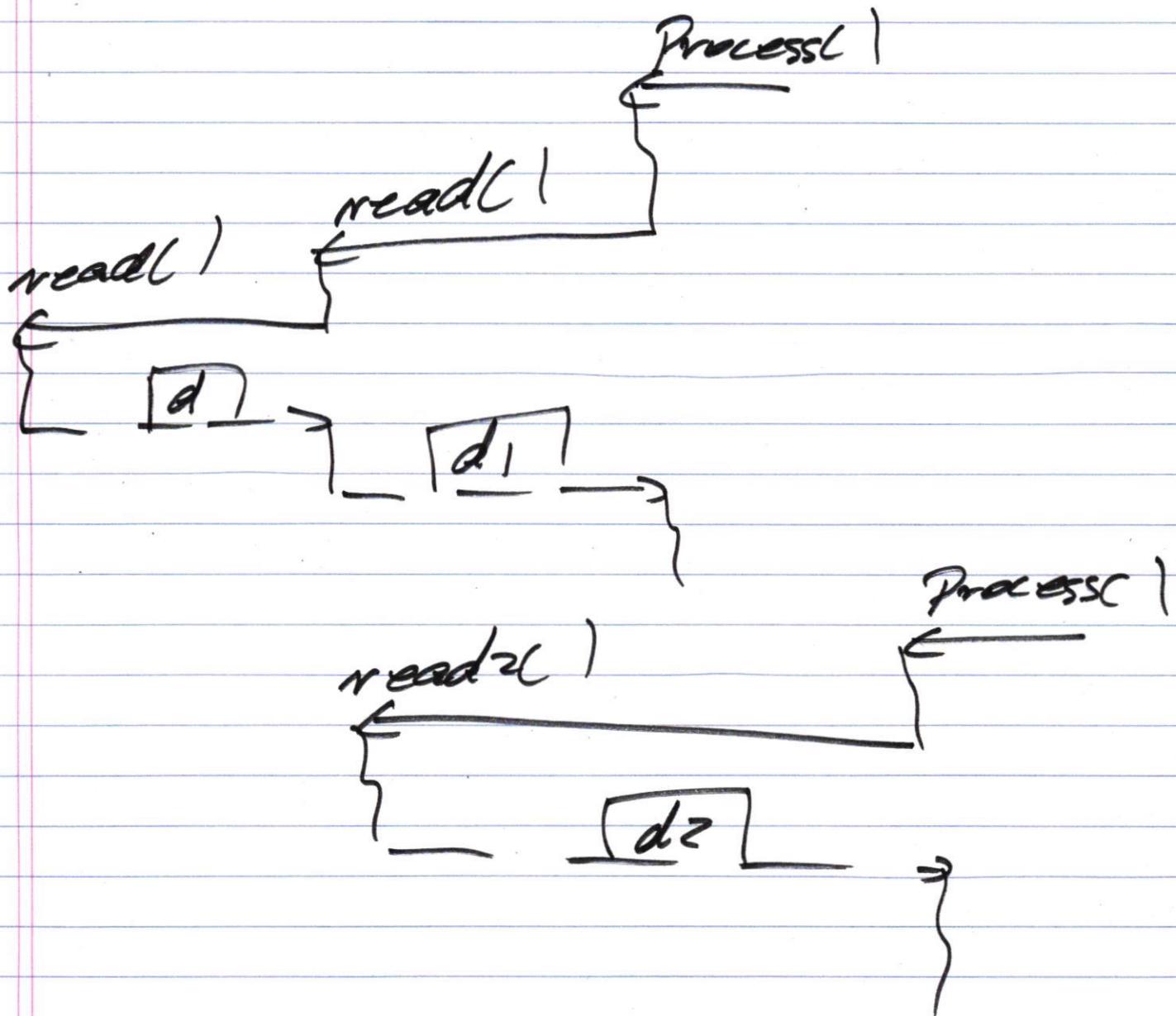
4

F1

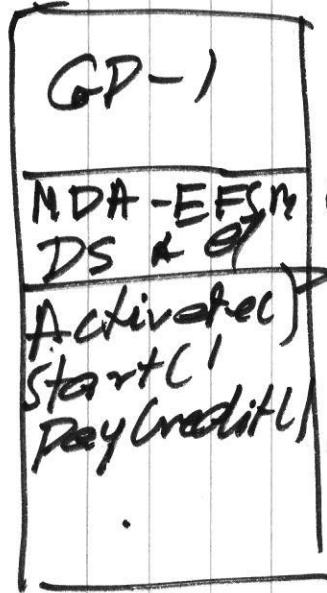
F

F2

F3



Project      Part # 1



GP-1 objects

GP-1, MDA-EFSM, OP, DS1

GP-2 objects

GP-2, MDA-EFSM, OP, DS2

- \* class diagram
- \* list of meta events.
- \* list of meta actions  
short description of responsibility for each action.
- \* MDA-EFSM state diagram.
- \* pseudo-code of all operations of GP-1 and GP-2.

## PROJECT-General Description

### PART #1

CS 586; Spring 2024

#### **Deadlines:**

**Part #1: MDA-EFSM (5 points): Monday, April 1, 2024**

Late submissions: 50% off

After **April 4** 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 Blackboard. Your submission should be as 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.

#### **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 (int a)	// the gas pump is activated where <i>a</i> is the price of the gas per liter
Start()	//start the transaction
PayCredit()	// pay for gas by a credit card
Reject()	// credit card is rejected
Cancel()	// cancel the transaction
Approved()	// credit card is approved
PayCash(int c)	// pay for gas by cash, where <i>c</i> represents prepaid cash
StartPump()	// start pumping gas
Pump()	// one liter of gas is disposed
StopPump()	// stop pumping gas

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

Activate (float a, float b, float c)	// the gas pump is activated where <i>a</i> is the price of Regular gas, <i>b</i> is //the price of Premium gas and <i>c</i> is the price of Diesel per Gallon
--------------------------------------	---

Start()	//start the transaction
PayCash(int c)	// pay for gas by cash, where <i>c</i> represents prepaid cash
Cancel()	// cancel the transaction
Premium()	// Premium gas is selected
Regular()	// Regular gas is selected
Diesel()	// Diesel gas is selected
StartPump()	// start pumping gas
PumpGallon()	// one Gallon of gas is disposed
Stop()	// stop pumping gas
Receipt()	// Receipt is requested
NoReceipt()	// No receipt

Both gas pumps are state-based components and are used to control simple gas pumps. Users can pay by cash or a credit card. The gas pump may dispose of 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:

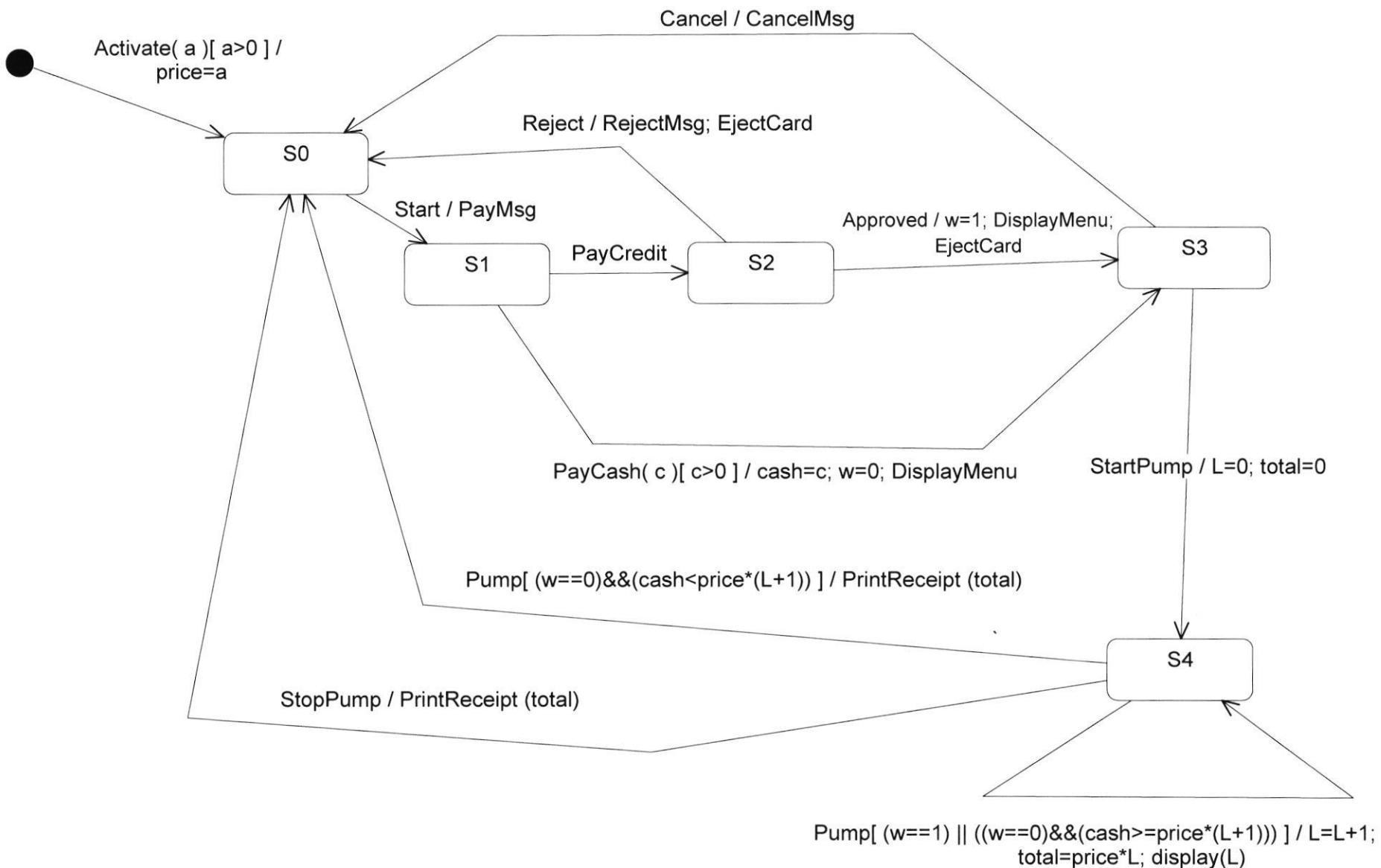
- a. Types of gasoline disposed
- b. Types of payment
- c. Display menu(s)
- d. Messages
- e. Receipts
- f. Operation names and signatures
- g. Data types
- h. etc.

The goal of this project is to design two GP components using the Model-Driven Architecture (MDA) covered in the course. An executable meta-model referred to as MDA-EFSM of GP components should capture the “generic behavior” of two GP components and should be de-coupled from data and implementation details. Notice that in your design there should be **ONLY** one MDA-EFSM for two GP 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.

