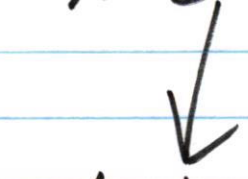
Homework #1 is posted

# OO design patterns

(*) item description pattern
(*) whole-part -"-
(*) observer -"-
(*) state -"-
* adapter -"-
* strategy -"-

# State Pattern

EFSM / FSM
state-based
model



state Pattern

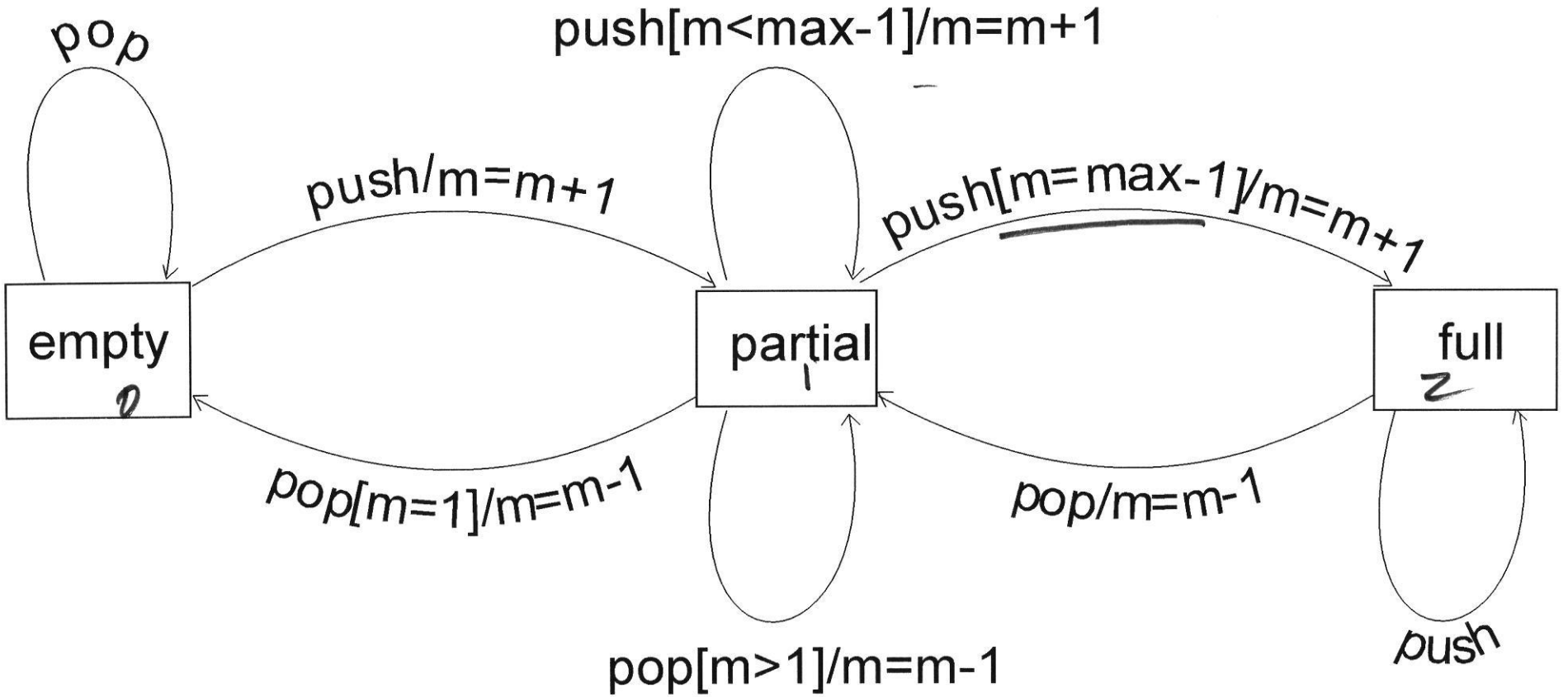
initial design

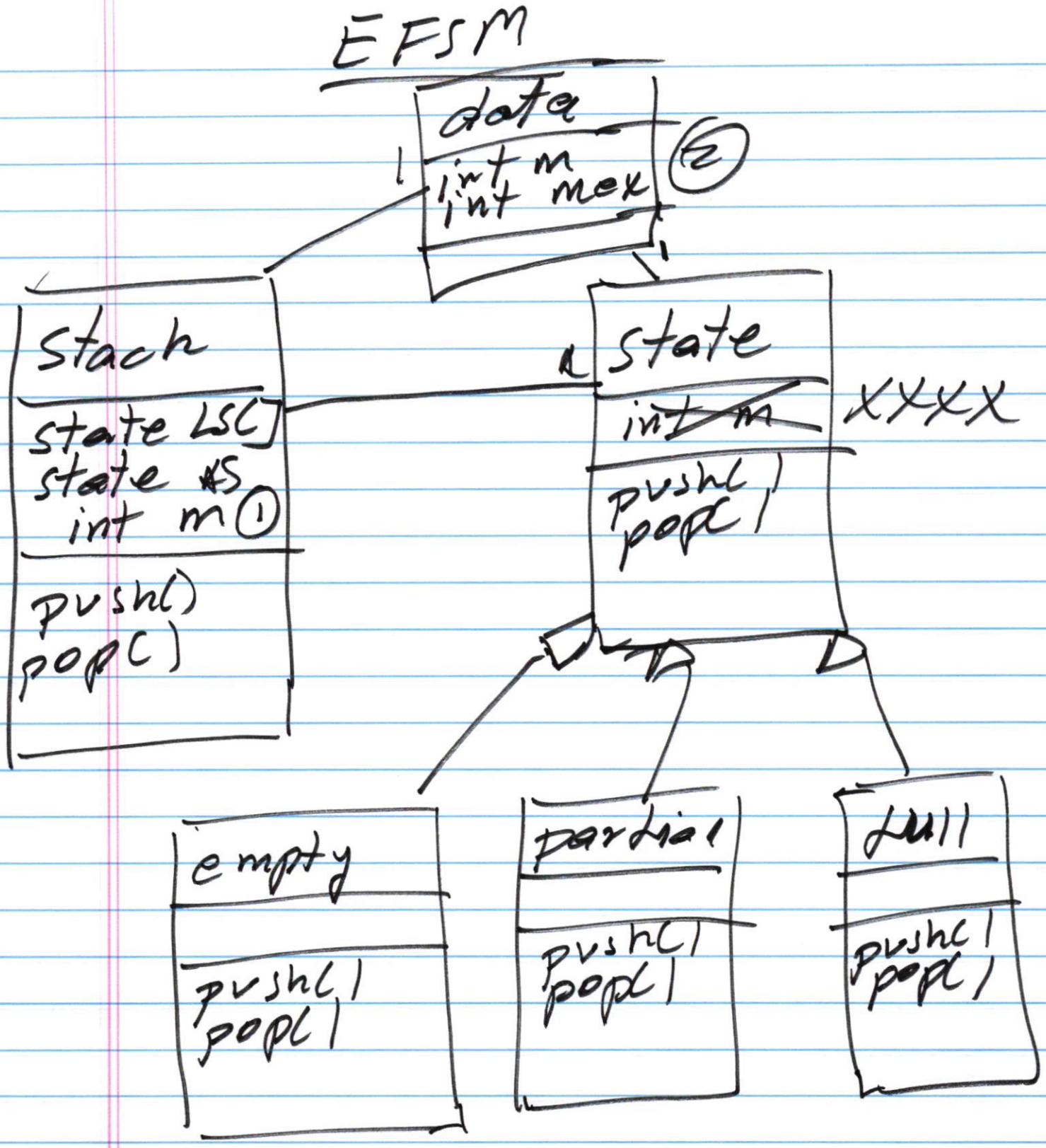
EFSM

stack push() pop()

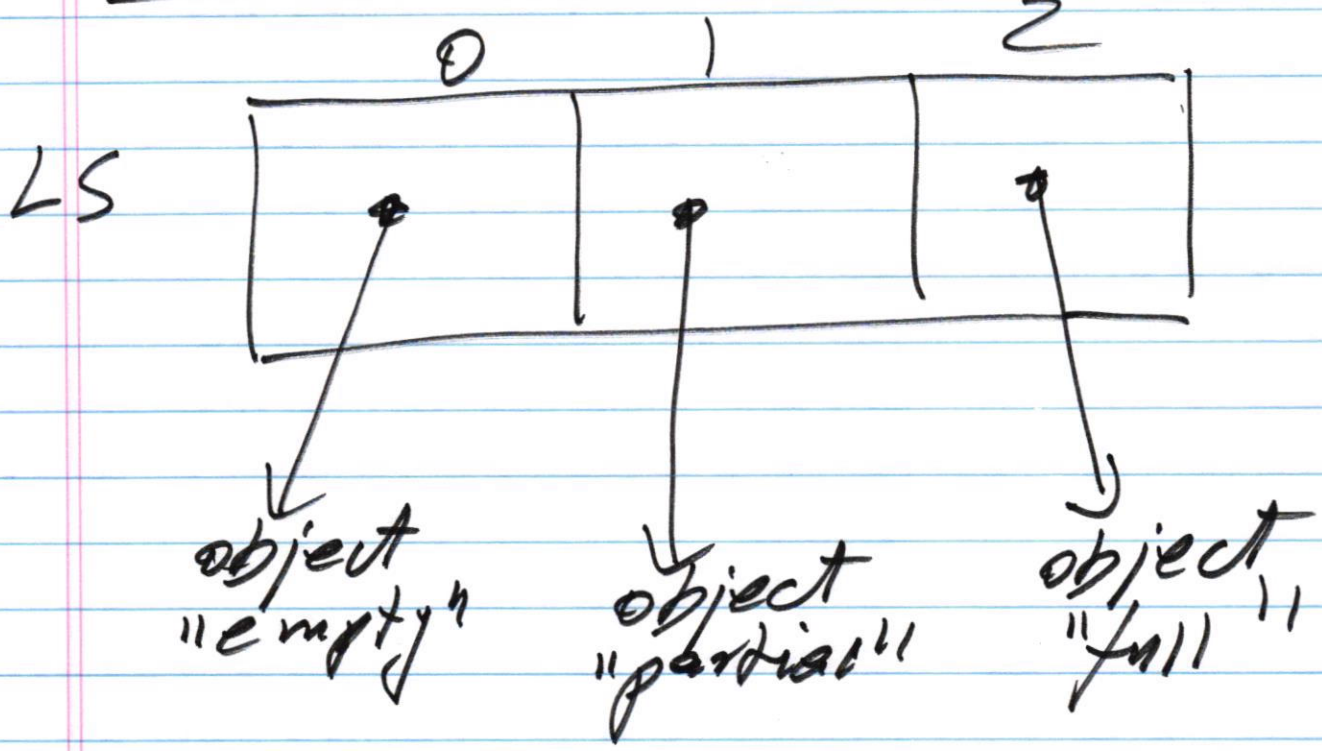
max=100

m=0 // counter




EFSM
data
int m
int max
2
1
Stack
state LSC
state *S
int m
1
push()
pop()
State
int m
XXXX
push()
pop()
empty
push()
pop()
partial
push()
pop()
full
push()
pop()

