- Project Part #2 is posted

- sample MDA-EFSM for Gas Pump components is posted.

- Homework #3 is posted

- Midterm graders are posted

Fault tolerant architecture

How to achieve high reliability/security of software system/components?

1. do more testing
2. by design

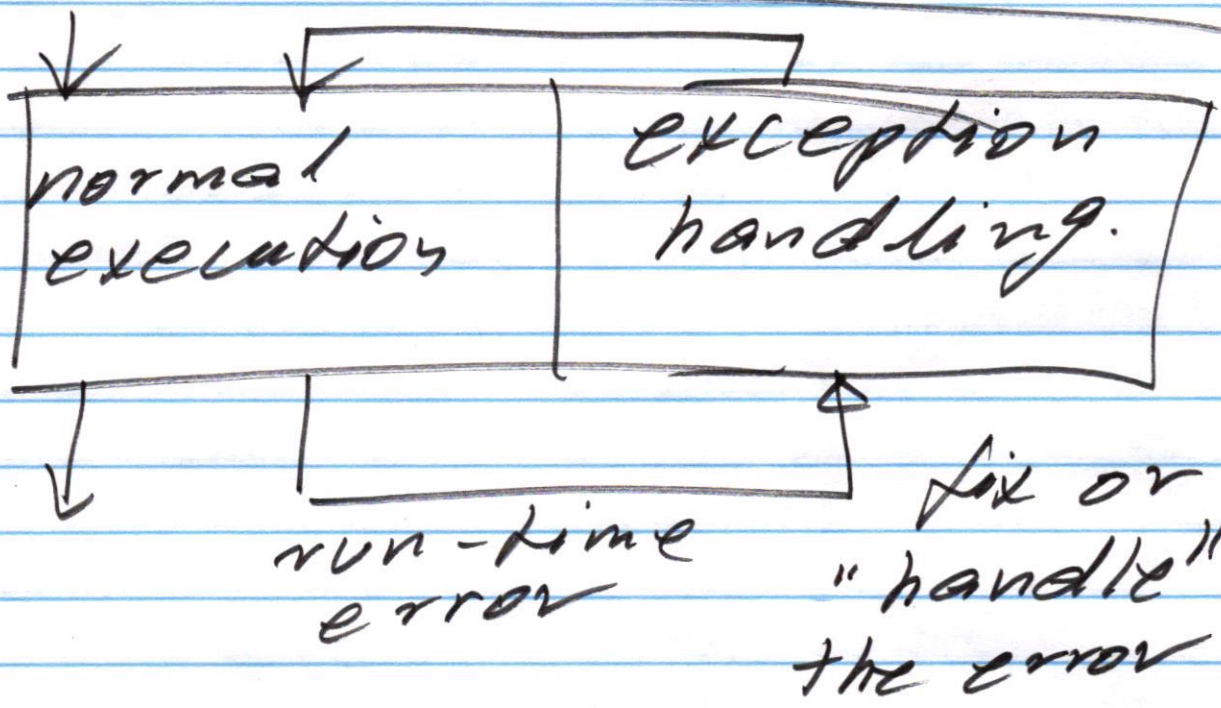

cost of testing
reliability
Testing is very expensive
perfect reliability

By design

What to do when a software component "behaves" incorrectly?

1. run-time error
   - system run-time error.
   - user defined run-time error
2. component produces incorrect output/outcome

error-handling mechanisms



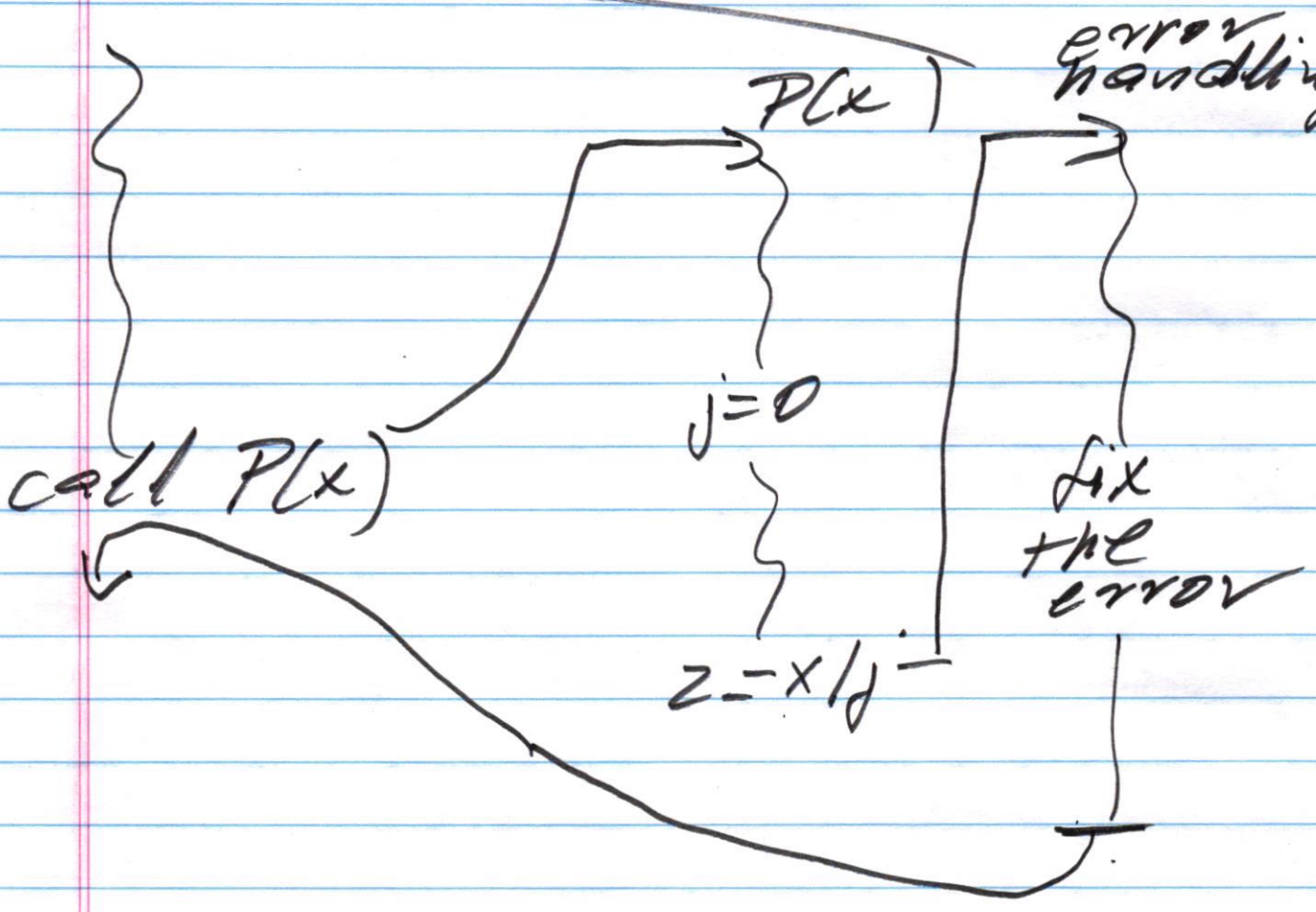
exceptions

* division by zero
* overflow
* . . .

