DDR

Date:

Sun Mon Tue Wed Thu Fri Sat

5.Sep.20

Lec 1

→ Why do we develop software? / Purpose of software dev?

It is to solve probs

Analysis → To understand dev about something written by someone.

Analyze → design → Implement

→ Technical Aspects of Software Dev

- Program constructs (compilation)
  - → Assignment (sequence) – expression (Numeric, String)
  - → Loop (Iteration)
  - → Condition (Selection)
- Program Organization
  - → Functions → Classes
  - → Procedures
  - → Folders → Files
  - (we organize to maintain)

CONCEPT

An idea or notion that we apply to things or obj in our awareness. Concepts are the recognition device (awareness hogi tou concept hoga or vice versa).

→ we recognize Things based on concept

Eg. Tangible → Person, car, pencil

Role → Doctor, owner, patient

Intangible → Time, quality, company (It is not a building, always on paper)

Relational → Marriage, ownership

Events → Sales, purchase, market crash

Others → strings, number, icon

Judgements → High pay, productive Job

CONCEPT TRIAD

① The term concept

② The intension → complete definition of concept

The test that determines whether or not concept applies to an object

③ The extension → The set of objects to which concept applies (aka examples)

→ If concept triad is incomplete, then:

- Concept without name → slows down communication.

- Concept without instance → has no value because one cannot create of that concept in anyone else's mind by giving examples.
- Concept without definition → they are not concepts, they are meaningless.

- Synonyms → concept with two names eg. client & customer.
- Homonyms → concept with two meanings eg. string (thread & type of var.)
  ↳ same word but different intention
- Type → A shared notion that applies to obj in our awareness (synonym of concept)
  ↑ recommended term for concept in obj oriented analysis community.

Object Oriented Concepts

bottom up → nothing is told, figure it

- Abstraction – Explanation/Description of concept whereas, Implementation is to follow that description.
- Class & Object – use top-down approach (when everything is told in an organized way); make class from object. Obj to class is a difficult approach; The key is to know the purpose/reason of creating obj/variable.
  → Have a strong reason for everything; create an obj & put a special identification on it.

Every object has 3 important Things:
(i) identity (ii) State at any particular instance (property of obj) (iii) behaviour

- Encapsulation – A compartment in which function, behaviour, fields & columns are put together.
- Info hiding – Restrict & protect data.
- Inheritance – Basing an obj/class upon another obj/class to retain similar implementation.
- Polymorphism – Ability of an obj to take on many forms.
  ① Adhoc – overload
  ② Subtype – override (pure OOP concept)
  ③ Parametric – Template/Generic

* we use overloading to fulfill a new request in middle of implementation.
* Info is ~~beta~~ hiding behind the interface.

- Interface – related to info hiding. A prog. structure/syntax that allows computer to enforce certain properties on an obj/class.
- Messaging – Interface provided to us & we call that interface.
  → function calling & parameter passing.
- Delegate (function pointer) – a part of language; not directly related to Obj Oriented.
- Relationships – to relate objects

Object Oriented Analysis

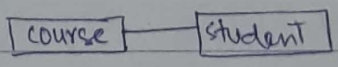
lec 2

Obj types are:

1. Person (role)
2. Product (anything produced) + service
3. Event (Result of event happening; Action association – time is important).
4. Spatial location
5. Temporal – time.
6. Organization – group.

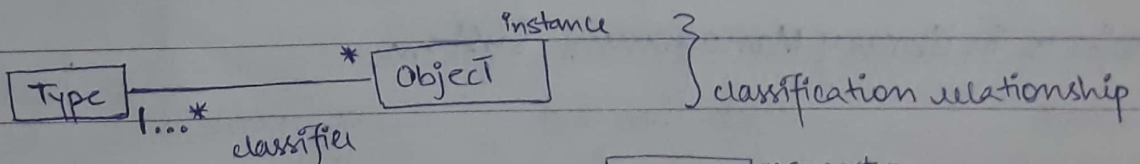
Relationships (Association, classification, Generalization, Aggregation, Composition)

→ Association – used to define a relation b/w two entities

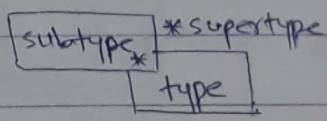


→ classification – to put an obj in a category.

It is the act/result of applying a concept (type) to an obj



→ Generalization & Specification



An act or result of distinguishing a concept (type) that completely includes or encompasses another

→ Aggregation/composition → an obj is contained in another

collection in which can be made by putting different objects.

→ The act/result of forming an obj whole using the other obj as its parts.

→ Generalization & classification — two concepts & have different hierarchies

→ Obj are instances b/c they have been classified.

→ Types of obj are subtypes b/c they have been specialized.

