Homework #1

is posted

OO design patterns

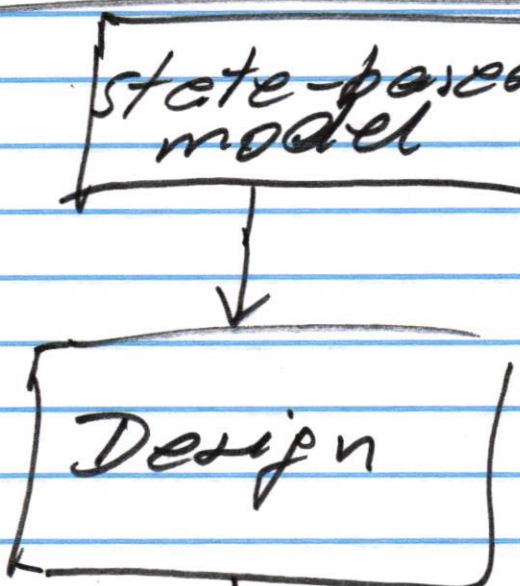
(*) item description pattern

(*) whole-part -"-

(*) observer -"-

* state -"-

Design state-based system



State Pattern

design the state-based system based on provided model

state pattern

state-based system

A system that can be characterized by a set of states

+

transitions between states

---

* embedded system
* event based -n-

State-based model

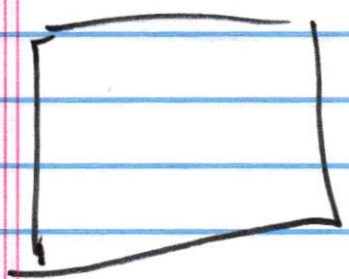
- FSM Finite State Machine
- EFSM: Extended Finite State Machine
- ...

FSM

a set of states

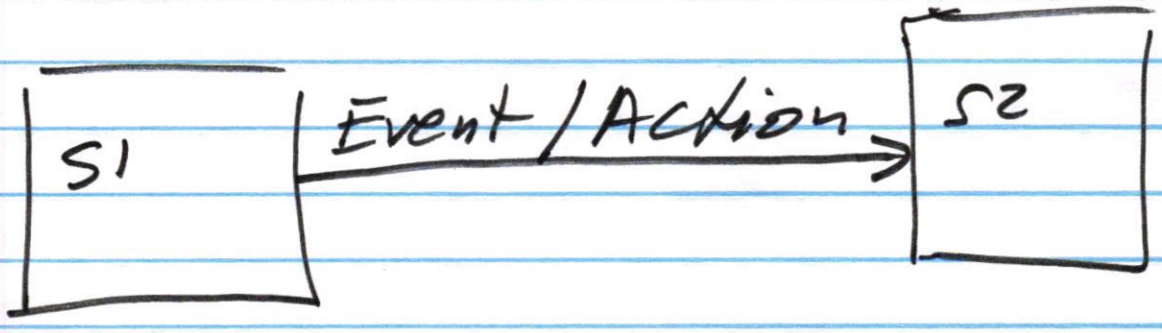
+

transitions between states



state

transition



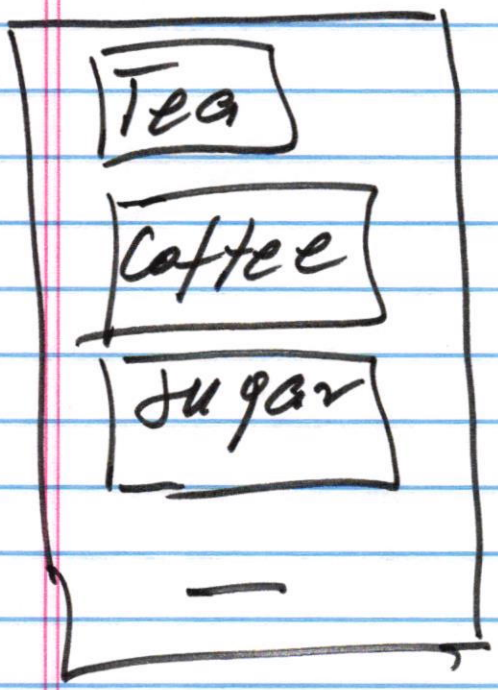
(1) system/model is in state S1

(2) "Event" is invoked

(3) transition from S1 to S2

(4) "Action" is performed

Vending Machine



Events

Tea: tea button is pressed

Coffee: coffee button is pressed

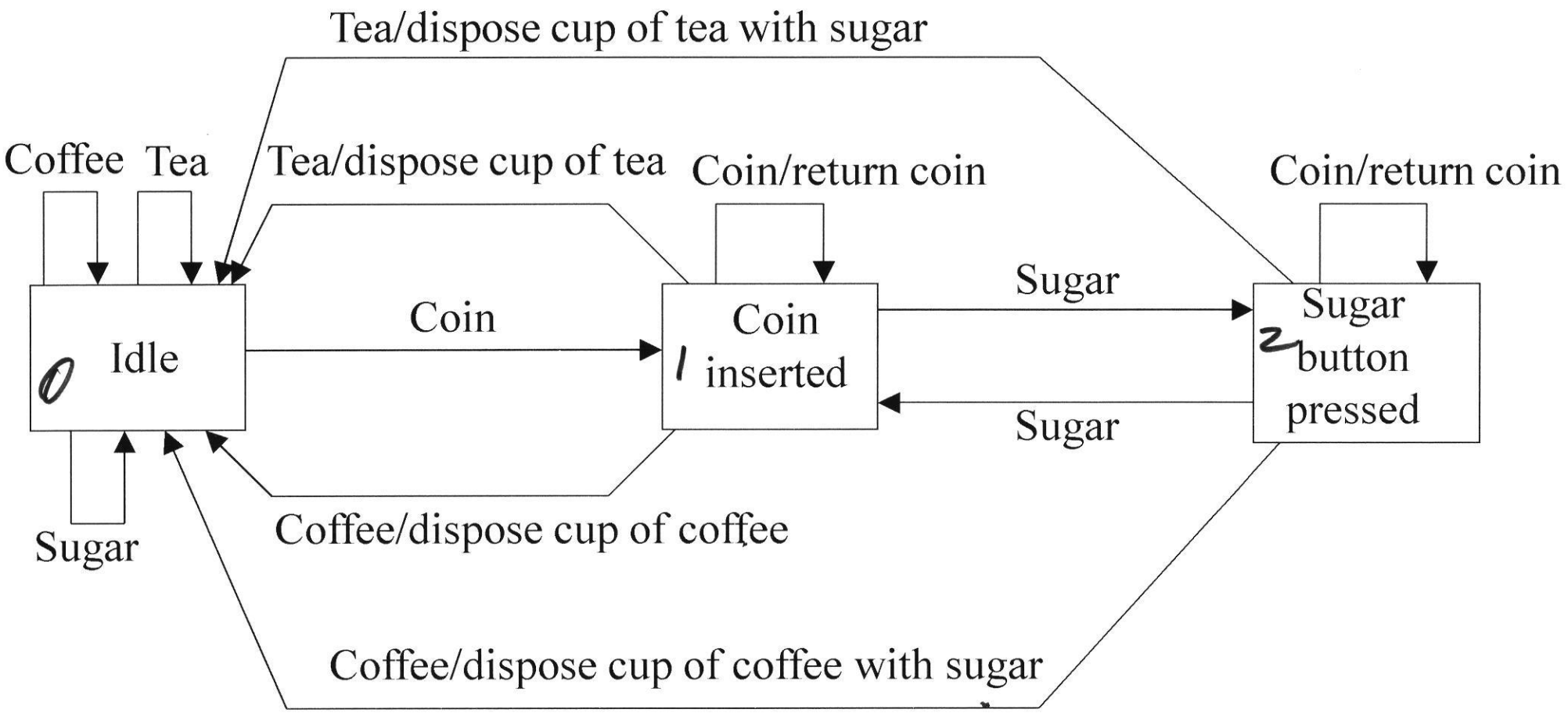
Sugar: sugar button pressed

Coin: coin is inserted

Actions

* dispose up of tea with/without sugar
* dispose up of coffee with/without sugar
* return coin