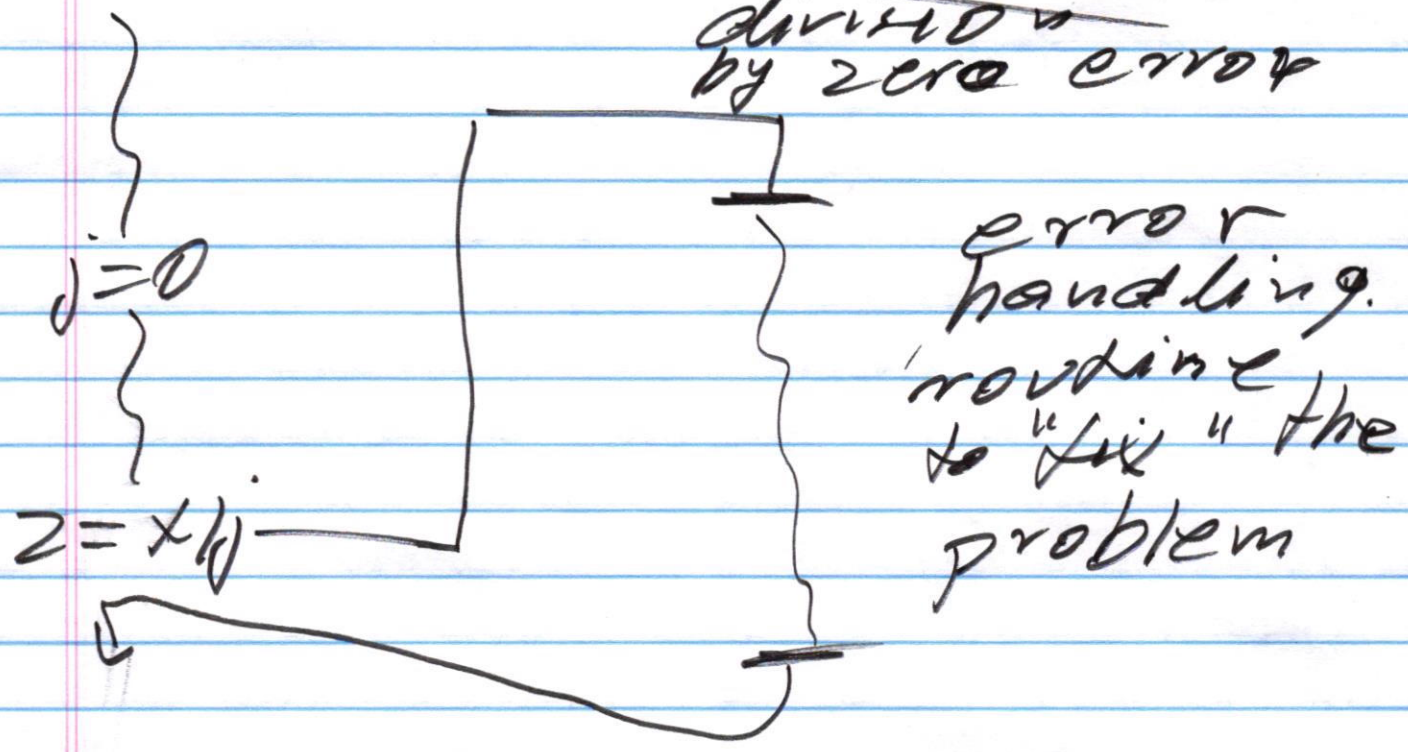
* list is full
* list is empty

} user defined exceptions

Termination model

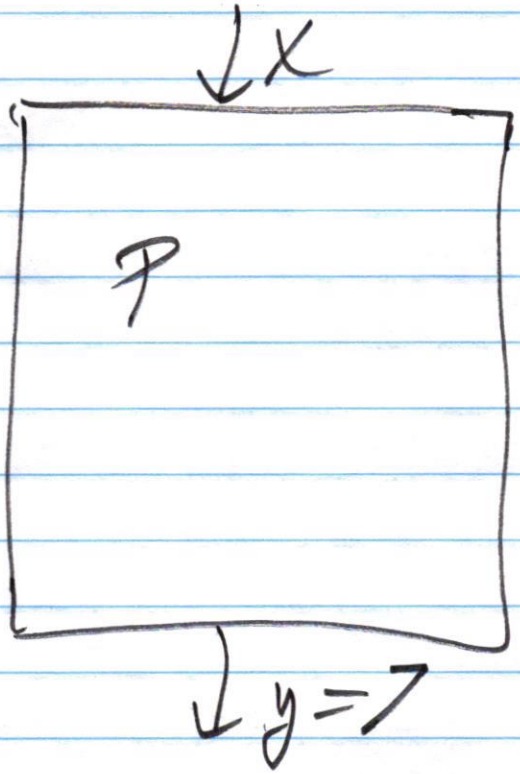


Resumption model



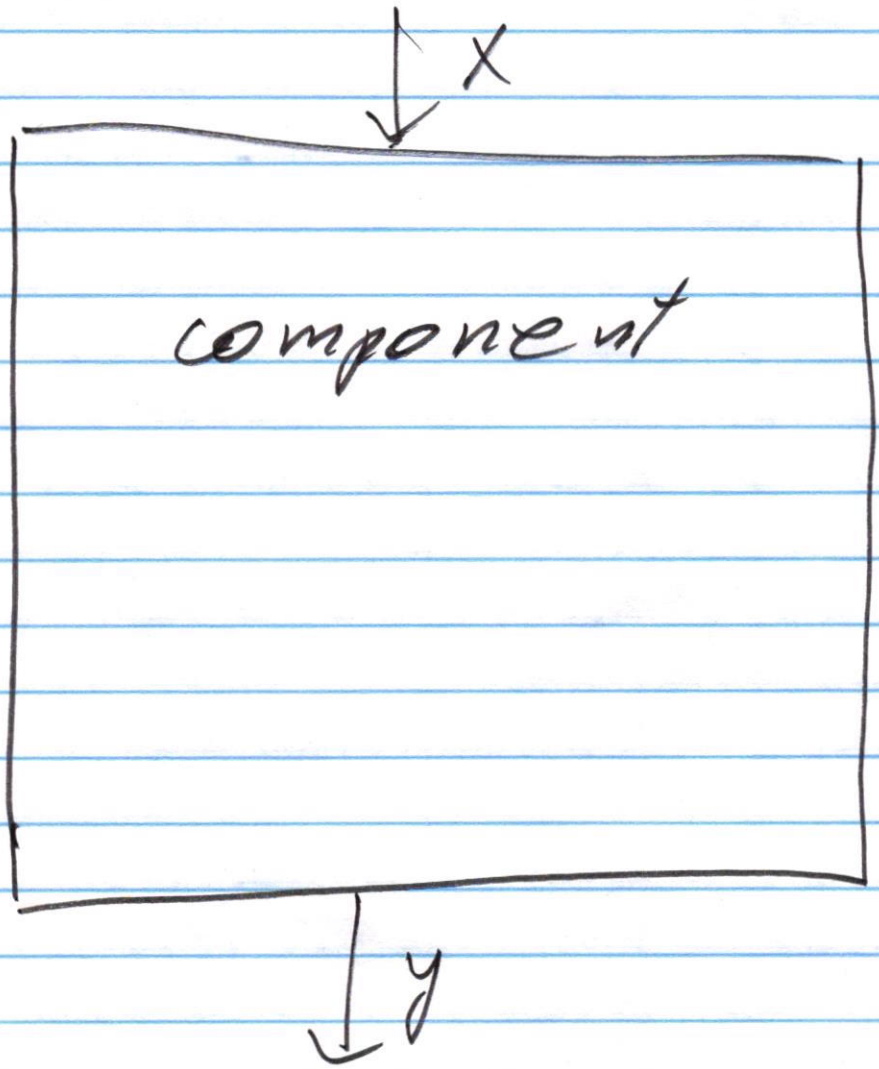
There is no run-time error

component produces incorrect output



but the correct output should be

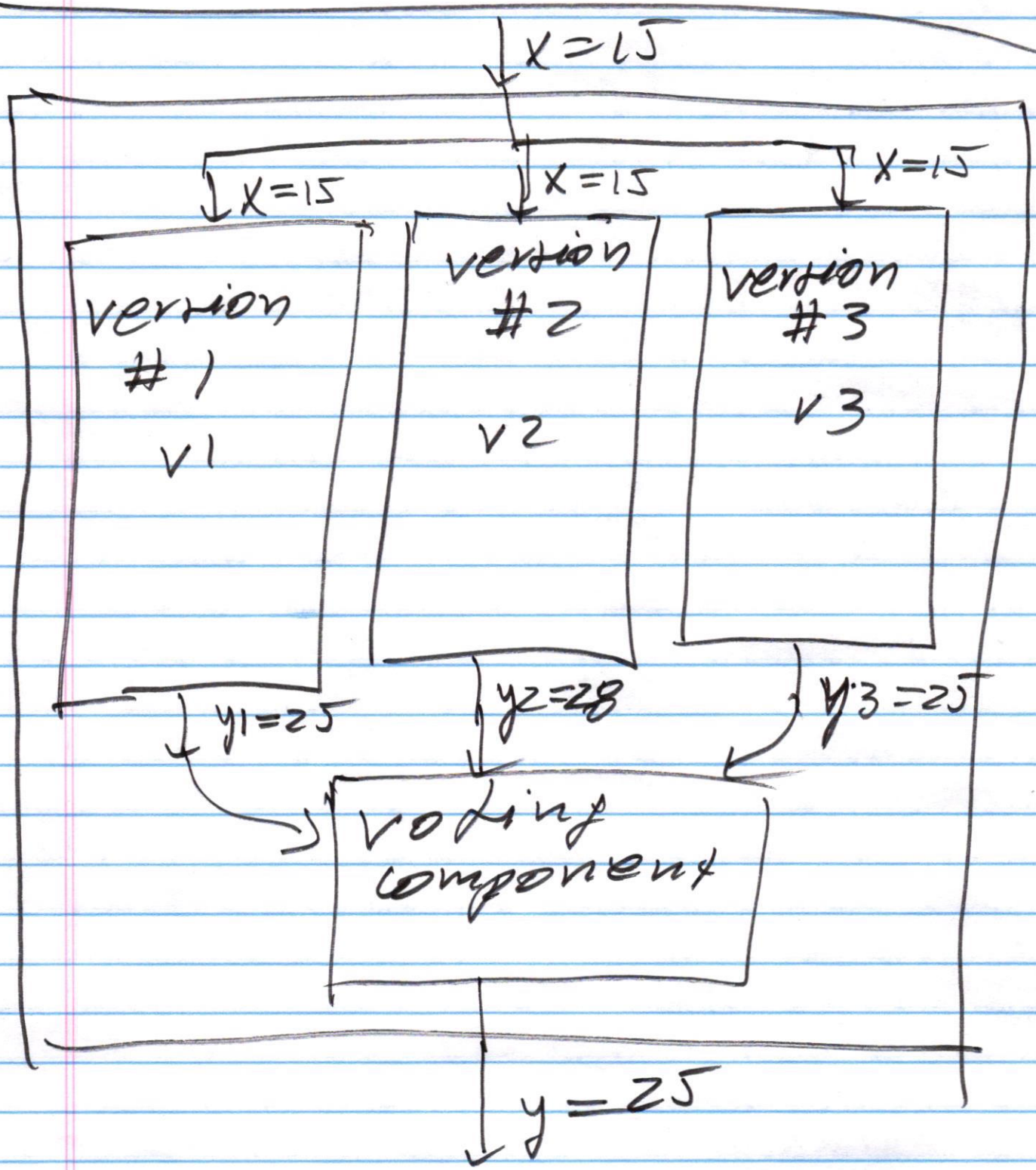
~~y=~~ $y=\bar{7}$


x
component
y

Fault-tolerant architecture

1. N-version architecture
2. Recovery Block —n—
3. N-self checking —11—
4.
5. — —

N-version architecture

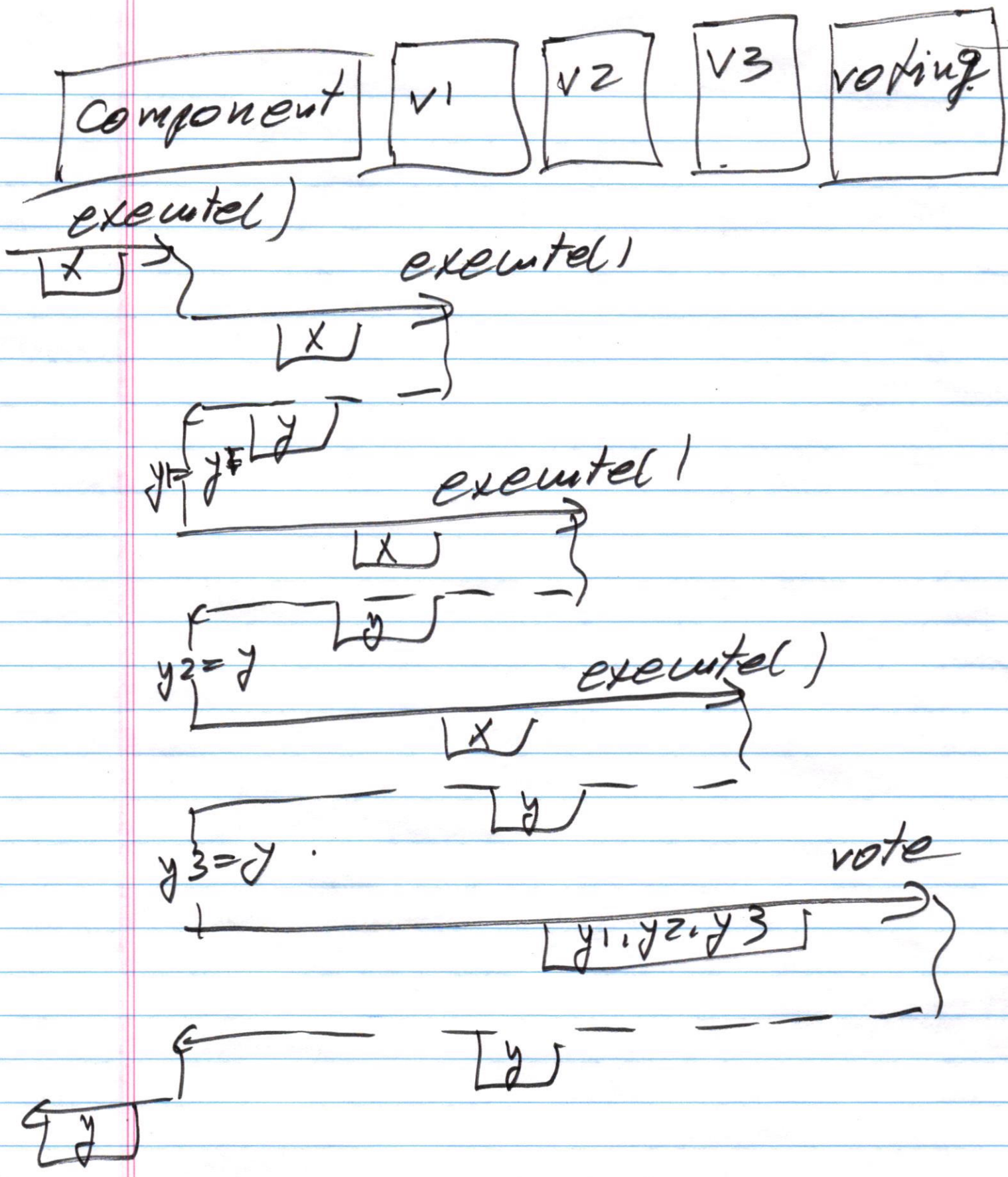


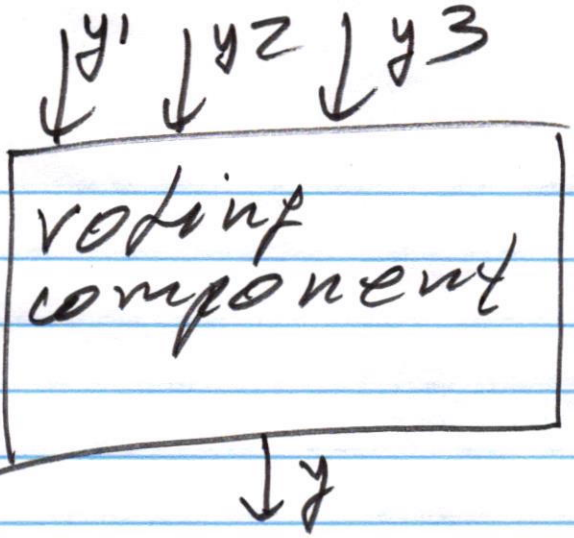
