Homework #1

OO design patterns

- item description patterns
- whole-part —"—
- observer —"—
- state —"—
- adapter —"—
- strategy —"—
- abstract factory —"—
- ...
- ...

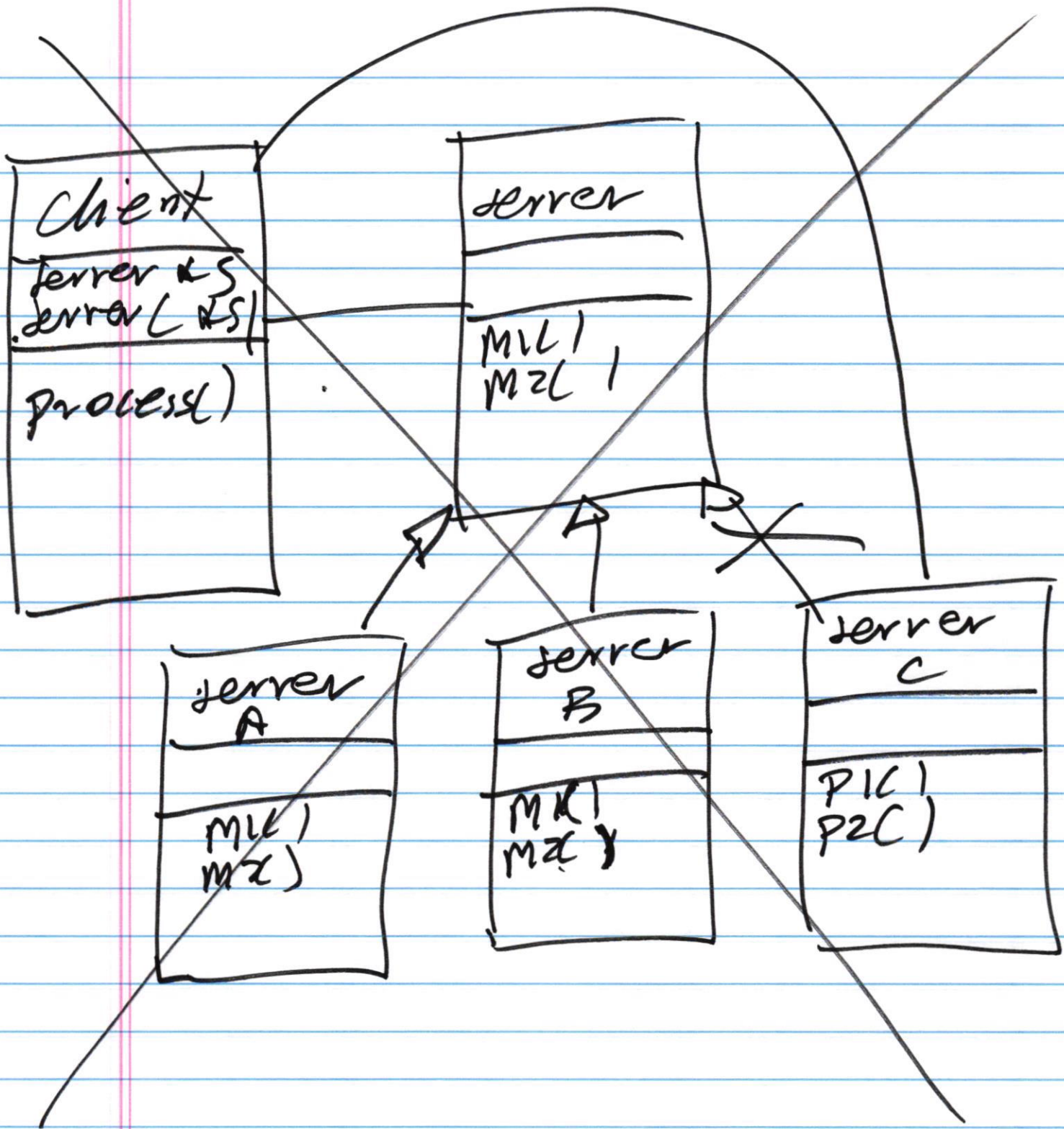
Adapter pattern

Problem:

incompatable interfaces/signatures for services

---

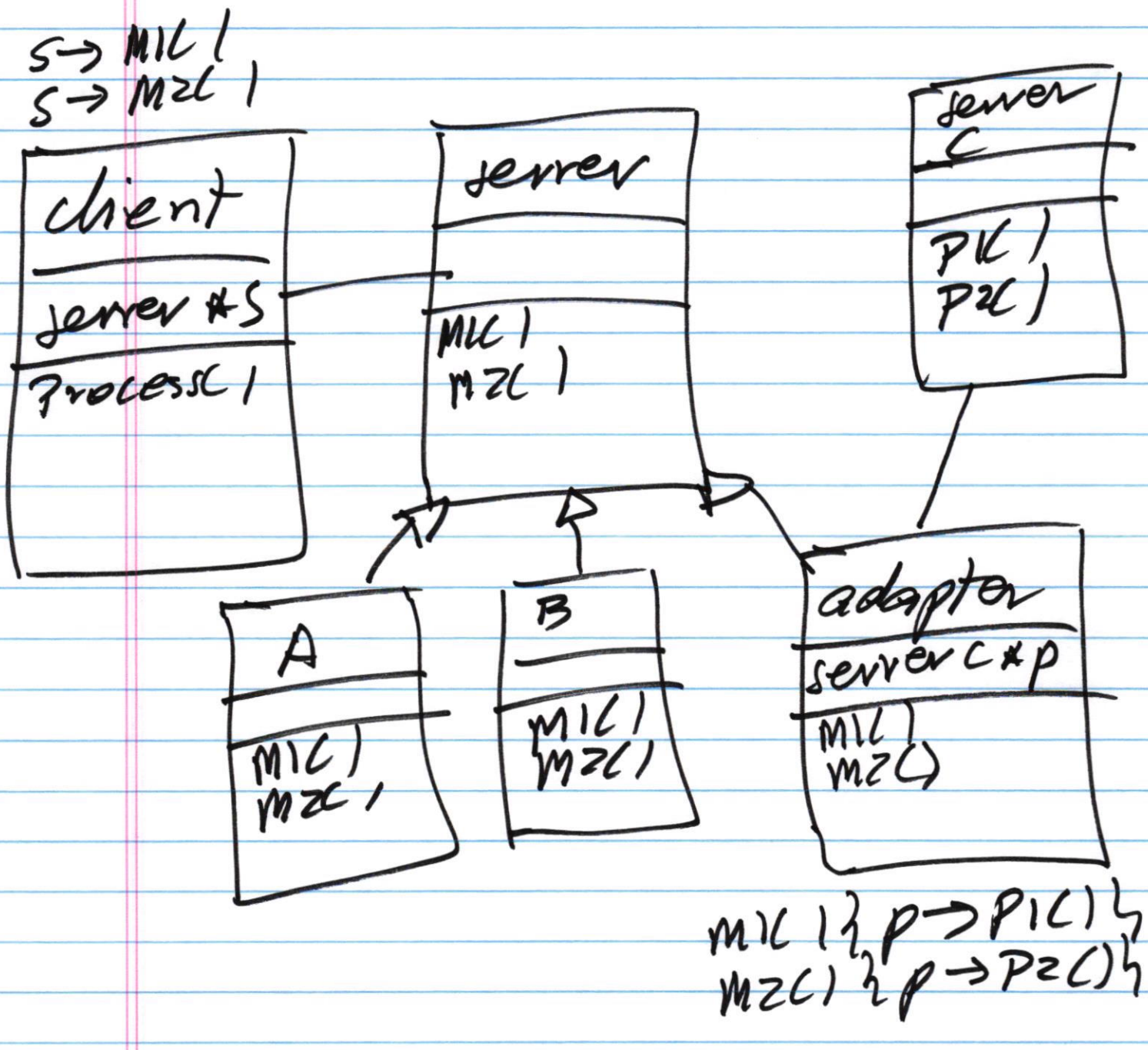
client	server
M(int)	P(int)
	← different names
M(int, String)	M(String, int)
	← different signature


Client
Server *S
Server(*S)
process()
Server
M1()
M2()
Server A
M1()
M2()
Server B
M1()
M2()
Server C
P1()
P2()

