

(A) Part #2 of the project
is posted

* Homework #3 is
posted

Project Part # 2

Design and implement
~~Part #1~~
your design based
on MDA-EFSM from
part #1

you can use your
MDA-EFSM providing
that it was correct

OR

you can use the
posted MDA-EFSM

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**):

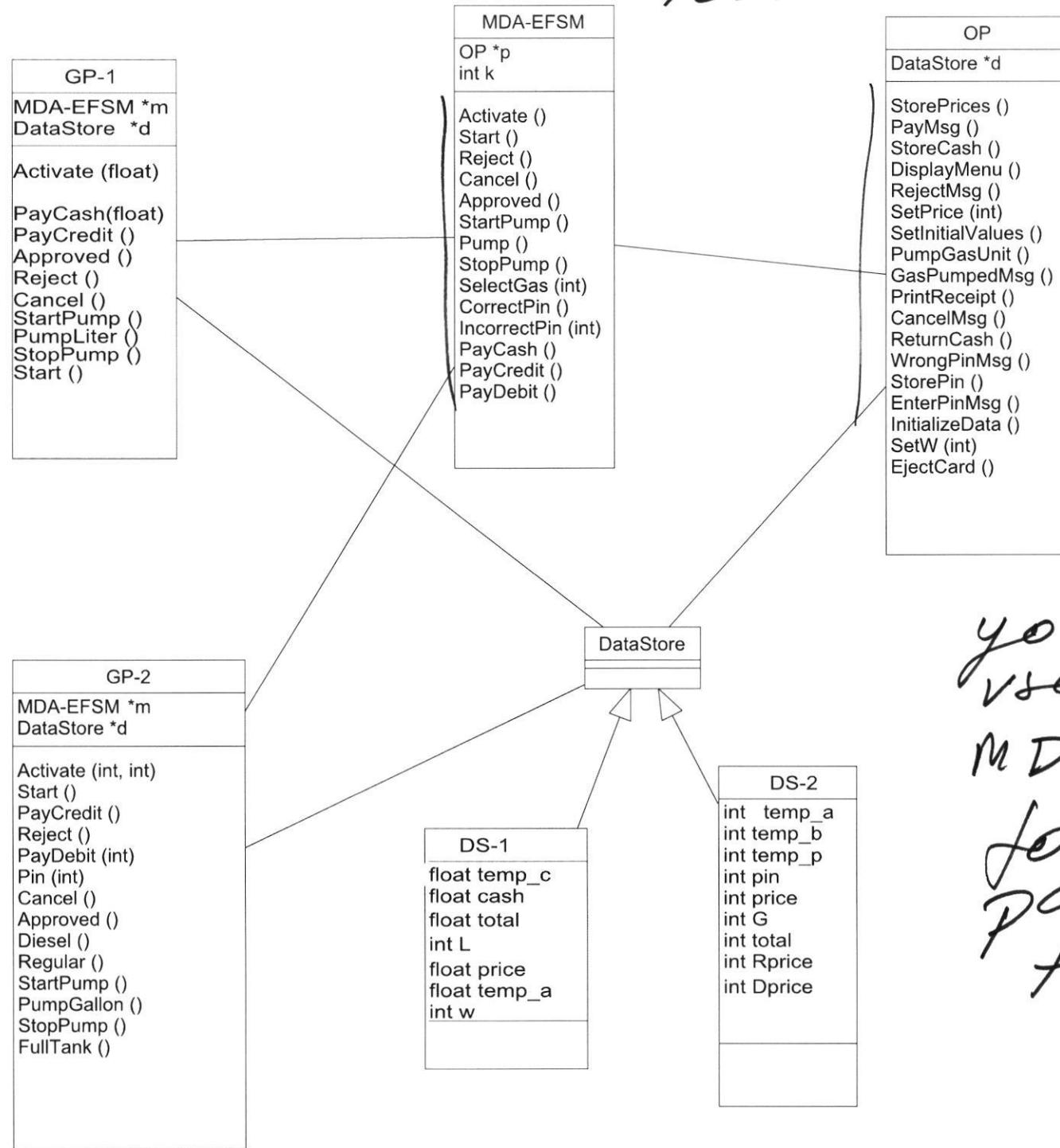
- state pattern
- strategy pattern
- abstract factory pattern.

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.

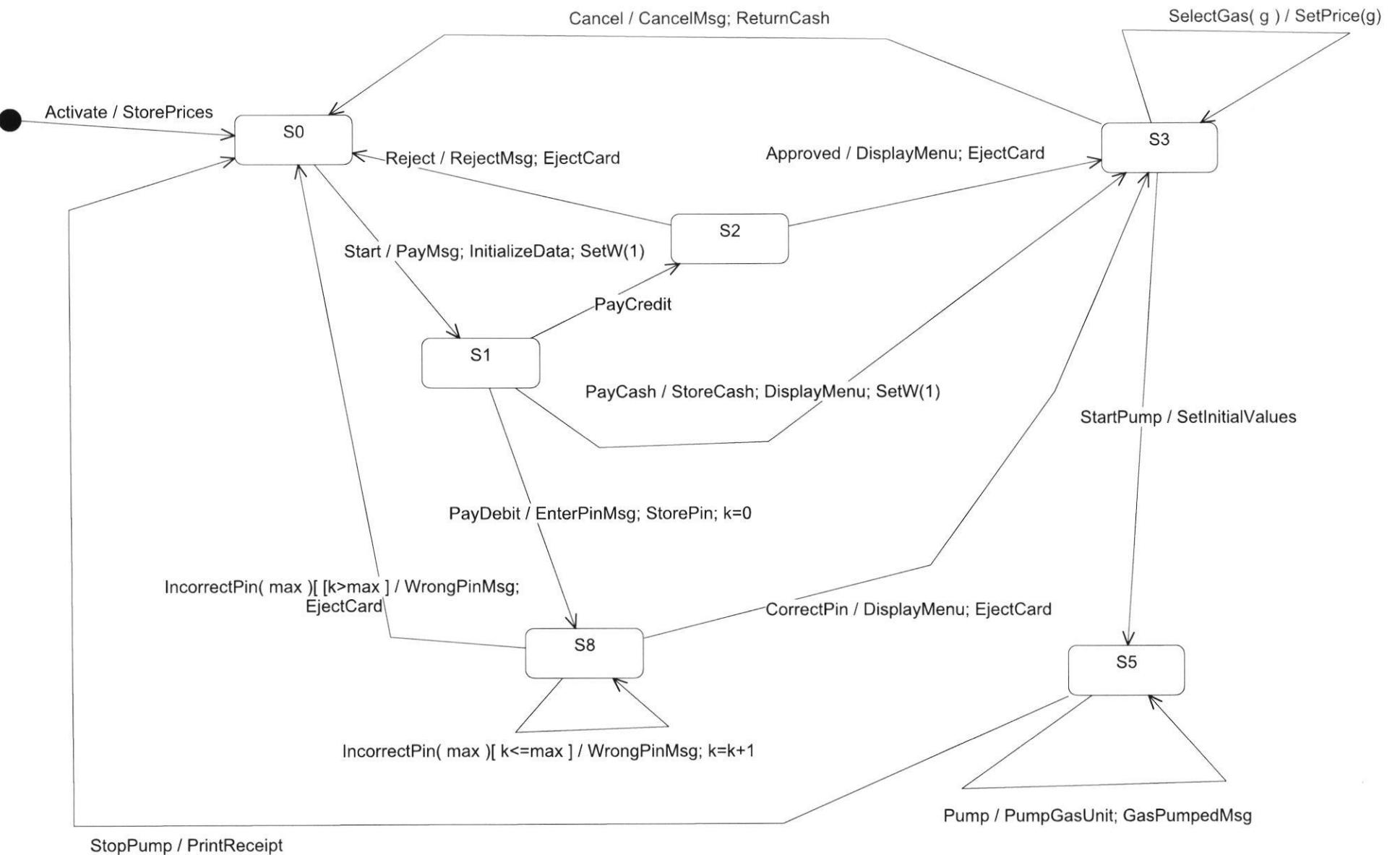
III: Project executables

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.

Posted MDA-EFSM



you can
 use posted
 MDA-EFSM
 for
 Part #2 of
 the project!