→ classification — relationship b/w type & Object.

→ Generalization/Specialization — relationship b/w type only.

✓ Generalization → Generalization → classification

✗ Generalization → classification → classification

Links — Relationships (UML Notation)

——▷ Inheritance	——→ Association
◆——→ Composition	- - -→ Creation
- - -▷ Realization	—— Association
◇——→ Aggregation	⊳—<

Class is a box ☐, when we've point out something, it is an object.

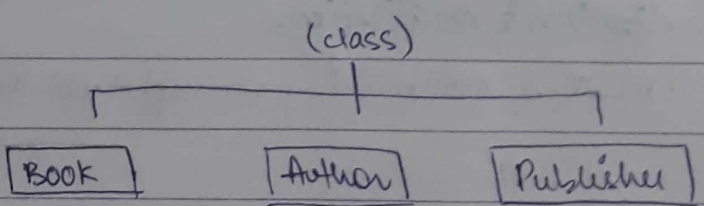
→ Objects in Library Management System

→ book → registration card → lib. members

→ shelf/section → staffs

Book:

- ISBN - Title - Author
- Publisher - Edition



(Book is an abstration); jis jagah significance hai, wo aik physical copy hai, warna it & will stay as an abstraction.

↓

if library has a book available

Date:

Sun Mon Tue Wed Thu Fri Sat

Issuance → event created in which their will be members, staff & date.
We'll create object of event.
Also along with that, we'll create 'return' event.
→ leave an empty field in issuance so that when it is returned, that field can be filled.
→ time is most important when an event occurs (Date + time)

→ It is important to know the limitations/conditions as well.

Validation & calculation → These are two main purposes for creating a system.
* Search & sort are part/type of calculation.
→ display/print/show are for secondary purposes.
→ to identify classes & obj & to find relationship b/w them
  • better relationship → better design.

→ If we have good analysis at the start → we can easily make changes without much efforts.

Program Structure

• Don'ts for structured prog
  → Goto (It ruins the follow of prog; difficult for programmer to trace).
  → Break, continue (unless switch statement)
  → Return (other than last line in a function)

• Side effects to avoid (minimize dependency)
  → Global variable
  → Static variable & obj (unless initialized once)
  → Global object

- Don'ts for better programming practices

① Getter & Setters (unless they have a purposeful name) b/c it violates info hiding & its ~~und~~unconditional — no restrictions

- If there is a purpose then make it else NOT.

② Mutation (don't change any obj)

eg. passed/sent an array with 5 val. & comes back with 7.

Objects & Classes

We should know:

① Obj recognition (changes)
② class Creation
③ Top down approach
④ Bottom Up Approach
⑤ Abstraction
⑥ Dependency

Obj Dependency:

There are ways an obj can be used. It depends on:

→ which class is creating an obj
→ when the obj is being created
→

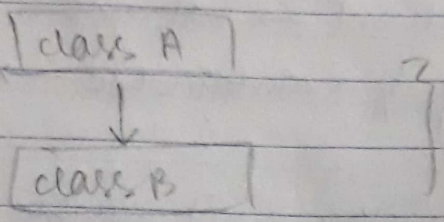
- Scope

→ The obj is defined at class level or method level

Creation → where & why is it being created. Which class is creating

Sun Mon Tue Wed Thu Fri Sat

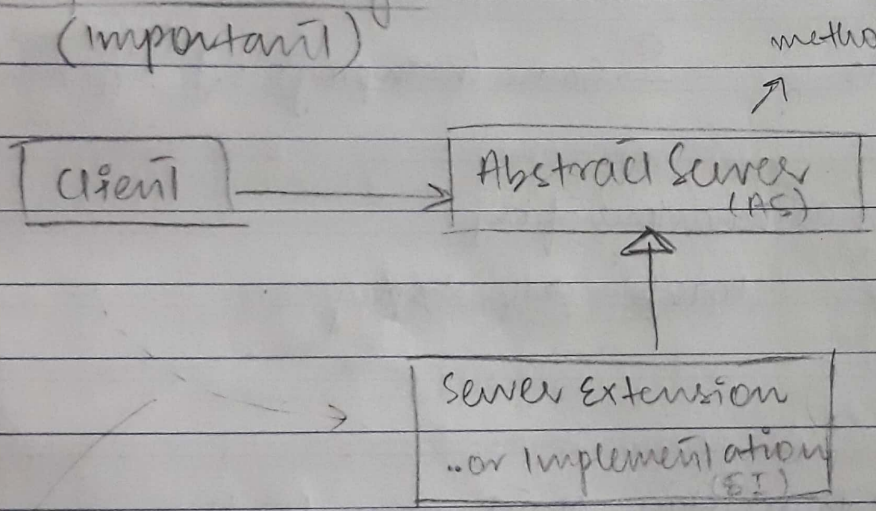
Example



class A is dependant on class B.