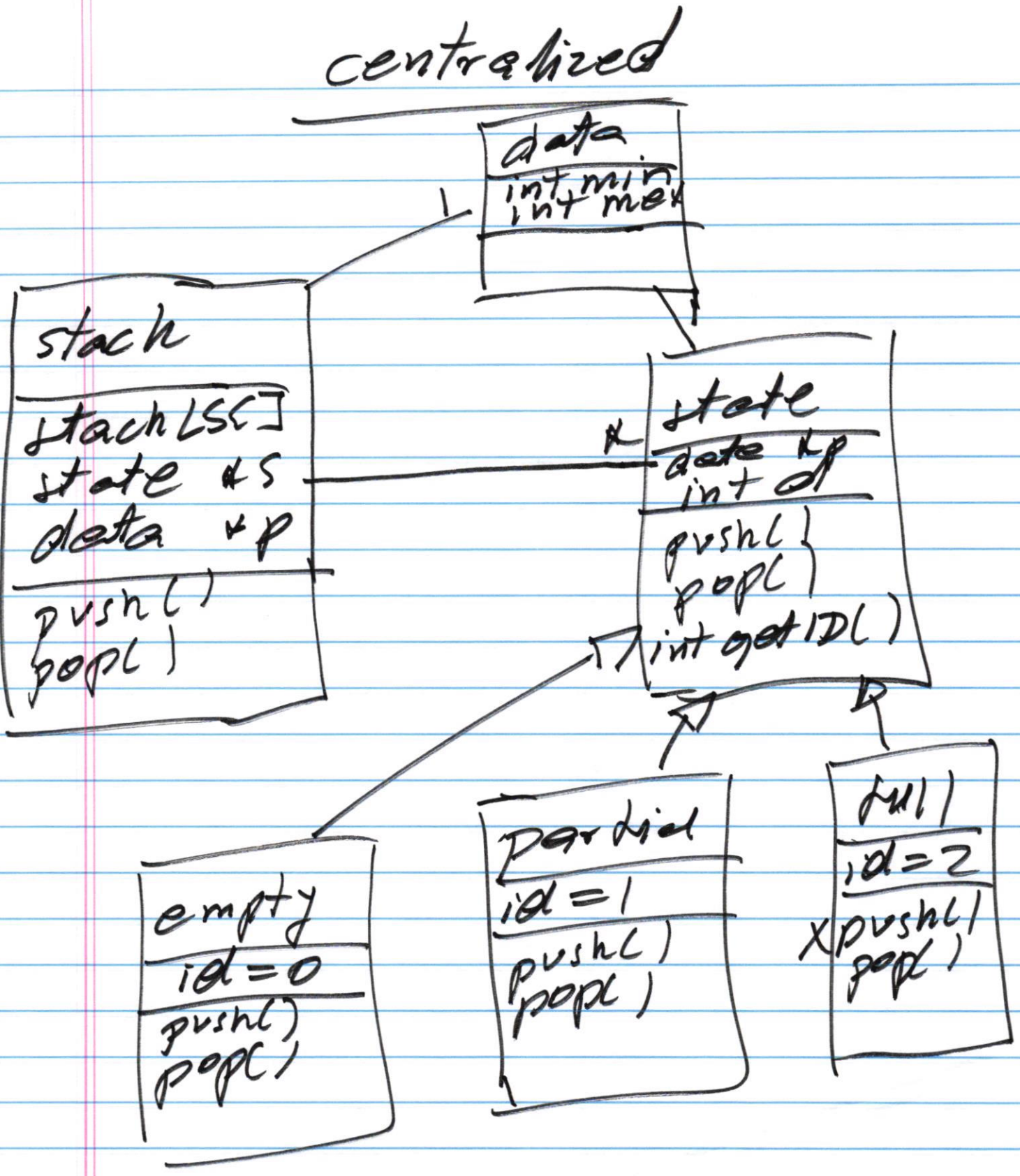
# stack class




S → points to the current state object

Initially: S = LS[0]

# centralized



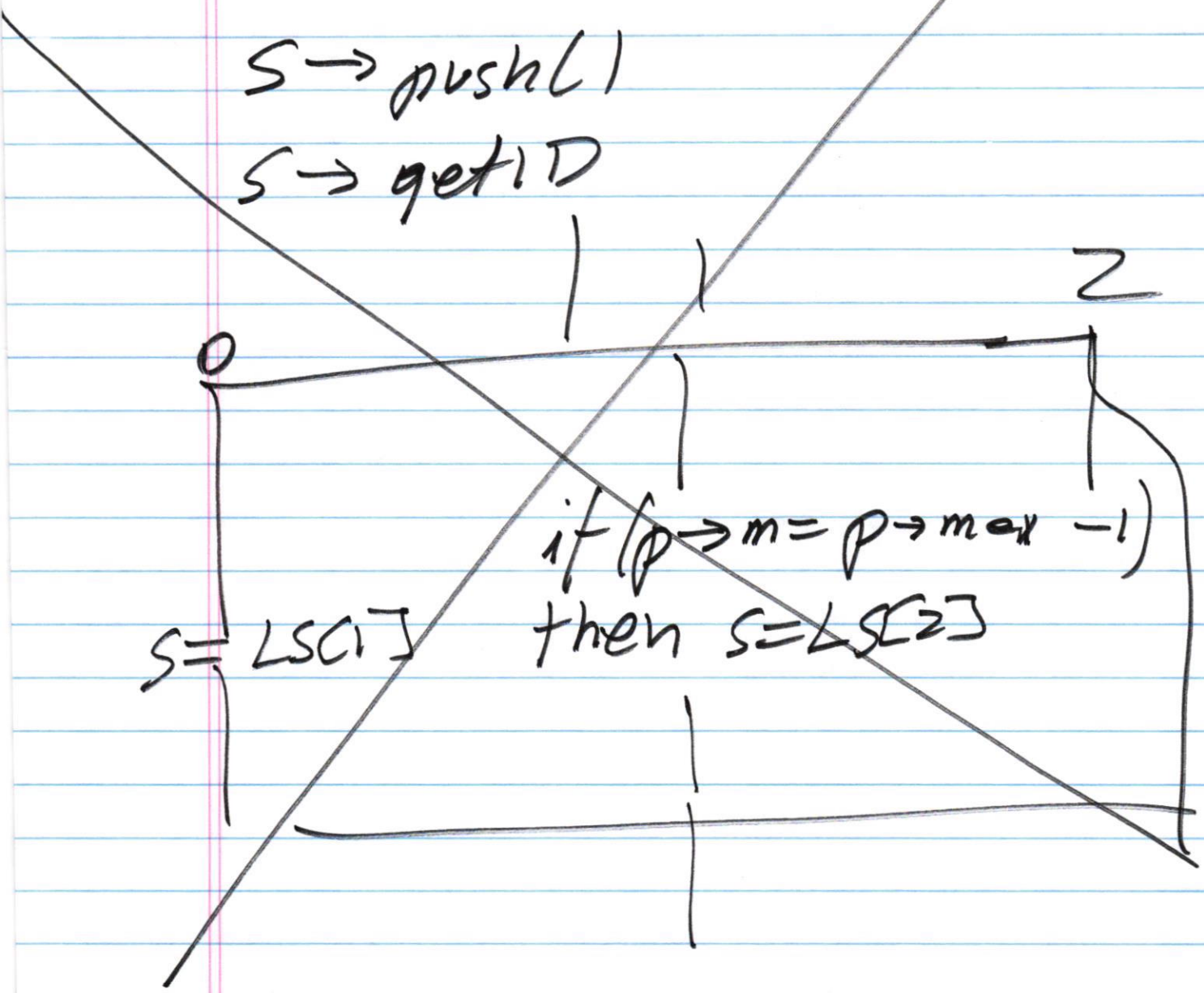
class stack

push()

incorrect solution

S → push()

