1. Project Part #2 is posted.

A sample MDA-EFSM for Gas Pump components is posted

2. Homework #3 is posted

PART #2: PROJECT DESIGN, IMPLEMENTATION, and REPORT

CS 586; Fall 2025

Final Project Deadline: **Friday, December 5, 2025**
Late submissions: 50% off
After **December 9,** the final project will not be accepted.

Submission: The project must be submitted on Canvas. 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* components using the Model-Driven Architecture (MDA) and then implement these *Gas Pump* components based on this design using the OO programming language. This OO-oriented design should be based on the MDA-EFSM for both *Gas Pump* 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, two sequence diagrams should be provided as described on the next page (Section 4 of the report).

After the design is completed, you need to implement the *Gas Pump* 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 *Gas Pump* components must be implemented.

Outline of the Report & 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 *Gas Pump* 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 the Input Processors of Gas Pumps*: GP-1* and *GP-2*

2. Class diagram(s) of the MDA of the *Gas Pump* 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 sequence diagrams for two Scenarios:
    a. Scenario-I should show how one liter of gas is dispensed in *GasPump-1*, i.e., the following sequence of operations is issued: *Activate(4.1), Start(), PayCash(5.2), StartPump(), PumpLiter(), PumpLiter()*

    b. Scenario-II should show how one gallon of Regular gas is dispensed in *GasPump-2*, i.e., the following sequence of operations is issued: *Activate(4, 7), Start(), PayDebit(123), Pin(124), Pin(123), Regular(), StartPump(), PumpGallon(), FullTank()*

II: Well-documented (commented) source code

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

- state pattern
- strategy pattern
- abstract factory pattern.

The source code must be submitted on Canvas. 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 *Gas Pump* 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 Canvas. If the executable is not provided (or not easily available), **20 POINTS** will be automatically deducted from the project grade.

Project

Part #1: Develop MDA-EFSM for two GP-components

Part #2

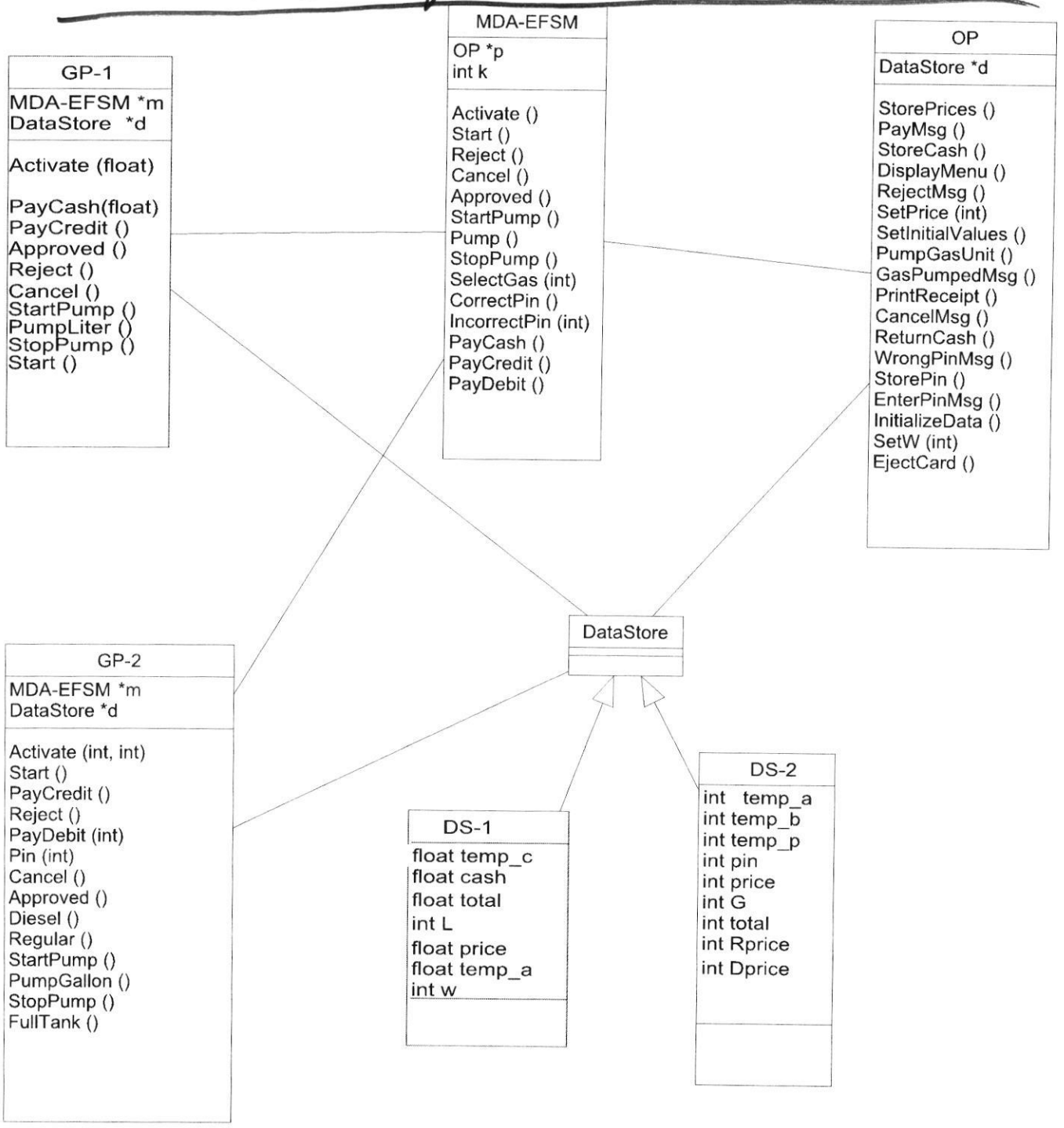
Based on MDA-EFSM from Part #1, design and implement GP components.

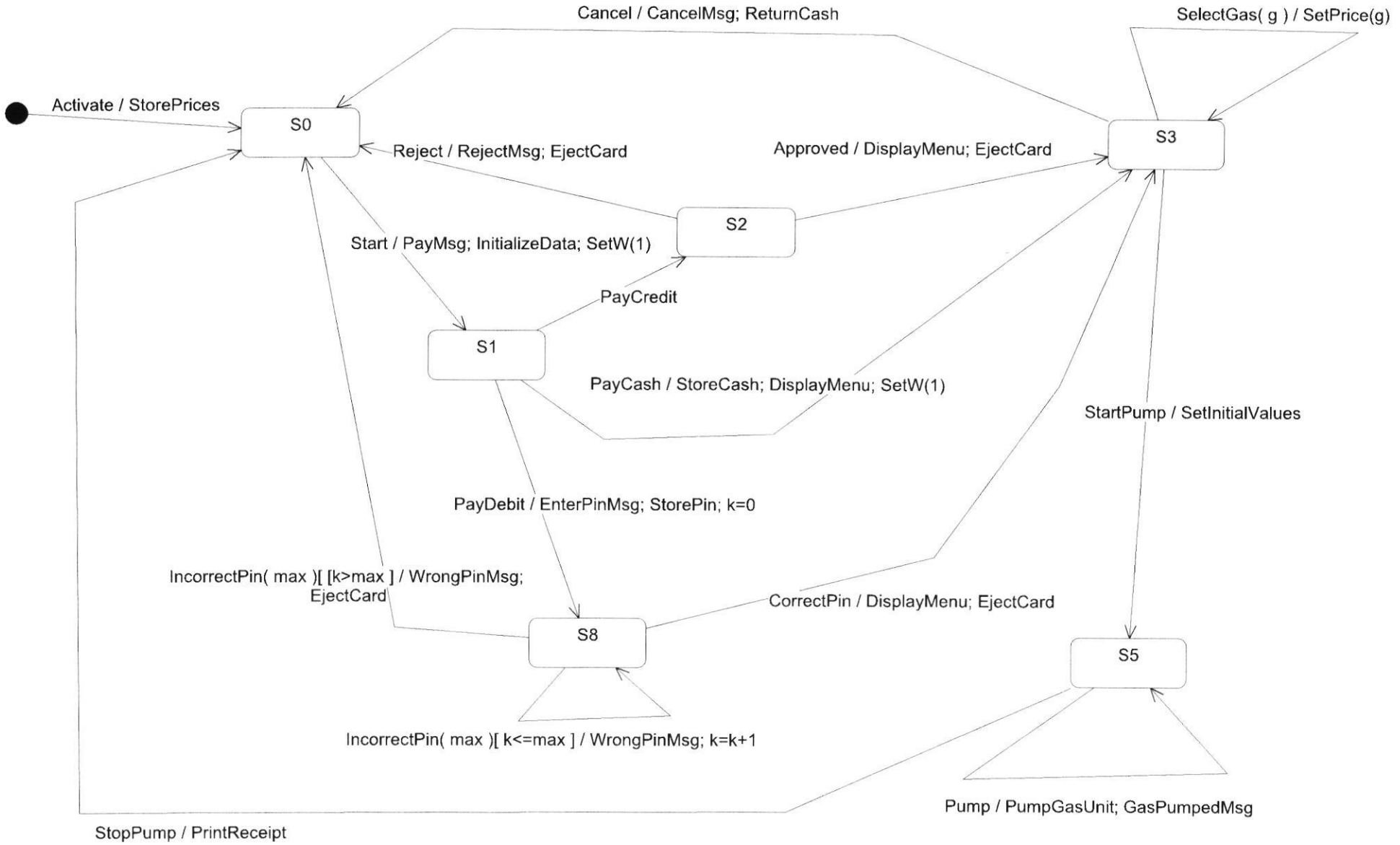
(a) If your MDA-EFSM is correct, then you can use it for part #2.

