- Homework #3 is posted
- Project Part #2 is posted
- Exam #3

  Monday, April 29

  5:00 - 7:00 pm

  Closed books and notes.

  The coverage is posted

COVERAGE FOR EXAM #3

CS 586; Spring 2024

Exam #3 will be held on **Monday, April 29, 2024,** between **5:00-7:00p.m.**
Location: 104 Stuart Building
The exam is a **CLOSED books and notes** exam.

Coverage for the exam:

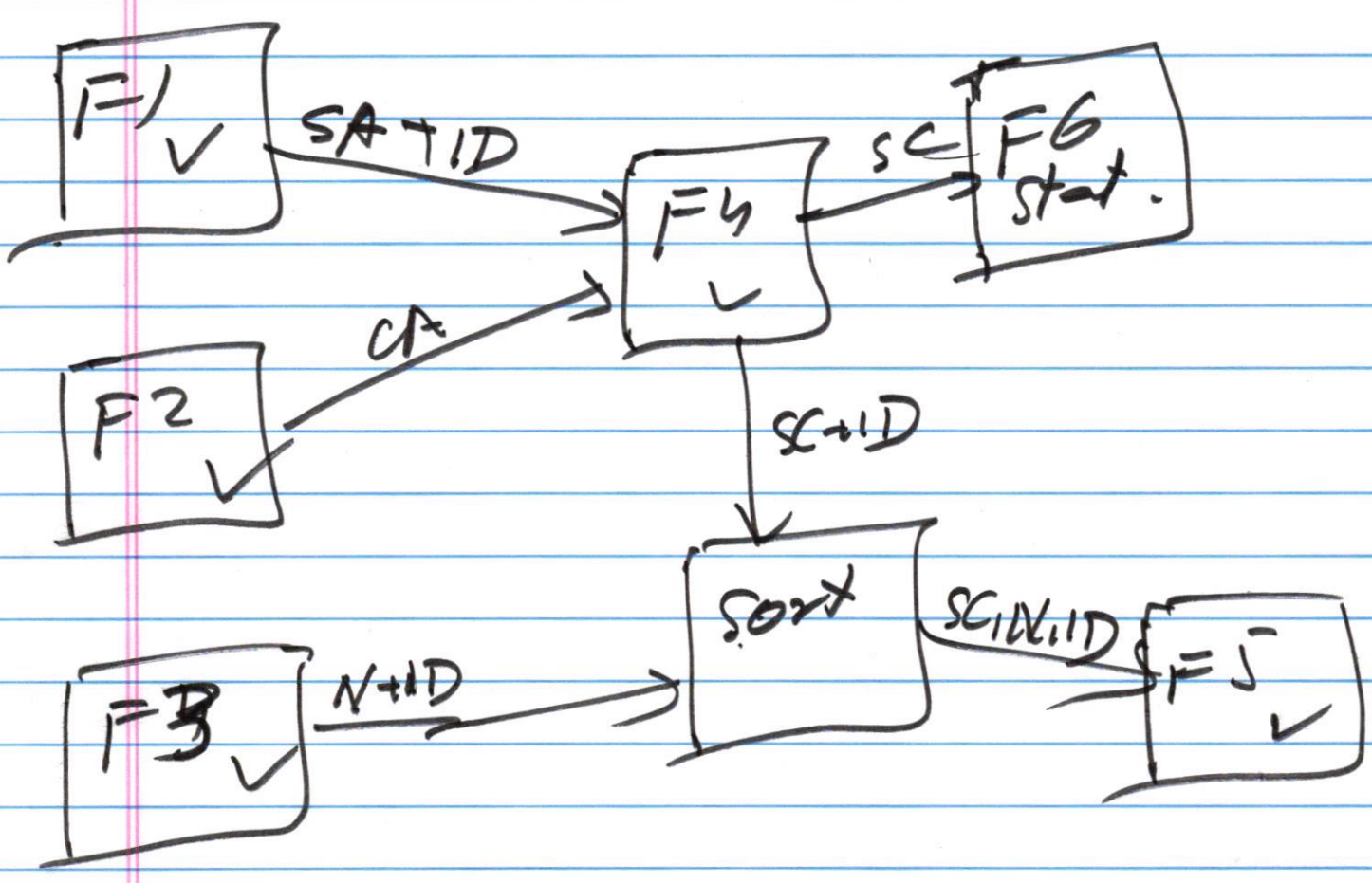
- OO design patterns: item description, whole-part, observer, state, proxy, adapter, strategy and abstract factory patterns. [Textbook: Sections 3.1, 3.2; Section 3.4 (pp. 263-275); Section 3.6 (pp.339-343); Handout #1, class notes]
- Interactive systems. Model-View-Controller architectural pattern [Textbook: Section 2.4, pp. 123-143]
- Client-Server Architecture
  - Client-Dispatcher-Server [Section 3.6: pp. 323-337]
  - Client-Broker-Server Architecture [Textbook: Section 2.3; pp. 99-122]
- Layered architecture [Textbook: Section 2.2; pp. 31-51]
- Pipe and Filter architecture [Textbook: Section 2.2; pp. 53-70]
- Adaptable Systems:
  - Micro-kernel architectural pattern [Textbook: Section 2.5, pp. 169-192]
- Fault-tolerant architecture [Handout #2]
  - N-version architecture
  - Recovery-Block architecture
  - N-Self Checking architecture
- Repository architecture [Textbook: Section 2.2; pp. 71-95]

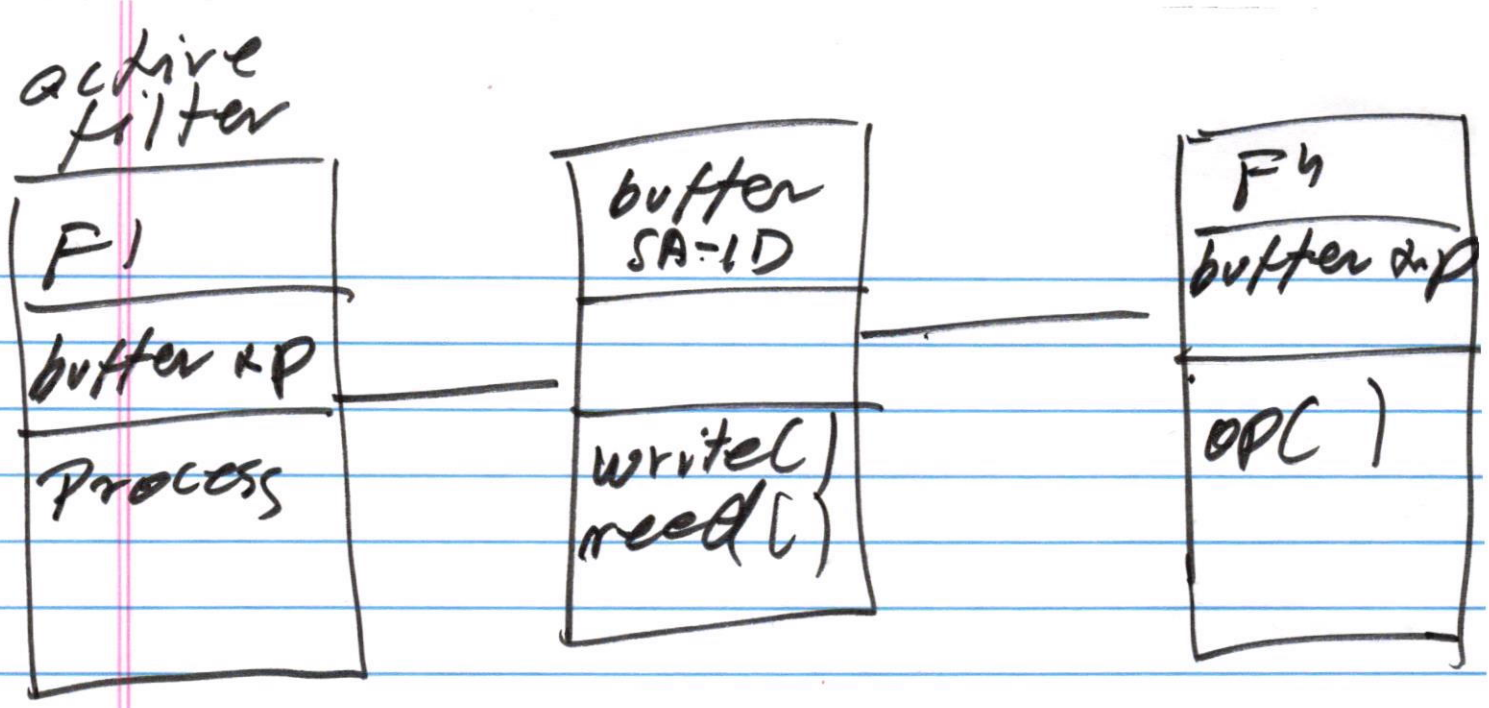
Sources:

- Textbook: F. Buschmann, et. al., Pattern-oriented software architecture, vol. I, John Wiley & Sons.
- Class notes
- Handouts

Homework #3

Problem #1
Pipes and Filters





Process()

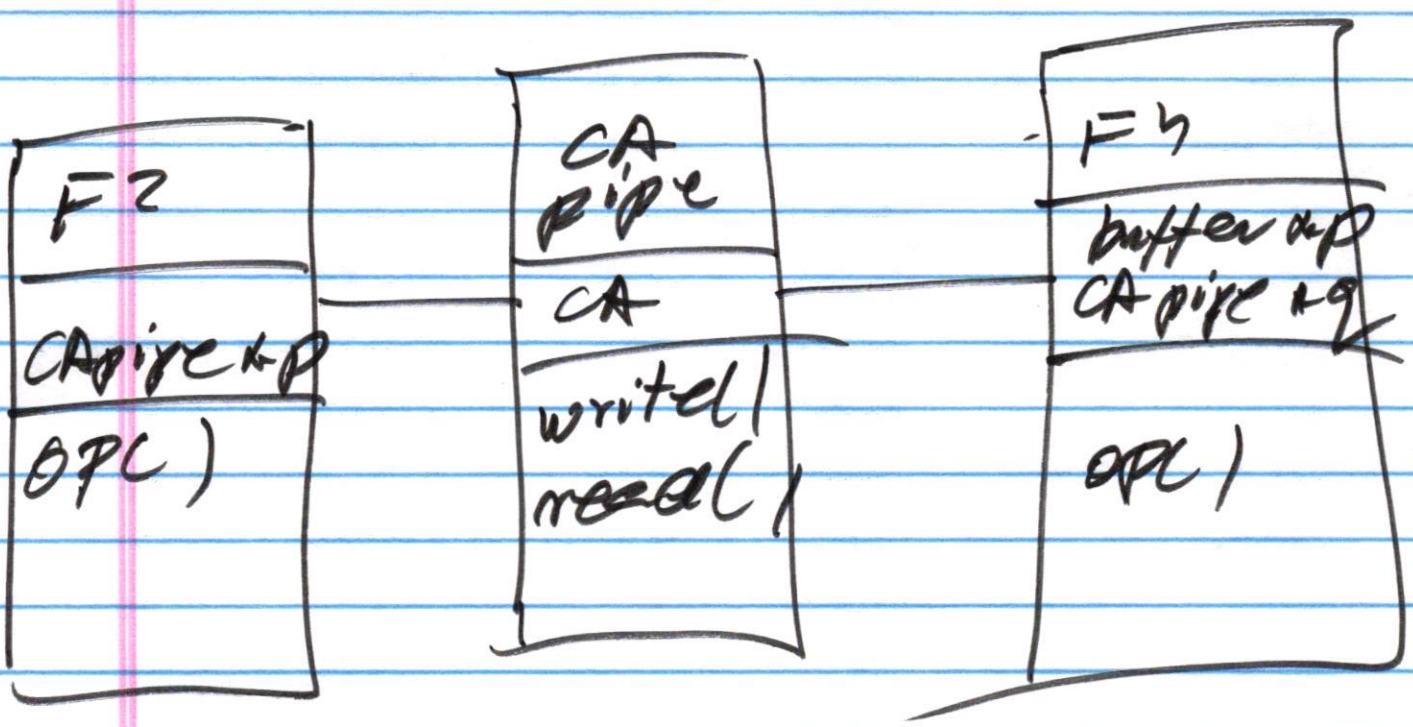
read SA+ID

p -> write(SA, ID)

OP()

~~p -> read~~

p -> read(SA, ID)



OPC)

read(CA)

p → write(CA)

}

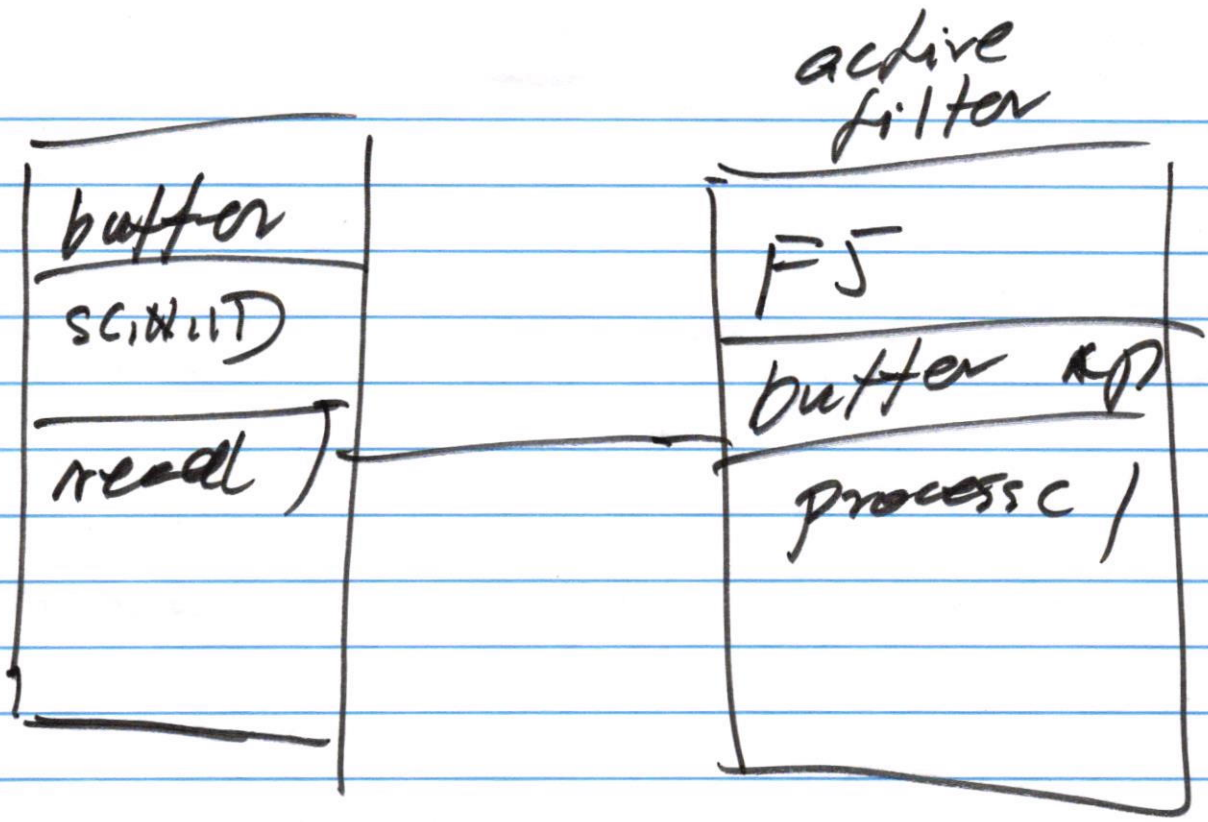
OPC)

q → read(CA)

p → read(SA, ID)

compute

store

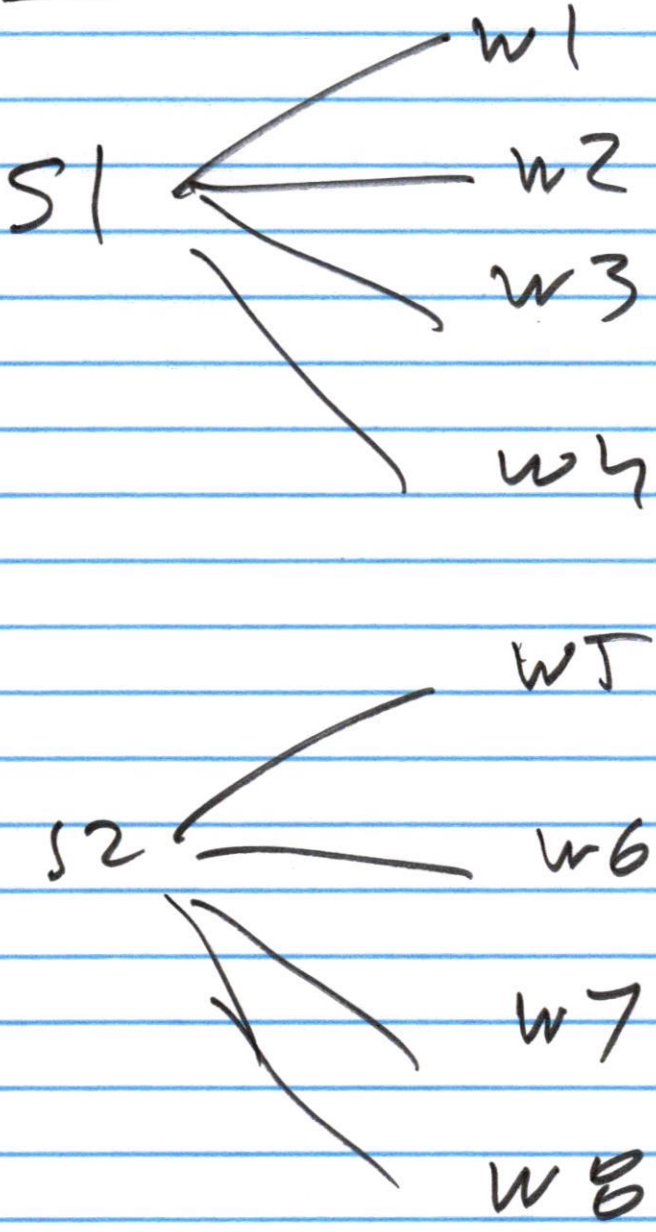


Process()