S → getID



"empty" class

push()

p->m = p->m + 1

---

"partial" class

push()

p->m = p->m + 1

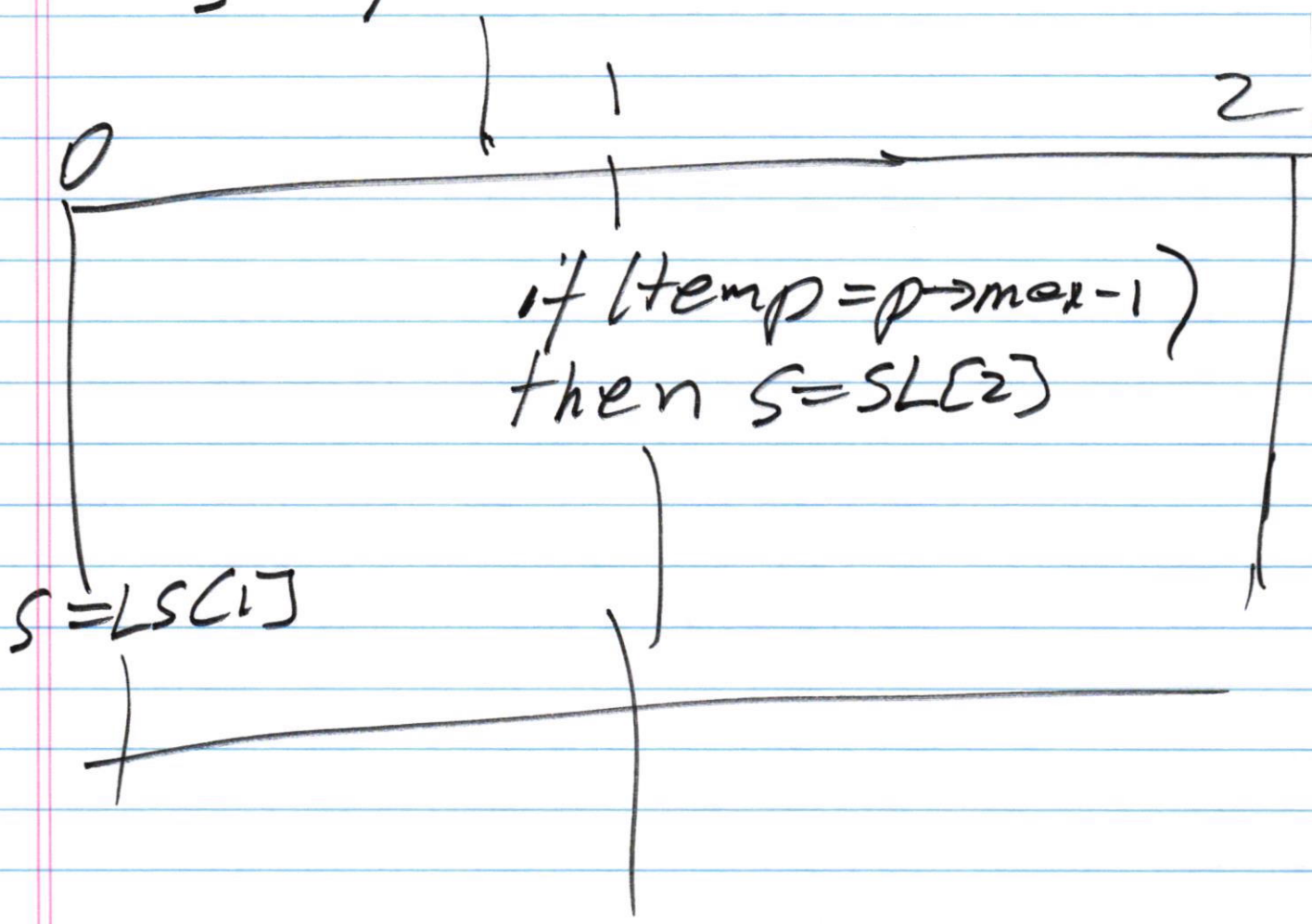
# class stack

## push()

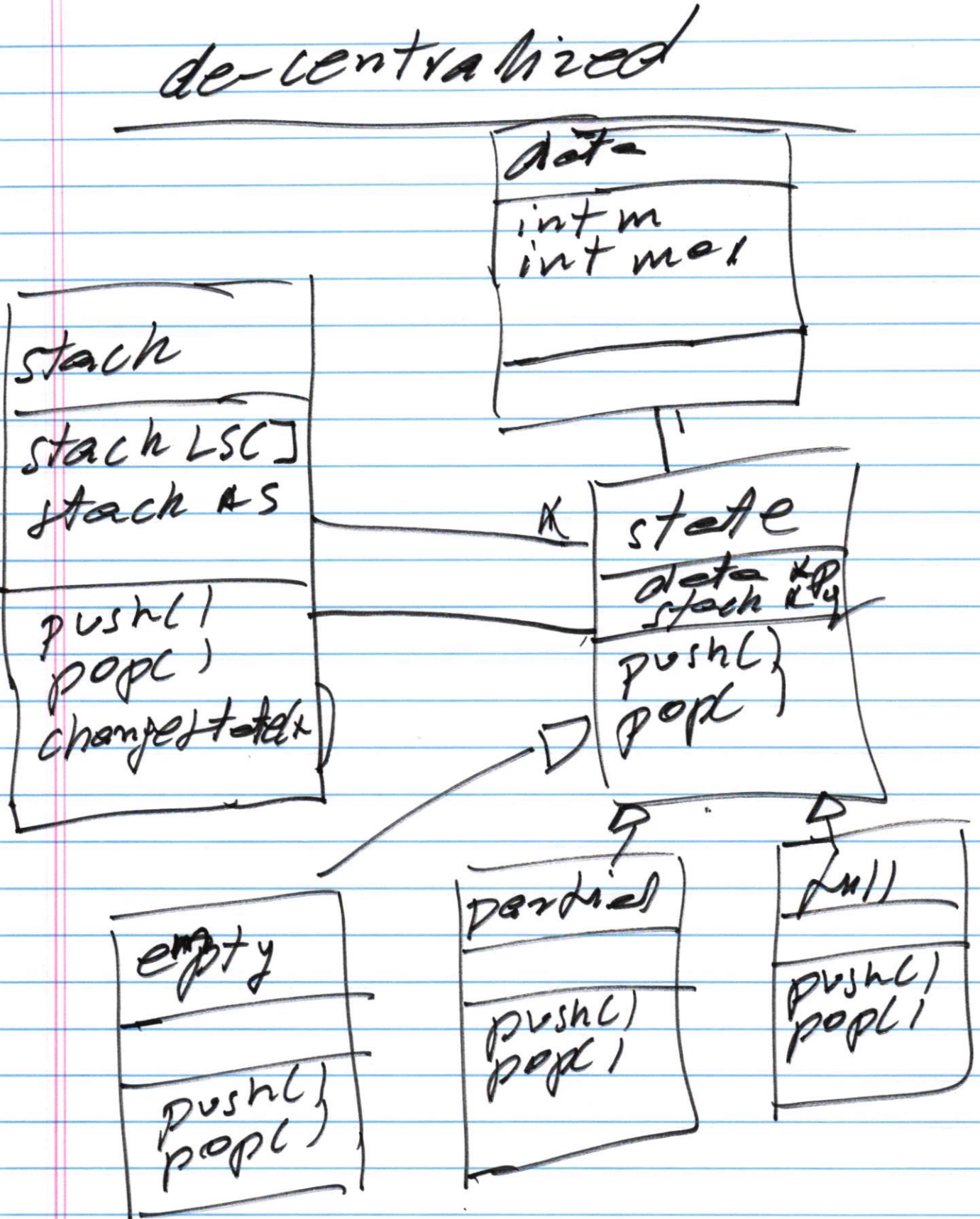
temp = p->m

s->push()

s->getID()



# de-centralized



class stack

push()

S → push()

pop()

S → pop()

changeState(x)

S = LS[x]

"empty" class

pvsh()

p->m = p->m+1

q -> changeState(1)

---

"partial" class

pvsh()

if (p->m < p->max-1) then

p->m = p->m+1

else if (p->m == p->max-1)

then p->m = p->m+1

q -> changeState(2)

end if.

**HOMEWORK ASSIGNMENT #1**

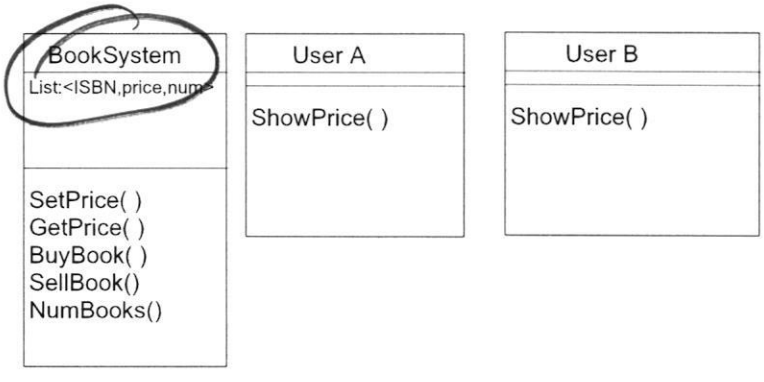
CS 586; Fall 2025
Due Date: **September 18, 2025**
Late homework: 50% off
After **September 23,** the homework assignment will not be accepted.

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

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

**PROBLEM #1** (40 points)
In the system, there exists a class *BookSystem* which keeps track of prices of books in the Book Market. This class supports the following operations: *SetPrice(price,ISBN)*, *GetPrice(ISBN)*, *BuyBook(ISBN)*, *SellBook(ISBN)*, and *NumBooks(ISBN)*. The *SetPrice(price,ISBN)* operation sets a new *price* for the book uniquely identified by *ISBN*. The *GetPrice(ISBN)* operation returns the current price of the book identified by *ISBN*. The *BuyBook(ISBN)* operation is used to buy a book identified by *ISBN*. The *SellBook(ISBN)* operation is used to sell a book identified by *ISBN*. The operation *NumBooks(ISBN)* returns the number of copies of a book identified by ISBN that are available in the system. Notice that each book is uniquely identified by *ISBN*.

In addition, there exist user components in the system (e.g., *UserA*, *UserB*, etc.) that are interested in watching the changes in book prices, especially, they are interested in watching the out-of-range book price changes. Specifically, interested users may register with the system to be notified when the price of the book of interest falls outside of the specified price range. During registration, the user needs to provide the boundaries (*lowprice*, *highprice*) for the price range for the specific book, where *lowprice* is the lower book price and *highprice* is the upper book price of the price range. At any time, users may un-register when they are not interested in watching the out-of-range book price changes of a specific book. Each time the price of a book changes, the system notifies all registered users (for which the new book price is outside of the specified price range) about the out-of-range book price change. Notice that if the book price change is within the specified price range for a given user, this user is not notified about this price change.




Design the system using the **Observer pattern**. Provide a class diagram for the system that should include classes *BookSystem*, *UserA*, and *UserB* (if necessary, introduce new classes and operations). In your design, it should be easy to introduce new types of user components (e.g., *UserC*) that are interested in observing the changing prices of books. Notice that the components in your design should be **decoupled** as much as possible. In addition, components should have **high cohesion**.

**In your solution:**

a. Provide a class diagram for the system. For each class, list all operations with parameters and specify them using **pseudo-code**. In addition, for each class, provide its attributes/data structures. Make the necessary assumptions for your design.

b. Provide two **sequence diagrams** showing:
   - How components *UserA* and *UserB* register to be notified about the out-of-range book price change.
   - How the system notifies the registered user components about the out-of-range book price change.

**PROBLEM #2** (60 points)

The ATM 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 |

A simplified EFSM model for the *ATM* component is shown on the next page.

