Homework #1 is posted

OO design patterns

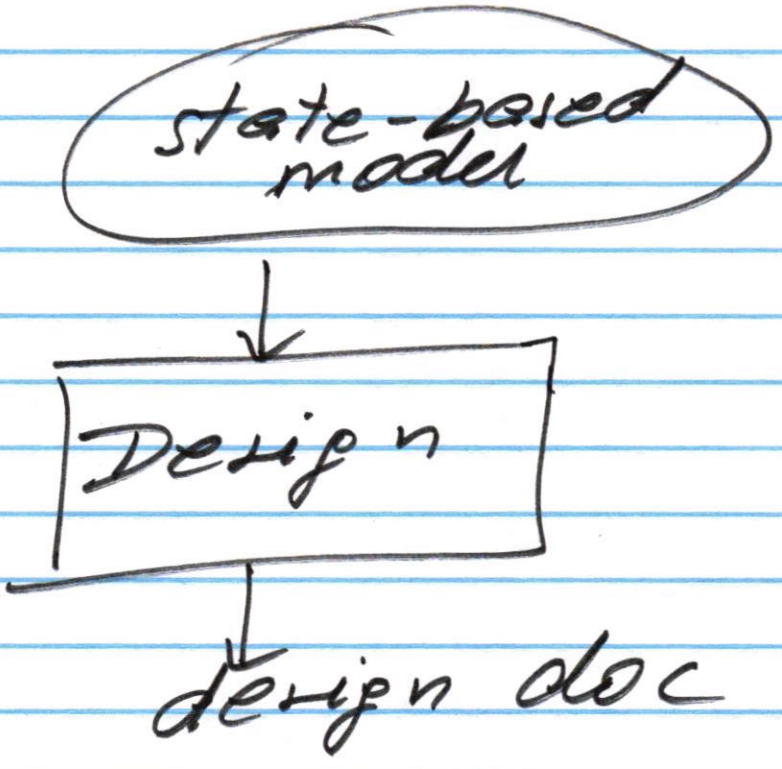
- item description pattern
- whole-part —"—
- observer —"—
- state —"—

State-based systems

a set of states

+

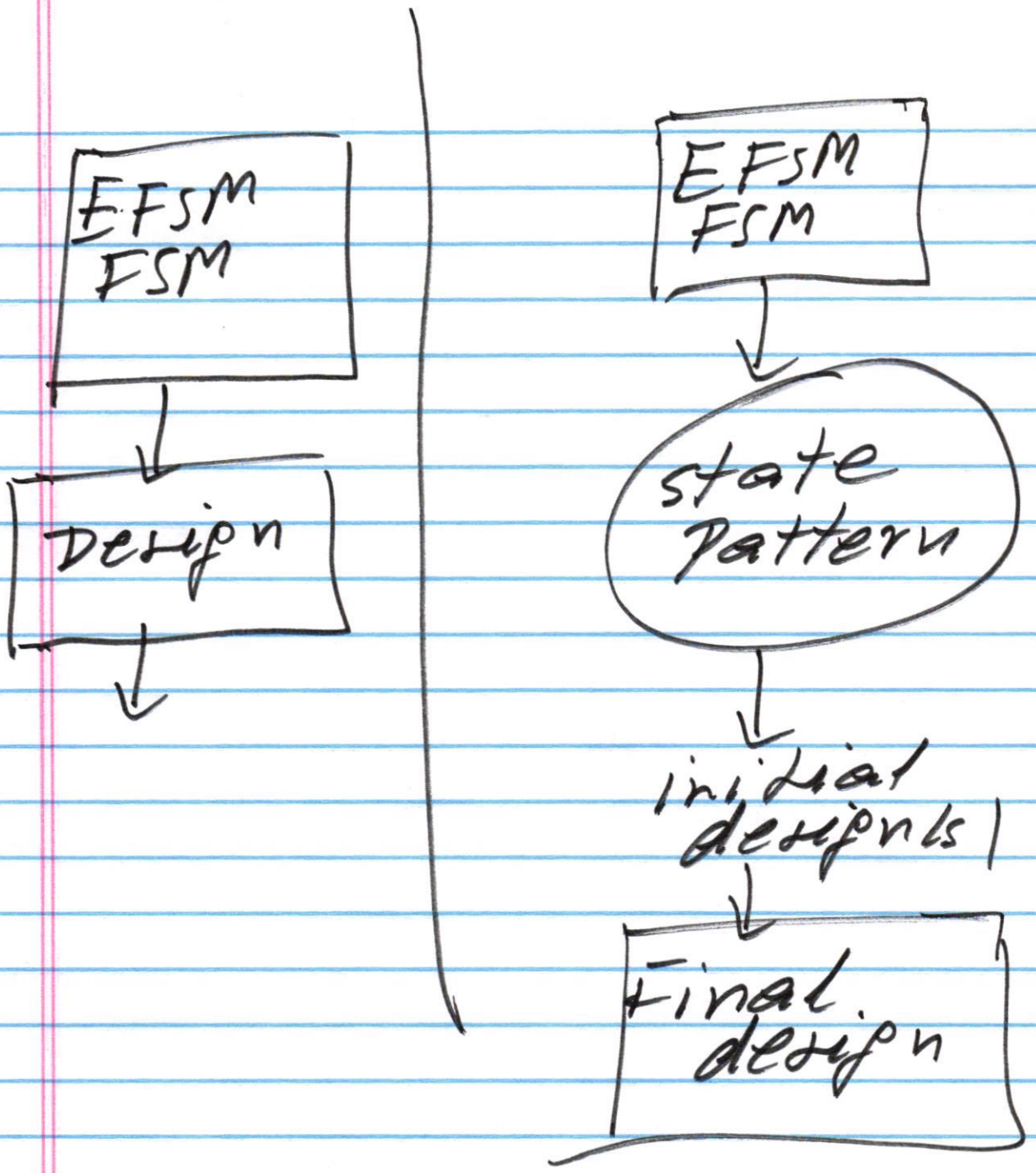
transitions between states based on events