or

(b) if your MDA-EFSM is not correct, you should use the posted MDA-EFSM.

Posted sample MDA-EFSM.





MDA-EFSM for Gas Pumps

MDA-EFSM Events:

Activate()
Start()
PayCredit()
PayCash()
PayDebit()
Reject()
Cancel()
Approved()
StartPump()
Pump()
StopPump()
SelectGas(int g)
CorrectPin()
IncorrectPin(int max)

MDA-EFSM Actions:

StorePrices	// stores price(s) for the gas from the temporary data store
PayMsg	// displays a type of payment method
StoreCash	// stores cash from the temporary data store
DisplayMenu	// display a menu with a list of selections
RejectMsg	// displays credit card not approved message
SetPrice(int g)	// set the price for the gas identified by *g* identifier as in SelectGas(int g)
SetInitialValues	// set *G* (or *L*) and *total* to 0;
PumpGasUnit	// disposes unit of gas and counts # of units disposed
GasPumpedMsg	// displays the amount of disposed gas
PrintReceipt	// print a receipt
CancelMsg	// displays a cancellation message
ReturnCash	// returns the remaining cash
WrongPinMsg	// displays incorrect pin message
StorePin	// stores the pin from the temporary data store
EnterPinMsg	// displays a message to enter pin
InitializeData	// set the value of *price* to 0 for GP-2; do nothing for GP-1
EjectCard()	// card is ejected
SetW(int w)	// set value for cash flag

Operations of the Input Processor
(GasPump-1)

```
Activate(float a) {
        if (a>0) {
            d->temp_a=a;
            m->Activate()
        }
}

Start() {
        m->Start();
}
PayCash(float c) {
        if (c>0) {
            d->temp_c=c;
            m->PayCash()
        }
}

PayCredit() {
        m->PayCredit();
}

Reject() {
        m->Reject();
}

Approved() {
        m-> Approved();
}
Cancel() {
        m->Cancel();
}

StartPump() {
        m->StartPump();
}
```

```
PumpLiter() {
if (d->w==1) m->Pump()
else if (d->cash>0)&&(d->cash < d->price*(d->L+1))
            m->StopPump();
else m->Pump()
}

StopPump() {
        m->StopPump();
}
```

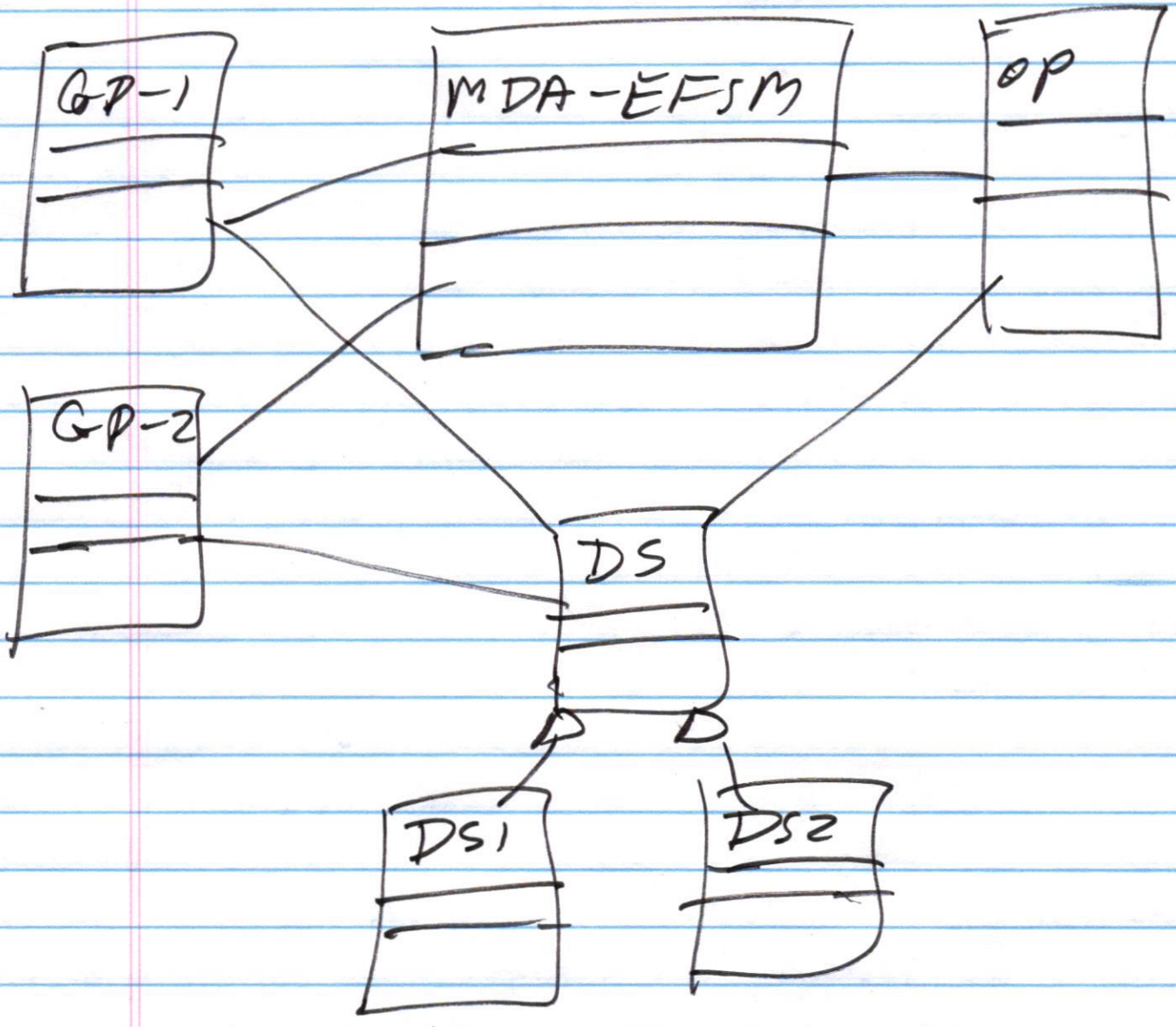
Notice:
cash: contains the value of cash deposited
price: contains the price of the selected gas
L: contains the number of liters already pumped
w: cash flag (cash: w=0; otherwise: w=1)
cash, *L*, *price, w* are in the data store
m: is a pointer to the MDA-EFSM object
d: is a pointer to the Data Store object

Operations of the Input Processor (GasPump-2)

```
Activate(int a, int b) {
        if ((a>0)&&(b>0)) {
            d->temp_a=a;
            d->temp_b=b;
            m->Activate()
        }
}

Start() {
        m->Start();
}

PayCredit() {
        m->PayCredit();
}

Reject() {
        m->Reject();
}

PayDebit(int p) {
        d->temp_p=p;
        m->PayDebit();
}

Pin(int x) {
        if (d->pin==x) m->CorrectPin()
        else m->InCorrectPin(1);
}

Cancel() {
        m->Cancel();
}

Approved() {
        m->Approved();
}

Diesel() {
        m->SelectGas(2)
}

Regular() {
        m->SelectGas(1)
}

StartPump() {
        if (d->price>0) m->StartPump();
}

PumpGallon() {
        m->Pump();

StopPump() {
        m->StopPump();
}

FullTank() {
        m->StopPump();
}
```