Different versions

1. different independent teams
2. different algorithms
3. — " — data structures
4. — " — languages
5. .
6. .
7. .

---

each version will have different defects


Component
V1
V2
V3
voting
execute()
x
execute()
x
y
y1=y
execute()
x
y
y2=y
execute()
x
y
y3=y
vote
y1, y2, y3
y
y



Input: y_1, y_2, y_3

output: y.

```
if (y1 == y2) return y1
if (y1 == y3) return y1
if (y2 == y3) return y2
```

→ all y_i are different

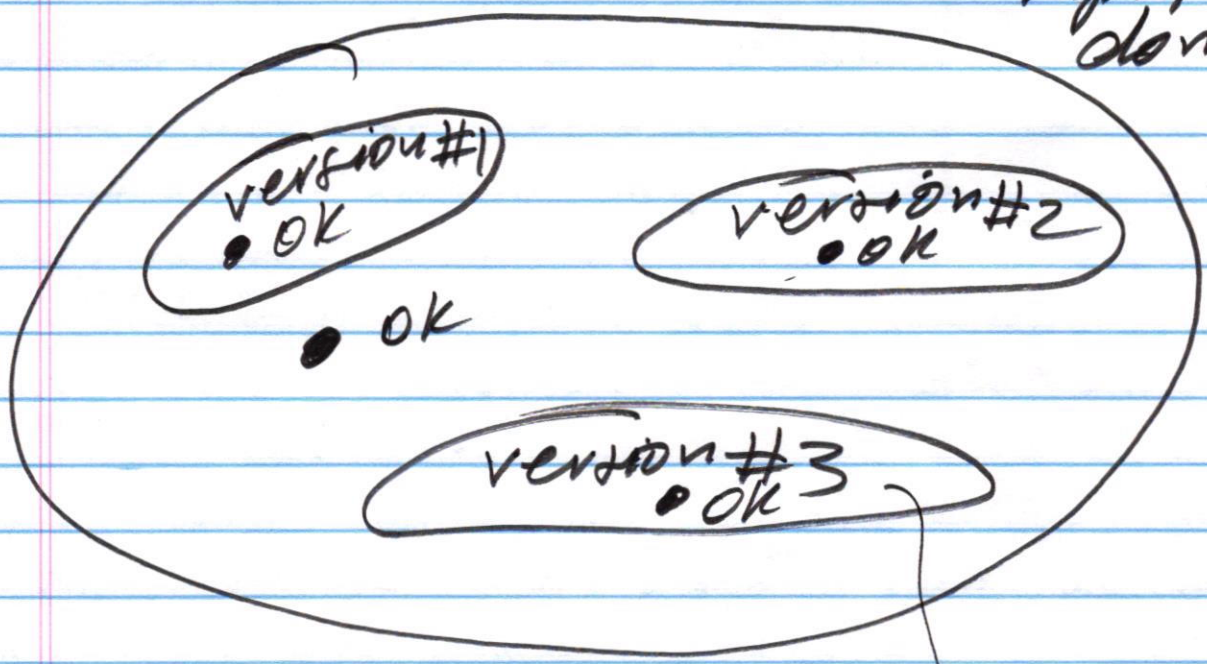
(I) y = randomly select y from y_1, y_2, y_3

or

(II) y = select y_i from "trusted" version

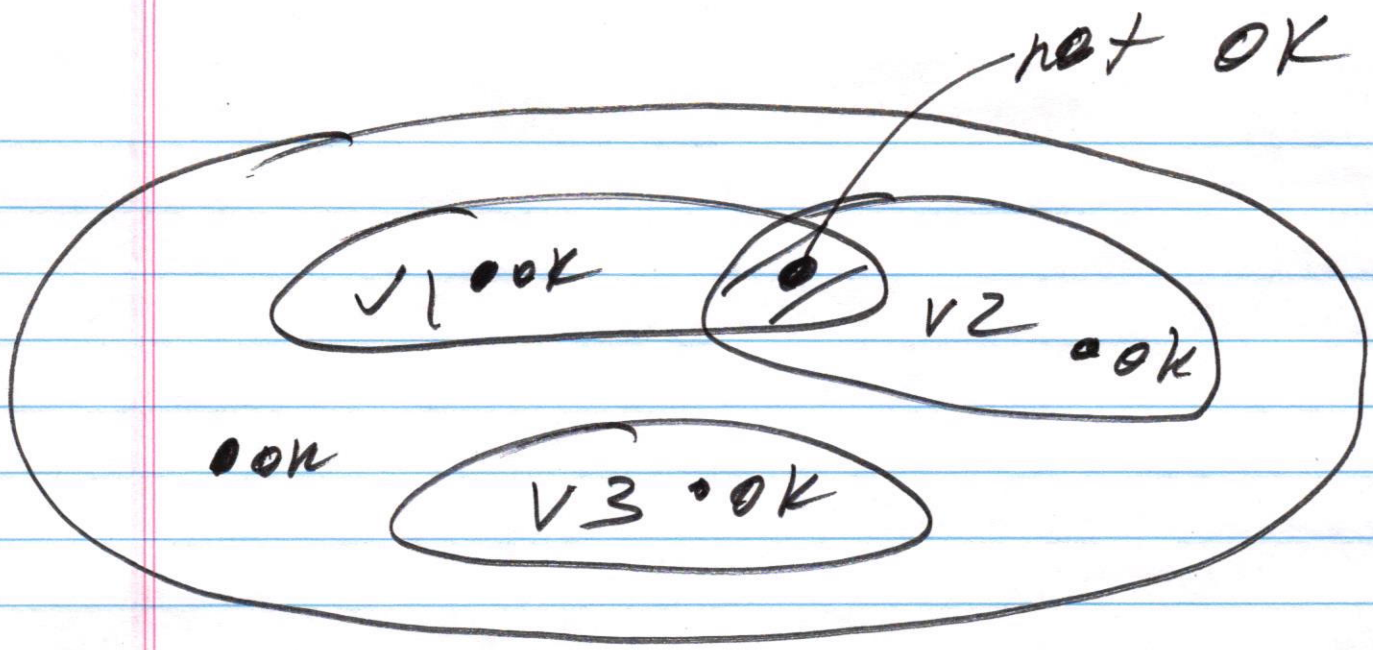
return y.