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 <i>g</i> identifier as in SelectGas(int <i>g</i>) |
| SetInitialValues | // set <i>G</i> (or <i>L</i>) and <i>total</i> 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 <i>price</i> 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
```

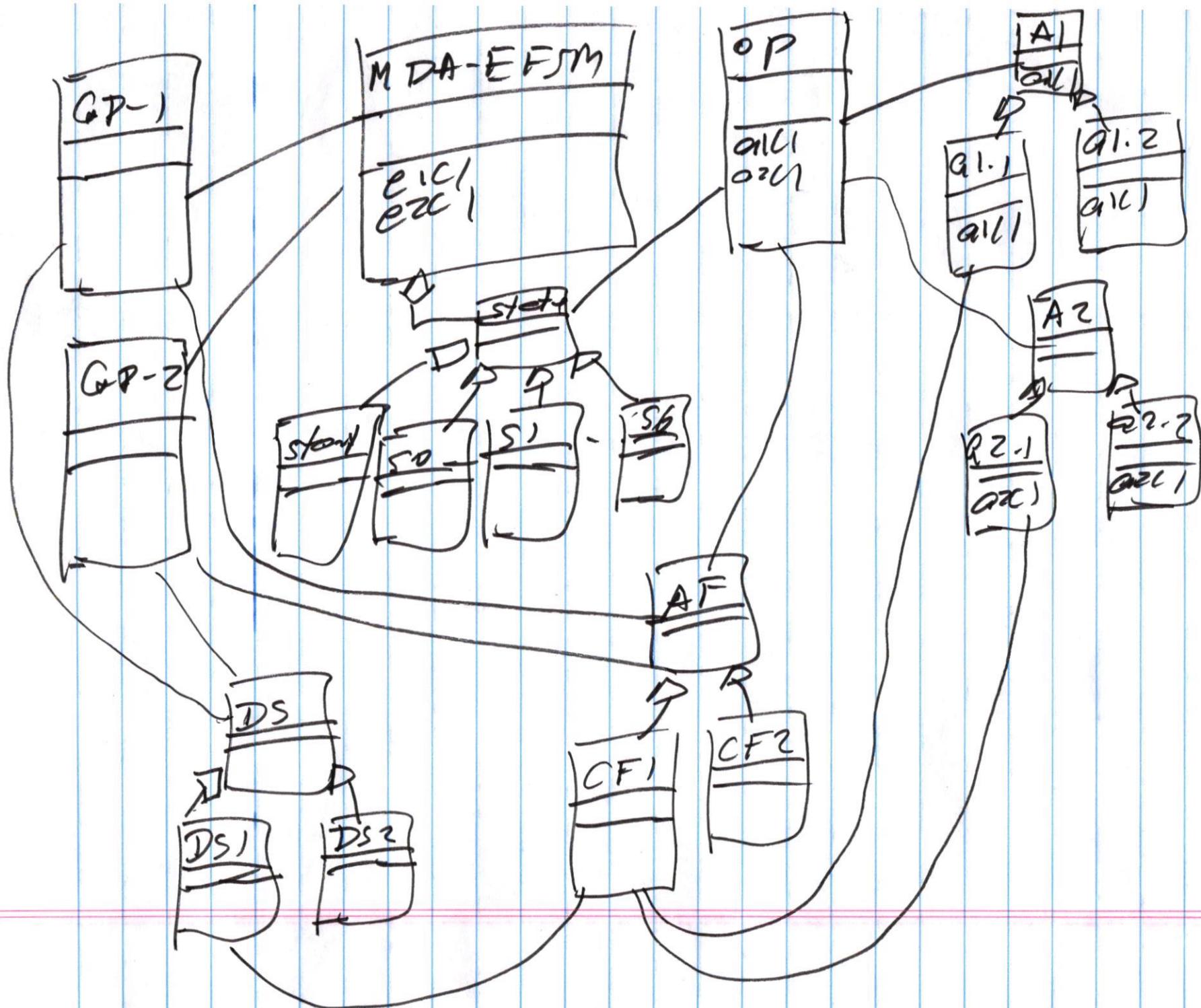
Design for part #2

Incorporate 3 patterns.

1. state pattern

2. strategy - " -

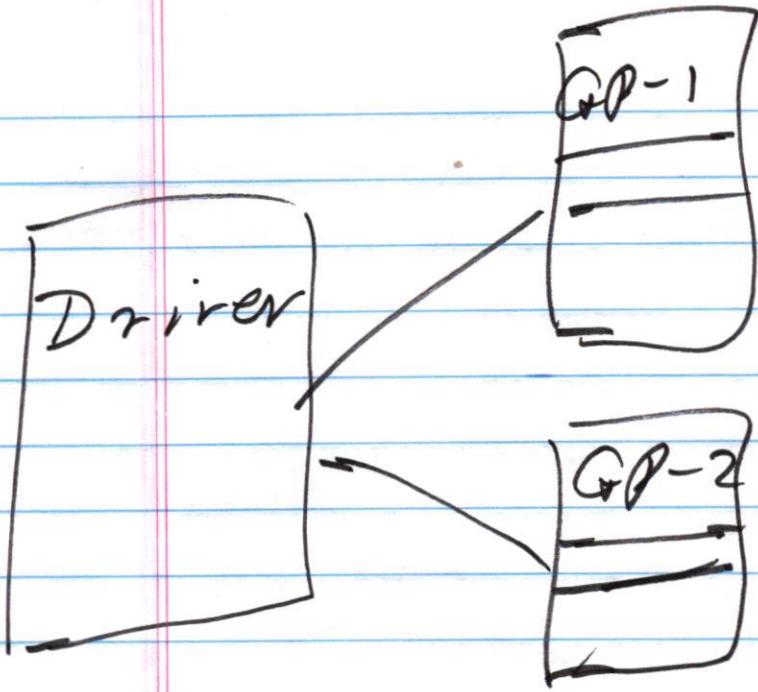
3. abstract factory - " -



Implementation of the design
in OO language
to execute and test the
design

Goal:

to be able to
invoke any operation
of GPs at any time



Driver

1. Select GP for execution
2. execute 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

DEMO!

*Postcode
Partial driver
for C++*

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

cout<< "                                     GasPump-1" << endl;
cout<< "                                     MENU of Operations" << endl;
cout<< "                                     0. Activate(float)" << endl;
cout<< "                                     1. Start()" << endl;
cout<< "                                     2. PayCredit" << endl;
cout<< "                                     3. Reject()" << endl;
cout<< "                                     4. Cancel()" << endl;
cout<< "                                     5. Approved()" << endl;
cout<< "                                     6. PayCash(float)" << endl;
cout<< "                                     7. StartPump()" << endl;
cout<< "                                     8. PumpLiter()" << 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-Star
tPump, 8-PumpLiter, 9-StopPump, q-quit"<<endl;
    ch=getch();
    switch (ch) {
        case '0': { //Activate()
            cout<<" Operation: Activate(float 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(float 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': { //PumpLiter
    cout<<" Operation: PumpLiter()"<<endl;
    gp1.PumpLiter();
    break;
};

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

};// endswitch

}; //endwhile
```

posted partial driver for Java

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        GasPump_1 gpl = new GasPump_1();

        System.out.println("GasPump-1");
        System.out.println("MENU of Operations");
        System.out.println("0. Activate(float)");
        System.out.println("1. Start()");
        System.out.println("2. PayCredit");
        System.out.println("3. Reject()");
        System.out.println("4. Cancel()");
        System.out.println("5. Approved()");
        System.out.println("6. PayCash(float)");
        System.out.println("7. StartPump()");
        System.out.println("8. Pump()");
        System.out.println("9. StopPump()");
        System.out.println("q. Quit the program");
        System.out.println("Please make a note of these operations");
        System.out.println("GasPump-1 Execution");

        char ch = '1';
        while (ch != 'q') {
            System.out.println("Select Operation:");
            System.out.println("0-Activate,1-Start,2-PayCredit,3-Reject,4-"
Cancel,5-Approved,6-PayCash,7-StartPump,8-PumpLiter,9-StopPump,q-quit");
            ch = sc.next().charAt(0);

