Foundation for Advancement of Science & Technology

Software design & Architecture

s/w engineering cycle: Inc RE A&D I T D/D O-M O/R

nature: s/w is logical entity unlike hardware.

Complexity → nature
- → large in size
- → people: salaries
- negative stakeholders: interested in failure of project.

] professional industry strength s/w

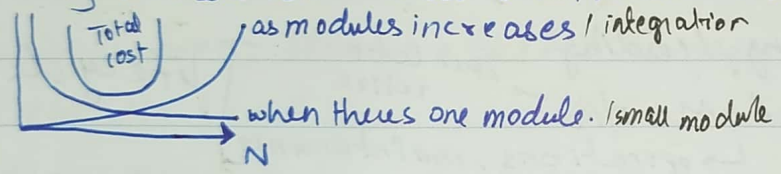
Design concepts

1- Abstraction = it is selective examination. You focus on imp things.
- → very imp skill for designers and helps in complexity.

2- Information hiding ~~/encapsulation~~: hide data structure & our algorithm.

3- Modularity. We split s/w into different components.

every module talks to all other modules except itself $\frac{N(n-1)}{2}$



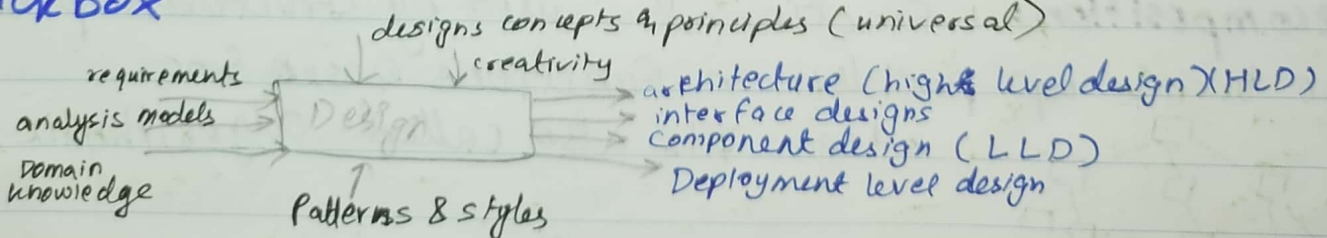
4- Functional independance - highly cohesive & less coupling.
- → coupling ↓ - degree of connectedness. minimize it.
- → cohesion ↑ - degree of single mindedness: does only 1 thing conceptually

5- Refactoring: improve internal structure without modifying external behaviour. remove redundant code.

6- (step-wise) Refinement: define steps gradually.

7- Patterns: good designers use patterns. We can reuse aspects of designs.

Blackbox



Foundation for Advancement of Science & Technology

science is foundation of engineering.
science → provides theory.
engineering → more practical
↳ applied science

Failure in s/ws. 1- largest airport in US (DIA) operations were delayed more than a year due to problems in baggage handling system.
2- Toyota had to recall more than 600,000 hybrid vehicles due to issues with hybrid system. Nissan had to recall more than 3 million cars due to problems with passenger classification system
3- Washington (state) continued for 10-13 years. Poison sentence reduction time calculation system had problem. More than 3000 prisoners got released earlier than their actual time.

* software → Program/Instructions
↳ Associated Documentations
↳ data (to run, test, configure the s/w & data generated by the software)

Implementation / Programming / coding.

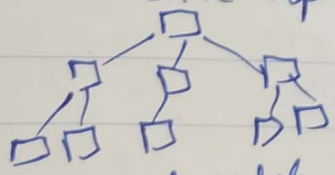
Inc. R E A : D I T D/D O-M O/R → obsolete retire
↳ operations, maintenance
lifecycle

what to design | delivery deployment | how to design
prob space | solution space

A:D
modeling phases.

Lecture 3

Decomposition: 1- Functional: used by structured paradigm. we break up problem in functions & sub func.
↳ eg. Its also called algorithmic



object oriented.

2- OO decomposition: decomposition in classes & objs.

if us use C lang you use functional
" " java " " OOP

Foundation for Advancement of Science & Technology

used to deal with complexity ↖ behaviour ↑ storing attributes & ↑function

OOP Paradigm: Abstraction, encapsulation, inheritance, polymorphism

*Abstraction: The act of creating classes is called classification.
Classes are abstractions of real world objects.

* Inheritance is most unique. Unique selling point.
* polymorphism: allows using common name for different functionality.
↳ any lang to be considered OOP must give option of inheritance.

- OOA (analysis) → OOD (design) → OOP
  ↳ modelling phases

Modling = are abstractions
↓not unique to SE. architects & engineers make models. eg. computer models for simulation, Making a sketch for painting.
- Benefit: to get general idea. Its cheap & time saving. To test our ideas or implement them. A way of understanding.
- primary purpose: improve our understanding.
- Used to convey to stakeholders our idea. (communicate)
- easy to modify models.
- whenever a model is created you use syntax or language.
↳ **UML**: unified modling lang.
- There existed many languages like OML but they all has small differences so they unified it. we'll study UML 2.x.
- Theres a group called OMG: obj management gop. owns UML specifications.

UML Models:

1- Class-based: class diagrams (ACD, DCD), objects diagrams.

2- State-based: state diagrams.

Foundation for Advancement of Science & Technology

2, 3 reveal dynamic info.

regular → SL

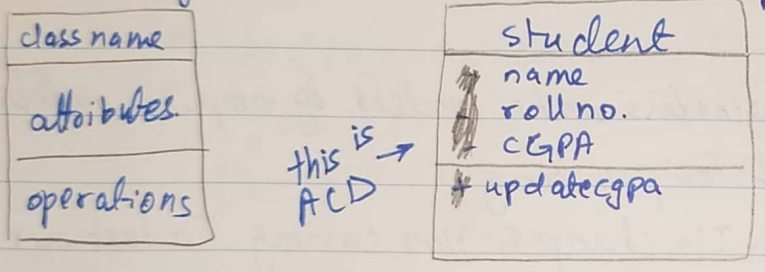
3- Interaction: activity diagram, swimlane, Usecase, UCD

sequence → SSD
→ DLSD.

- ClassD. = don't reveal time. Are static.
- why we create diff models? = one model doesn't give all info from all ~~aspects~~ perspective. We need info from all perspective.

Class Diagram → starting point of modelling.

- Rectangular box with 3 compartments.



→ appear in ACD
↓
doesn't have aspects of solution space.

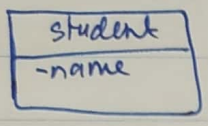
← features = 4

* DCD has more info than ACD. DCD = ACD ++
* when you start DCD, you start with ACD & then add info. Datastructures: hashmaps, trees.
* Design is in sol. space. It could have extra classes that is not in ACD because we need it for implementation.
* UI related classes not in ACD or problem domain.
* Problem space, sol. space.
  ACD DCD.
* In sol. space we're worried about implementation
* - = for private, + : public, # protected] visibility
  ~ : packaged ⇒ fellow classes in same package can see it.

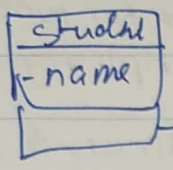
Conventions:- all class names start with capital word & written in center.
- names of attributes are ~~in~~ lower camel case except abbreviations eg CGPA. They're left aligned.

Bring pencil.

Foundation for Advancement of Science & Technology

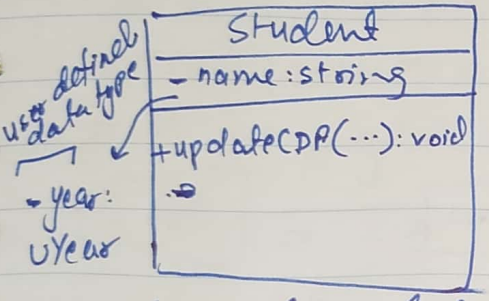


← why operations compartment not drawn?
it means we don't know its operations.



→ if this left empty then it means it has no operations

- attributes: -name: string
- operation: + updateCGPA(newCGPA: float): void ← it can be empty



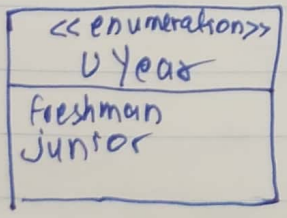
← This is DCD.
we're in sol. space.

*parameters can be in ACD.
* updateGPA() → we know about its parameters.
updateGPA → we don't know

- syntax for default val.: -CGPA: float = 0.0
- name: string [1] ← single valued.] → means every student can have only 1 name
  +rollno: ~~string~~ float [1] = 0.0
- phoneNumber: string [*]] every student can have 0 or more phone nos.

multiplicities:
1
2
5
5..17
* = 0..*
1..*

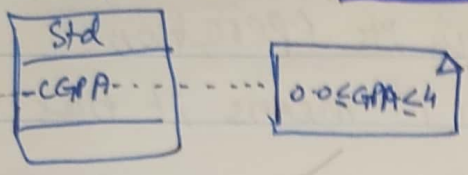
* if -name: string written then by default its [1].
* [0..1] ← std has 0 or 1 phone no. only
↑ this can be any no.
* we can specify 1 val or range. WE USE 2 DOTS in RANGE [0..1]



← UYear is a class & you made a variable of UYear in std class.

Foundation for Advancement of Science & Technology

* You can write comments.

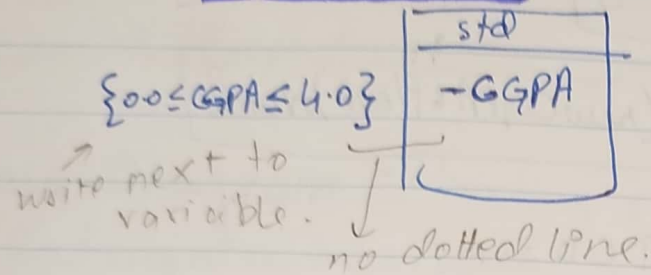


] to add comments.

() = used for func.
[] = multiplicity
{} = constraint.

doosri side pae banao.

To add constraints

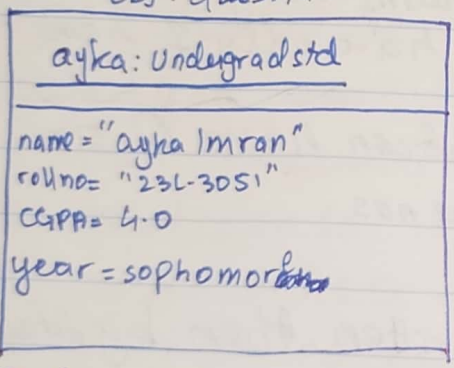


- all class names are singular nouns.
eg. Students ← 's WRONG.

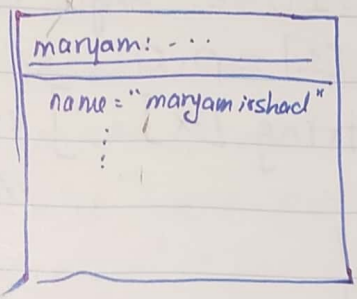
Object Diagram

- Obj are instances of classes.

obj : classname.

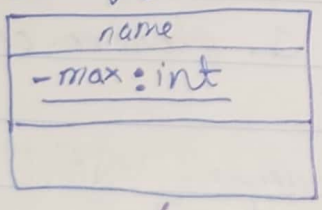


← underline



can be capital eg.

Static var.



← underline it & try to write it on top

* no. of attributes + method = features

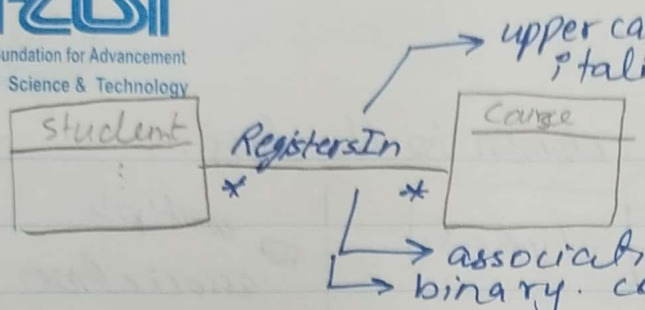
* -/percentage : float

derived attribute

↳ derived from CGPA.

percentage = (CGPA/4 * 100)

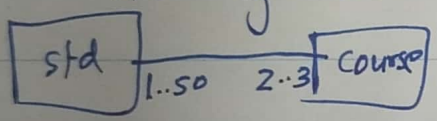
eg. circle
-radius : float -/circumference : float.
-area : float

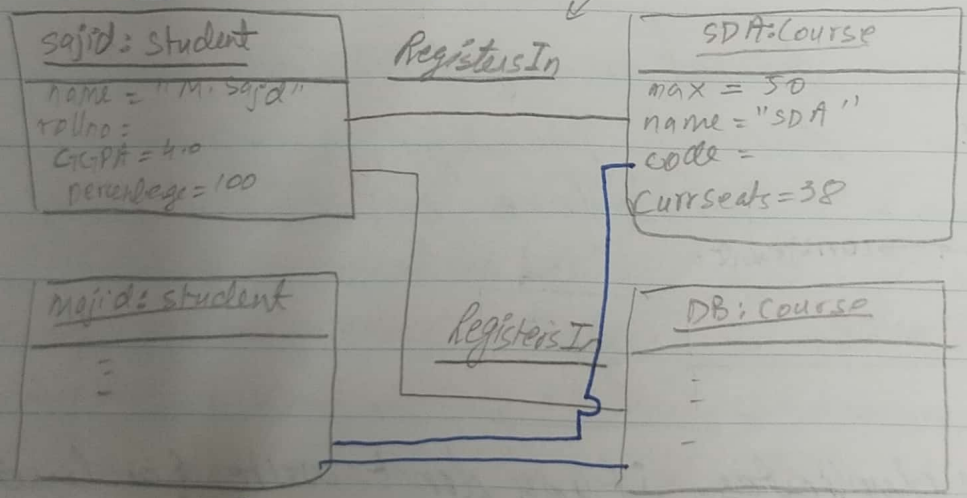


→ association.
→ binary. connects 2 classes

* name it such a way that its easy to read from left to right. Must be stright line. or like ⌐

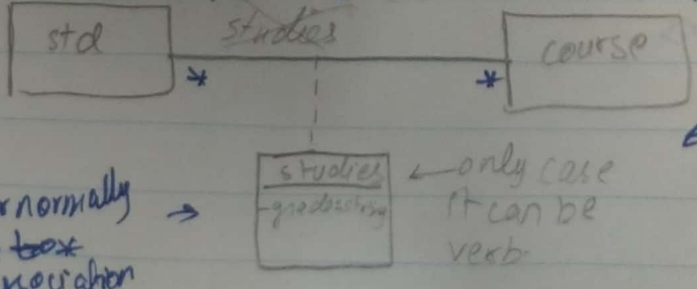
* multiplicity:
  - stand at student. ask 1 student can register in how many courses? many so write * in courses.
  - now stand on courses & ask ~~how many~~ course can enroll how many students? * multiple.
  * if no. given then write 2..3





* class diagram shows only association

* cardinality = count. = is on objects
* multiplicity is constraint on cardinality. on classes
  * majid didnot fulfill multiplicity as it does not have association with course.



appear normally in * to * association →

← only case it can be verb

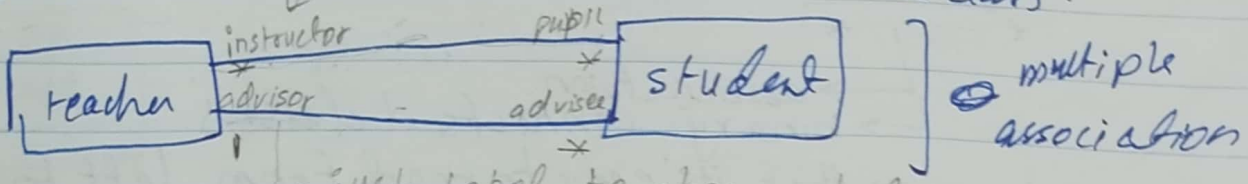
grade variable can be in which class? none. grade consists of combination of student & course so thats why another class of variable is made.
* remove "studies" on the line & make another class.

Foundation for Advancement of Science & Technology

* draw lines with scale.

not italic

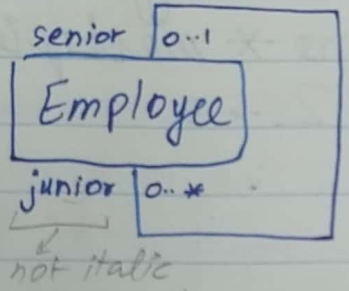
* attribute name need to be ~~common~~ unique to the class.



* advisor is a sudo attribut of student class. opposite class. student class can't have an attribute with same name advisor.

Uniary association

* no need to use rules. It will slow you down.



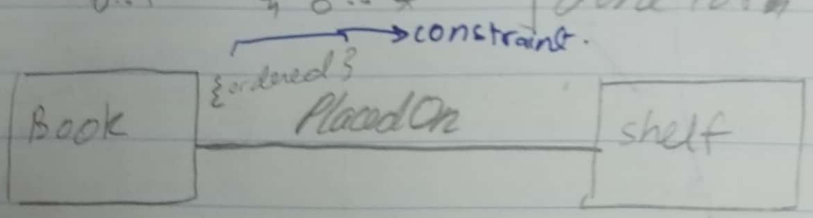
italic

not italic

is not

* senior most person ~~does~~ managed by anyone
* junior " " doesn't manage anyone

done for extreme cases



* sets dont have duplicates. if you don't write {ordered} it means ~~Do~~ by default book is ordered.

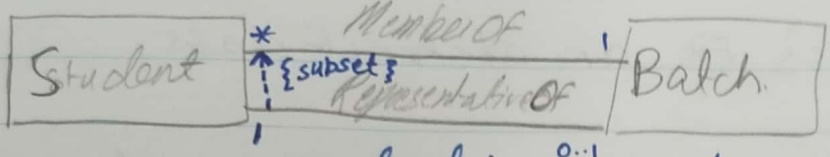
{bag} ← no order but duplicates
{sequence} ← can have duplicates & ordered
{ordered} ← order but no duplicates

order \ duplicates	Yes	No
Yes	sequence	ordered
No	bag	set

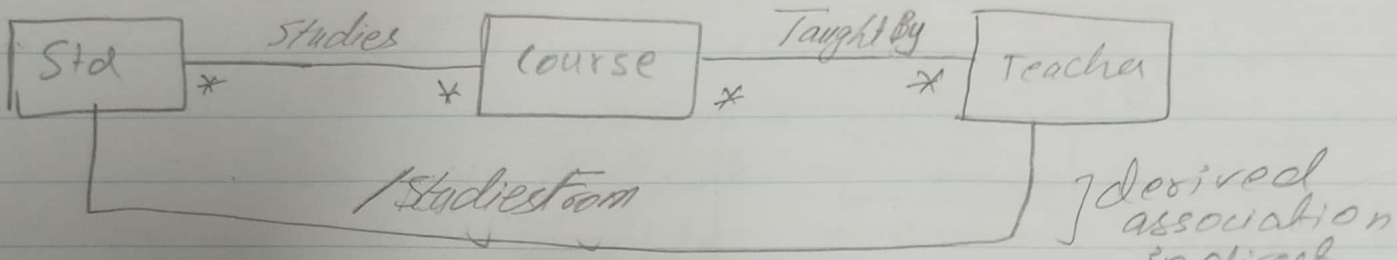
Foundation for Advancement of Science & Technology

* why over modularization not recommended? Increases integration cost.

representative = CR, a monitor or head.



* a representative is also the member of batch.
eg. ayka (representative of Batch 23) is also in Batch 23

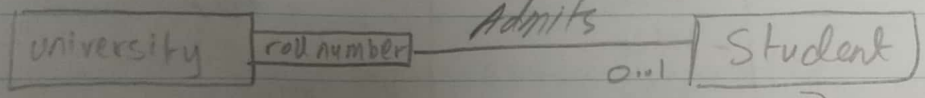


* there is indirect connection blw Std & Teacher
* used for optimization purposes.

New Lecture

Qualified association

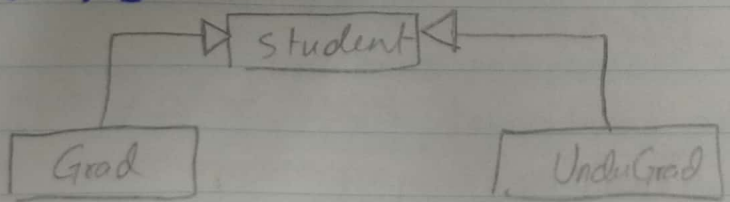
* it adds extra info.



(class + student ka aik variable)

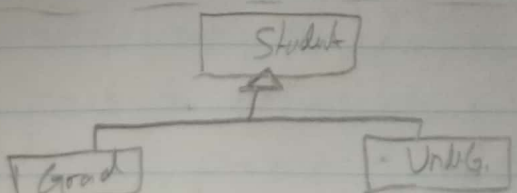
* means: university + rollno. uniquely identifies std.
* Why 0..1 ? kuch student ko abhi roll number nahi mila hota.
* Try to make qualifier on right side.
* rollno. is attribute of student.