Vending Machine

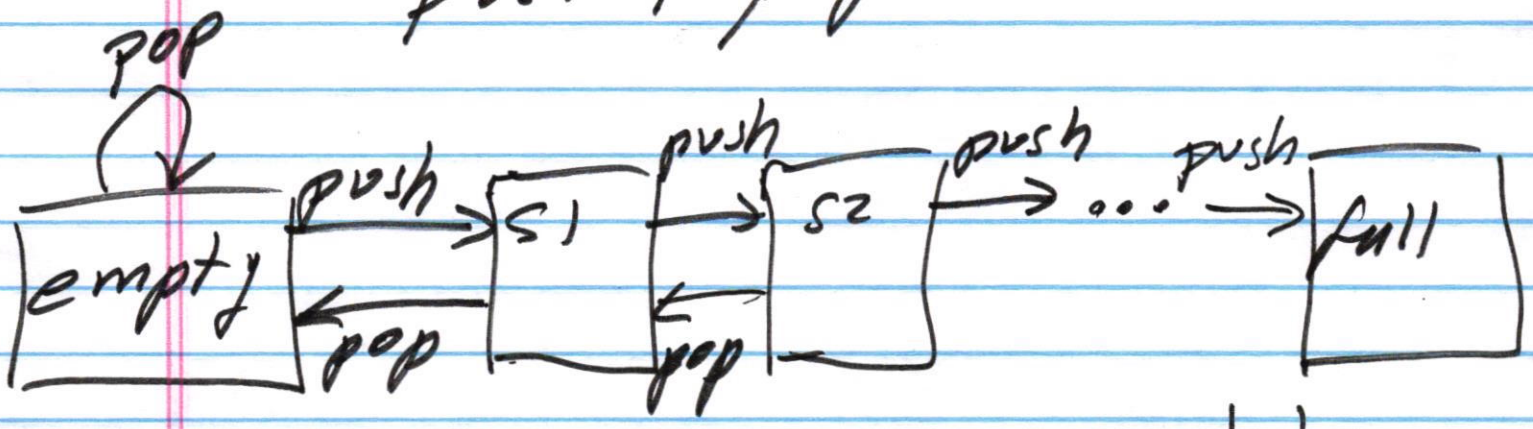


FSM: Finite State Machine

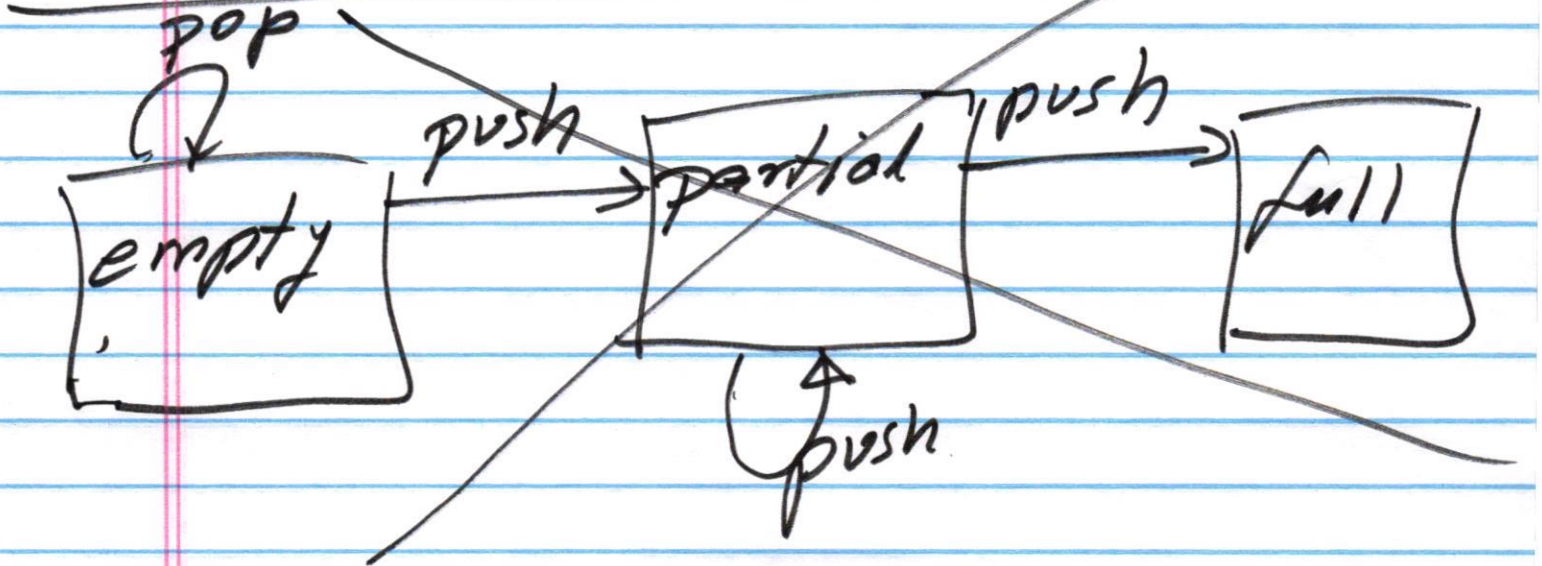
state explosion!!

component: stack

push, pop



state explosion!!



