Project Part #2 is posted

Exam #3

Monday, April 29

5:00 – 7:00 pm

closed books and notes.

The coverage is posted

This is a comprehensive exam

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

Project part #2

Model Driven Architecture to design and implement two GP components.

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.

no pseudocode

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.

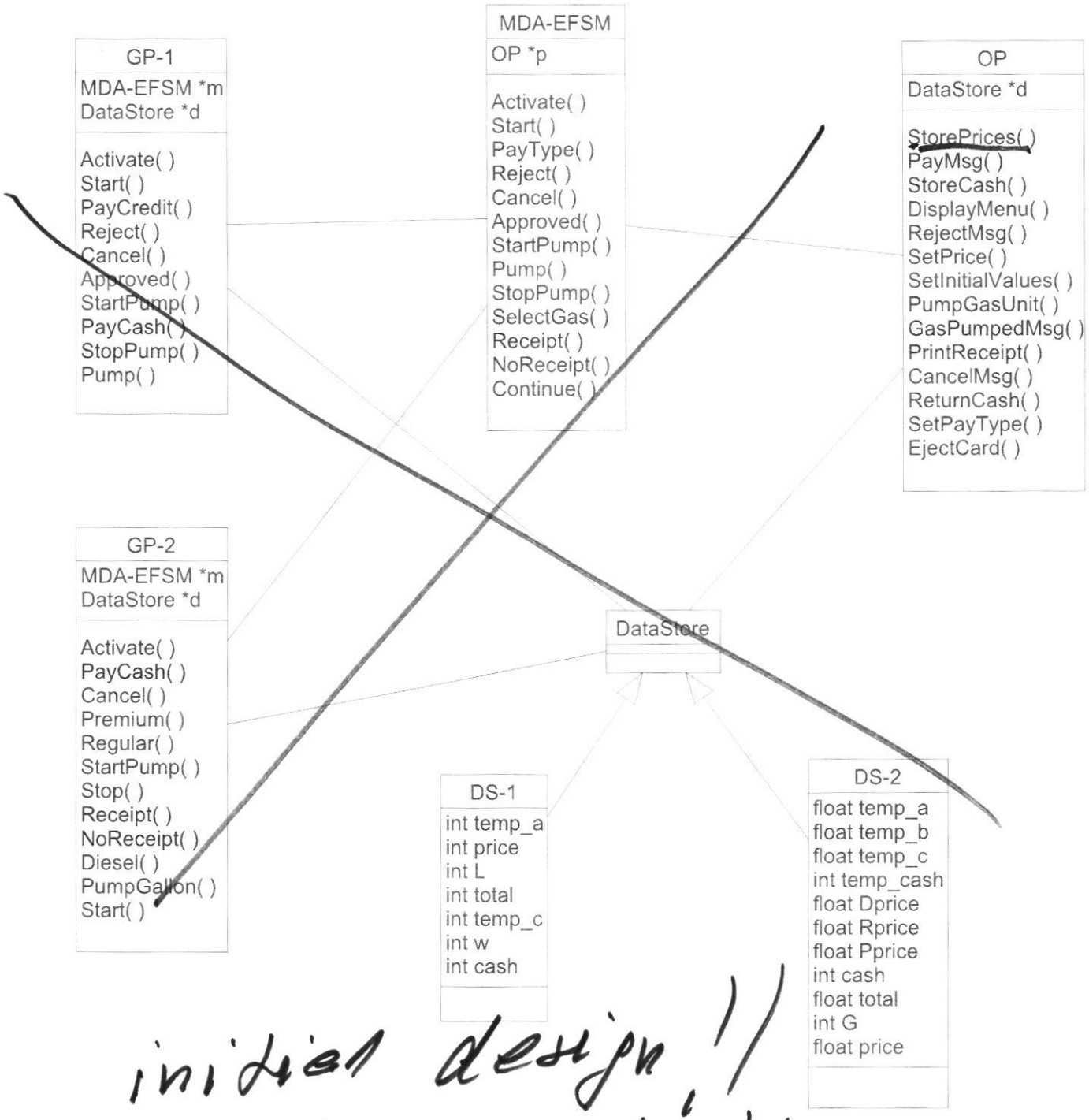
Part #1

MDA-EFSM

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

You may use the posted MDA-EFSM for part #2.

if your MDA-EFSM from part #1 was not correct



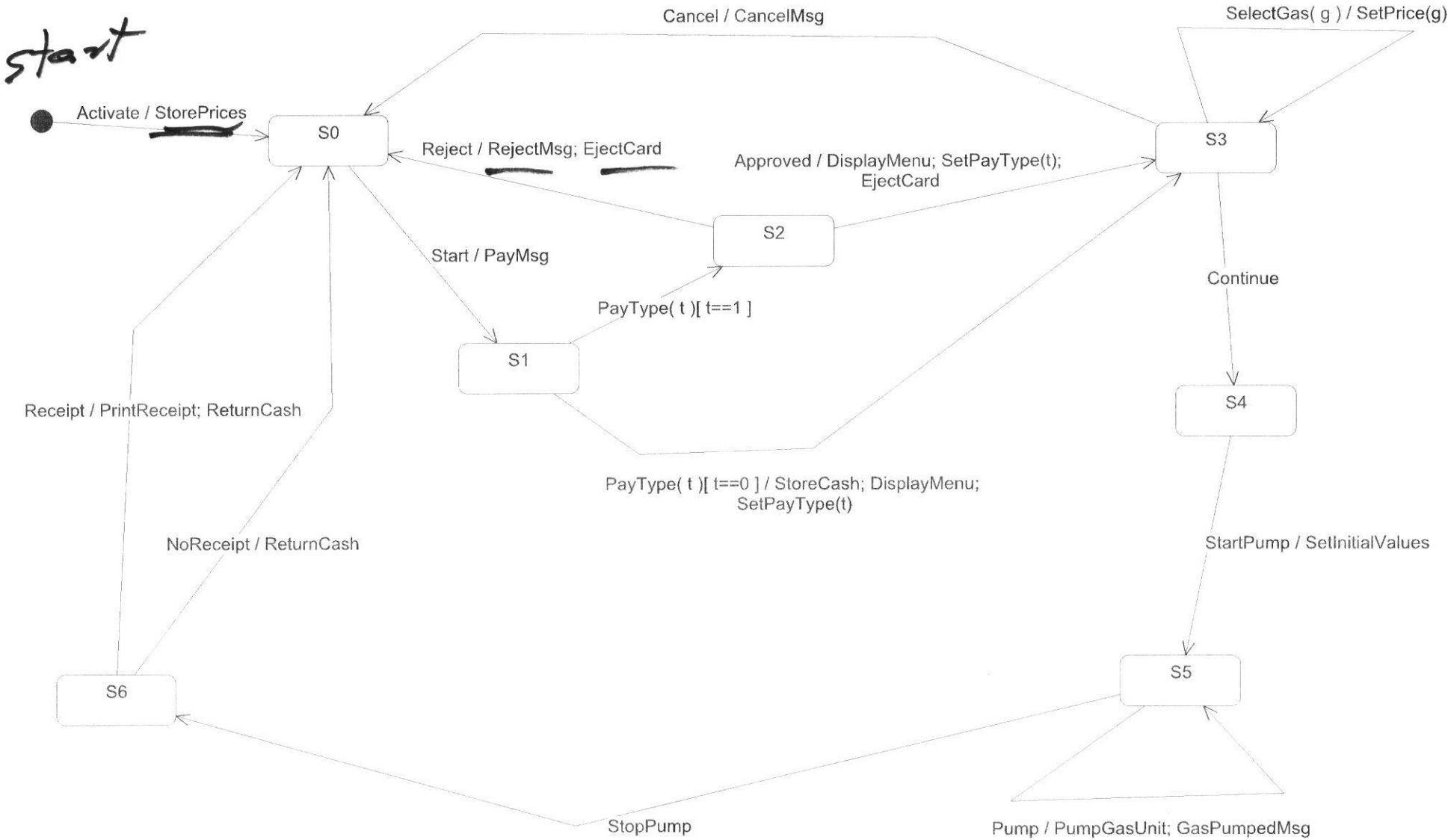
initial design // is not acceptable for part #2 because 3 patterns are missing.