            switch (ch) {
                case '0': {
                    System.out.println("Operation: Activate(float a)");
                    System.out.print("Enter value of the parameter a: ");
                    float a = sc.nextFloat();
                    gpl.Activate(a);
                    break;
                }
                case '1': {
                    System.out.println("Operation: Start()");
                    gpl.Start();
                    break;
                }
                case '2': {
                    System.out.println("Operation: PayCredit()");
                    gpl.PayCredit();
                    break;
                }
                case '3': {
                    System.out.println("Operation: Reject()");
                    gpl.Reject();
                    break;
                }
            }
        }
    }
}
```

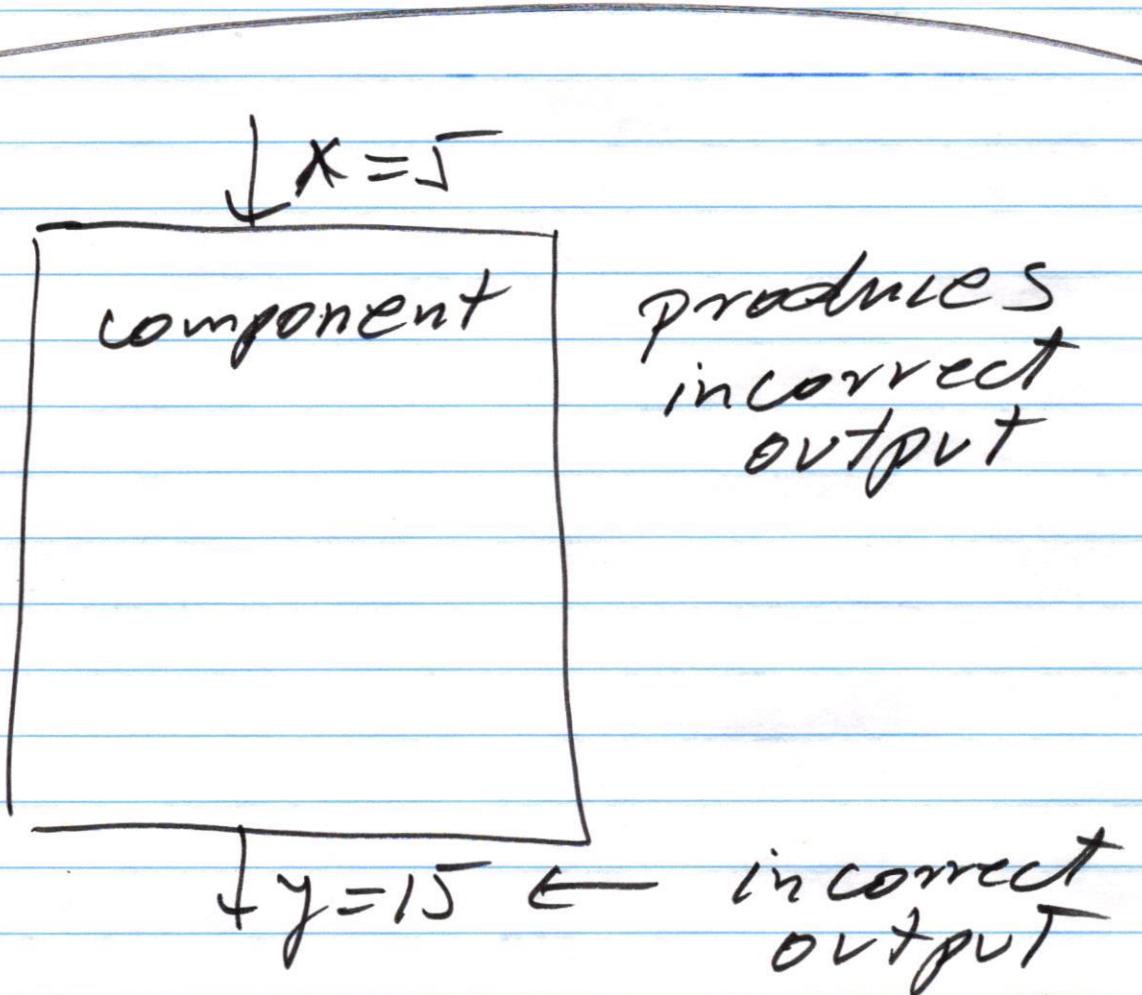
```

        case '4': {
            System.out.println("Operation: Cancel()");
            gp1.Cancel();
            break;
        }
        case '5': {
            System.out.println("Operation: Approved()");
            gp1.Approved();
            break;
        }
        case '6': {
            System.out.println("Operation: PayCash(float c)");
            System.out.print("Enter value of the parameter c: ");
            float c = sc.nextFloat();
            gp1.PayCash(c);
            break;
        }
        case '7': {
            System.out.println("Operation: StartPump()");
            gp1.StartPump();
            break;
        }
        case '8': {
            System.out.println("Operation: PumpLiter()");
            gp1.PumpLiter();
            break;
        }
        case '9': {
            System.out.println("Operation: StopPump()");
            gp1.StopPump();
            break;
        }
        case 'q': {
            System.out.println("Quitting program...");
            break;
        }
        default: {
            System.out.println("Invalid option. Try again.");
        }
    }
}

sc.close();
}
}

```

Fault tolerant architecture



expected $y = 25$

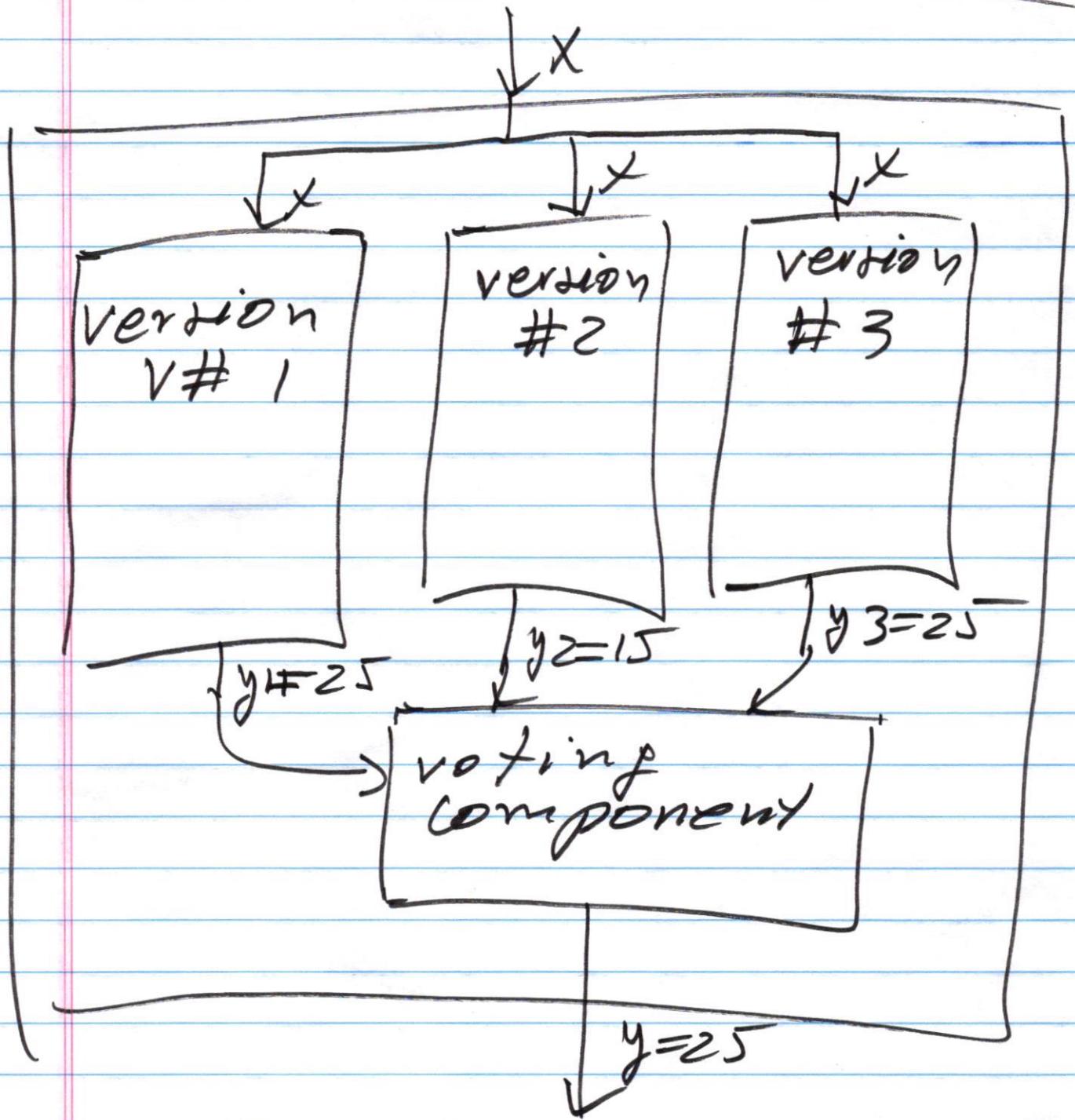
There is no exception
during execution

Fault-tolerant architecture

(*) N-version architecture

- * Recovery-block -n-
- * N-self checking -n-
- * . . .

N-version architecture



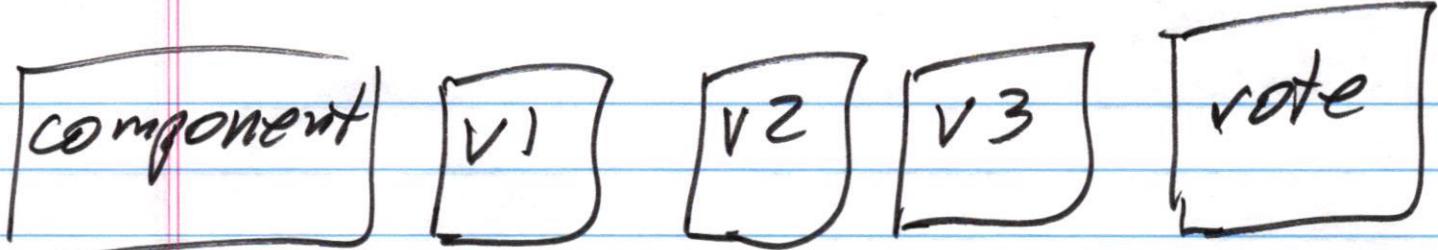
Different versions

- ↳ different teams
- ↳ — — 1) — algorithms
- ↳ — — 2) — data structures.
- ↳ ...

$\text{vote}(y_1, y_2, y_3)$