Notice:
pin: contains the pin in the data store
m: is a pointer to the MDA-EFSM object
d: is a pointer to the Data Store object
SelectGas(g): Regular: g=1; Diesel: g=2

Initial Design



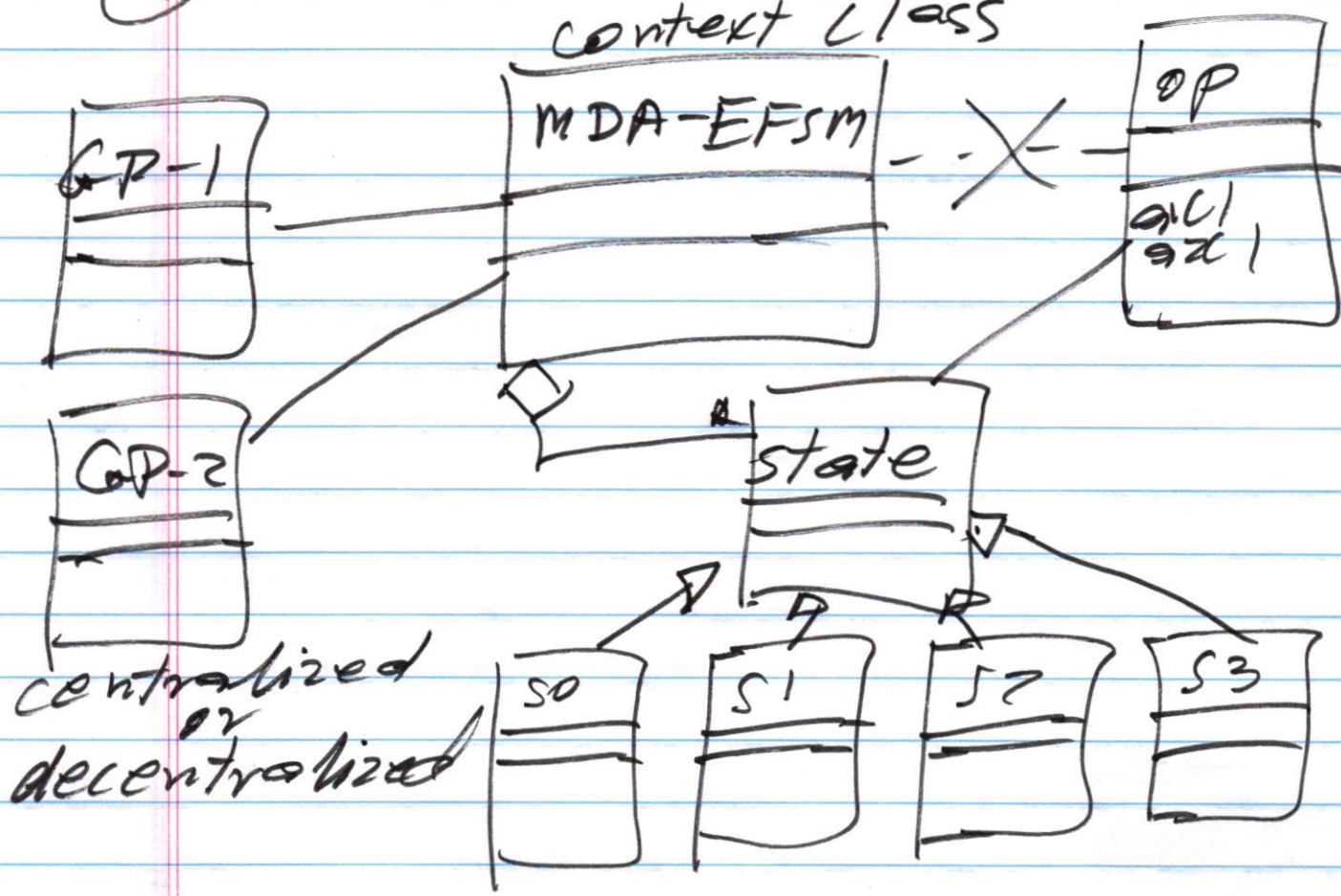
This design is NOT acceptable for part #2.

3 patterns MUST be incorporated.

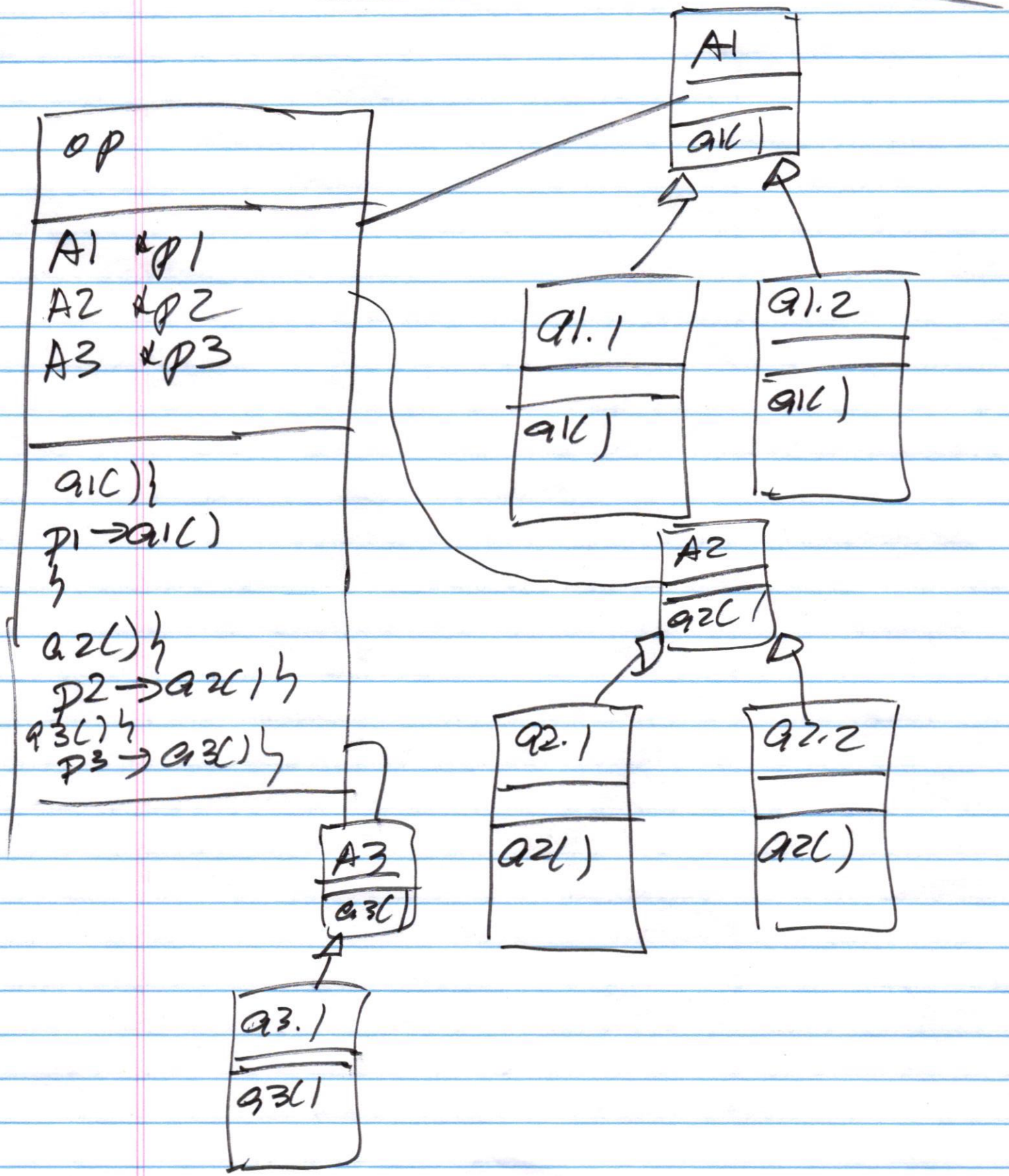
1. state pattern
2. strategy —"—
3. abstract-factory —"—