Adapter pattern

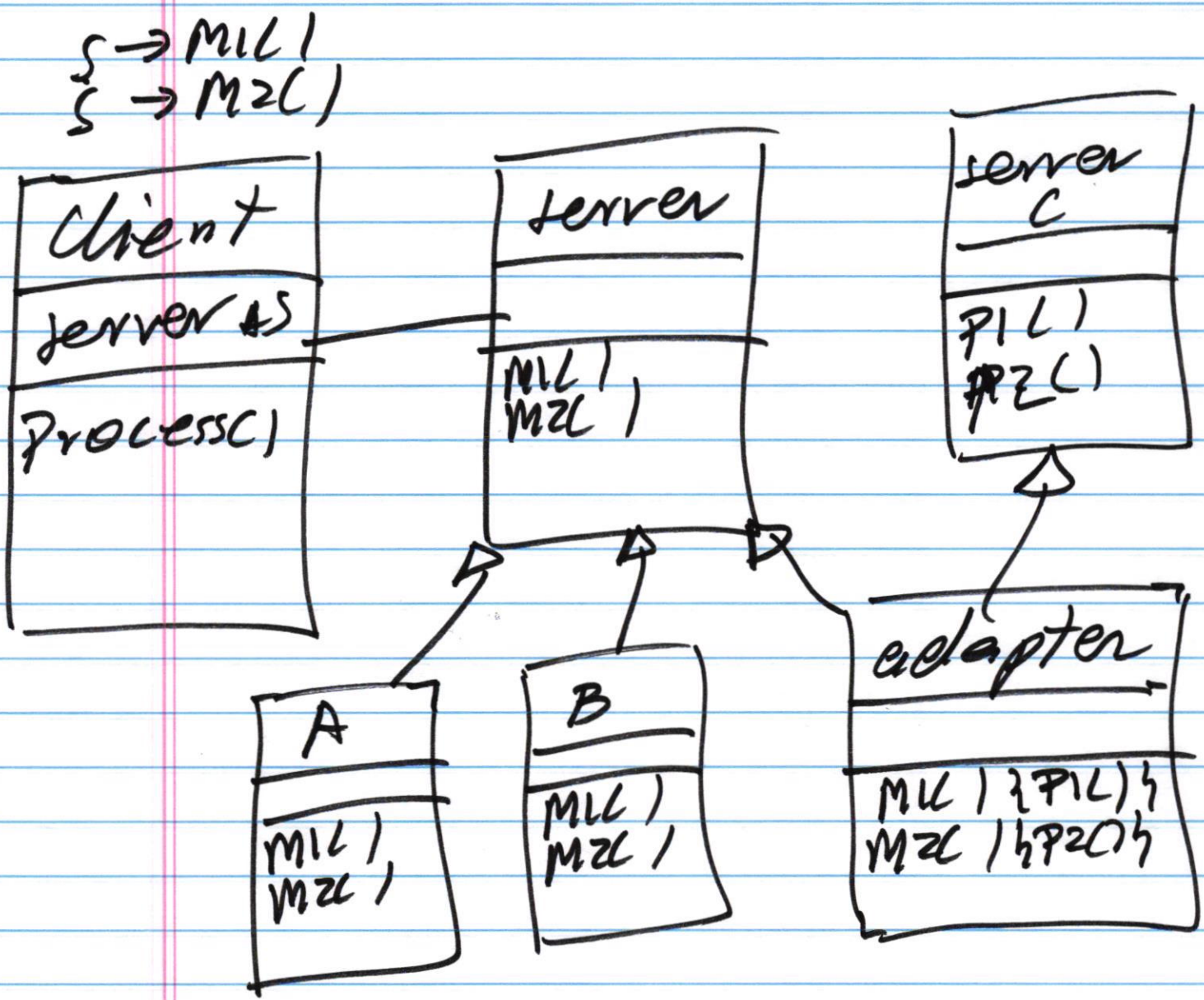
solution #1

association-based solution

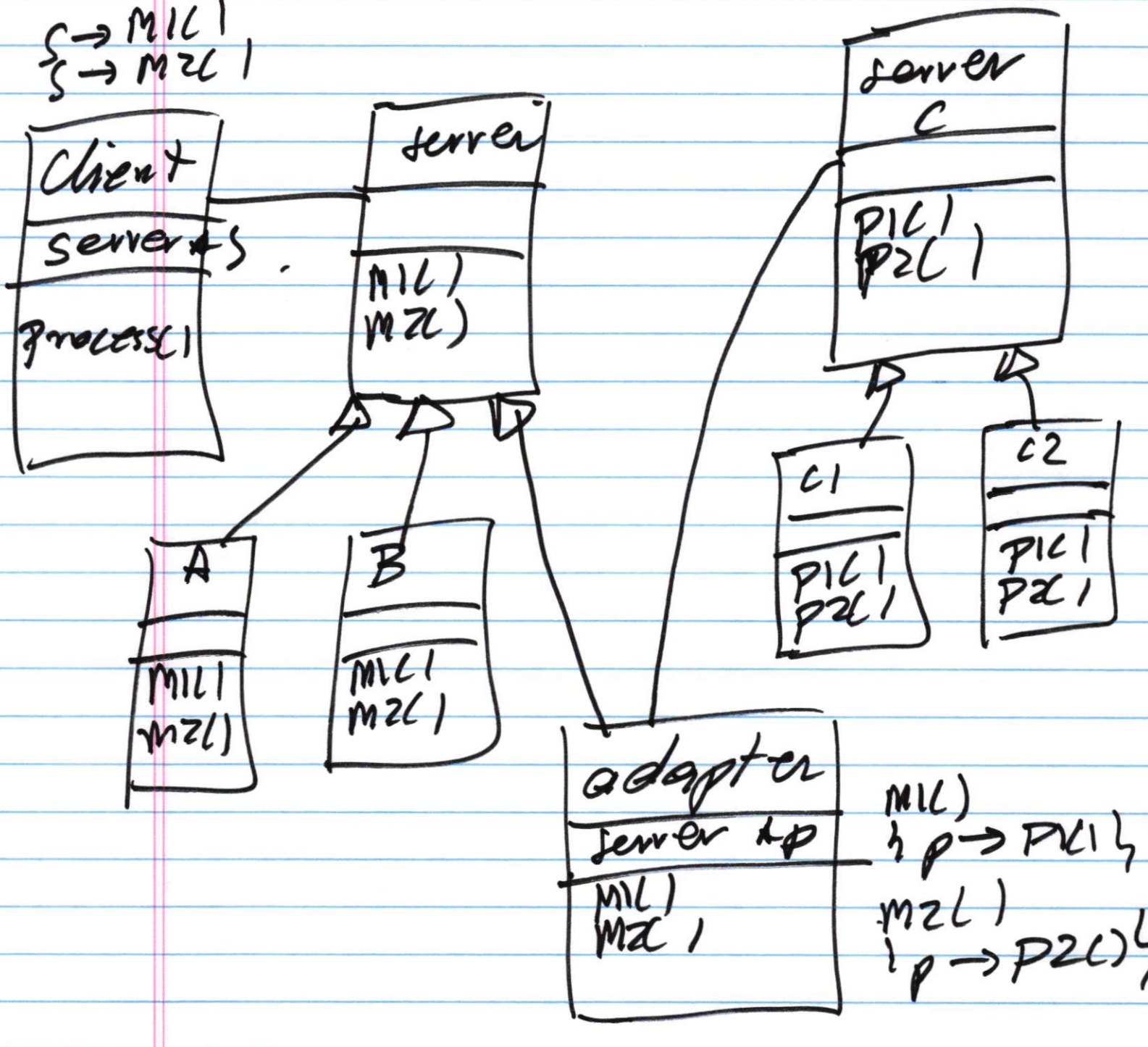


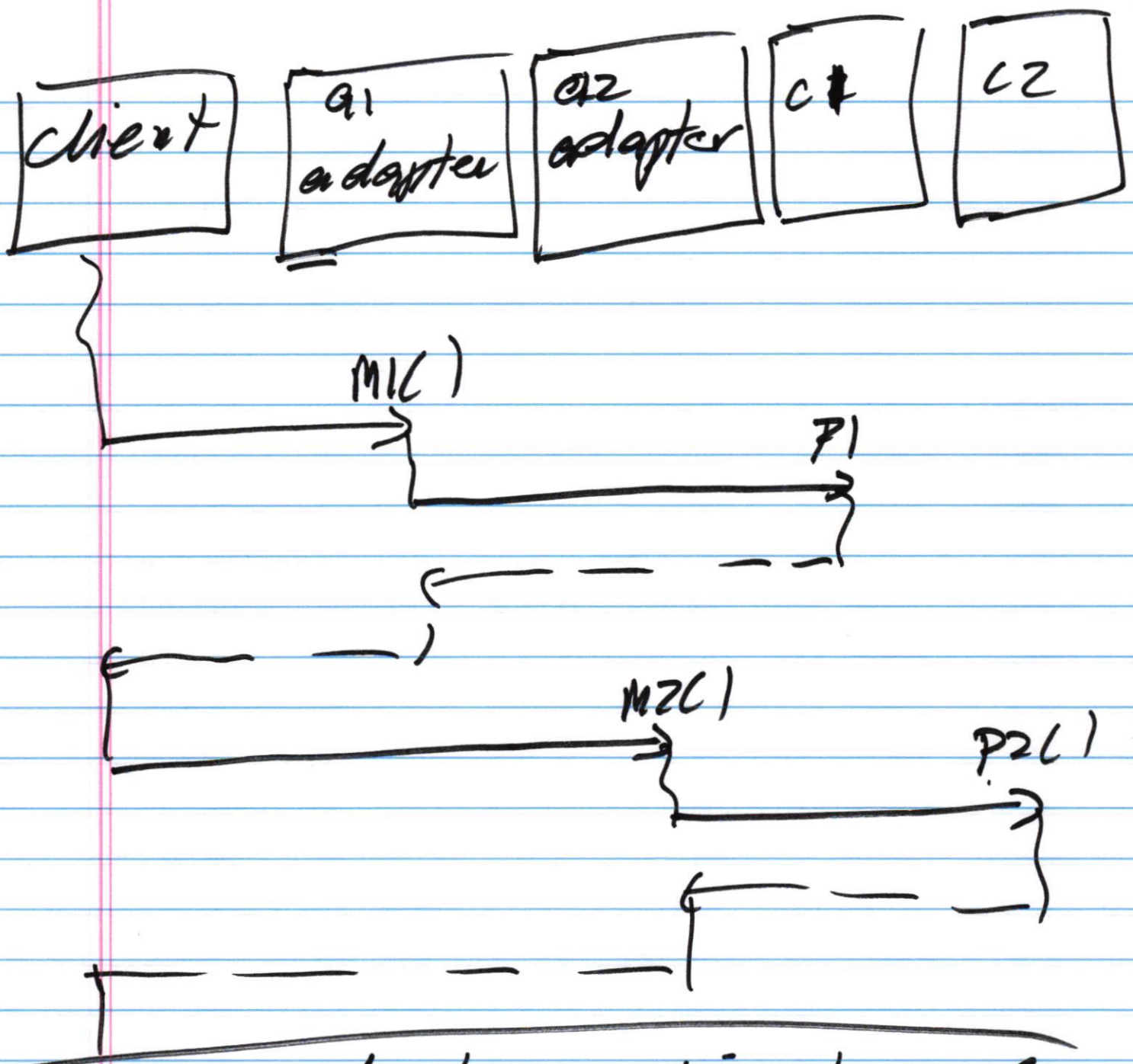
