Homework #1 is posted

OO Design Patterns

Motivation:

Reuse of design solutions.

- item description pattern
- whole-part —11—
- observer —n—
- state —11—

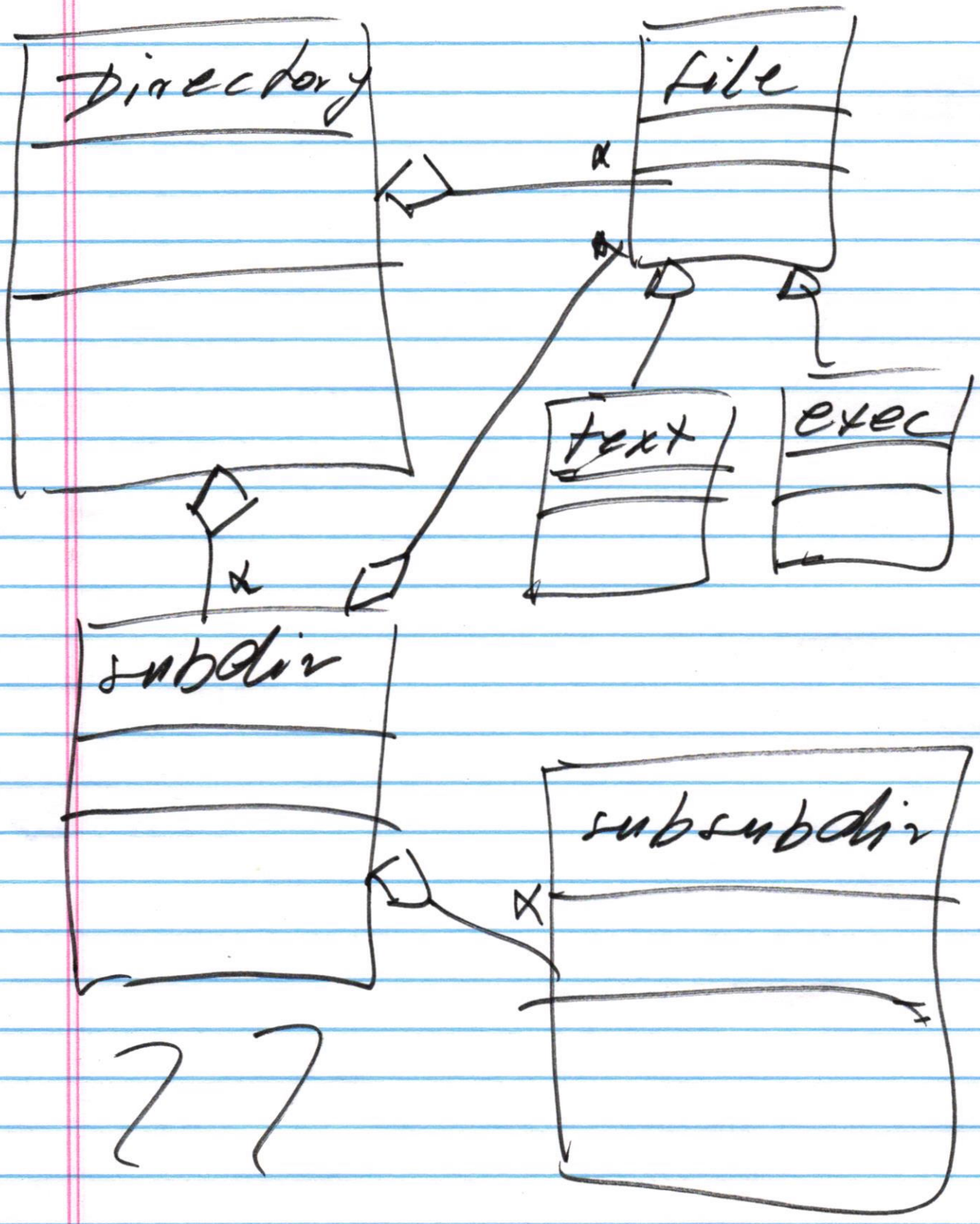
Problem

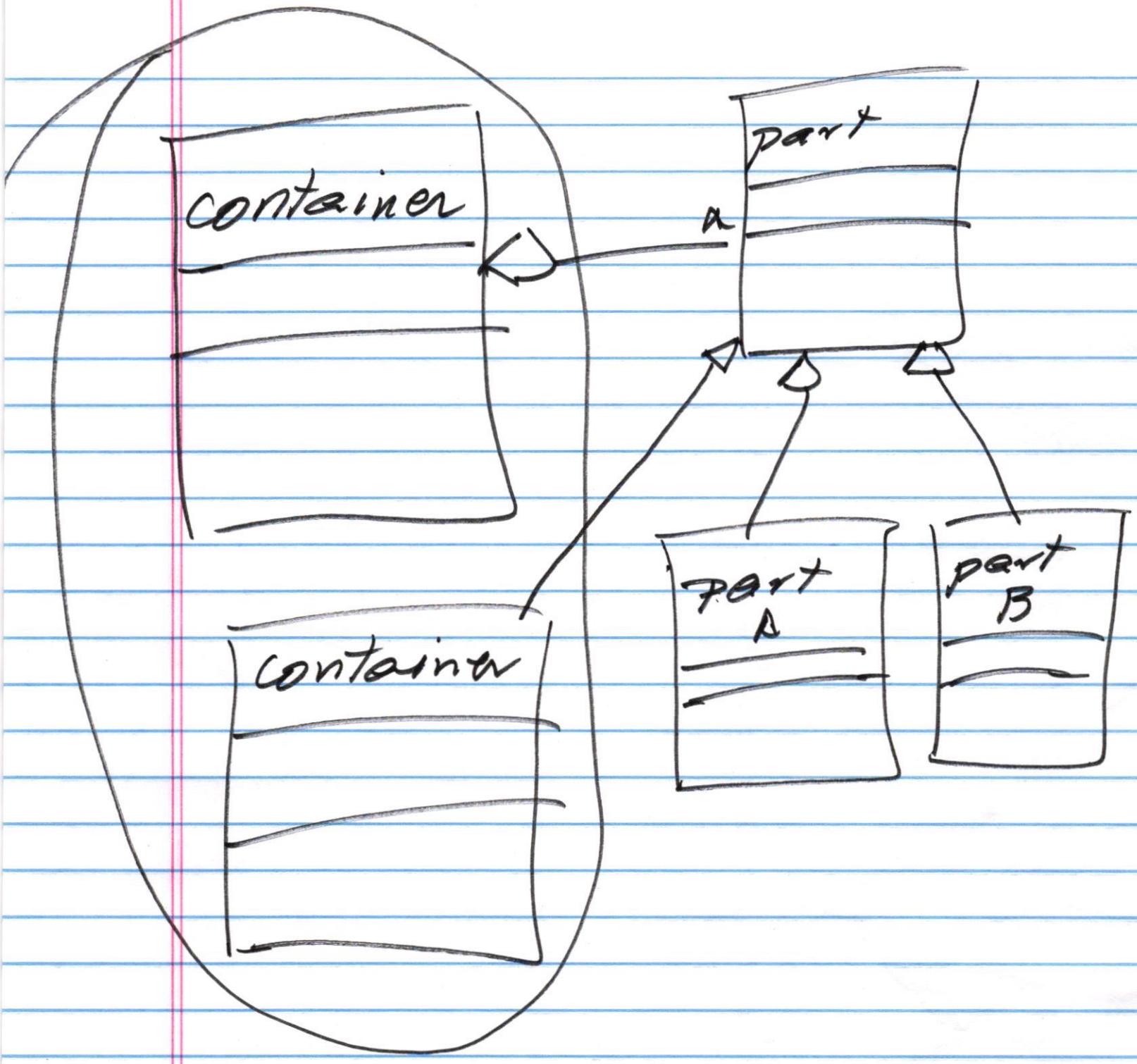
A container consits of many parts of different types with its own ~~part~~ type