EFSM

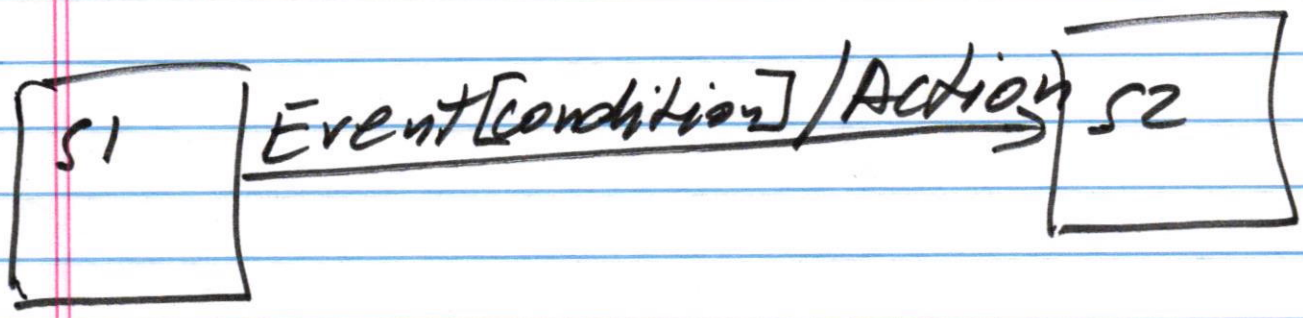
Extended Finite State Machine

⇓

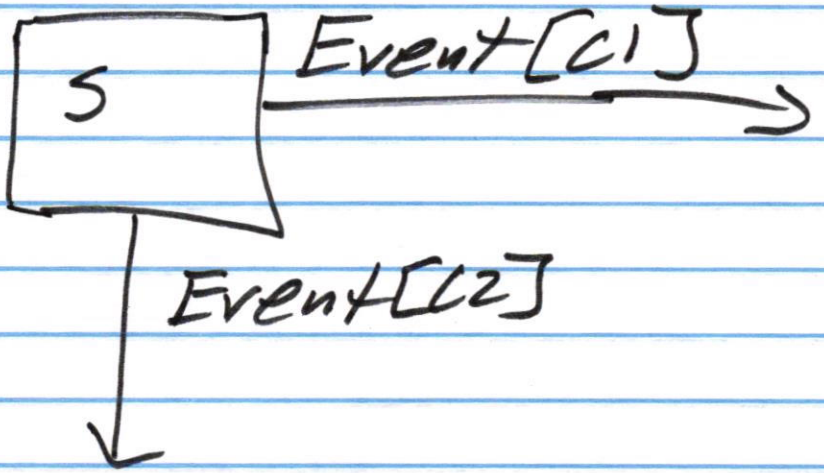
FSM

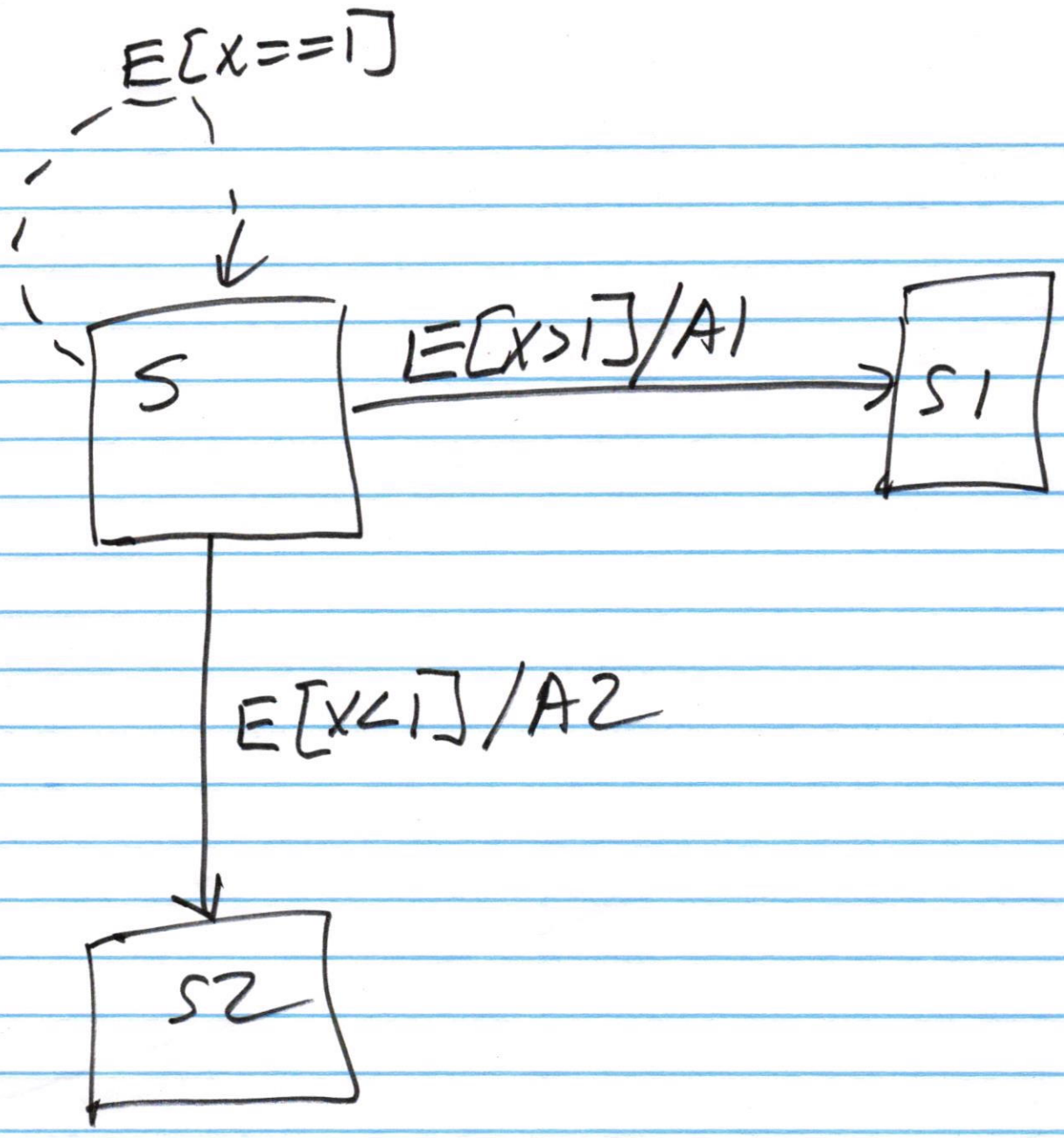
+

* variables
* conditions associated with transitions.