solution #2

inheritance-based solution



association-based solution



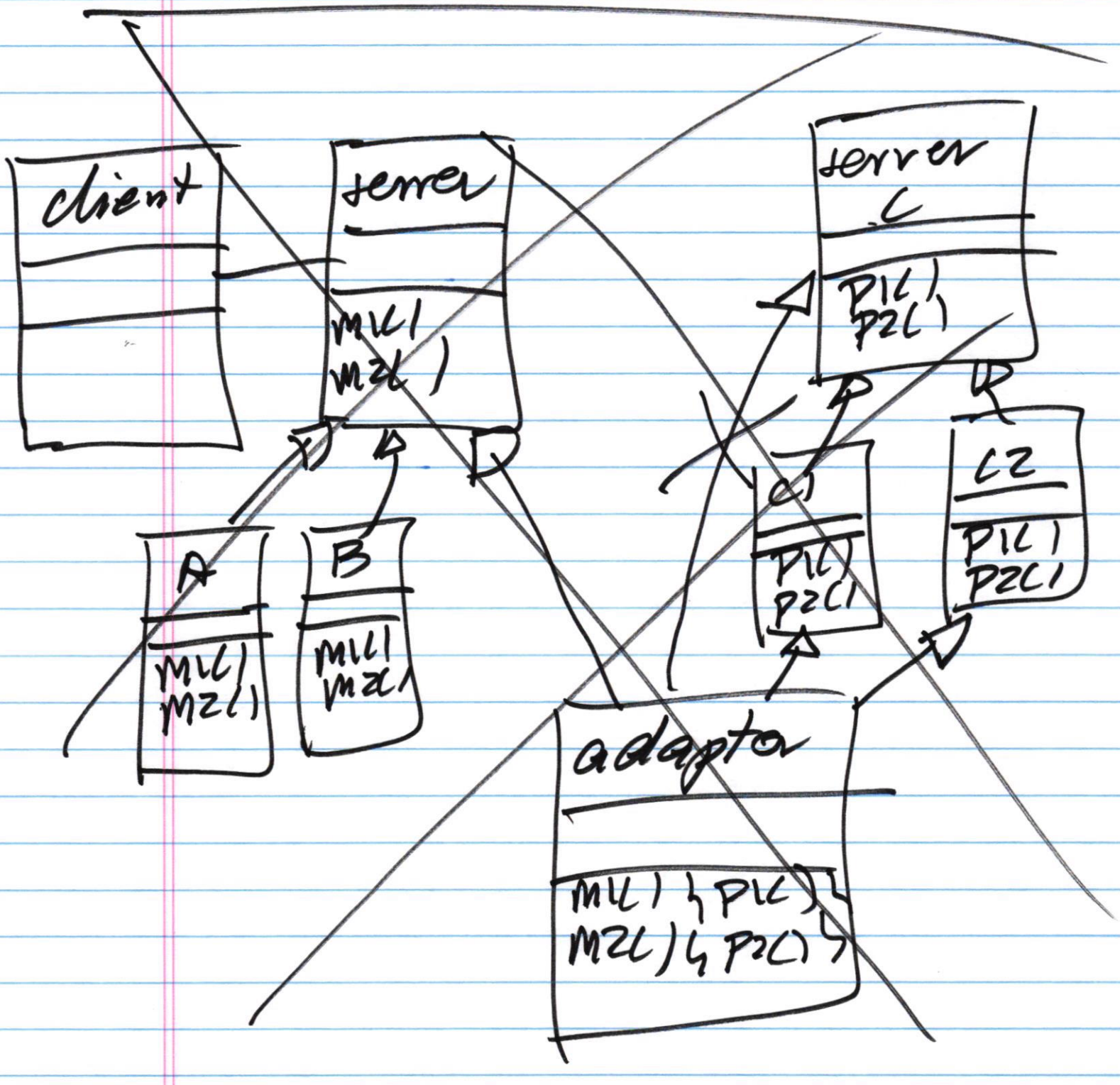


Two adapter objects are created "a1" and "a2"