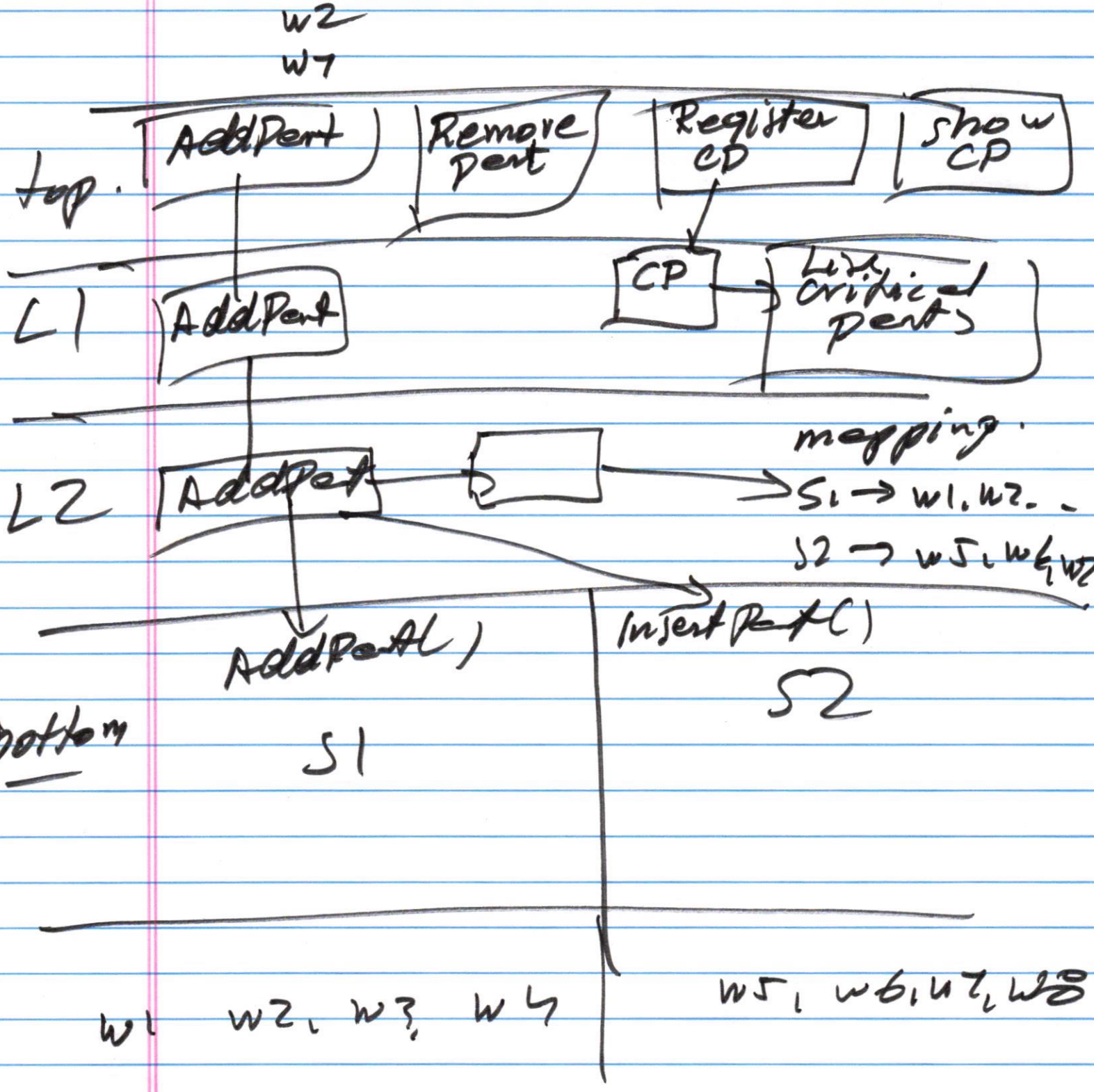
P->read(SC,N,ID)
print(SC,N,ID)