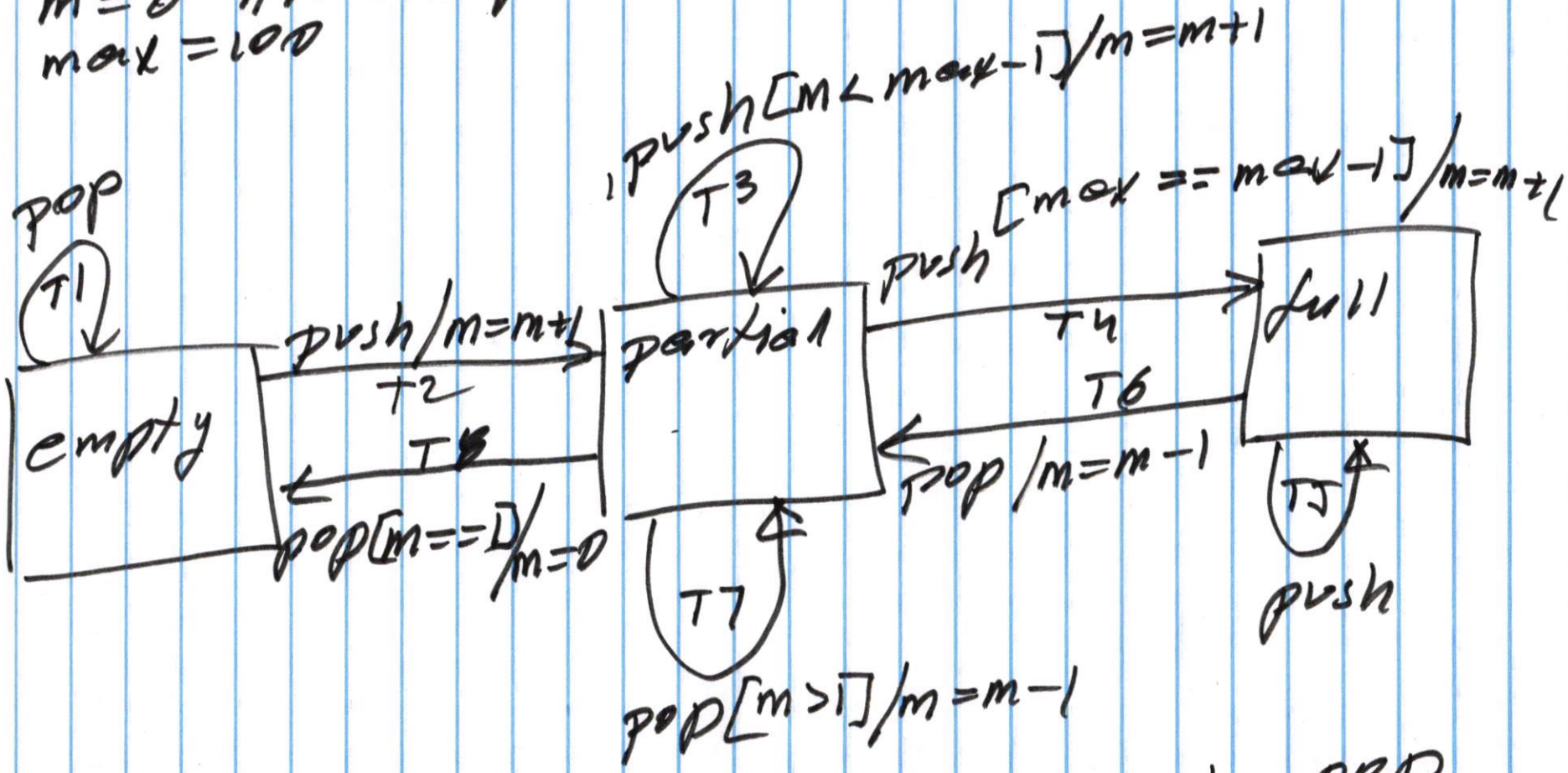


1. System is in S1
2. "Event" is invoked
3. "condition" is true
4. transition from S1 to S2
5. "Action" is performed



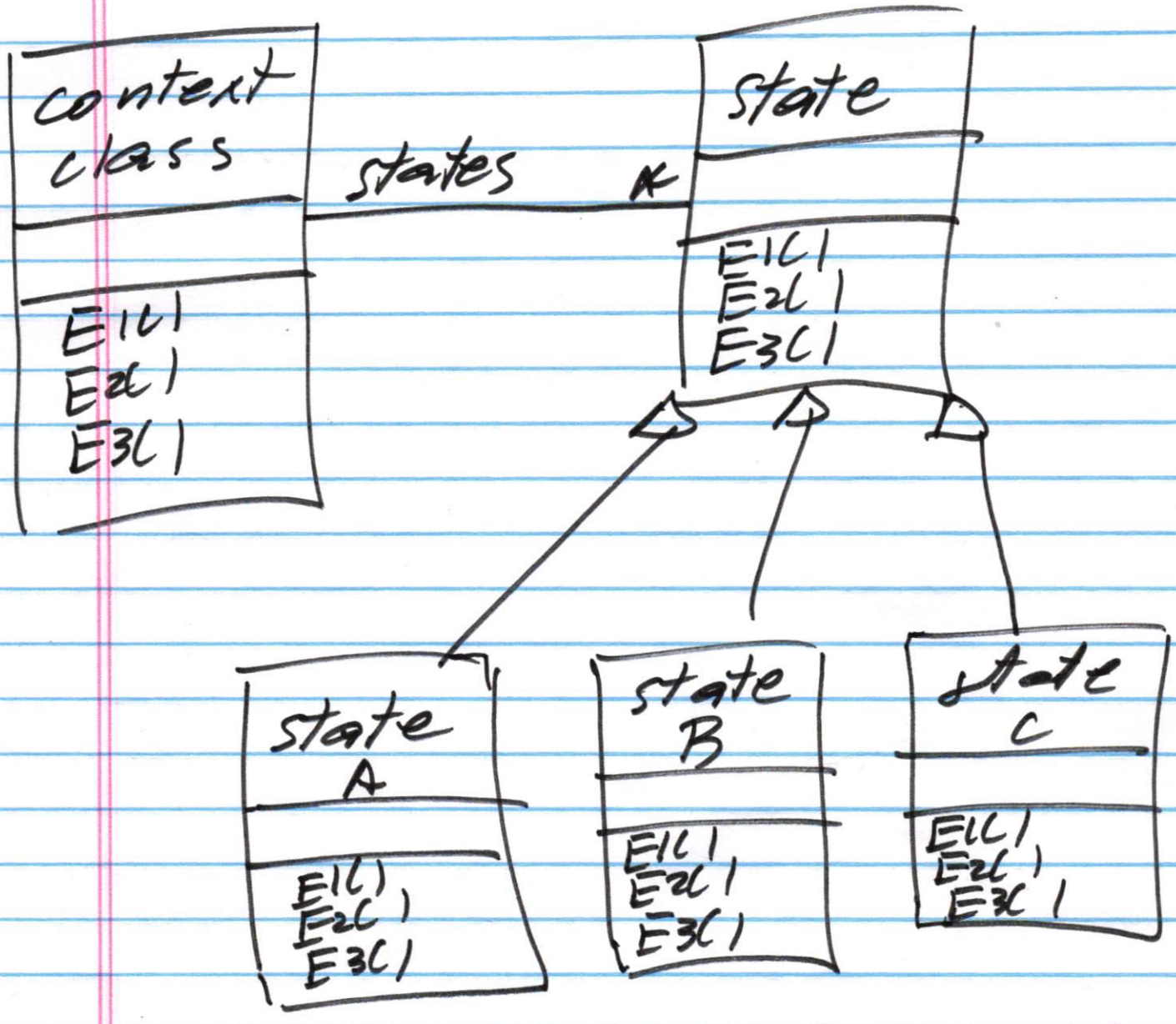

E[x==1]
S
E[x>1]/A1
S1
E[x<1]/A2
S2

m: counter
m = 0 // initially
max = 100



push, pop, pop, push, push, pop
T2 T8 T1 T2 T3 T7

State Pattern



Responsibilities

1. perform actions
2. change states

---

1. state classes are responsible for performing actions