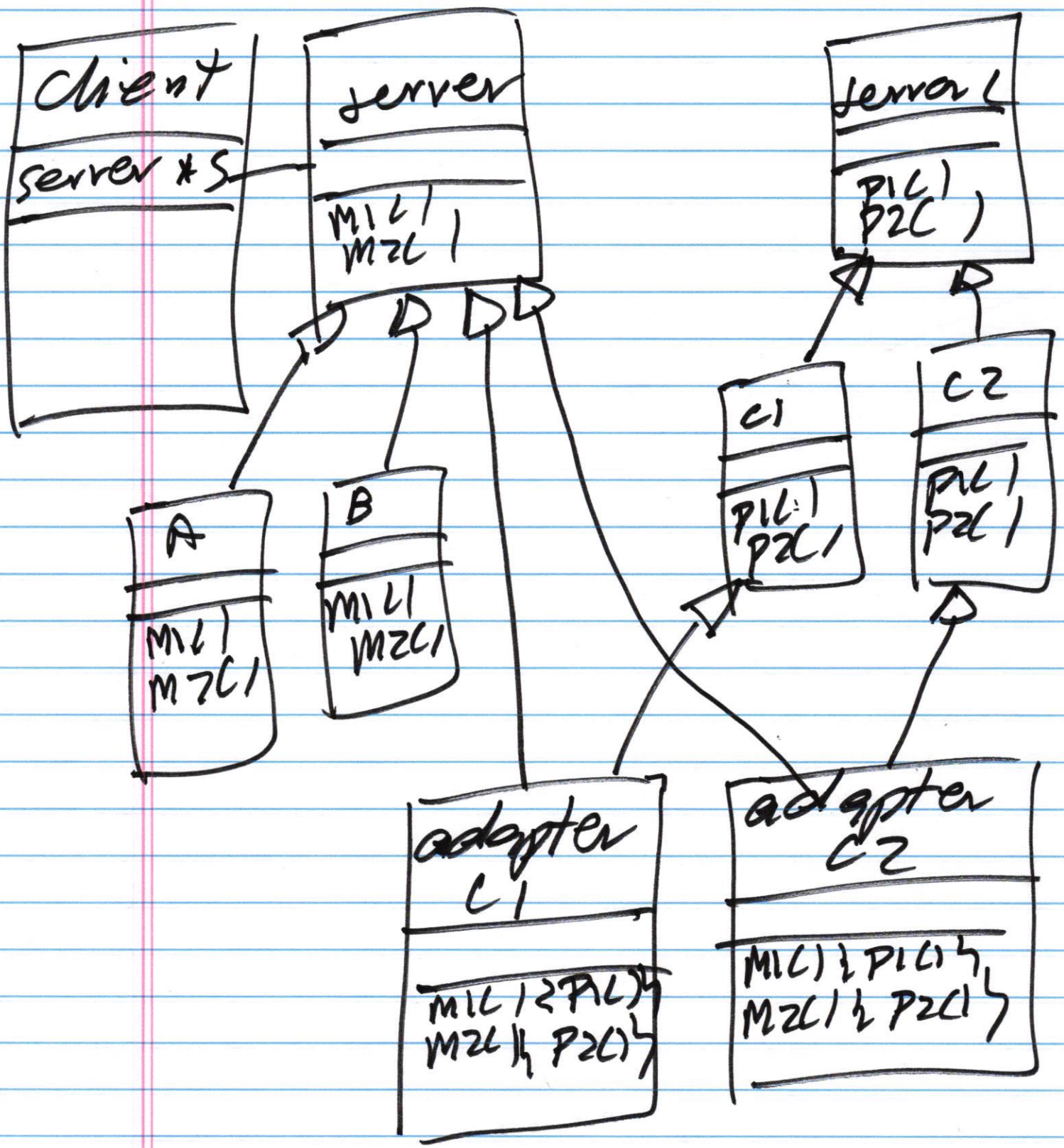
a1 → points to object C1

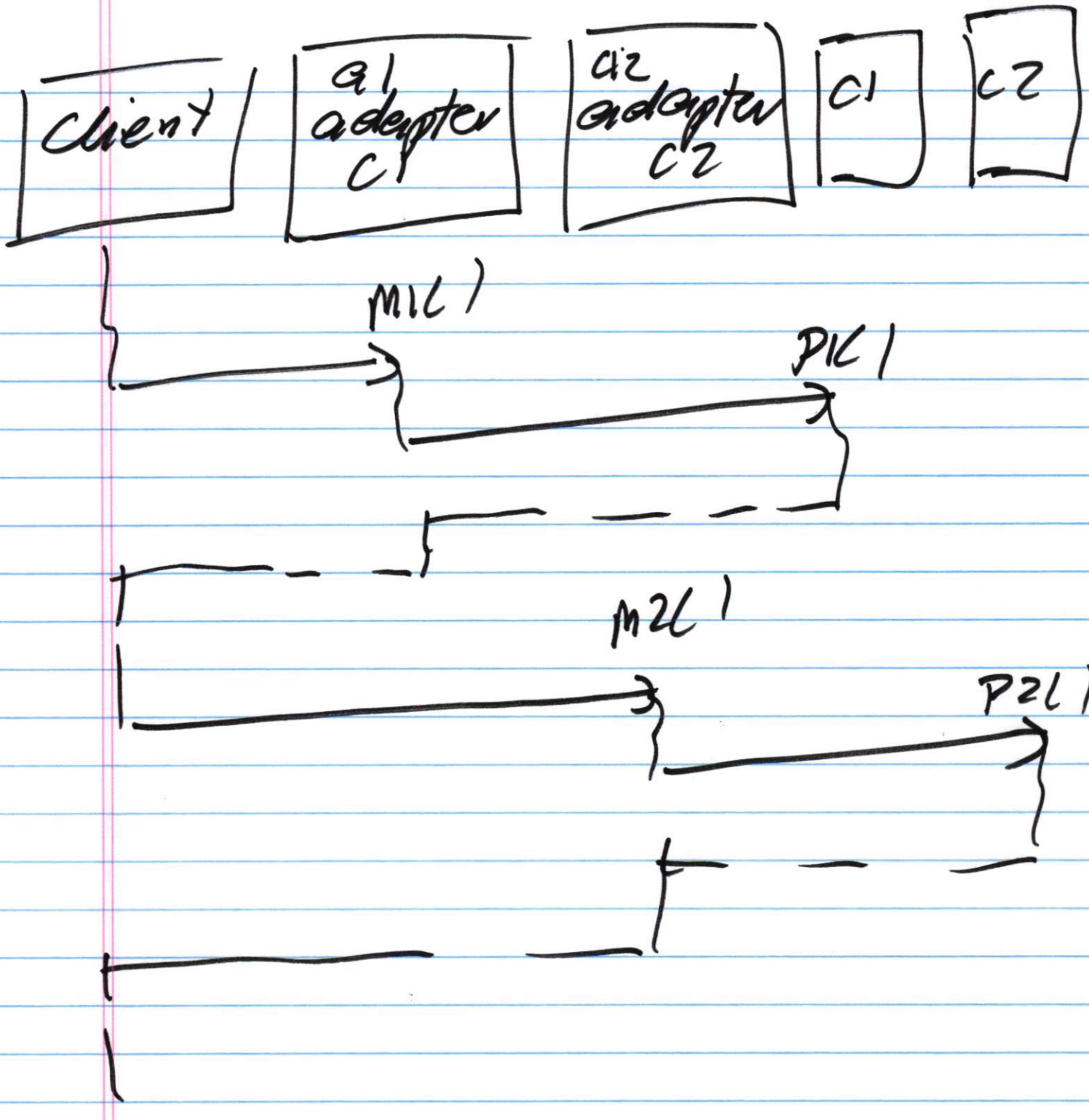
a2 → points to object C2

inheritance based solution



S → M1 M2


client
server *S
server
M1()
M2()
A
M1()
M2()
B
M1()
M2()
server2
P1()
P2()
C1
P1()
P2()
C2
P1()
P2()
adapter C1
M1() { P1() }
M2() { P2() }
adapter C2
M1() { P1() }
M2() { P2() }


client
a1 adapter C1
a2 adapter C2
C1
C2
M1C1
P1C1
M2C1
P2C1

strategy pattern

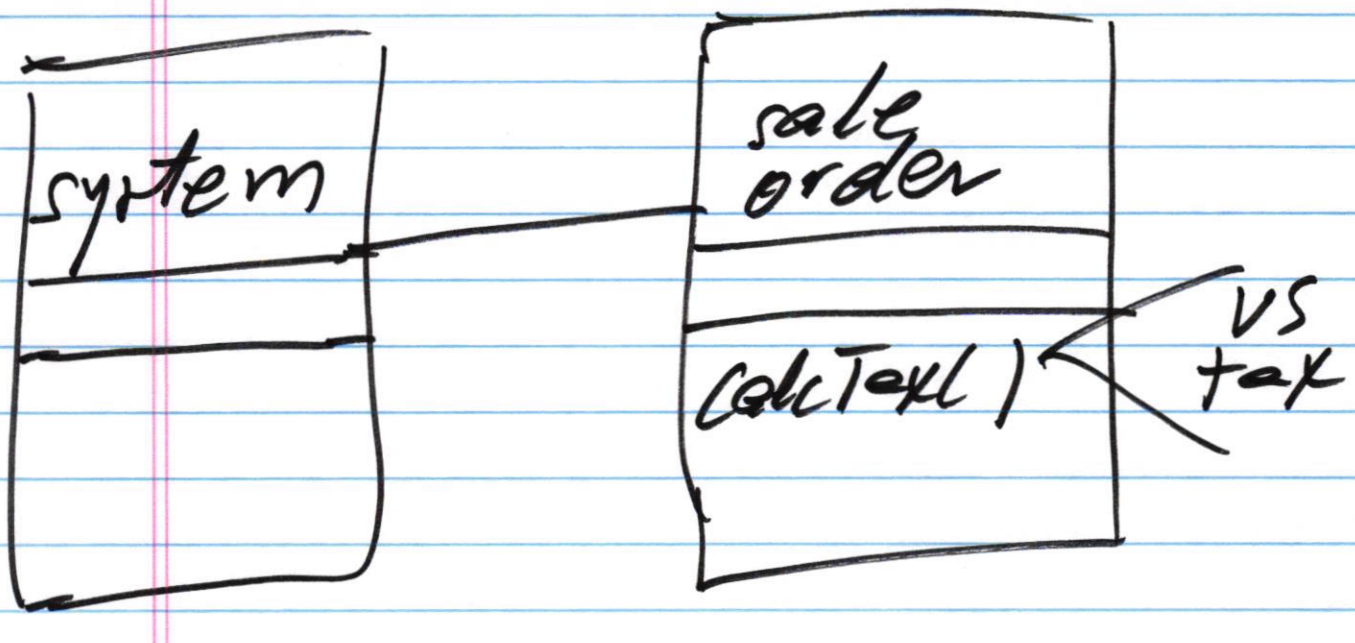
* a family of algorithms / requirements.