Inheritance:



* is-a relationship.
* binary association

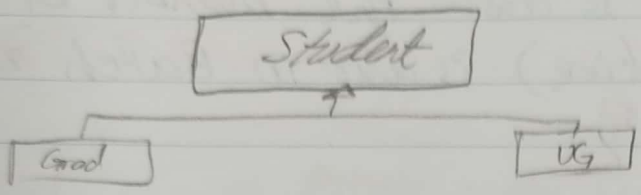
this is not better



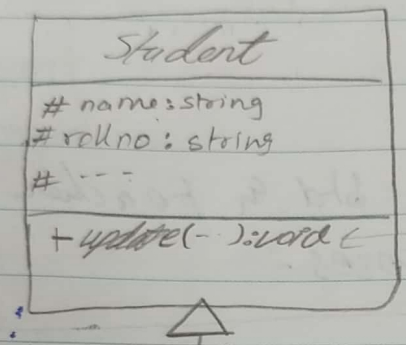
it is

How to make a class an abstract class.

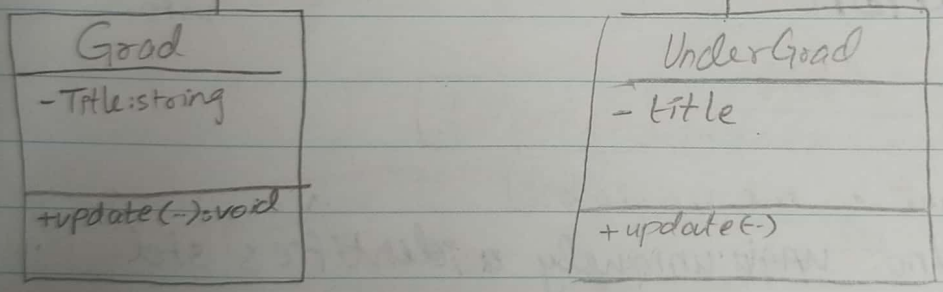
Make it italics



- abstract class needs only 1 virtual function

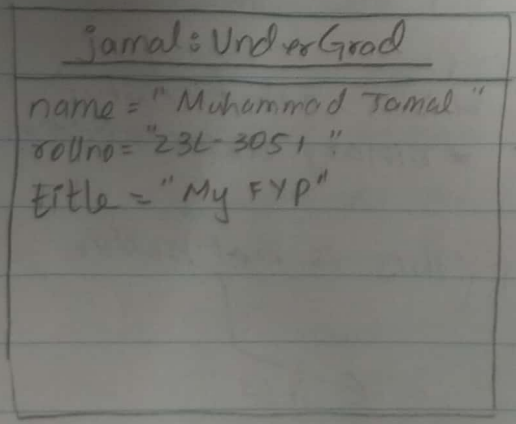


italics becz its abstract function/operation



same update function but don't make it italic in child classes

Object diagram

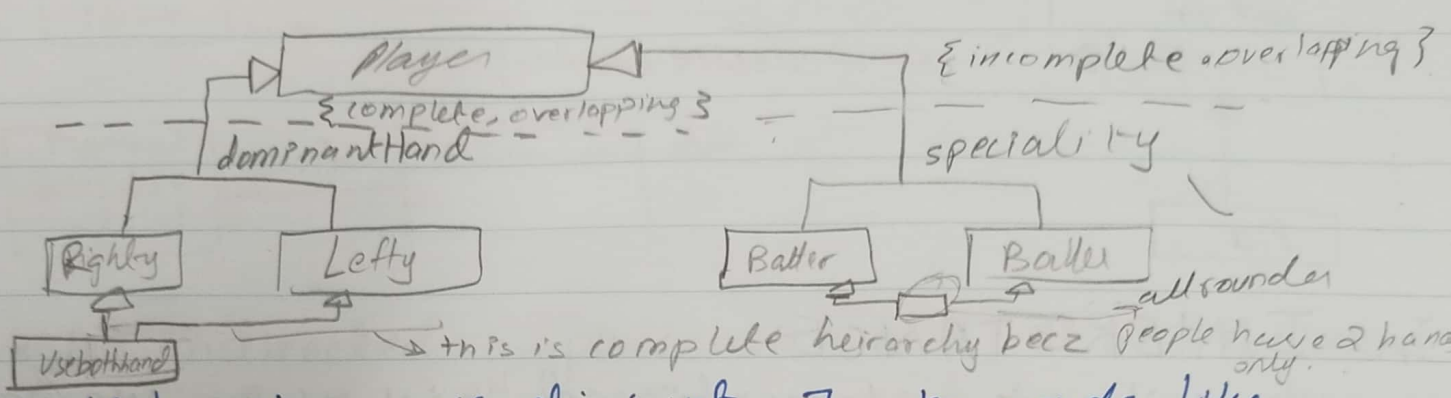


] → they are in abstract class

undergrad has total 5 features: name, rollno, ---, title & update()

Foundation for Advancement of Science & Technology

Generalization set name.

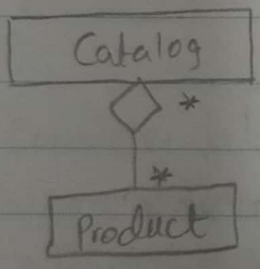


* overlapping vs disjoint
* complete vs incomplete.

→ keywords like ordered. constraints.

* overlapping is becz of "use both hands". complete means covering all possibilities.

Lecture 6.

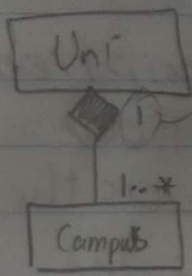


aggregation (has-a)

↳ has extra properties eg.
regular binary association are bidirectional/ symmetrical

↳ But it has anti symetrical association

↳ called Transitivity. } also property of inheritance



not mentioned many time

← This is for composition → special type of aggregation

← not regular aggregation.

← uni has campuses.

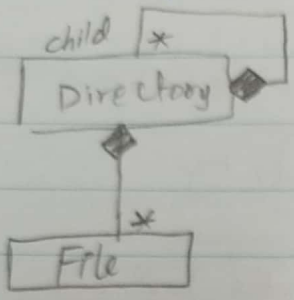
Extra properties of composition.

- same properties from aggregation eg. antisymmetry
- The multiplicity of composite end (◆) is 1. (almost 1)
- Both part & whole has coincident life time. if whole is deleted, part is deleted.

Foundation for Advancement of Science & Technology

* = 0 or more
1..* = 1 or more

* if HEC blacklist Uni then all its campuses are blacklisted.

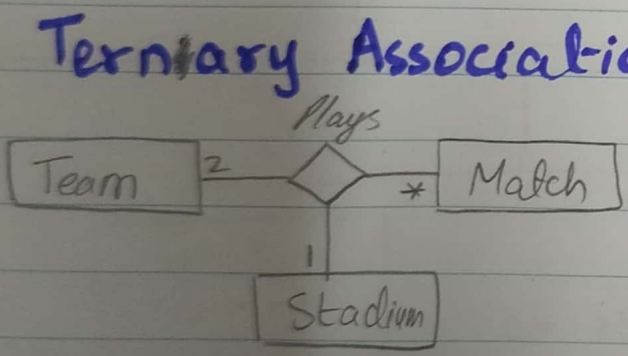


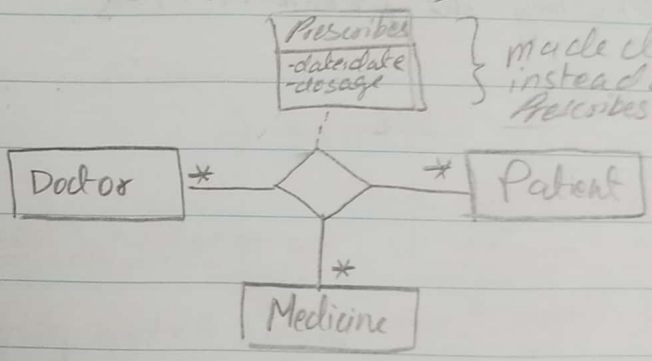
← recursive composition

In INHERITANCE:

* multiplicity is not mentioned in inheritance. We dont have to mention becz its always 1 to 1. Also becz ~~one~~ there are 1 instance of each class

Ternary Association



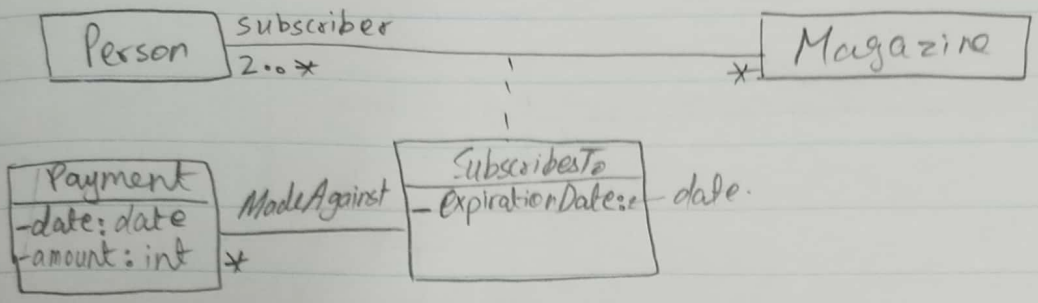


- fake association
- Difference is checked by looking at multiplicity.
* Ternary association has * in all corners.

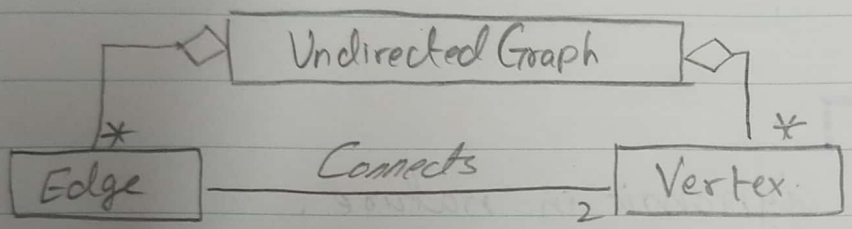
-- Genuine association

Q. A person may have (means may not have too) multiple (2 or more) magazine subscriptions. A magazine has multiple ~~asses~~ subscribers. For each subscription, it is important to track the date and amount of each payment as well as the current expiration date.

* multiple means: 2..*
* It is not mentioned "For each payment" thats why there is not a multiplicity on subscribes to



Q. An undirected graph consists of a set of vertices and a set of edges. Edges connect pairs of vertices.

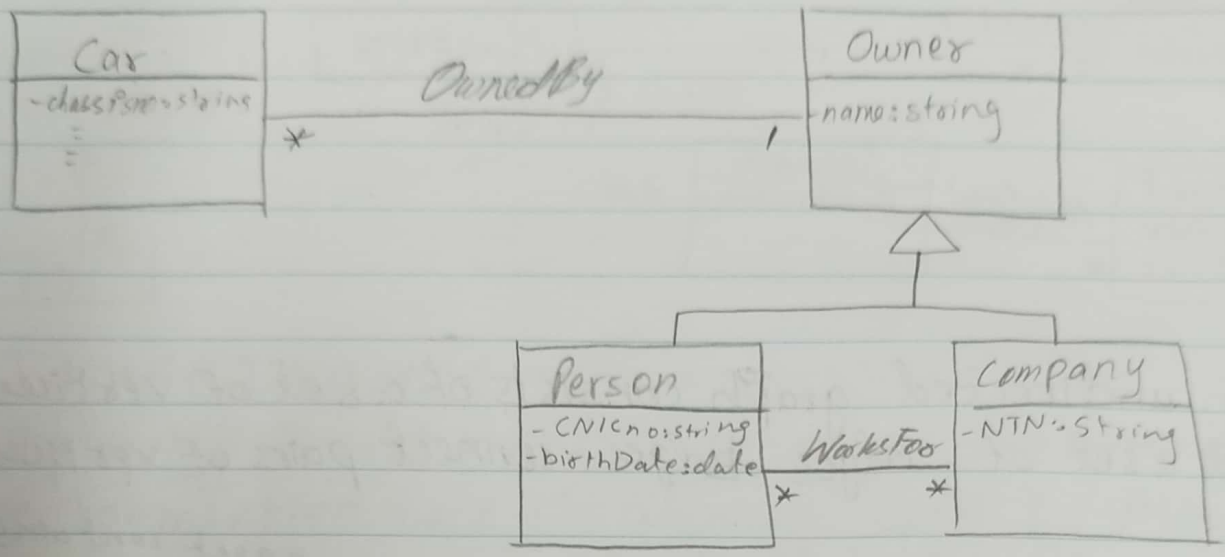


* a set contains 0 values so its 0 or more.

Q. Consider a car ownership system. Each car has an owner. Every owner can own multiple cars. An owner could be a person or a company. A person can work for multiple companies and a company can hire multiple persons. Each owner has a name and each car has a chassis no., model, make and license no. Every person has a CNIC no and birthdate and every company has an NTN.

Foundation for Advancement of Science & Technology

ch: 3 4 4 ← BOOKS
3.5 } not part of syllabus.



Lecture - 7

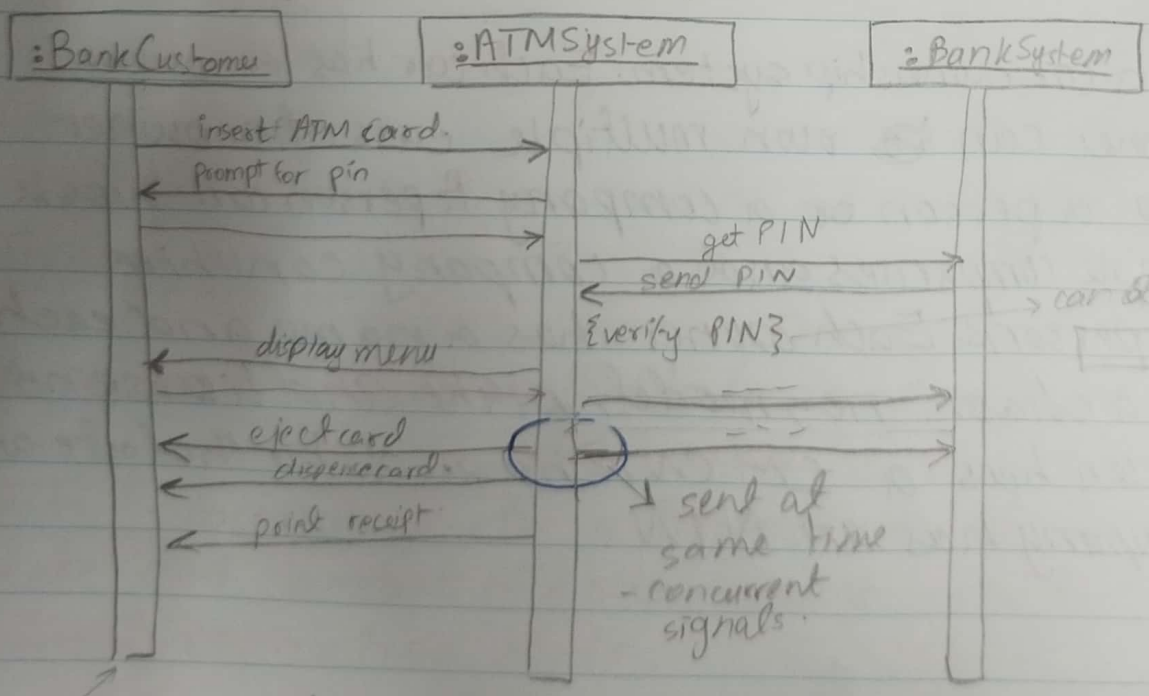
Sequence Diagram = dynamic in nature.

drawn in analysis = system Level SD.

" " design = design Level SD.

object boxes = anonymous

system level analysis phase SD



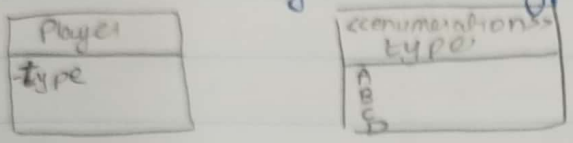
life line.

* draw different SD for different scenario.
* 1 for one ~~case~~ usecase but different for every alternate action.

asynchronous

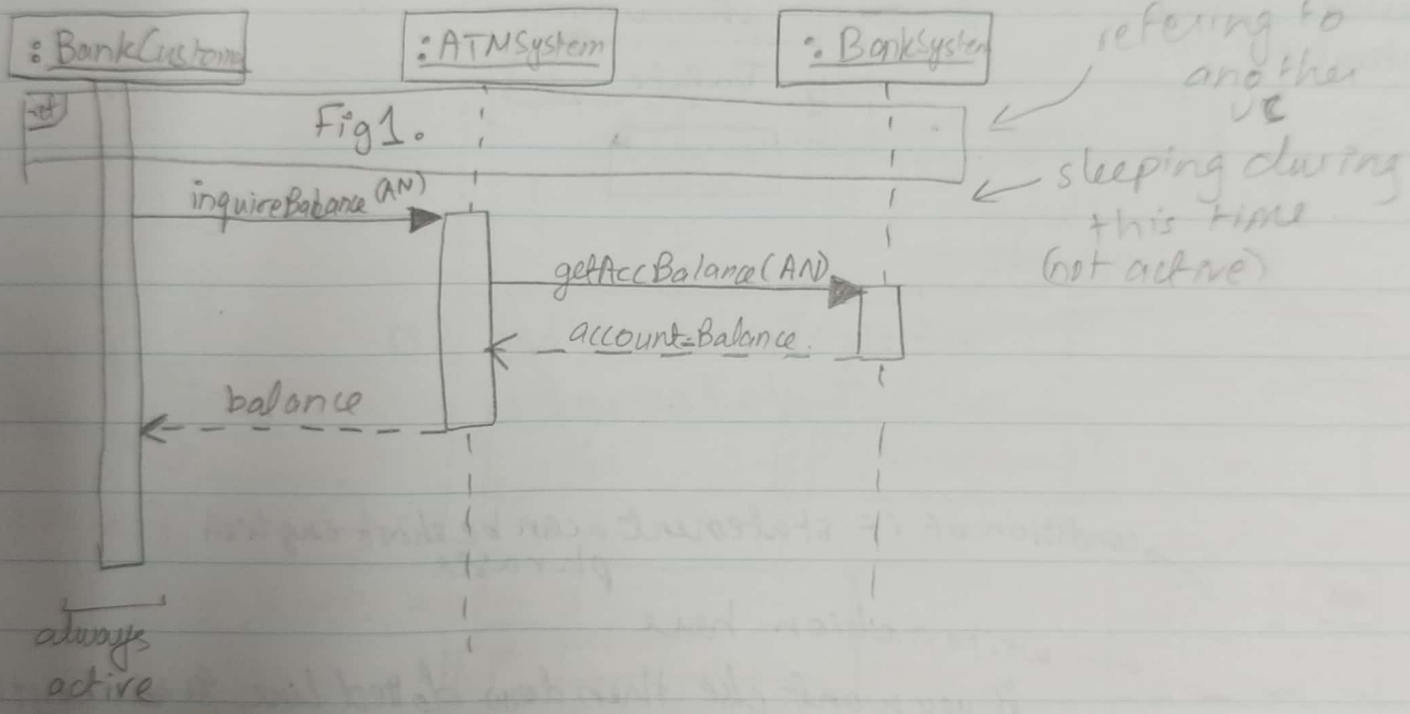
Foundation for Advancement of Science & Technology

Ex: The class "Player" has types (A, B, C, D). It is fixed.



← Do this when values are fixed.

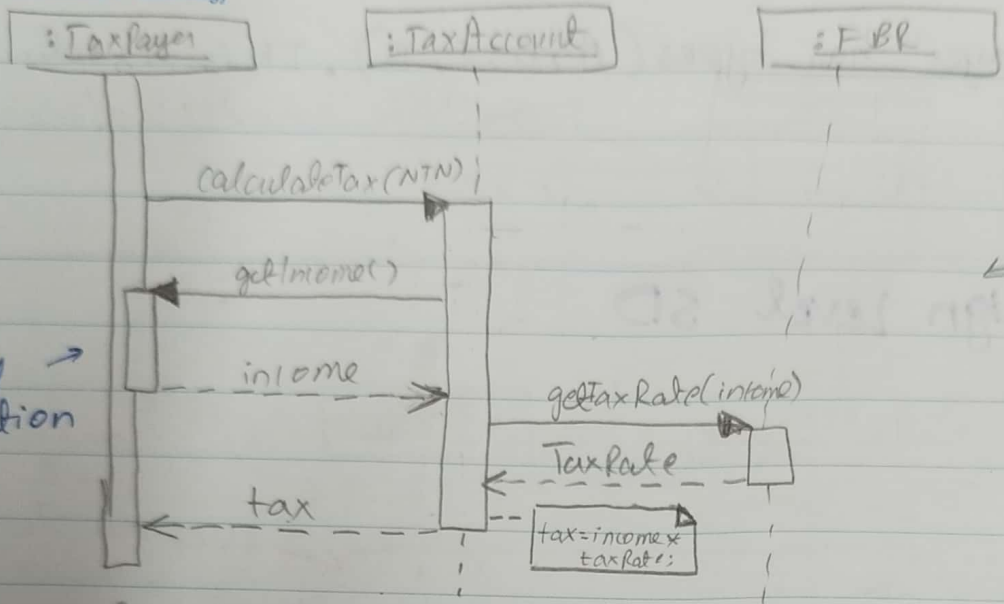
2.8 : Design Level SD



- inquireBalance is a function which returns "balance" variable. AN is a parameter. = account no.
- If its a void func then return line ~~must be straight~~ should be dotted too
- ~~design~~ Class ATMSystem has func. inquireBalance.
- These are synchronous message calls.