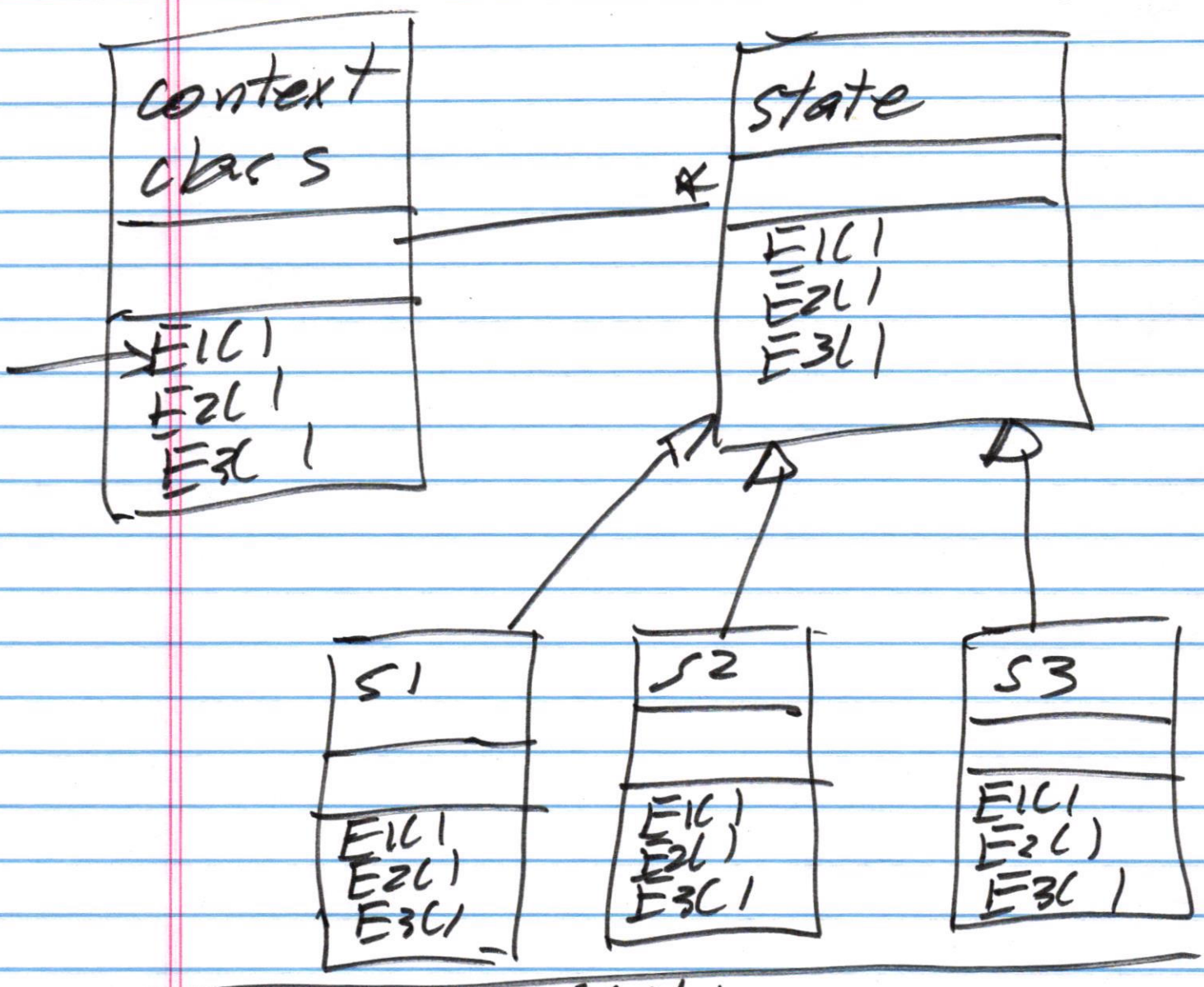
state-based modeling.

FSM: Finite state Machine

EFSM: Extended FSM


EFSM
FSM
Design
EFSM
FSM
state
Pattern
initial
design(s)
final
design

state Pattern



Responsibilities:

(1) performing actions

(2) changing states

(1) state classes are responsible for performing actions

(2) changing states.