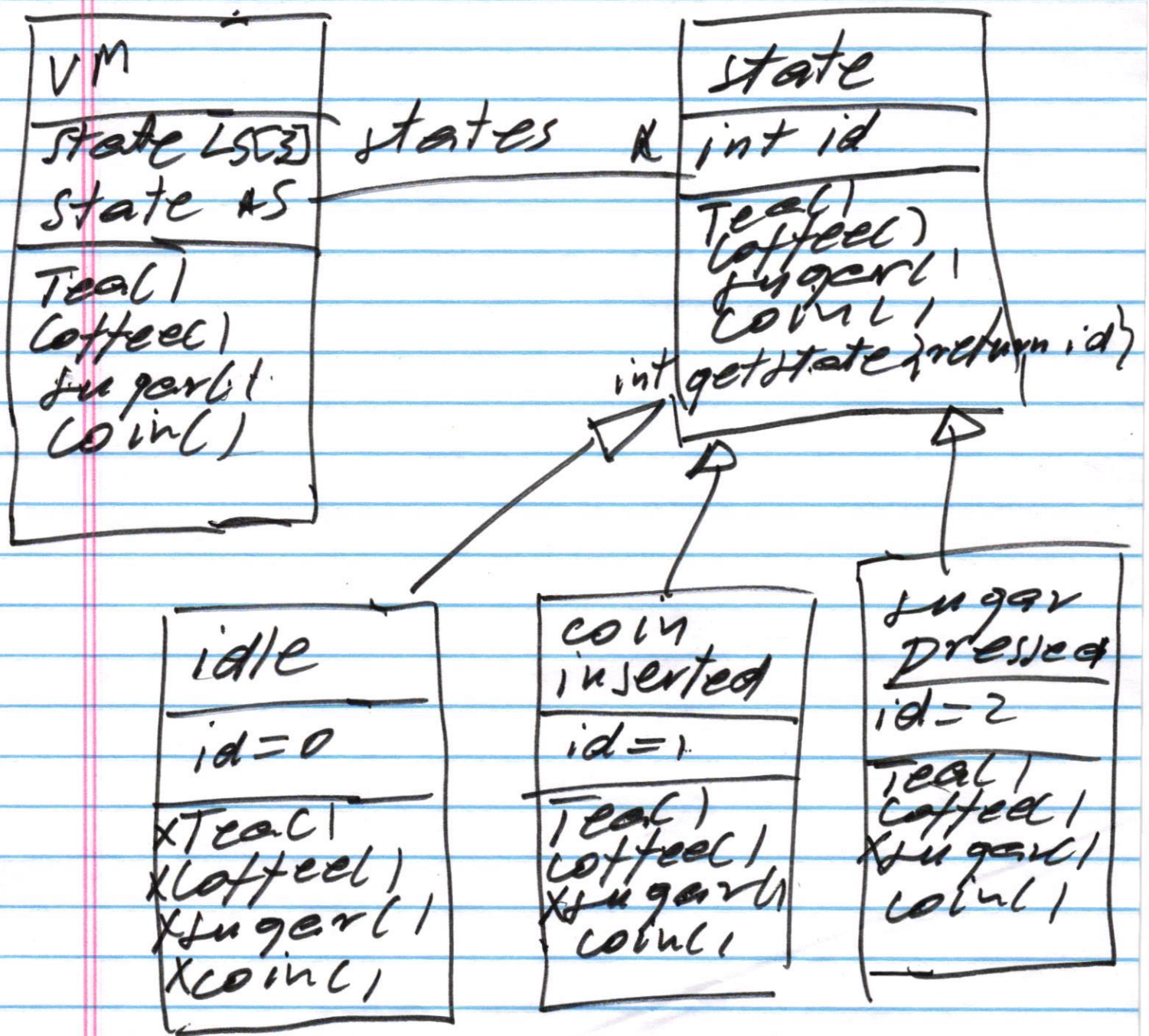
2. changing states

   (a) centralized solution
   change of state is done by context class

   (b) decentralized solution
   change of state is done by state classes

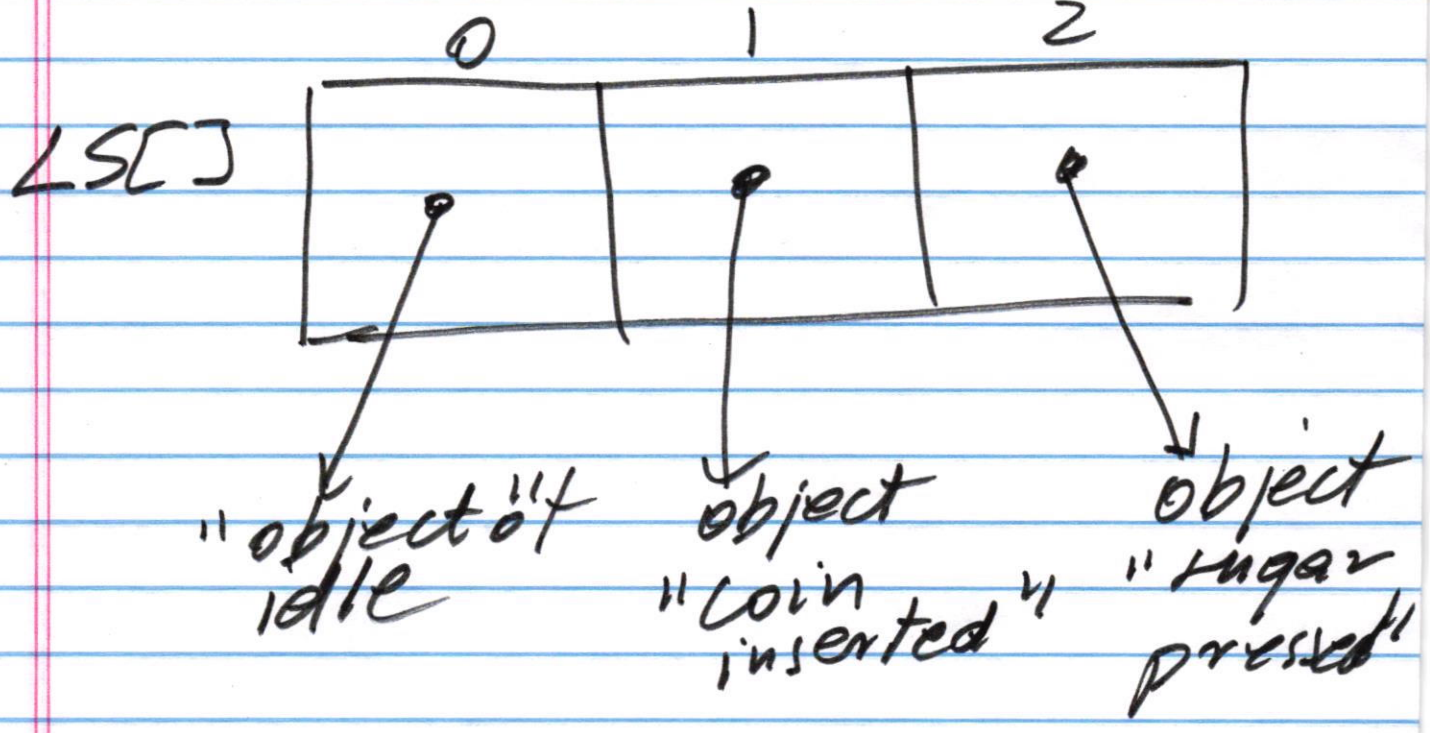
state pattern

centralized solution



VM class

list of states



S is a pointer to current state

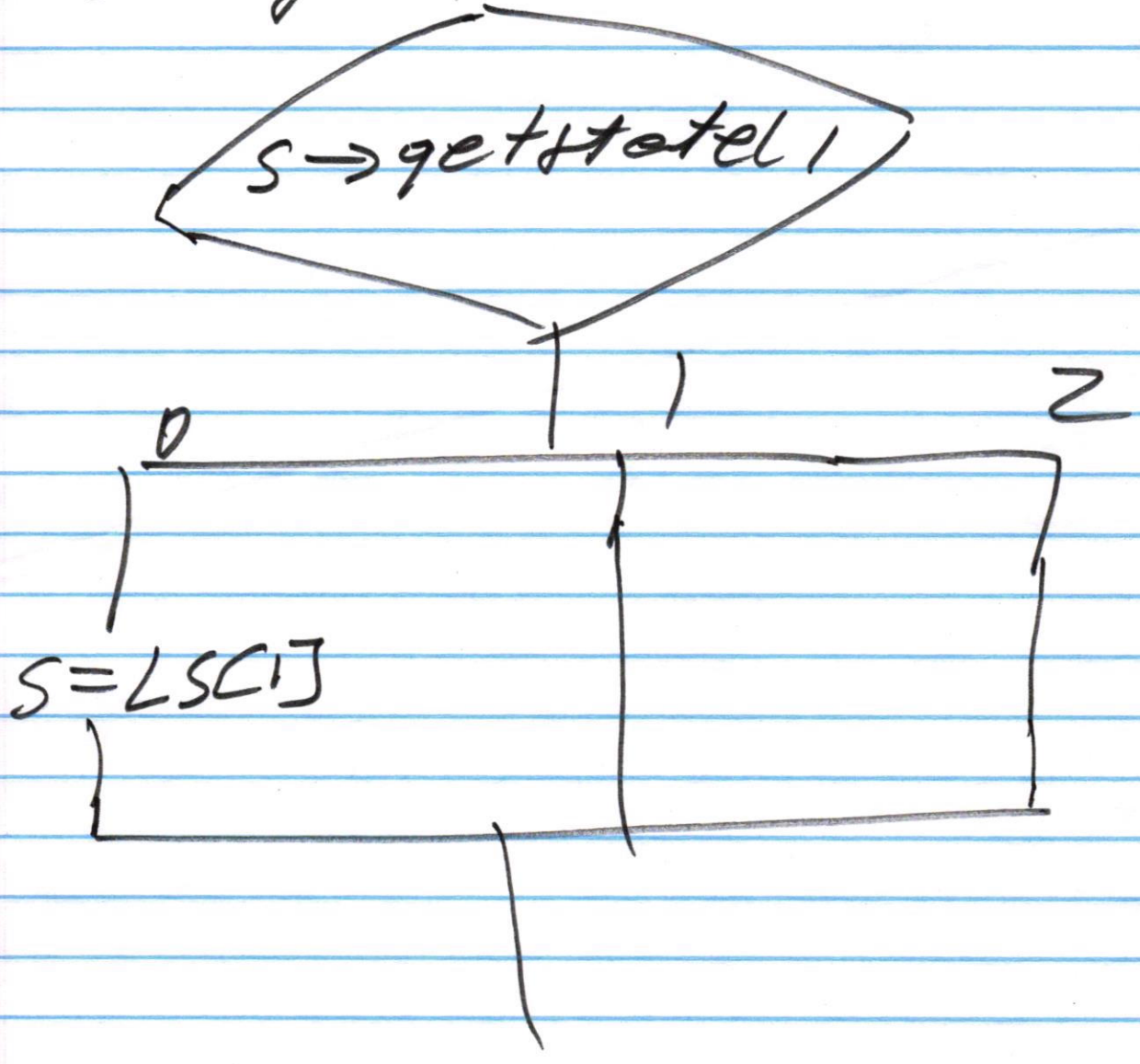
Initially, S points to "idle" object.