---

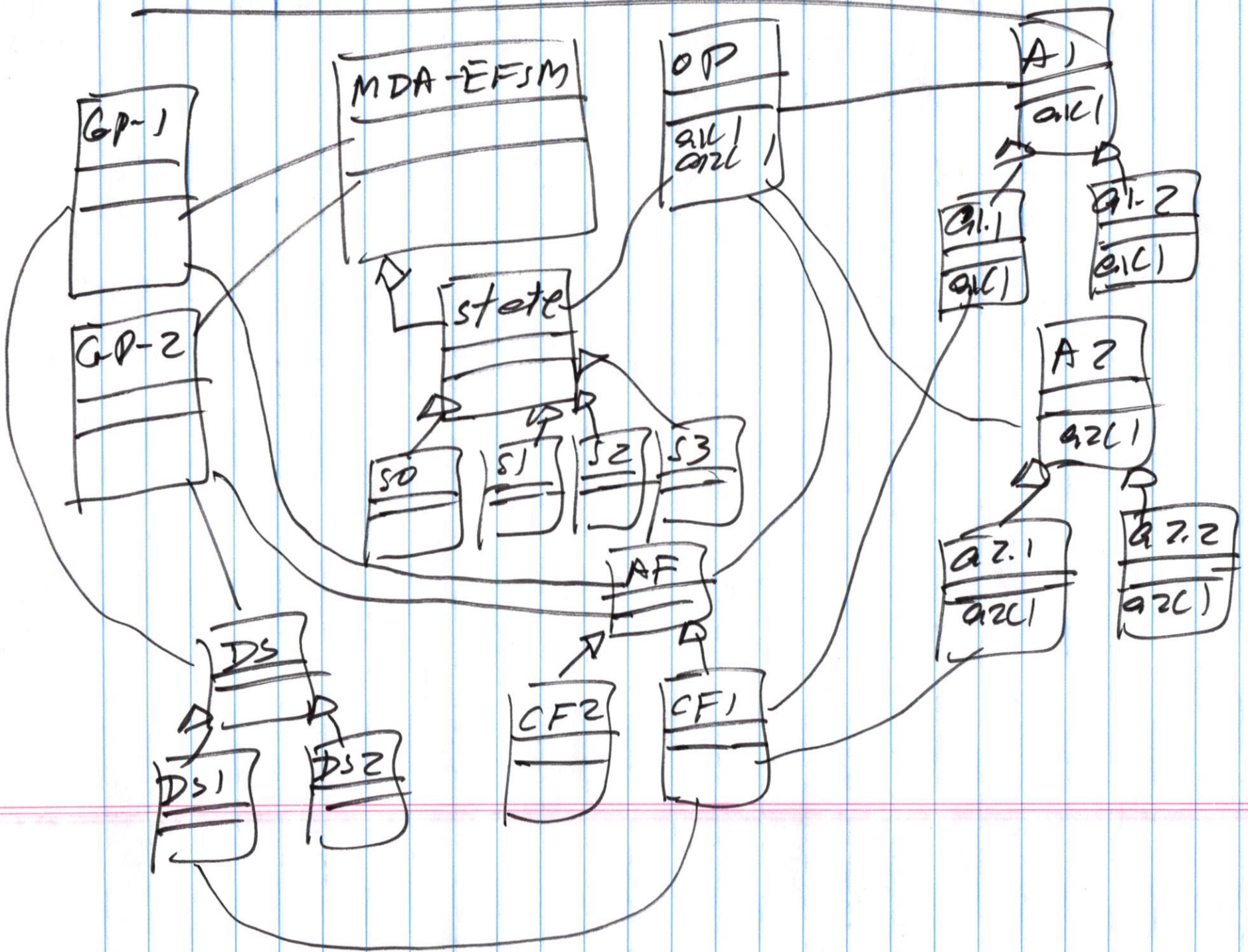
1. State Pattern



strategy pattern



3. Abstract factory pattern



of classes

GP : 2

OP : 1

Data store : 3

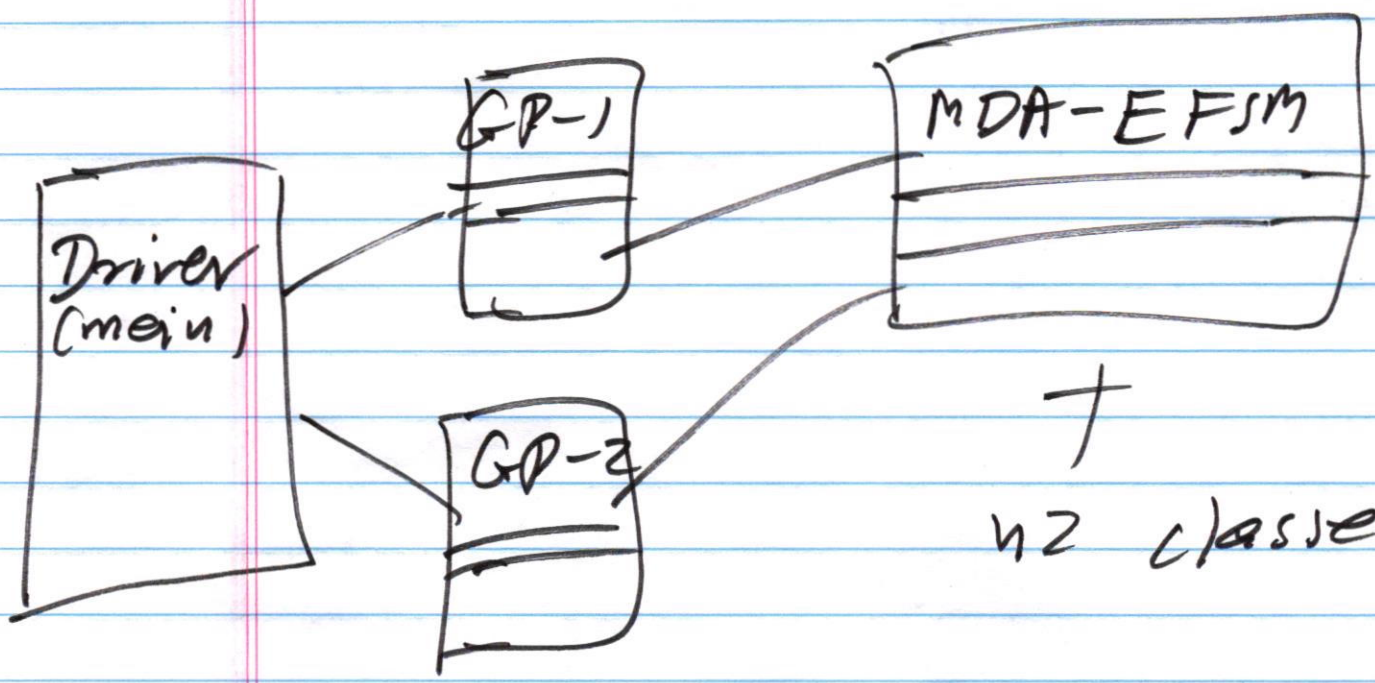
MDA-EFSM : 1 + 1 + 4 = 6 4 states.