MDA-EFSM Events:

Activate()
Start()
PayType(int t) //credit: t=1; cash: t=0;
Reject()
Cancel()
Approved()
StartPump()
Pump()
StopPump()
SelectGas(int g) // Regular: g=1; Diesel: g=2; Premium: g=3
Receipt()
NoReceipt()
Continue()

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 and computes Total
GasPumpedMsg() // displays the amount of disposed gas
PrintReceipt() // print a receipt
CancelMsg() // displays a cancellation message
ReturnCash() // returns the remaining cash
SetPayType(t) // Stores pay type *t* to variable *w* in the data store
EjectCard() // Card is ejected



MDA-EFSM for Gas Pumps

Operations of the Input Processor (GasPump-1)

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

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

PayCredit() {
        m->PayType(1);
}

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

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

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

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

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

Notice:
cash: contains the value of cash deposited
price: contains the price of the gas
L: contains the number of liters already pumped
w: pay type flag (cash: w=0; credit: 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(float a, float b, float c) {
        if ((a>0)&&(b>0)&&(c>0)) {
            d->temp_a=a;
            d->temp_b=b;
            d->temp_c=c
            m->Activate()
        }
}

PayCash(int c) {
        if (c>0) {
            d->temp_cash=c;
            m->PayType(0)
        }
}


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

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

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

Premium() {
        m->SelectGas(3);
        m->Continue();
}

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

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

PumpGallon() {
if (d->cash < d->price*(d->G+1))
            m->StopPump();
else m->Pump()
}

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

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

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

Notice:
cash: contains the value of cash deposited
price: contains the price of the selected gas
G: contains the number of Gallons already pumped

cash, *G*, *price* are in the data store
m: is a pointer to the MDA-EFSM object
d: is a pointer to the Data Store object

Part #2

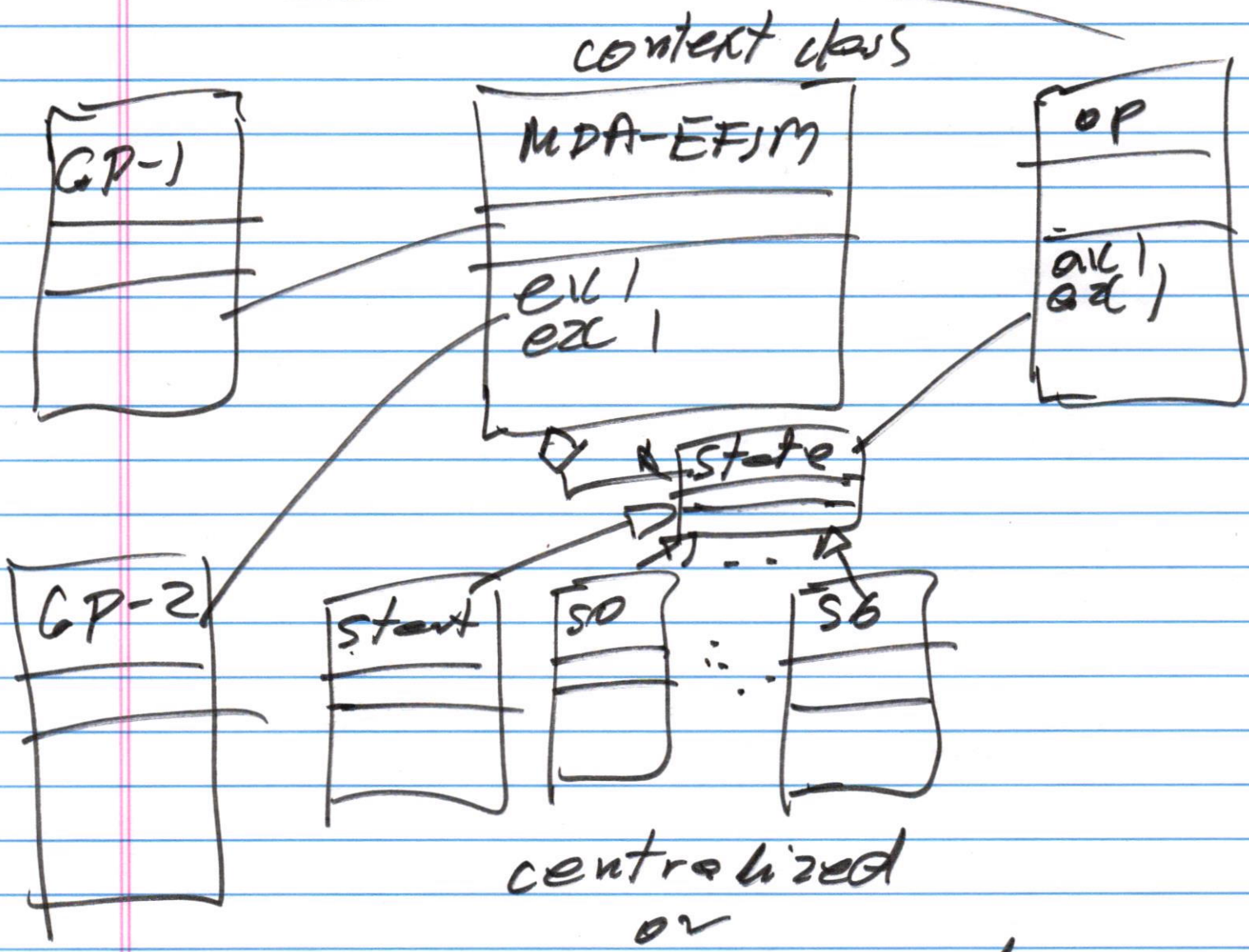
(1) Design GP components using MDA-EFSM from part #1

you must incorporate into your design 3 patterns.