* It is imp to know whose responsibility it is to create a class

Software Design
(important)



our dependency should only be on behaviour

```
AS as = new SI();
```

*This was the dependency that we want to show but we don't do this directly

Server — that serves something. , client → That needs to be served.
lesser dependency , more reusability & maintainable → so more abstract.

- Reusable design. — Multiplication function is not reusable — its recallable.
- Implemented method — jiska code likha wa hoga
  (If design is abstract — it can be reusable — only a design is reusable.
  (↳ It is never reusable!)

→ Game Tic Tac Toe's objects:
① → players ② → boards ③ → Game (main prog) ; ↓ top down approach

All exposed methods are known as behaviour of obj.
→ ~~players: play this turn / / ; board → clear, display~~

Obj: → behaviour
Game → run
Board → checkstatus, addplayer, start, abort, move, finish
Player → register

* Monolithic → UI & logic is tied up together
Board AddPlayer(Player P) { ... p.register("x") }

(some internal method) next
{ string symbol = find → symbol
s = P.checksymbol()
P.register(symbol) }

when you make a class, think before who will call it & what it'll do.

Game
Run()
{ Board b = new Board();
Player p1 = new Player();
Player P2 = new Player();

b.checkstatus()
b.addplayer(p1)
b.add player(p2)

* board is initialized without a player so cannot start if we run
b.start()
b.move(p1, 5)
b.move(P2, 3)
b.move(p1, 3)

Player
checkstatus()
Unregister
→ (game board initialized again)

if finish or abort

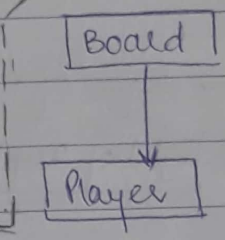
Date:

Sun Mon Tue Wed Thu Fri Sat

testing → run different testcases.

→ The algorithm is the rule (b/c conditions & process to follow)

→ If we want to add a ludo player, we'll make a new class

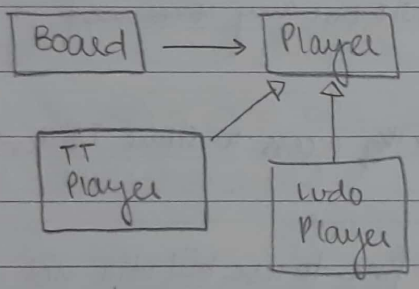


player dependant on board.
player has reference address of board;

To implement rules:

① Place a condition (if-then-else, switch) — modifying a func. whenever there is a change → increase complexity
(ruins OOP) — (we should not do this)

② Make a new class



OOP approach
→ Bottom up approach
→ Player is the abstract class.
→ implementing abstract class is usually Bottomup approach
→ TT Player, Ludo Player are concrete class

```
Board b = new Board();
Player p1 = new TTPlayer();
Player p2 = new LudoPlayer();
```

If we apply OOP & not if-then-else, so we'll change the code.

→ every concrete class will have a different behaviour & rules but same general name i.e. player
→ by using OOP approach, we'll omit if-else & implement rules.
→ to make abstract class; then make concrete classes & then extend (inherit) further classes from it → This is reusable approach; to reuse a design

Date:

Sun Mon Tue Wed Thu Fri Sat

Coupling — degree of interaction b/w 2 modules (one func calling another func) or behaviour

→ if tightly coupled — we avoid.

↳ depends on degree of coupling.

① Content coupling → undesirable } maintainability cost max.

if one refers to inside → internal / private part of some other. ~~data~~

(↳ don't use friend func.)

② Common coupling → if they refer to global data

global variable is a shared variable without any ~~concerned~~ concurrency control

③ External coupling → if they share direct access to same I/O device or are tied to the same part

if concurrency rules are there → it's fine to use but still not desirable.

④ Control coupling → it is common but still not desirable.

one algo is controlling a piece of other algo (kinda switch, menu).

- not automatic, baki ka parameter control kr raha.

⑤ Stamp coupling → better ; if we are providing the data but data is stored somewhere else in the memory but we're passing a reference of the data (eg. passing address of an array).

- there is a chance that the func can overwrite ~~the~~ at the memory address
- the called function should have the ability of being immutability.

Date:

Sun Mon Tue Wed Thu Fri Sat

⑥ Data coupling — safest & most desirable — loosely coupled.

- if one can call the other directly and communicate using parameter.
  — a simple list of input & outputs.

(external koi chez nai dei ; validate → compute → result) — no effect on other module.

Cohesion

Calling multiple functions under a function. The relation b/w those functions should be strong.

① Functional — most desirable — strong cohesion.

Function is doing just one job — single responsibility.

eg. read transaction record.