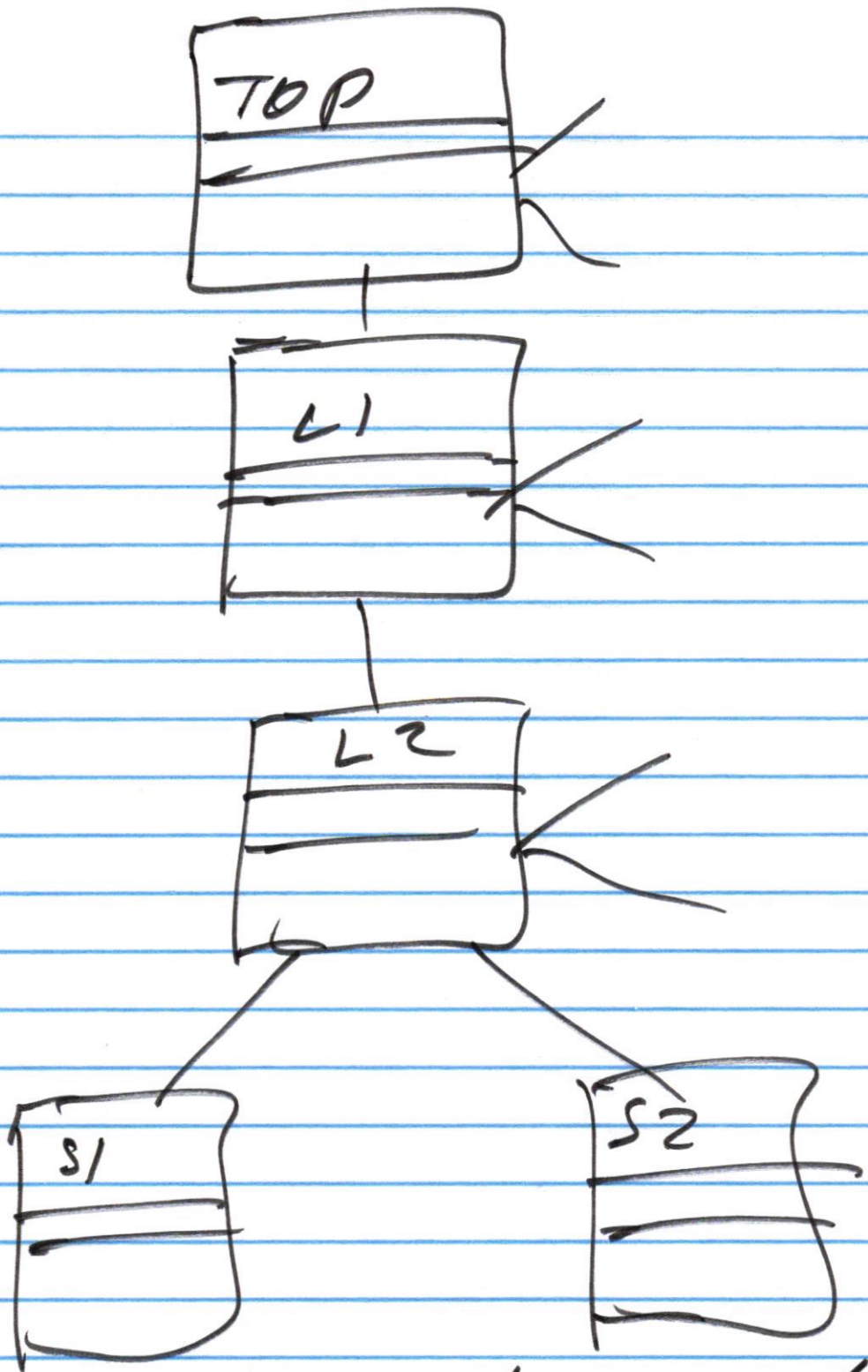
Problem #2

Strict Layered Architecture



4 layers

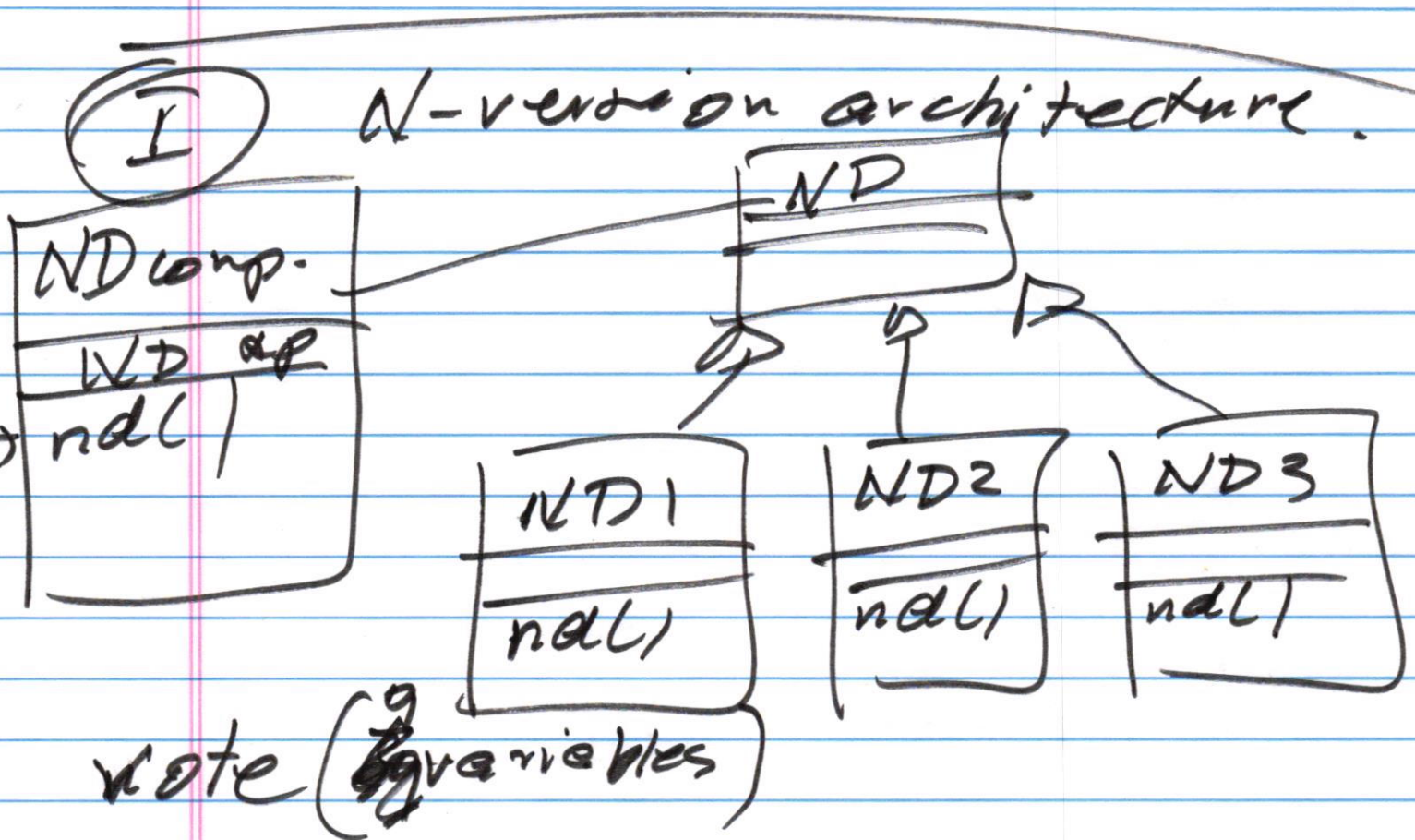

w2
w7
AddPart
Remove part
Register CP
show CP
top.
L1
AddPart
CP
List Critical parts
mapping.
L2
AddPart
S1 → w1, w2, ..
S2 → w5, w6, w7
InsertPart()
AddPart()
S2
bottom
S1
w1 w2, w3, w4
w5, w6, w7, w8



Do not provide pseudocode for S1 and S2

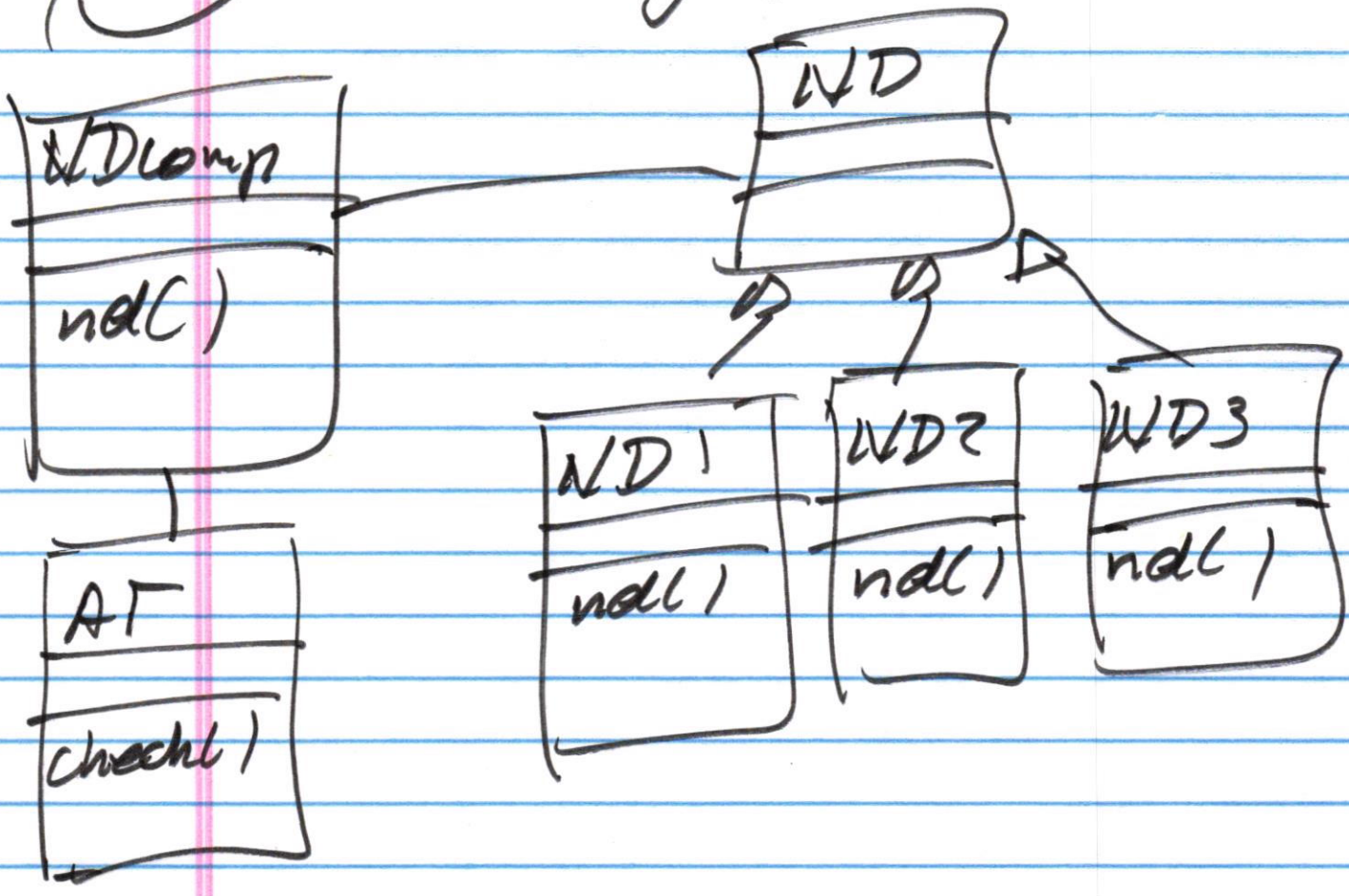
Problem #3

Fault tolerant architecture



no majority!
return randomly
one of the outputs.

(2) Recovery Block Architecture.



AT: acceptance test

check(m, d, y, m1, d1, y1)

≡

return pass/fail

HOMEWORK ASSIGNMENT #3

CS 586; Spring 2024
Due Date: **April 16, 2024**
Late homework 50% off
After **April 22,** 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 the Blackboard. The submission **must** be as one PDF file (otherwise, a 10% penalty will be applied).

PROBLEM #1 (35 points)
Consider the problem of designing a system using **Pipes and Filters** architecture. The system should provide the following functionality:

- Read student's test answers together with student's IDs.
- Read student's names together with their IDs.
- Read the correct answers for the test.
- Compute test scores.
- Compute test statistics: mean and standard deviation
- Report test scores in **descending score order** with student names and their IDs.
- Report test statistics

It was decided to use a Pipe and Filter architecture using the existing filters. The following existing filters are available:
Filter #1: this filter reads student's test answers together with student's IDs.
Filter #2: this filter reads correct answers for the test.
Filter #3: this filter reads student's names together with their IDs.
Filter #4: this filter computes test scores.
Filter #5: this filter prints test scores with student names in the order as they are read from an input pipe.

Part A:
Provide the Pipe and Filter architecture for the Grader system. In your design, you should use all existing filters. If necessary, introduce additional Filters in your design and describe their responsibilities. Show your Pipe and Filter architecture as a directed graph consisting of Filters as nodes and Pipes as edges in the graph.