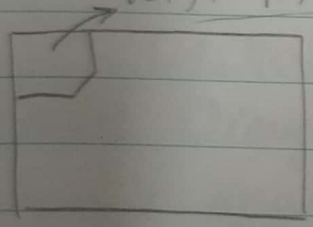
NEXT PAGE

int ~~set~~ calc.
{
getIncome
}



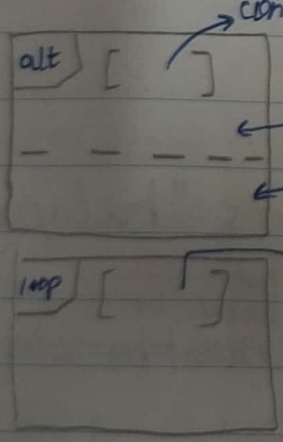
nested activation

nested function call



ref = refer to another SD.
alt = if statement.

called interaction frames.



condition of if statement = can be short english phrases

write action here

if you want else then draw dotted line & everything below dotted line is else.

terminating condition.

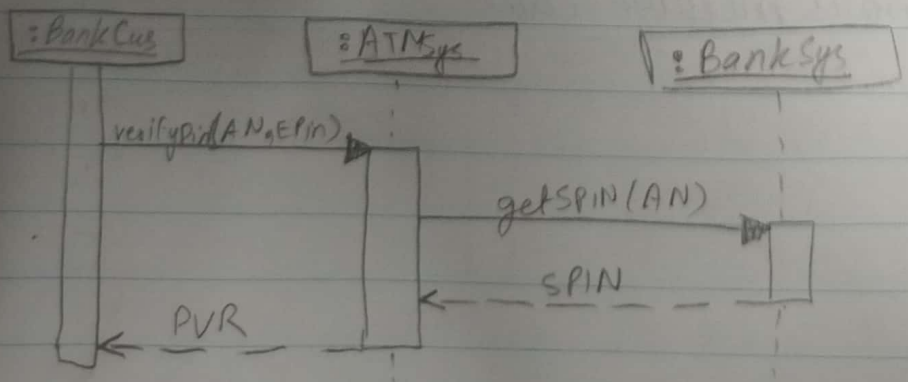
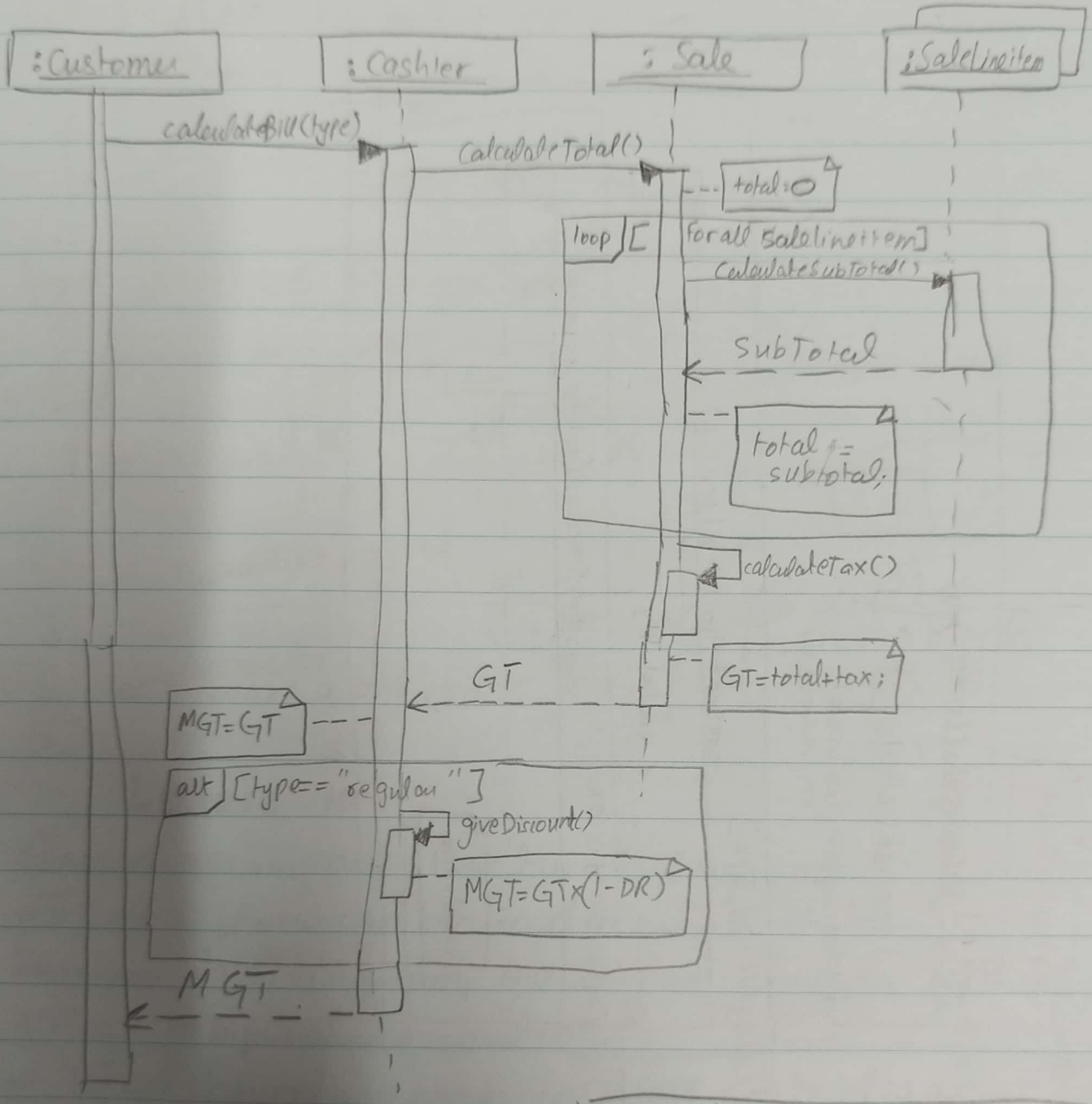
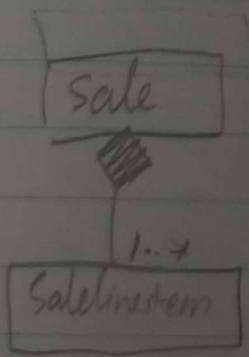


Fig 1. DSD for verifyPIN & UC

Foundation for Advancement of Science & Technology

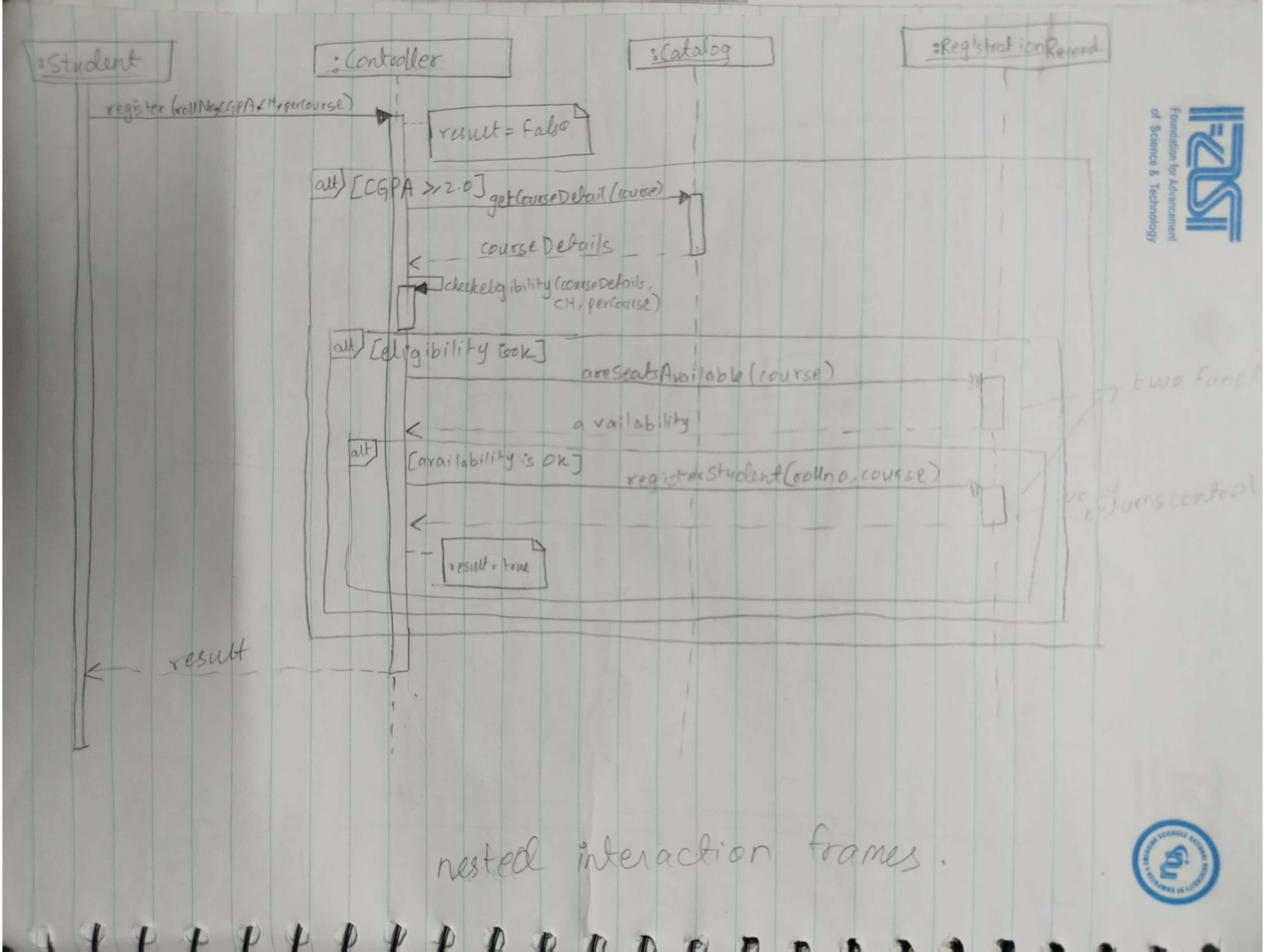
Calc Tax
(getin C)



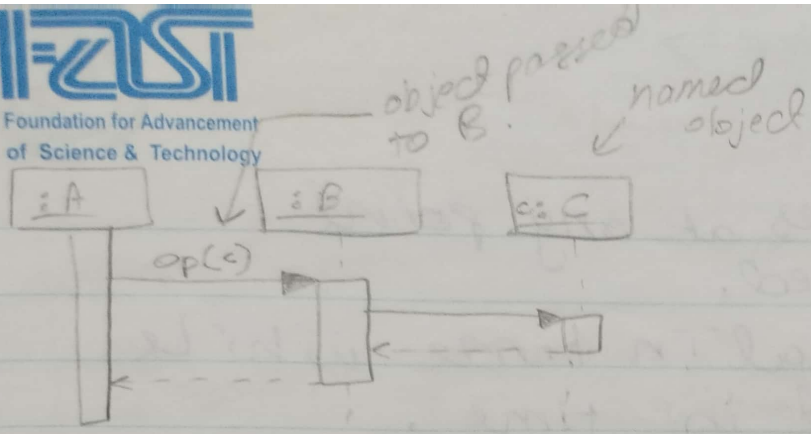


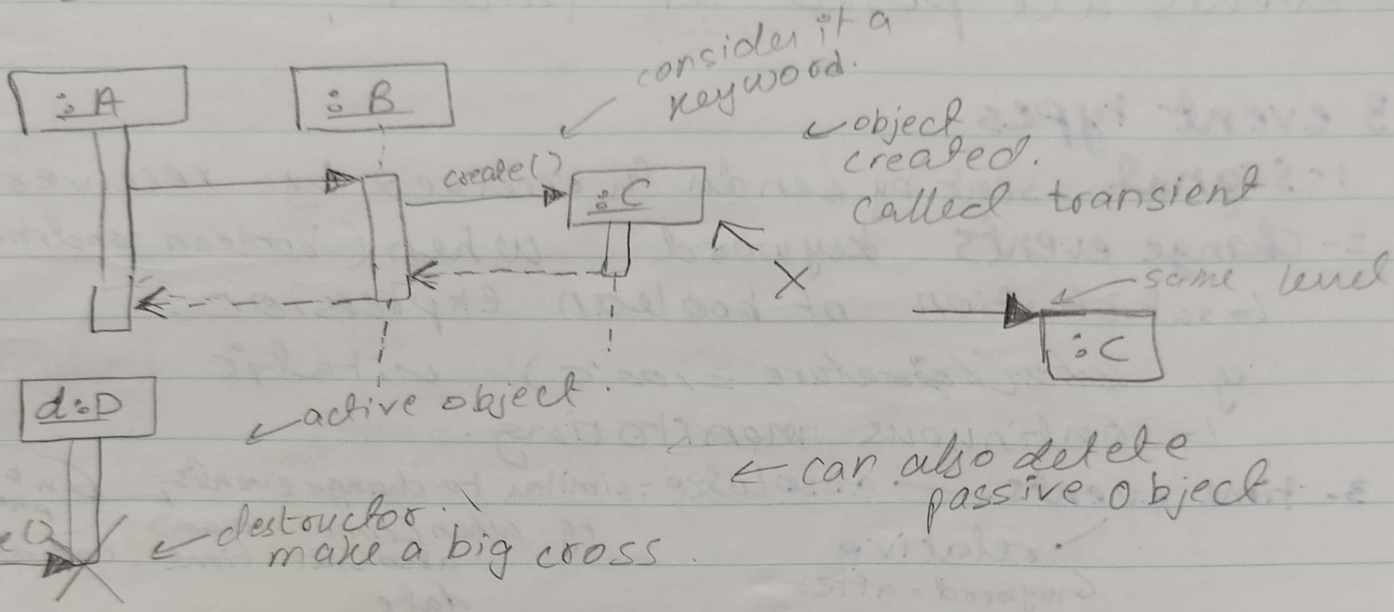
S#	item	UP	Qty	ST
1.	Eggs	300	1	300
2.	Bread	200	2	400

total: 700
tax: —
GT
MGT: —



nested interaction frames.

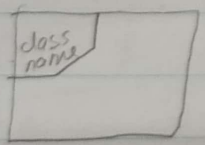


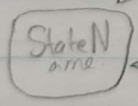


Lecture 10

- class diagram are structural diagrams
- obj of a class can be in different states.
- ~~These mo~~ State diagram made for classes with objects giving temporal information.

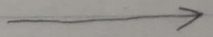
* ~~bound~~ boundary exists in ~~class~~ state diagram.





* objects of class has state diagram. not classes itself.

state means: snapshot of obj at any point in time. Combination of attribute values and links.



Foundation for Advancement of Science & Technology

eventName → italic & lowerCamelCase

* events are occurances at any point.
  eg. course registered.
* states are interval in time while events are point in time.

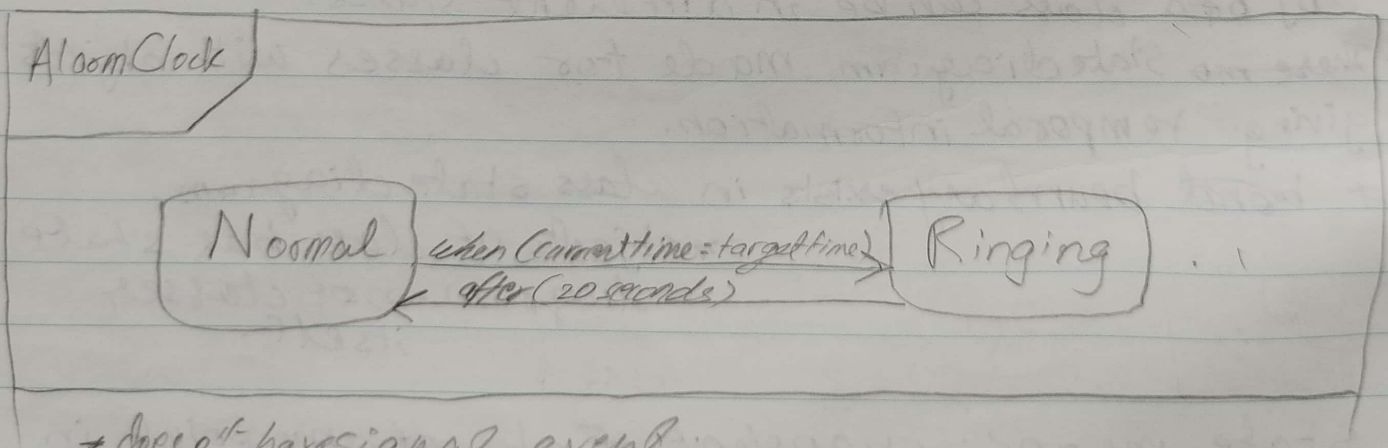
3 event types

1- **Signal** = sent by sender & received by receiver.

2- **Change events** keywoord when (boolean condition)
  ↳ satisfaction of boolean expression.
  eg. when(temperature ≥ 100°C) ← italic
  ↳ continuous monitoring.

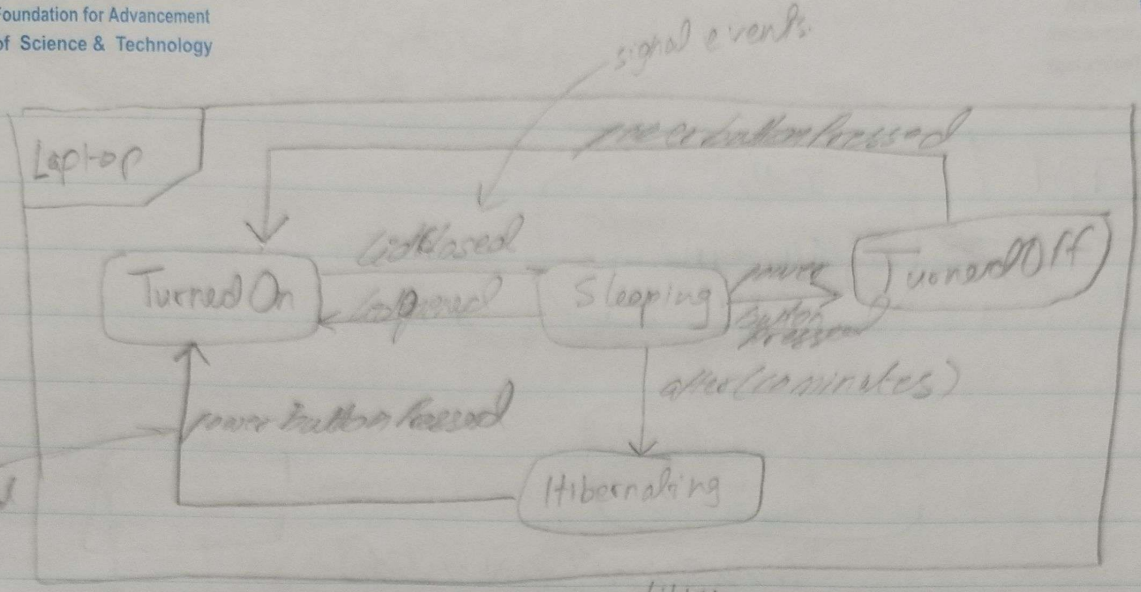
3- **time events** — absolute : similar to change events. ← can be phrases
  eg. when(time == 10 am)
  absolute time. can add date.
  — relative
  ↳ keywood = after
  after(100 seconds) ← italic

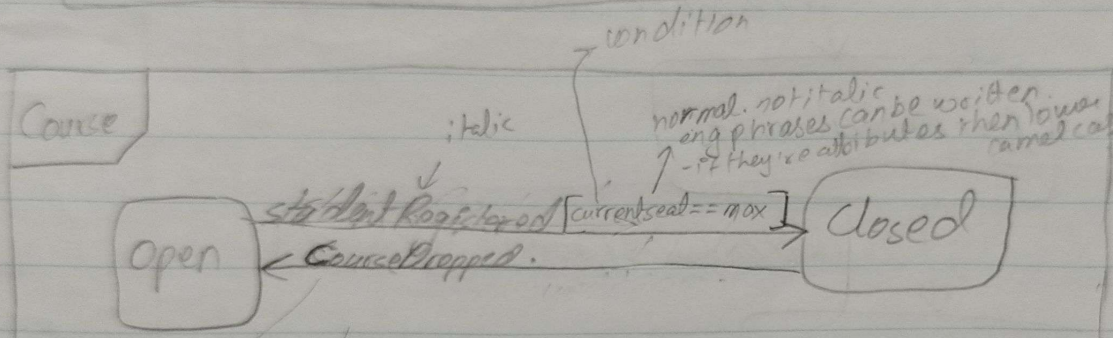
* all events and conditions are written in italics.



* does not have signal event.

Foundation for Advancement of Science & Technology



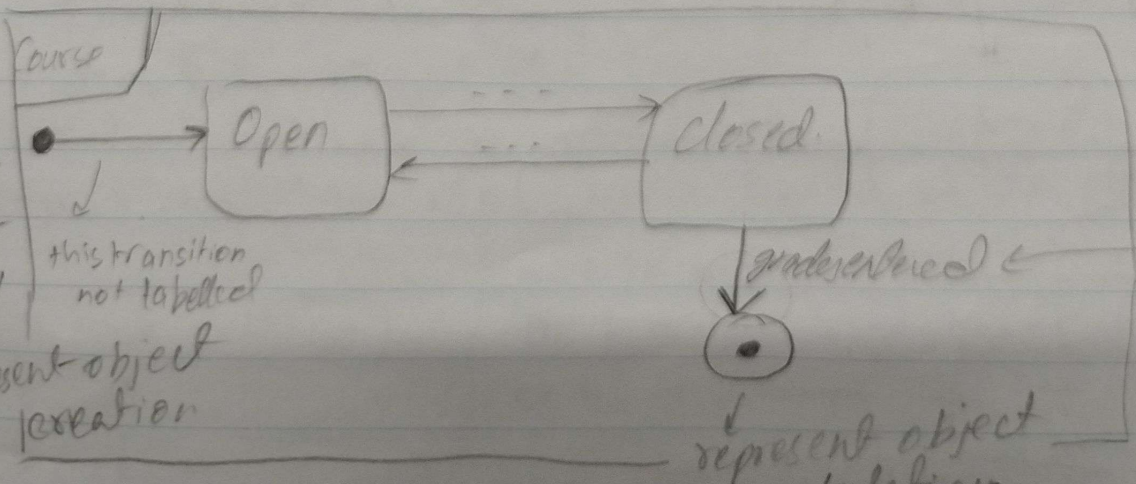


↳ why 'when' not used?
or why not change event
↳ becz change event is continuous checking while this only checks when required. This one is more optimal.

signal guarded by boolean condition

↳ called continuous loop state Diagram.