solution:

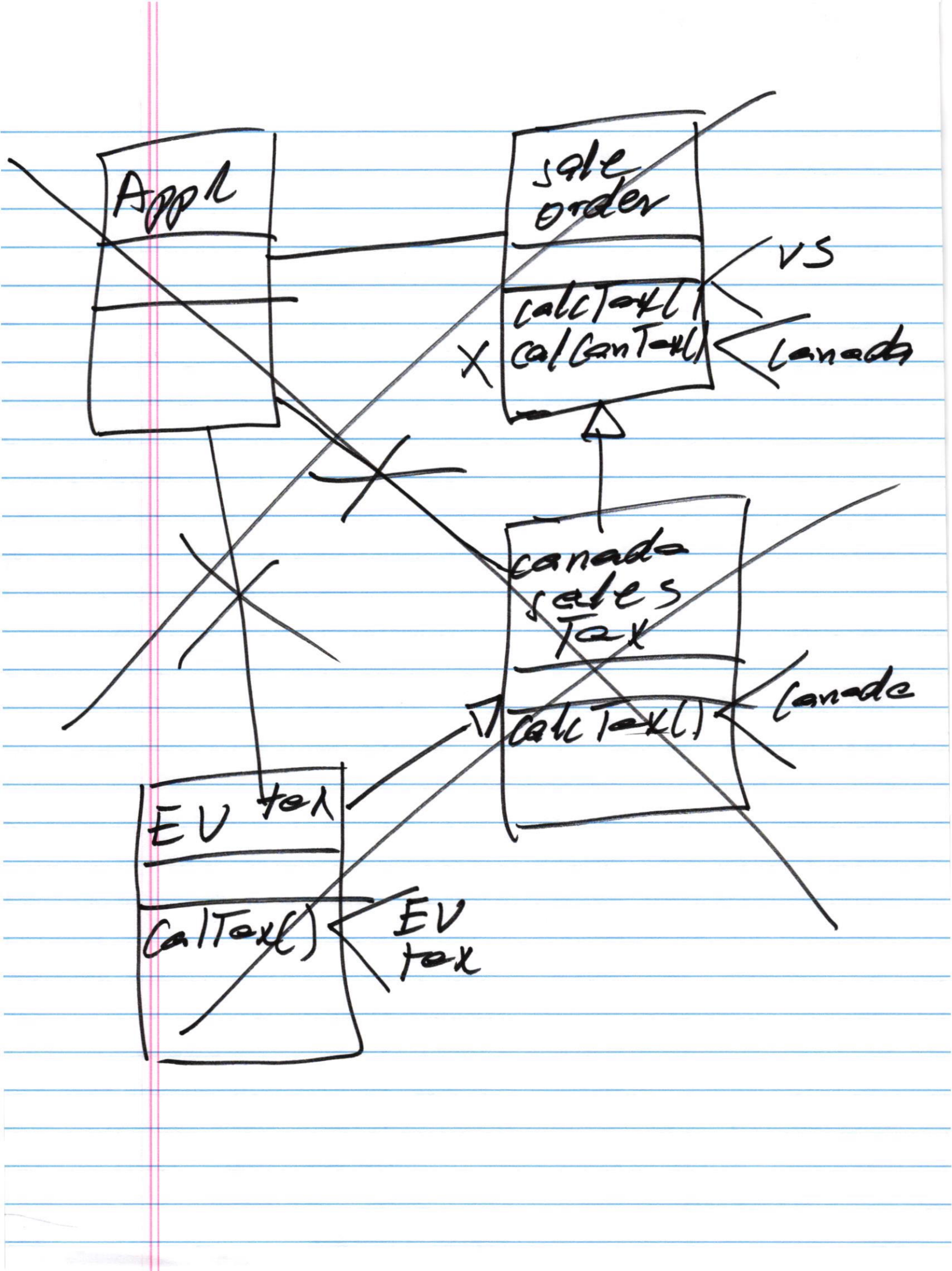
encapsulate each of them in a seperate class