Part B:
1. For the Pipe and Filter architecture of Part A, it is assumed that filters have different properties as shown below:
   a. Filter #1: active filter with buffered output pipe
   b. Filter #2: passive filter with un-buffered push pipes
   c. Filter #3: passive filter with un-buffered pull-out pipes
   d. Filter #4: passive filter with un-buffered pull-out pipes
   e. Filter #5: active filter with buffered input pipes
2. Use object-oriented design to refine your design. Each filter should be represented by a class. Provide a class diagram for your design. For each class identify operations supported by the class and its attributes. Describe each operation using **pseudo-code**. In your design, filters **should not be aware** of other filters.
3. Provide a sequence diagram for a typical execution of the system based on the class diagram of Step 2.

PROBLEM #2 (30 points)

There exist two inventory systems/servers (*Server-S1* and *Server-S2*) that maintain information about machine parts in warehouses, i.e., they keep track of the number of machine parts in warehouses. Machine parts may be added or removed from the warehouses and this should be reflected in the inventory system. Both servers (inventory systems) support the following services:

Services supported by **Server-S1**:

void AddPart (string w, string p)	//add part *p* to warehouse *w*, where *p* is a part ID
void DeletePart (string w, string p)	//deletes part *p* from warehouse *w*
int GetNumParts (string p)	//returns the total number of part *p* in all warehouses
int IsPart (string p)	//returns 1, if part *p* exists; returns 0, otherwise

Services supported by **Server-S2**:

void Insert_Part(string p, string w)	//adds part *p* to warehouse *w*
void Remove_Part(string p, string w)	//deletes part *p* from warehouse *w*
int Get_Num_Of_Parts(string p)	//returns the total number of part *p* in all warehouses
int Is_Part(string p)	//returns 1, if part *p* exists; returns 0, otherwise

The goal is to combine both inventory systems and provide a uniform interface to perform operations on both existing inventory systems using the **Strict Layered architecture**. The following top-layer interface should be provided:

void Add_Part (string p, string w)	//adds part *p* to warehouse *w*
void Remove_Part (string p, string w)	//deletes part *p* from warehouse *w*
int GetNumOfParts (string p)	//returns the total number of part *p* in all warehouses
int Is_Part (string p)	//returns 1, if part *p* exists; returns 0, otherwise
RegisterCriticalPart(string p, int minimumlevel)	
UnRegisterCriticalPart(string p)	
ShowCriticalParts()	

Notice that the top layer provides three additional services (*RegisterCriticalPart()*, *UnRegisterCriticalPart()*, and *ShowCriticalParts()*) that are not provided by the existing inventory systems. These services allow watching the status of critical parts. The user/application can register, *RegisterCriticalPart(string p, int minimumlevel)*, a critical part by providing its minimum level, i.e., a minimal number of parts of a specified part that should be present in all warehouses. When the number of parts of a critical part (a registered part) reaches the level below the minimum level, the system should store, e.g., in a buffer, the current status (number of parts) of the critical part. The current status of all critical parts whose level is below the minimum level can be displayed by invoking *ShowCriticalParts()* service. The service *UnRegisterCriticalPart()* allows removing a specified part from a list of critical parts.

Major assumptions for the design:

1. Users/applications that use the top layer interface should have the impression that there exists only one inventory system.
2. The bottom layer is represented by both inventory systems (i.e., inventory systems S1 and S2).
3. Both inventory systems should not be modified.
4. Your design should contain at least **four** layers. For each layer identify operations provided by the layer and its data structure(s).
5. Show call relationships between services of adjacent layers.
6. Each layer should be encapsulated in a class and represented by an object.
7. Provide a class diagram for the combined system. For each class list all operations supported by the class and major data structures. Briefly describe each operation in each class using **pseudo-code**.

PROBLEM #3 (35 points)

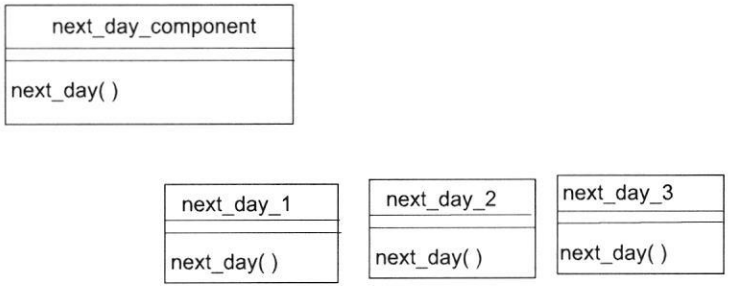
Suppose that we would like to use a **fault-tolerant architecture** for *next_day_component* that is supposed to compute the date of the next day. The *next_day* operation of this component accepts as an input three integer variables: *month*, *day*, and *year,* and returns the date of the day after the input date. An interface of the *next_day* operation is as follows:

void next_day (**in**: int month, int day, int year; **out**: int month1, int day1, int year1)

where parameters *month*, *day*, and *year* represent an input date, and parameters *month1*, *day1*, and *year1* represent an output date (the date of the day after the input date).

For example, for the input date: *month*=12, *day*=15, *year*=2022, the *next_day* operation returns *month1*=12, *day1*=16, *year1*=2022.

Suppose that three versions of the *next_day* component have been implemented using different algorithms. Different versions are represented by classes: *next_day_1*, *next_day_2,* and *next_day_3*.



Provide two designs for the *next_day component* using the following types of fault-tolerant software architectures:

1. N-Version architecture
2. Recovery-Block architecture

For each design provide:

1. A class diagram. For each class identify operations supported by the class and its attributes. Specify in detail each operation using **pseudo-code** (you do not need to specify operations *next_day()* of the *next_day_i* classes; only new operations need to be specified).

2. A sequence diagram representing a typical execution of the *next_day component*.

Background for the *next_day* component:

Since a year is 365.2422 days long, leap years are used for the "extra day" problem. If we declared a leap year every fourth year, there would be a slight error. The Gregorian Calendar resolves this by adjusting leap years to century years. Thus a year is a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400, so 1992, 1996, and 2000 are leap years, while the year 1900 is a common year.

check (2,28,1900, 2,29,1900)

Fail !!

Project Part #2.

Design & implement
GP components
based on Part #1

MDA-EFSM

if your MDA-EFSM
was correct, use it
for part #2
otherwise.

use the posted
MDA-EFSM !!!

In the design you need to incorporate 3 patterns

1. State Pattern → MDA-FSM

2. strategy pattern op + actions.

3. Abstract Factory pattern

to initialize a selected GP- component

PART #2: PROJECT DESIGN, IMPLEMENTATION, and REPORT

CS 586; Spring 2024

Final Project Deadline: **Thursday, April 25, 2024**
Late submissions: 50% off
After **April 30** the final project will not be accepted.

Submission: The project must be submitted on Blackboard. The hardcopy submissions will not be accepted.

This is an **individual** project, not a team project. Identical or similar submissions will be penalized.

DESIGN and IMPLEMENTATION

The goal of the second part of the project is to design two *Gas Pump (GP)* components using the Model-Driven Architecture (MDA) and then implement these *GP* components based on this design using the OO programming language. This OO-oriented design should be based on the MDA-EFSM (for both *GP* components) that was identified in the first part of the project. You may use your own MDA-EFSM (assuming that it was correct) or you can use the posted sample MDA-EFSM. In your design, you **MUST** use the following OO design patterns:

- state pattern
- strategy pattern
- abstract factory pattern

In the design, you need to provide the class diagram, in which the coupling between components should be minimized and the cohesion of components should be maximized (components with high cohesion and low coupling between components). In addition, a sequence diagram should be provided as described on the next page (Section 4 of the report).

After the design is completed, you need to implement the *GP* components based on your design using the OO programming language. In addition, the driver for the project to execute and test the correctness of the design and its implementation for the *GP* components must be implemented.

The Report and Deliverables

I: REPORT

The report **must** be submitted as one PDF-file (otherwise, a **10% penalty will be applied).**

1. MDA-EFSM model for the *GP* components
   a. A list of meta events for the MDA-EFSM
   b. A list of meta actions for the MDA-EFSM with their descriptions
   c. A state diagram of the MDA-EFSM
   d. Pseudo-code of all operations of Input Processors of Gas Pump: *GP-1* and *GP-2*

2. Class diagram(s) of the MDA of the *GP* components. In your design, you **MUST** use the following OO design patterns:
   a. State pattern
   b. Strategy pattern
   c. Abstract factory pattern

3. For each class in the class diagram(s), you should:
   a. Describe the purpose of the class, i.e., responsibilities.
   b. Describe the responsibility of each operation supported by each class.