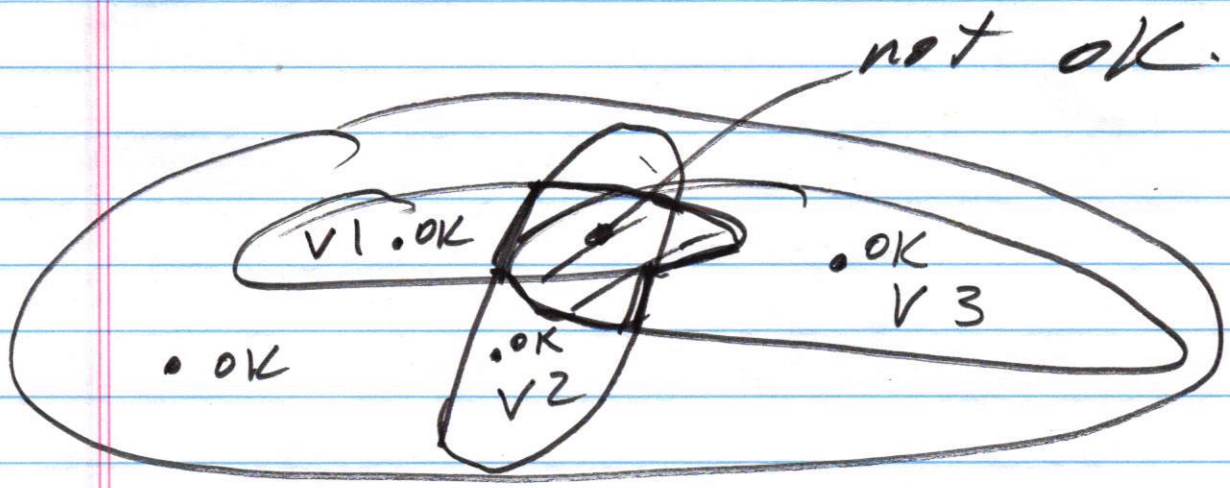
ideal situation



perfectly reliable component

a set of inputs on which version #3 produces incorrect output


not OK
V1 .OK
V2
.OK
.OK
V3 .OK


not OK.
V1 .OK
.OK
V3
.OK
V2
.OK

identify "critical" components.

---

use this architecture for the "critical" components that require "high" reliability

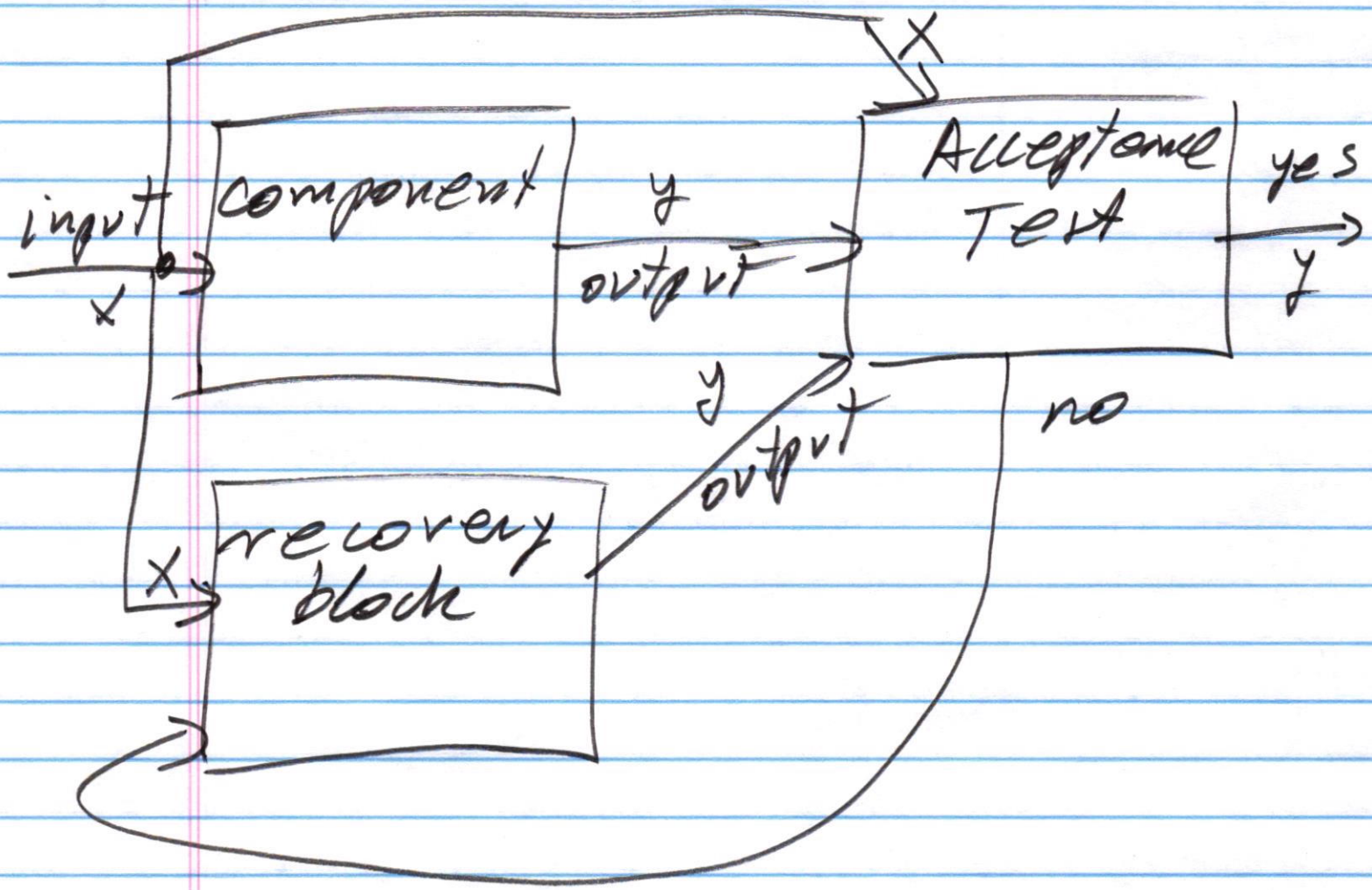
Advantages

- improved reliability / security.
- improved fault-tolerance.
- relatively easy to implement
- voting is very easy to implement
- versions can be executed in parallel.

Disadv

- extra cost in developing versions
- increased maintenance cost
- performance degradation if versions can only be executed sequentially