Design the system using the **State design pattern**. Provide two solutions:

- **a decentralized** version of the State pattern
- **a centralized** version of the State pattern

Notice that the components in your design should be **decoupled** as much as possible. In addition, components should have high **cohesion**.

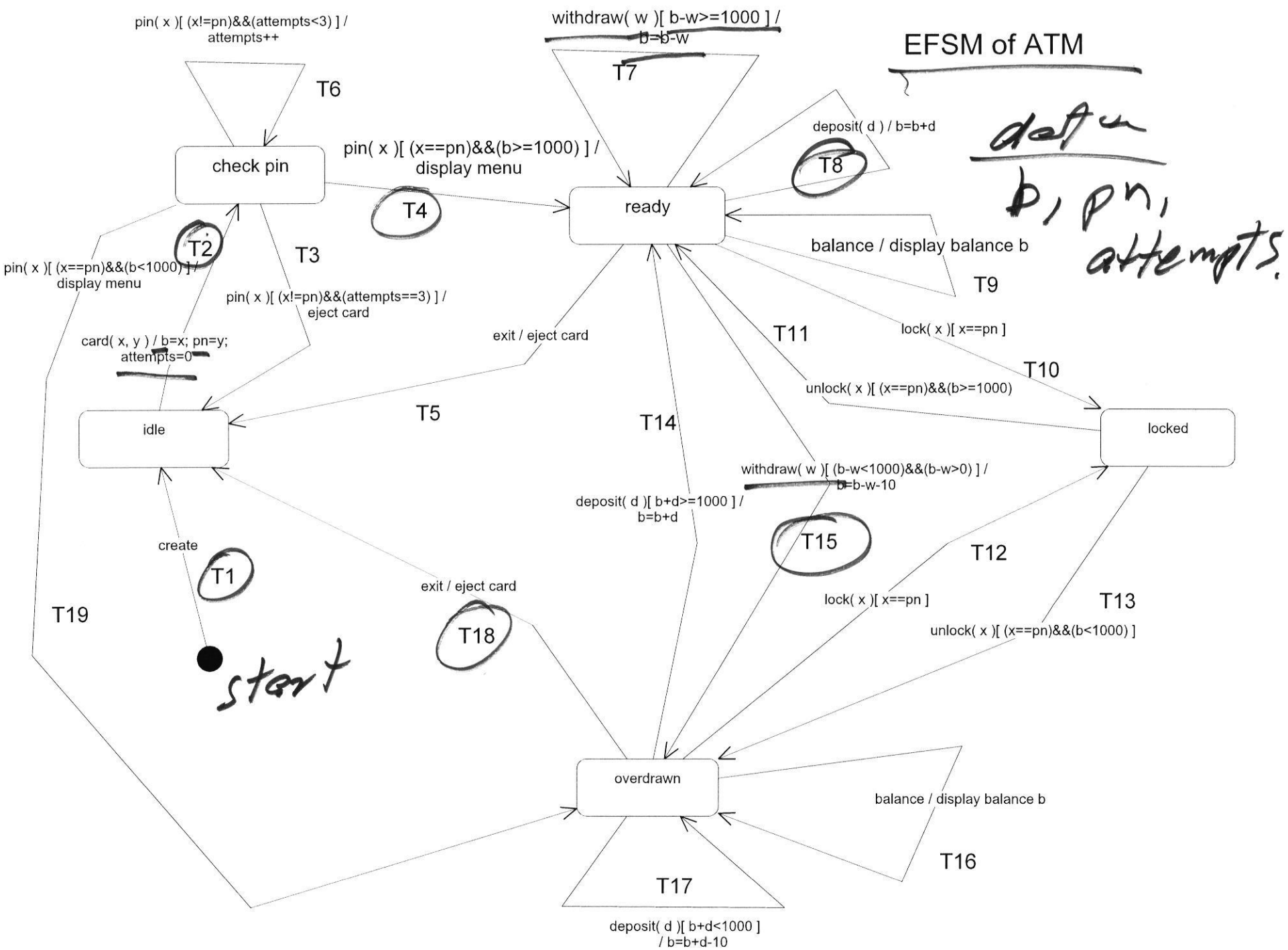
**For each solution:**

a. Provide a class diagram for the system. For each class, list all operations with parameters and specify them using **pseudo-code**. In addition, for each class, provide its attributes and data structures. Make the necessary assumptions for your design.

b. Provide a **sequence diagram** for the following operation sequence:

*create(), card(1100,"xyz"), pin("xyz"), deposit(300), withdraw(500), exit()*

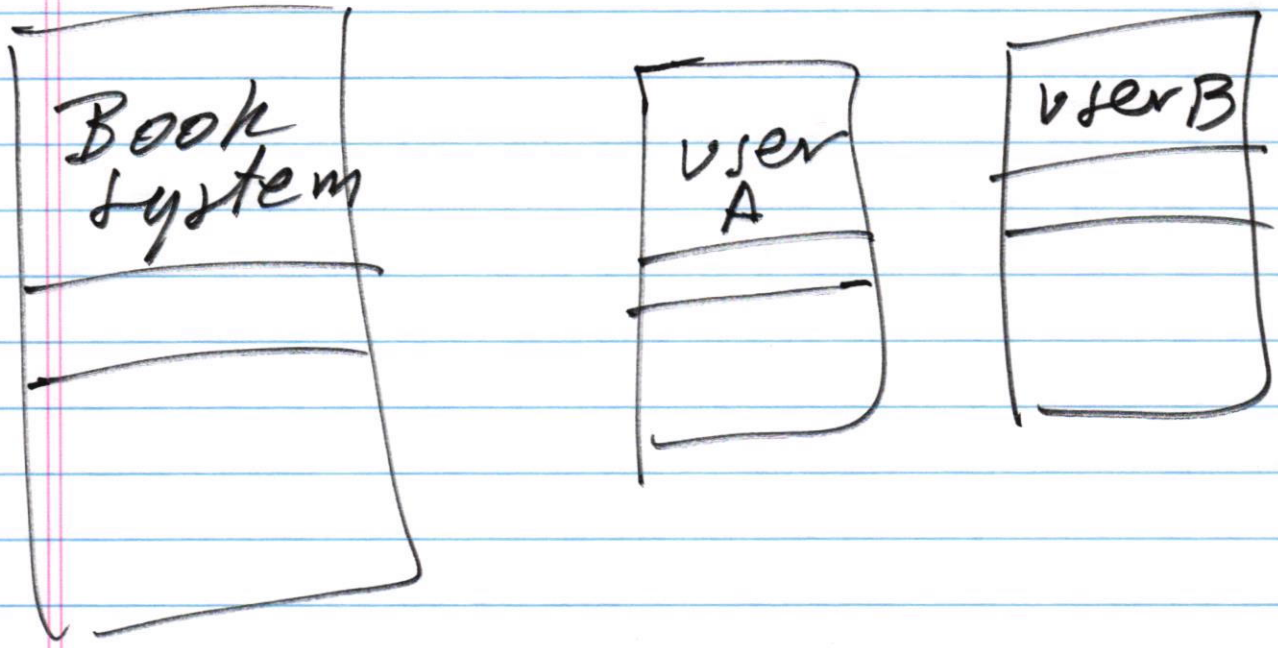
When the EFSM model is "executed" on this sequence of operations, the following sequence of transitions is traversed/executed: $T_1$, $T_2$, $T_4$, $T_8$, $T_{15}$, $T_{18}$


EFSM of ATM
data
b, pn, attempts
pin( x )[ (x!=pn)&&(attempts<3) ] / attempts++
T6
withdraw( w )[ b-w>=1000 ] / b=b-w
T7
deposit( d ) / b=b+d
T8
check pin
pin( x )[ (x==pn)&&(b>=1000) ] / display menu
T4
ready
T2
T3
pin( x )[ (x==pn)&&(b<1000) ] / display menu
pin( x )[ (x!=pn)&&(attempts==3) ] / eject card
balance / display balance b
T9
card( x, y ) / b=x; pn=y; attempts=0
exit / eject card
T11
lock( x )[ x==pn ]
T10
unlock( x )[ (x==pn)&&(b>=1000)
idle
T5
T14
locked
withdraw( w )[ (b-w<1000)&&(b-w>0) ] / b=b-w-10
deposit( d )[ b+d>=1000 ] / b=b+d
T15
T12
create
T1
exit / eject card
lock( x )[ x==pn ]
T13
unlock( x )[ (x==pn)&&(b<1000) ]
T19
T18
start
overdrawn
balance / display balance b
T16
T17
deposit( d )[ b+d<1000 ] / b=b+d-10