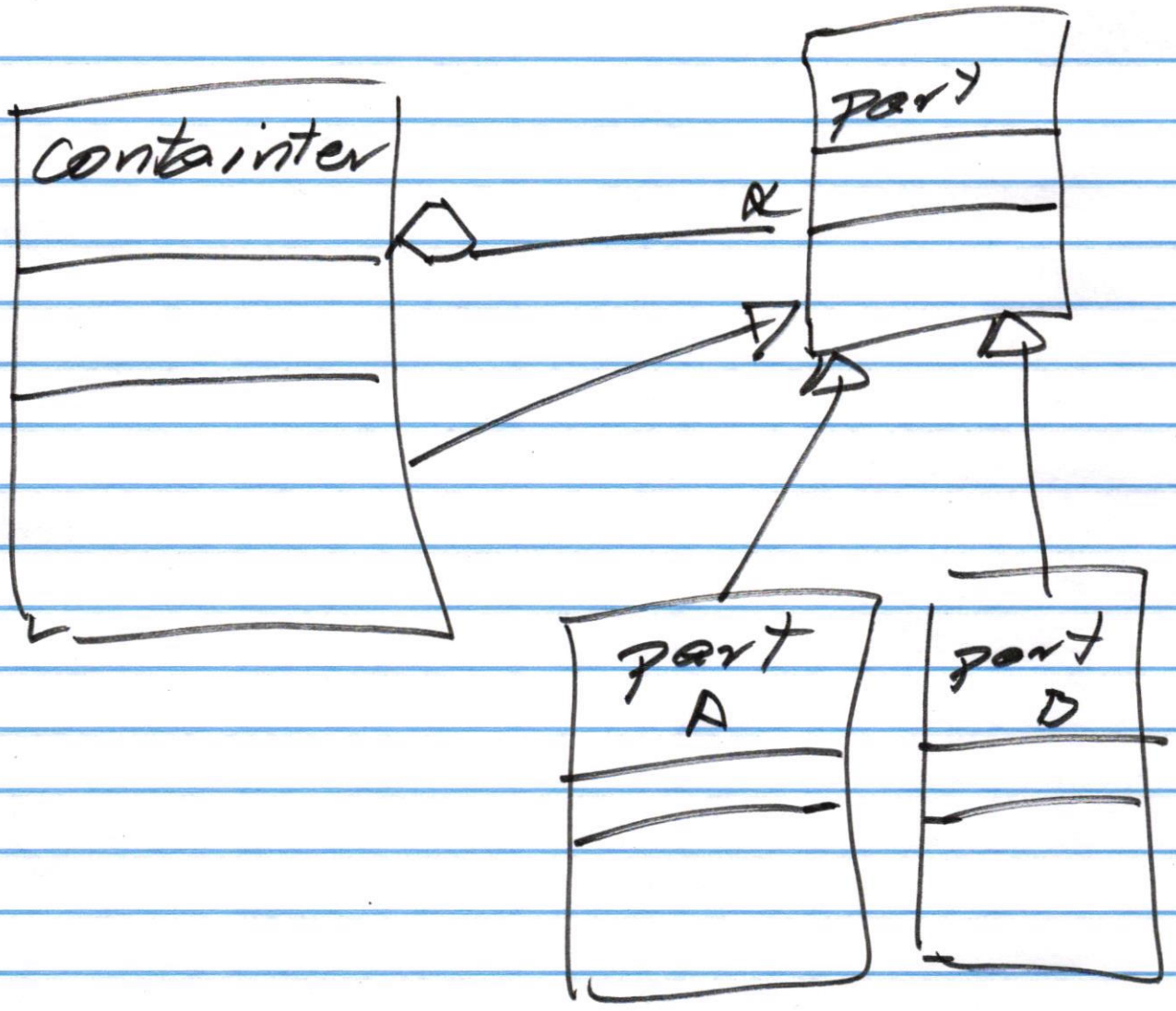
---

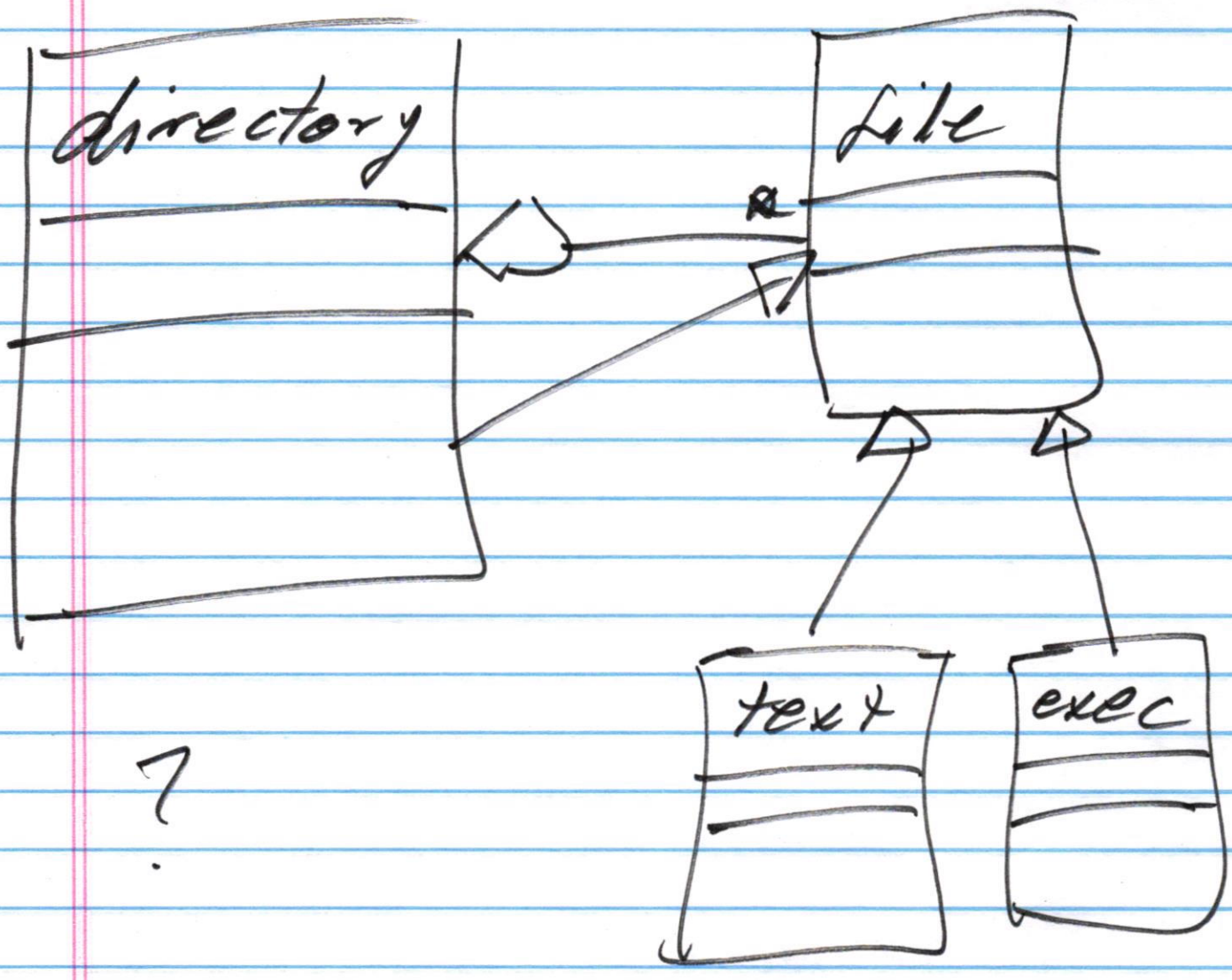
File system

* Directory < files
  directory
* files < text
  exec
  graph

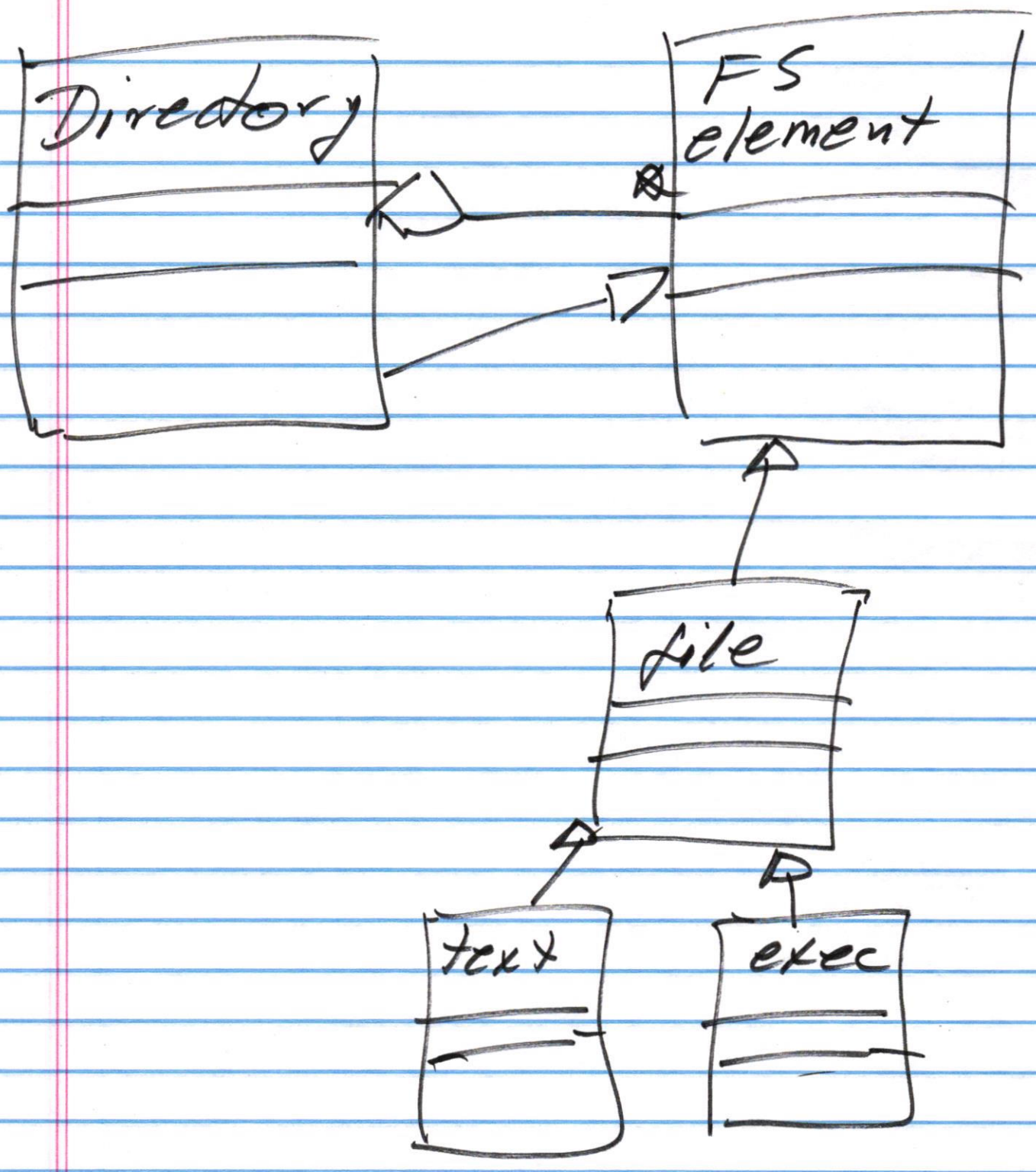

Directory
file
text
exec
subdir
subsubdir


container
container
part
a
part
A
part
B


container
R
Part
Part
A
Part
B


directory
file
text
exec

?


Directory
FS element
File
text
exec

Polymorphism

using the same name to get different services

---

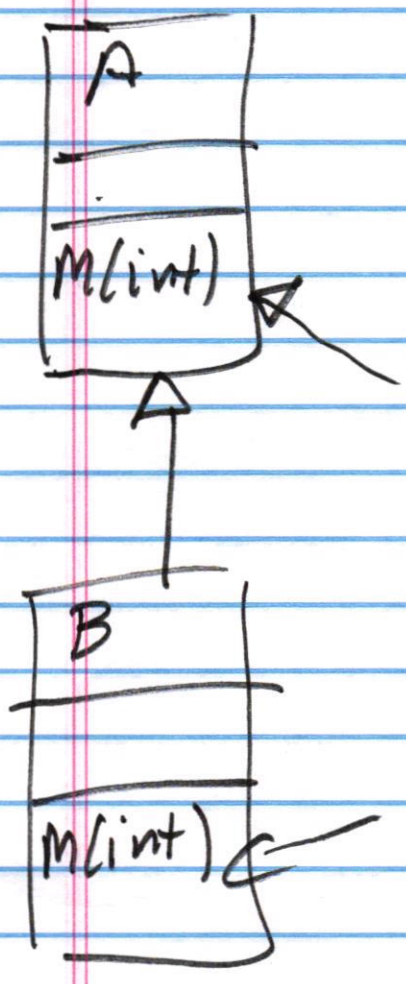
procedural languages.

F(int) → F(x)

F(int, int) → F(x, y)