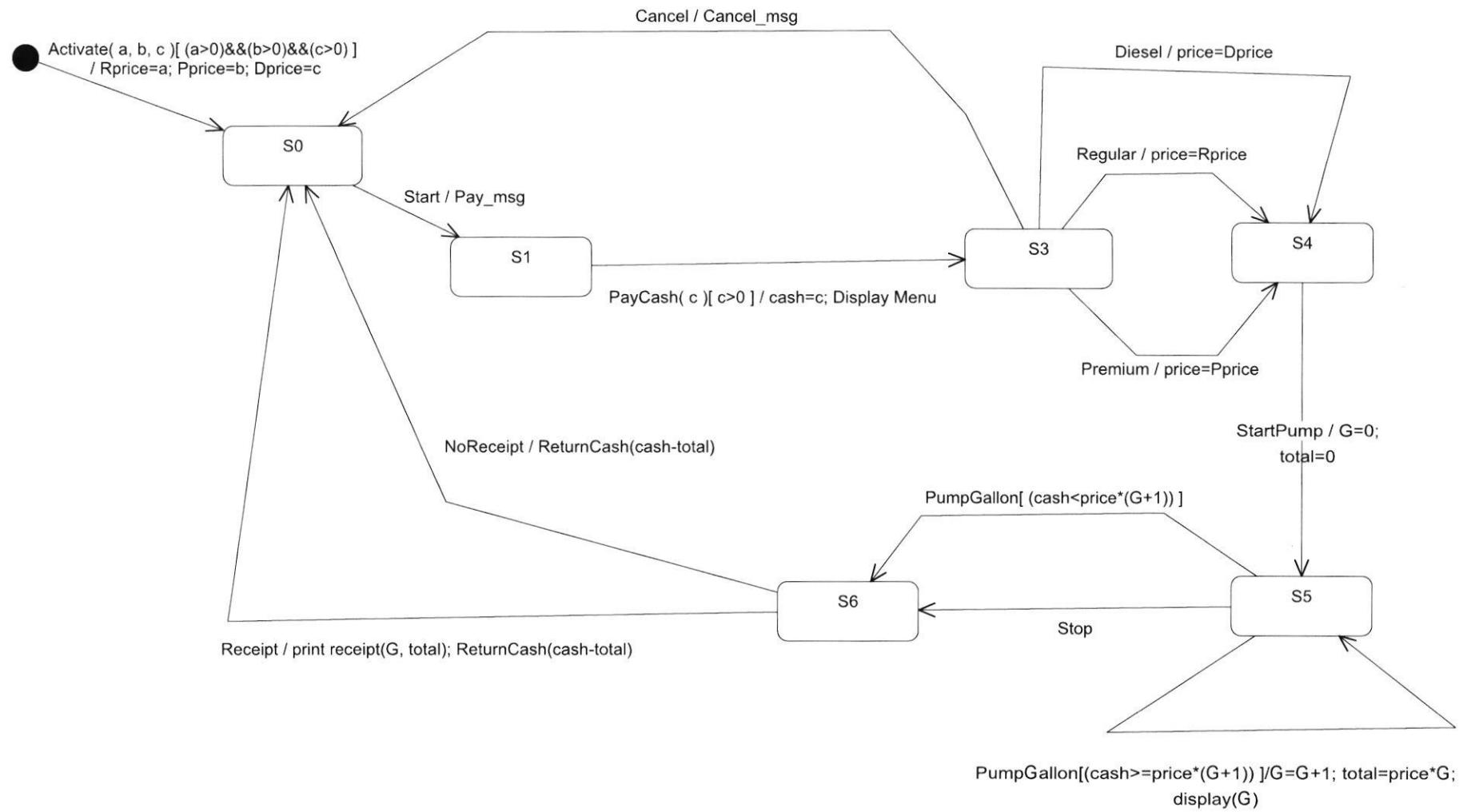
#### MDA-ESM REPORT SUBMISSION

The MDA-EFSM model report for the *GP* 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 Input Processors of *GP-1* and *GP-2*