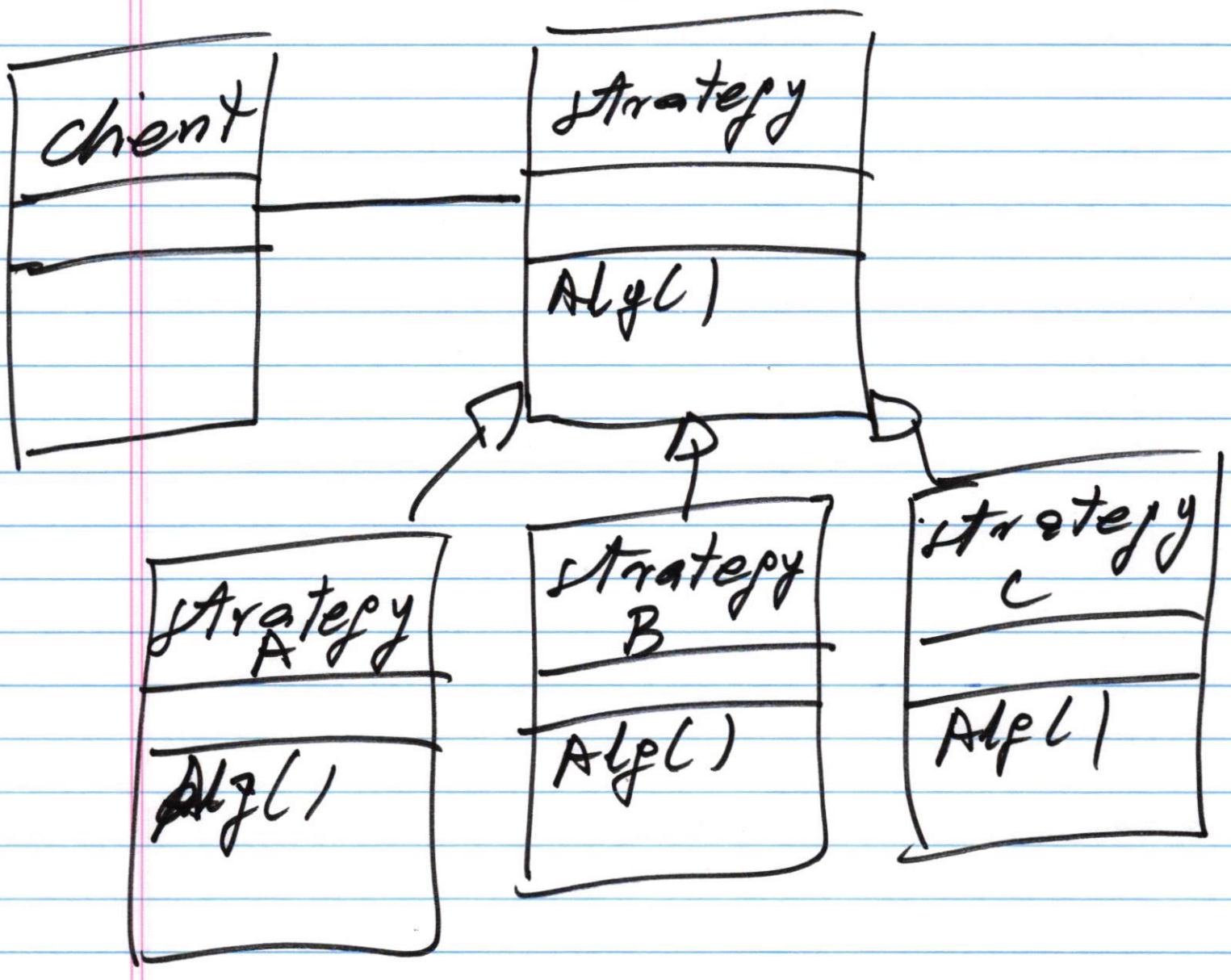
strategy pattern

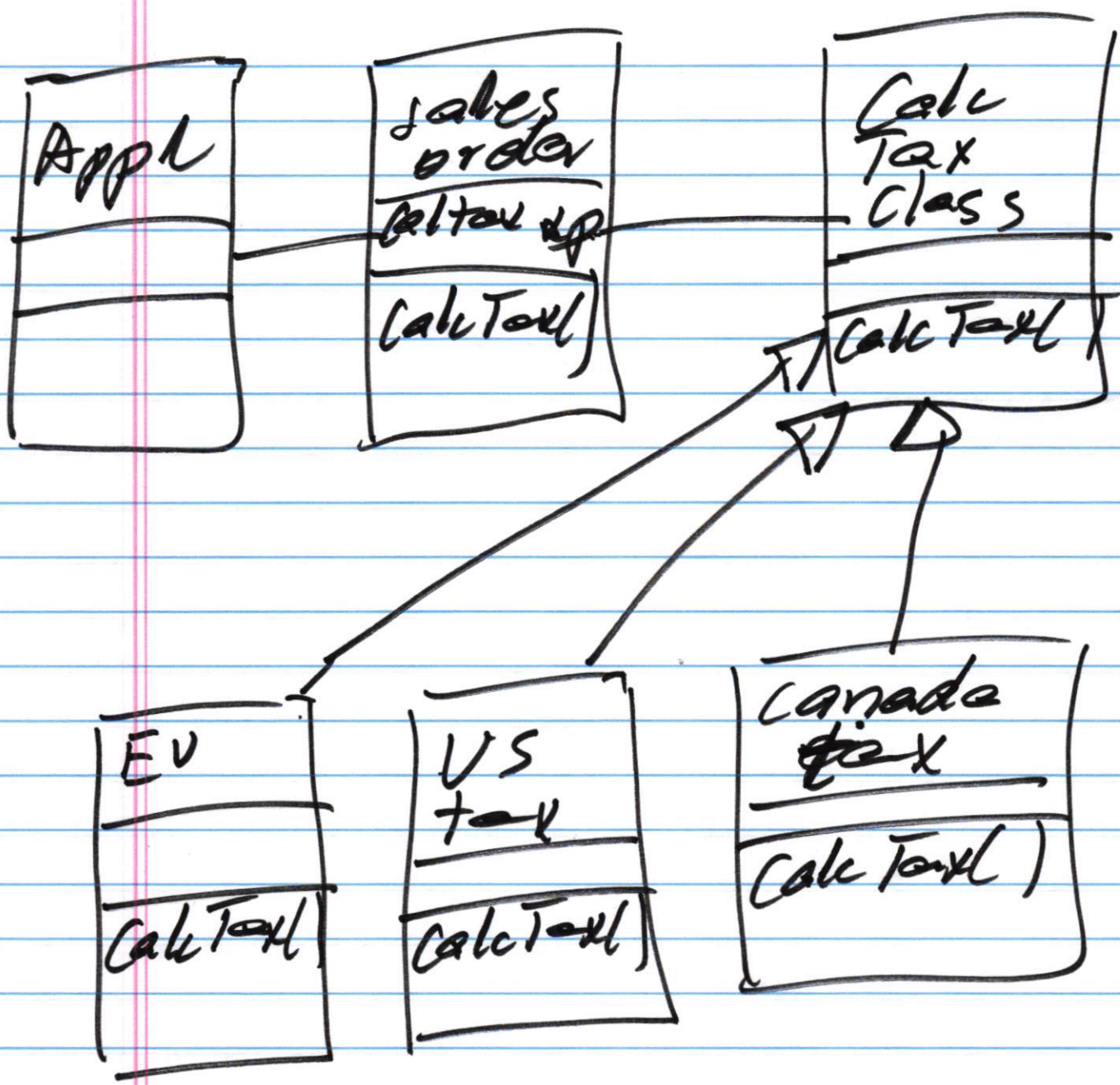
new requirement



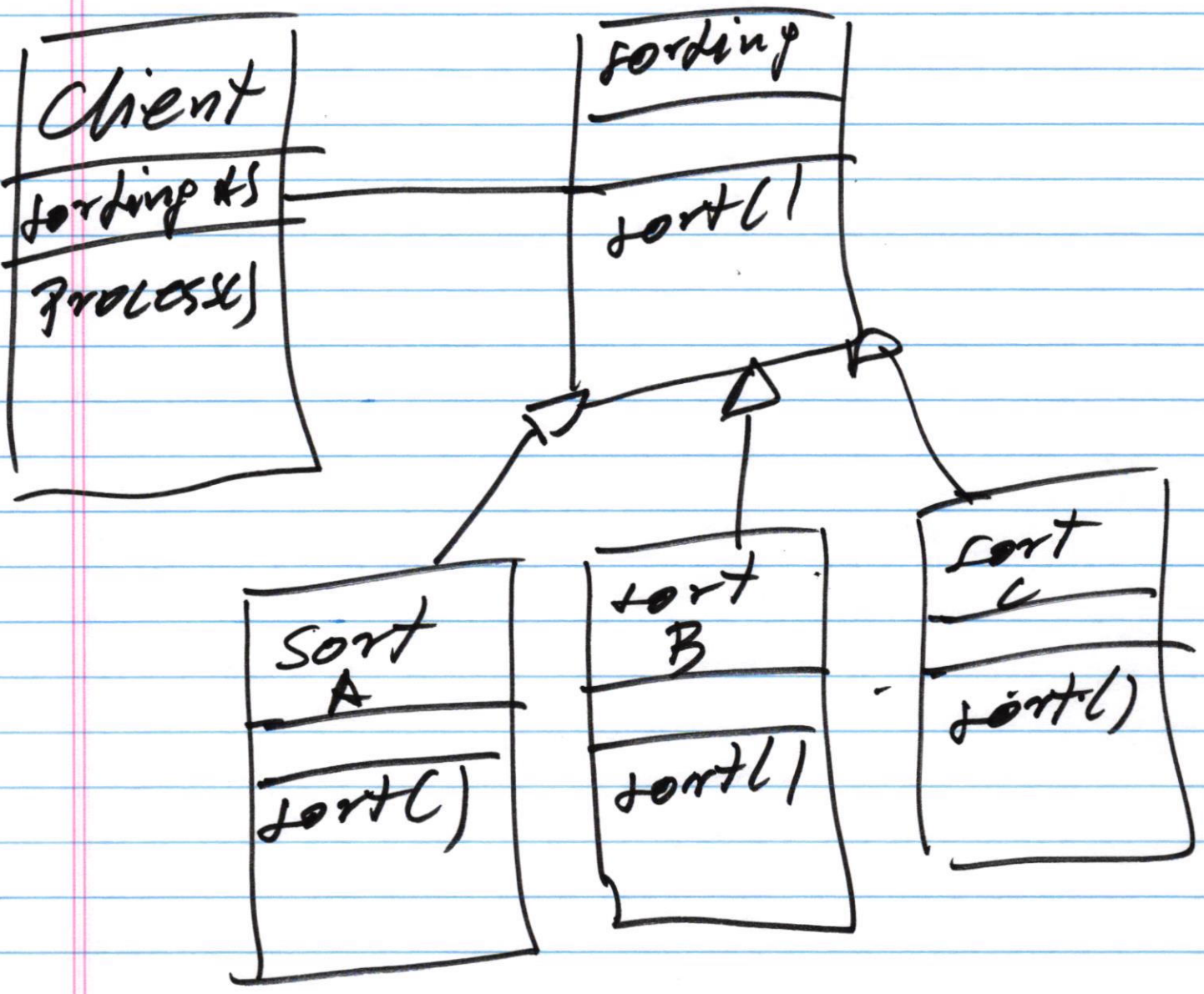
1. fill out order
2. process
:
6. handling tax computation


Appl
sale order
calcTax()
calCanTax()
VS
X
canada
canada sales Tax
calcTax()
canada
EV tax
calTax()
EV tax


Client
Strategy
Alg()
Strategy A
Alg()
Strategy B
Alg()
Strategy C
Alg()


Appl
sales order
calTax ref
calcTax()
Calc Tax Class
calcTax()
EU
calcTax()
US tax
calcTax()
canada tax
calcTax()

s->sort()


Client
sortingAS
Process()
Sorting
sort()
Sort A
sort()
Sort B
sort()
Sort C
sort()

Abstract Factory Pattern

Problem

objects need to be created in some coordinated way.