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

2. Implementation of your design in OO prog. language in order to execute it and test it!!

State Pattern

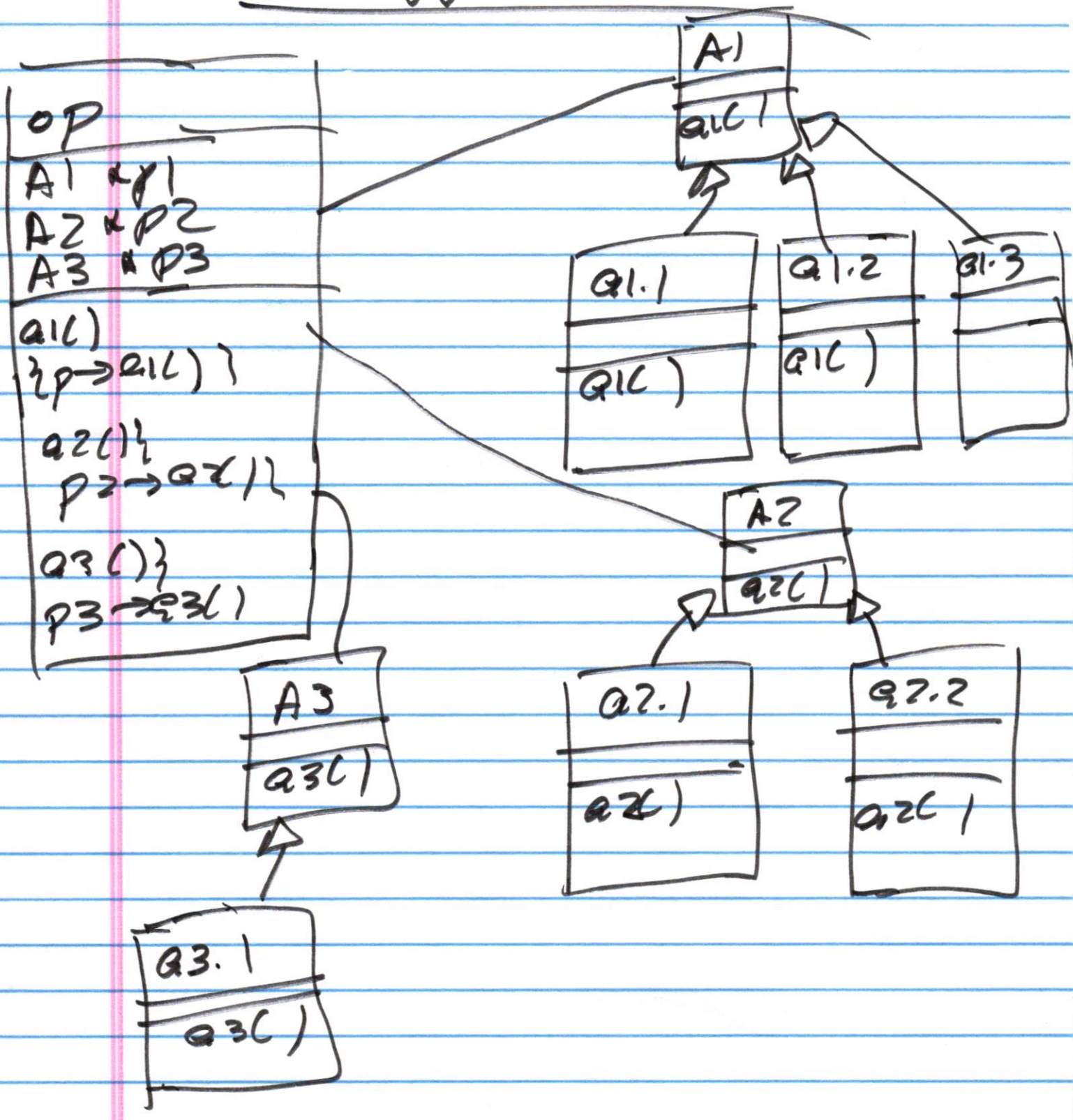


centralized

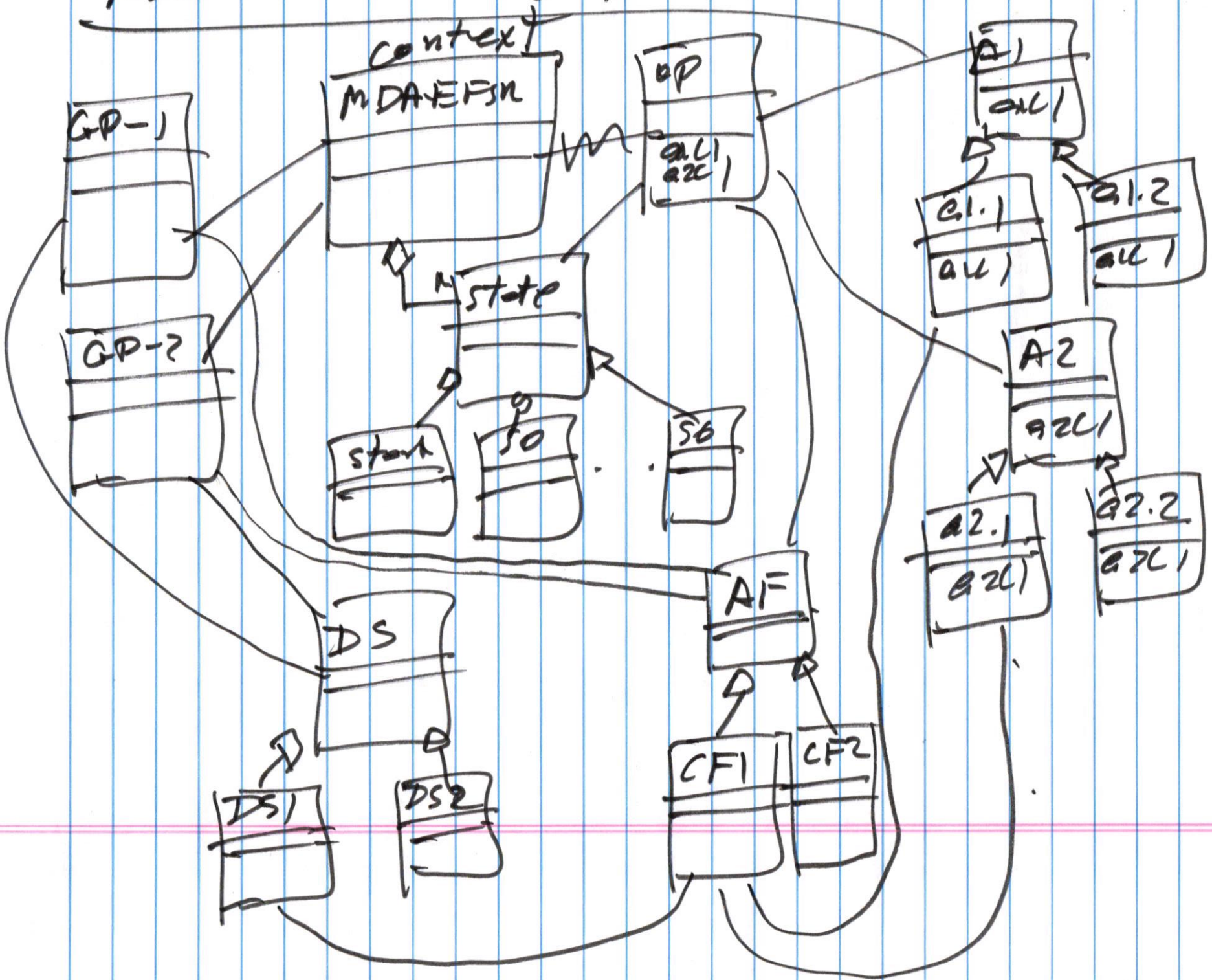
or

de-centralized version!!

Strategy Pattern



Abstract Factory pattern



one class diagram

# of classes	# of Classes
1. GP-1, GP-2	2
2 OP	1
3. Data Store	3
4. MDA-EFSM + state classes 1+1+8	10
5. Strategy pattern (action classes) (1+2)*14 = 42	

Total # of classes = 61 classes!!