Abstract factory : 3

Action classes : 3 · 10 = 30 10 actions

Total # of classes : 45 classes

Implementation of the design



Driver

1. select the GP [GP-1 or GP-2] for execution.
2. Execute the selected GP. We should be able to invoke any operation of selected GP (together with input ~~data~~) at any time.
3. to Test the correctness of the design

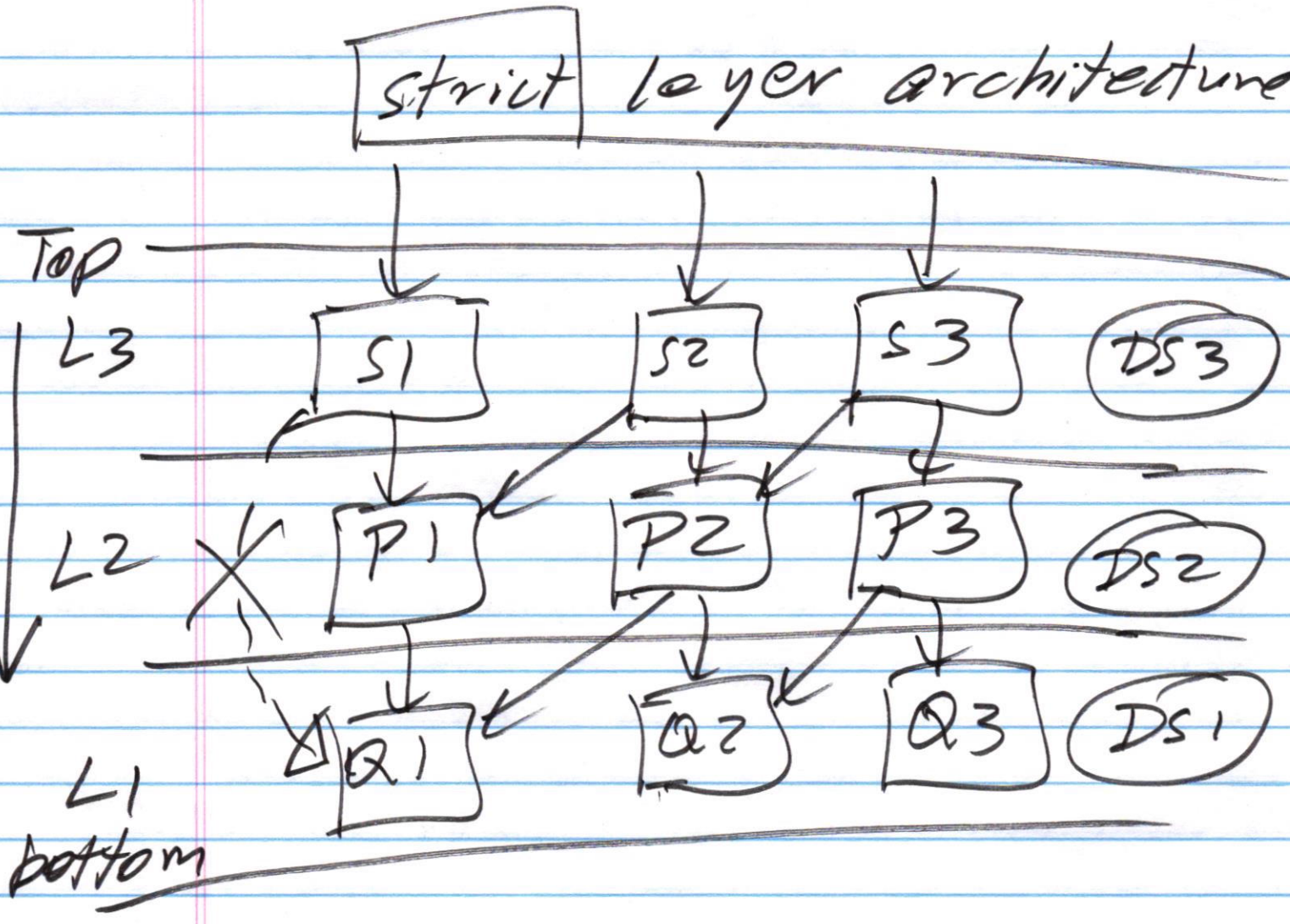
Layered architecture

a set of components
+
relationships between components

---

component: layer

relationship: call relationship.


strict layer architecture
Top
L3
S1
S2
S3
DS3
L2
P1
P2
P3
DS2
L1
Q1
Q2
Q3
DS1
bottom

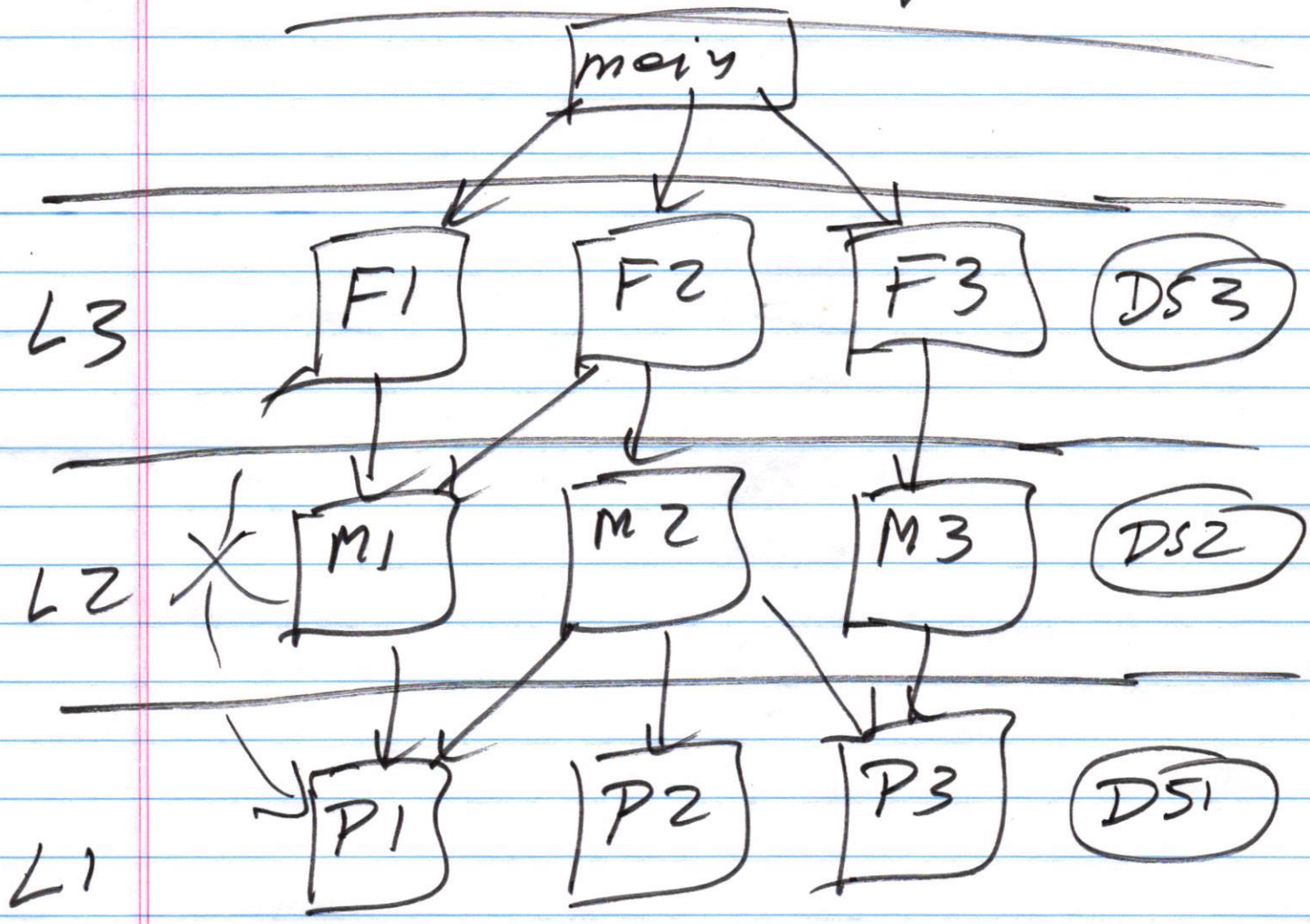
"strict" layered architecture.