F(int) → error

polymorphism related to inheritance

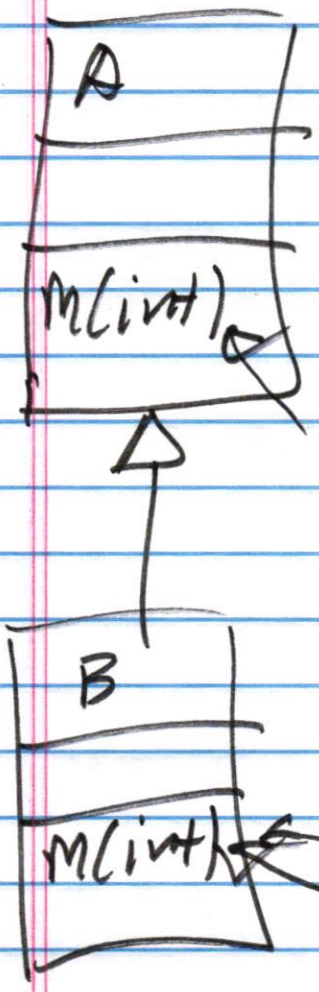


static objects

```
A a;
B b;

a.M(x)
b.M(x)
```

dynamic objects




```
A *pa;
B *pb;

pa = new A
pb = new B
```

(1)

(2)

pa → M(x)

pb → M(x)

pa = pb

pa → M(x)

Observer pattern

publisher - subscriber

Problem:

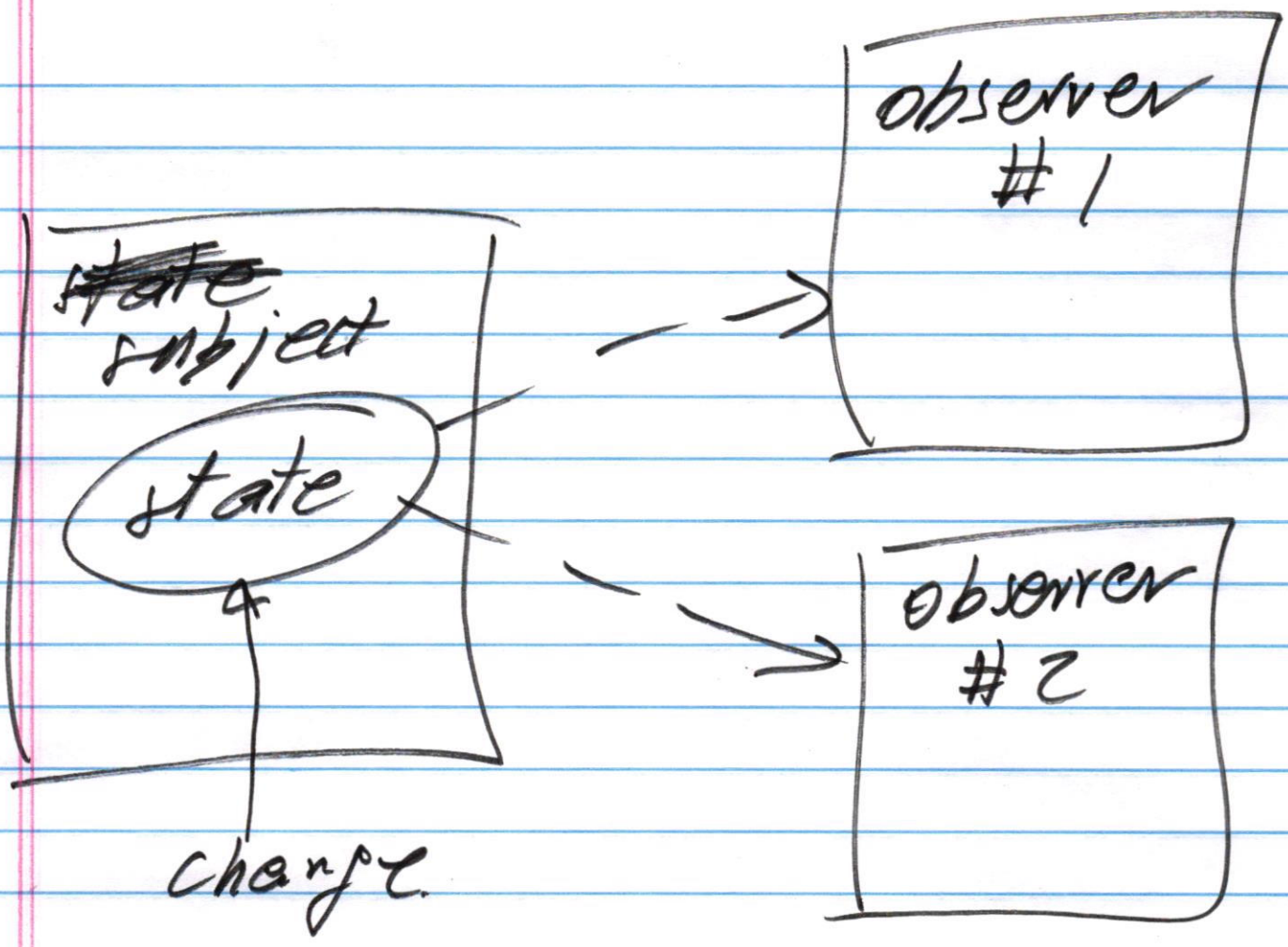
* a change of "state" occurs in a "subject"

* there exist "observers" that are interested in the new state of the subject

Assumptions

* subjects and observers should be de-coupled as much as possible.