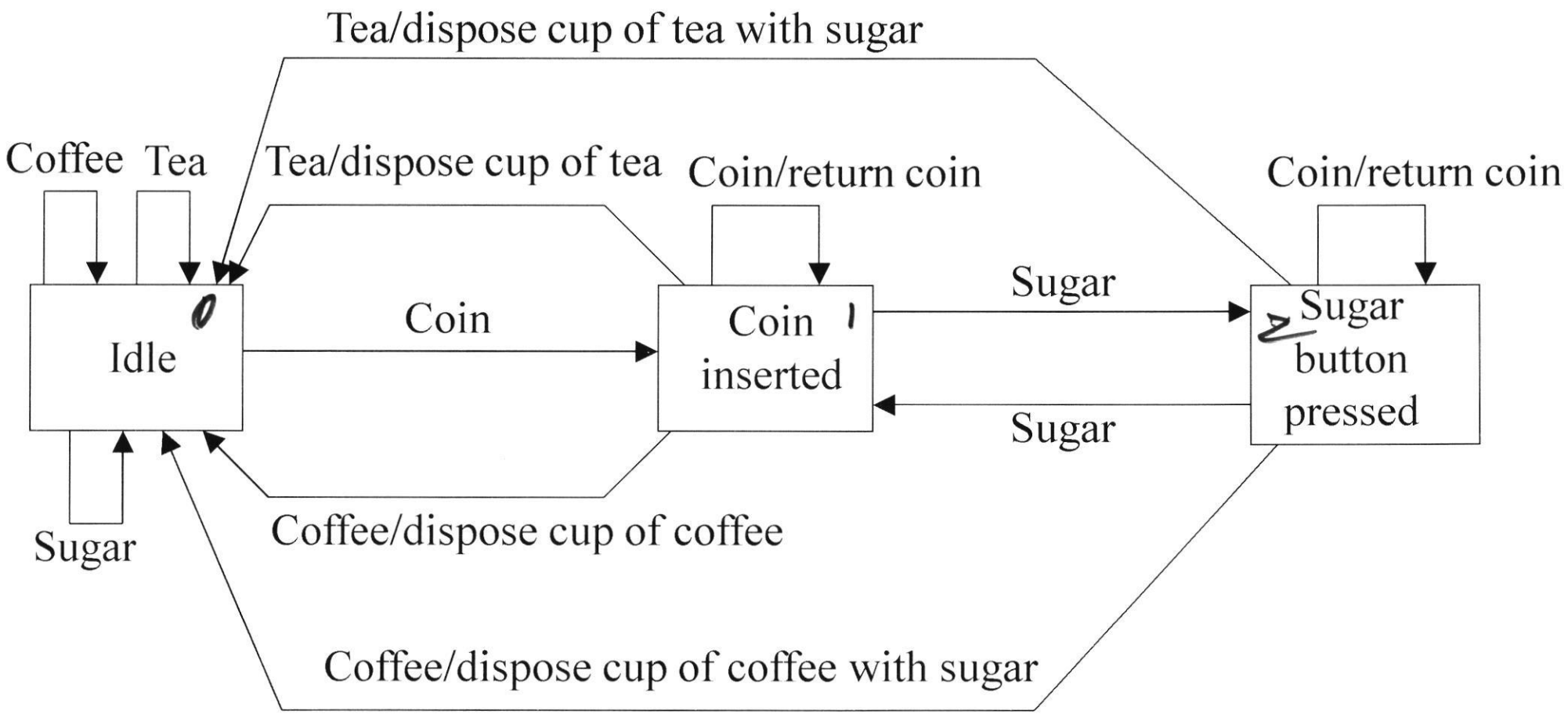
(a) centralized solution

context class changes states

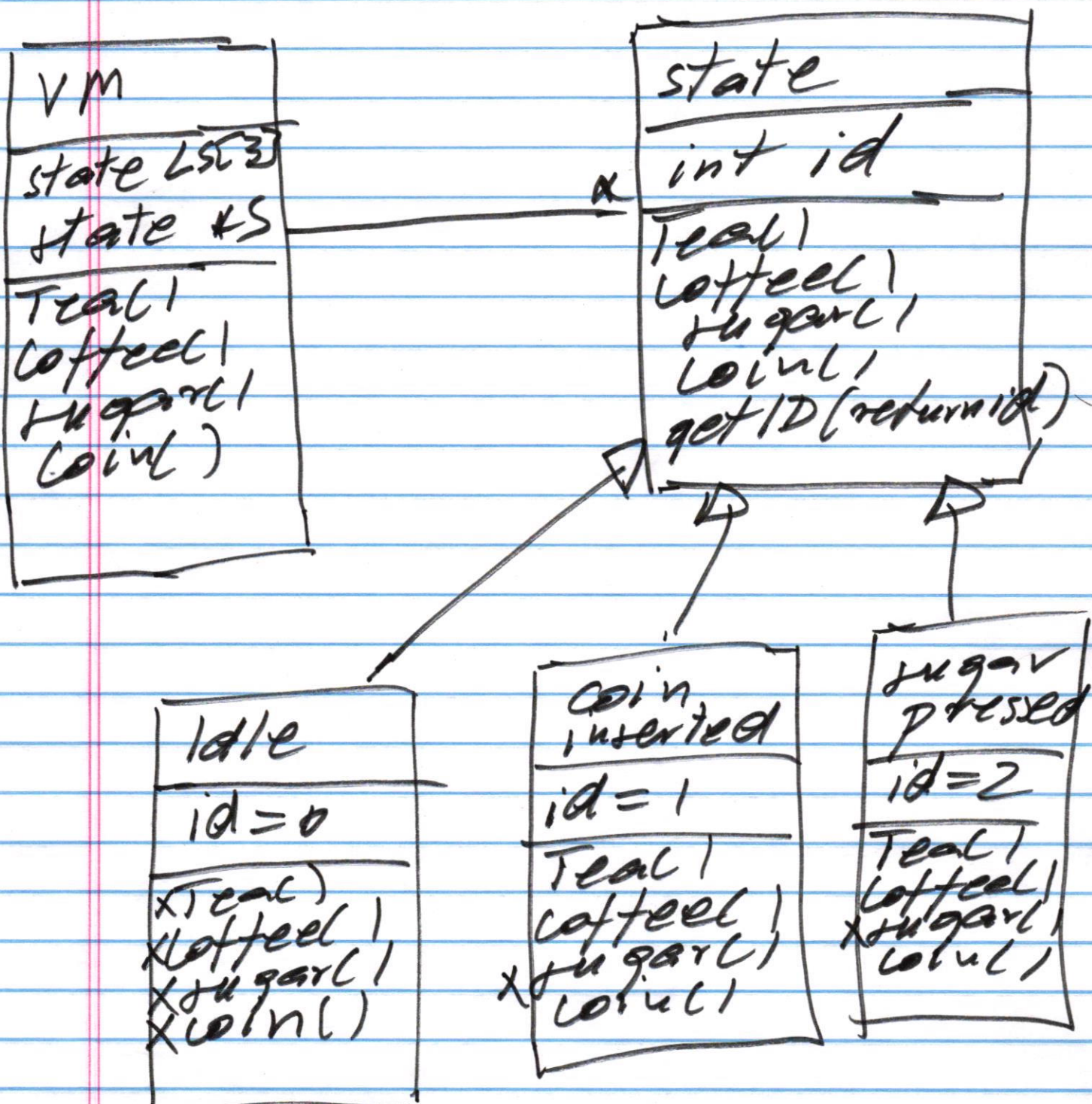
(b) de-centralized solution

state classes change states.

Vending Machine

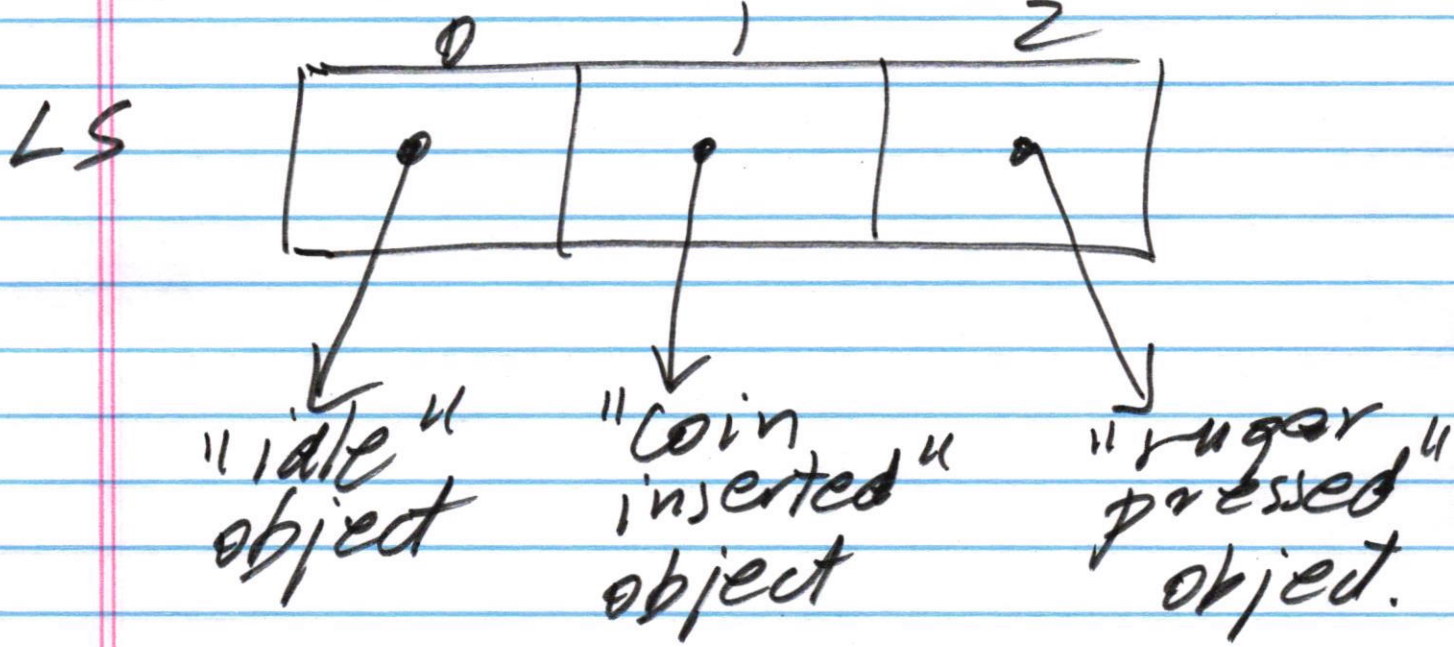


centralized version



VM class

list of states



S points to the current state object.

Initially

S = LS[0]