```
if ( $y_1 == y_2$ ) return  $y_1$ 
if ( $y_1 == y_3$ ) return  $y_1$ 
if ( $y_2 == y_3$ ) return  $y_2$ 
 $y = \text{randomly selected}$   $y_1, y_2, y_3$ 
return  $y$ .
```

↳



execute
 $\boxed{x} \rightarrow$ executed)

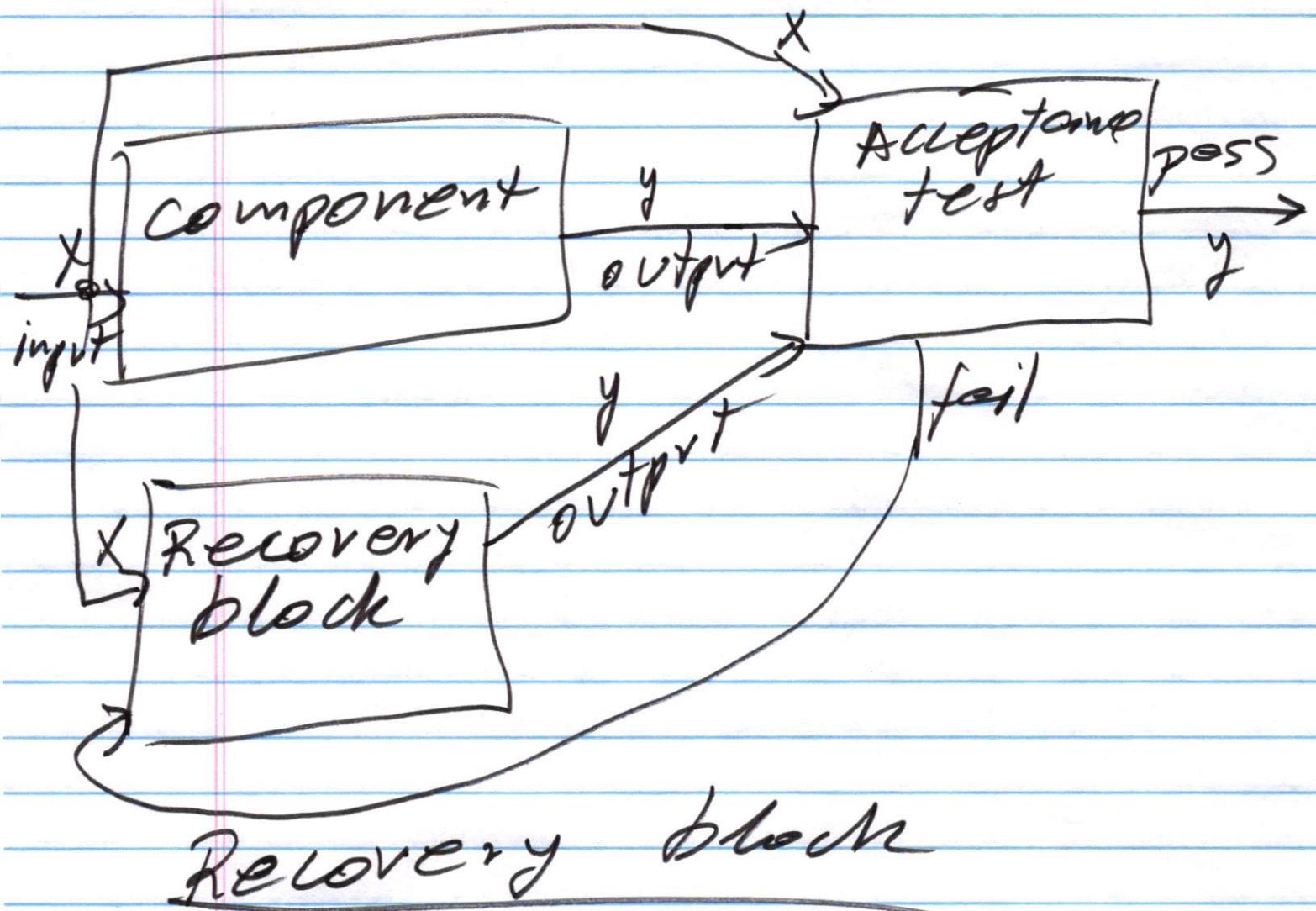
$y_1 = y$ $\boxed{x} \rightarrow$ executed)

$y_2 = y$ $\boxed{x} \rightarrow$ executed)

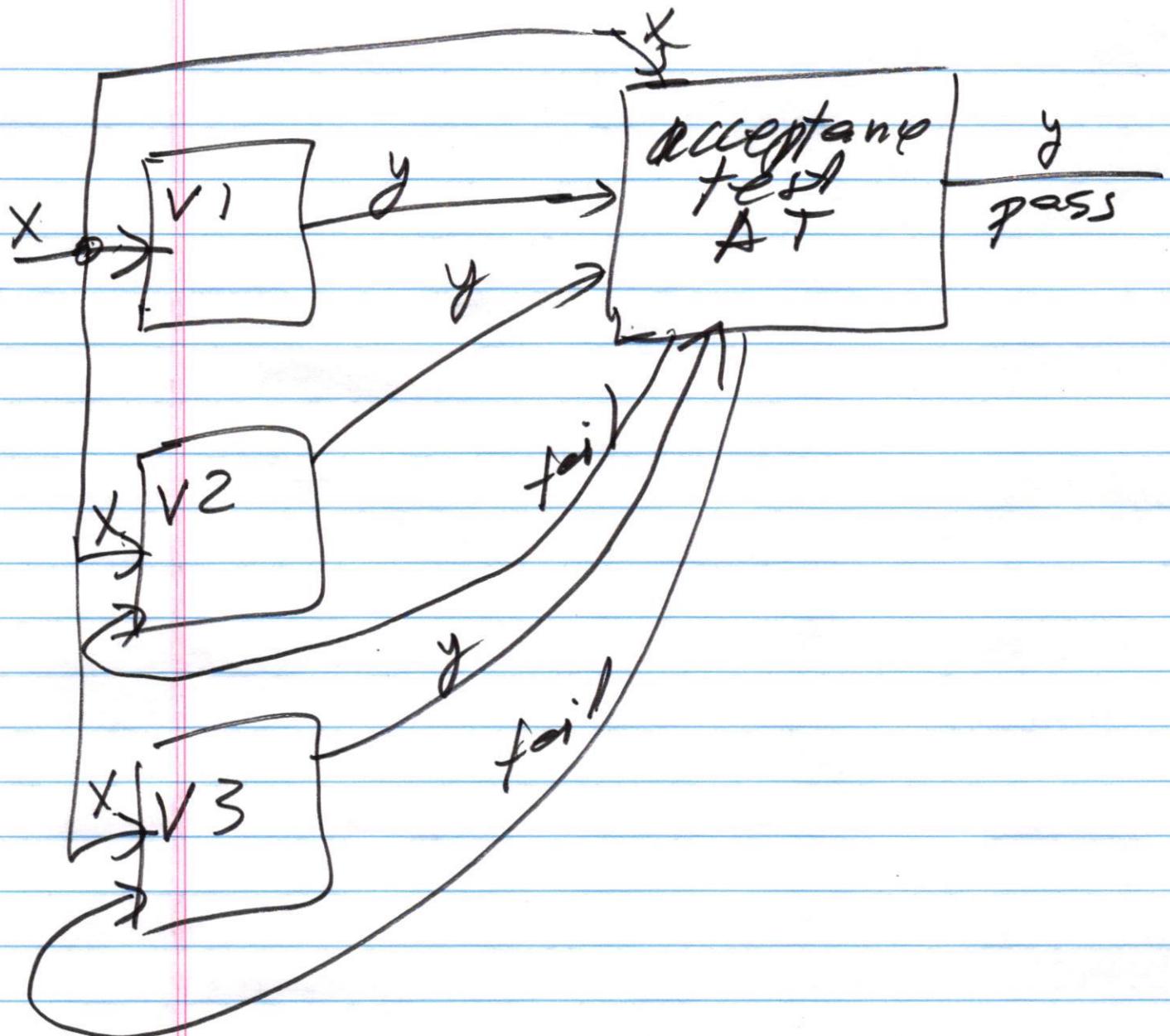
$y_3 = y$ $\boxed{x} \rightarrow$ voted)

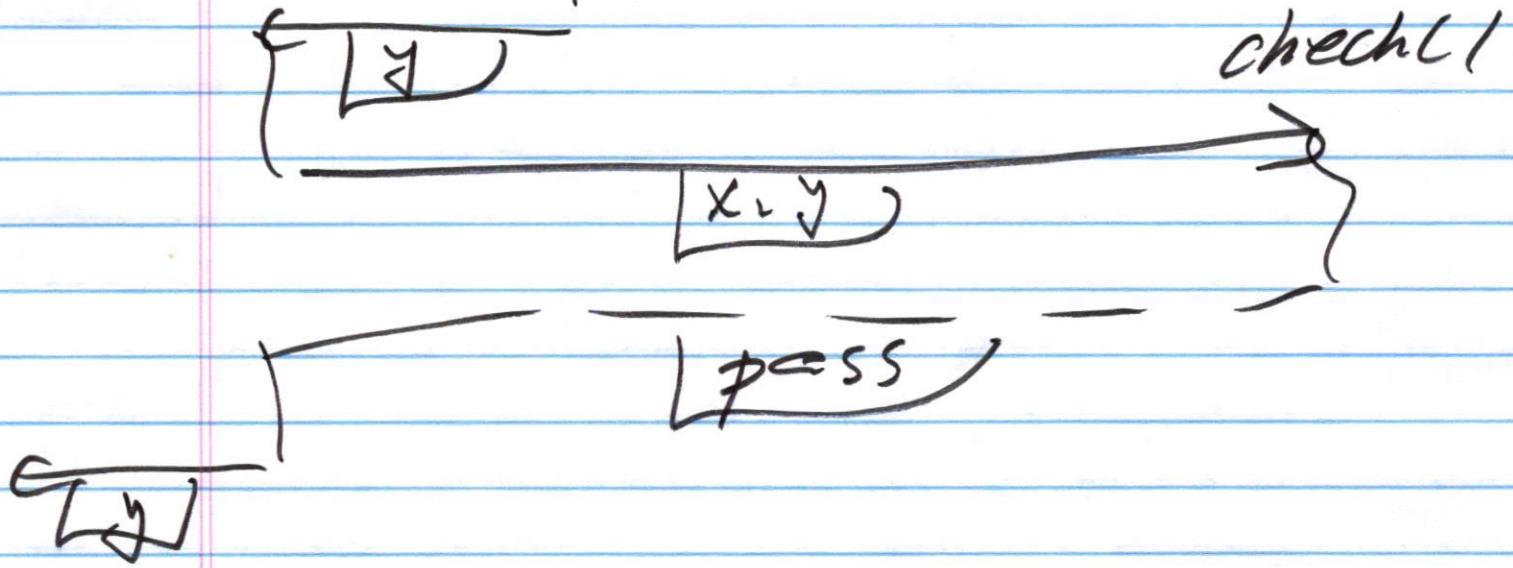
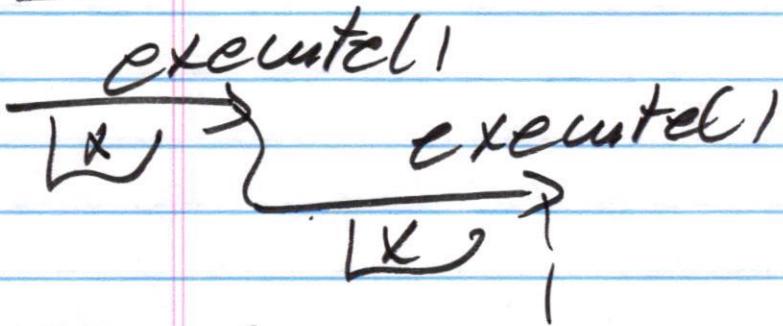
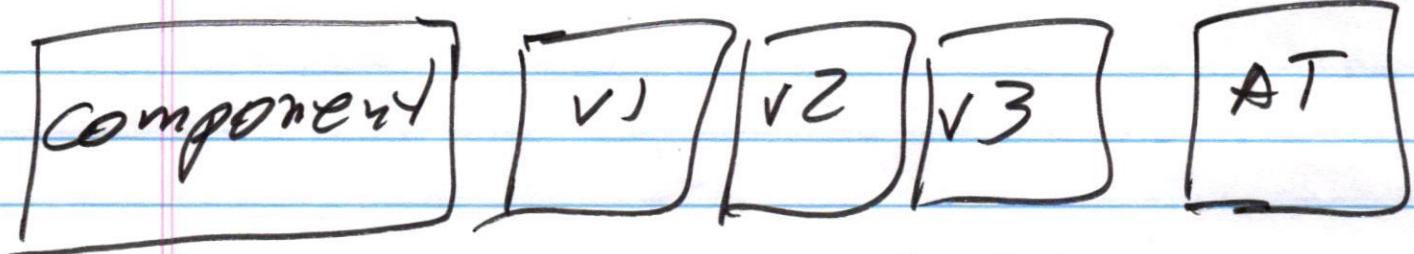
Ty

2. Recovery Block architecture.

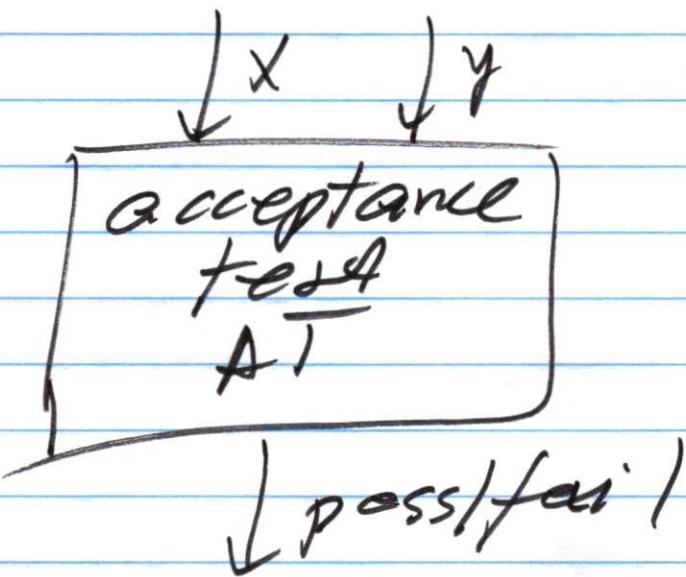


- Recovery block