S = LS[0]

VM class

coin()

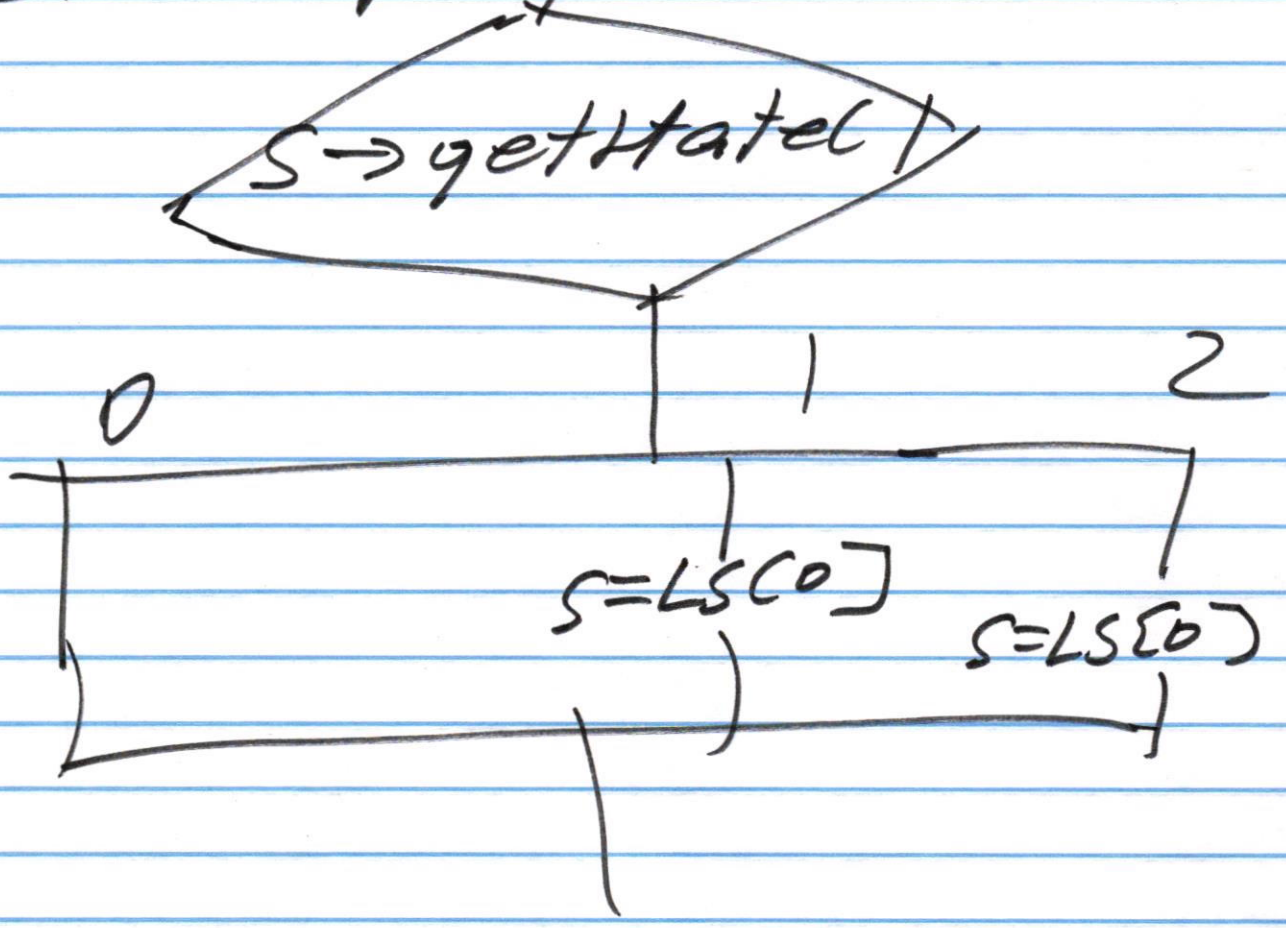
1. S -> coin() // perform action
2. change state.