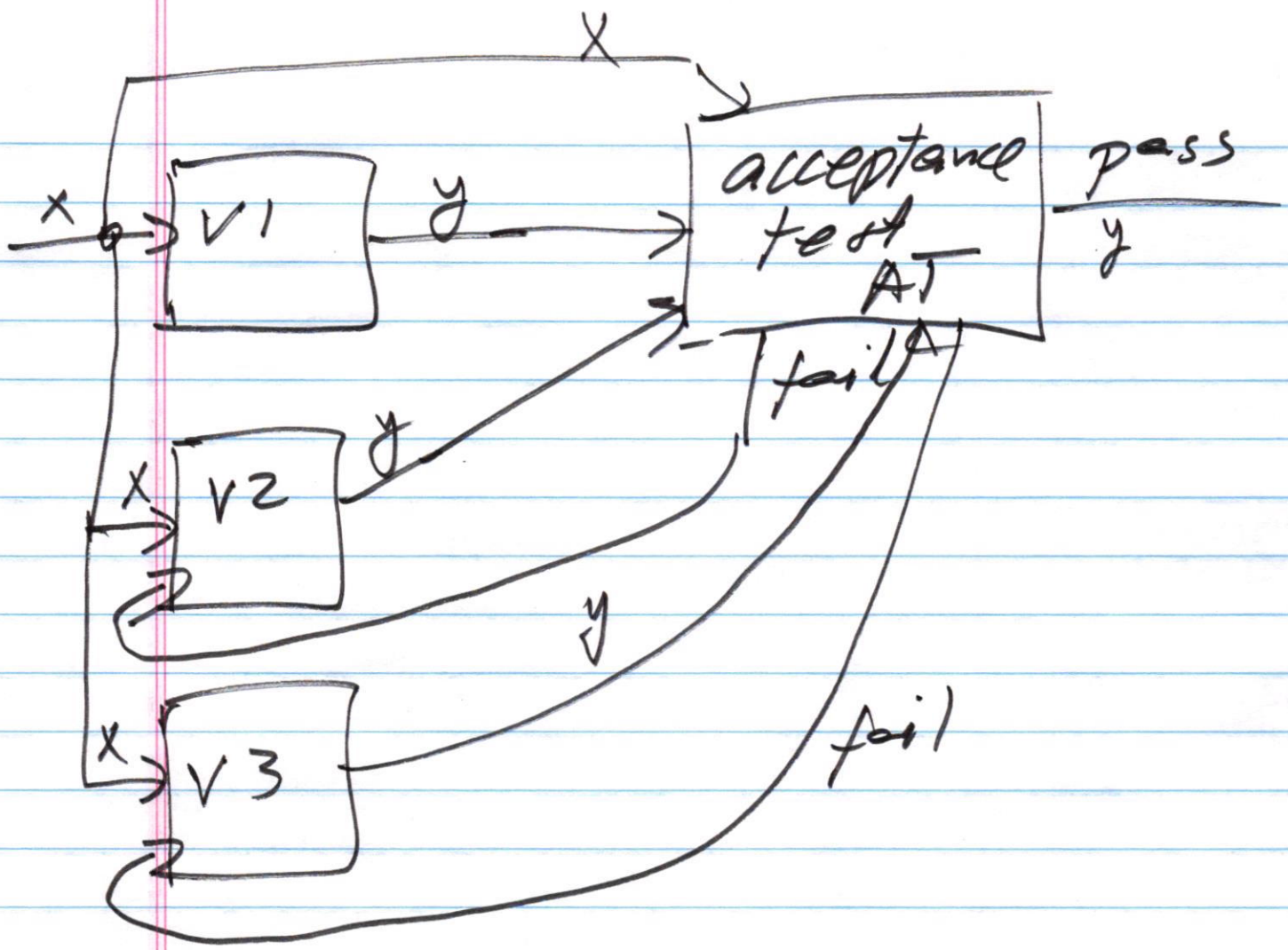
2. Recovery Block Architecture

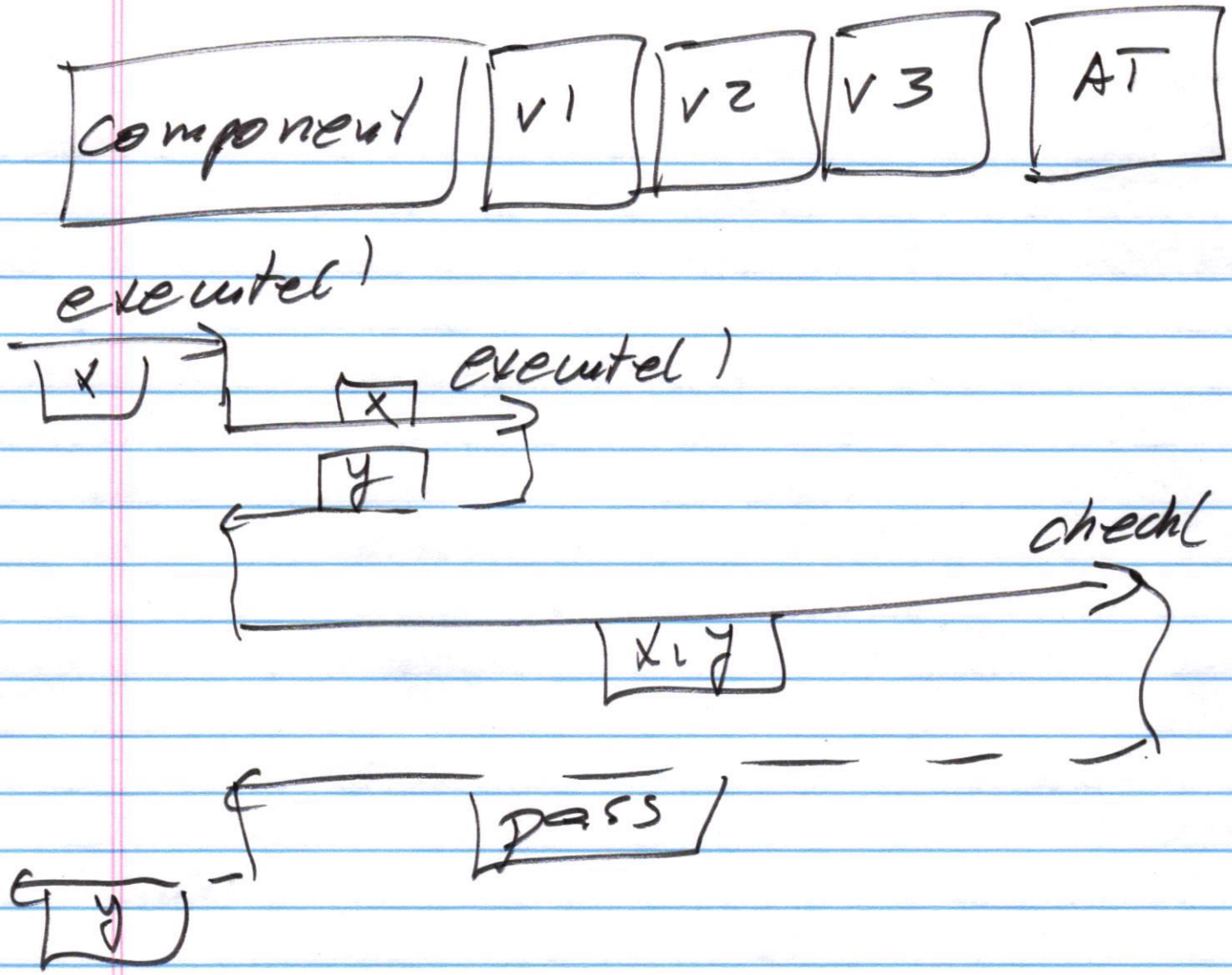


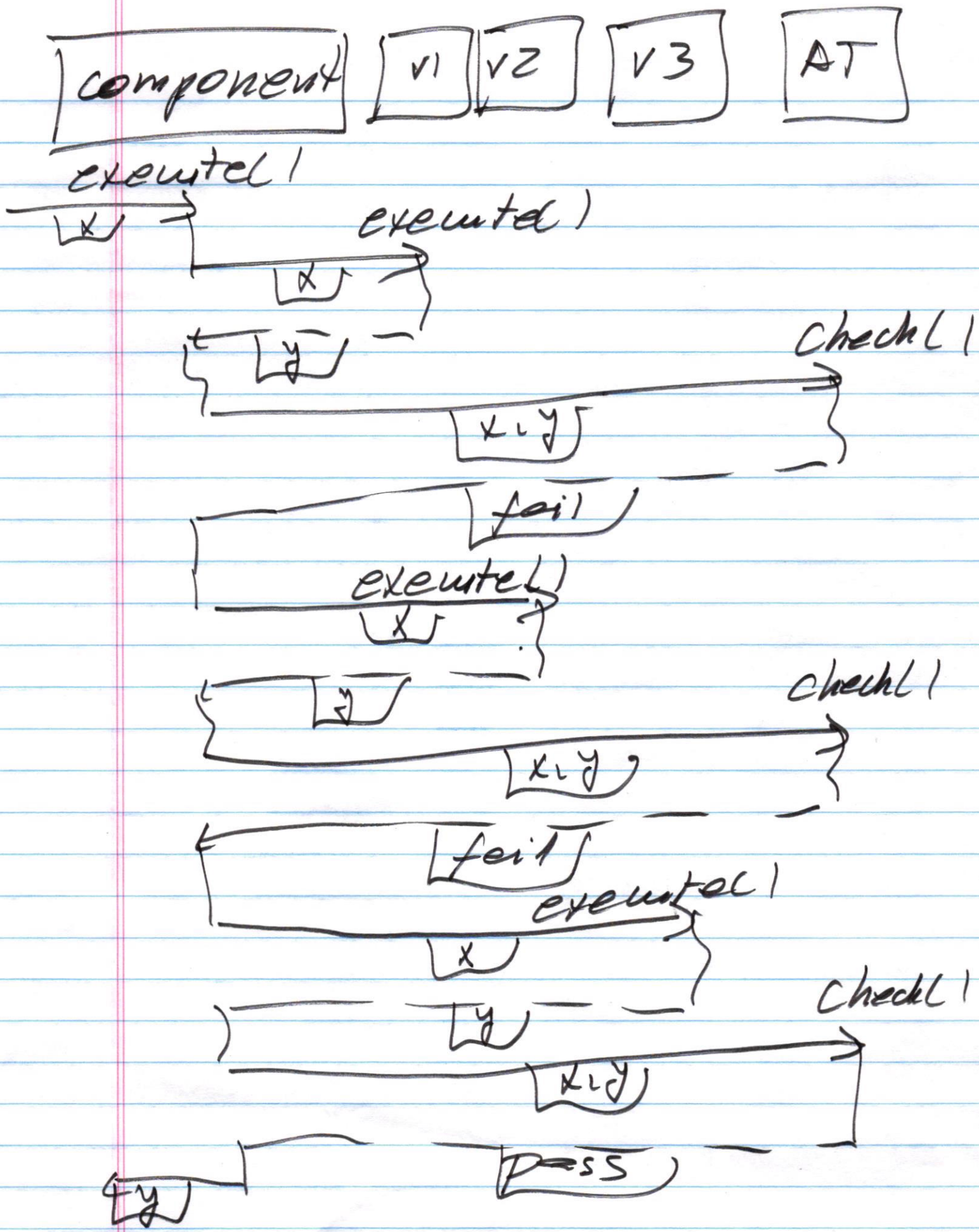
Recovery Block

(a) another version

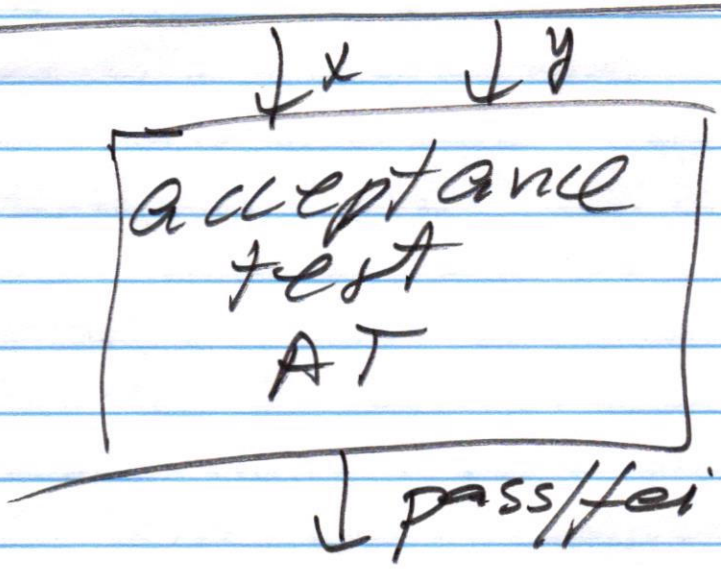
(b) old but "reliable" version of the component which is slow


x
x
V1
y
acceptance test
AT
pass
y
fail
y
x
V2
y
x
V3
fail


component
V1
V2
V3
AT
execute()
x
execute()
x
y
check()
x, y
pass
y

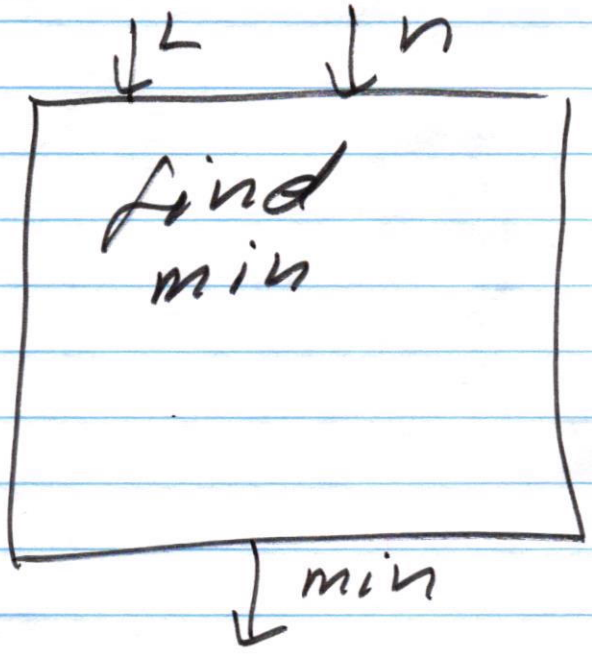

component
V1
V2
V3
AT
execute()
x
execute()
x
y
check()
x, y
fail
execute()
x
y
check()
x, y
fail
execute()
x
y
check()
x, y
pass
y