* a set of layers
* each layer provides services to the layer above
* supports information hiding.
* each layer interacts with at most two layers.
  * layer above
  * layer below

Designing layers

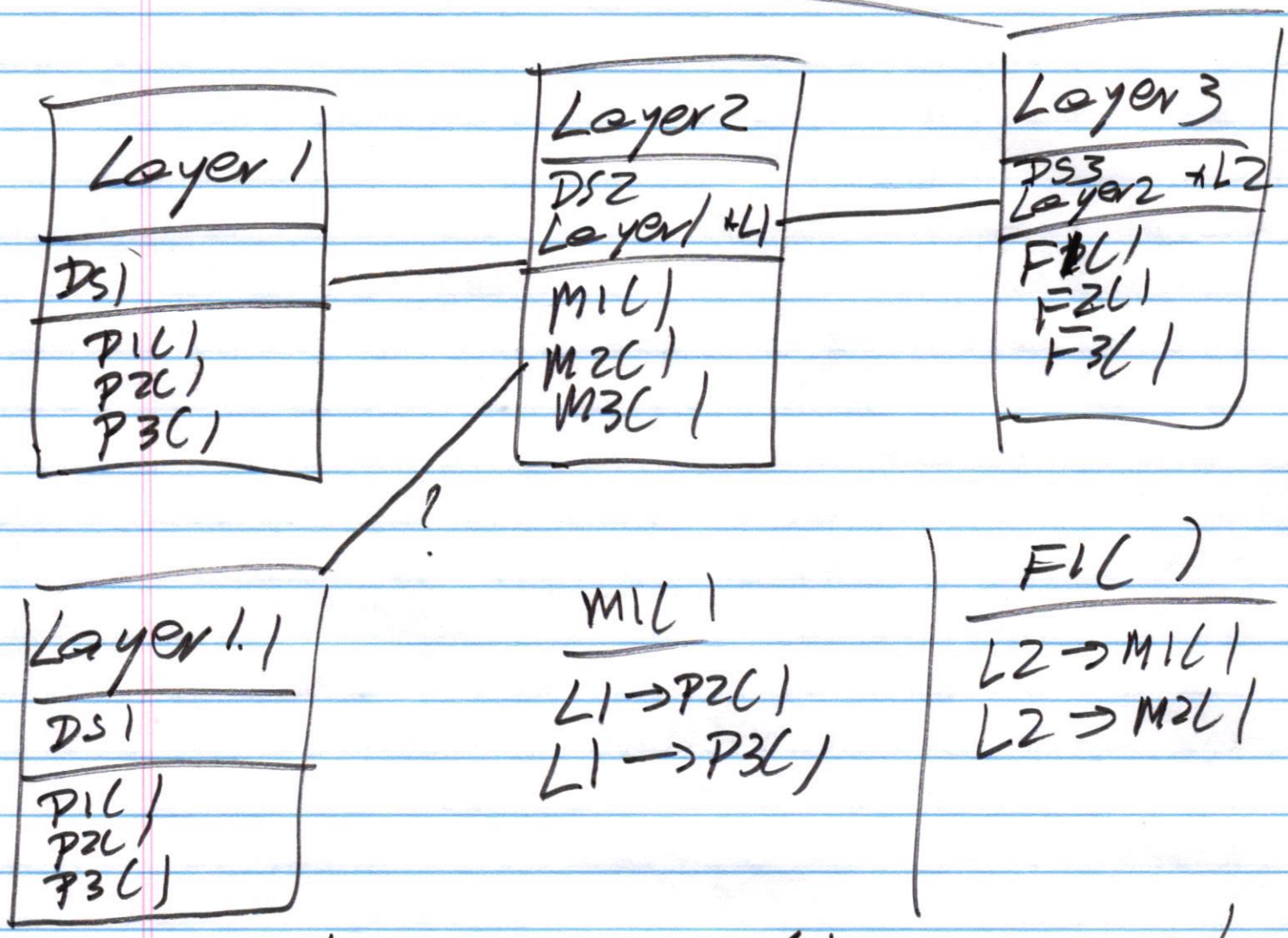
Top layer 3	F1(1), F2(1) F3(1)	DS3
layer 2	M1(1), M2(1), M3(1)	DS2
layer 1 bottom	P1(1), P2(1), P3(1)	DS1

Modular Design



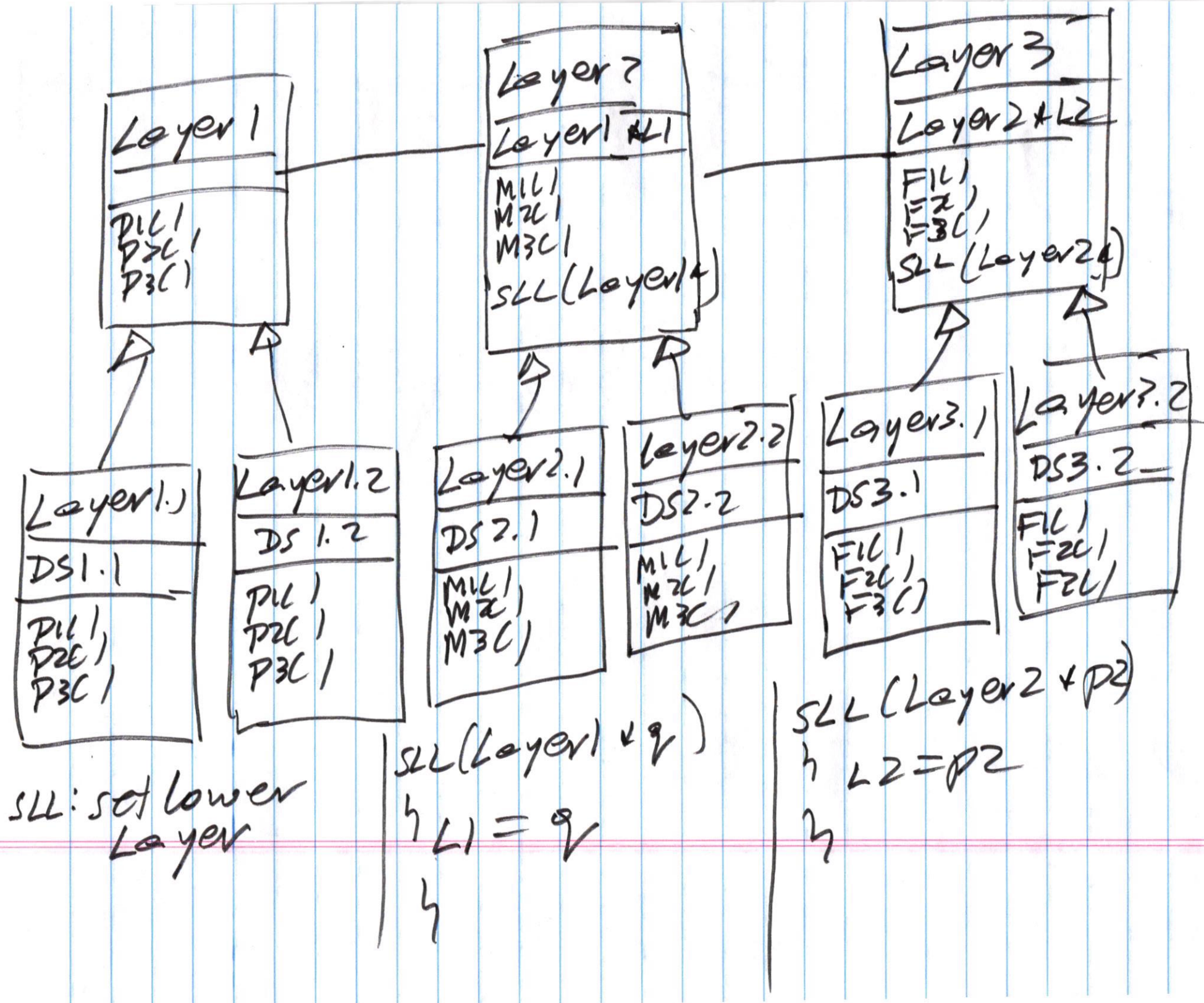
Not a good idea!