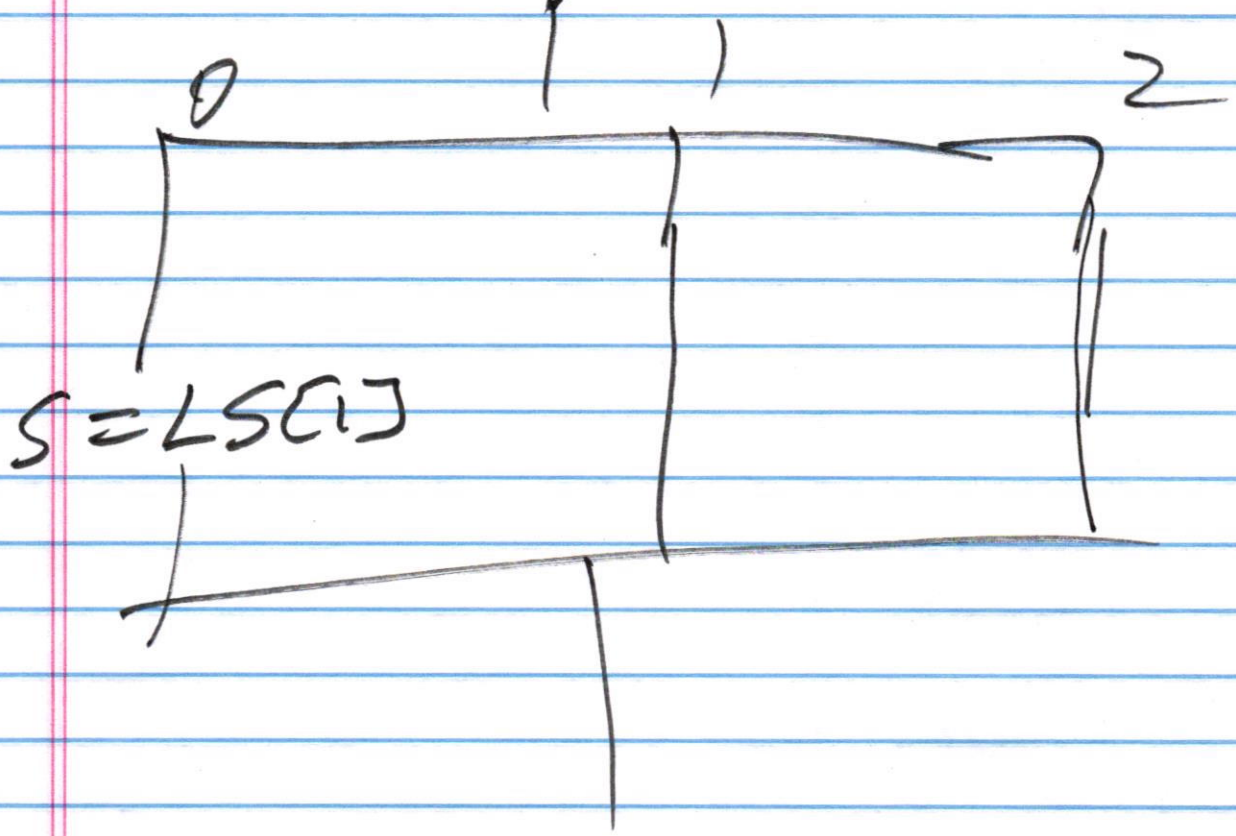
VM class

operation coin()

1. S → coin() //action

2. change of state.

S → getID()

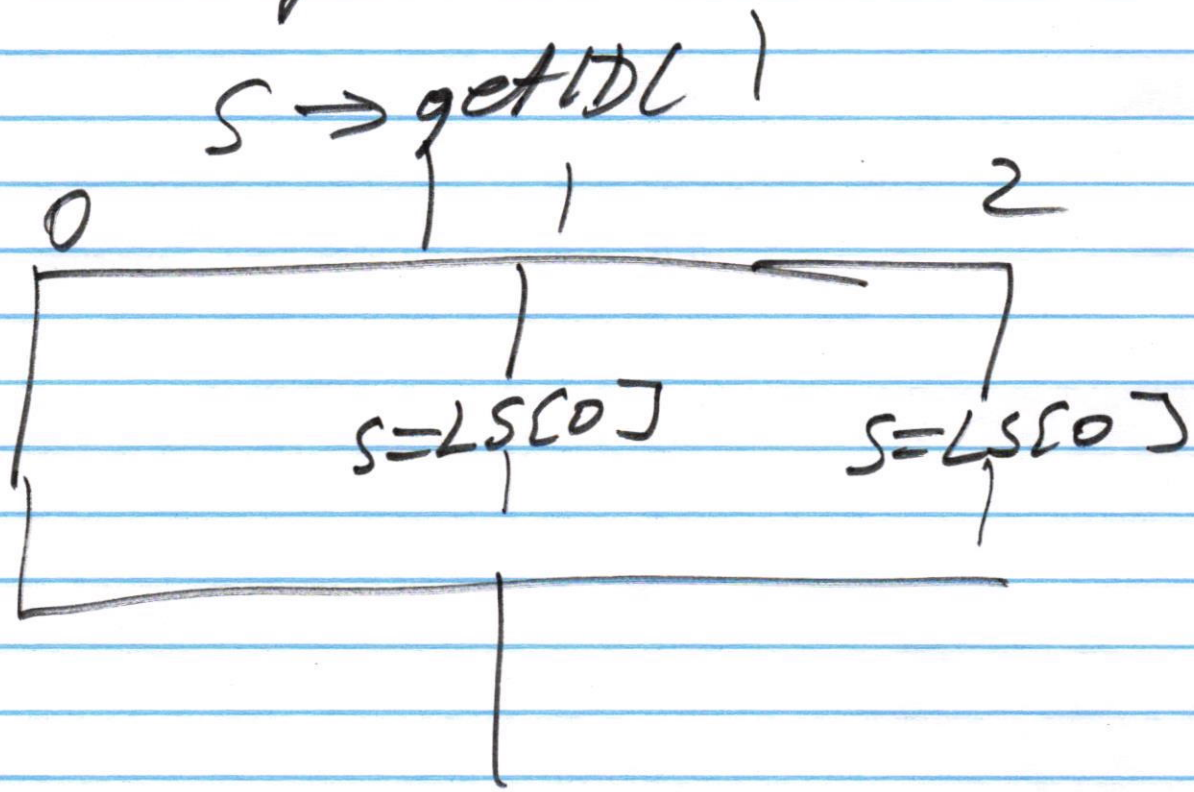


class VM

Tea() operation

1. S → Tea() // action

2. change state.