4. Dynamics. Provide two sequence diagrams for two Scenarios:
   a. Scenario-I should show how one liter of gas is disposed in the Gas Pump *GP-1* component, i.e., the following sequence of operations is issued:
   Activate(4), Start(), PayCredit(), Approved(), StartPump(), Pump(), StopPump()
   b. Scenario-II should show how one gallon of Premium gas is disposed in the Gas Pump *GP-2* component, i.e., the following sequence of operations is issued:
   Activate(4.2, 7.2, 5.3), Start(), PayCash(10), Premium(), StartPump(), PumpGallon(), PumpGallon(), Receipt()

II: Well-documented (commented) source code

In the source-code you should indicate/highlight which parts of the source code are responsible for the implementation of the three required design patterns (**if this is not indicated in the source code, 20 points will be deducted**):

- state pattern
- strategy pattern
- abstract factory pattern.

The source-code must be submitted on the Blackboard. Note that the source code may be compiled during the grading and then executed. If the source-code is not provided, **15 POINTS** will be deducted.

III: Project executables

The project executable(s) of the *GP* components with detailed instructions explaining the execution of the program must be prepared and made available for grading. The project executable should be submitted on Blackboard. If the executable is not provided (or not easily available), **20 POINTS** will be automatically deducted from the project grade.

Strict Layered architecture

* components: layers
* relationship: call relationship.

---

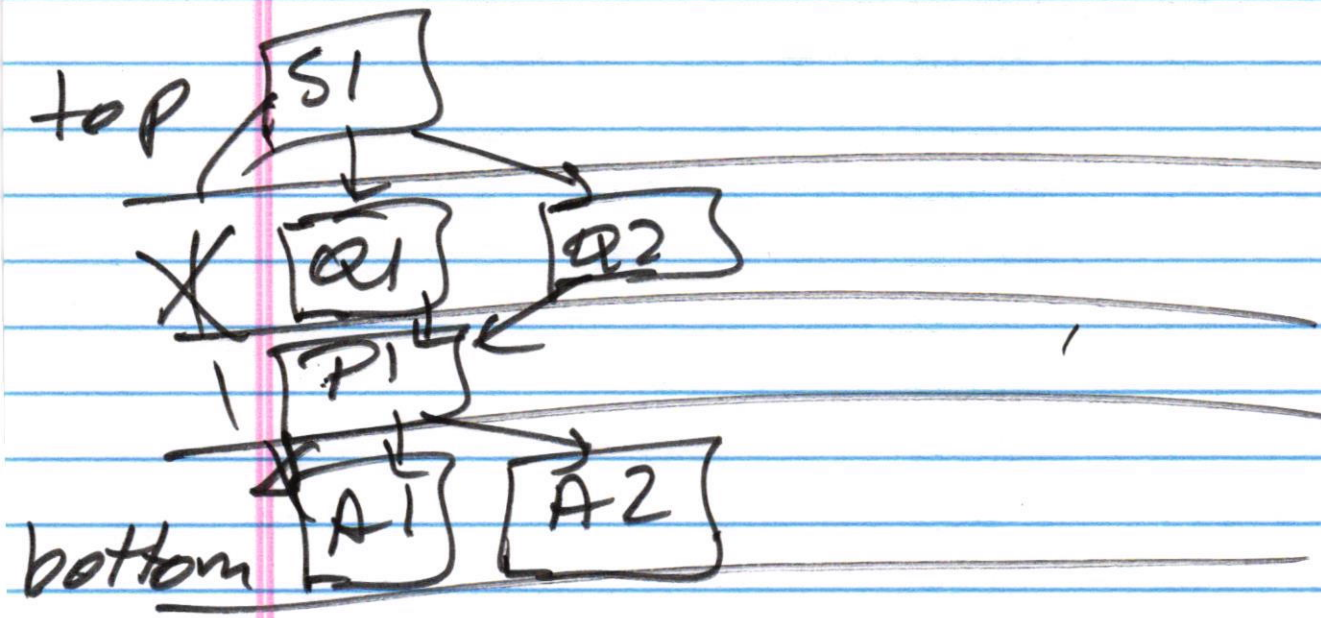
a set of layers.

each layer interacts with two layers.

(1) layer above: providing services.

(2) layer below: getting services.