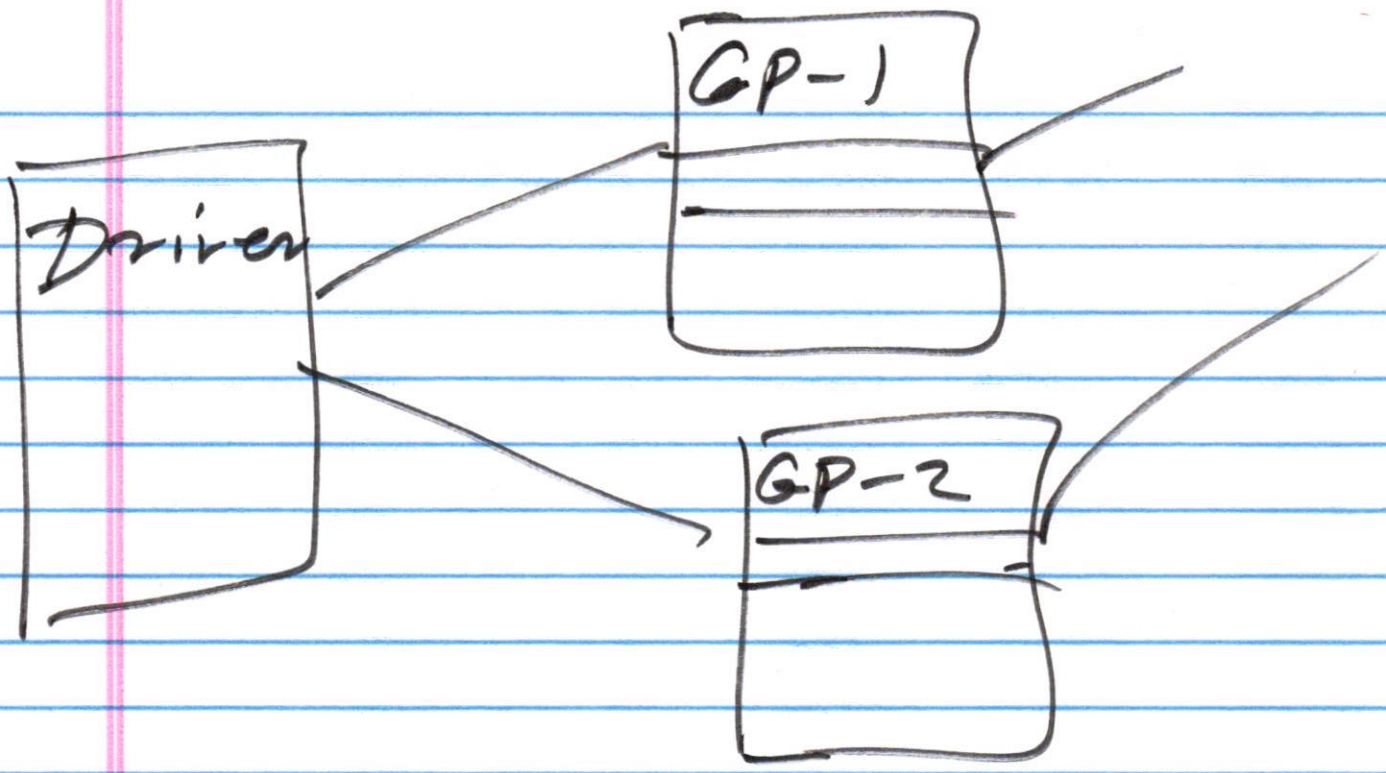
Implementation of your design using OO language.

---

execute and test the design.

---

you must create a test driver (main) to execute and test the design!!

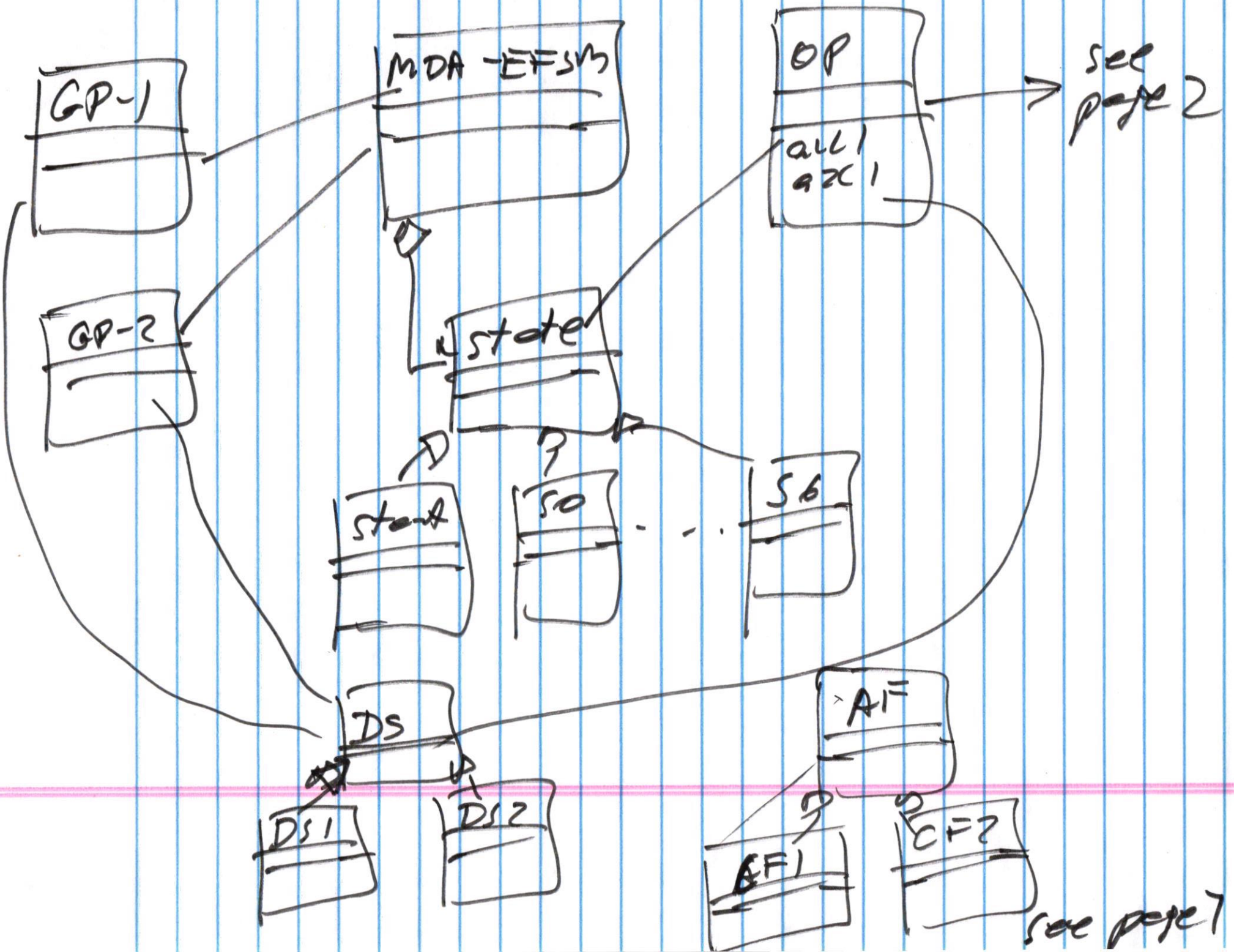


to invoke any operation et selected GP at any time!!