- There is another type of SD called One-Shot



transition labelled

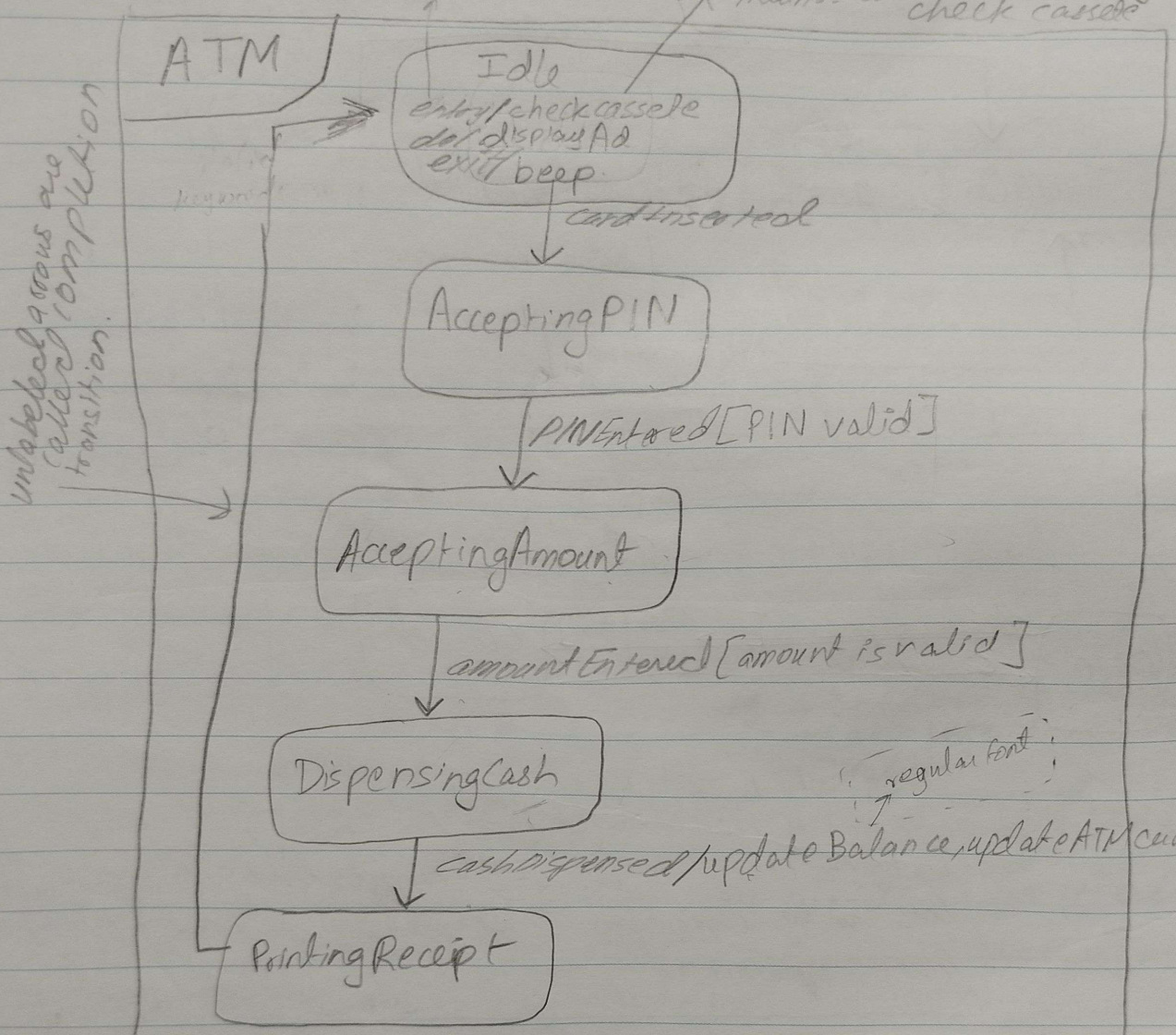
Foundation for Advancement of Science & Technology

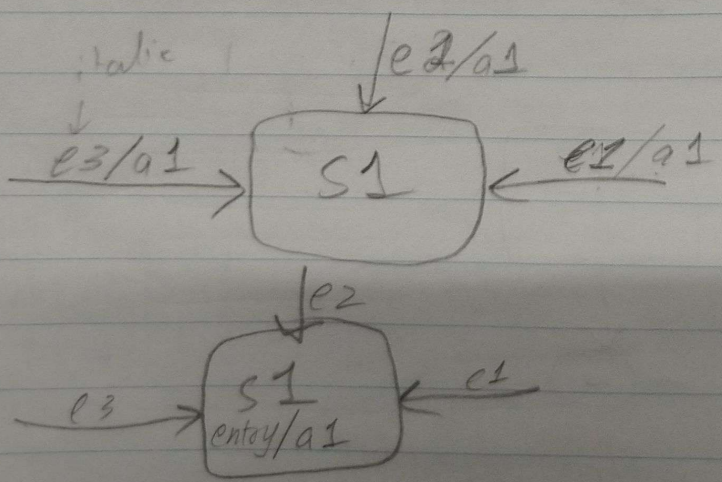
entry, do, exit are keywords.

normal eng.

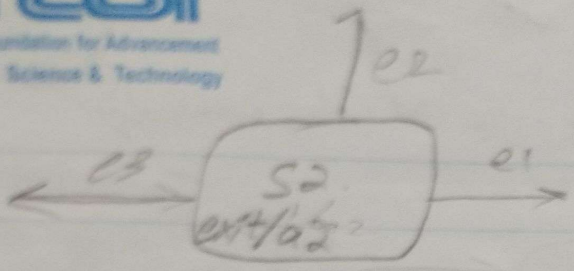
italic

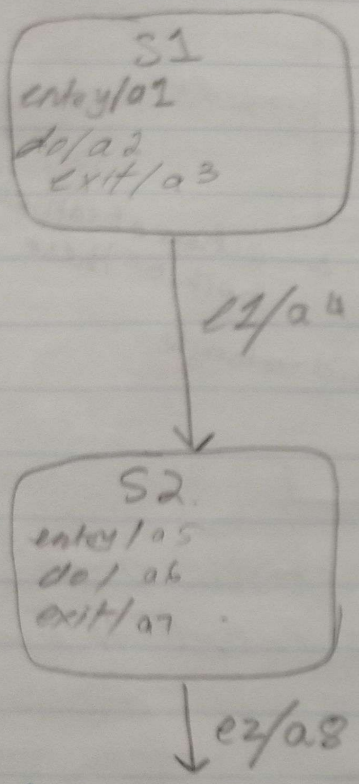
activity means - at time of entry check cassette





whatever event happen, a1 is performed so take out the common

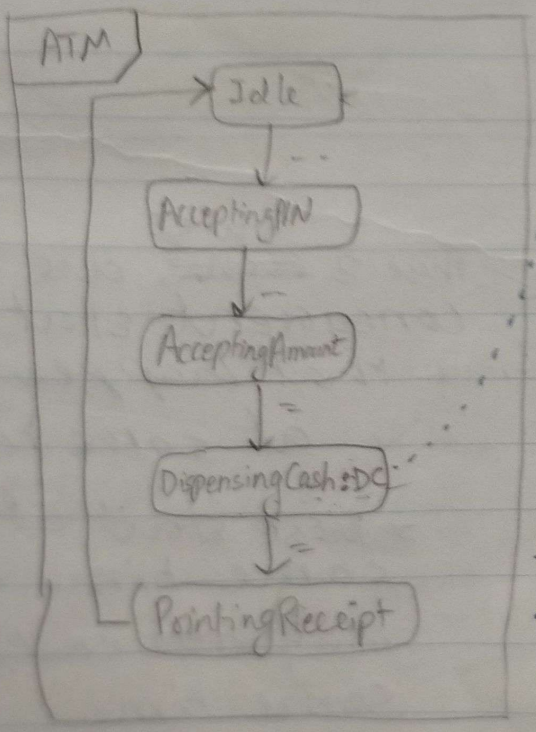


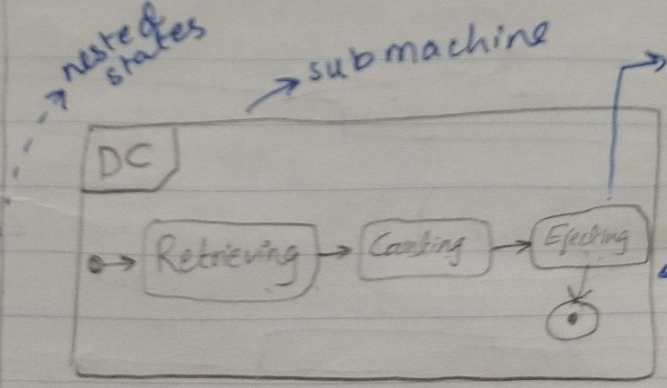


→ a2 can be completed or continues. a2 can be interrupted. How to we know S1 is complete? when a2 completes

Lecture - 11

State diagrams + package





- Details of state can be shown in nested states
- Done to maitain same level of abstraction
- DC has low level states that don't have to be included in main diagram.

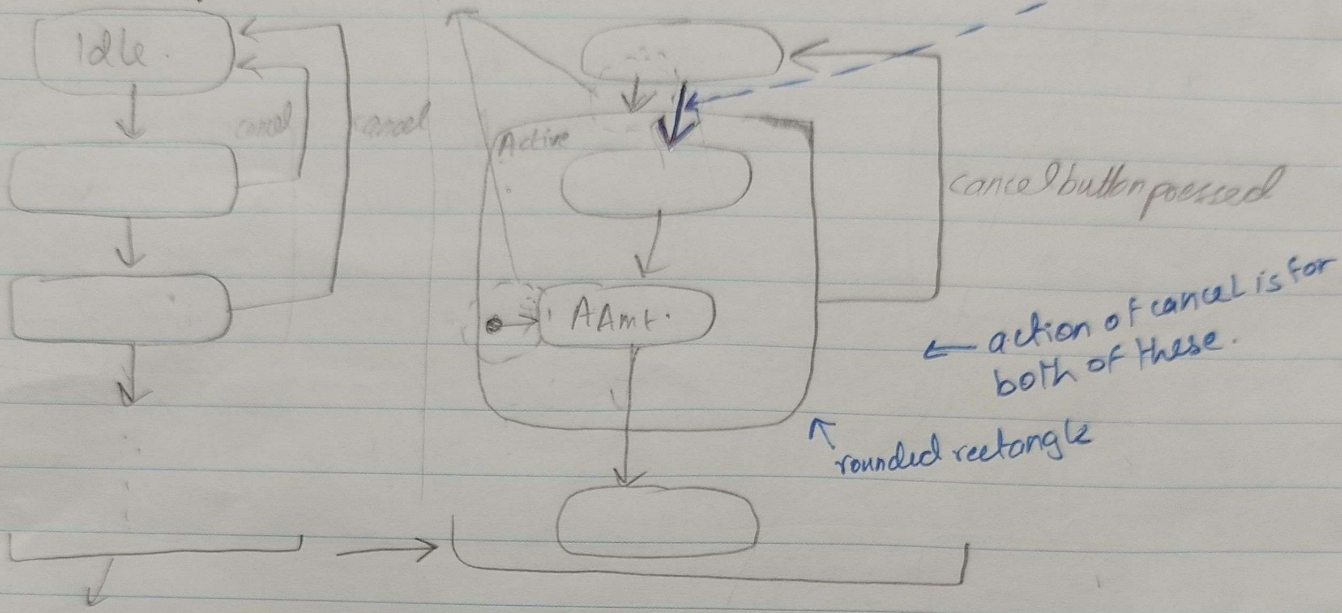
- all states in state diagram should be at same level of abstraction

Foundation for Advancement of Science & Technology

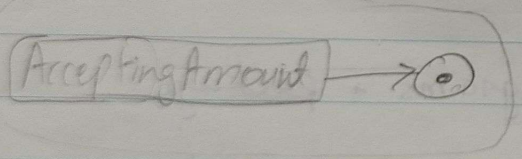
if both not drawn then there's confusion in diagram so it will become malformed diagram

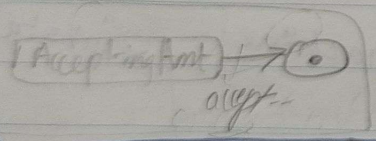
Composite states

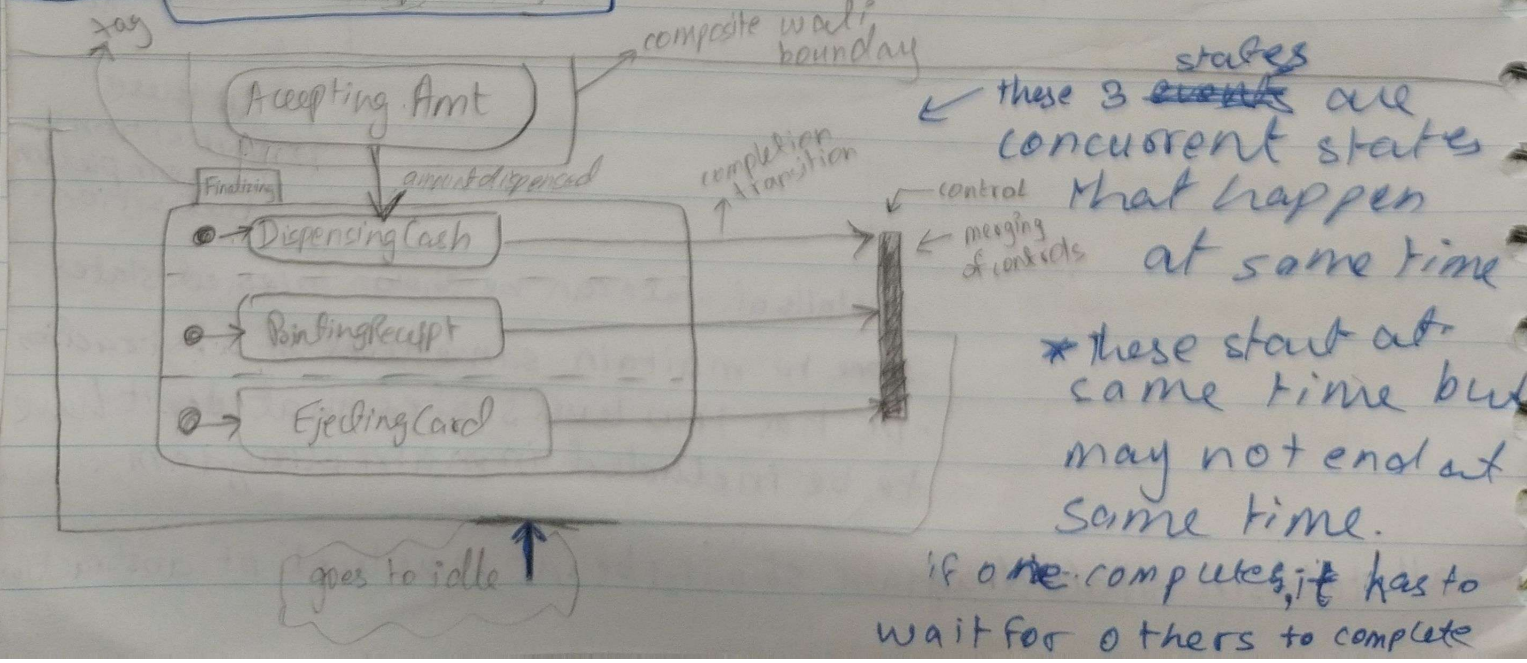
these two are same as the arrow,



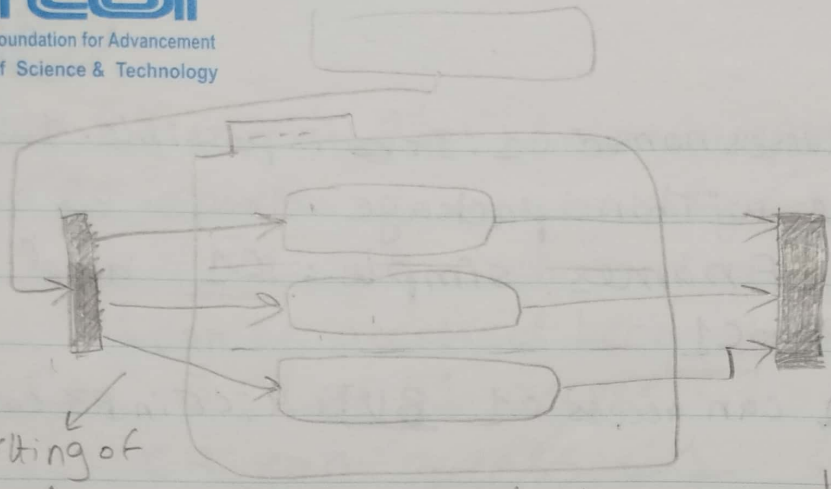
when two consecutive states have same arrows going to same state with same label then composite states are made.







Foundation for Advancement of Science & Technology



same as before. no difference

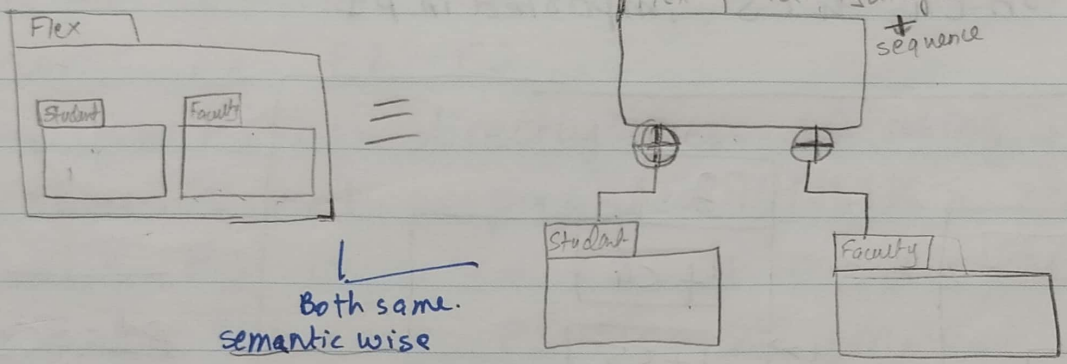
splitting of controls or forking of control.

dynamic ↑ (behavioral) vs ↑ static (structural)

state, use case activity diagrams + sequence

class, obj + packages

Package Diagrams



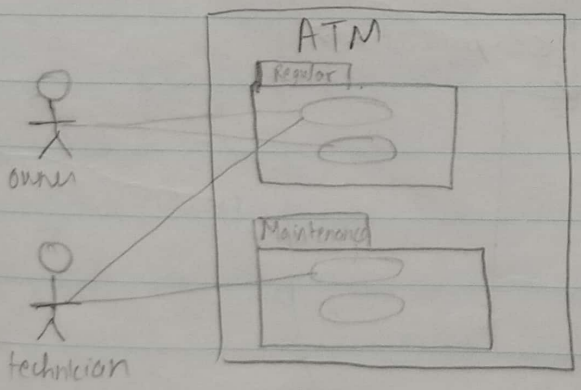
Both same. semantic wise

⊕ ✗

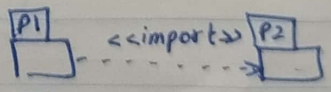
⊕ ✓

use cases have Package diagram too

- Package Diagram are used for organizing components
- Packages can contain classes

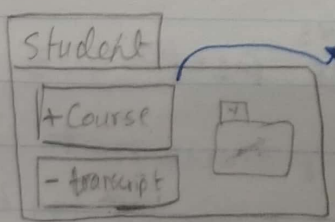


- Student is containing package
- X is contained package.
- Transcript is visible to Course and also X package and all sub sub packages



P1 can import P2.