VM class

teal() operation

1. S -> teal() // action
2. change of state

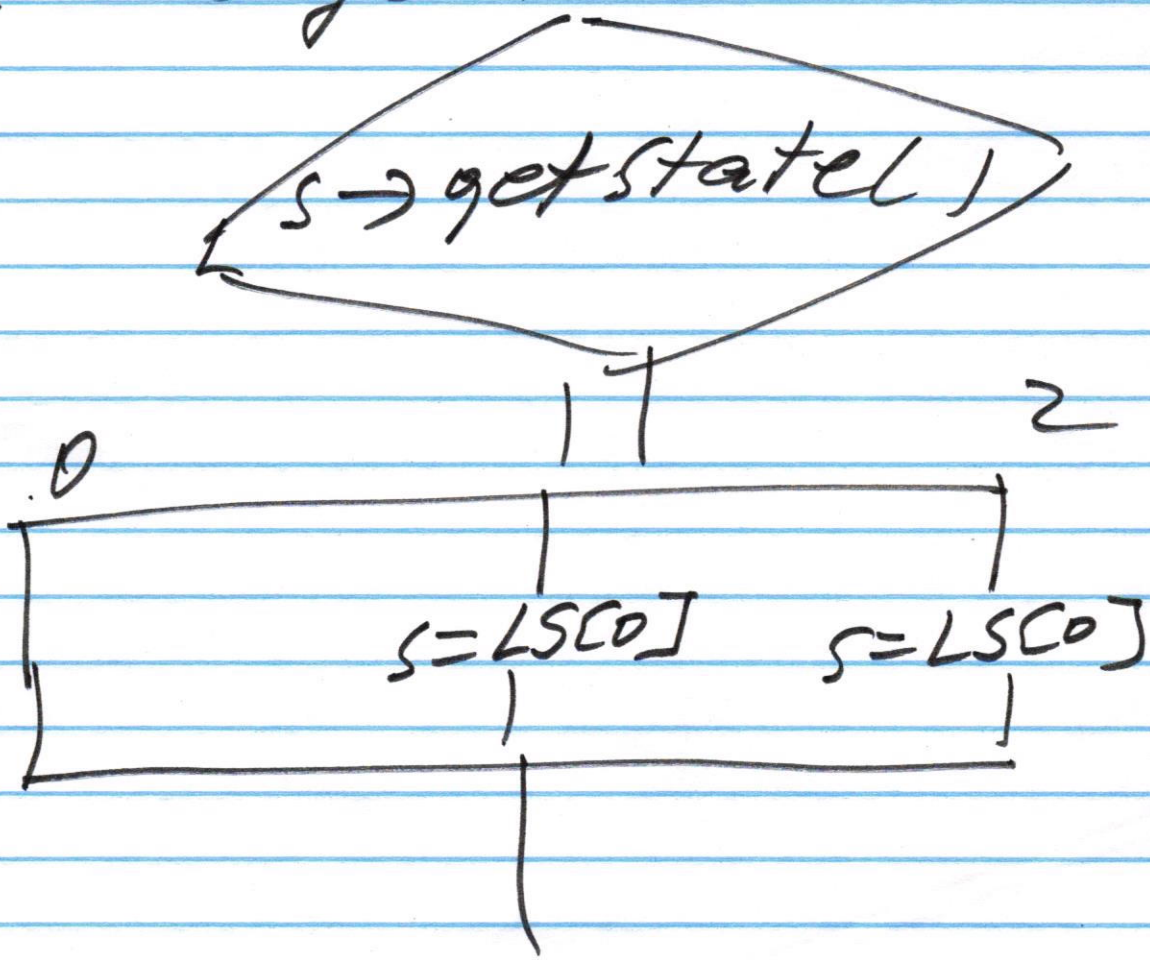


VM class

coffeeC) operations

1. S→coffeeC) //Action

2. Change State

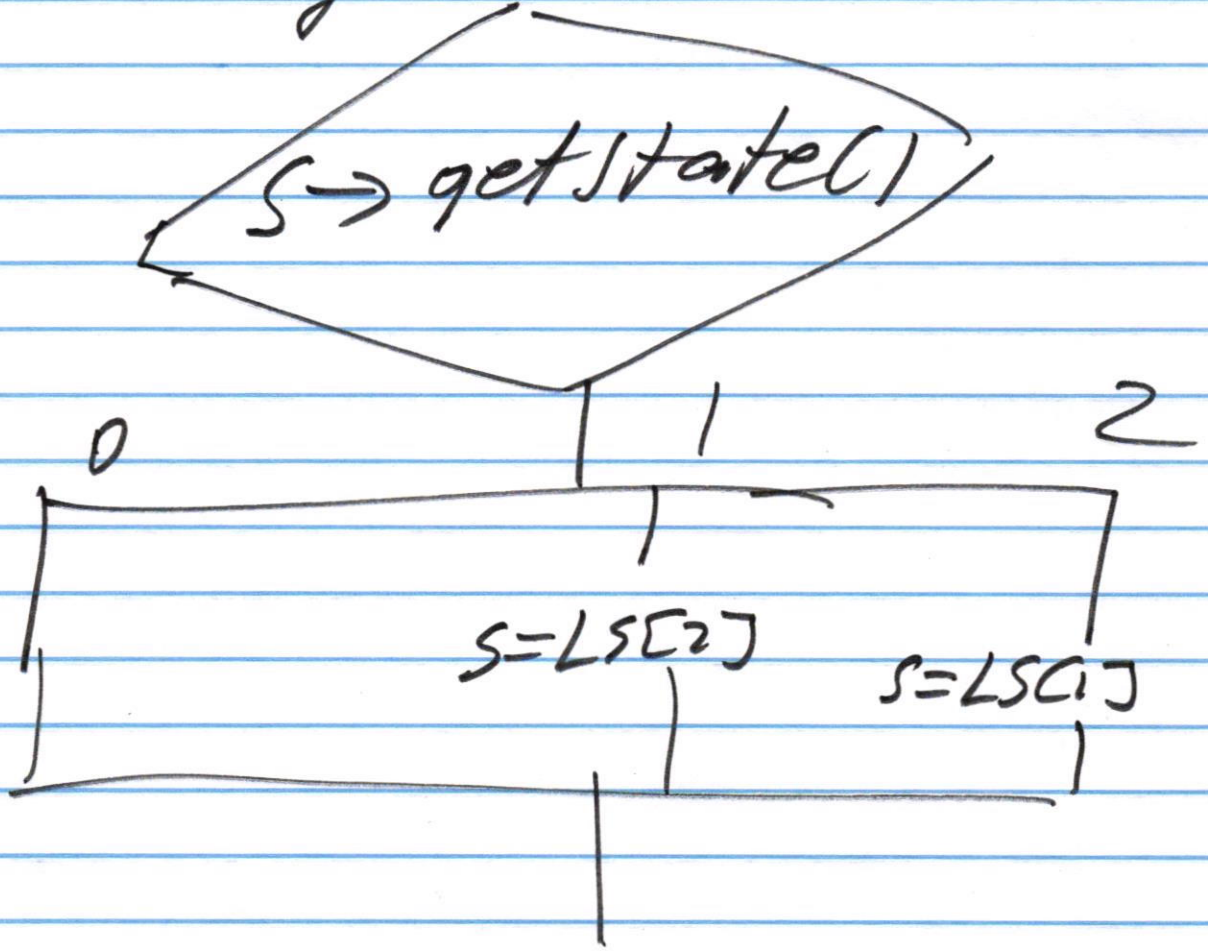


VM class

fuqar() operation

1. S → fuqar() // Action

2. change state



coin inserted class

Tea() // dispose cup of tea

coffee() // dispose cup of coffee

coin() // return coin

sugar pressed class

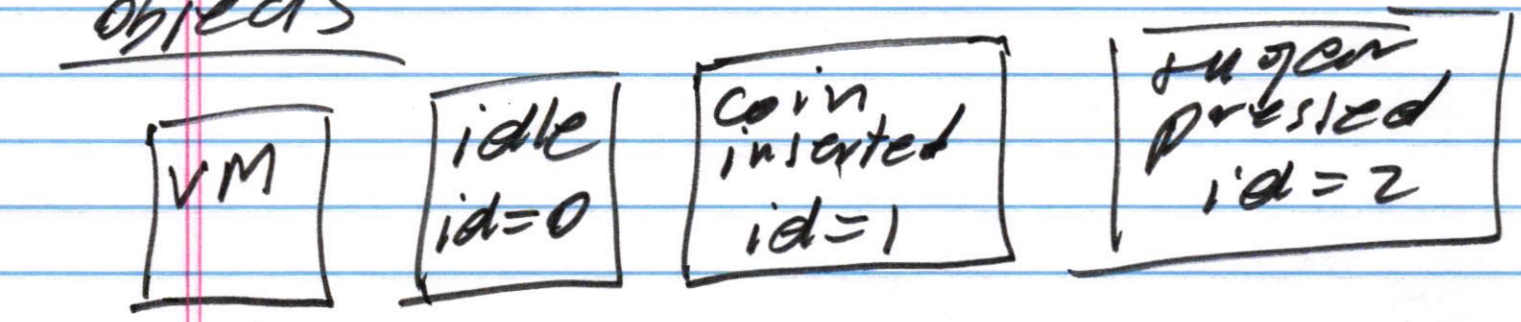
Tea() // dispose cup of tea with sugar

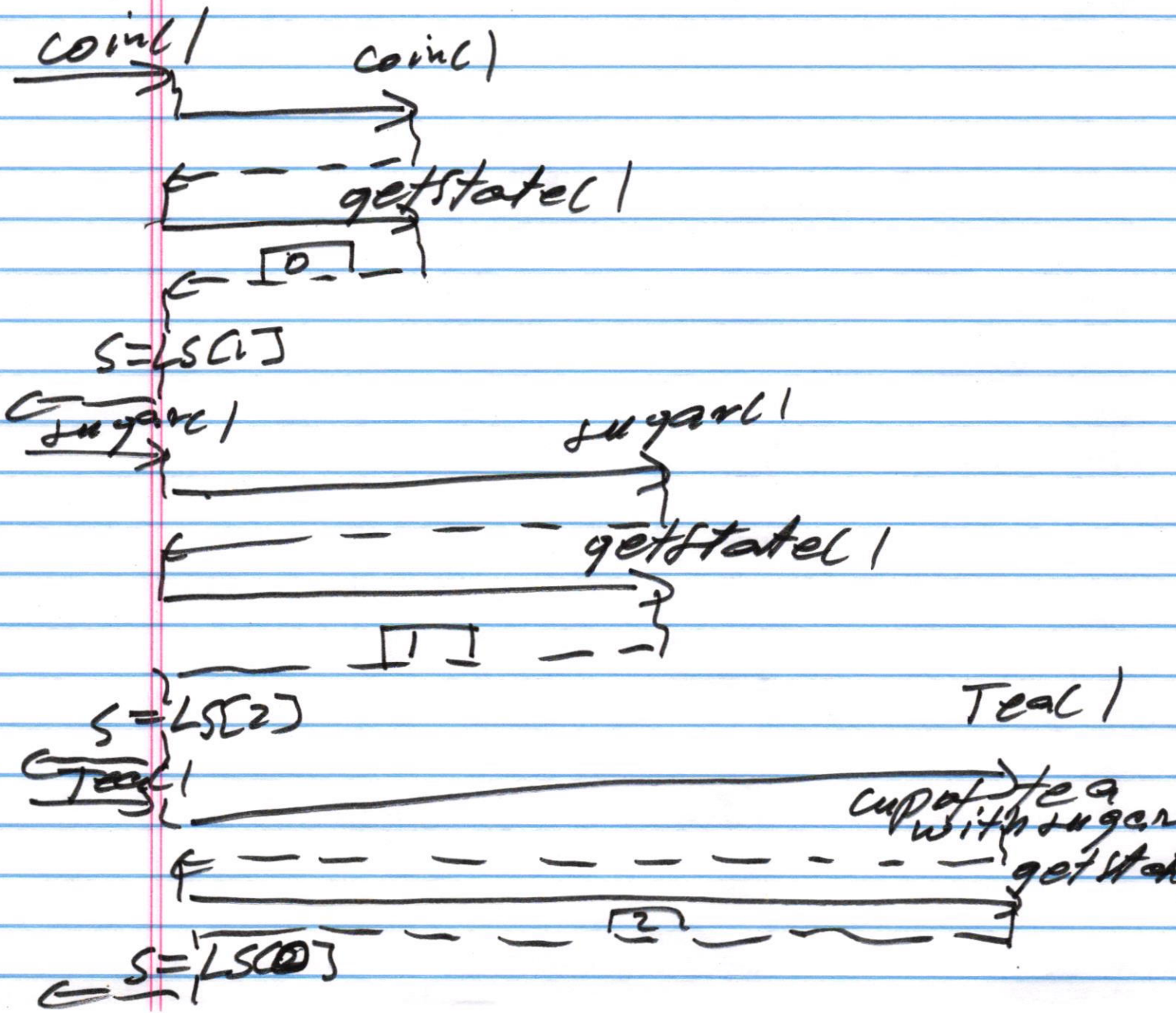
coffee() // dispose cup of coffee with sugar