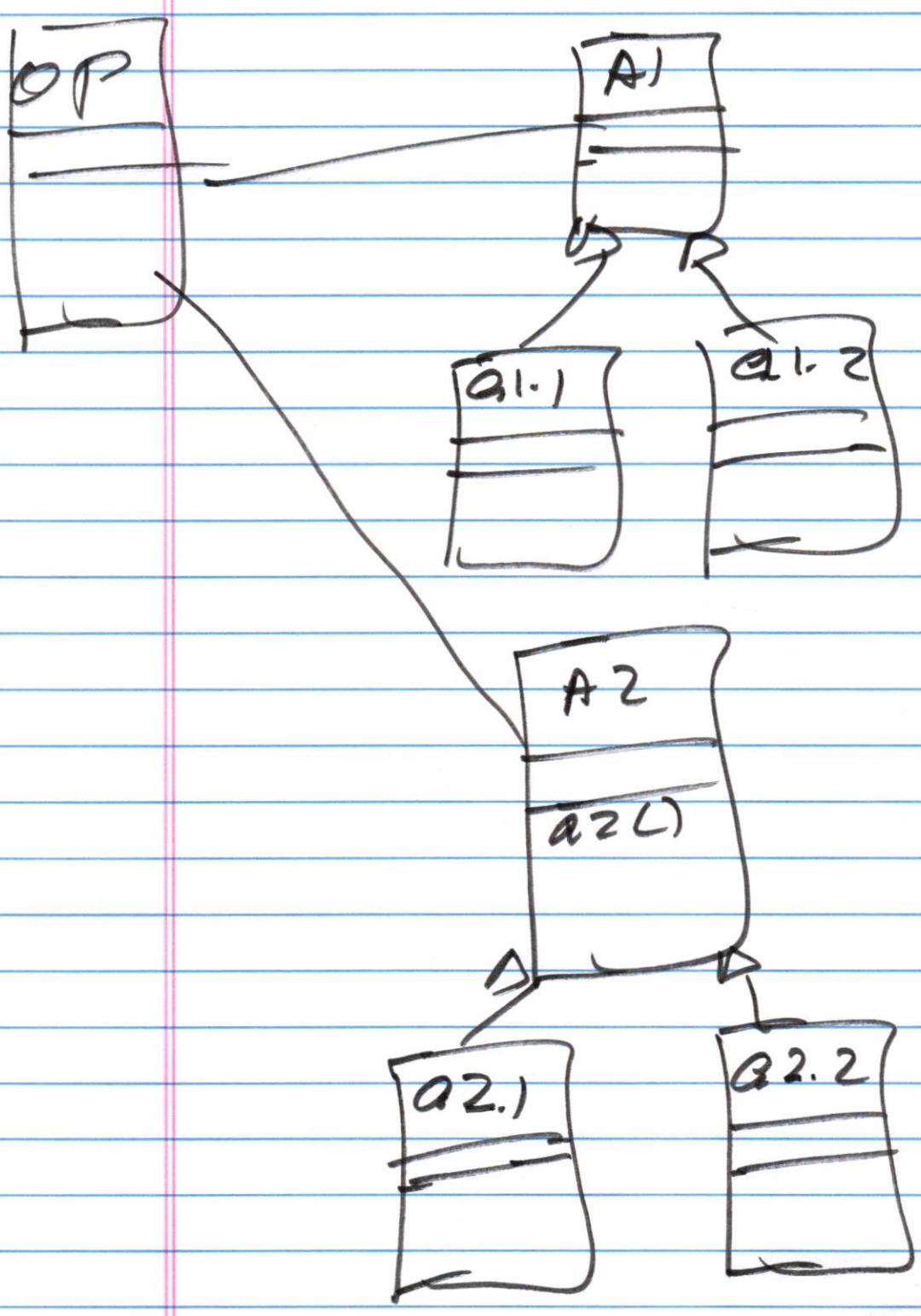
see demo!!

```
main () { // partial driver
….
GasPump_1 gp1;


      cout<< "                              GasPump-1" << endl;
      cout<< "                       MENU of Operations" << endl;
      cout<< "          0. Activate(int)" << endl;
      cout<< "          1. Start()" << endl;
      cout<< "          2. PayCredit" << endl;
      cout<< "          3. Reject()" << endl;
      cout<< "          4. Cancel()" << endl;
      cout<< "          5. Approved()" << endl;
      cout<< "          6. PayCash(int)" << endl;
      cout<< "          7. StartPump()" << endl;
      cout<< "          8. Pump()" << endl;
      cout<< "          9. StopPump()" << endl;
      cout<< "          q. Quit the program" << endl;

      cout<< "  Please make a note of these operations" << endl;
      cout<< "            GasPump-1 Execution" << endl;
          ch='1';
  while (ch !='q') {
      cout<< "  Select Operation: "<<endl;
      cout<<"0-Activate,1-Start,2-PayCredit,3-Reject,4-Cancel,5-Approved,6-PayCash,
             7-StartPump, 8-Pump, 9-StopPump, q-quit"<<endl;
      ch=getch();
      switch (ch) {
        case '0': { //Activate()
                        cout<<"  Operation:  Activate(int a)"<<endl;
                        cout<<"  Enter value of the parameter a:"<<endl;
                        cin>>a;
                        gp1.Activate(a);
                        break;
                  };

        case '1': { //Start
                        cout<<"  Operation:  Start()"<<endl;
                        gp1.Start();
                        break;
                    };

        case '2': { //PayCredit
                        cout<<"  Operation:  PayCredit()"<<endl;
                        gp1.PayCredit();
                        break;
                    };

        case '3': { //Reject
                        cout<<"  Operation:  Reject()"<<endl;
```

```
                    gp1.Reject();
                    break;
                };

    case '4': { //Cancel
                    cout<<"  Operation:  Cancel()"<<endl;
                    gp1.Cancel();
                    break;
                };

    case '5': { //Approved
                    cout<<"  Operation:  Approved()"<<endl;
                    gp1.Approved();
                    break;
                };

    case '6': { //PayCash
                    cout<<"  Operation:  PayCash(int c)"<<endl;
                    cout<<"  Enter value of the parameter c:"<<endl;
                    cin>>c;
                    gp1.PayCash(c);
                    break;
                };

    case '7': { //StartPump
                    cout<<"  Operation:  StartPump()"<<endl;
                    gp1.StartPump();
                    break;
                };


    case '8': { //Pump
                    cout<<"  Operation:  Pump()"<<endl;
                    gp1.Pump();
                    break;
                };

    case '9': { //StopPump
                    cout<<"  Operation:  StopPump()"<<endl;
                    gp1.StopPump();
                    break;
                };
  };// endswitch

}; //endwhile
```


GP-1
MDA-EFSM
OP
see page 2
all
ax
GP-2
state
D
?
start
S0
S6
DS
AF
DS1
DS2
AF1
CF?
see page 7