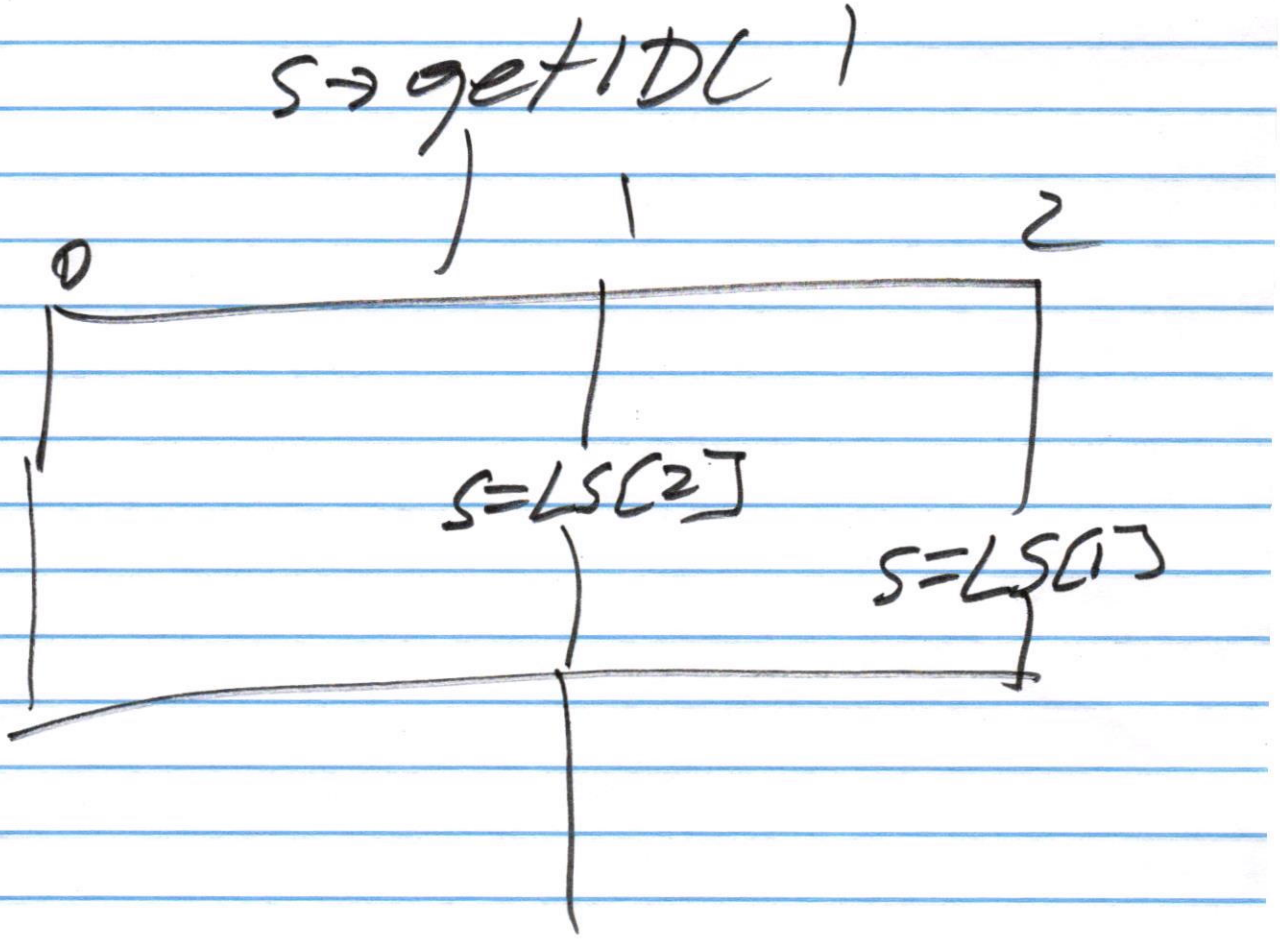


VM class

sugar() operation

1. S → sugar() // action

2. change state



"coin inserted" class

Tea(): dispose cup of tea

Coffee(): dispose cup of coffee

Coin(): return coin

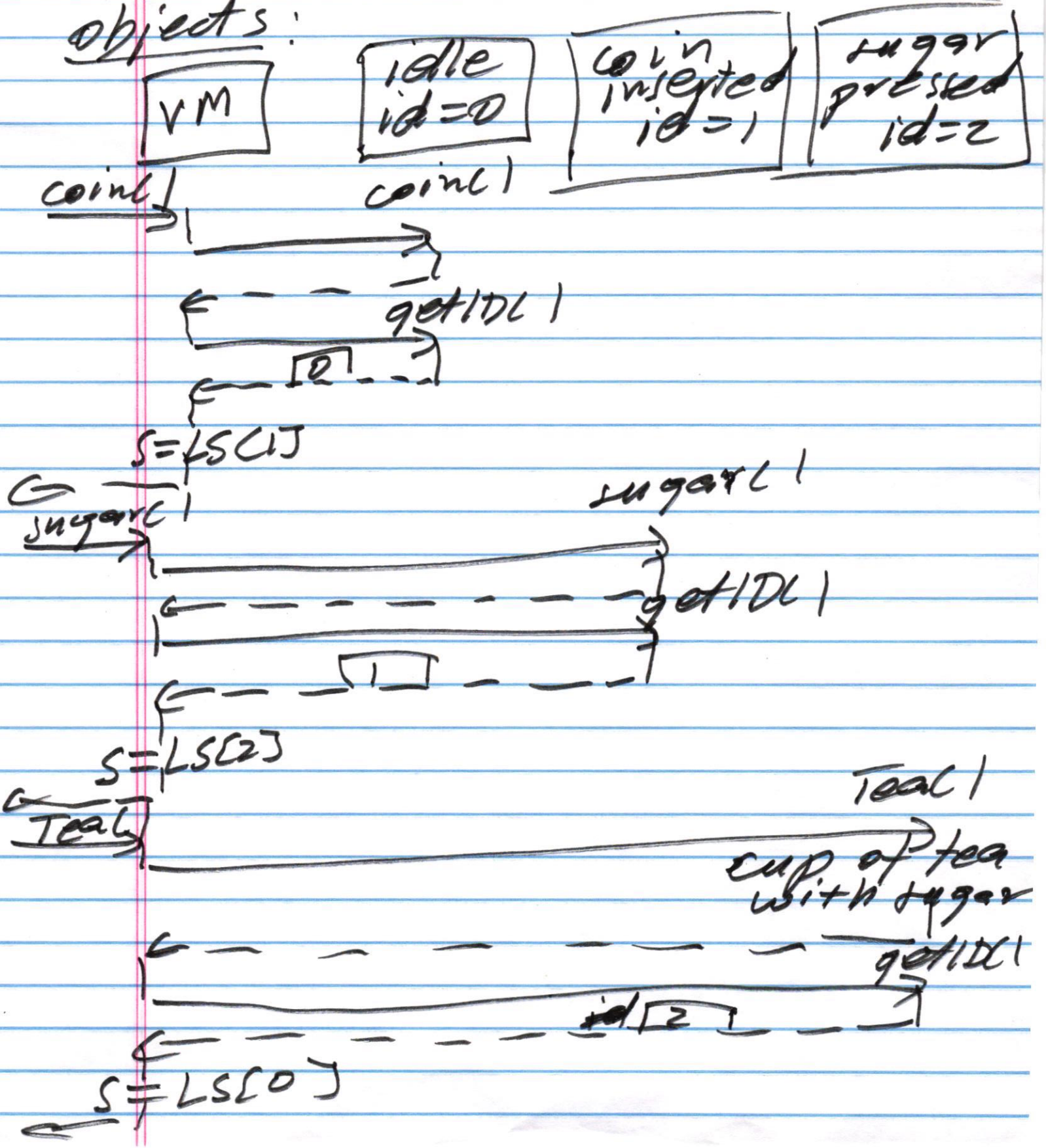
"sugar pressed" class

Tea(): dispose cup of tea with sugar

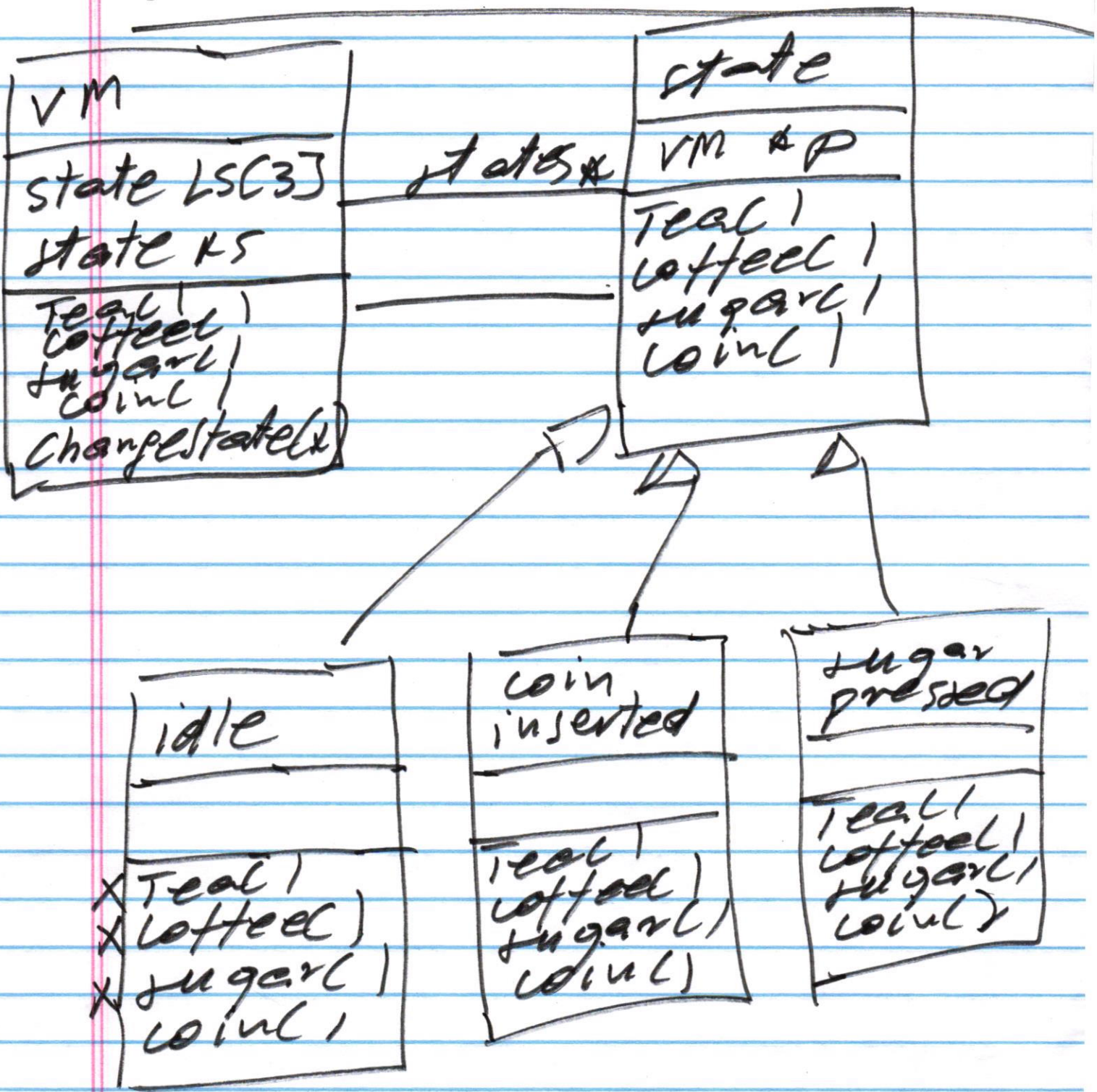
coffee(): dispose cup of coffee with sugar