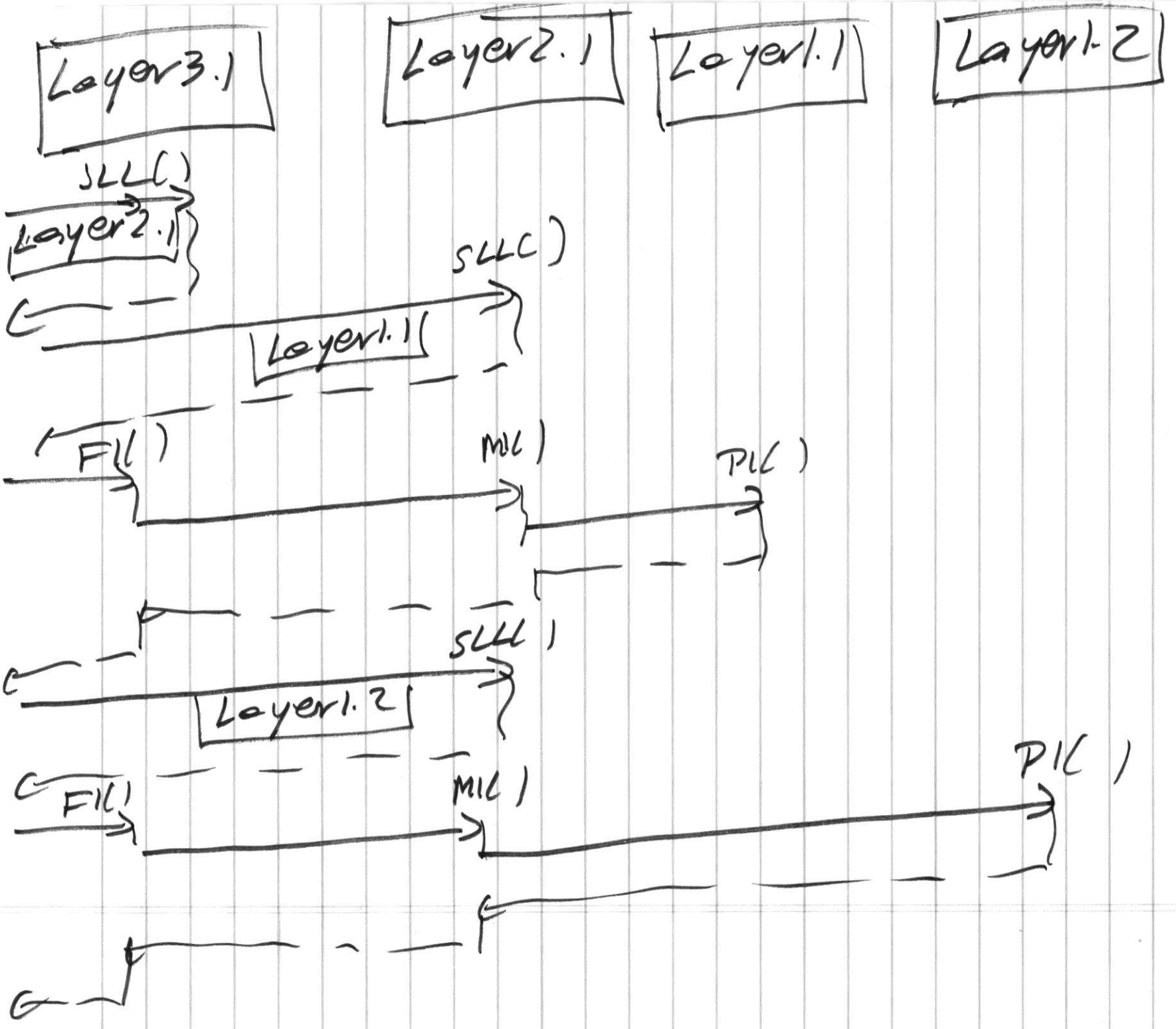
OO design



strong coupling between layers !!

???


Layer1
P1L()
P2L()
P3L()
Layer2
Layer1 *L1
M1L()
M2L()
M3L()
SLL(Layer1*)
Layer3
Layer2*L2
F1L()
F2L()
F3L()
SLL(Layer2*)
Layer1.1
DS1.1
P1L()
P2L()
P3L()
Layer1.2
DS1.2
P1L()
P2L()
P3L()
Layer2.1
DS2.1
M1L()
M2L()
M3L()
Layer2.2
DS2.2
M1L()
M2L()
M3L()
Layer3.1
DS3.1
F1L()
F2L()
F3L()
Layer3.2
DS3.2
F1L()
F2L()
F2L()
SLL: set lower Layer
SLL(Layer1*q)
{ L1 = q
}
SLL(Layer2*p2)
{ L2=p2
}


Layer3.1
Layer2.1
Layer1.1
Layer1.2
SLLC)
Layer2.1
SLLC)
Layer1.1
FIC)
MIC)
PIC)
SLLC)
Layer1.2
FIC)
MIC)
PIC)

Strict Layer Architecture

Adv

* proven approach of reducing complexity.
* supports reuse.
  * different versions of layers can be easily exchanged
* supports independence of layers

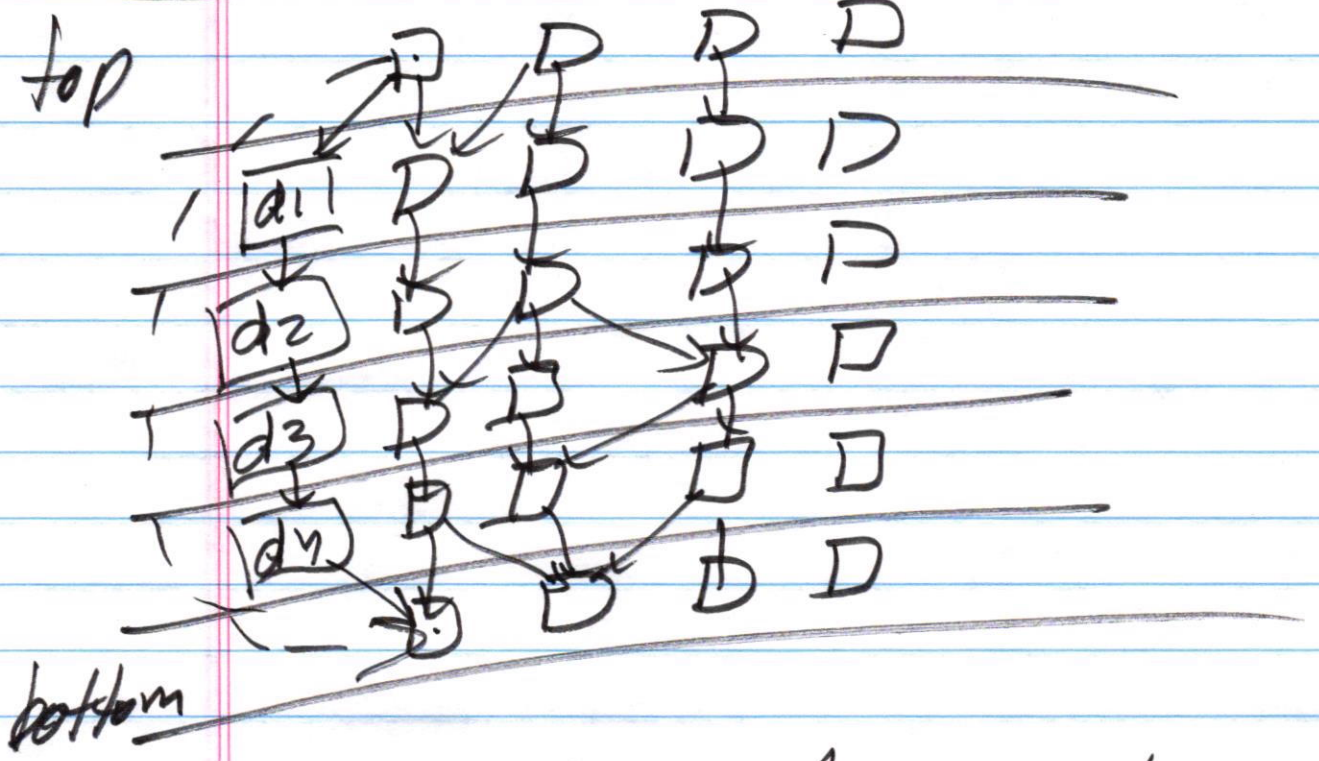
Disadvantages

* Not all systems can be structured in layered fashion.

2. it may be hard to allocate responsibilities to layers.

3. performance problems

"strict" layered architecture



Can this be relaxed?

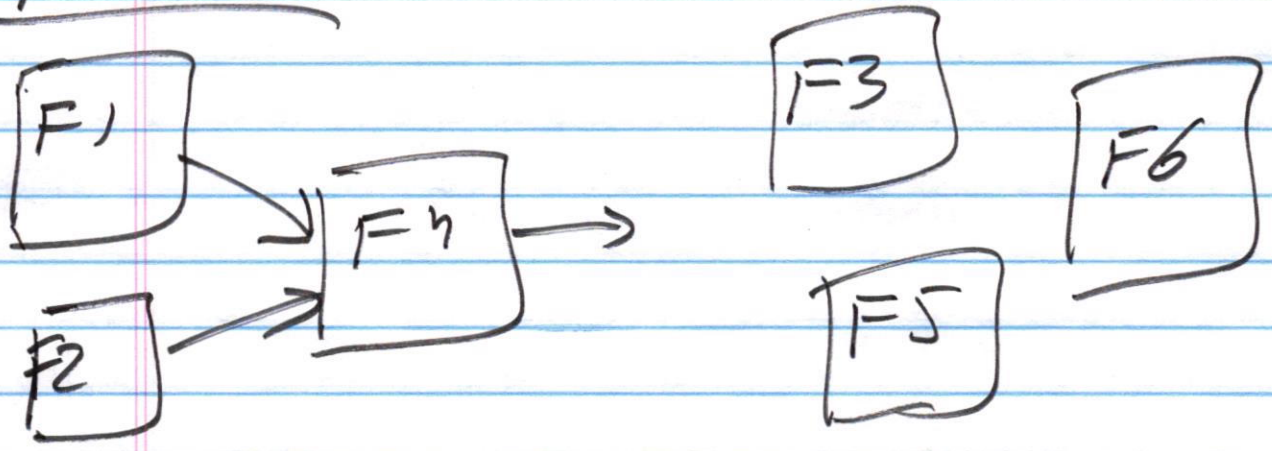
"relaxed" layered architecture!!