Package Visibility : + public - private



package contain classes

Dependancy:

<<import>> - - - - - →

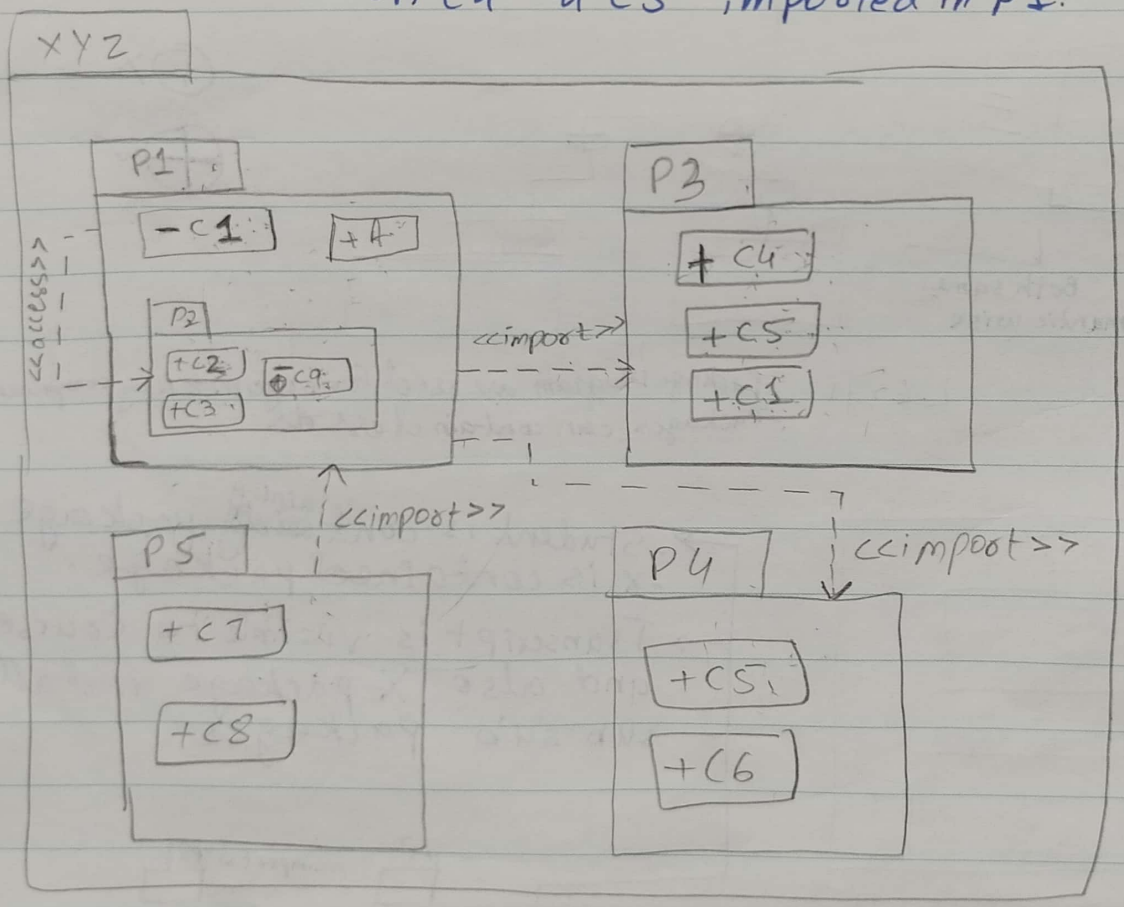
<< access >> — — — — →

Foundation for Advancement of Science & Technology

* P1 & P2 both have classes named C1. It is possible. The names need to be unique within package.
* There are 2 types of names. simple : C1 and fully qualified : P1::C1
* C3, C2 & C9 in P2 can access C1 BUT C4, C5 in P3 can't

P1 imports P3

↳ Content of P3 becomes part of namespace of P1 except when there's a clash, the clasing component isn't imported. P3::C1 is not imported in P1. only C4 & C5 imported in P1.



to access

P1::C1

fully qualified name

C1

not fully qualified name

* what is benefit of import? classes of P1 can use simple names to use P3 classes.
* if C5 needs to access C3 then it writes P1::P2::C3
* Now **P1 also imports P4**
  ↳ clash with C5 so both C5 in P4 & P3 are not imported.
  - now only C6 and C4 are imported. But C5 can be accessed by P4::C5

Foundation for Advancement of Science & Technology

Now P5 imports P1. chain of imports: → also P2 package

↳ P5 imports C4 & C6 only. but can access C2 like P2::C2 instead of P1::P2::C2

- C1 can be accessed becz its private

* Can C1 of P1 access C9 of P2 ? NO but C9 can access C1

- Remember: import doesn't change visibility

P1 accesses P2.

↳ Before to access C2 & C3, P1 needed fully qualified name i.e P2::C2.

↳ But now P1 can directly access C2 using simple name.

* access is not propagated UNLIKE IMport.
* when P5 imported P1, it imported P3 & P4 too.
* But in this case P5 does import P1 but can't access P2 directly.
* ACCESS is only for P1, not for packages importing P1

- Access is limited type of import.

Foundation for Advancement of Science & Technology

Component Level Design

* coupling & cohesion are functional decomposition idk.
* Cohesion = degree of singlemindedness.
* coupling = " " connectiveness

Cohesion Types

1- Functional = strongest type. Exhibited by func., operation
↳ known as perfect or atomic cohesion.
↳ does one thing & return control.

2- Layer : multiple layers. Top layers access services of bottom layers but not vice versa.
↳ used in MVC architechture.
Also high level.

3- Communicational - informational cohesion.
All Func. that access same data are combined in class.
Also high level.

* Try to strive for these levels.

Lower Levels:

4- Procedural = function are called one after another. Functions are bunched together in order. in same class.

5- Sequential: Its special case of Procedural. Difference is that now data is passed too.
order + data passed

eg. The data returned from Func1 is given to Func2.

6- Temporal
All Func. that are called at same point in time eg. at start or end or in middle.
eg. ~~POST~~ POST. power on self time.

7- Utility lowest type.
All Func. that belong to one general heading are punched in same class eg. all func. relate to stats are combined in statistics class.
↳ also called coincidental cohesion.

Difference btw high & low level

access same data. ← → data is not shared here.

Coupling Types:

-cant eliminate coupling

1- Content =worst type.
One class can access & modify content of another class.
↳ eg. making everything public.
You're not hiding anything.
↳ Friend class & func. leads to this.
↳ inheritance can also do " " ".

2- Common

↳ They share data. Example: global variables.

↳ unattended side affects occur when global vars are accessed by different classes

3- Routine call

↳ functions call each other. Nested func. calls.

4- Data : special type of ↑

now data is transfered through parameters.

```
f1()
{
    f2(d)
}
```

← means f1 & f2 must rely on same interpretations of 'd'

* NASA lost a space ship becz of different interpretation of this.

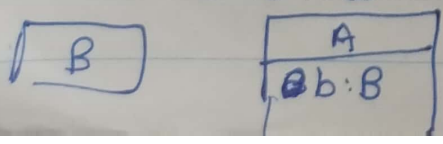
5. Control : special type of data.

If flow of f2 depends on d eg.

If d is true then if statement is executed. On value of 'd' the func works differently.

6- Type use

Classes are abstract. One class uses another class



← A class has B class variable.

Foundation for Advancement of Science & Technology

* if B changes, A has to change.
* if A has any func. which has local var & that var is B.

7- Stamp = function signatures
If A class has.
+ f1() : B
+ f2(p1 : B) : void } this is stampt.
↳ related to func. Prototypes.

- import (packages import)
- export

* Coupling can result in team dependencies.

SOLID PRINCIPLES

history: assembled by Robert C. Martin.

1- Single Responsibility Principle (SRP)
2- Open-Closed " " (OCP)
3. Liskov substitution " (LSP)
4- Interface Segregation " (ISP)
5. Dependency Inversion " (DIP)

SRP = each class should have a single responsibility

OCP = classes should be open for extention but closed for modification. Extention is done by making sub-classes.

Foundation for Advancement of Science & Technology

LSP = named after dr. Liskov. (Female) → got PHD from Stanford
↳ says: subclasses should be substitutible for parent-classes.
↳ only happen when child class preserves the semantic meaning of superclass.

ISP → should segregate interface. Should have small sized multiple client interface.

DIP: code should depend on ~~extraction~~ abstraction but not on concretions.

JAVA CODE from Websites

```
Class Invoice()
{
    public Invoice() { .. }
    public double CalculateTotal() { ... }
    public void printInvoice() { ... }
    public void savefile() { .. }
}
```

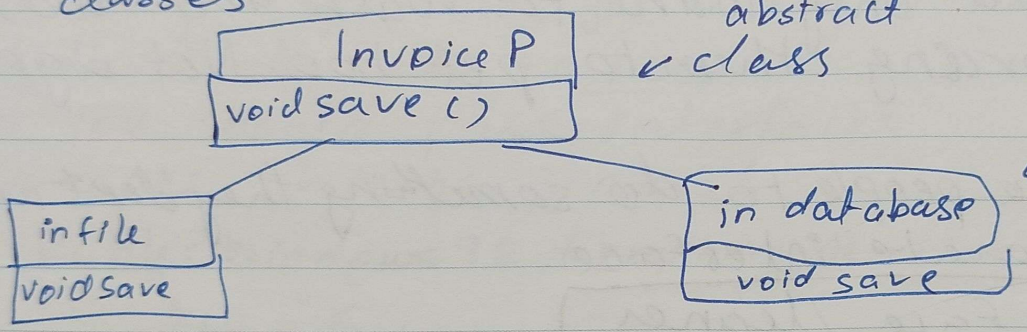
* It violates SRP becz it has multiple responsibilities
* if any of the function changes, you have to change class
* if in future you want to print book name then you have to change class or function.
* it was saving in file but lets say you want to save in database now => change class
* Divide it in classes.

Foundation for Advancement of Science & Technology

* make one class to print & one class to save to file.
* makes classes single minded.

OCP

* if you want to save to database, you add another func. in invoice class = WRONG
  - You extend class, not modify it.
  - You make interfaces. Similar to abstract classes



* make design more flexible.
* The code remains generic.

Liskov

* square class extends Rectangle.
* subclasses should not change semantic meaning of classes
* Rectangle is a shape in which width & height are different. This meaning has to be preserved.
* if width is changed for rectangle only width is changed however if width is changed for square height changes too so now its not a rectangle. It changes meaning of rectangle.

Foundation for Advancement of Science & Technology

ISP

```
Public interface BearKeeper {
 void washbear();
 void Feedbear();
 void petthebear(); }
```

* whatever class implement this interface have to implement all 3 functions.
* Petting the bear is dangerous.
* So if people only want to wash and feed, you'll be forcing them to pet too. Not a good idea.
* Dont force people to do something they don't want
* Create separate interface.

```
public interface Cleaner
{ void washBear(); }
```

```
Public inter.. Petter
{ pet() }
```

```
public interface Feeder
{ .. Feed. }
```

* So now a person can implement anything and you're not forcing them to do anything

Foundation for Advancement of Science & Technology

Dependency

* depend on abstoaction

```
class Windows98Machine {
    final StandardKeybard key;
    //     Monitor         mon;
    public Windows98Machine() {
        mon = new Monitor();
        key = new StandardKeyboard();
    }
```

final → const

constructor

* eg qwerty etc..

* using a specific type of keyboard. You have direct calls to new
* You should pass interfaces
* make things general.

```
Public Windows98 (Keyboard k, Monitor M) {
    key = K;
    mon = M;
}
```

---

* underneath all these You still have inheritance & polymorphism

---

New Lecture : Design Patterns.

Foundation for Advancement of Science & Technology

Problem + solution

* A design pattern is an invarient solution to a recurring problem within a certain context.

* goocy design pattern: bocad crumbs ← a way to show your path.
* ecommerce design .. = shopping cart.
* In coicket, a context can be a pitch or weather so there are many contexts.

	creational design pattern	structural	behavioral	purpose.
class				scope
object				

2 x 3 table used

- classification = tellus a lot about design patterns
- 3 x 2 = 6 design patterns.

Class scope: deal with relationship b/w classes & sub classes. This relationship are static and inheritance.

object scope: dynamic relationship: that change at run time.

Creational - deals with creation.
↳ instantiation

Structural: composition

behavioral: interaction b/w classes

* The overwhelming design patterns are in last row: object wali row.
= 24 total designs.
Object has = 20
class = 4.

Foundation for Advancement of Science & Technology

* Templates give us consistency.
  ↳ help us to compare different designs.

* It has many fields.

- Every design pattern has:

Template

Fields:
- Name & Classification → most of them have 1 word name
- Intent → a brief summary of design pattern (2 sentence long)
- A.K.A → alternate names
- Motivation → gives real world example (specific example)
- Applicability ← general examples
- Structure → given as a design class diagram
  - ↳ general solution
  - The classes are called participants
  - ~~The relations~~
- Participants
- collaborations → relationships b/w classes
- sample code
- Implementation
- known uses → used in any system

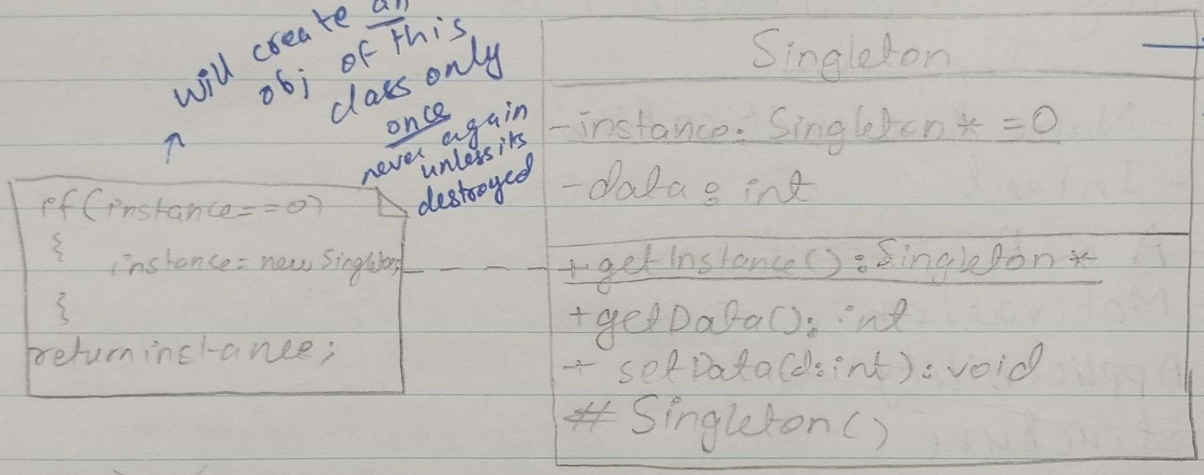
Adv of design patterns = reusing • improves productivity & quality
- makes design more flexible & elegant.

Disadv = use with caution. We use extra classes so it makes design more complex. Not always applicable. Sometimes performance degrades slightly.

* The template also has consequences field.

Singleton Classification: Object Creation

scope (dynamic relationship) — purpose

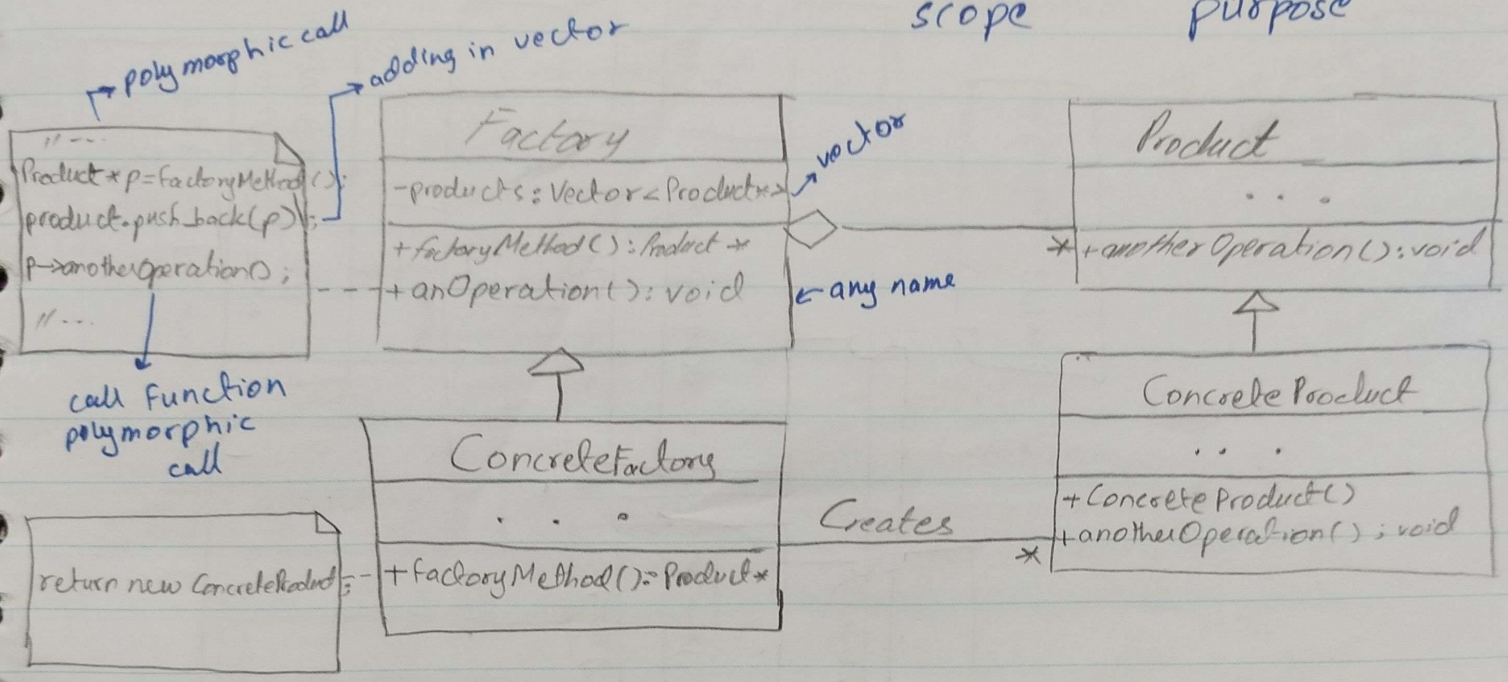


Singleton s;

* a set that has exactly 1 member is called singleton.
* a singleton class has only 1 object.
* we're making ecommerce system so there will be only 1 admin.
* ★ The template had a variable for structure which is generic solution so singleton comes here.
* To ~~protect~~ prevent user from creating classes using ~~any~~ constructor, we make constructor protected.
  Other outside world can't call constructor now
* The class itself is responsible for creating an obj for itself.
* Not giving creation control to clients (pp)

Factory Method Class Creational

scope purpose



* The knowledge of which type of object to create is not present in main (parent) class. The subclasses know which type to create.
* Object creation is delayed to child classes.
* This design has many variants & basic structure of 1 variant is drawn on top.
* The client code will remain unchanged. Clients only have to deal with ~~concrete~~ abstract concepts.
* You can keep on adding more & more objects at lower level.
* Both parent classes can be concrete & ~~can~~ will provide default implementation of factoryMethod() & anOperation().
* essence of this pattern is ~~redirection~~ extra level of indirection

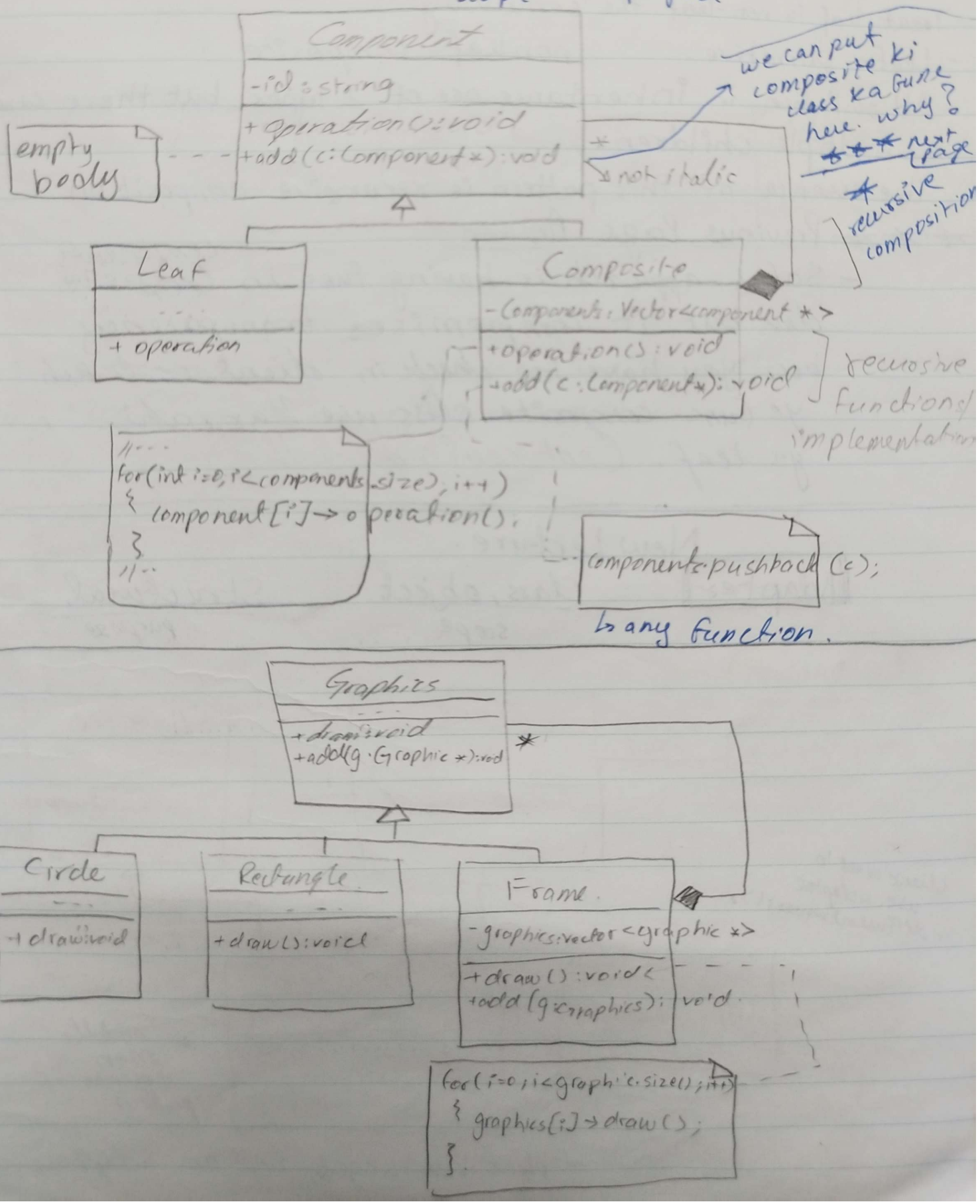
MSOffice
-Files: Vector<File *>
+CreateFile(): File*
+newFile(): void
File f = createFile();
files.push_back(f);
f->open();
Word
+CreateFile(): File *
return new DOCX();
PowerPoint
+CreateFile(): File *
return new PPTX();
Excel
+CreateFile(): File *
return new XLSX();
Create
Create
Create
File
-name: string
-createDate: date
+open(): void
DOCX
+open(): void
+DOCX();
PPTX
+open(): void
+PPTX();
XLSX
+open(): void
+XLSX();

Foundation for Advancement of Science & Technology

→ deals with relationship b/w objects

Composition

object scope | structural purpose



- Tree type structure
- Treat leaf & non-leaf the same way
- leaf = primitive. non leaf = composite
- All children in Inheritance are of 2 types. but there can be multiple children.
- The essence of this pattern is recursive composition.