* subjects should not have a direct knowledge about observers

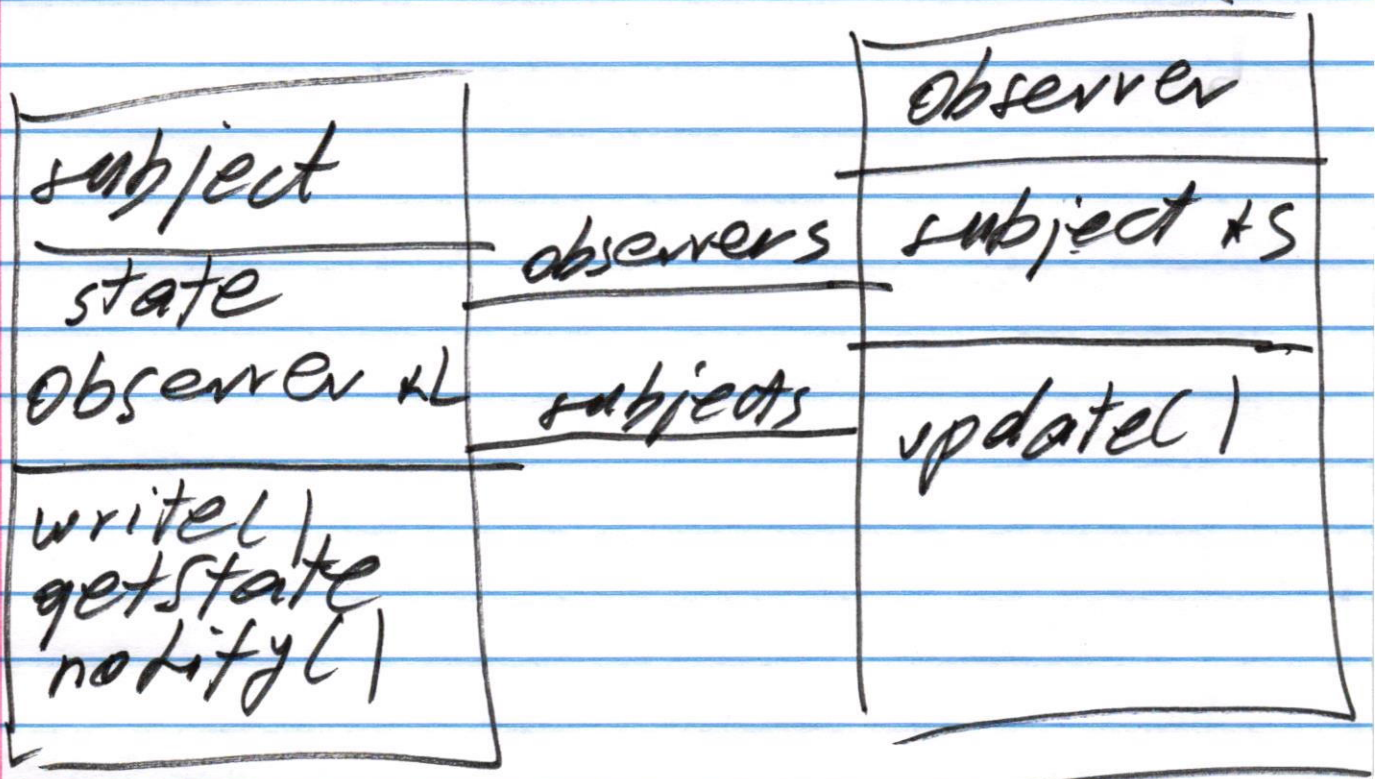


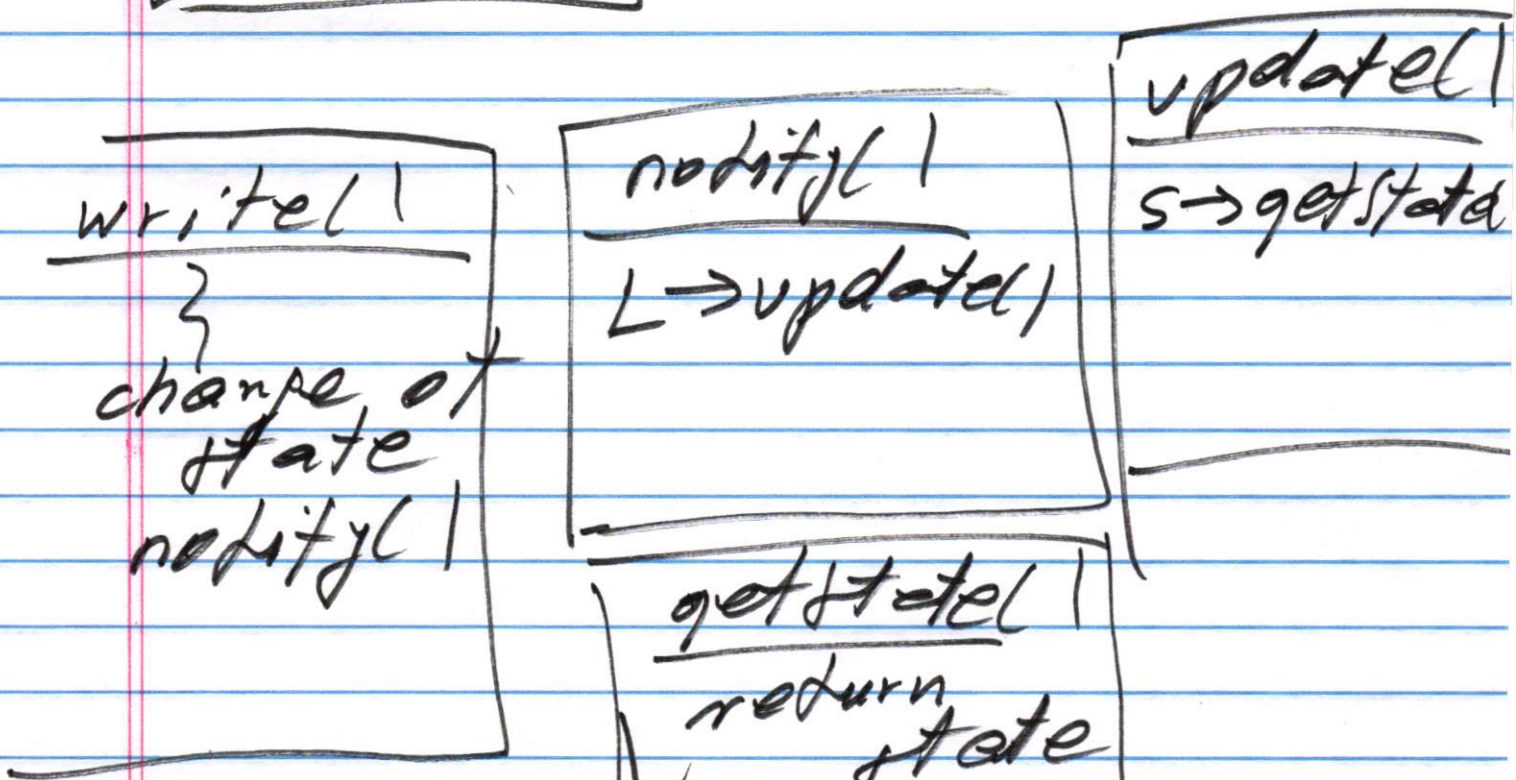
Solution:

Introduce the ~~noti~~ notification mechanism into the subject

the subject can "indirectly" notify interested observers.