strict layered architecture

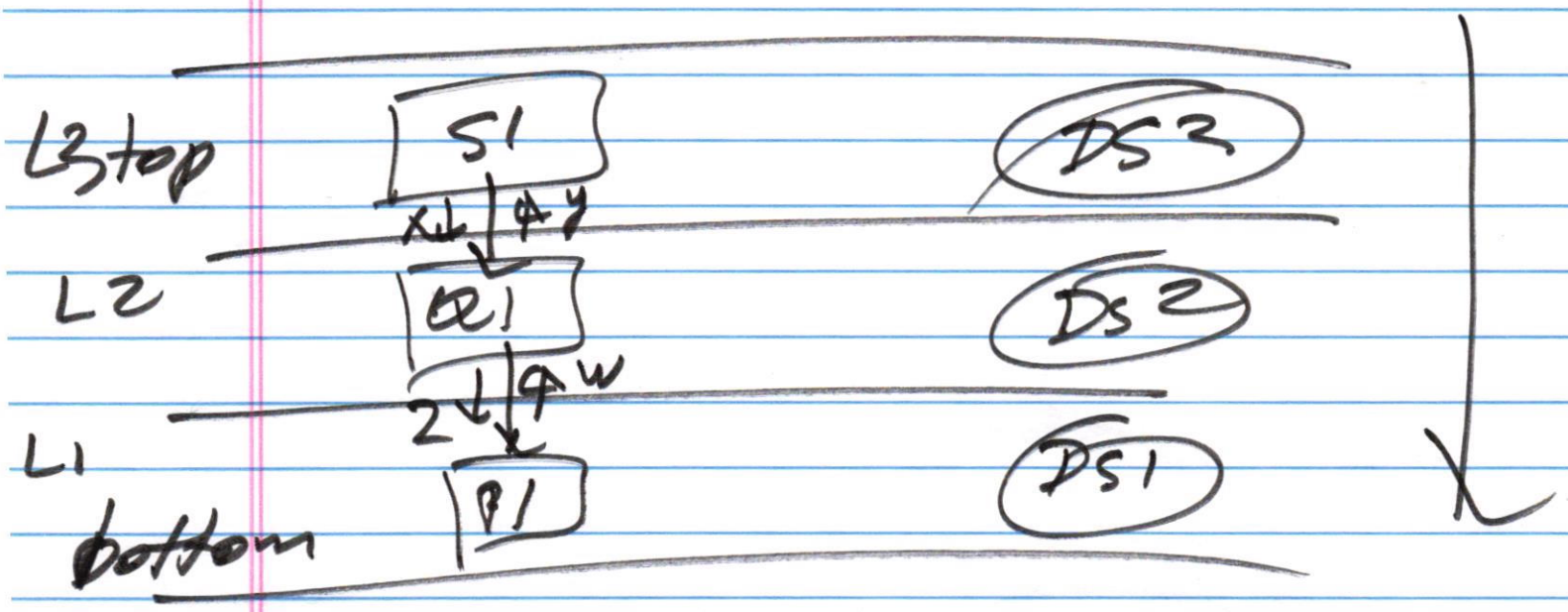


communication between layers

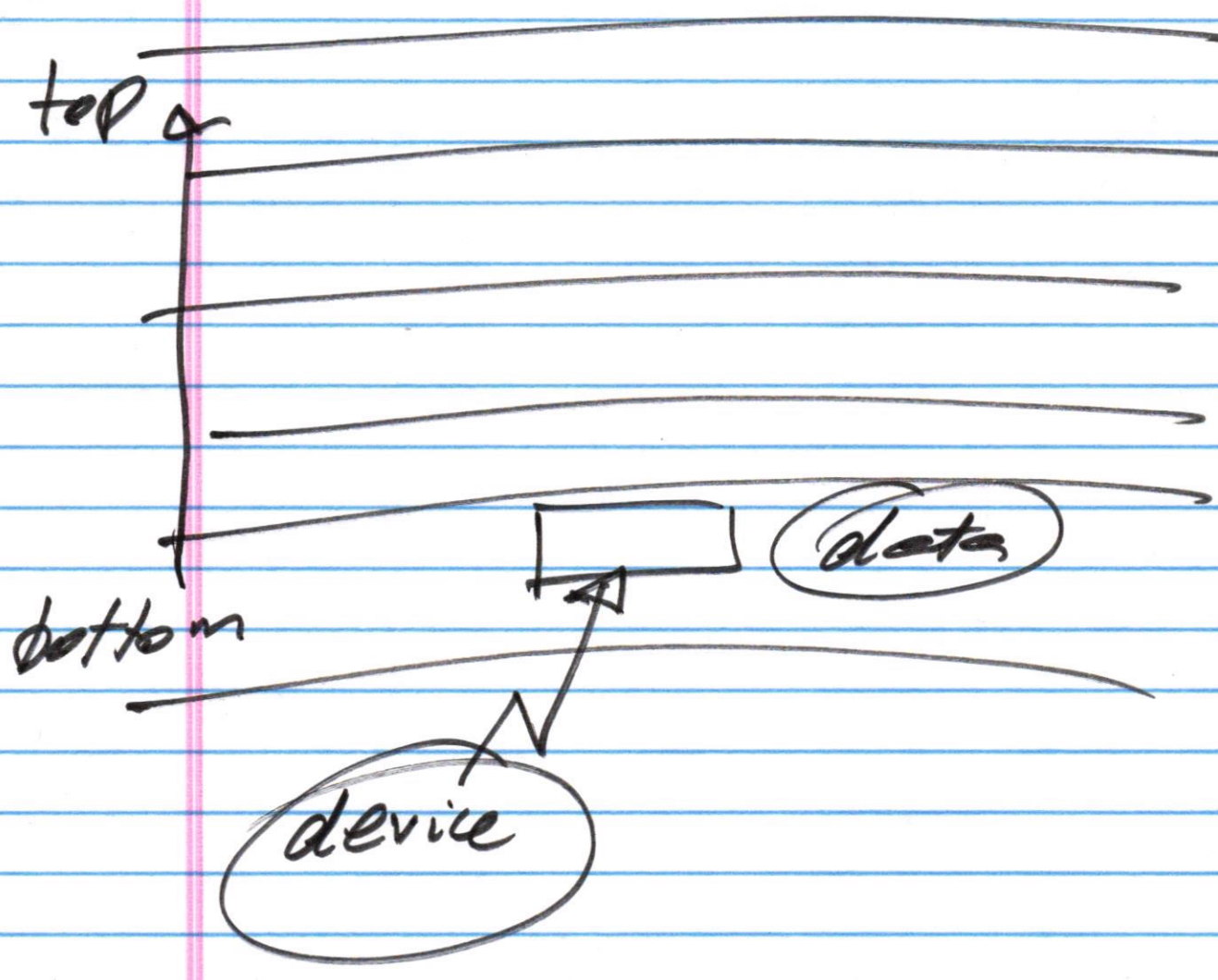
(1) top-down communication

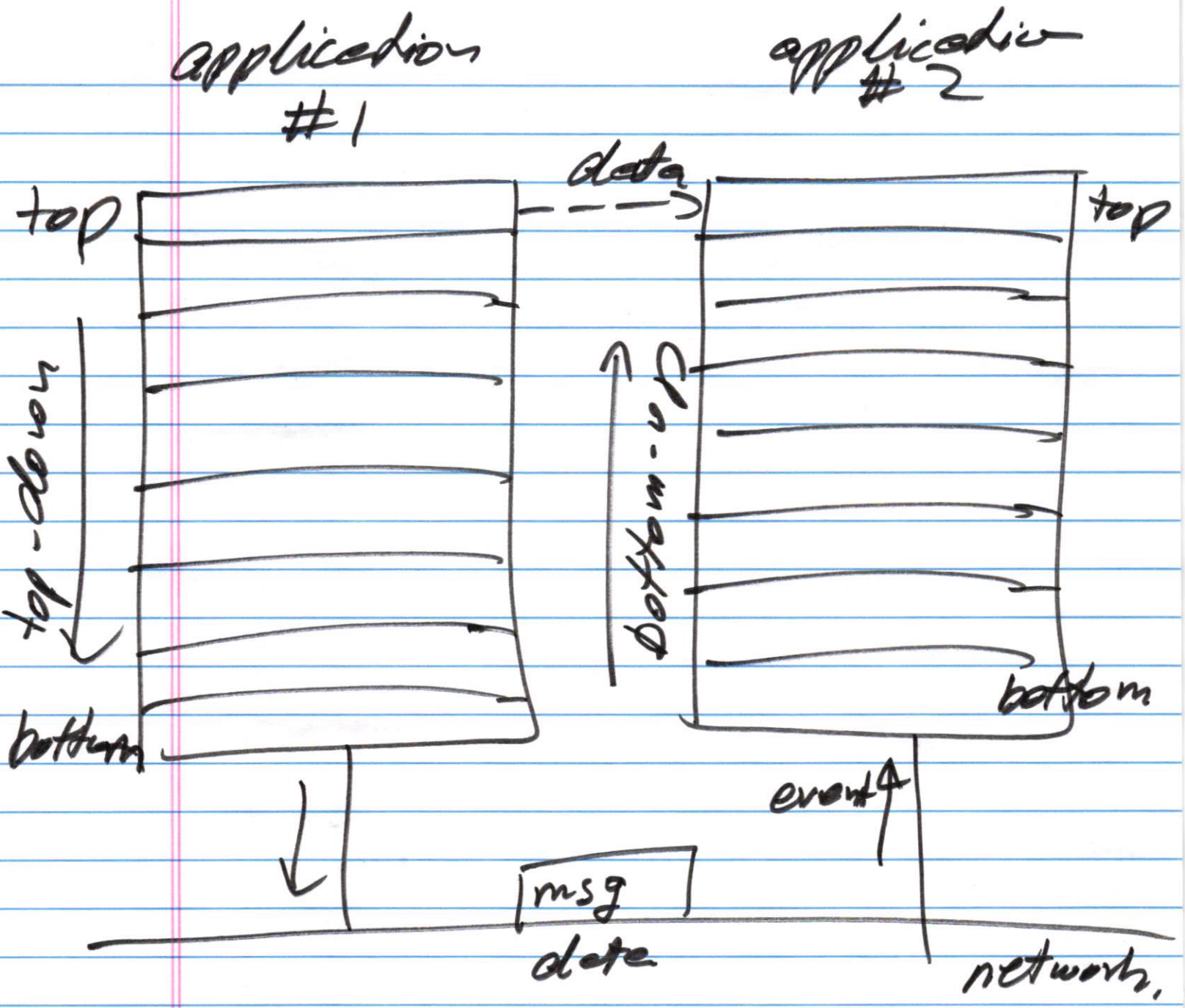
(2) bottom-up — ll —

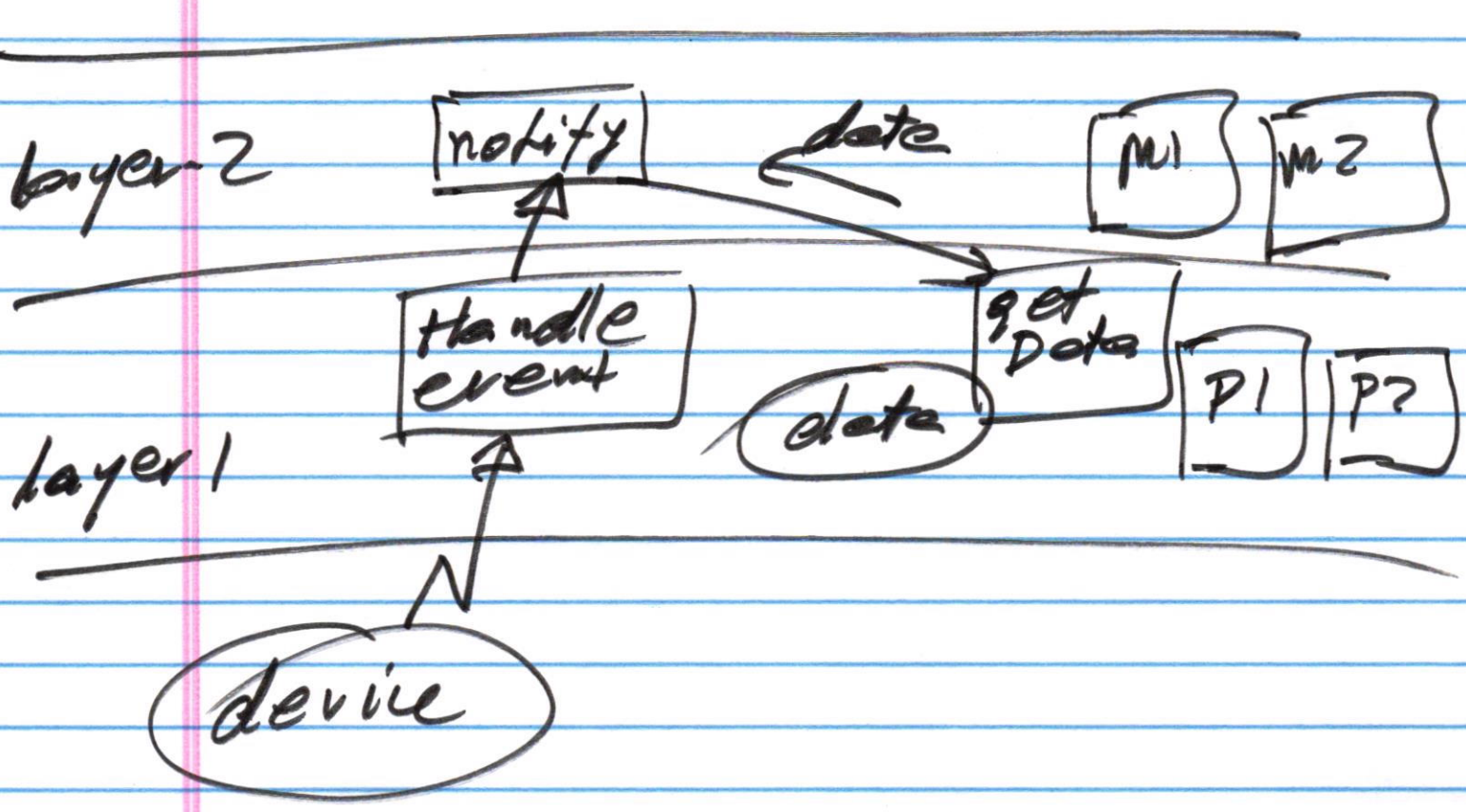
Top-down communication

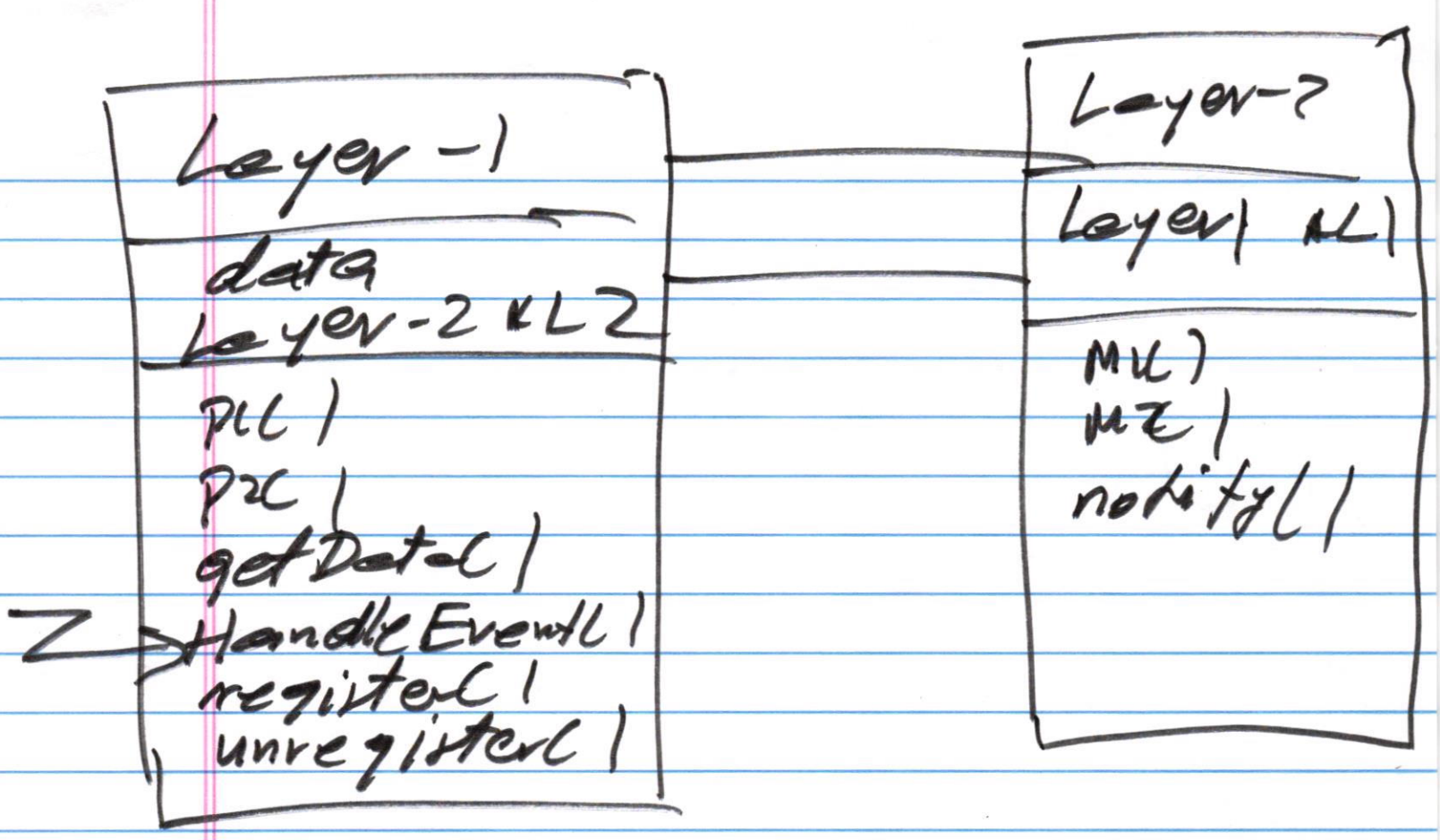


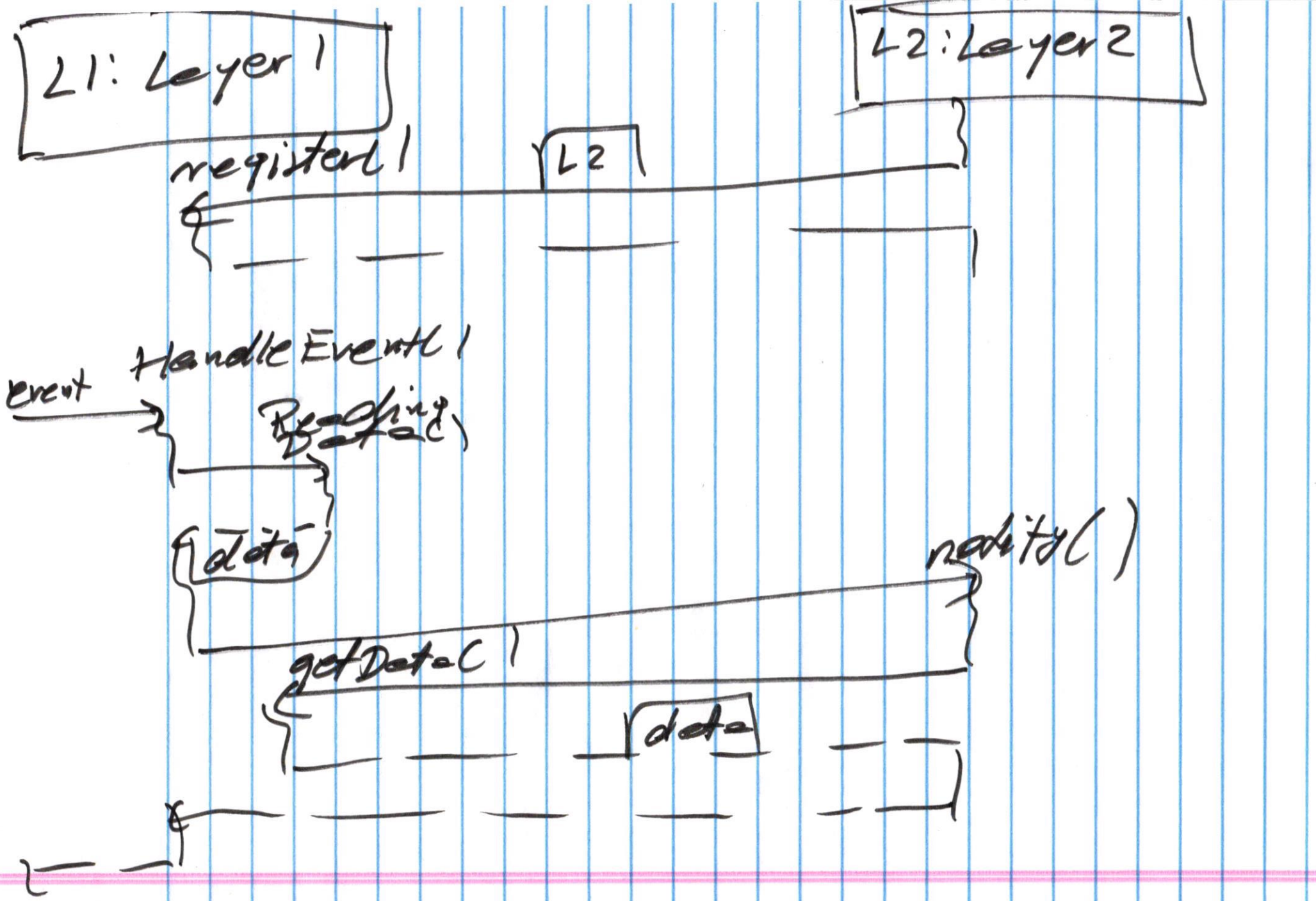
Bottom-up communication




application #1
application #2
data
top
top
move-down
move-up
bottom
bottom
event
msg
data
network


layer-2
notify
data
M1
M2
Handle event
get Data
data
P1
P2
layer 1
device


Layer-1
data
Layer-2 *L2
p1()
p2()
getData()
HandleEvent()
register()
unregister()
Layer-2
Layer1 *L1
M1()
M2()
notify()


L1: Layer 1
L2: Layer 2
register()
L2
HandleEvent()
event
BackingData()
data
notify()
getData()
data