- (a) old ~~but~~ reliable component but slow
 - (b) another version



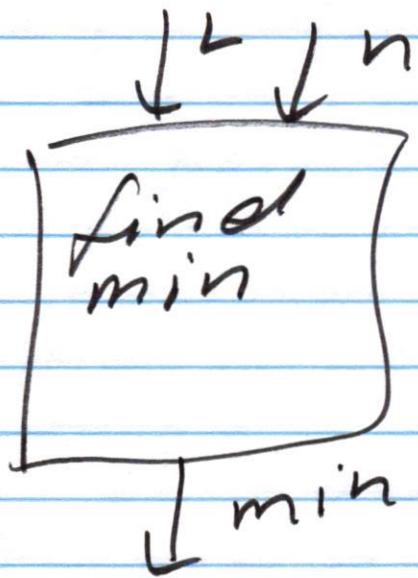


Acceptance Test



1. to determine if output y is "acceptable"
2. acceptance test must be fast
it cannot be another version of the component.
3. it may be difficult to identify an acceptance test

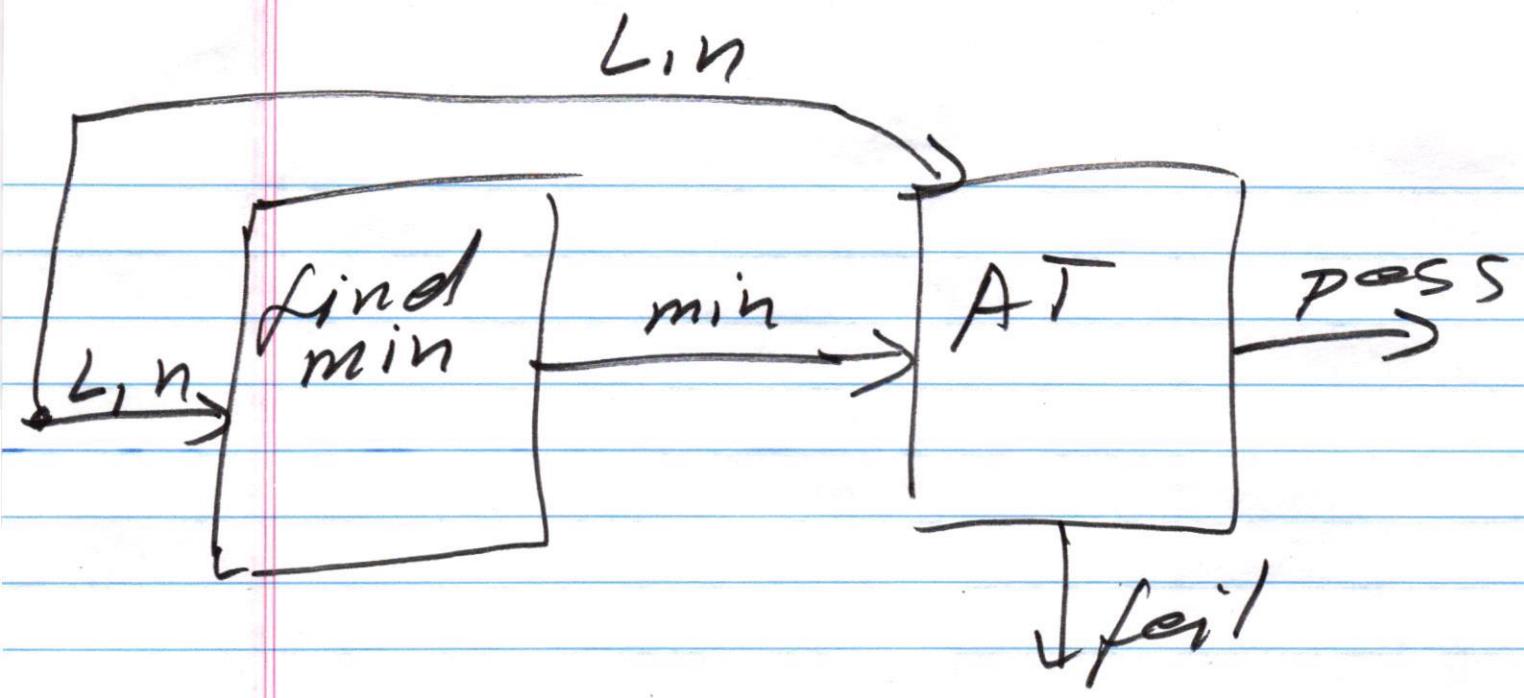
component to find
a minimum on
the list of integers.

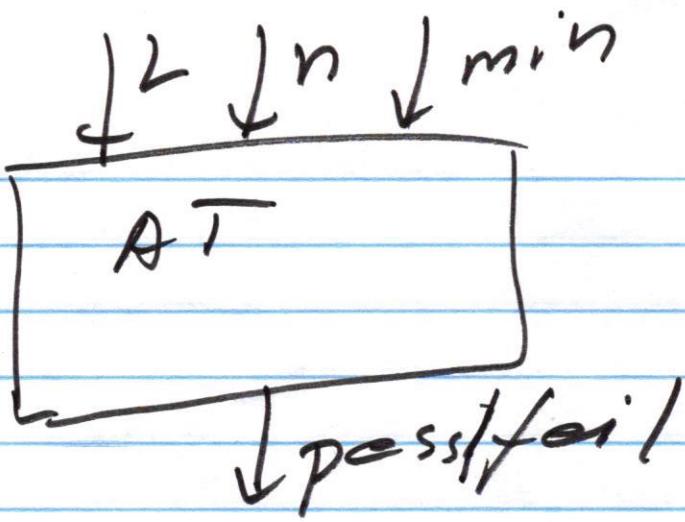


L: list of integers

n: # of elements

min: min element on the
list



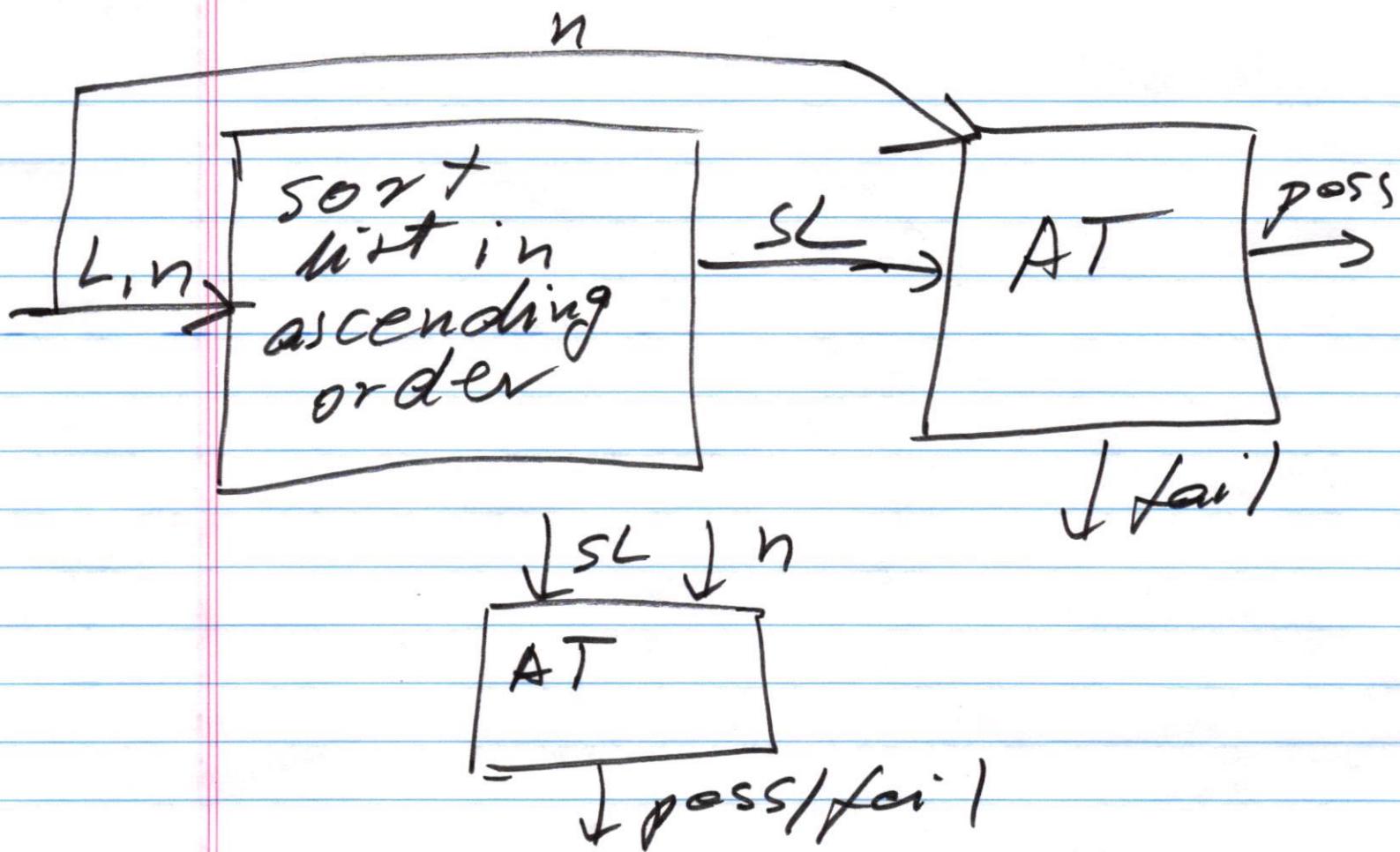


for $i=1$ to n do
 if ($\min > L[i]$) return fail
 end
 return pass

$L: 4, 2, 3, 8$ | input
 $n = 4$

-