Acceptance Test

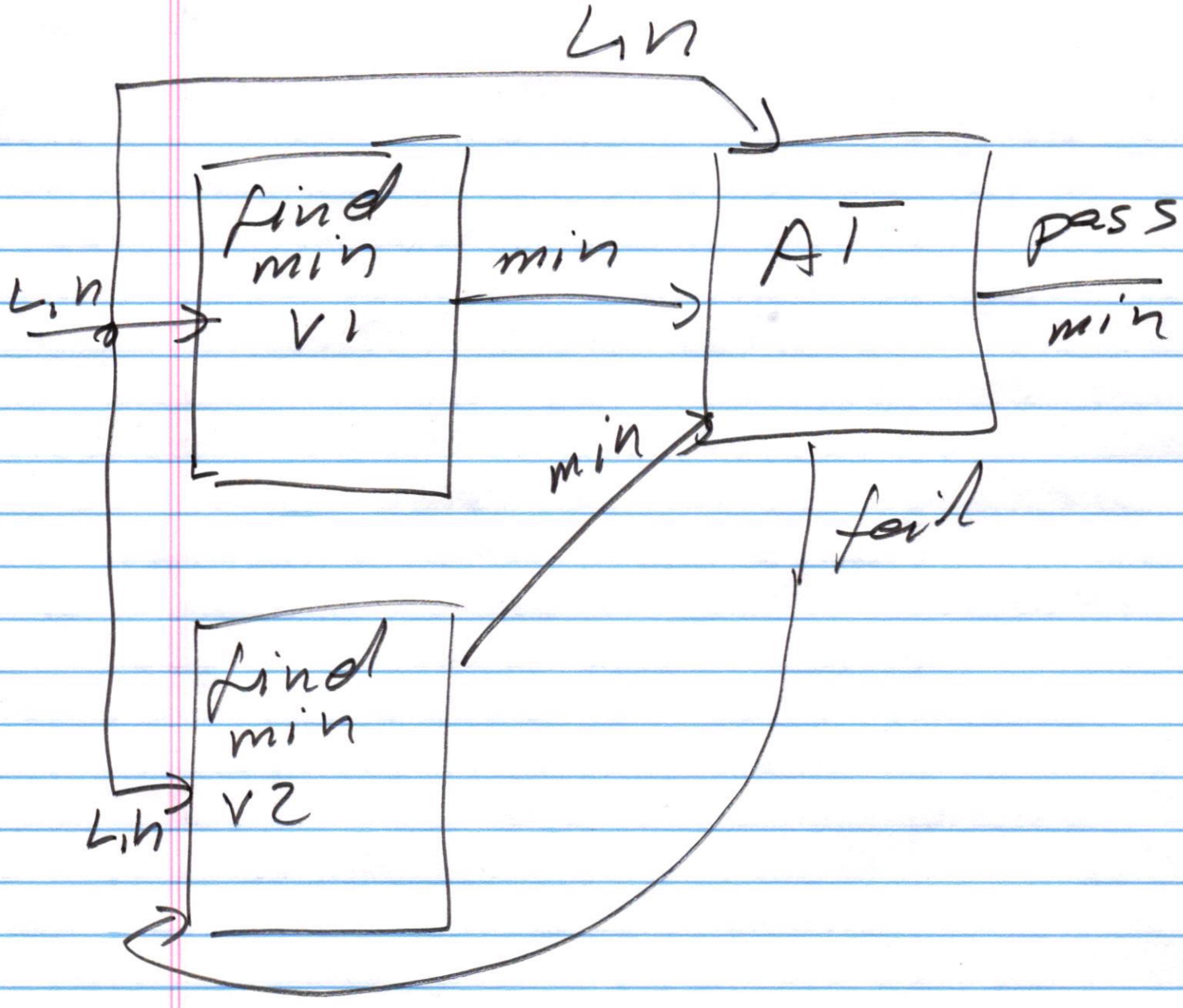


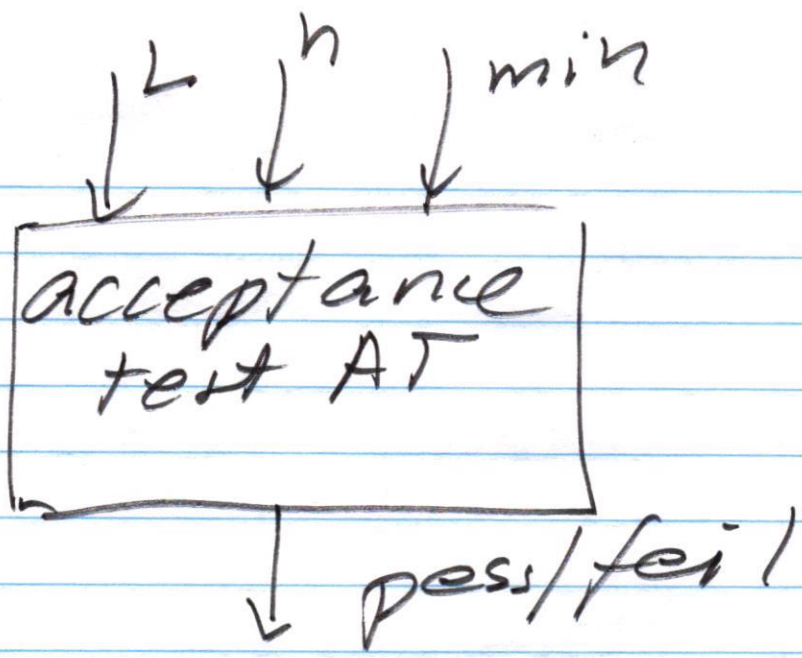
1. to determine if output y is "acceptable"
2. acceptance test must be very fast
   it cannot be another version of the component.
3. it may be difficult to construct "reliable" and fast acceptance test.

a component computes the minimum element on the list of integers



L: list of integers.
n: # of elements on the list
min: output minimum element in L


L, n
find min V1
min
AT
pass
min
L, n
min
fail
find min V2
L, n



AT: for i=1 to n do
    if (min > L[i]) return fail
    endfor
    return pass

L: 4, 2, 7, 8 } input
n: 4

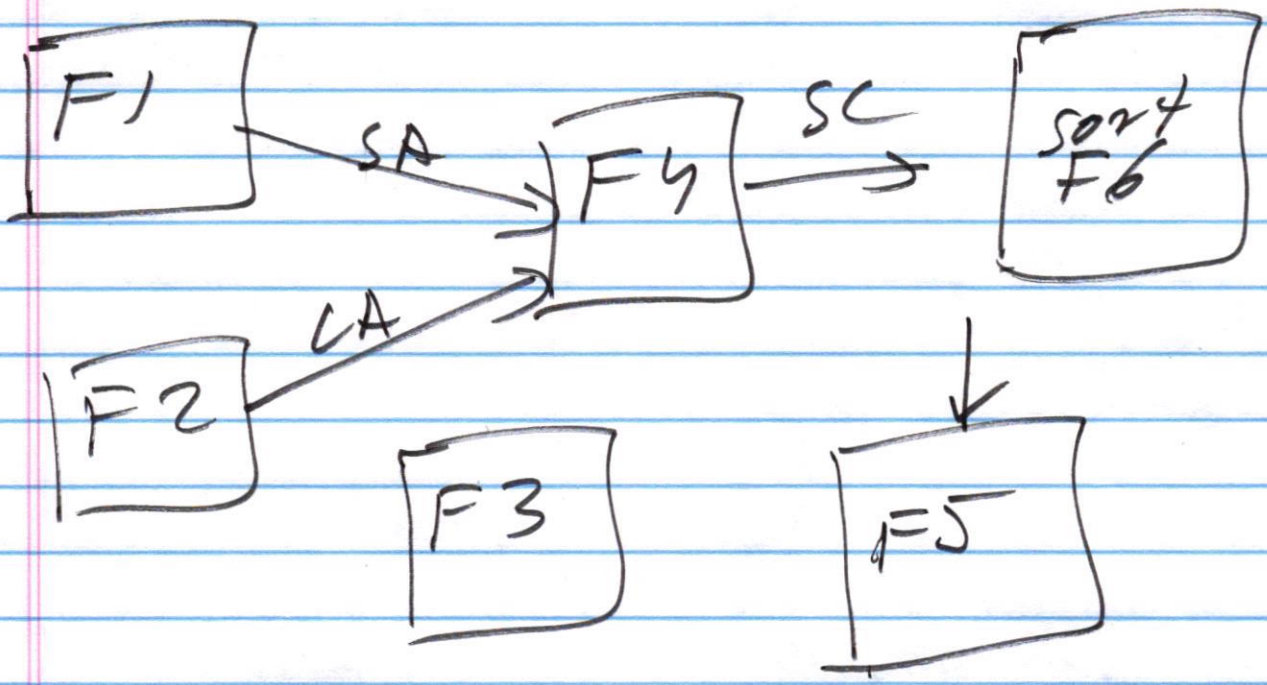
(a) min = 2 ⇒ pass ok.
(b) min = 3 ⇒ fail ok.
(c) min = 1 ⇒ pass not ok.