Page #2



GP-1 objects

GP1, mda-efsm, start, S0, ... S6, OP

GP, a1

cf1

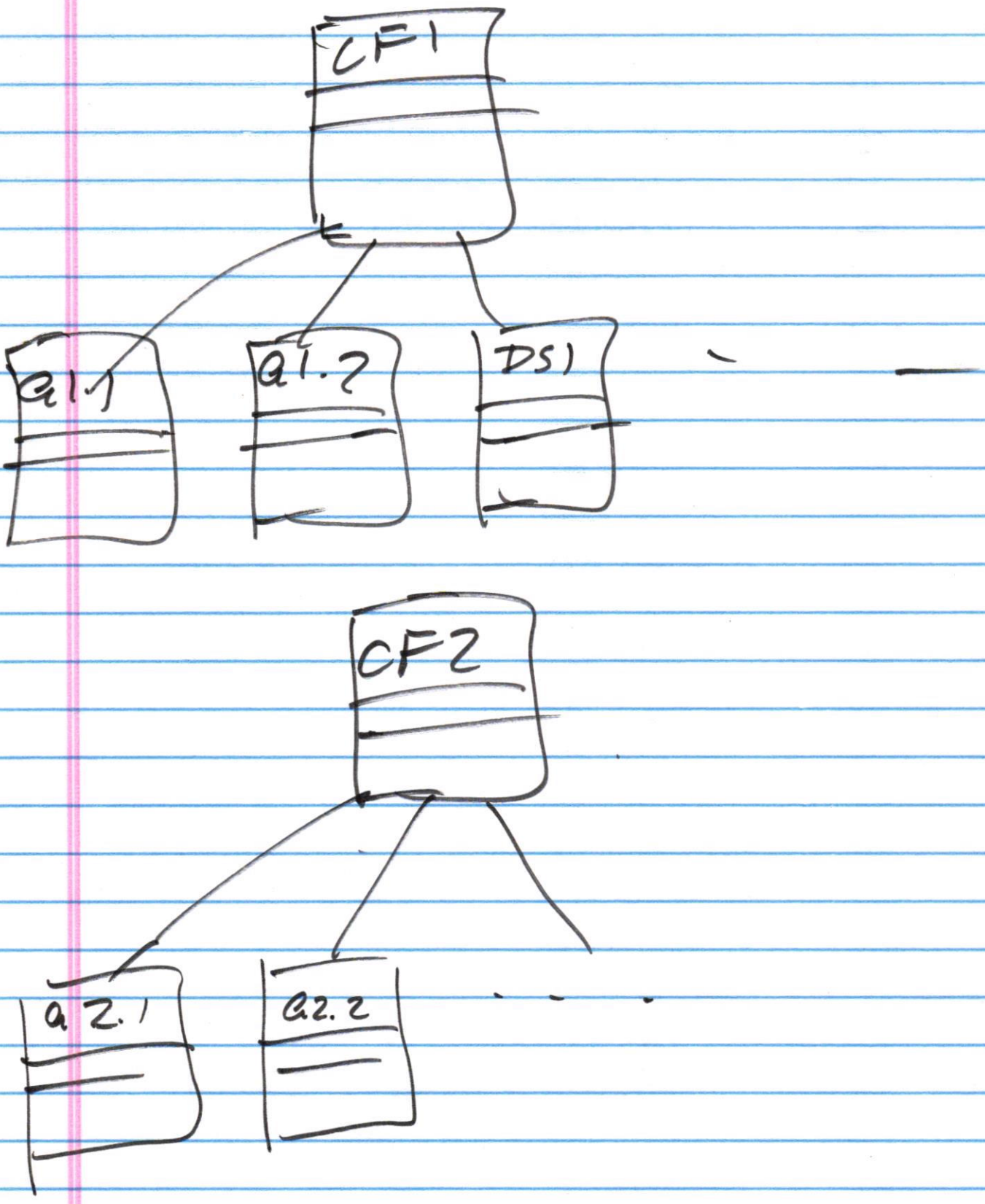
DS1, a1.1, a2.1, ..., a14.1

GP-2 objects

GP-2, mda-efsm, start, S0, ..., S6, OP

cf2

DS2, a2.1, a2.2, ..., a14.2


CF1
Q1.1
Q1.2
DS1
CF2
Q2.1
Q2.2

Layered Architecture

components: Layers

relationship: call relationship

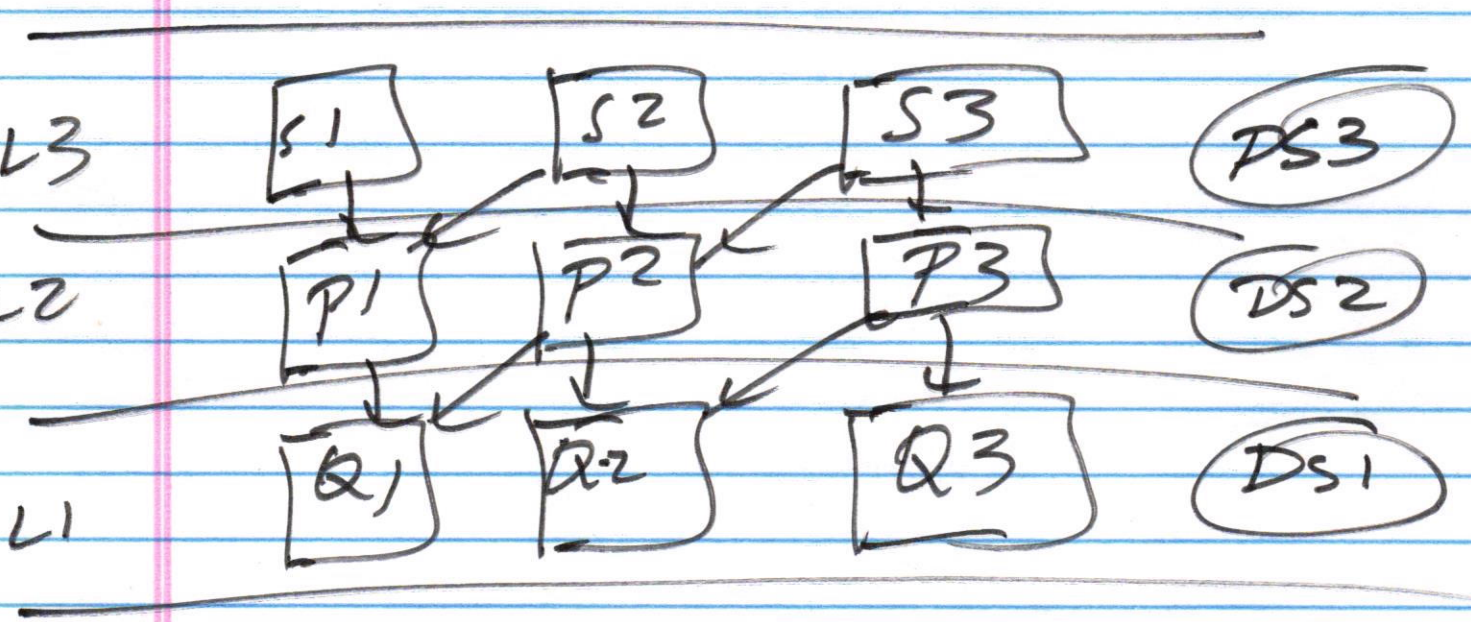
"strict" layer architecture

(a) each layer provides services to the layer directly above.

(b) each layer interacts with two layers.