- (a) $\min = 2 \Rightarrow \text{pass} \quad \text{OK.}$
 (b) $\min = 3 \Rightarrow \text{fail} \quad \text{OK}$
 (c) $\min = 1 \Rightarrow \text{pass} \quad \text{not OK.}$

SL: Sorted List



for $i=1$ to $n-1$ do
 if ($SL[i] > SL[i+1]$)
 return fail
 end
 return poss

$L: 4, 2, 8, 7$
 $n = n$

$SL: 2, 7, 4, 8$
fail on

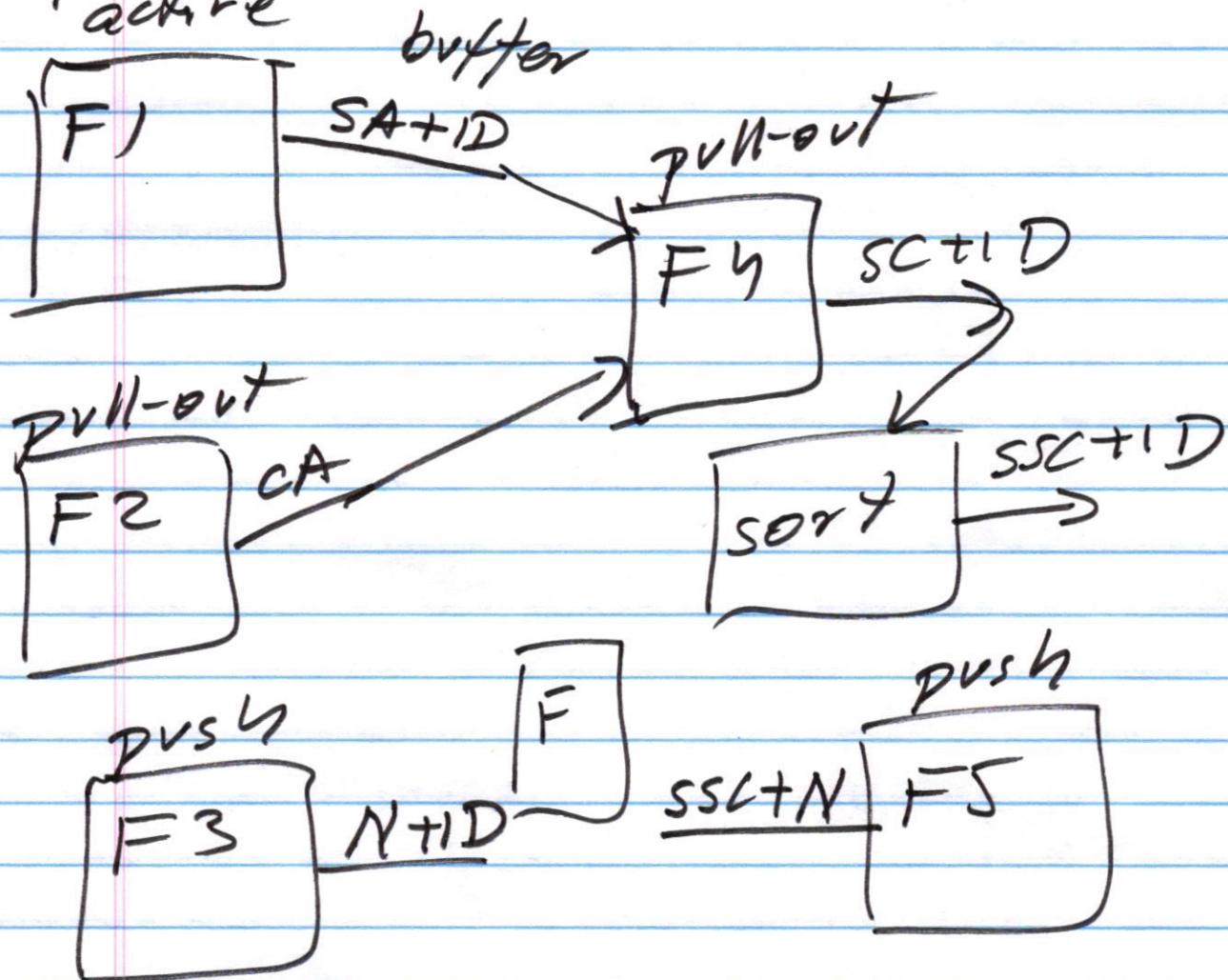
$SL: 2, 4, 7, 8$
pass OK

$SL: 2, 3, 7, 8$
pass not OK.

Homework #3

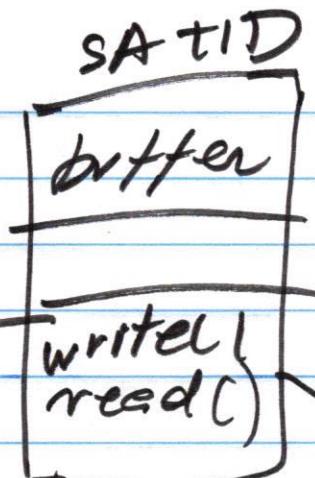
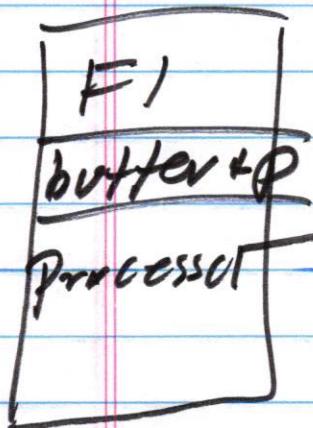
Problem #1 Pipes and filters

part A
active

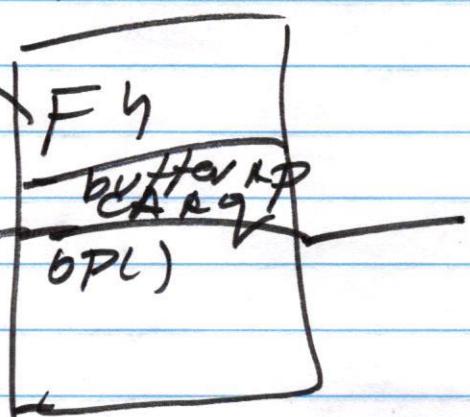


SSC: sorted scores
N: Names

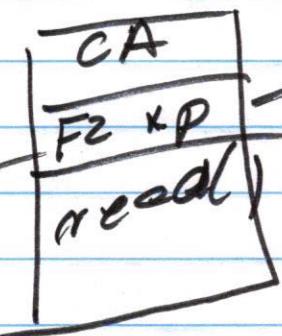
active



pull-out