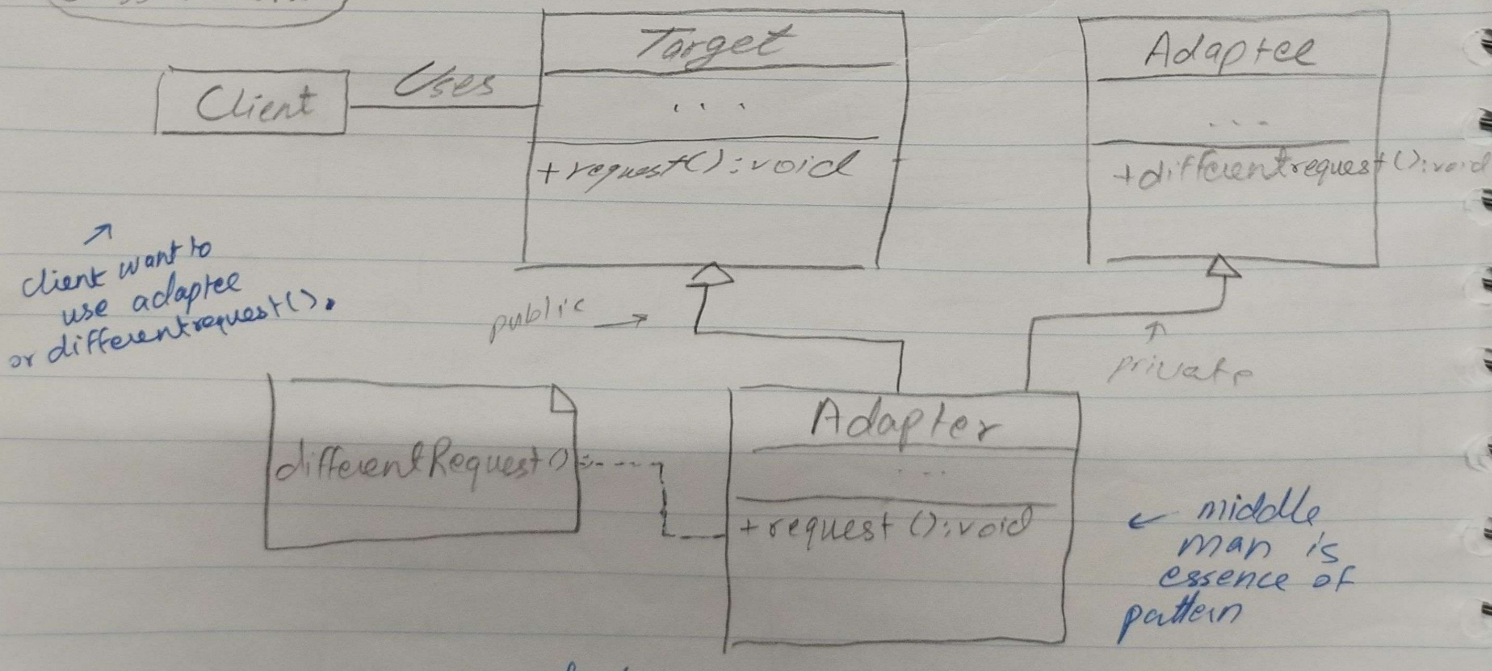
*** Previous Page Answer

- Safer approach is having func in components class but it compromises transparency becz you have to check in client code kuh ye func composite class use kar rahi ya leaf. (not confirmed tho)

New Lecture.

Adapter — Class, object (scope) — Structural (purpose) — dynamic in nature

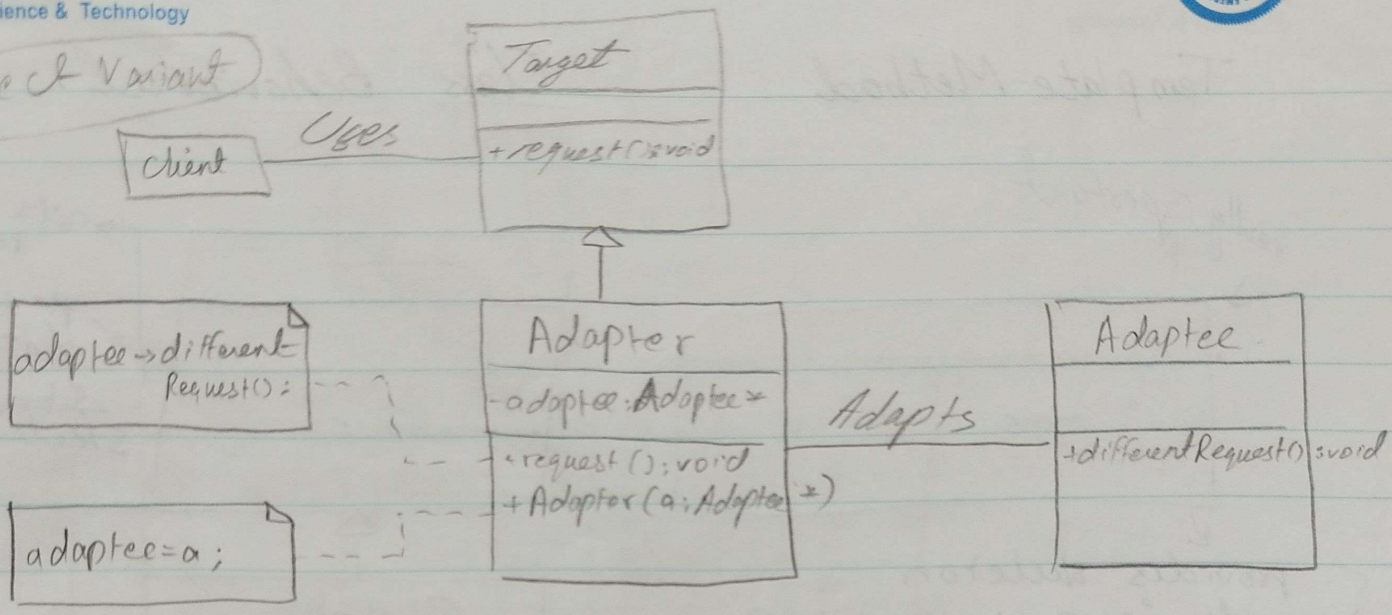
Class Variant



adapter is a target and an adaptee

Foundation for Advancement of Science & Technology

Object Variant



* Two classes want to talk to eachother but they can't becz of mismatch in interface so adaptor used to communicate.

Foundation for Advancement of Science & Technology

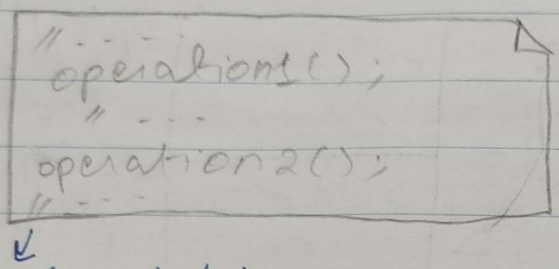
give interaction

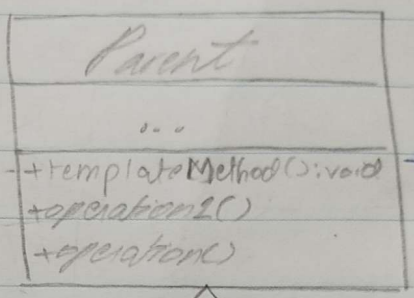
Template Method

Class Behavioral

scope purpose.

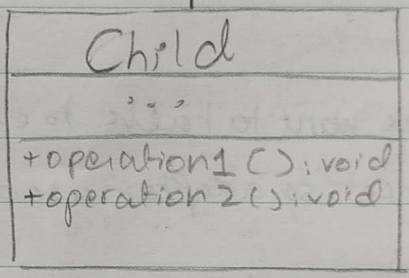
really important





acts as template

parent decide ~~Structu~~ skeleton



provides skeleton.
- order is very very imp.

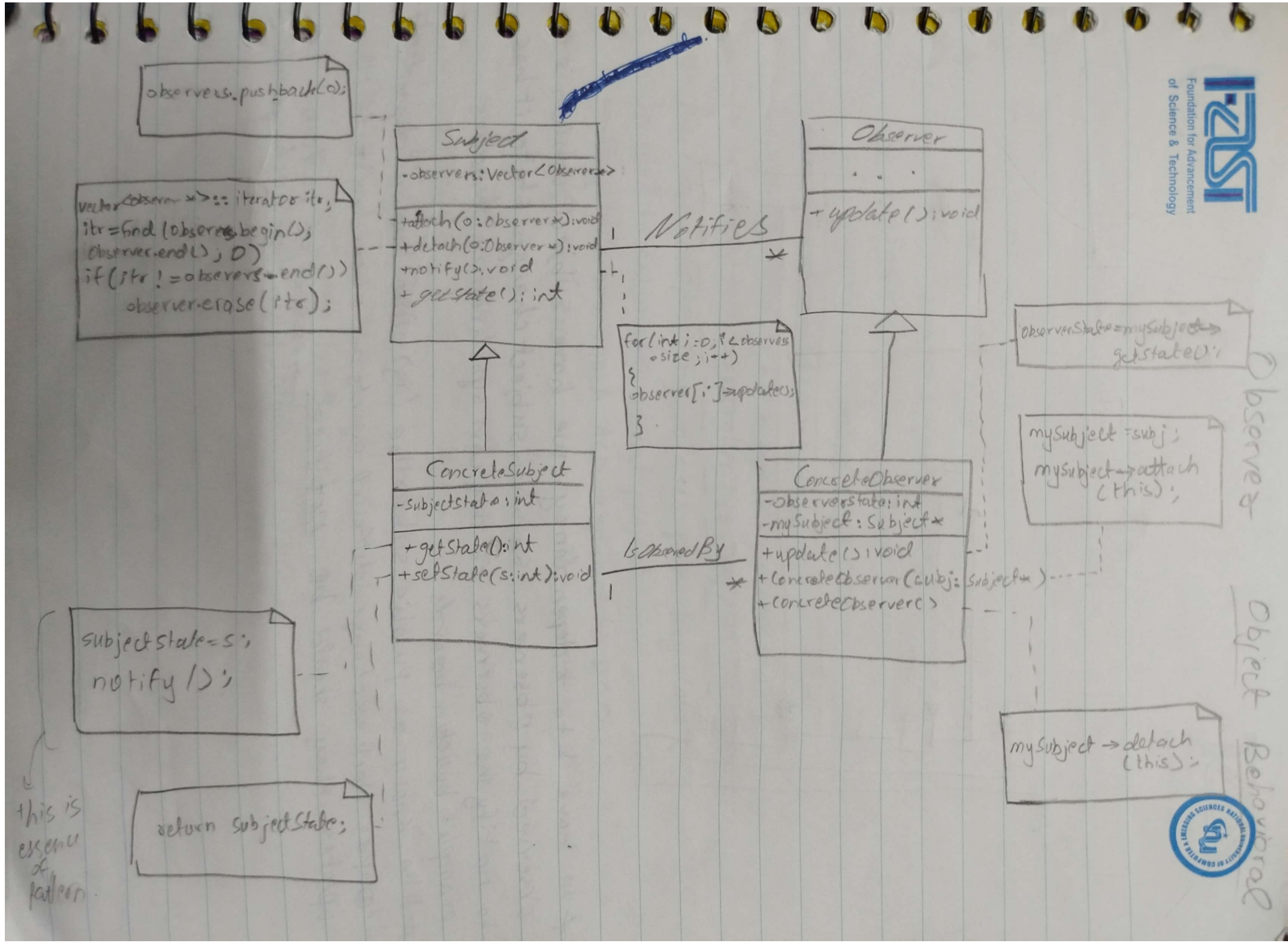
- You filter out common invariant parts in parent & you let children deal with vaiant parts.
  → invariant part is the sequence in which these functions are called. Operation 1 has to be called before Operation 2. whether its called in loop or if statement. Doesn't matter.

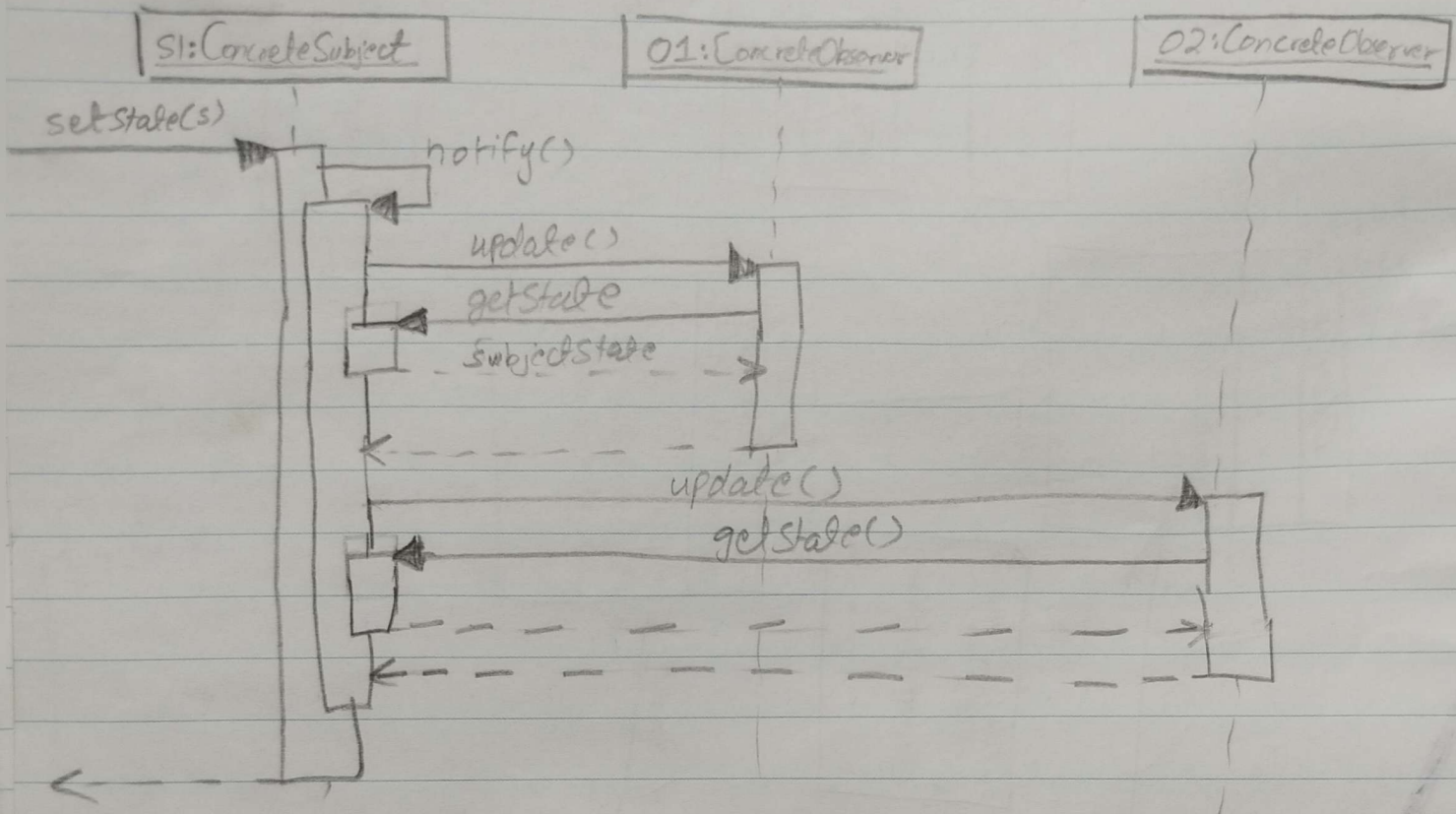
- You keep structure as it & may change it
- Its a good idea to make template method a non virtual function which cannot be overridden by child classes
- Don't let outside world access operation 1 & operation 2. so you can make them protected

input() process() output()

Take input array ↓ Sort ↓ output arr.

} the sequence of function calls cannot be changed but implementation can be changed.


observers.push_back(o);
Subject
-observers:Vector<Observer*>
+attach(o:Observer*):void
+detach(o:Observer*):void
+notify():void
+getState():int
Notifies
Observer
+update():void
vector<observer*>::iterator itr;
itr=find(observers.begin(),
observer.end(), 0)
if(itr!=observers.end())
observer.erase(itr);
for(int i:0,i<observers
.size;i++)
{
observer[i]->update();
}
ConcreteSubject
-subjectState:int
+getState():int
+setState(s:int):void
IsObservedBy
ConcreteObserver
-observerState:int
-mySubject:Subject*
+update():void
+ConcreteObserver(subj:Subject*)
+ConcreteObserver()
observerState=mySubject->
getState();
mySubject=subj;
mySubject->attach
(this);
mySubject->detach
(this);
subjectState=s;
notify();
return subjectState;
this is
essence
of
pattern
Observer Behavioral



* we have 1 to * dependancy. we have subject which is observed by observers. So when subject changes state, it has to notify the observers so they can synchronize themselves.
example = bulletin boards
* Also known as public subscribe (ig). Also known as dependants.
example: stock exchange. different agents are observing it. if it updates, they should be notified.
→ essence of pattern.
notify when there's a change in state

Foundation for Advancement of Science & Technology

Lorez & kidd (LK)

* planning require estimation of sources.
  ↳ we need estimate of how big system is.

1- Number of Scenario scripts (~~NSUS~~ NSS)
   ↳ count no. of use cases / ovals
2 - Number of sub-system (NSUS)
   ↳ count no. of packages
3- No. of key classes (NKC)
   ↳ use ACD to count no. of classes.
   more classes, bigger system
4- no. of supporting classes (NSC)
   ↳ persistant classes.
   ↳ std, faculty in flex are key classes.
   ↳ registration controller is supporting classes
5- Average supporting classes per key class (ASC) = $\frac{NSC}{NKC}$
   ↳ used to estimate current project.
   ↳ This tells how much ACD expand to DCD
6 - Class Size (csize) = NO + NA → no. of attributes
   ↳ no. of operators
   csize = no. of features but count the no. of inherited features too.
7- Depth (d)
   ↳ how deep class is. at top depth = 0. then it increases downwards
8 - Number of operations added (by subclasses) (NOA)
   ↳ only for subclasses
9- " " " overriden (NOO)
10 - Specialization Index (SI) = $\frac{NOO * D}{NO}$ → depth

* This is LK. it focuses more on size for project management.

Foundation for Advancement of Science & Technology

Chidamber & Kemerer (CK)

→ related to all classes

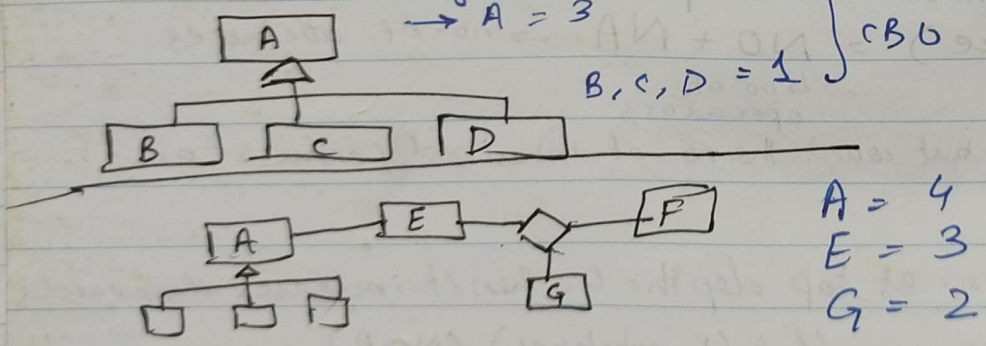
1- Weighted Methods per class (WMC) = $\sum_{i=1}^{n} C_i$
   ↳ sum of complexities of n methods
   ↳ assuming all methods have complexity = 1 (not possible in real)
   so WMC = n

* WMC & CSize are similar. WMC doesn't count inherited functions

2- Depth of inheritance tree (DIT) same as D.

3- No. of children (NOC) ~~for children not parent~~
   ↳ count of immediate children not grand children.

4- Coupling b/w objects (CBO)
   ↳ CK focuses on not only size but quality
   ↳ count no. of associations, inheritance etc. Just don't count unary



5- Response for a class (RFC)
   ↳ count no. of operations + no. of remote operations called by those operations
   (no. of external functions called.)