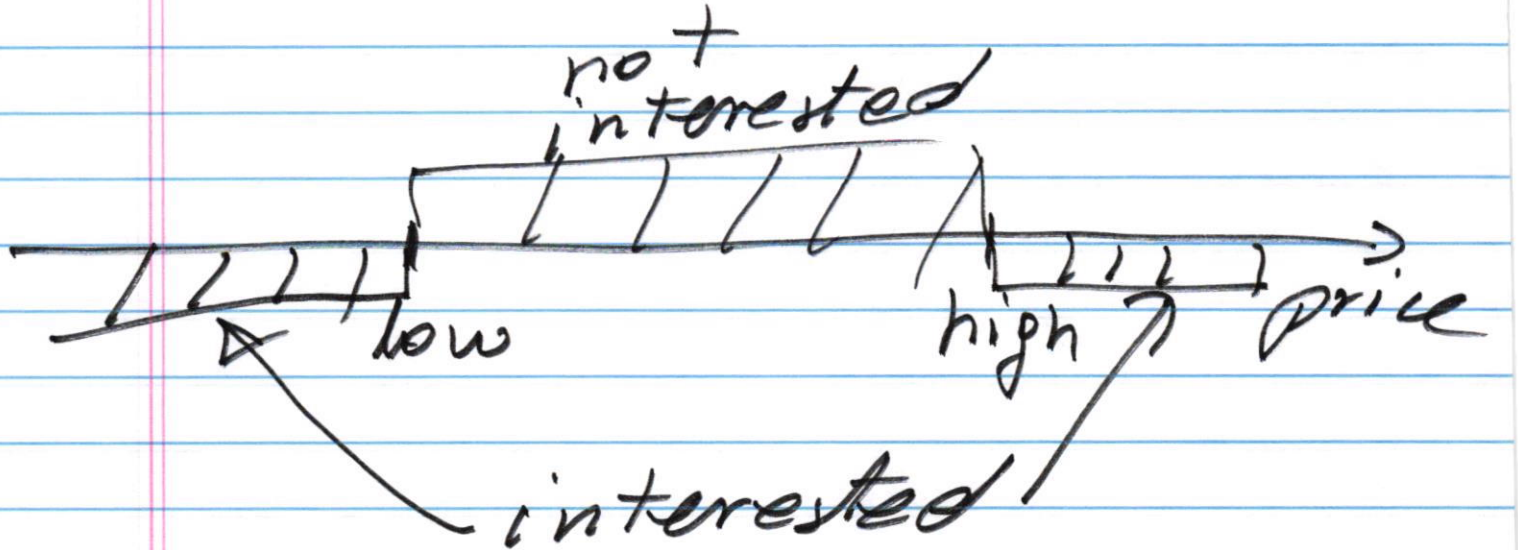
# Homework #1

## Problem #1

## Observer pattern



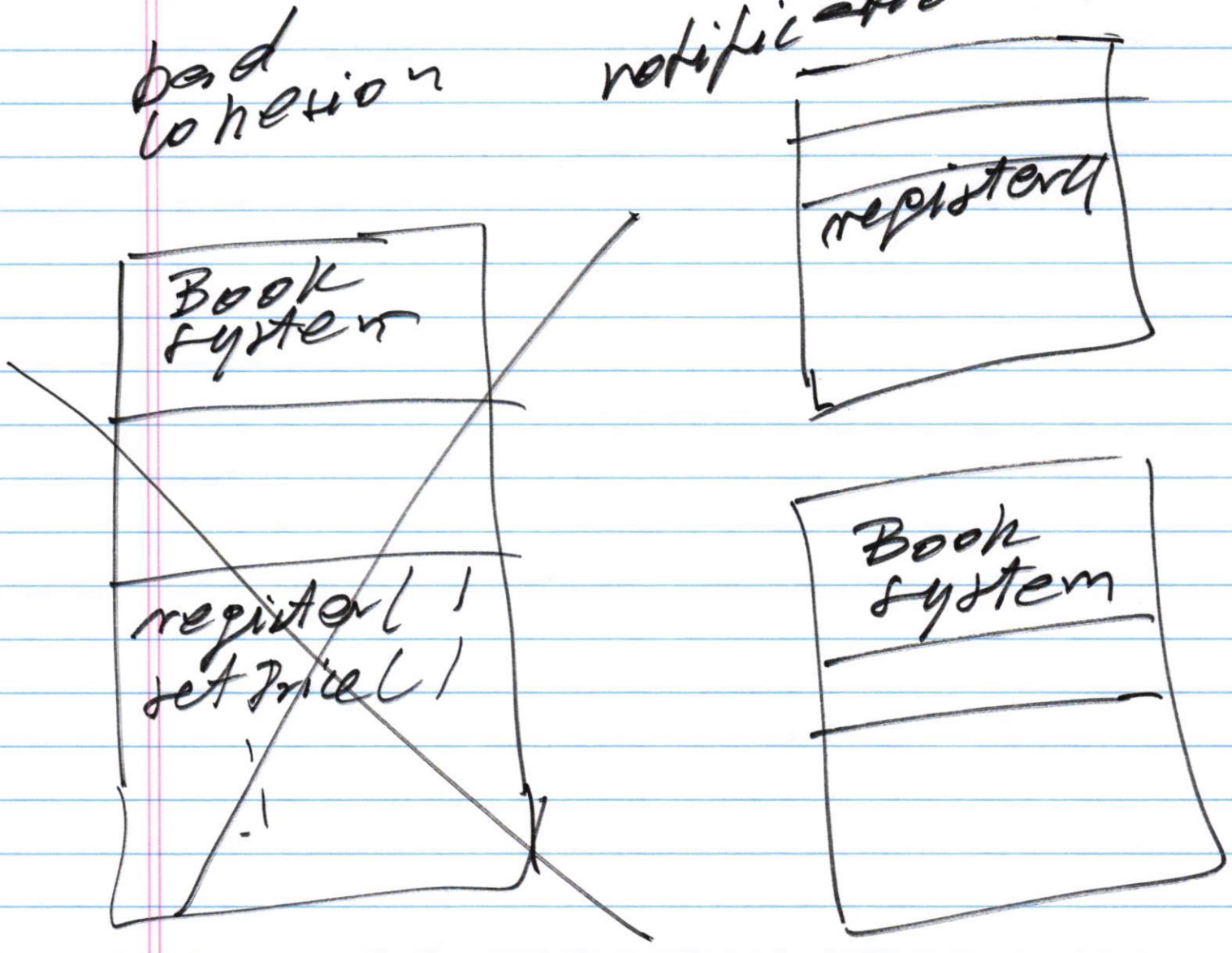
out-of-range price change.




register(isbn, low, high, user pointer)

introduce notification mechanism

bad cohesion



sample problem with solution

**PROBLEM #1**

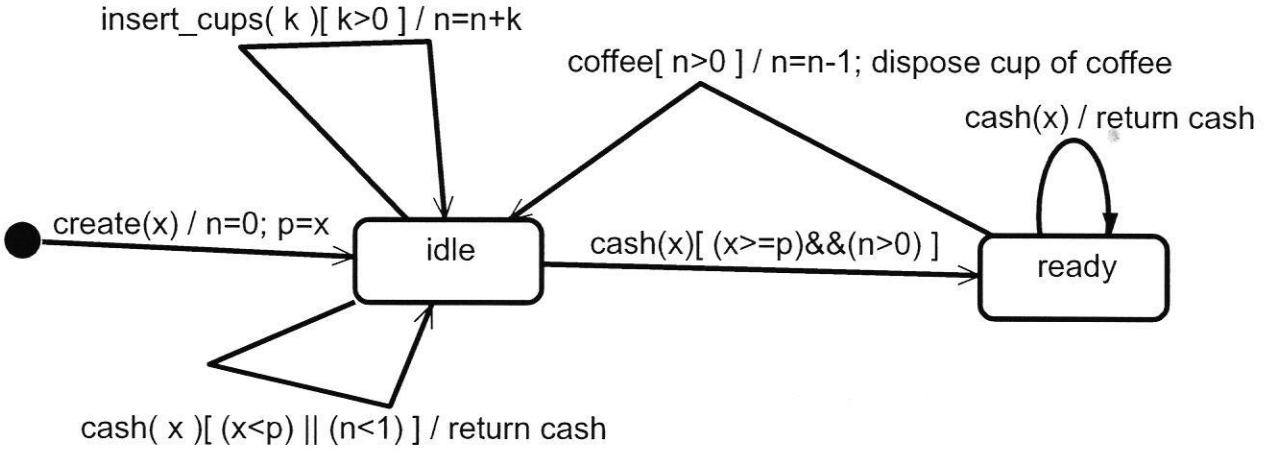
An EFSM (Extended Finite State Machine) of a component is shown below. The component supports the following operations: *create(float x), cash(float x), insert_cups(int k), coffee()*

Design the system using the **State design pattern**. You should use the **de-centralized** version of this pattern.

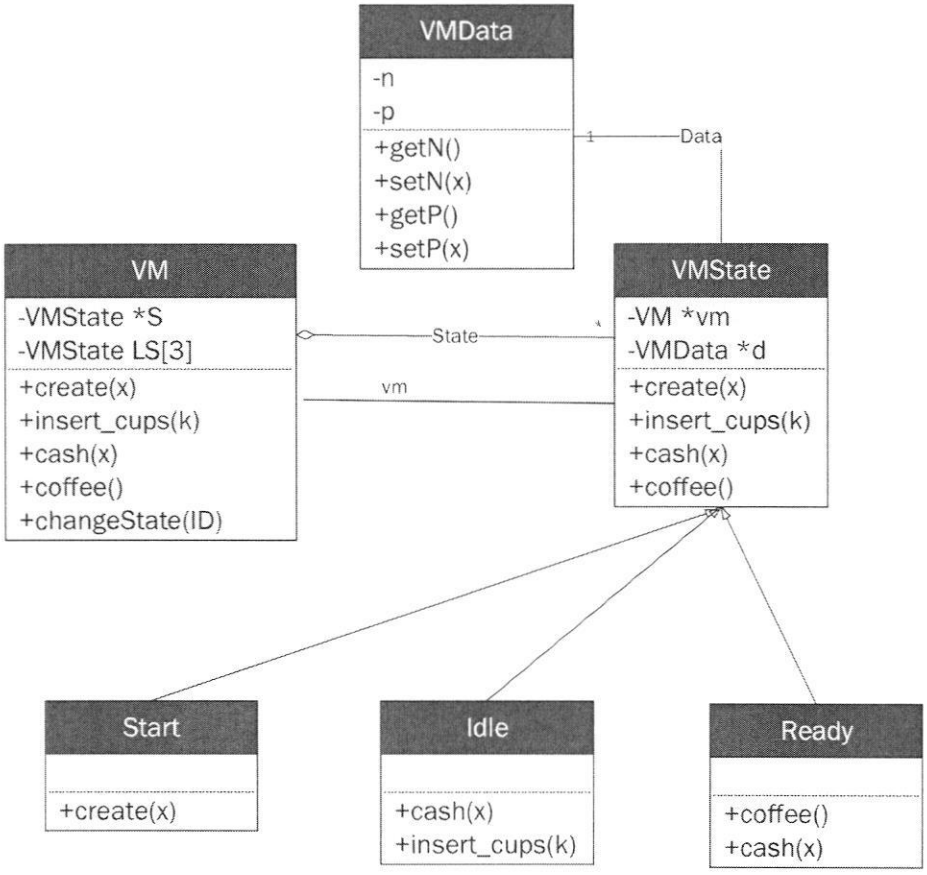
In your solution:

a. Provide a class diagram for the component. For each class list all operations with parameters and specify them using **pseudo-code**. In addition, for each class provide its attributes and data structures. Make the necessary assumptions for your design. Notice that the components in your design should be de-coupled as much as possible. In addition, components should have high cohesion.