**Figure 1: EFSM for gas pump GP-1**



**Figure 2: EFSM for GasPump-2**

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

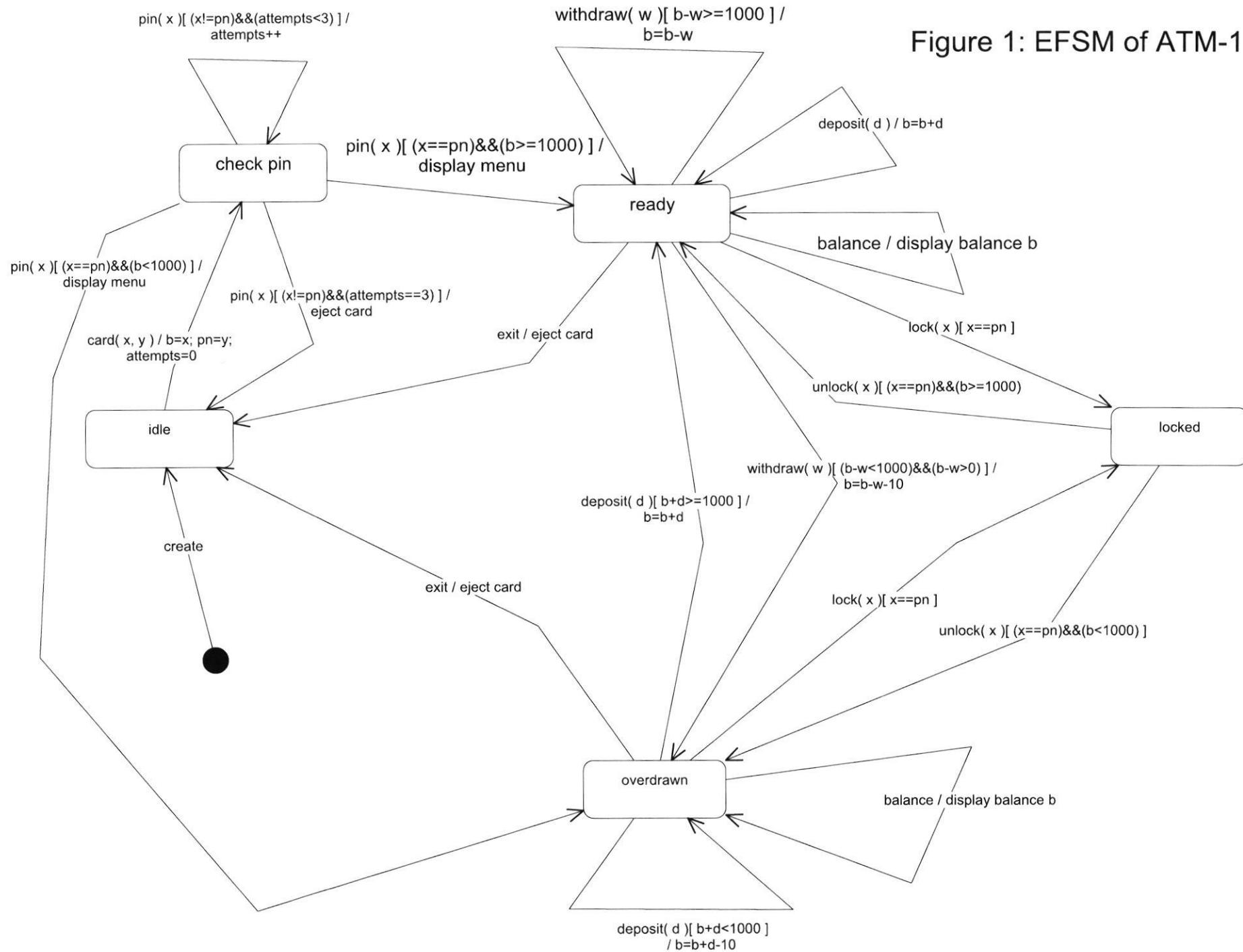
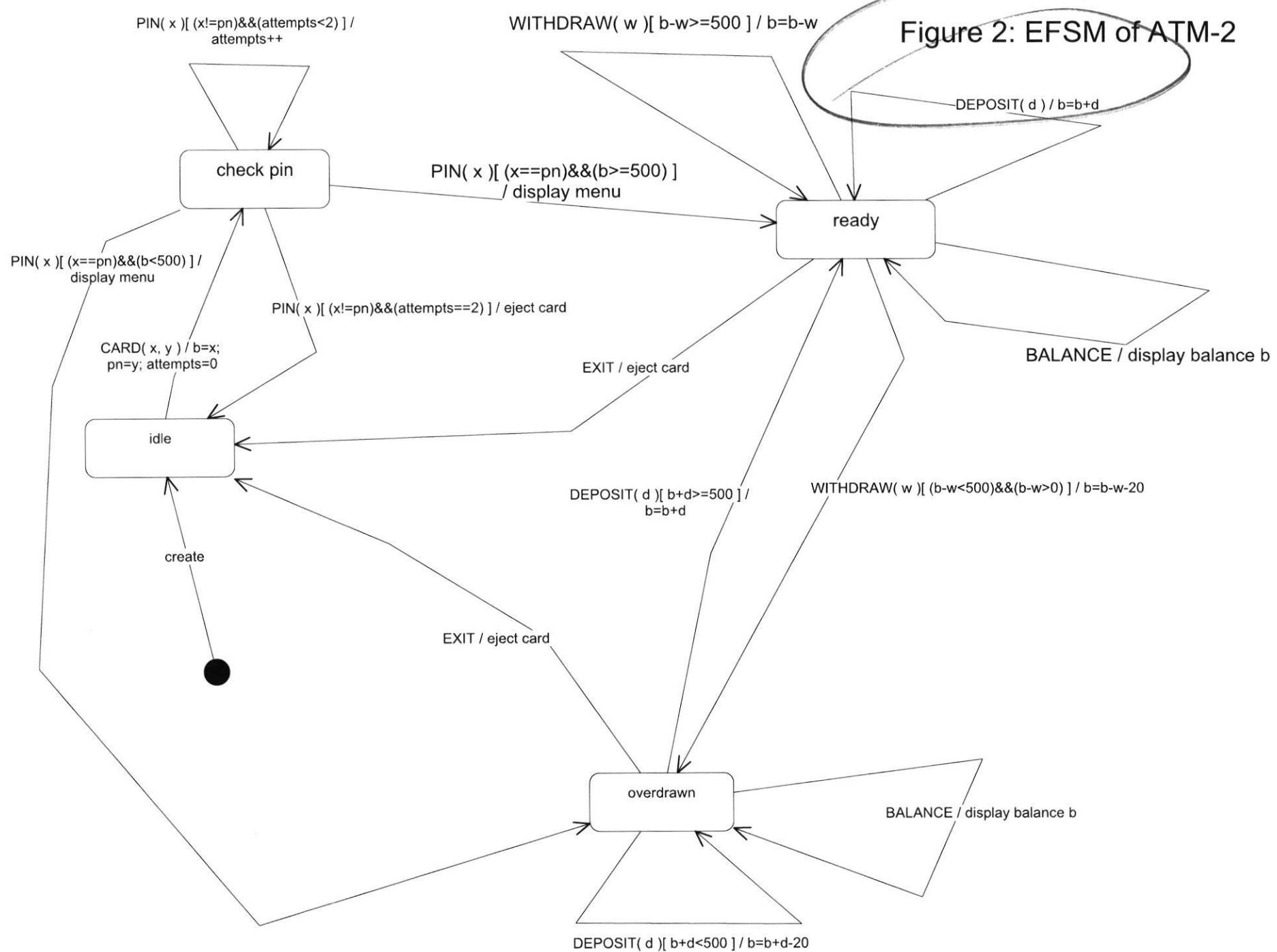
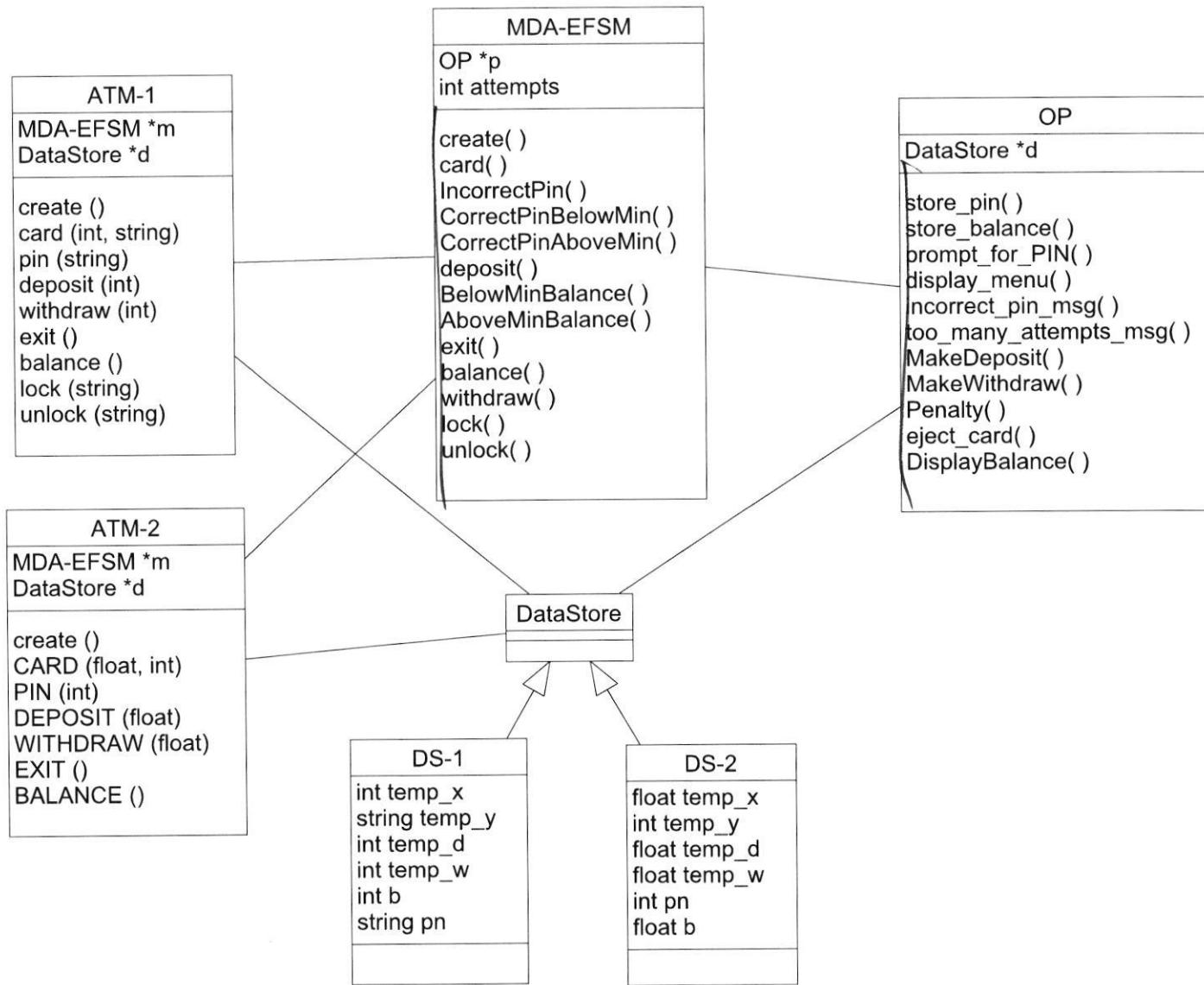