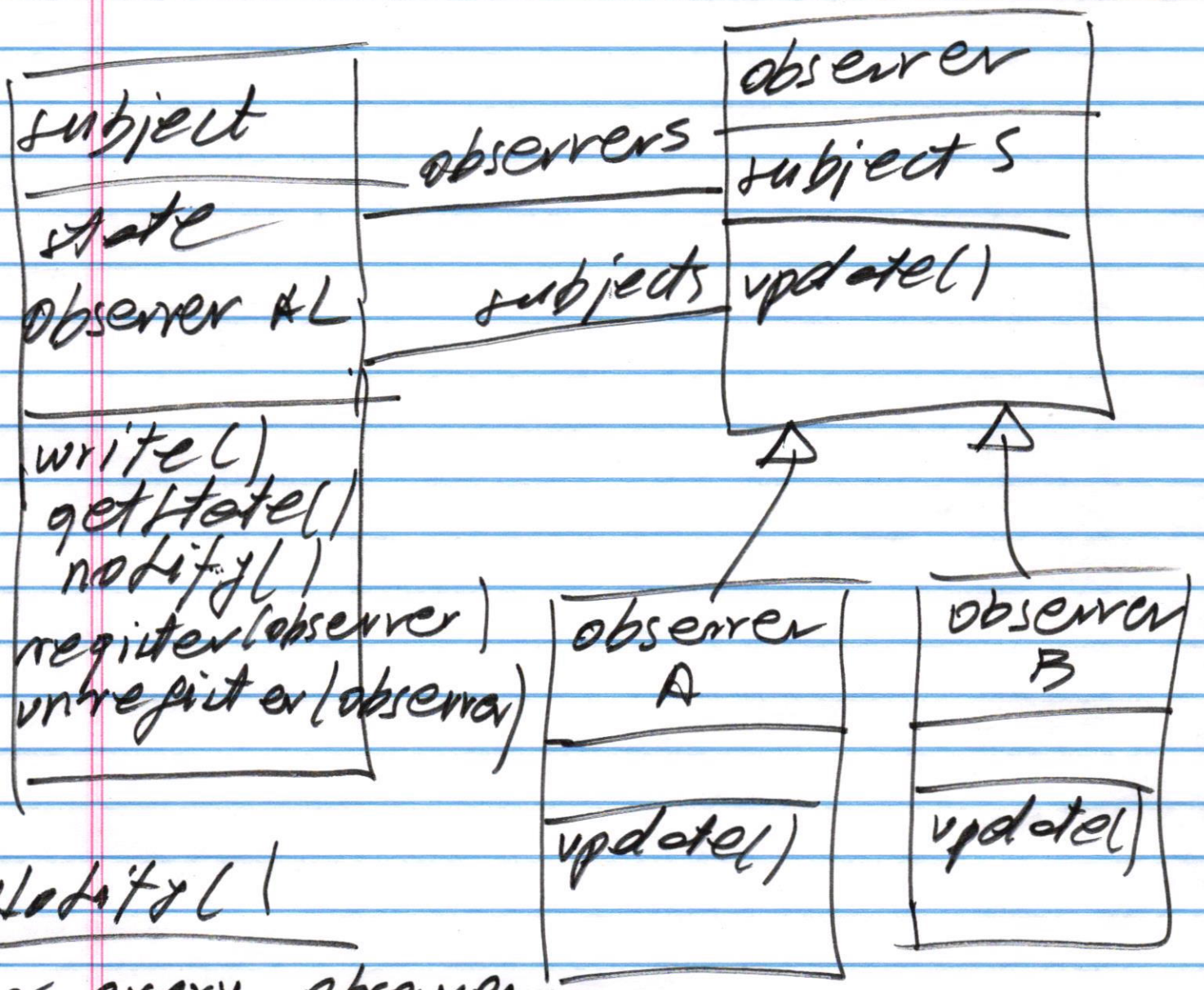
one subject one observer





one subject

many observers of different types



Notify()

for every observer in list L

L → update()

class diagram captures static aspects of the design.