Homework #3

Problem #1

Pipes and Filters architecture.

Part A

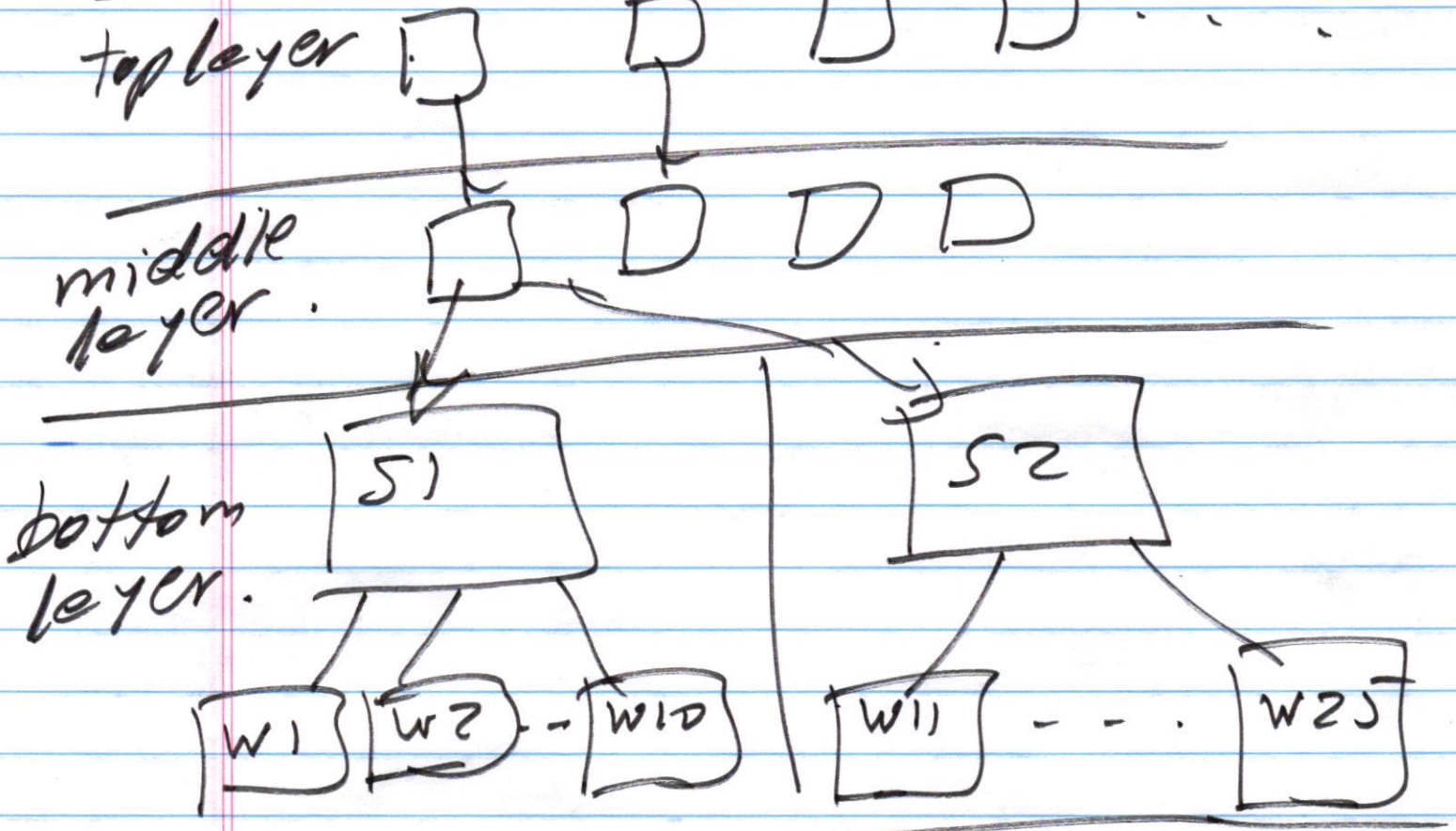


Part B

class diagram.

Problem #2

strict layered architecture



class diagram

HOMEWORK ASSIGNMENT #3

CS 586; Fall 2025
Due Date: **November 24, 2025**
Late homework 50% off
After **November 28,** 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 (35 points)
Consider the problem of designing a system using the **Pipes and Filters** architecture. The system should provide the following functionality:

- Read the student's test answers together with the student's IDs.
- Read students' names together with their IDs.
- Read the correct answers for the test.
- Compute test scores.
- Report test scores in an **ascending** order with respect to scores with student names

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 students' test answers together with students' IDs.
Filter #2:	This filter reads the correct answers for the test.
Filter #3:	This filter reads students' 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 in which 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 pull-out pipes
   c. Filter #3: passive filter with un-buffered push pipes
   d. Filter #4: passive filter with un-buffered pull-out pipes
   e. Filter #5: passive filter with un-buffered push 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 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
```

Services supported by **Server-S2**:

```
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
```

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. Neither inventory system should be modified.
4. Your design should contain at least **three** 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 the *RemoveDuplicates* component that removes duplicates from a list of integers within a *low-high* range. The *unique()* operation of this component accepts as input integer parameters *n*, *low*, *high*, and an integer array *L*. The component removes duplicates whose values are greater than or equal to *low* but smaller than or equal to *high*. The output parameters are (1) an integer array *SL* that contains the list of integers from list *L* without duplicates within the *low-high* range, and (2) an integer *m* that contains the number of elements in list *SL*. An interface of the *unique()* operation is as follows:

void unique (**in** int n, int low, int high, int L[]; **out** int SL[], int m)

L is an array of integers,
n is the number of elements in list *L*,
low is the lower bound for removing duplicates,
high is the upper bound for removing duplicates,
SL is an array of unique integers from list *L*,
m is the number of unique elements in list *SL*
Notice: *L* and *n* are inputs to the *unique()* operation. *SL* and *m* are output parameters of the *unique()* operation.

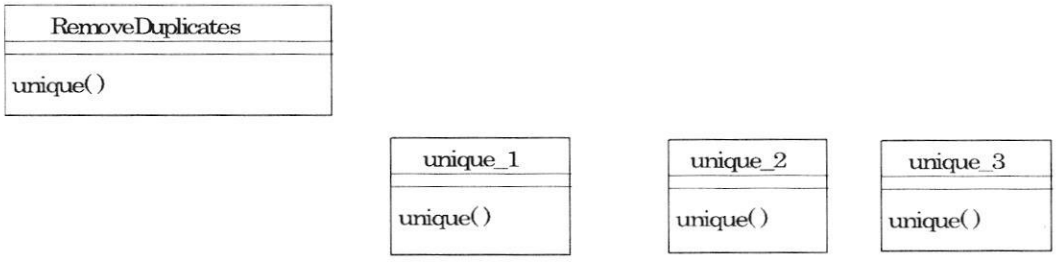
For example, for the following input:

n=8, *low*=2, *high*=6, *L*=(1, 7, 1, 2, 5, 2, 7, 5)

The *unique*() operation returns the following output parameters:

m=6, *SL*=(1, 7, 1, 2, 5, 7)

Suppose that three versions of the *RemoveDuplicates* component have been implemented using different algorithms. Different versions are represented by classes: *unique_1*, *unique_2, and unique_3*.



Provide two designs for the *RemoveDuplicates* 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 *unique()* of the *unique_i* classes; only new operations need to be specified).
2. A sequence diagram representing a typical execution of the *RemoveDuplicates* component.