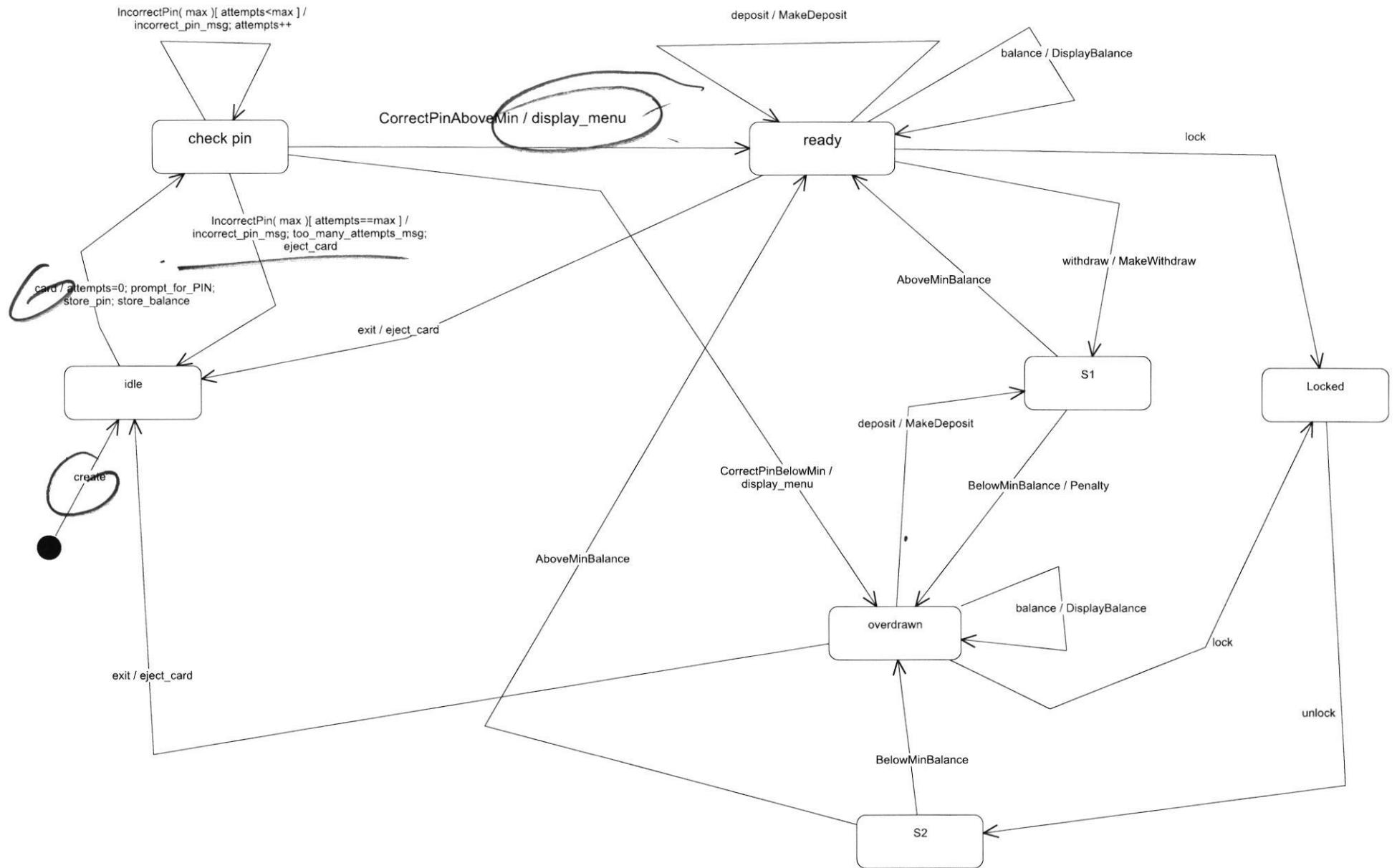


Figure 2: EFSM of ATM-2







**MDA-EFSM 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 <i>pin</i> in data store
store_balance	// stores pin from temporary data store to <i>b</i> 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 #