Coin(): return coin

sequence diagram



de-centralized solution



VM class

coin() operation

s -> coin()

Tea() operation

s -> Tea()

---

coffee() operation

s -> coffee()

---

sugar() operation

s -> sugar()

---

changeState(int x)

s = LS[x]

"idle" class

coin()

1. no action
2. change state

```
p->changeState(1)
```

"coin inserted" class

coin()

1. return coin
2. no change of state

Tea()

1. dispose cup of tea
2. change state.

   p → changeState(0)

---

coffee()

1. dispose cup of coffee
2. change state.

   p → changeState(0)

---

sugar()

1. no action
2. p → changeState(2)

"sugar" pressed" class

Tea()

1. dispose cup of tea with sugar
2. p → changeState(0)

coffee()

1. dispose cup of coffee with sugar.
2. p → changeState(0)

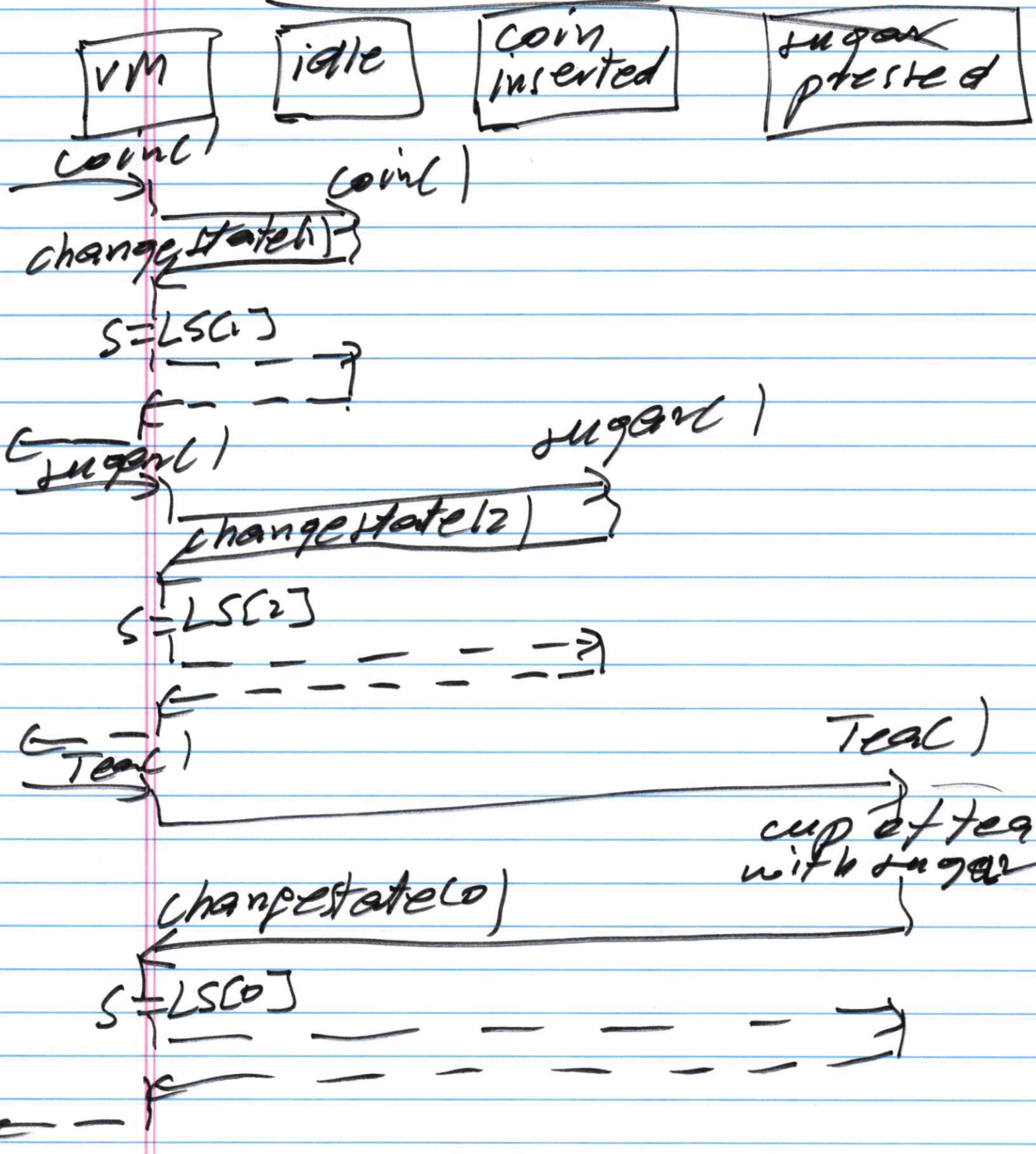
sugar()

1. no action
2. p → changeState(1)

coin()

1. return coin
2. no state change.

sequence diagram



centralized version

1. state classes perform actions.