coin() // return coin.

sequence diagram

objects

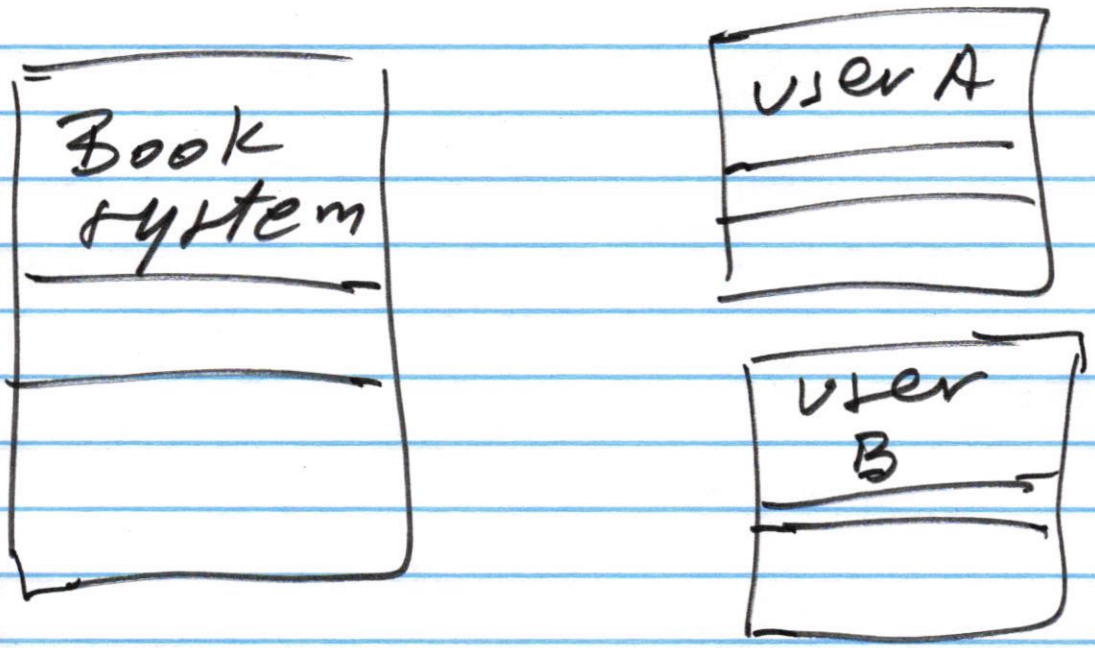




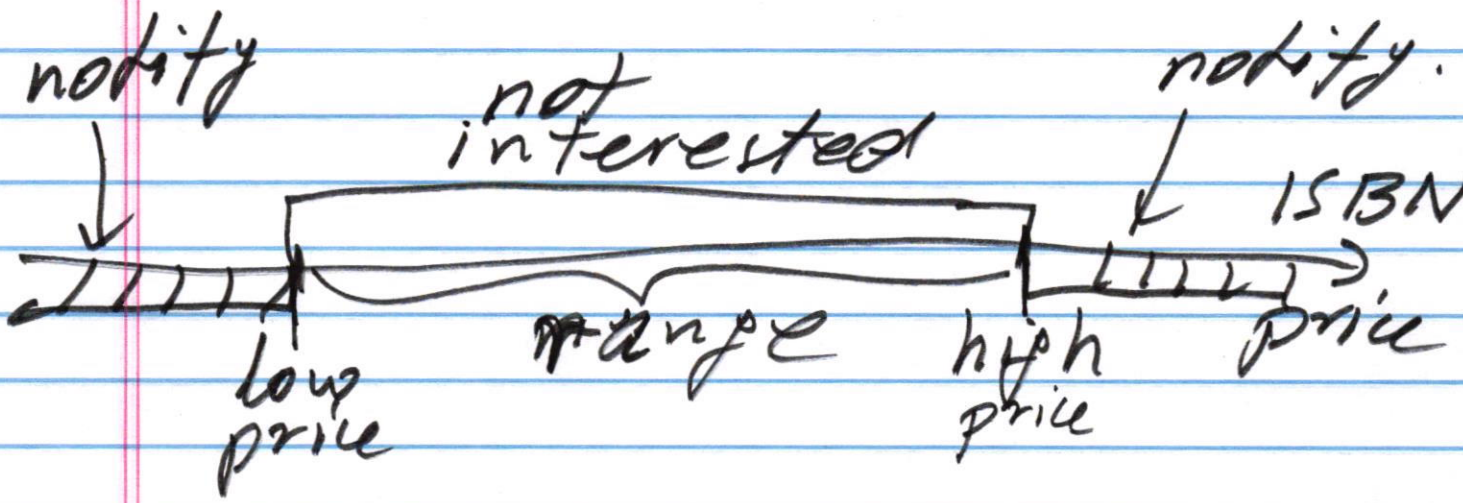
Homework #1

Problem #1

Observer pattern



users/observers are interested in out-of-range price change



Observer pattern

users/observers must

register/unregister

register(ISBN, low, high, pointer to user)

unregister(ISBN, pointer to obs.)

Problem #2

State pattern

HOMEWORK ASSIGNMENT #1

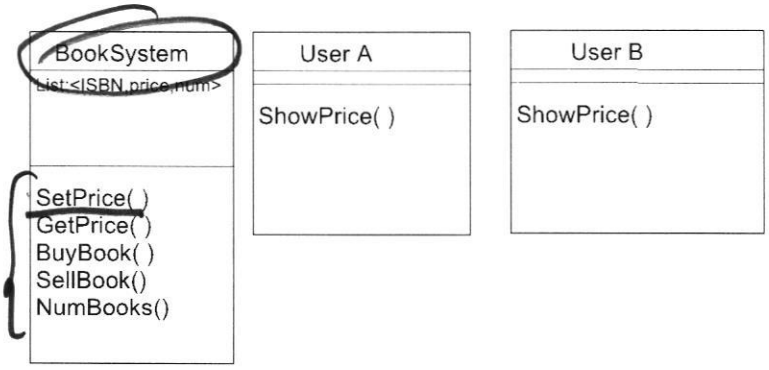
CS 586; Fall 2025
Due Date: **September 18, 2025**
Late homework: 50% off
After **September 23,** the homework assignment will not be accepted.

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}

b, pn, attempts

EFSM of ATM