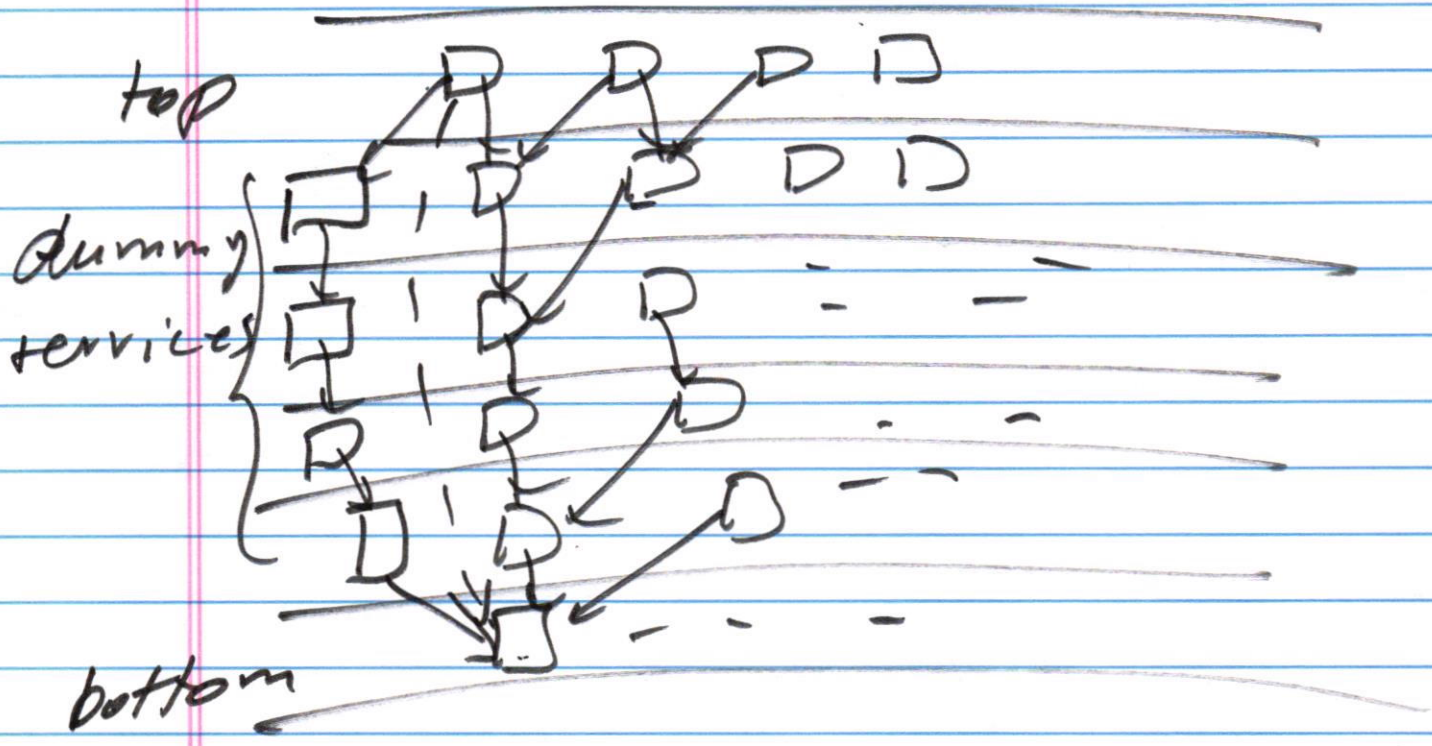
(1) layer above

(2) layer below



strict layer architecture.

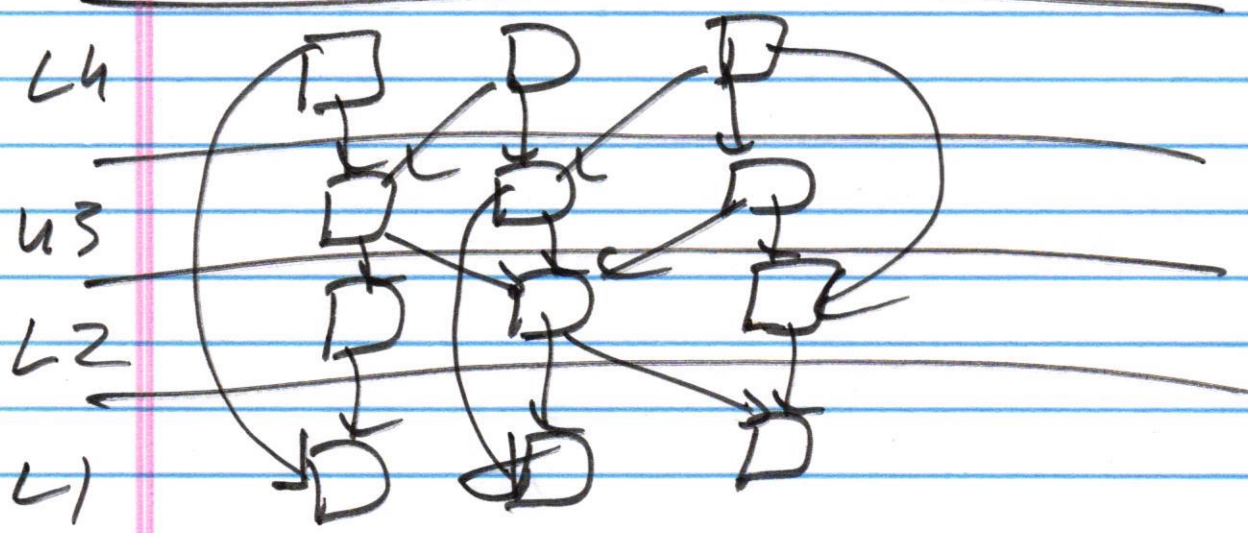
"performance" problems.

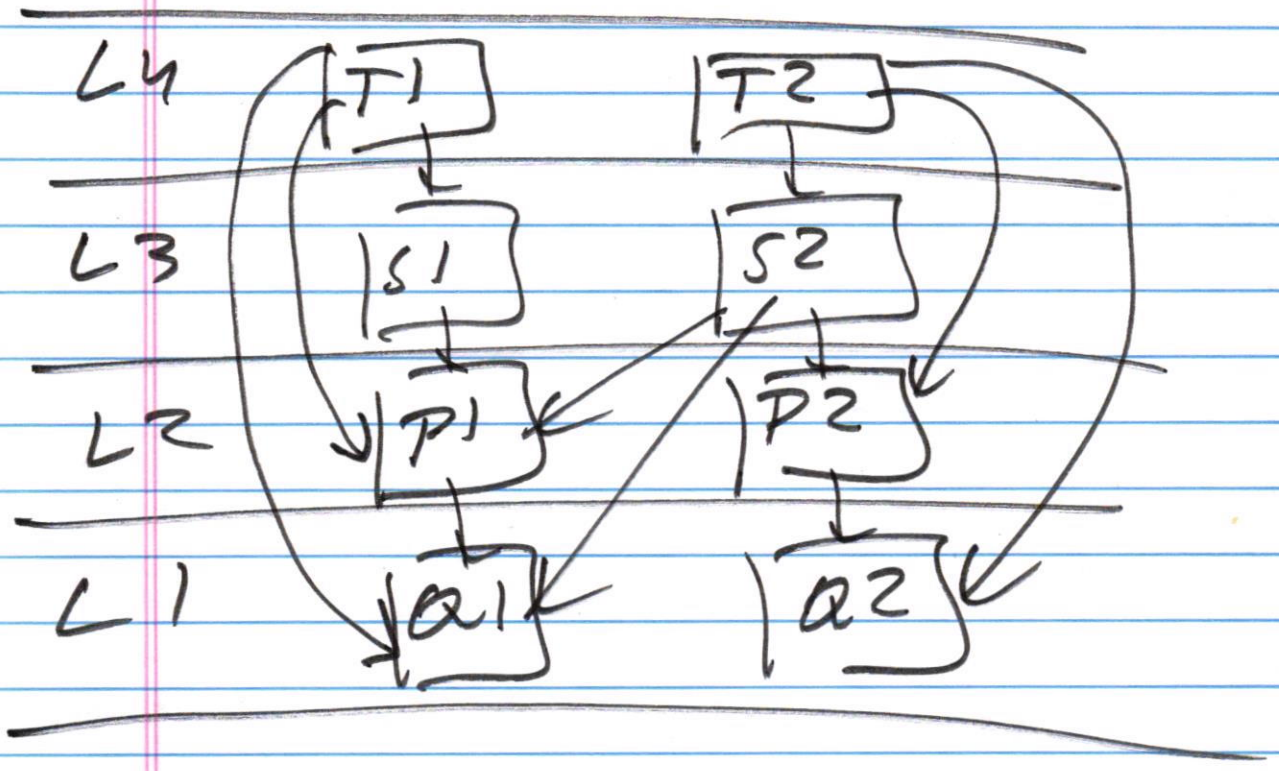

top
dummy
services
bottom

"Relaxed" layer architecture

Property:

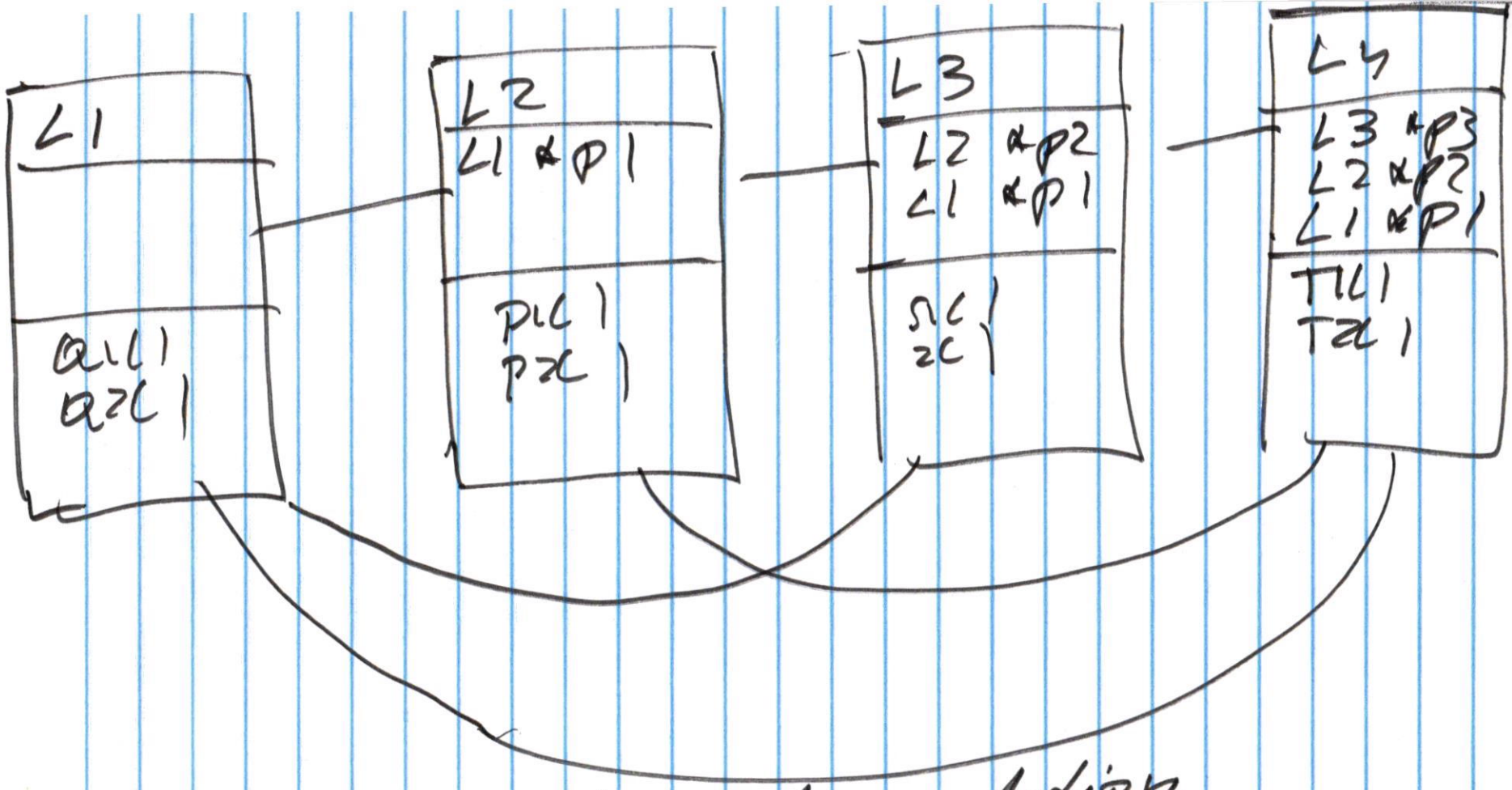
each layer can call "directly" services of all layers below





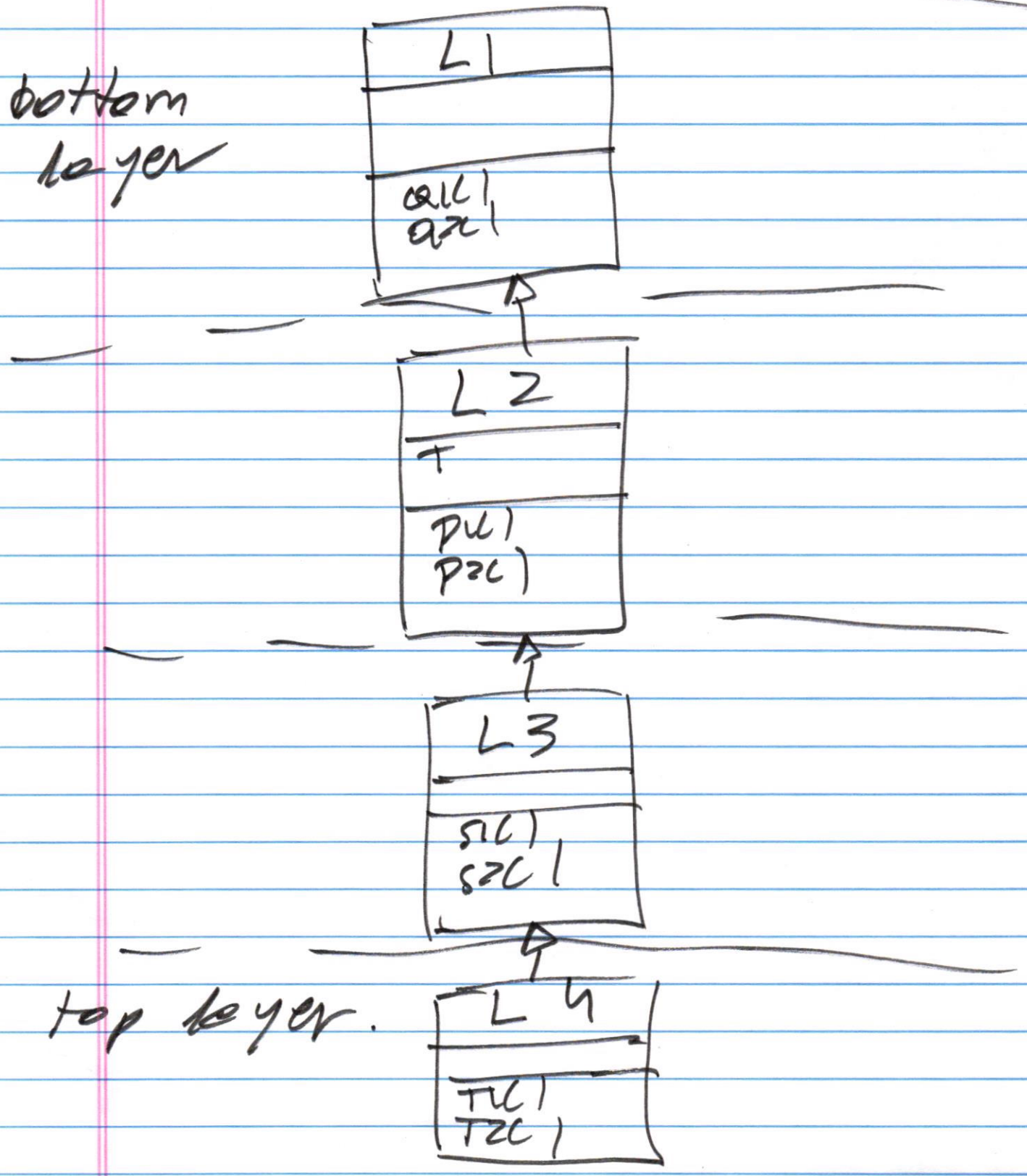
Adv: improved performance

Disadv: loss of maintainability



Association-based solution.

Inheritance based solution



Adv

relatively easy to implement

Disadv

very strong coupling between layers.