b. Provide a **sequence diagram** for the following operation sequence:
*create(2.5), insert_cups(10), cash(3), coffee()*



**De-centralized State Pattern**




Class "VM"

S //points to current state object
LS[0] //points to "Start" Object
LS[1] //points to "Idle" Object
LS[2] //points to "Ready" Object

S = LS[0] // initialize state object to "Start"

Operations

**changeState(ID){**
S = LS[ID]
**}**

**create(x){**
S->create(x)
**}**

```
insert_cups(k){
    S->insert_cups(k)
}

cash(x){
    S->cash(x)
}

coffee(){
    S-> coffee()
}
```

Class "VMState"

Operations

| create(), insert_cups(), cash() and coffee() are abstract operations |
|---|

Class "Start"

Operations

```
create(x){
    d->setN(0)
    d->setP(x)
    vm->changeState(1)          //change VM state from "Start" to "Idle"
}
```

Class "Idle"

Operations

```
cash(x){
    IF ( x >= d->getP() ) && (d->getN() > 0 ) THEN
        vm ->changeState(2)     //change VM state from "Idle" to "Ready"
    ELSE IF ( x < d->getP() ) || (d->getN() < 1 ) THEN
        return cash
    ENDIF
}

insert_cups(k){
    IF  k > 0 THEN
        numberOfCups = d->getN()
        numberOfCups = numberOfCups + k
```

```
        d->setN(numberOfCups)
    ENDIF
}
```

Class "Ready"

Operations

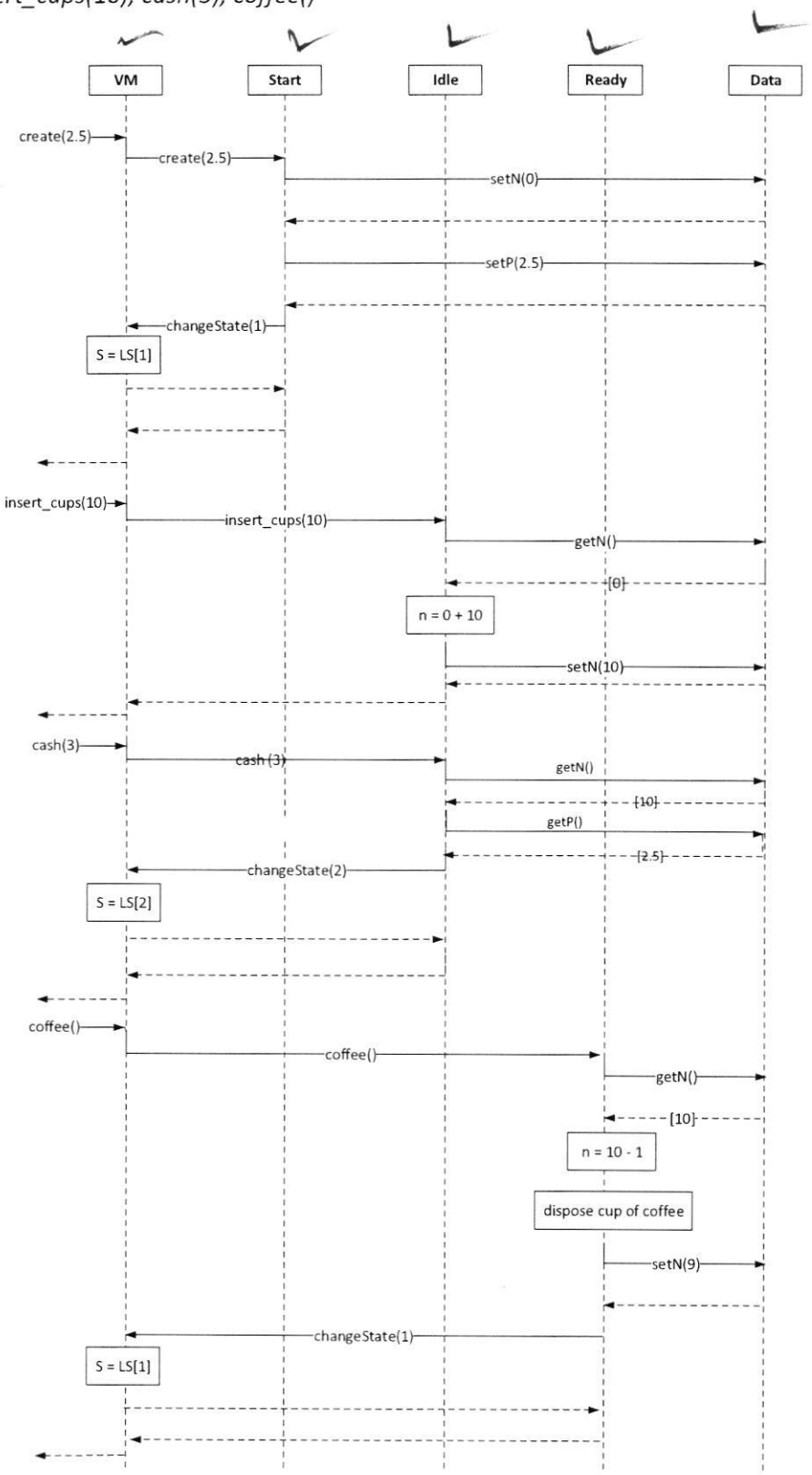
```
coffee(x){
    IF d->getN() > 0 THEN
        numberOfCups = d->getN()
        numberOfCups = numberOfCups - 1
        d->setN(numberOfCups)
        dispose cup of coffee
        vm->changeState(1)      //change VM state from "Ready" to "Idle"
    ENDIF
}

cash(x){
    return cash
}
```

Class "VMData"

```
n       // number of cups
p       // price


getN(){
   return n
}

setN(int x){
    n = x
}


getP(){
   return p
}

setP(int x){
    p = x
}
```

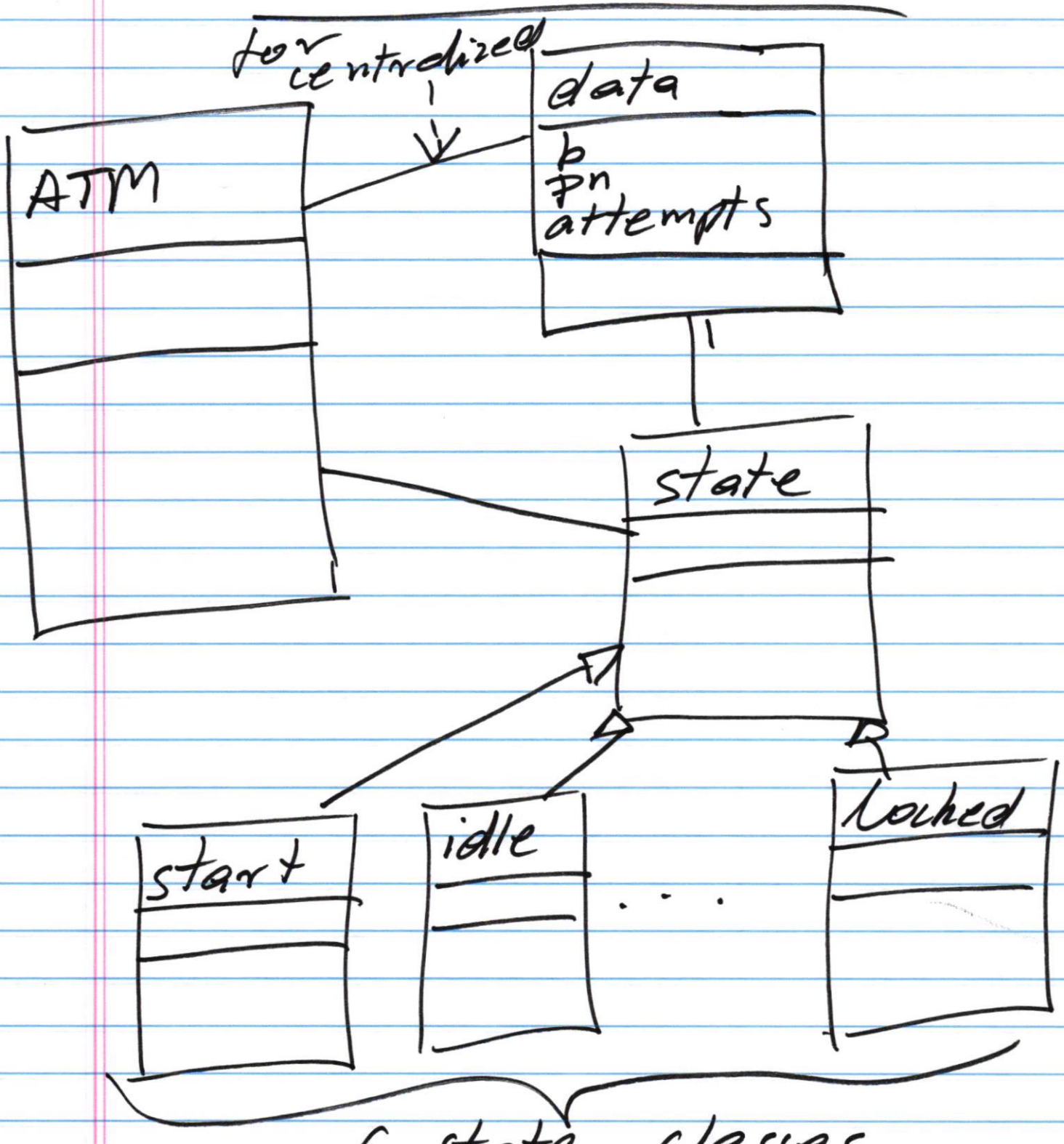
**De-centralized Pattern – Sequence Diagram**