Application needs to be de-coupled from the problem of creating these objects.

drivers < print
display.

Display

LRDD: Low Resolution
Display Driver

HRDD: High Resolution
Display Driver

Print

LRPD: Low Resolution
Print Driver

HRPD: High Resolution
Print Driver

Low Capacity Machine

LRDD
LRPD

High Capacity Machine

HRDD
HRPP

Midrange Machine

HRDD
LRPD

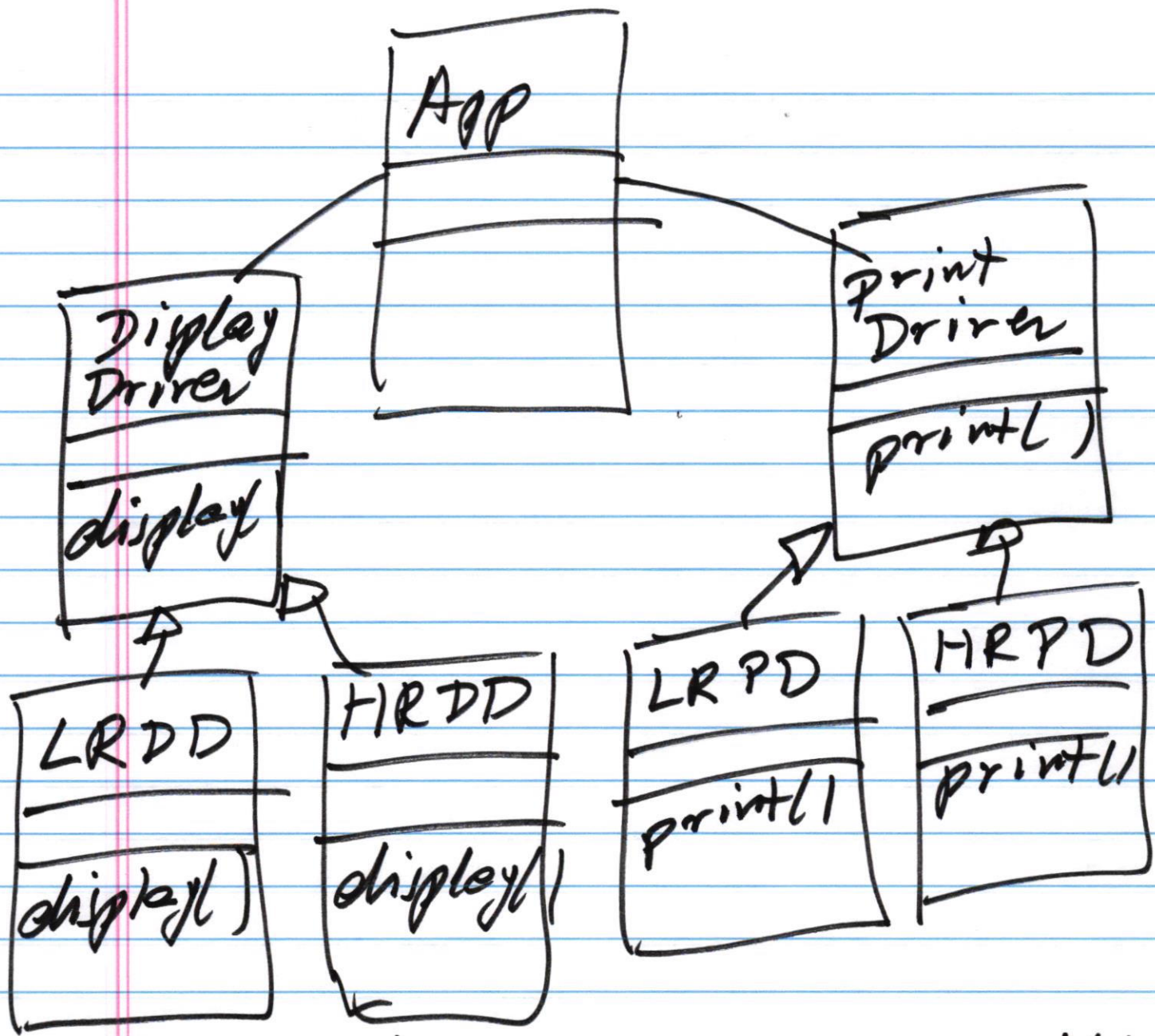
Display : 10 types of drivers

Print: 10 types of driver

$10 \cdot 10 = 100$ configurations

D1: 10 types
D2: 10 —"—
⋮
D10: 10 —"—

# of configurations $= 10^{10}$



Application is responsible for creating "correct" drivers for different configurations.

Bad design!!