2. context class changes states

3. state classes are de-coupled

4. context class and state classes. are strongly coupled

Decentralized solutions

1. state classes
   (a) perform actions
   (b) change states

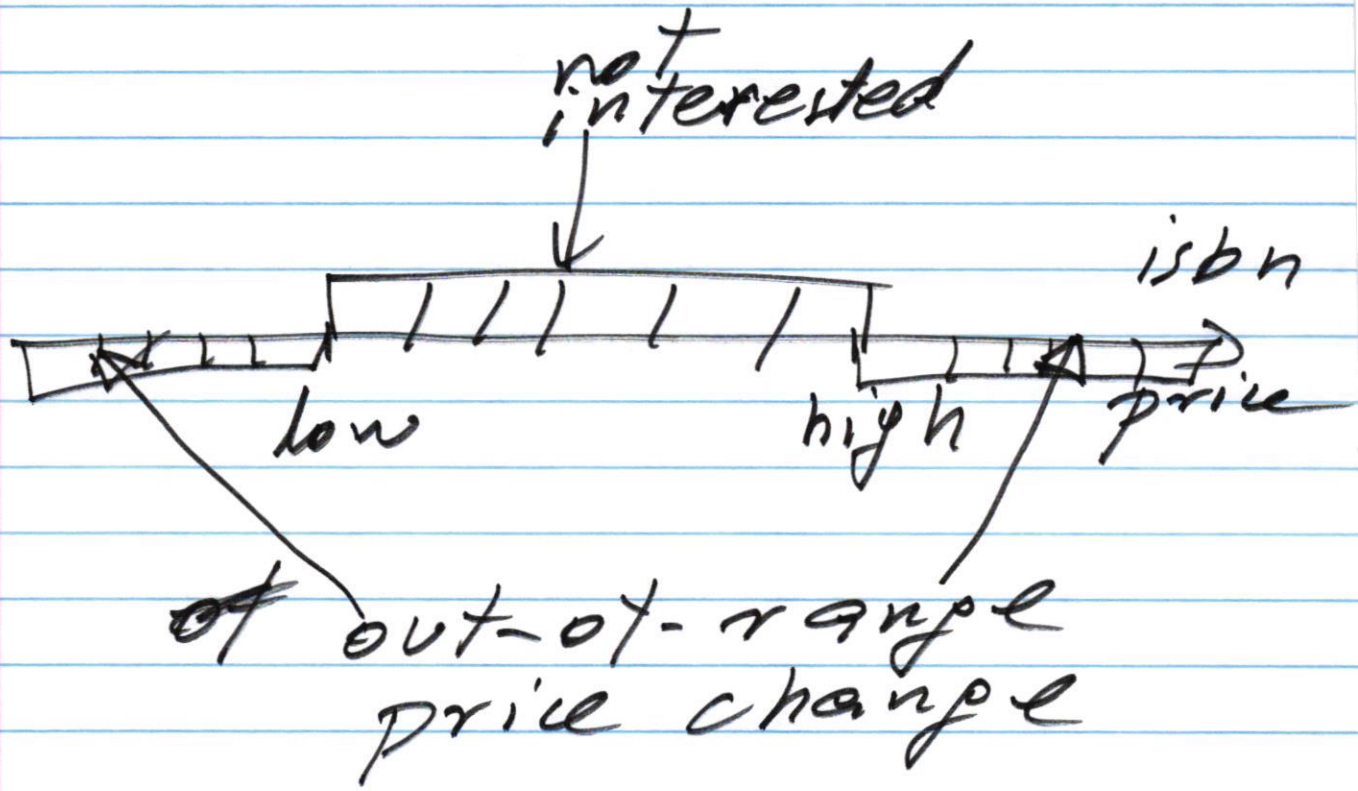
2. state classes are strongly related

3. context class and state classes.
   weak couplings.

Homework #1

problem #1

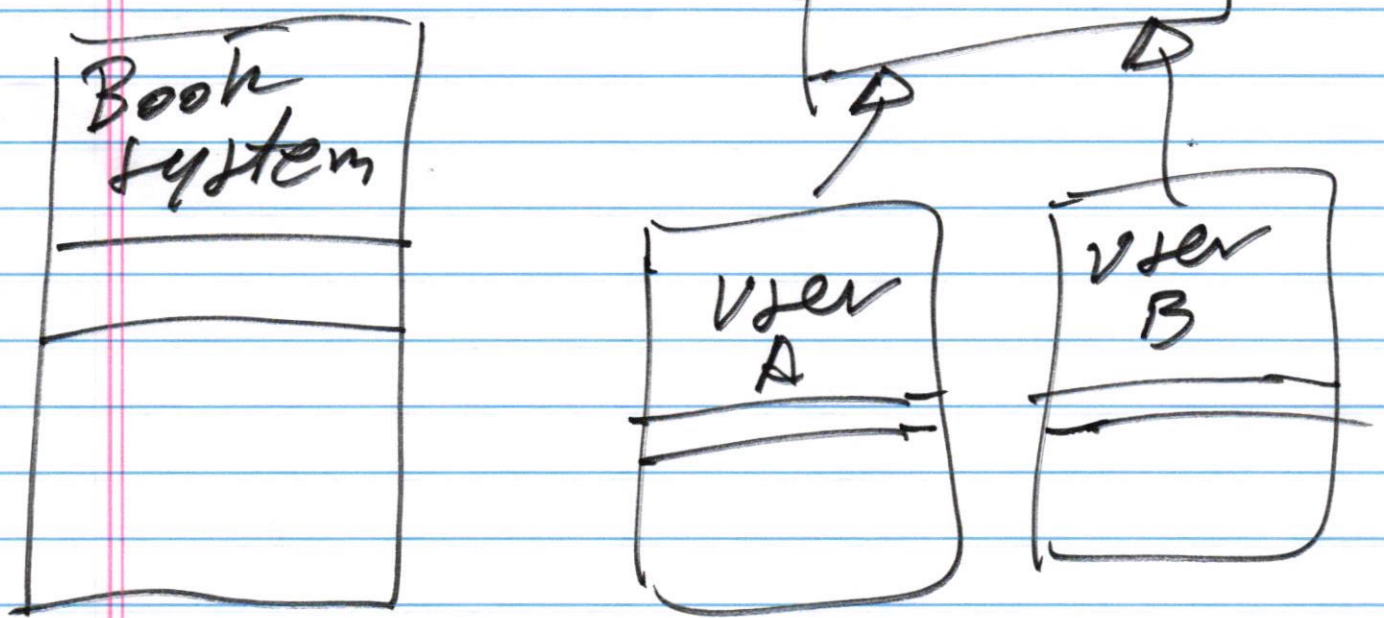
Observer pattern



Notification

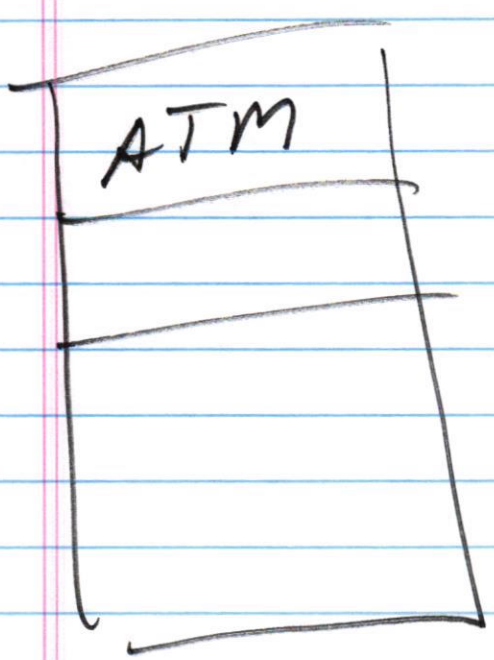
register(isbn, low, high, user pointer)

unregister(isbn, user pointer)