*create(2.5), insert_cups(10), cash(3), coffee()*



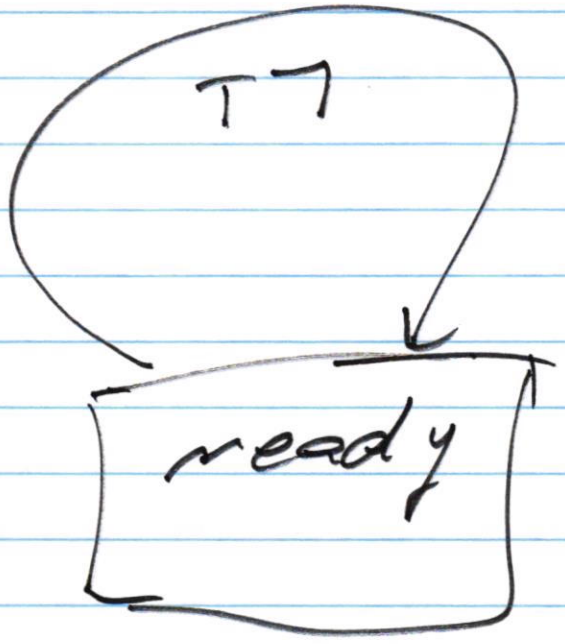
# Problem #2

## State Pattern



transition T7

withdraw(w) [b-w ≥ 1000] / b = b-w



# Adapter pattern
(wrapper)

**Problem:**

incompatible interfaces/signatures for services

---

Client

Server

m(int) → different names of services → p(int)

---

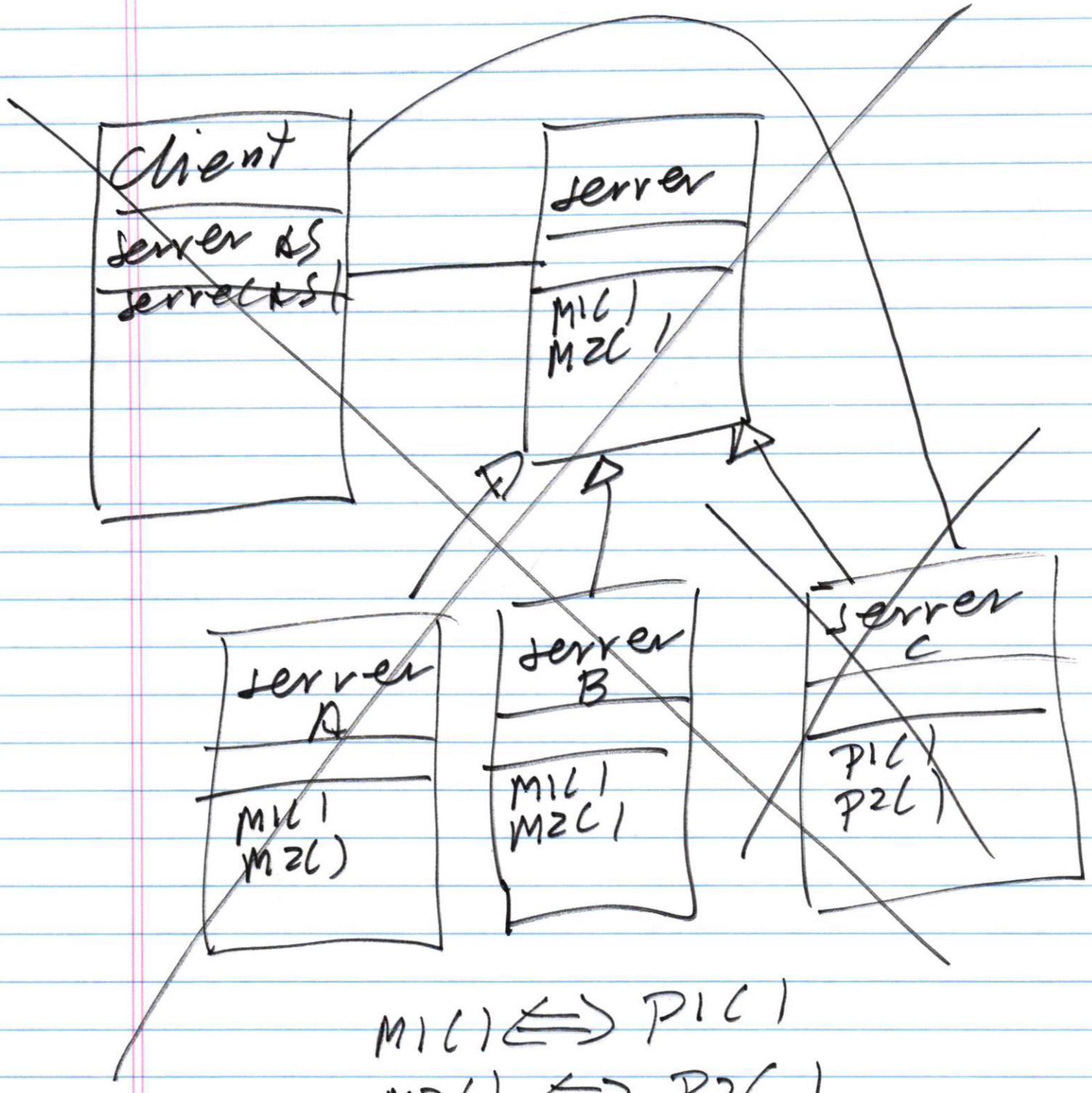
client

m(int, string)

server

m(string, int)

different signatures



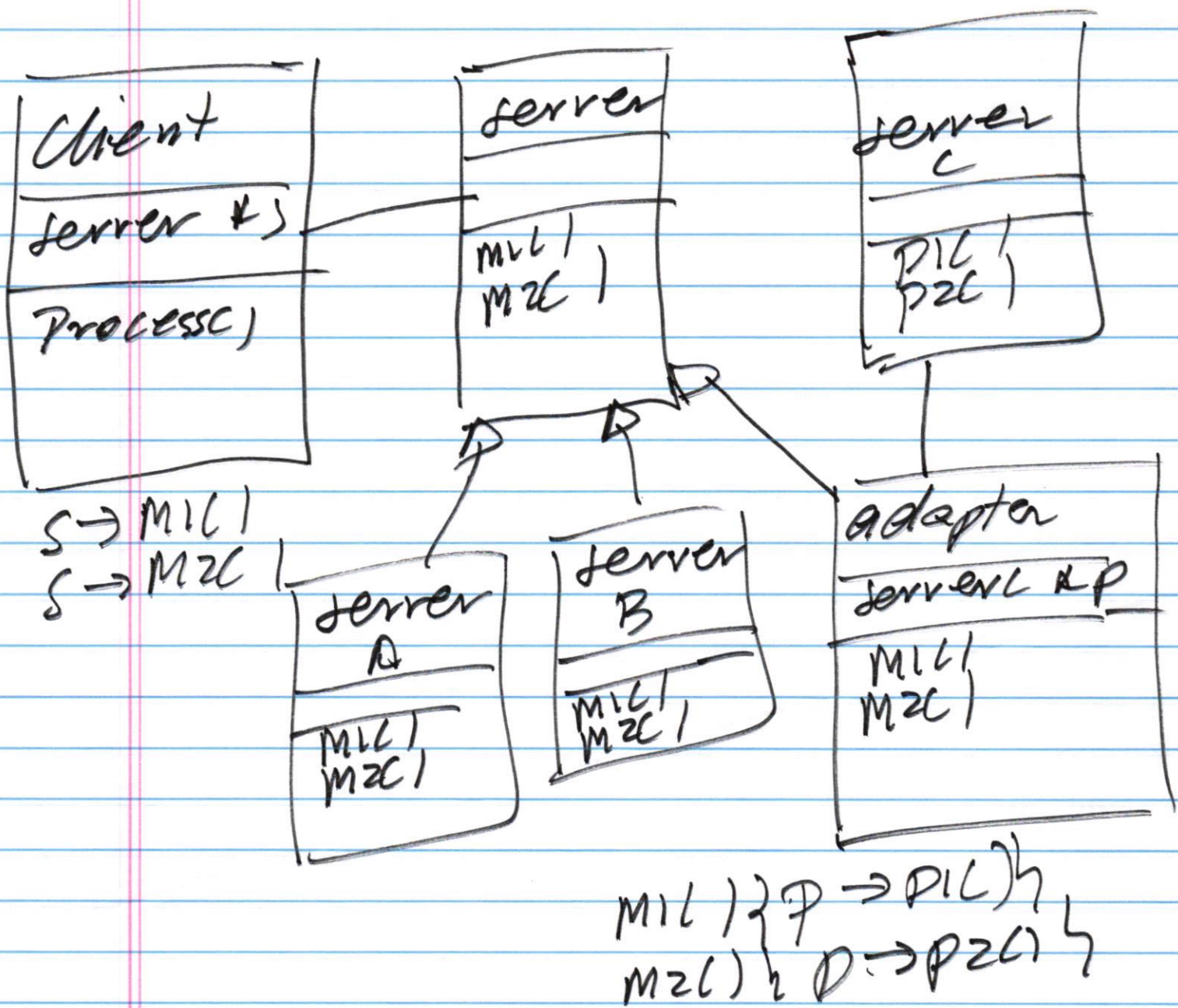

M1() ⟺ P1()

M2() ⟺ P2()