Homework #3

Problem #1

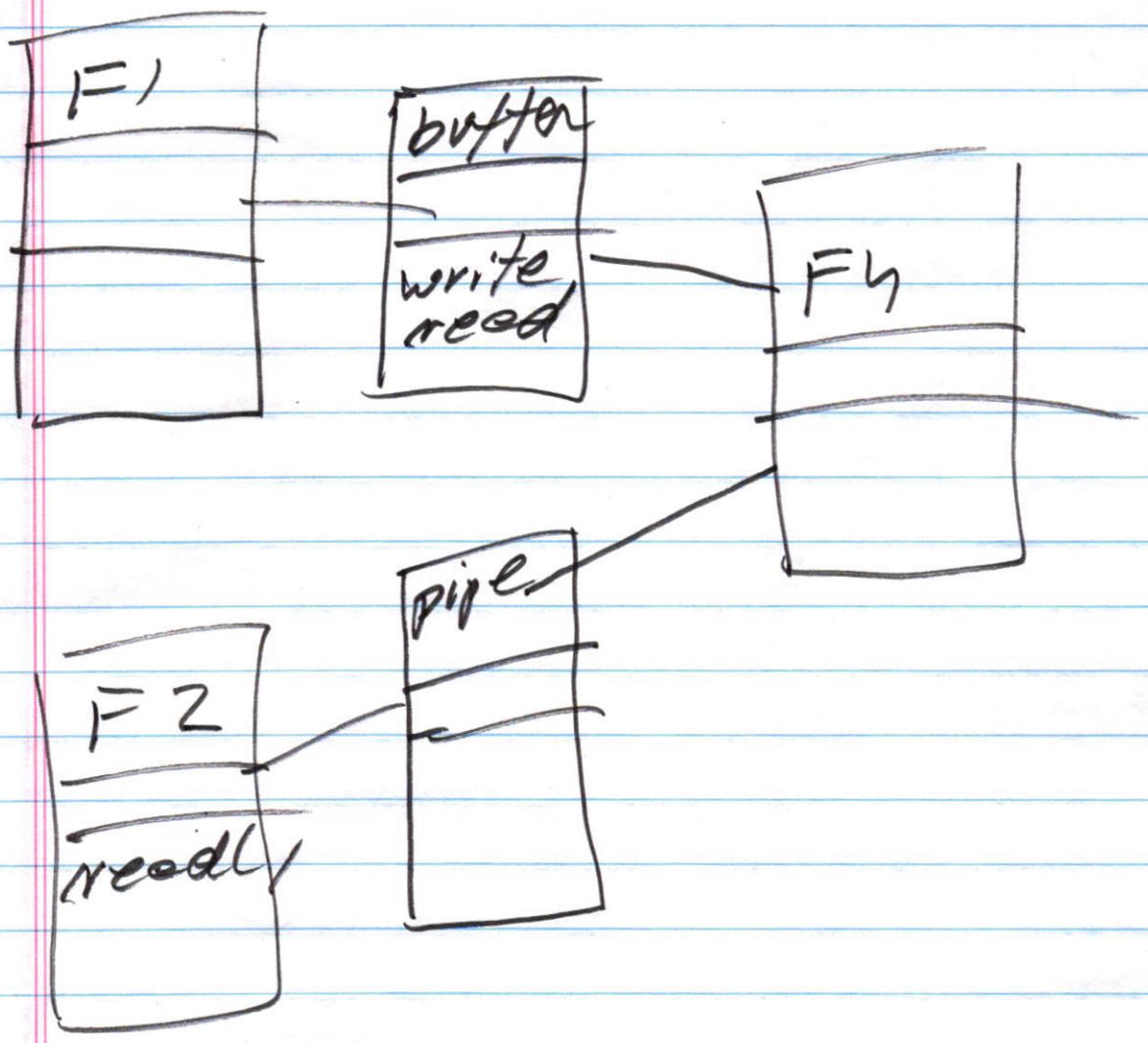
Pipes and Filters.