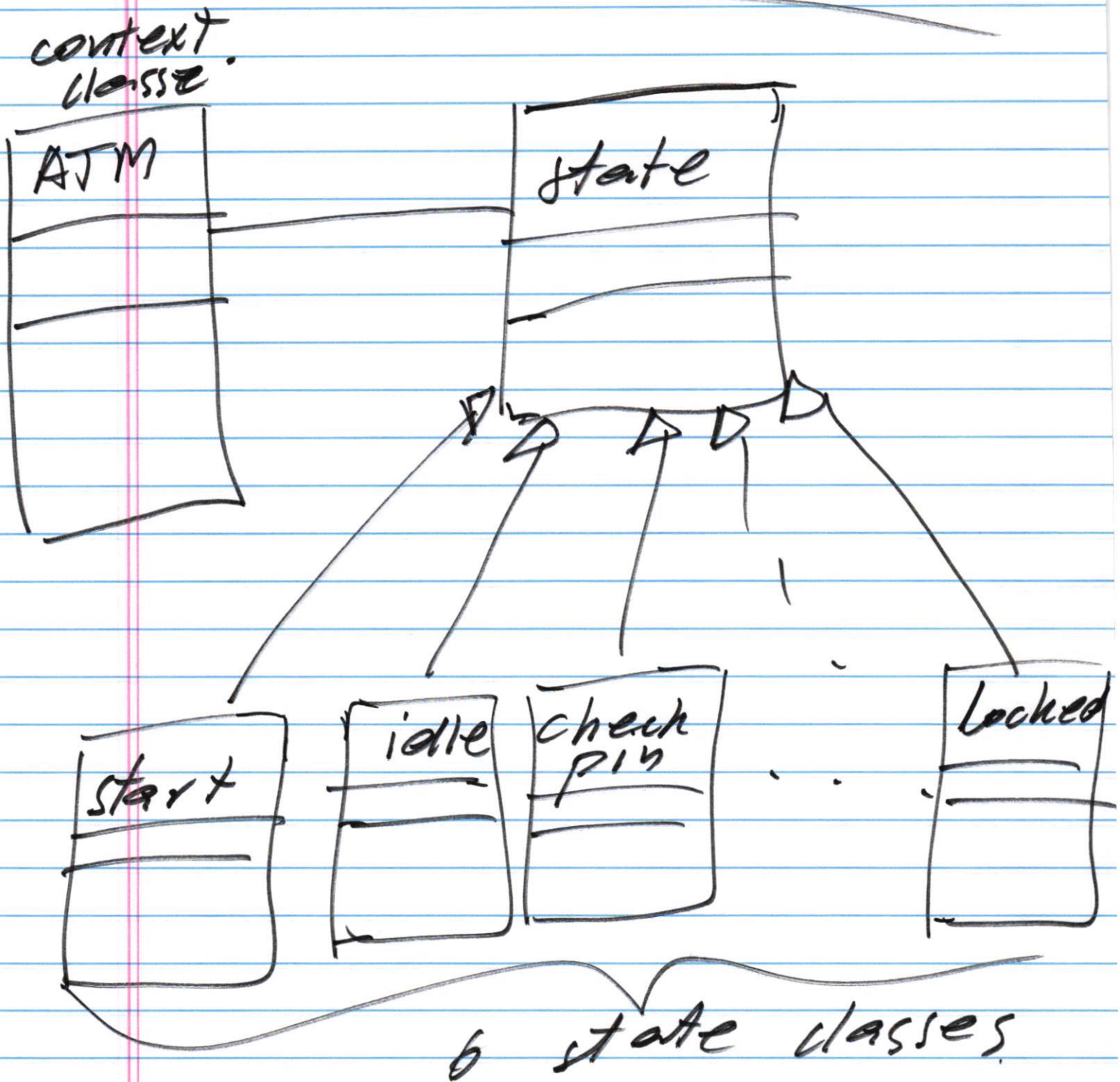


Problem #2

State pattern



state pattern



part1. centralized version
of state pattern

part2. de-centralized
version of state
pattern.

HOMEWORK ASSIGNMENT #1

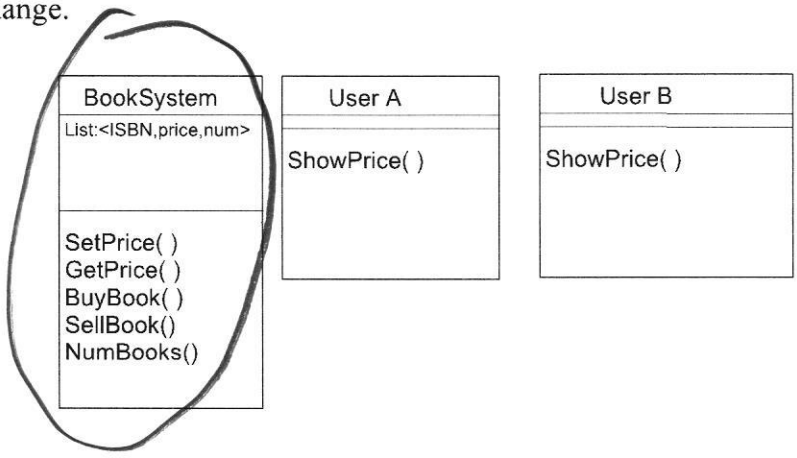
CS 586; Fall 2025
Due Date: **September 18, 2025**
Late homework: 50% off
After **September 23,** the homework assignment will not be accepted.

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**.

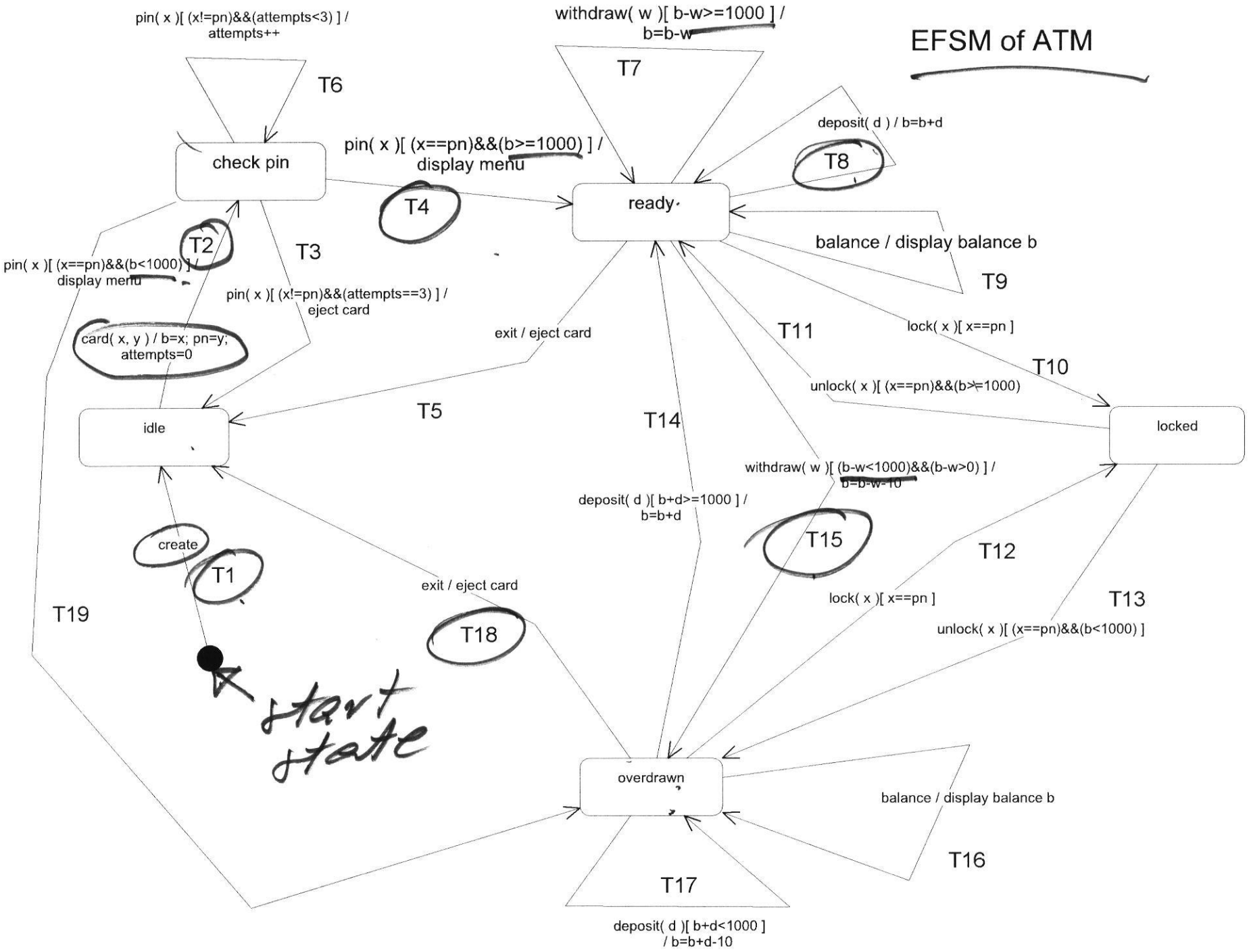
For each solution:

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