# Adapter Pattern

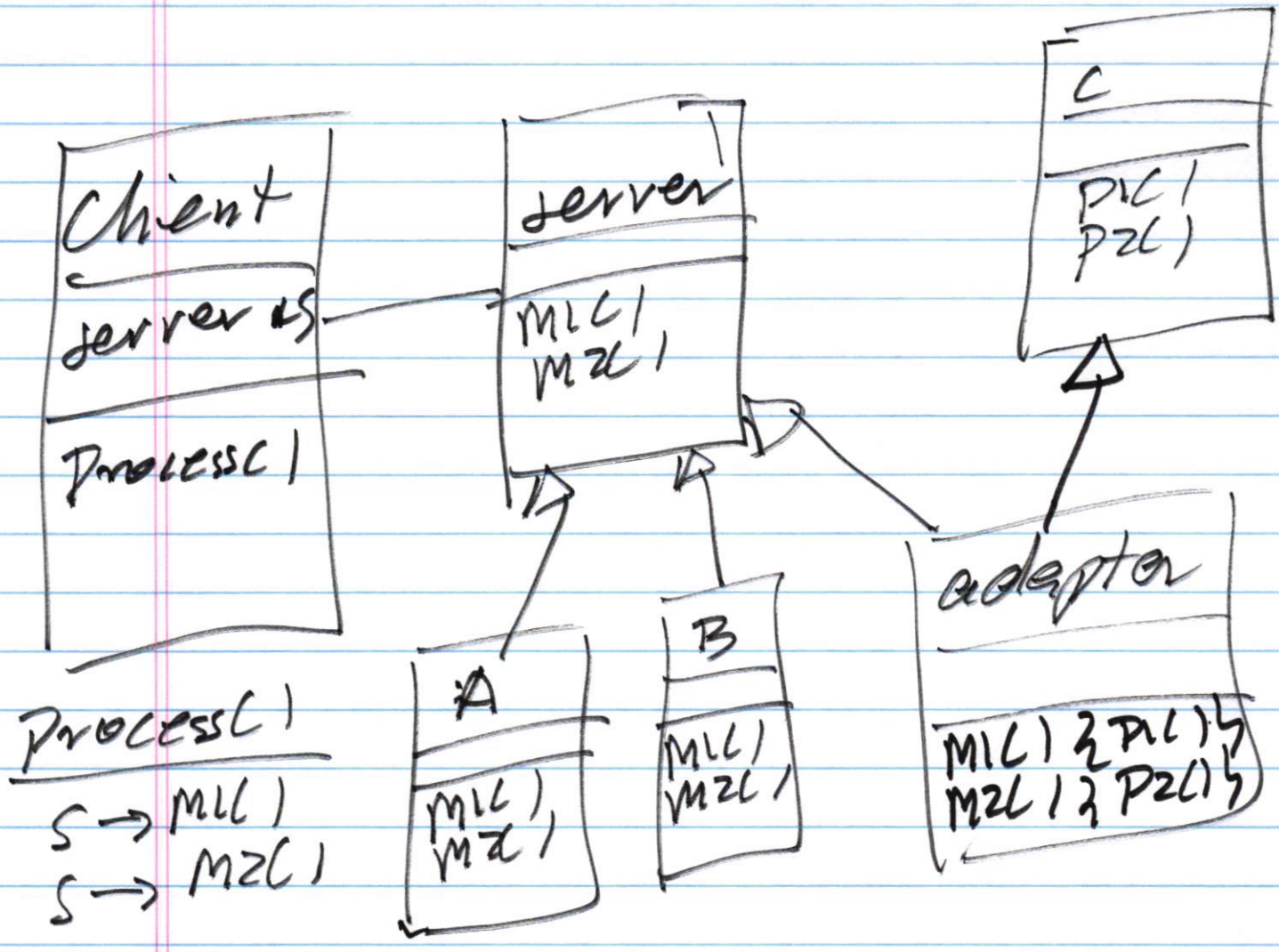
## solution #1

### Association-based solution

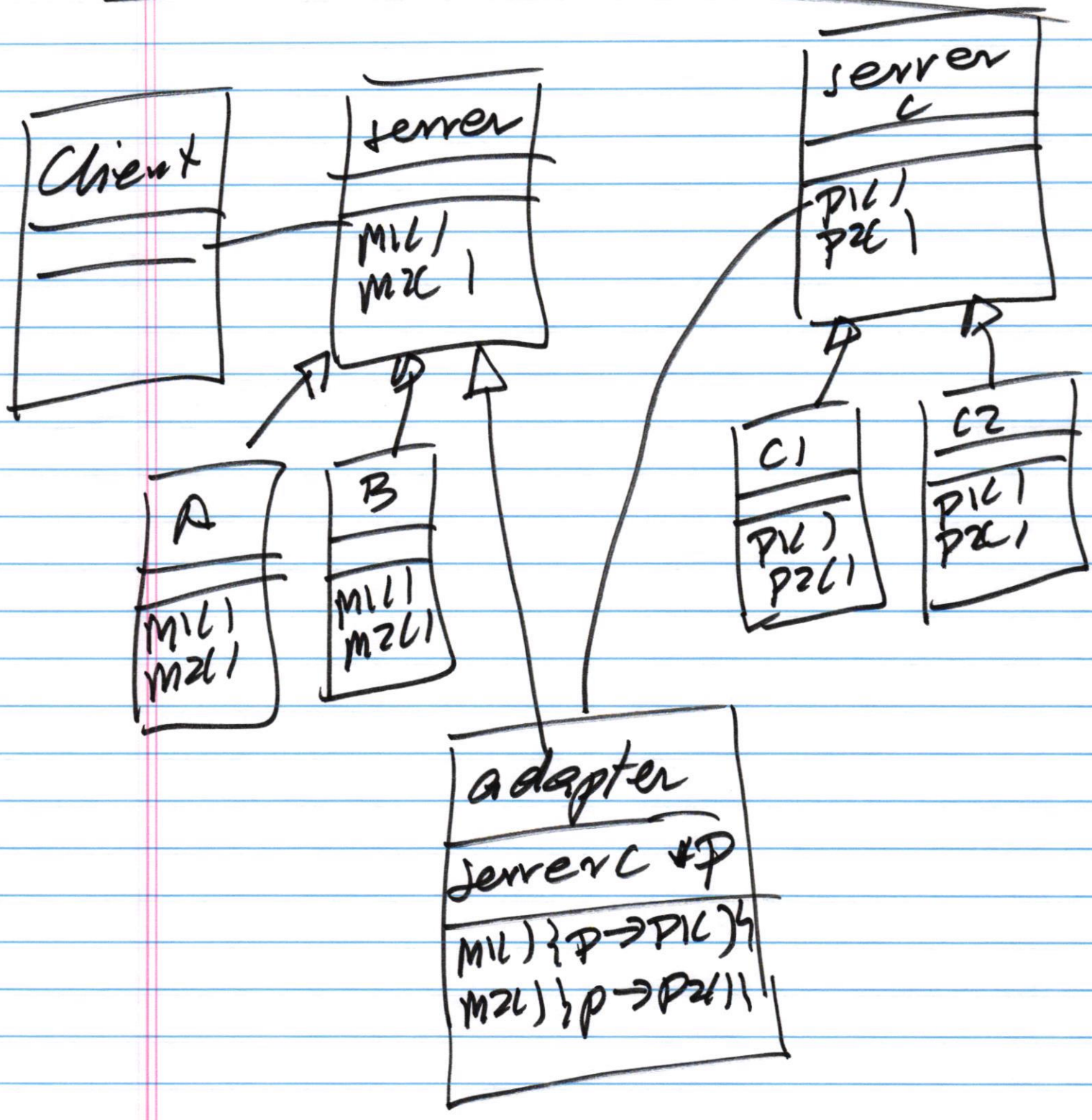


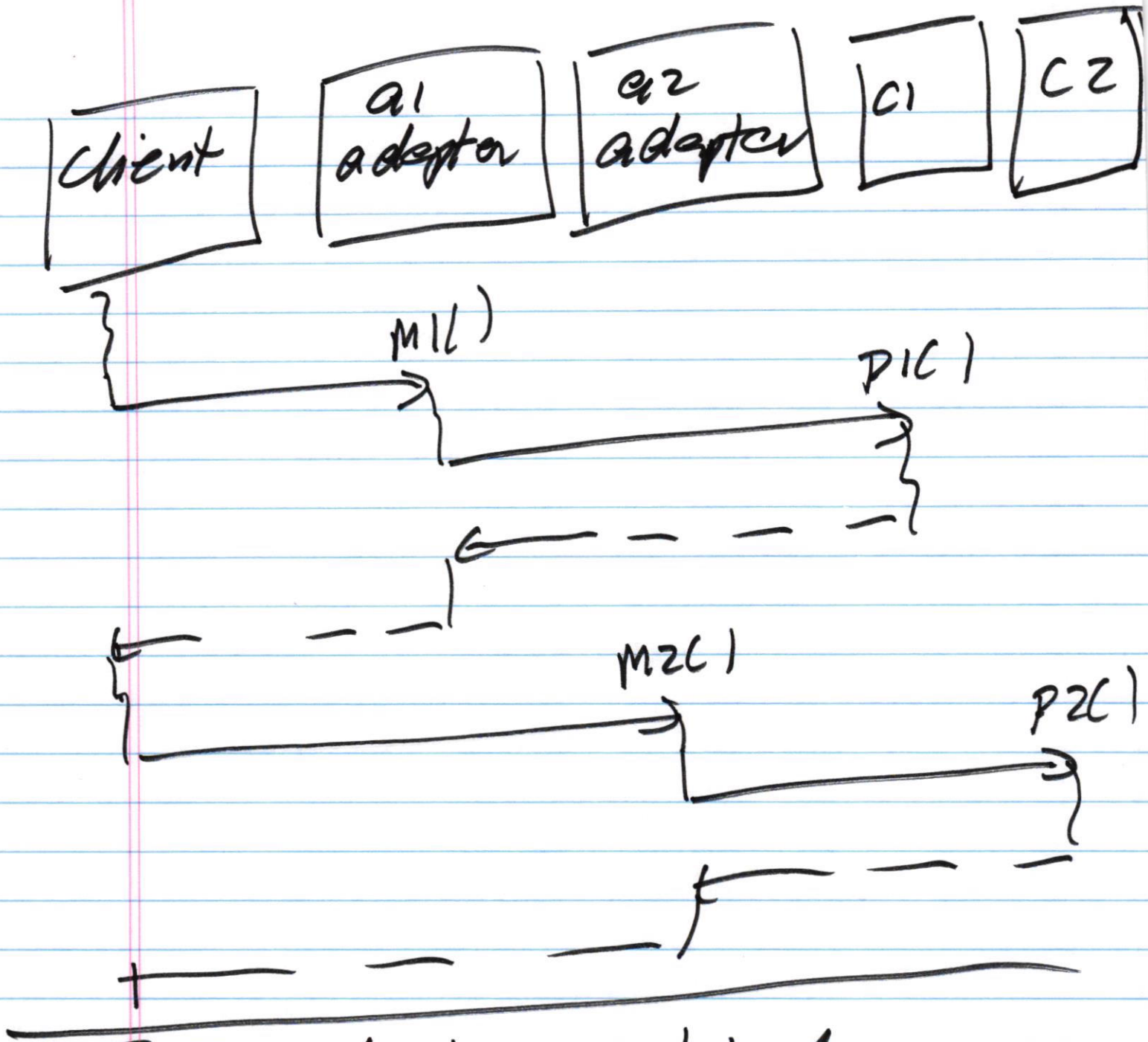
# solution #2

## inheritance-based solution



# association-based solution






Two adapter objects are created: a1 and a2

a1 → points to C1 object

a2 → points do C2 object