6- Lack of cohesion of Methods (LCOM)
   ↳ calc for every class.
   ↳ every class has its own LCOM value.
   ↳ most complicated

Foundation for Advancement of Science & Technology

$I = \{ i_1, i_2, i_3 \dots i_m \}$ ← set of all instance of class, all non static variables

$M = \{ M_1, M_2, \dots M_n \}$ ← set of methods of a class

$M_1 \rightarrow I_1 \quad M_2 \rightarrow I_2 \quad \dots M_n \rightarrow I_n$

$P = \{ (I_v, I_s) \mid I_v \cap I_s = \phi \}$ ← set of all pairs with common

$Q = \{ \quad // \quad \mid I_v \cap I_s \neq \phi \}$

$\frac{n(n-1)}{2} - \frac{|M|(|M|-1)}{2}$ → Total useful pairs

$= |P| + |Q|$

$LCOM = |P| - |Q| \quad ; \quad |P| > |Q|$

$= 0$ otherwise

LCOM can't be -ve.

we do pair wise comparison..

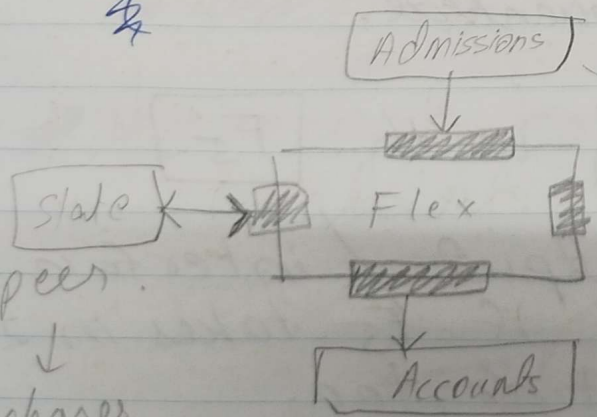
(I_1, I_1) → yeh nahi karna → not useful

(I_1, I_2) ✓

(I_2, I_1) → useless

Next Lecture

- Architecture is high level design.
- what was discussed till now was low level / component level.
- Architecture Context Diagram (ACD) (not a UML Diagram).
  (idea is to put system in context).



Admissions → superordinate ← tells which systems your system talk to

sends info to flex

State ← peers ↓ shares info

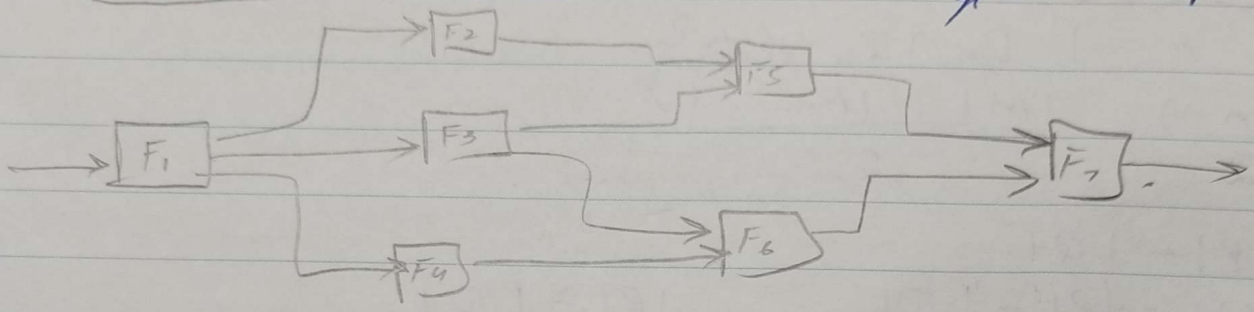
Accounts ← subordinate ↳ receive info.

Foundation for Advancement of Science & Technology

Styles → decide overall structure of flex.

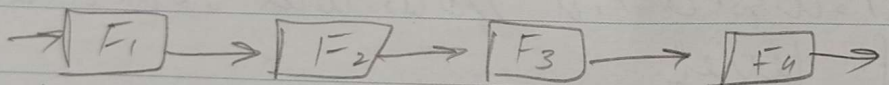
* many styles exist. (Not UML Diagram) (simple box & arrow diagram)

1- Data Flow → 1 example



These boxes perform processing.

* also called pipe & filter.
* boxes are called filters.
* arrows job is to transfer & called pipe.
* used in image processing or compilers.



↳ also pipe and filter] minimal user interaction

but its a simple case.

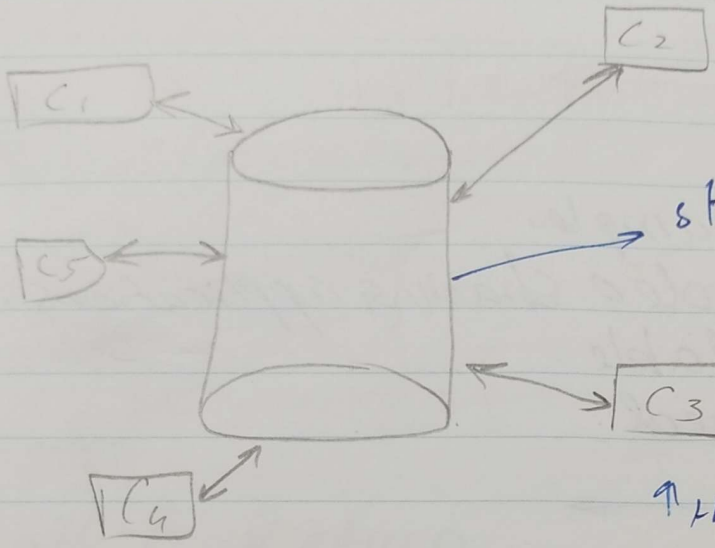
↳ It is a special case of PAF called Batch Sequential.

* I can replace [F5] with [F5']

but in ki input, output / interface same hona chahiye. If F5 takes in 2 arrows, F5' must do too.

Foundation for Advancement of Science & Technology

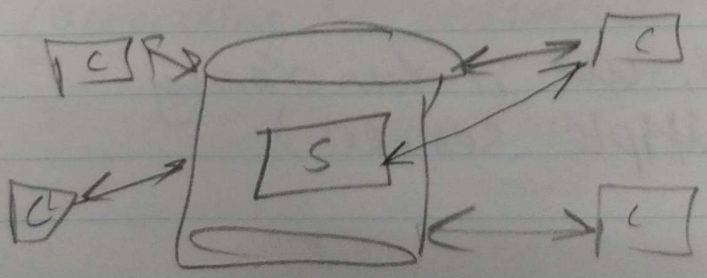
2- Data Centered.



↑ this was storage & retrieval

- They store & retrieve component from repository.
- Same as Observer Design pattern.
- if repository state changes, tell all Cs
- You can add as many components (Cs) as you want.
- Low coupling
- if data source fails, everything fails.
- used in mailing lists., bulletin boards.

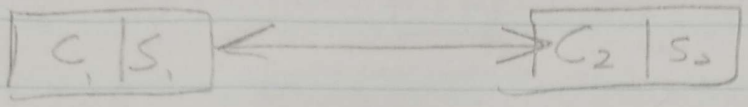
- special case client & server



← client req. service.
← server provide service.

Foundation for Advancement of Science & Technology

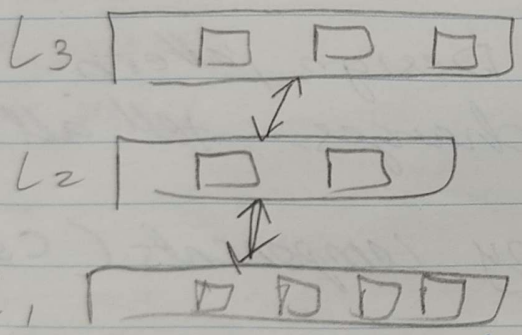
Simple case — Peer to Peer (P2P)



* bit torrent example.
  file, audio, video sharing applications.
* its very scalable.
* " decentralized.

⟷

3- Layered architechture (Three tyred) — ig. idk the spellings.



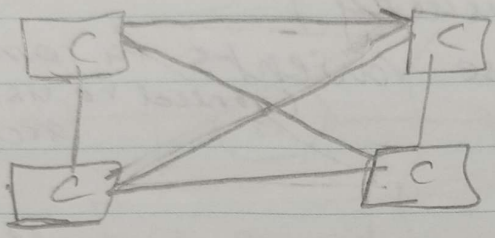
* still has client server thing going on.
* coupling is contained.
* OSI is example of layered.
* when security is imp. People use layered.
  ↓
  is Non func.

* if availability imp, use redundancy (have multiple servers)

Foundation for Advancement of Science & Technology

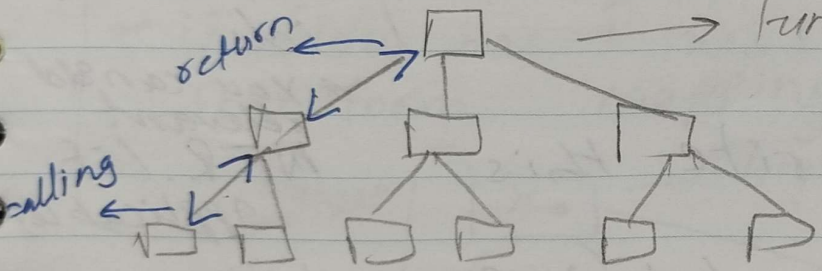
* some people also consider DCD an architecture.

4- Object oriented



these Cs are actual classes

5- Call & Return



main function calls other functions.

* these boxes are not at same level of abstraction. The level decreases downwards
* This is why its different from other architectures
* The decision makers are at top & the workers are at the bottom.

Architecture Styles & pattern. → limited scope
style has bigger scope
(according to pressmen)

Foundation for Advancement of Science & Technology

Documentation SAD

- Ar CD (1 page long)
- B & A architecture (box & arrow
  ↳ using 1 or more architecture design.
  (1 page usually)
  (you can use variants & combine them)
  (no need to use pure architecture)
- Description (table)

S#	CName	CDesc	Req. Ref → reference to requirement usecase ke ids.

component name ↑

* for every box write this
* you can add relevant NFR. (if applicable)

- Rationale (explain / justify why you chose the architure you chose)

New lecture

Architectural Derivation

Structural Design (SD)

Starting Point: lowest level DFD.
↳ most refined

ending point: Call and return architecture

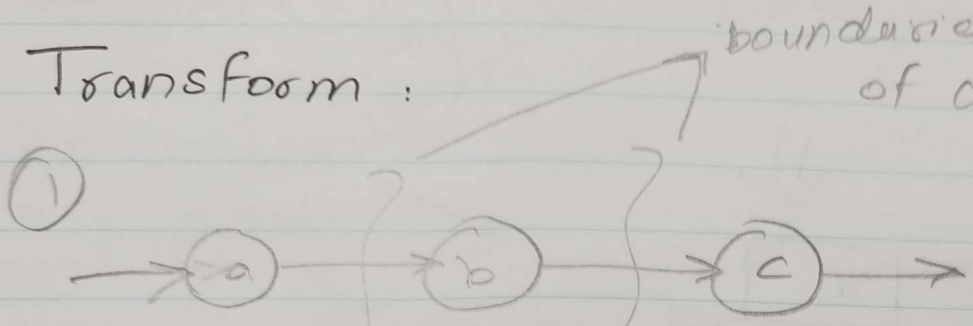
(SD)

Transform mapping

Transaction.

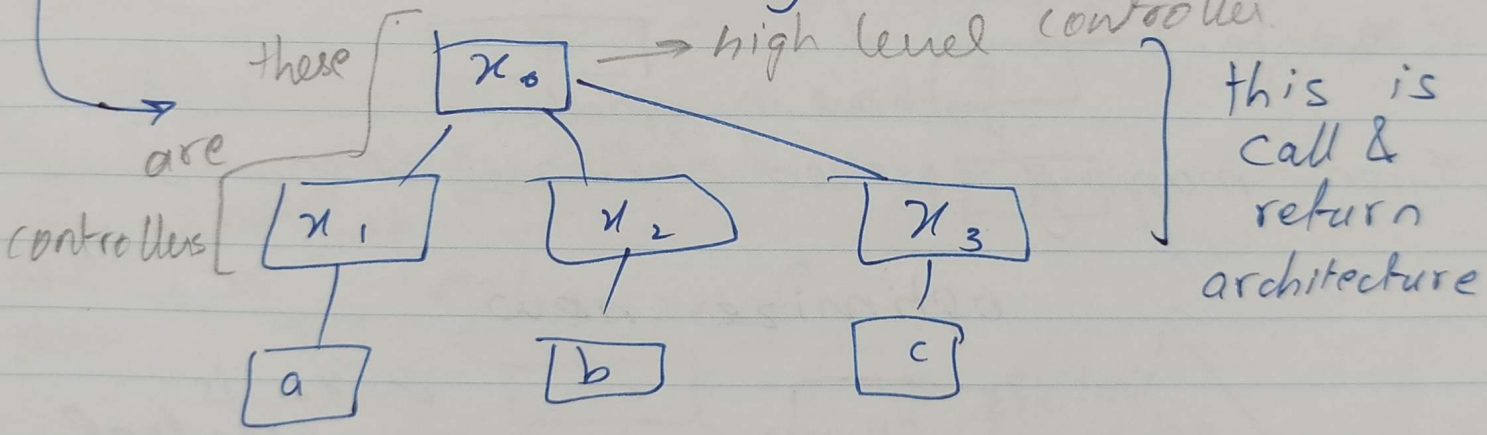
from DFD we can define if flow type is transform or transaction

Transform :



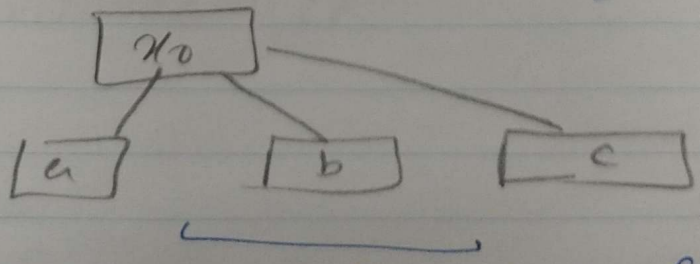
input process output.

* if flow has these 3 components use Transform mapping.



now optimize.

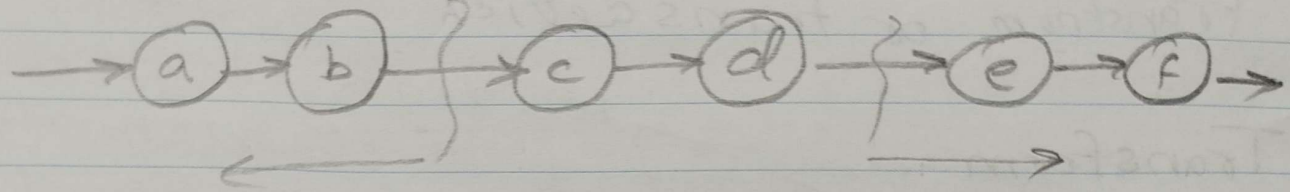
* since x_1, x_2, x_3 are doing nothing, optimize it by removing them



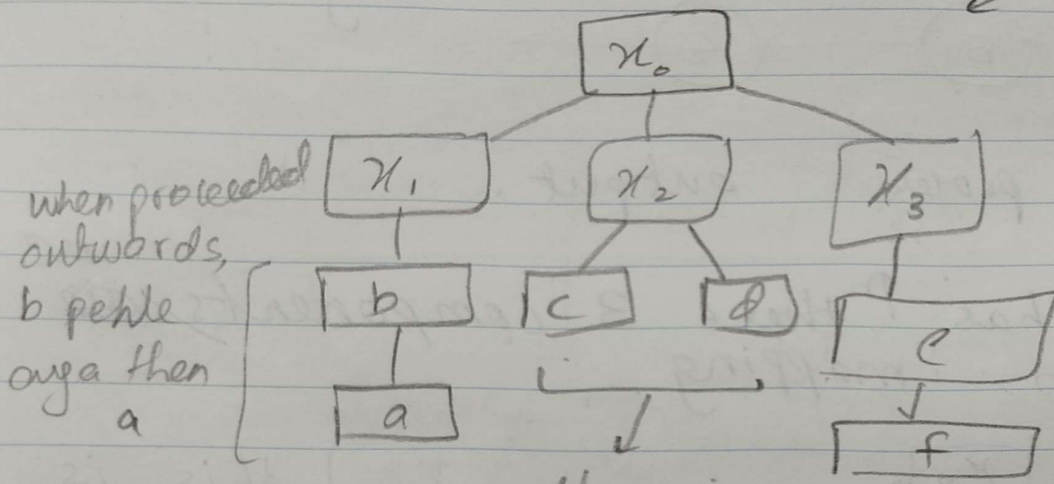
final architecture.

Foundation for Advancement of Science & Technology

(2) (example of Transform.



use x_0 at top & use $x_1, x_2 \ldots$



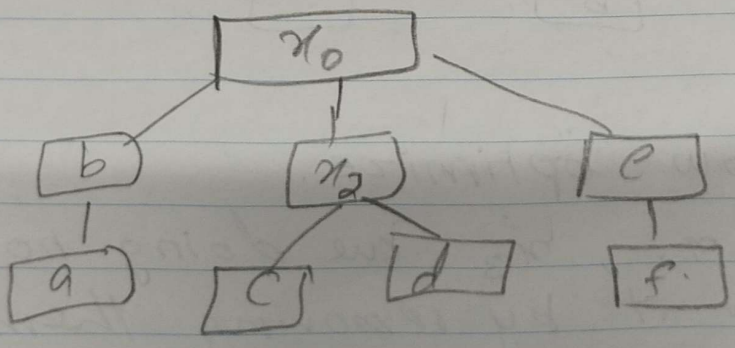
when proceeded outwards, b pehle aya then a

e pehle aya then f

both are in middle.

* for mapping proceed outward of boundary.

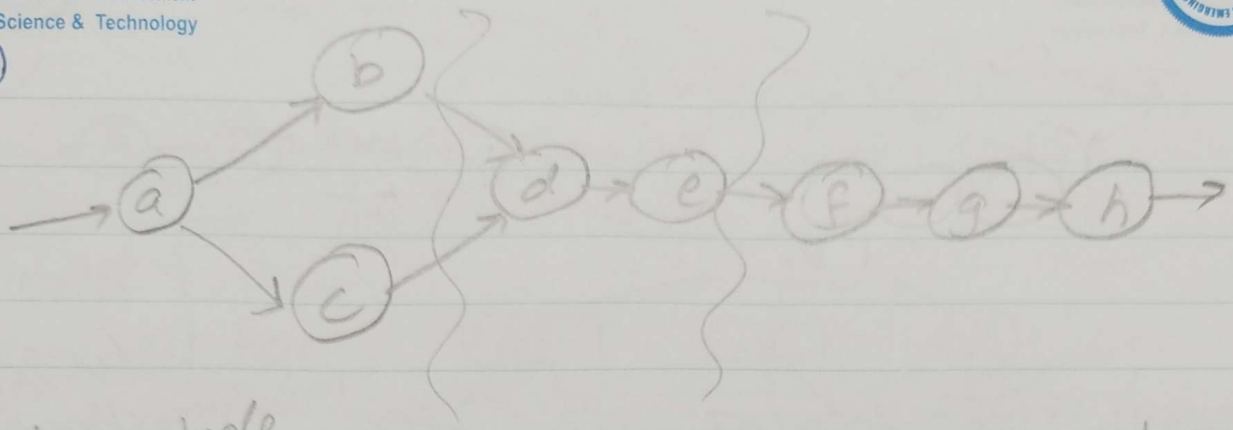
optimize now



x_2 is neeclect.

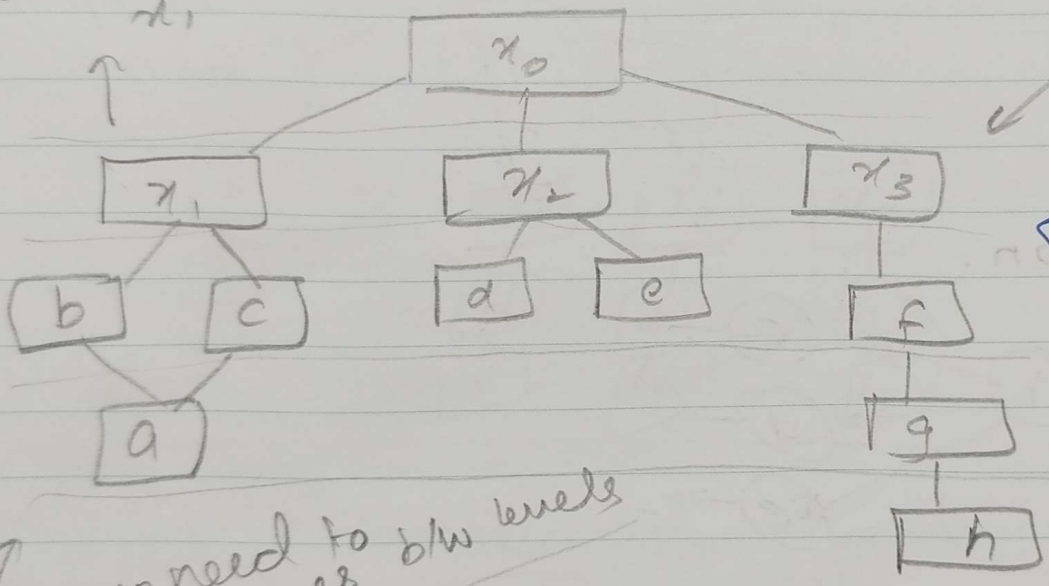
Foundation for Advancement of Science & Technology

(3)



can't implode x_1

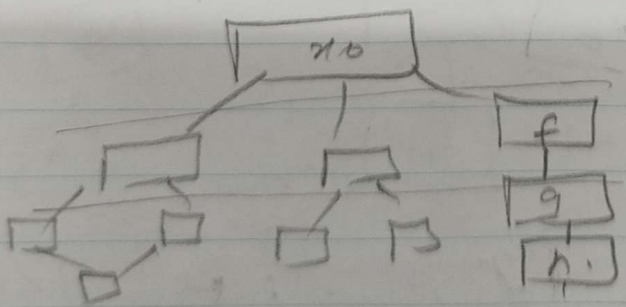
always 3 controllers at start.



x_3 remove it / implode it

align. no need to draw these lines b/w levels

optimized.