how to show the behavior of the design

sequence diagram

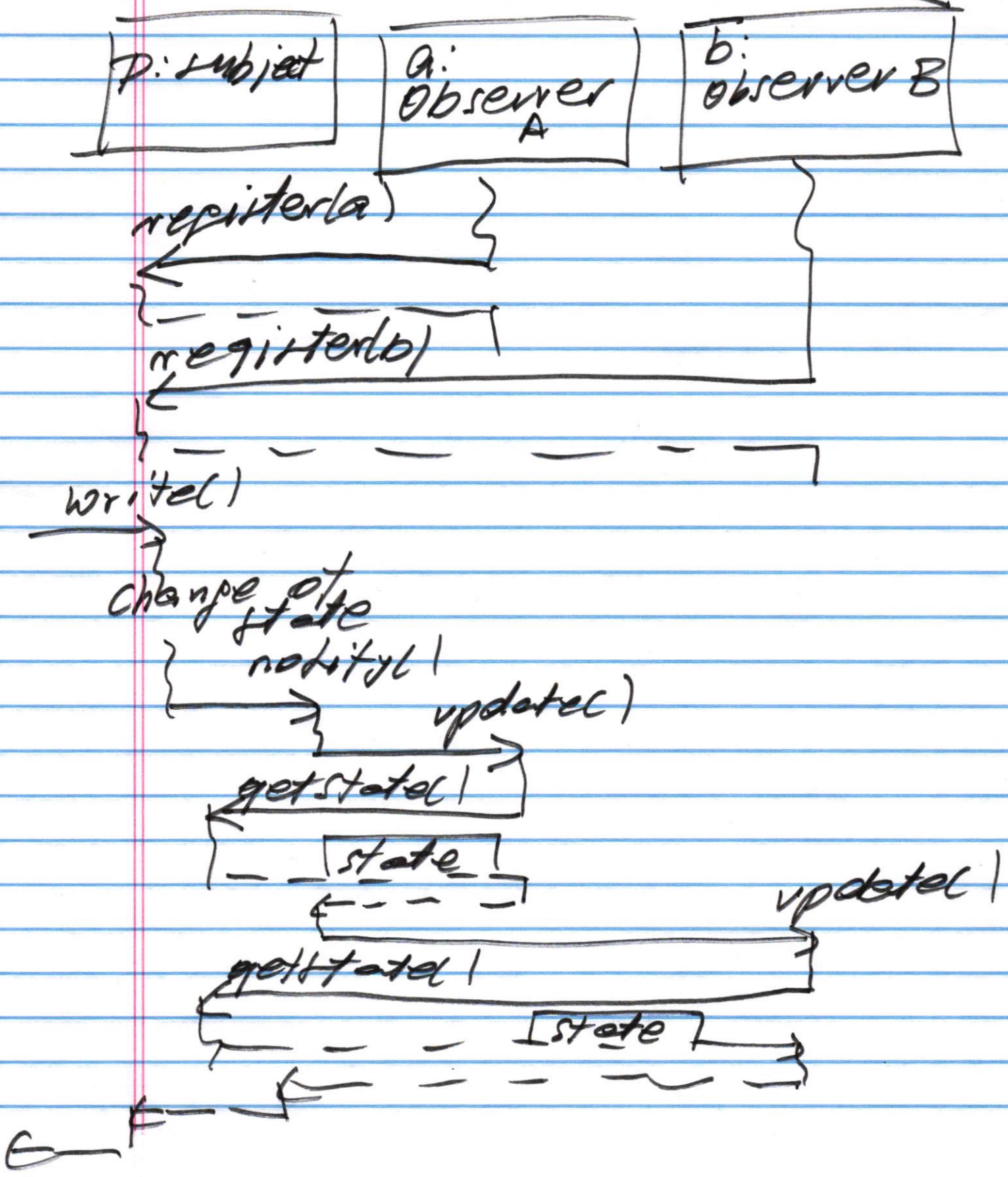
=

a set of objects

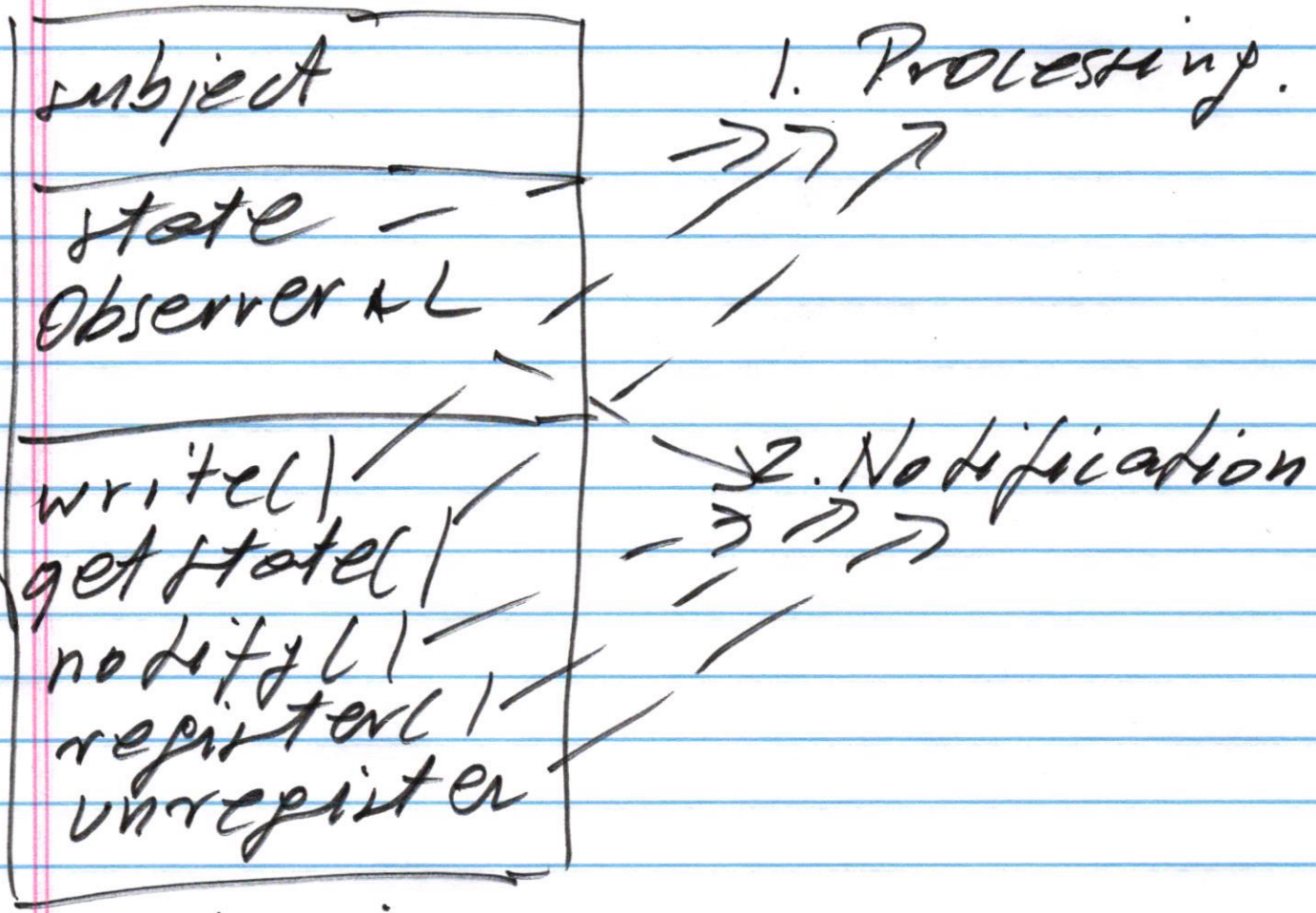
+

call relationships between objects for a given scenario

sequence diagram



Responsibilities of the subject

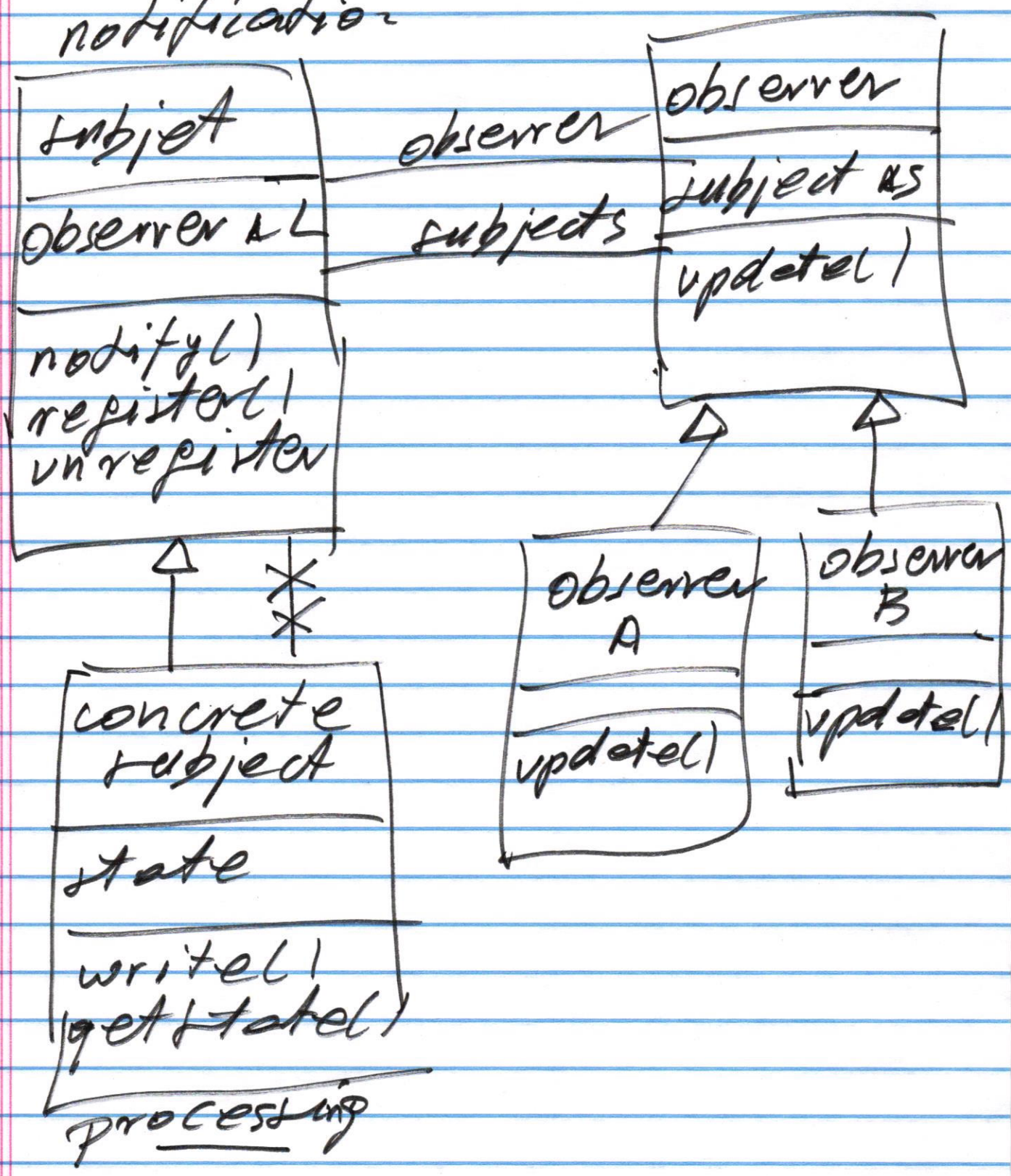


cohesion

1. processing.
2. Notification

} not related responsibilities

bad cohesion!!


notificador
subject
observers
notify()
register()
unregister()
observer
subjects
observer
subject
update()
concrete subject
state
write()
getState()
processing
observer A
update()
observer B
update()

State Pattern

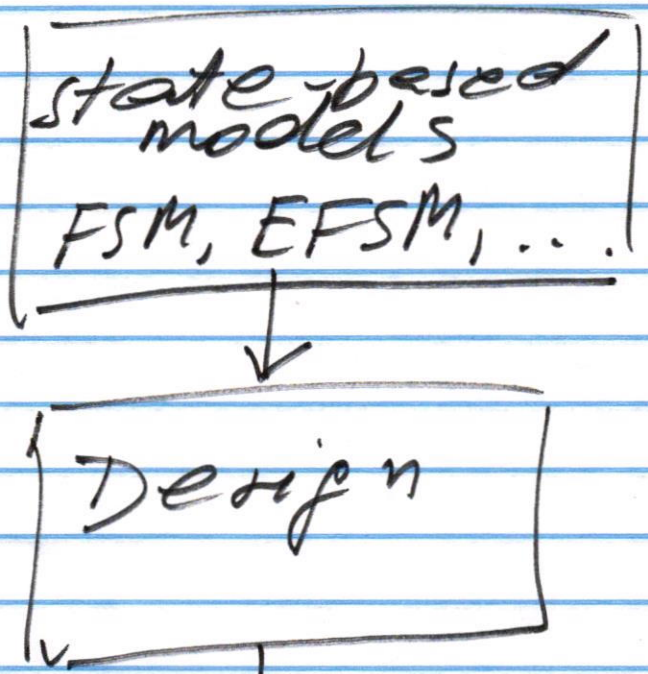
state-based systems

A system that can be characterized by
a set of states
+
transitions between states based on events

---

* control systems
* embedded -"-
* communication -"-
* ...

Design state-based system



State pattern

to design the state based system based on the state model

State-based languages

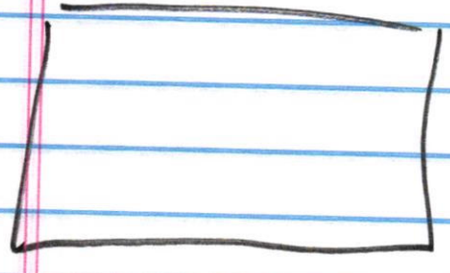
- FSM: Finite State Machine
- EFSM: Extended FSM
- VFSM: Virtual FSM
- ~
- .