pull-out



OPC)

P → read(SA, ID)
q → read(CA)

compute scores SC

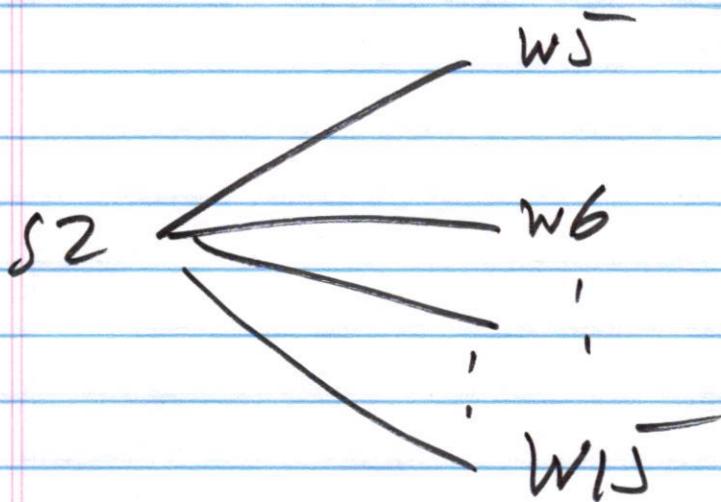
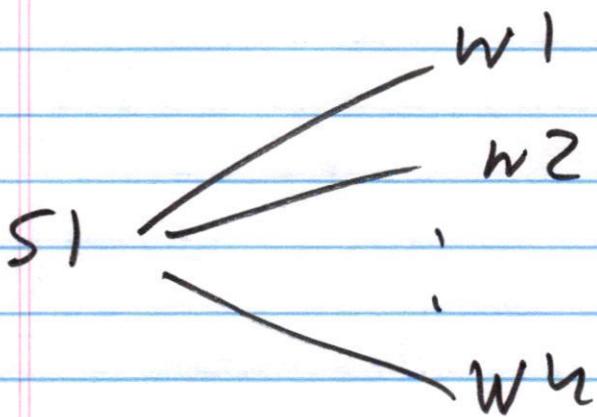
processor

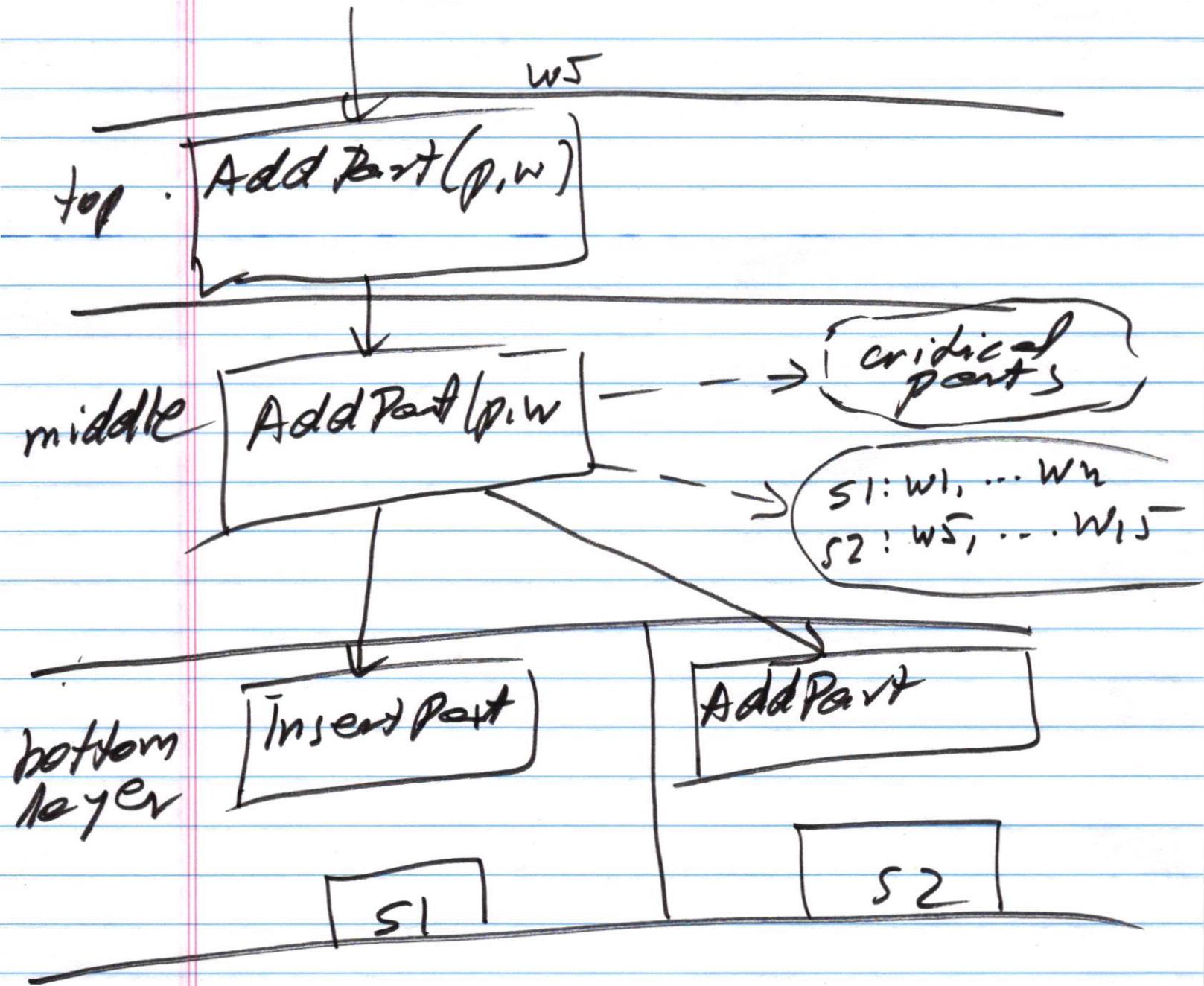
read SATID
P → write(SA, ID)

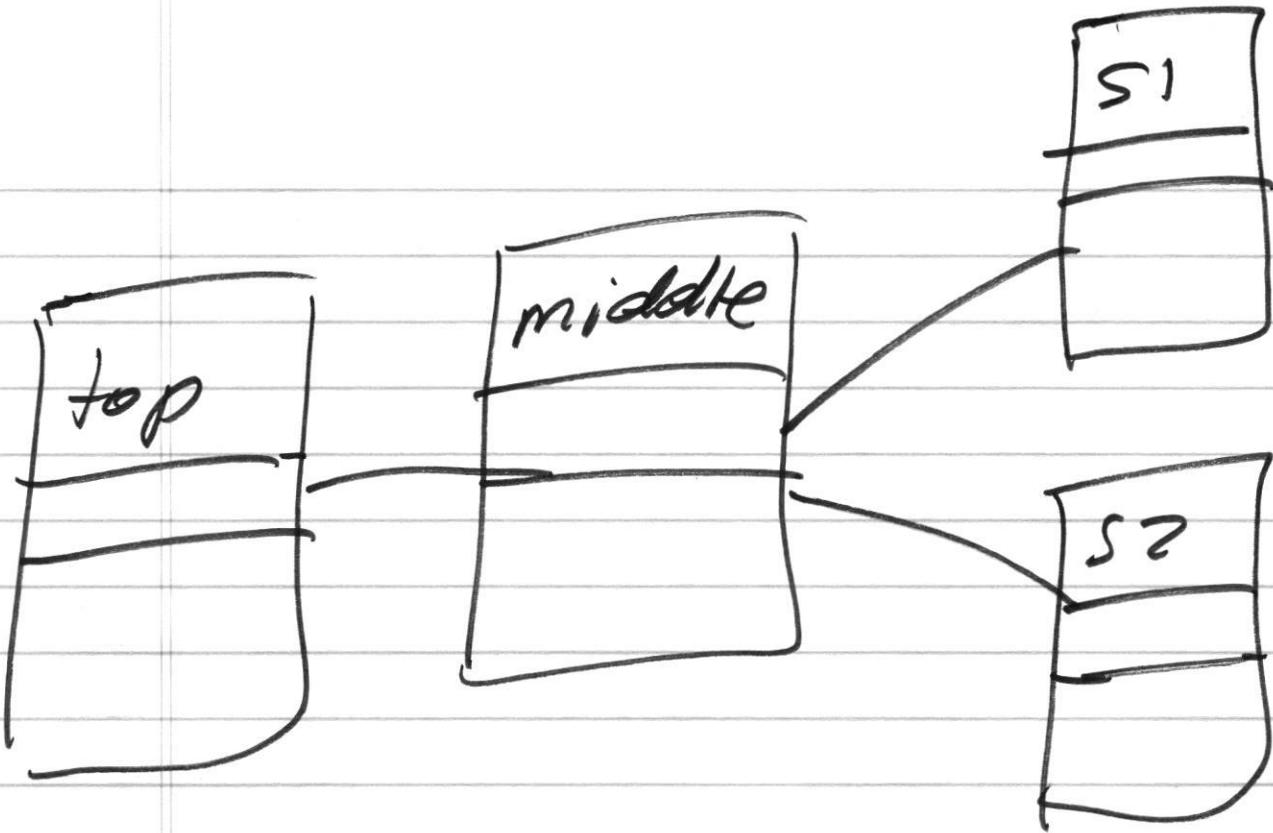
jj

Problem # 2

strict layered architecture







Problem #3

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

Services supported by **Server-S2**:

| | |
|--------------------------------------|---|
| void AddPart (string w, string p) | //add part <i>p</i> to warehouse <i>w</i> , where <i>p</i> is a part ID |
| void DeletePart (string w, string p) | //deletes part <i>p</i> from warehouse <i>w</i> |