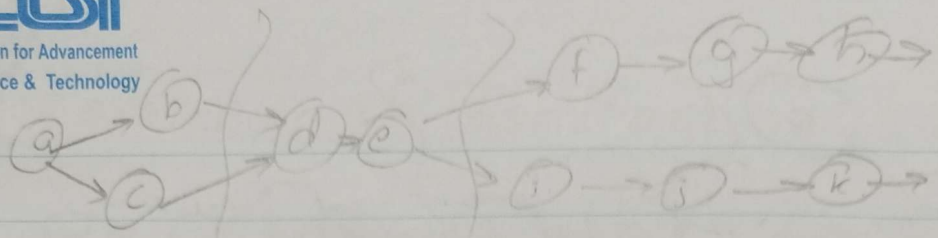
like line me ho

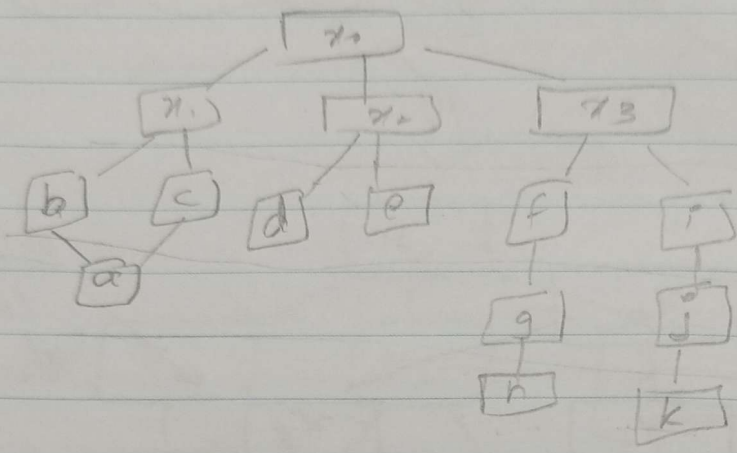
make sure all boxes are aligned

final architecture.

Foundation for Advancement of Science & Technology

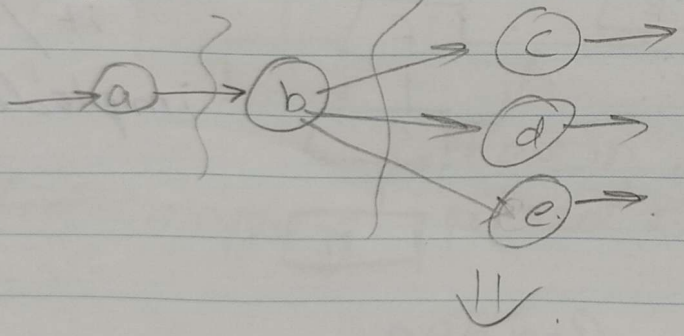
4

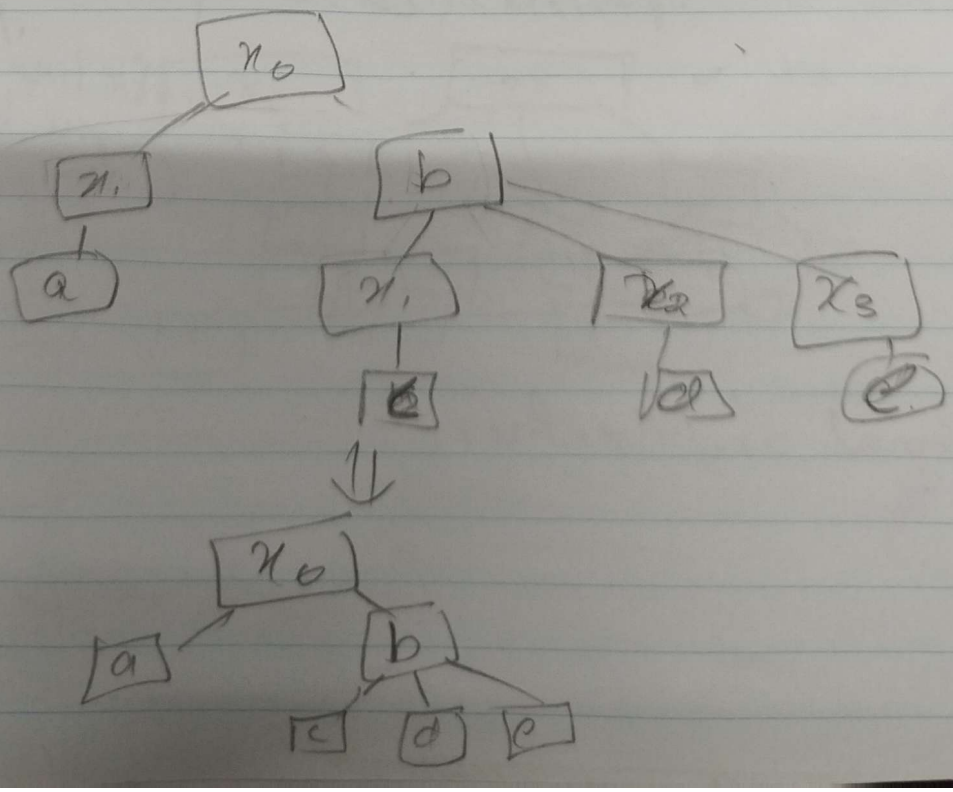




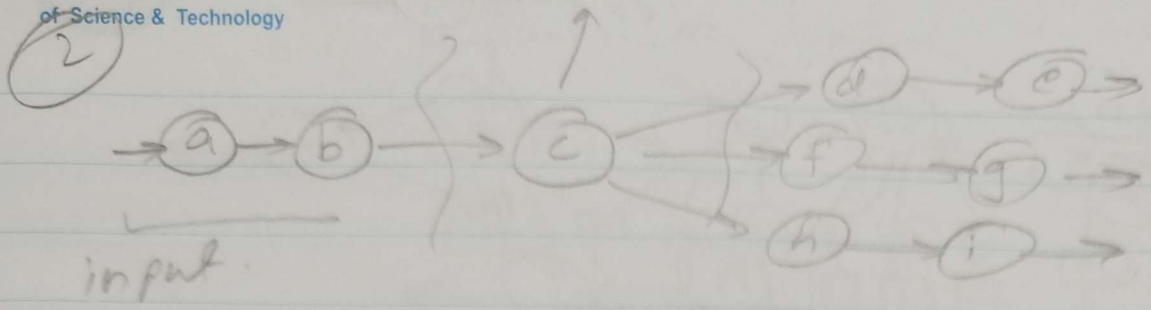
this is final since it can't be optimized further.

Transaction.





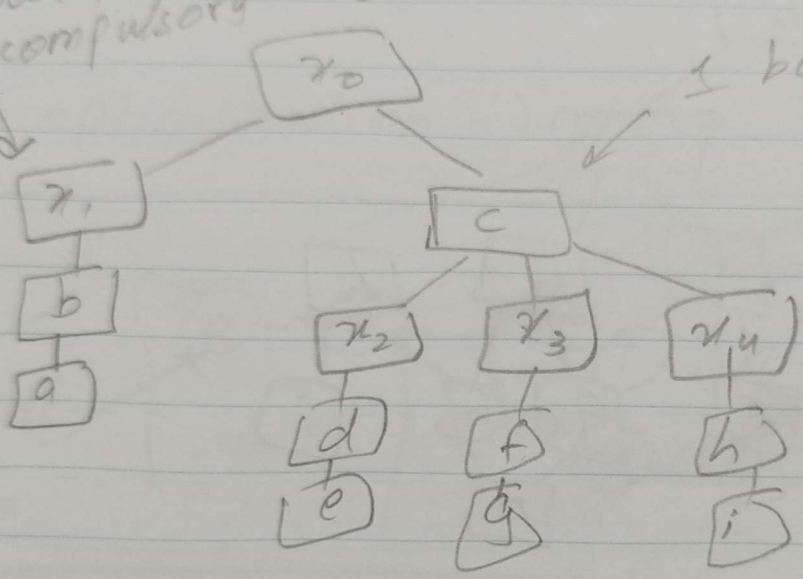
transaction center (dispatcher)

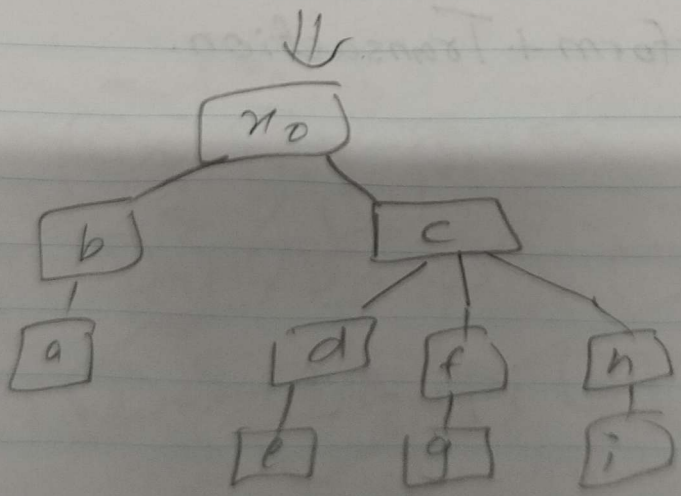


input

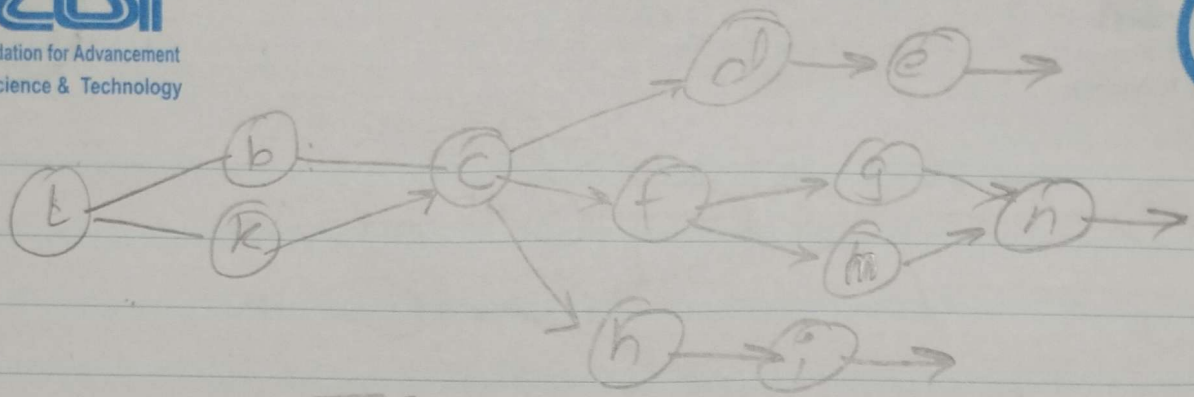
controller for input compulsory

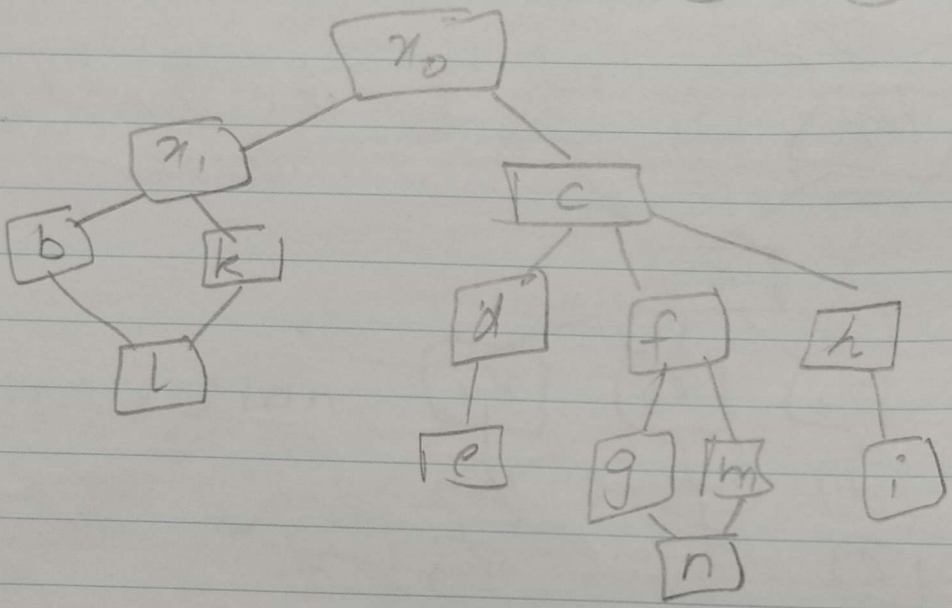
box for transaction center.





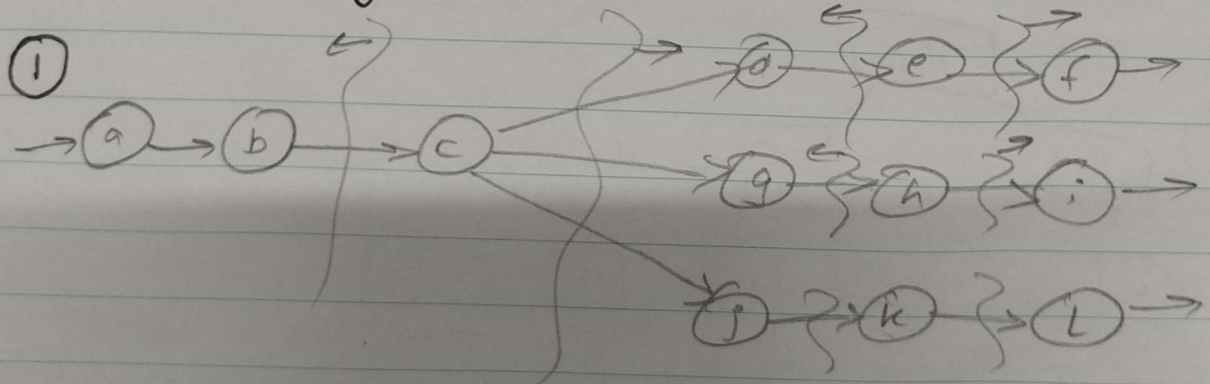
(3)





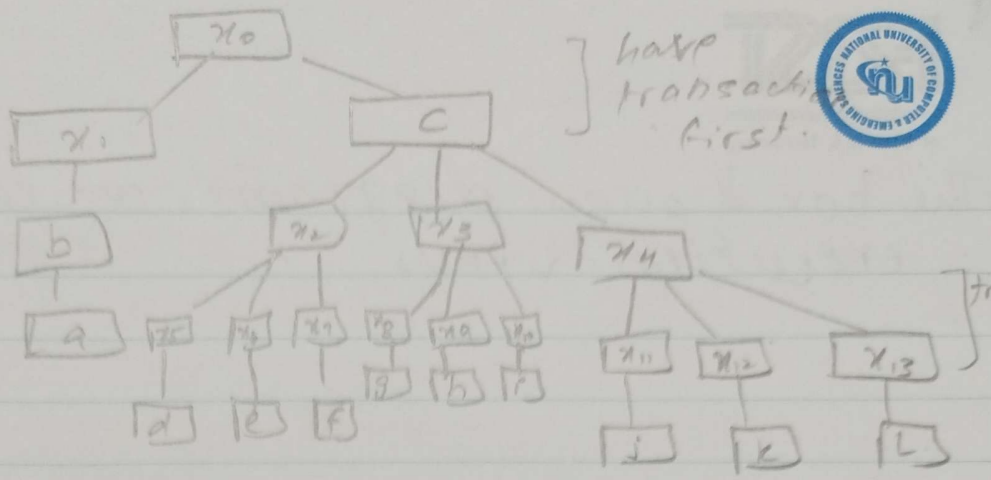
Now combining Transform + Transaction.

(1)



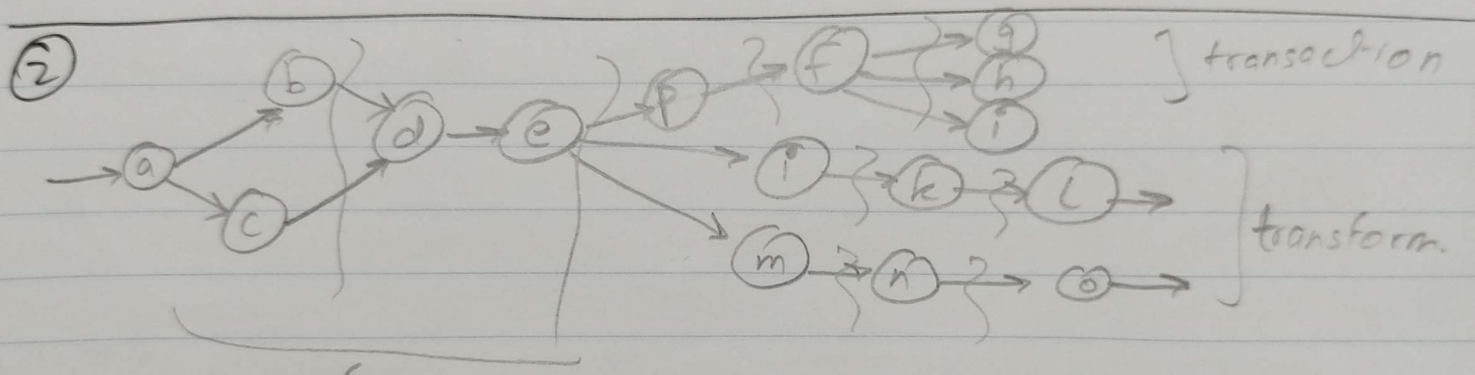
* read from left to right. First you encounter transaction

Foundation for Advancement of Science & Technology

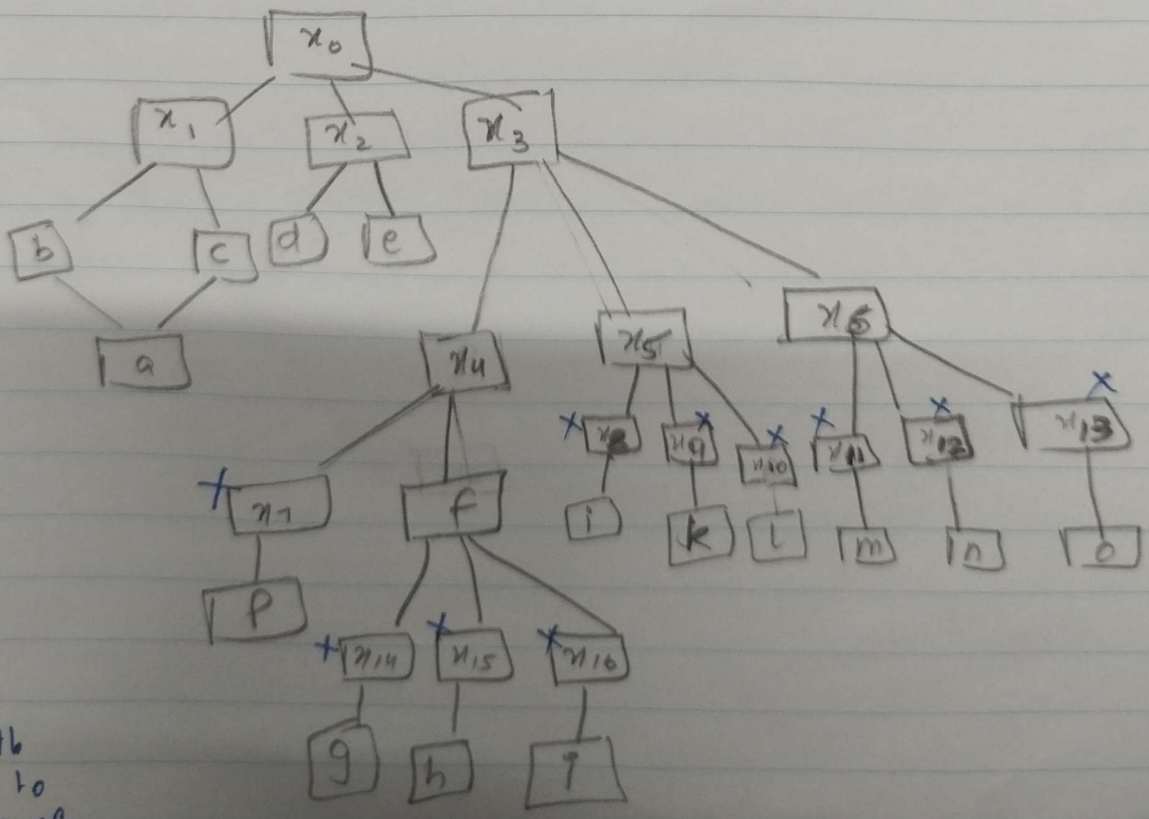


* for optimization = remove $x_1, x_3 \rightarrow x_{13}$.

(2)



transform.



cross with blue to remove