Part A



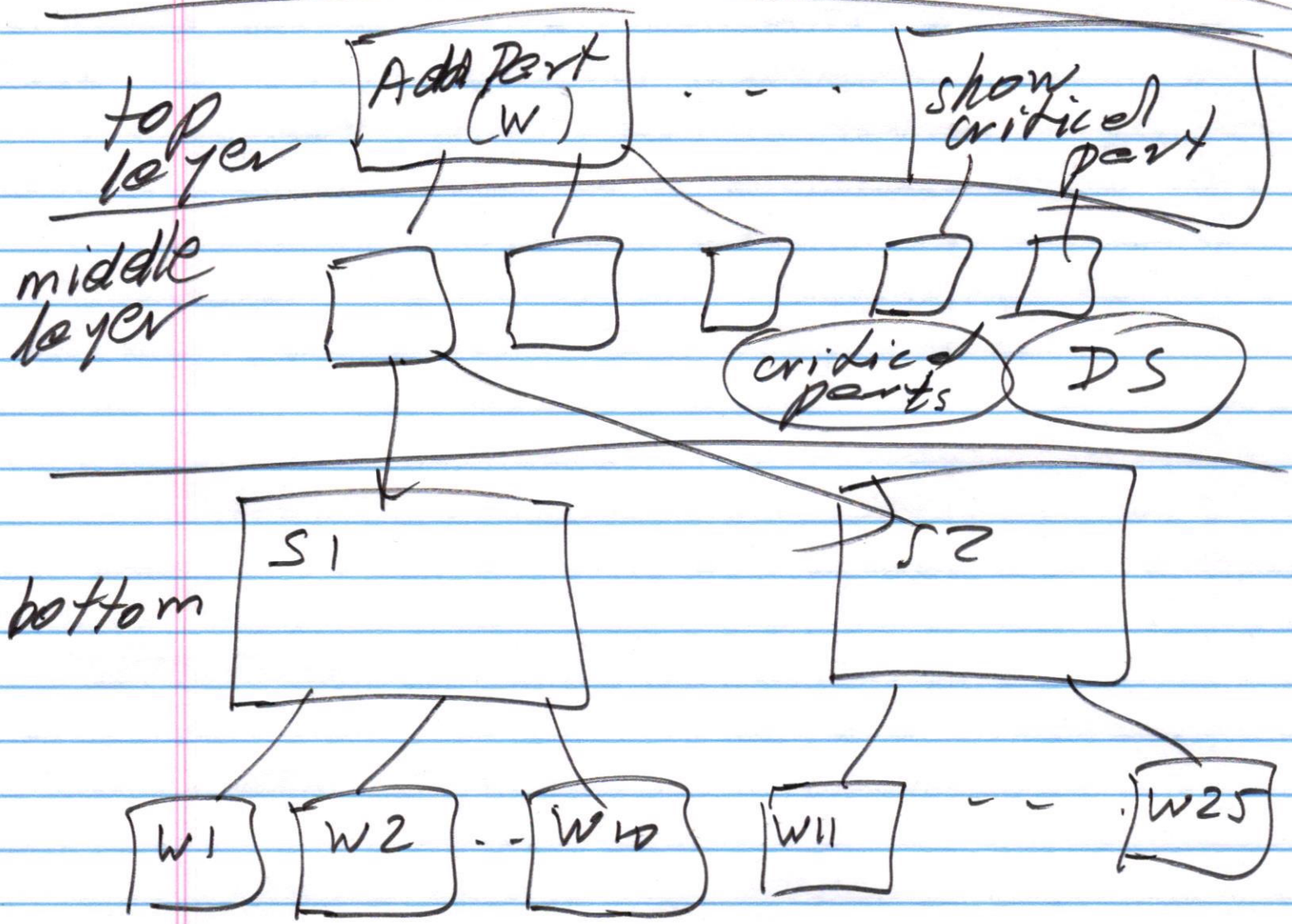
Part B

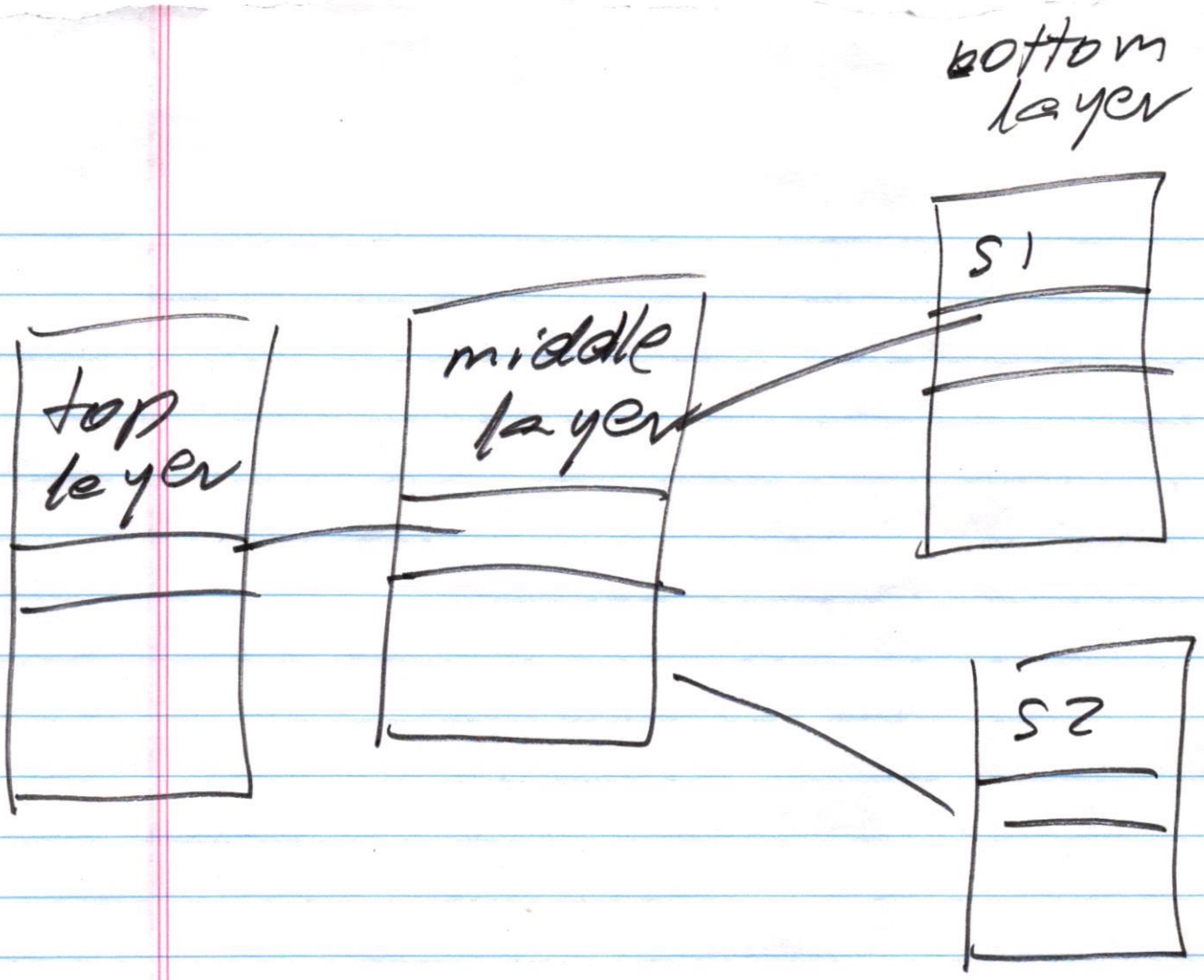
class diagram



Problem #2

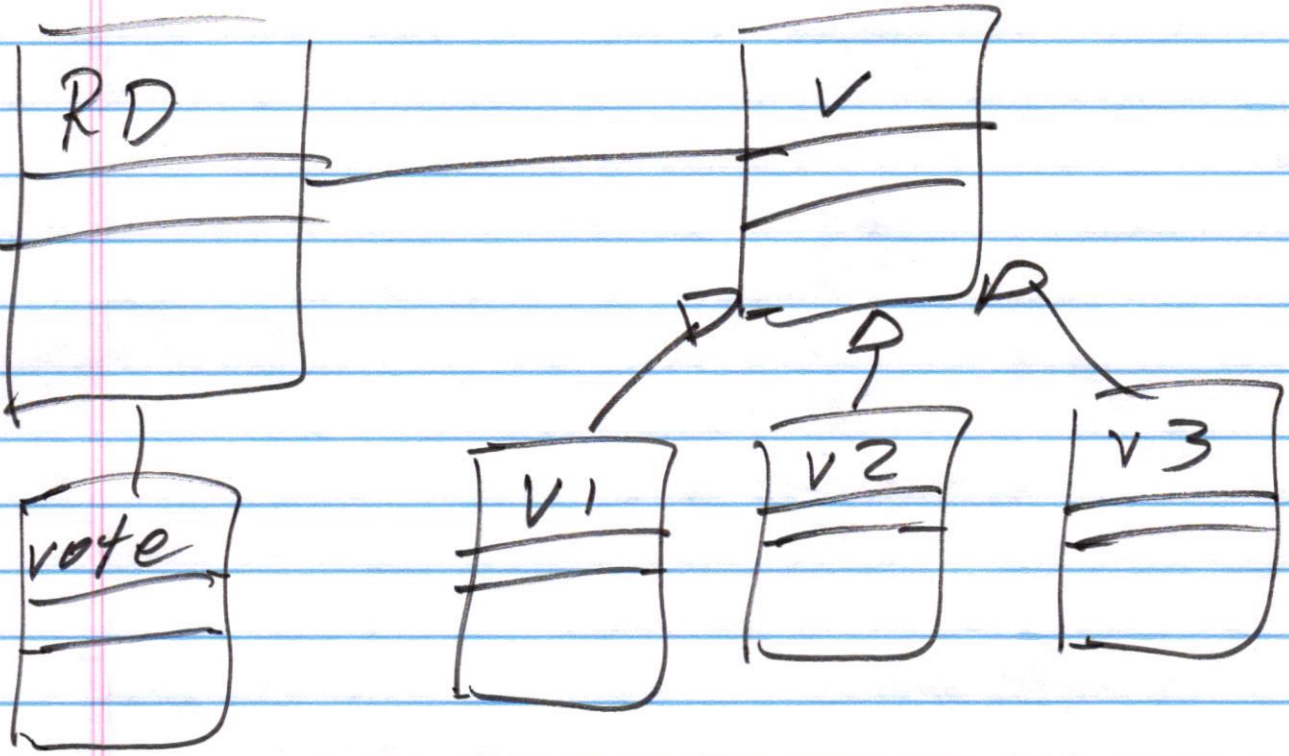
strict Layered architectu-



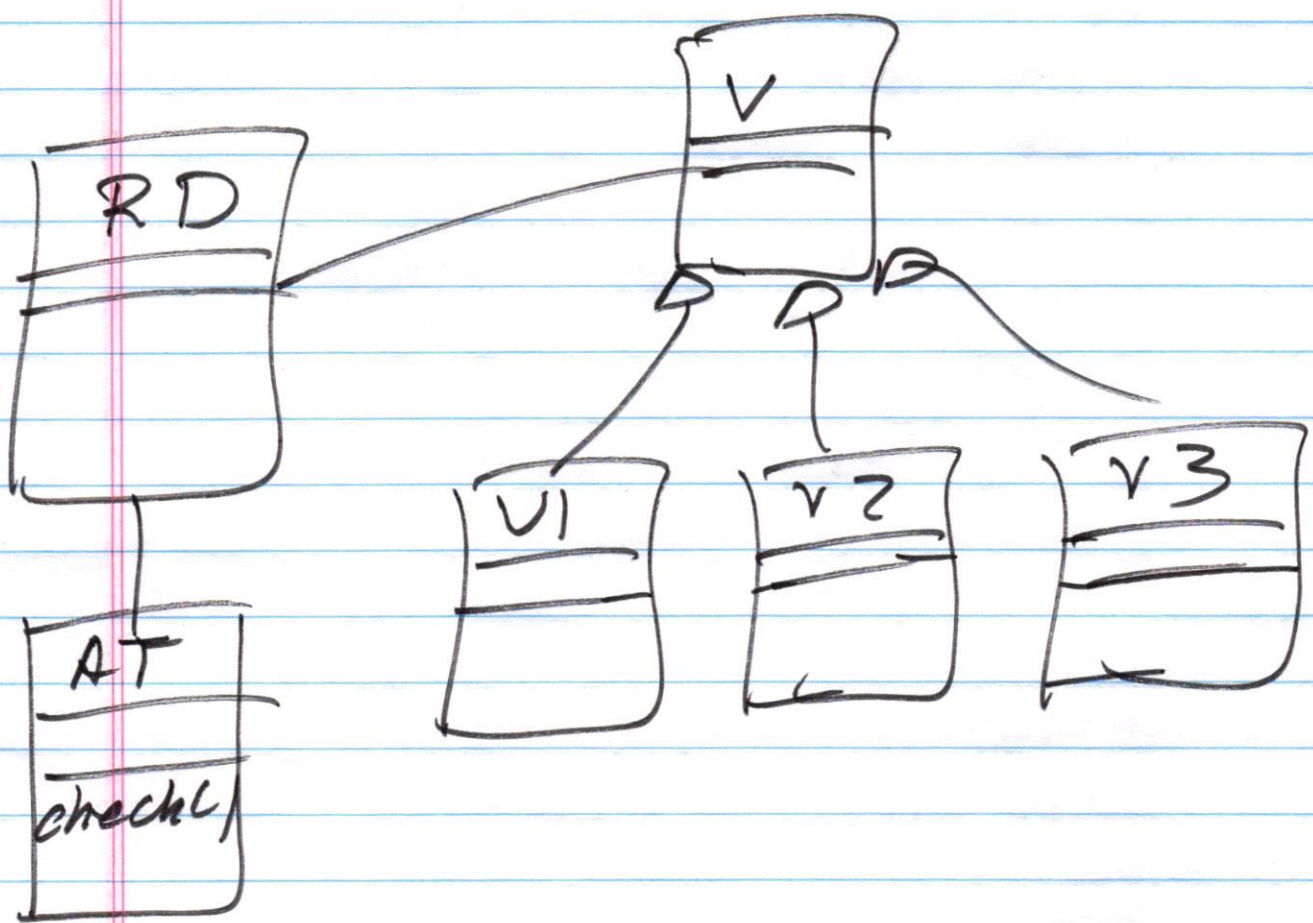

bottom layer
S1
middle layer
top layer
S2

Problem #3

N-version architecture.



(2) Recovery Block Architecture.



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:

top

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.

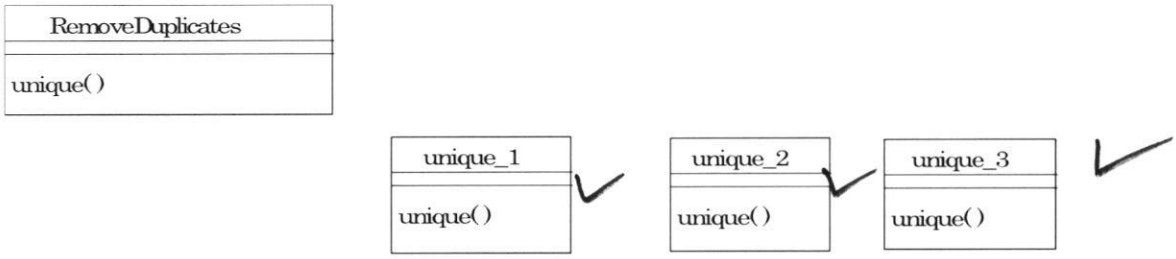
For example, for the following input:

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

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.