| int GetNumParts (string p) | //returns the total number of part <i>p</i> in all warehouses |
| int IsPart (string p) | //returns 1, if part <i>p</i> 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 <i>p</i> to warehouse <i>w</i> |
| void Remove_Part (string p, string w) | //deletes part <i>p</i> from warehouse <i>w</i> |
| int GetNumOfParts (string p) | //returns the total number of part <i>p</i> in all warehouses |
| int Is_Part (string p) | //returns 1, if part <i>p</i> 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.

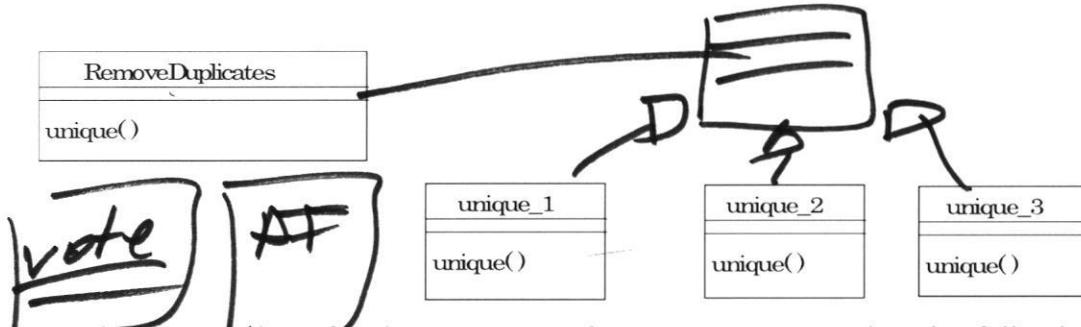
For example, for the following input:

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.