Homework #1

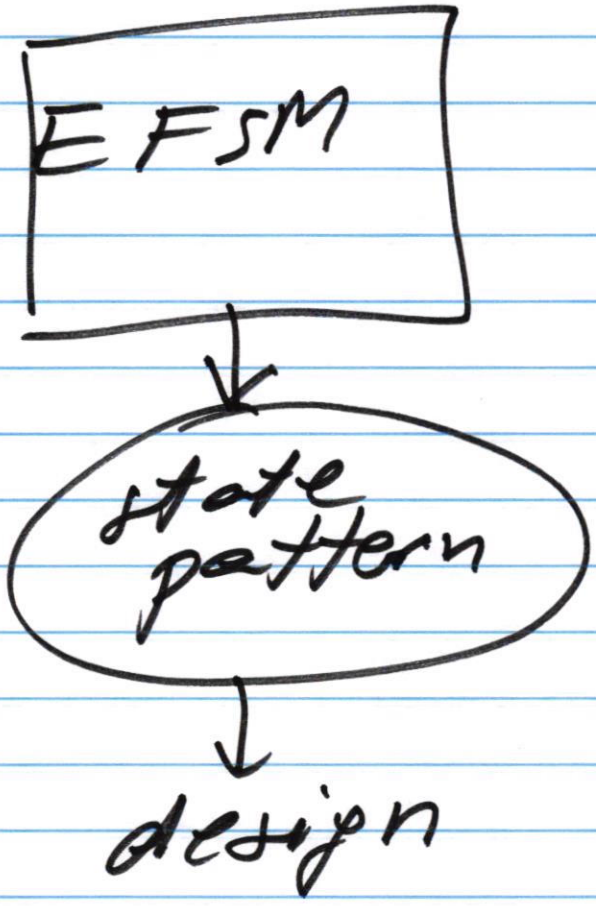
September 18

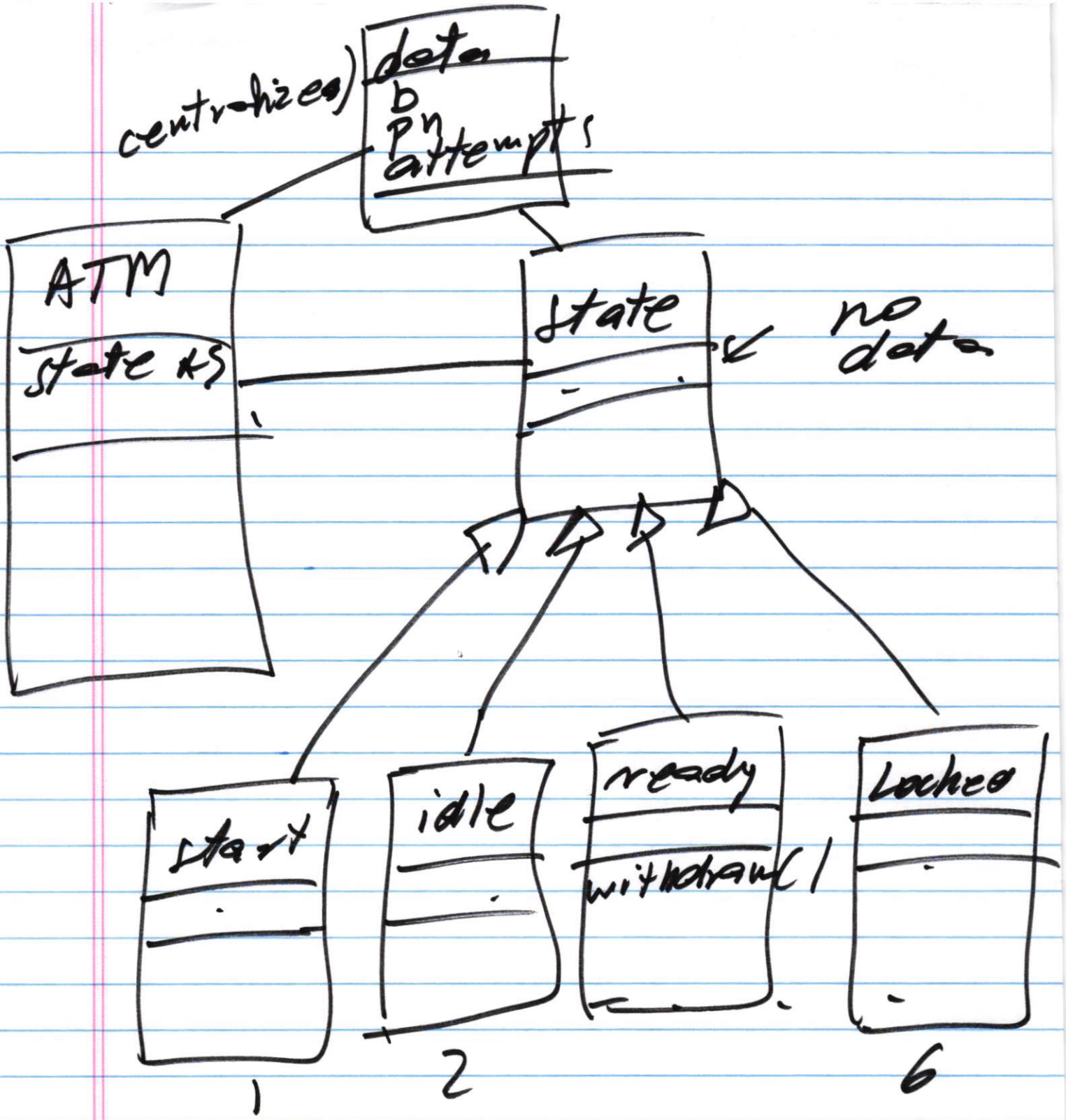
late 50% penalty.

After September 23
the homework will not
be accepted.

Problem #2

State Pattern




centralized
data
b
pn
attempts
ATM
state *S
state
no data
start
idle
ready
withdraw(
locked
1
2
6

de-centralized

ATM class

create()
s → create()

balance()
s → balance()

centralized solution

withdraw(w)[b-w ≥ 1000] /

b=b-w

ATM class

withdraw(w)

S → withdraw(w)
change state.

```
if (b-w ≥ 1000)
    b = b-w
    change state.
```

OK.

~~X~~

```
b = b-w
if (b-w ≥ 1000
    then change state
```

wrong!!

* class diagram

* sequence diagram.

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.

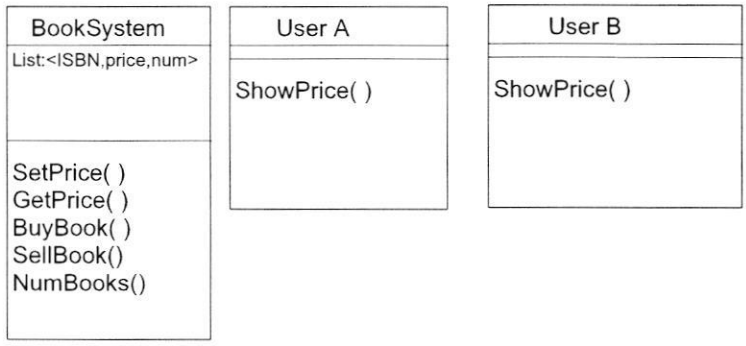
This is an **individual** assignment. **Identical or similar** solutions will be penalized.

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**.

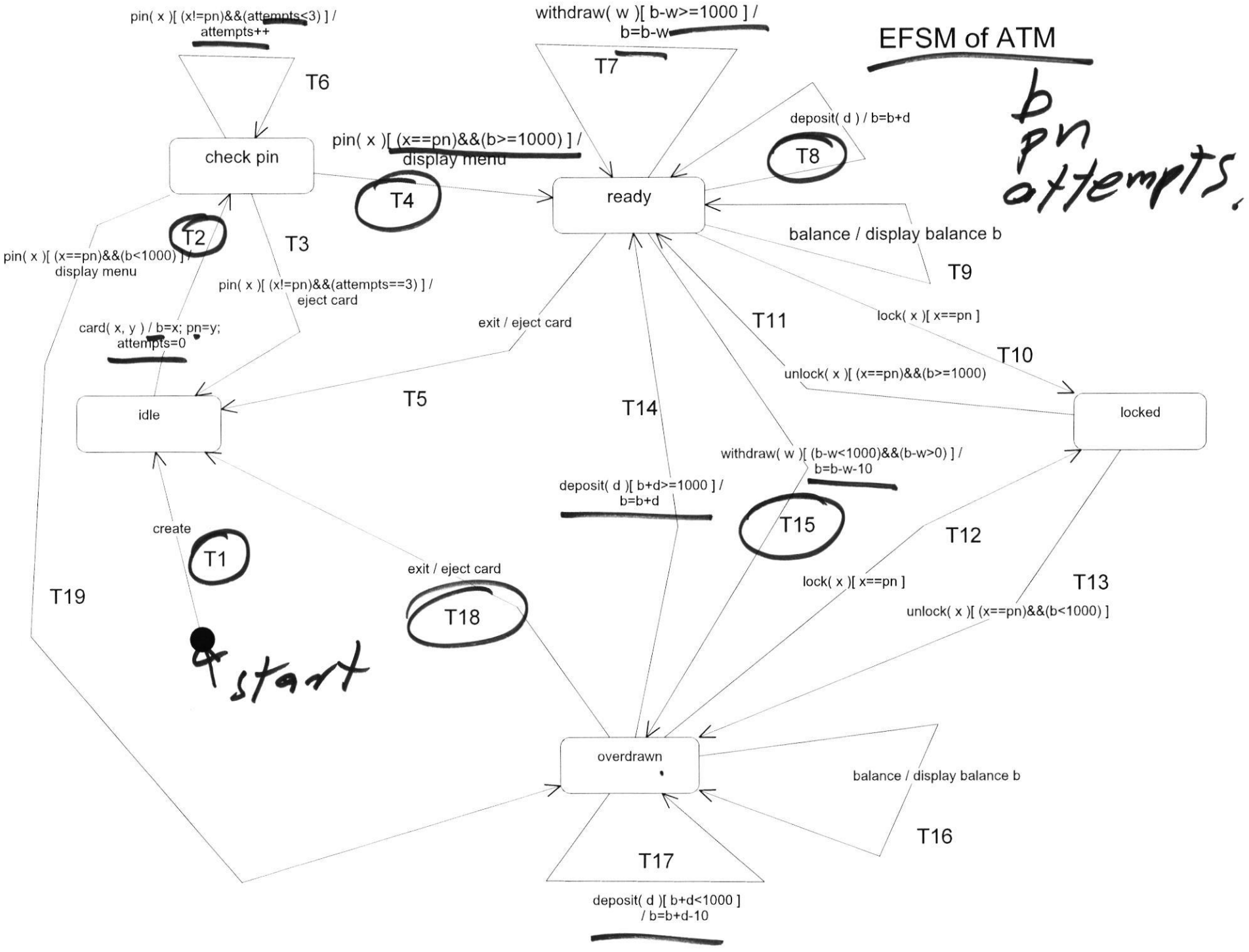
For each solution:

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
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)
T5
T14
idle
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
T18
lock(x)[x==pn]
T13
unlock(x)[(x==pn)&&(b<1000)]
T19
start
overdrawn
balance / display balance b
T16
T17
deposit(d)[b+d<1000]
/ b=b+d-10