Finite State Machine

a set of states

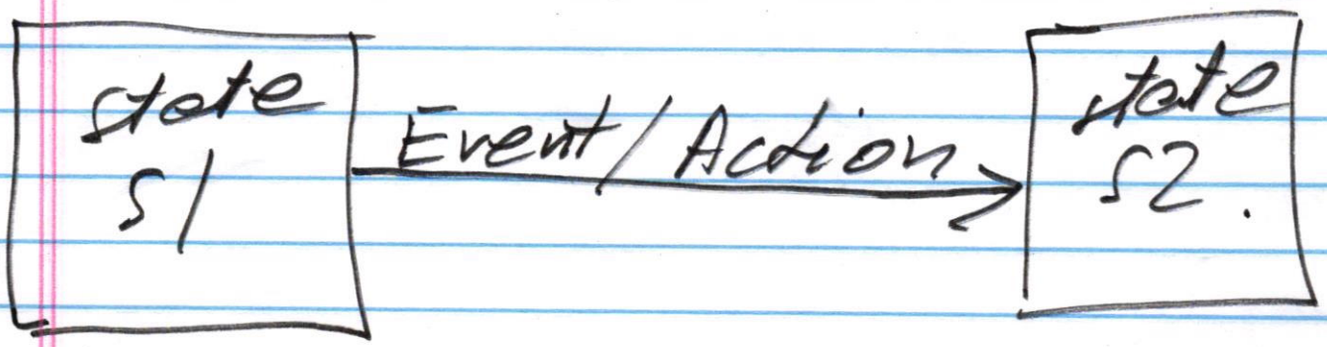
+

~~the~~ transitions between states



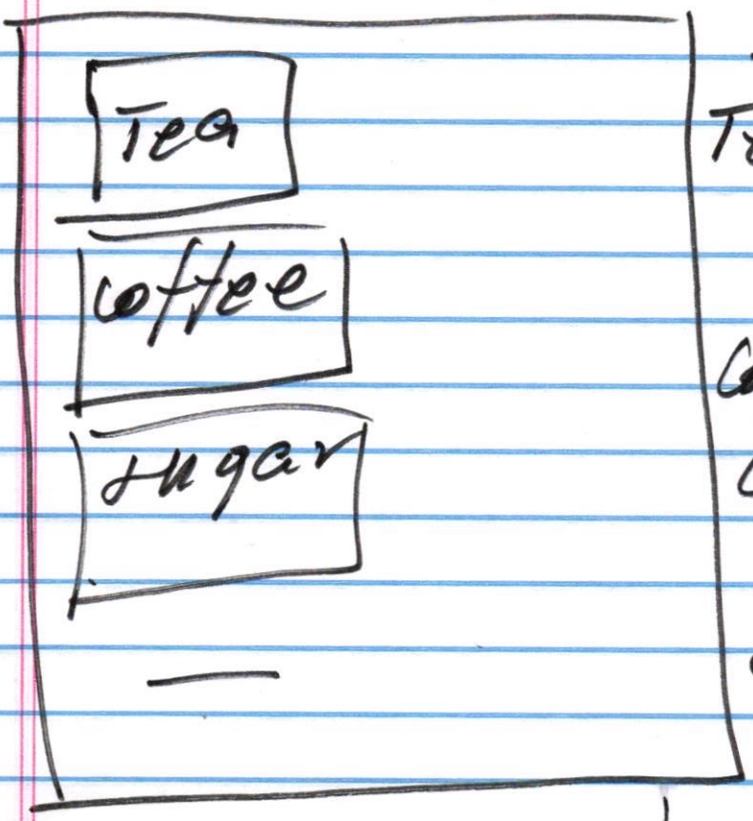
state

transition



1. system is in state S1
2. "Event" is invoked
3. transition from S1 to S2
4. "Action" is performed.

simple vending Machine



Events

Tea: tea button is pressed

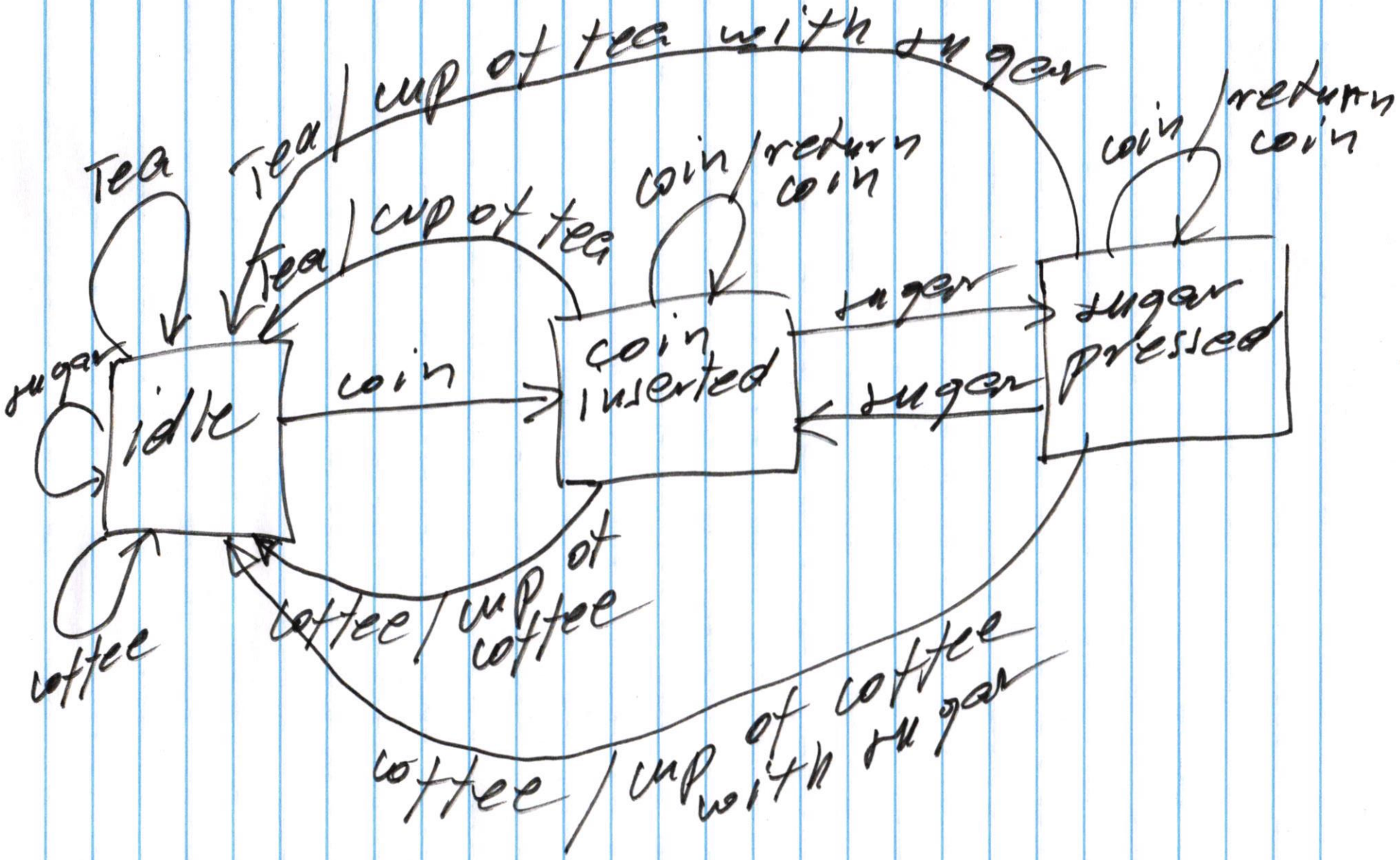
coffee: coffee button is pressed

sugar: sugar button is pressed

coin: coin is inserted

Actions:

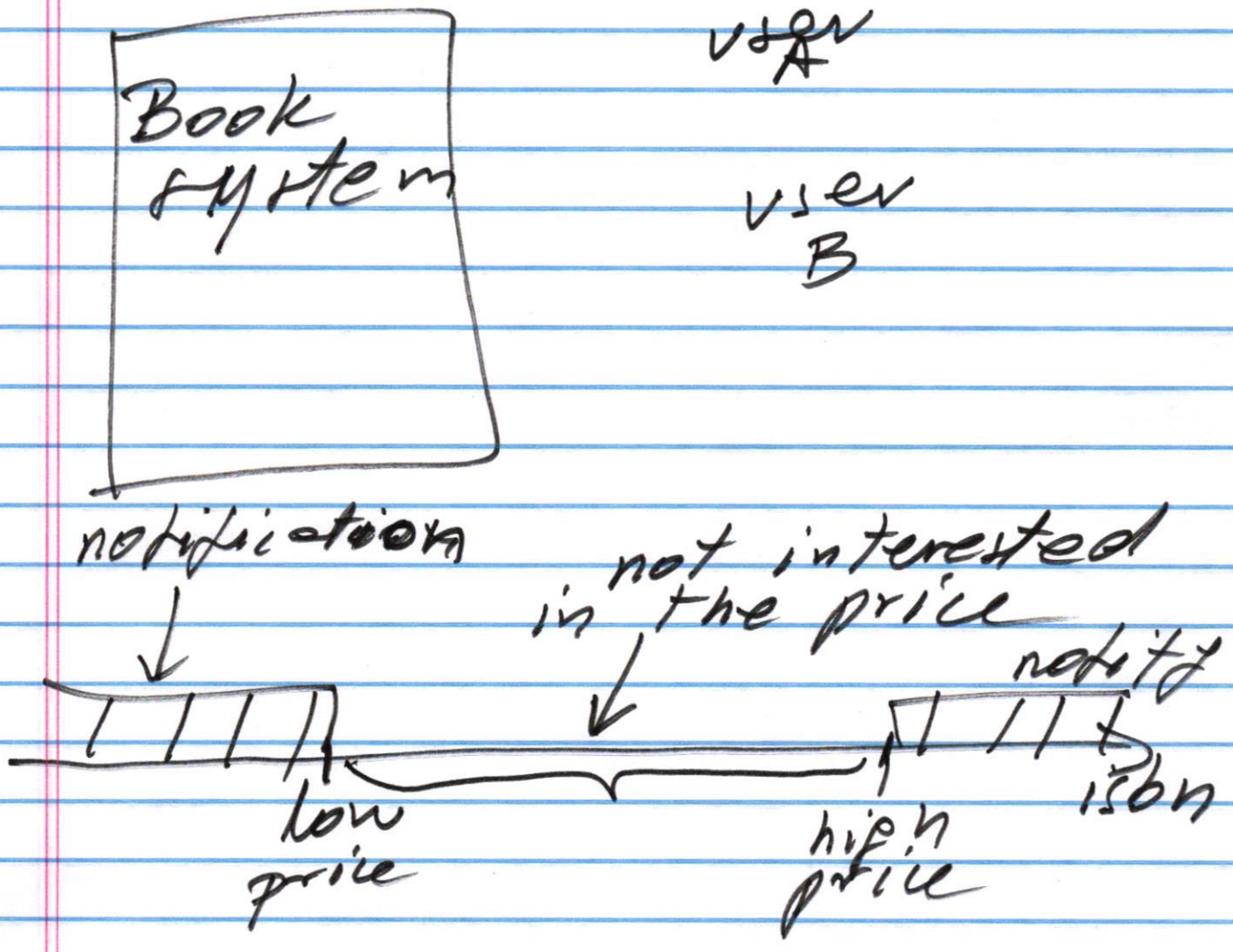
* cup of tea with/without sugar
* cup of coffee — " — " —
* return coin


tea / cup of tea with sugar
tea
tea / cup of tea
coin / return coin
coin / return coin
sugar
idle
coin
coin inserted
sugar
sugar
sugar pressed
coffee
coffee / cup of coffee
coffee / cup of coffee with sugar

Homework #1

Problem #1

Observer pattern



HOMEWORK ASSIGNMENT #1

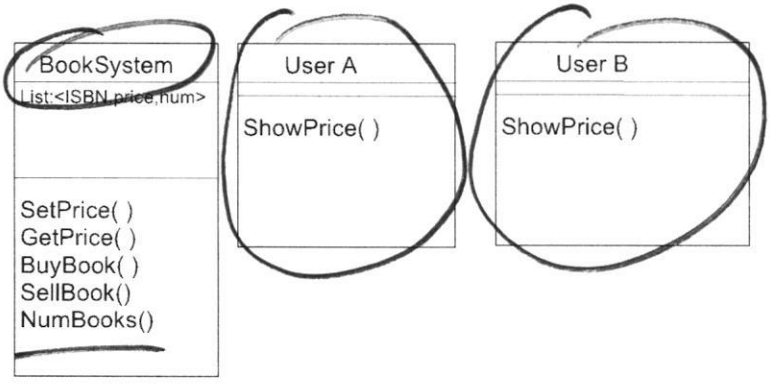
CS 586; Fall 2025
Due Date: **September 18, 2025**
Late homework: 50% off
After **September 23,** the homework assignment will not be accepted.

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**.

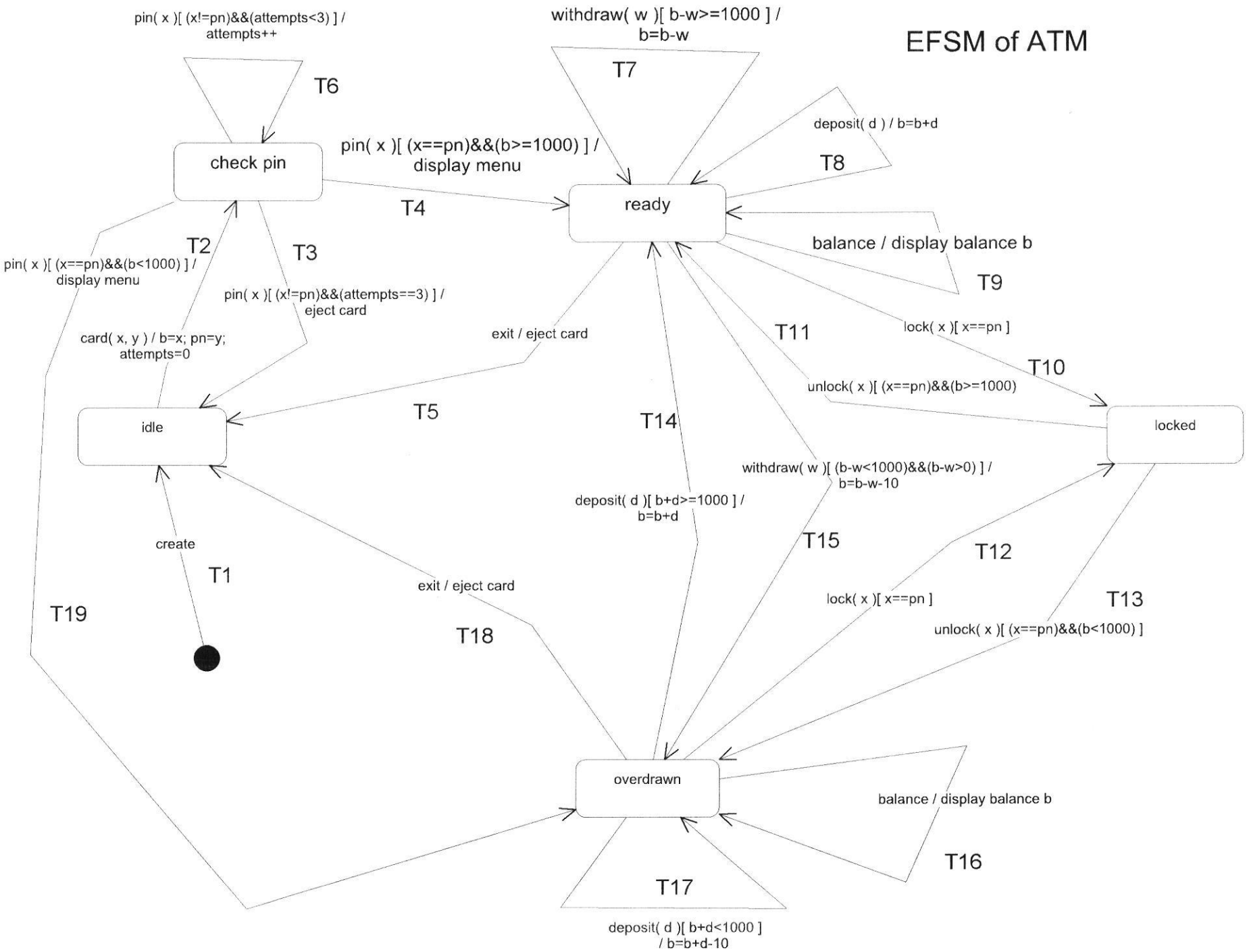
For each solution:

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
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
overdrawn
balance / display balance b
T16
T17
deposit(d)[b+d<1000] / b=b+d-10