② Sequential — whose ~~activities~~ elements are involved in activity such that the output of first goes in another (input data). Sequentially called, one by one, using one another's data. One should be completed in order for other to happen.

③ communicational — dependency is lesser.

→ Sharing same data but not eachother. Elements contribute to activities that use same input or output data.

Eg.

④ Procedural — follows a sequence ; no data sharing.

whose elements are involved in different & possibly unrelated activities in which control flow

Eg. cook meal & get ready for party.

Date:

Sun Mon Tue Wed Thu Fri Sat

⑤ Temporal — one whose elements are involved in activities that are related in time; sequence nōt followed (some degree of relation b/c time is imp). At certain time, some routines may run.

⑥ Logical — controlled coupling se logical cohension achieve krtay hain
→ No data is being shared, sab bahir se control ho raha.
→ No relation in b/w.
→ Eg. menu items, mouse click se run kr rahay.

⑦ Coincidental — when routines are nōt related & no data sharing.
No seq, no interaction — no cohesion āt all.

Date:

Sun Mon Tue Wed Thu Fri Sat

Symptoms of Poor Design (these should be minimum)

- Rigidity — design is hard to change
- Fragility — design is easy to break
- Immobility — design is hard to reuse
- Viscosity — it is hard to do right thing
- Needless complexity — over design
- Needless Repetition — Mouse abuse
- Opacity — Disorganized expressions

Requirement ke naal:
what? — defines scope
how? — imp to follow rules
↳ in it ki is imp to know can estimate things from it.

(pehli dafa kam krke estimate nai kr saktay — need to have a little experience)
→ need to perform lesser program — reprog/ rework is considered bad.
↳ we should do testing / debugging for better performance

Rigidity — hard to change b/c every change forces another changes in pain of sys. (dependant modules)
The more modules must be changed → more rigid the design is
Eg. dependency on concrete classes (changing an API in a class, then we'll have to change that API everywhere in the dependant classes)

Fragility — Changes cause the sys to break in places that have no conceptual relationship to the part that was changed.
→ The tendency of a system to break in many places when a single change is made — new probs are created.
→ On every fix, the software breaks at unexpected places
Example: — dependency on concrete classes., no edge cases or bound checks.
↓
null point/exceptions occur if obj are not handled properly
↓

at the source wo mention krein that prog will not cater null obj — handle the exception at that point only — don't take that null obj further.

COPY

→ for knowing the type of a variable, know its purpose first, because underline type might be same but actual-type is different.

Immobility — hard to disentangle the sys in components that can be reused in other sys.

→ It contains part that can be useful in other sys but effort & risk to seperate the code is high — cost of rewriting is less compared to risk to seperate those part — reusing is not prefered.

↑ the useful modules have too many dependencies.

So its better to use abstract class rather than reusing code seperately.

eg — too many constant, global variables, coupling.

→ * reusability means to move abstract classes.

Viscosity — doing right Things is harder to do than the right Thing.

→ when design preserving methods are more more difficult to use than the hacks (to bypass the code), the viscosity of design is high.

• Hack is cheaper to be implemented than the sol within design.

→ when development environment is slow & inefficient, developer will be tempted to do wrong thing (choose easy way, use hack).

→ facing resistance while coding / not smooth. because many validations.

eg. using public member field, highly complex design, low perf. code.

Needless Complexity — design contains unfastructure that adds no direct benefit, many constructs added that are not used once,

↓ this makes the code complex + difficult to understand.

eg. too much generic code, long hierarchy of interfaces & classes.

Needless Repetition

→ design contains repeating structure that could be unified under a single abstraction.

Date:

Sun Mon Tue Wed Thu Fri Sat

→ The prob is due to developer's abuse of cut/paste of code.
→ It is hard to maintain & understand the sys with duplicate code.
eg. make use of switch statement, duplicate code on different classes.
• when coding, also make test cases (loop case, if/else)

Opacity

It is hard to read & understand; code doesn't express its intent well.
^too many comments in the code (hard to read actually code)
→ code is not consistent (equal spacing, && or AND – use one); code is written in a convulated manner.
eg: → too much commented code, ~~too much~~ incorrect name convention
→ algo written in a complex ~~code~~ manner.

Signs of Good Design (Opp. of Design)

- Adaptability, Robustness, Reusability, Fluidity, Simplicity, Terseness {unneeded duplication of code}
Perspicuity (clear + concise + organized)

DESIGN PRINCIPLES — SOLID

Single Responsibility Principle (SRP)
Open Close Principle (OSP)
Liskov Substitution Principle (LSP)
Interface Segregation Principle (ISP)
Dependency Inversion Principle (DIP)

→ Way to think, design & implement code; difficult practice

A class should have one reason to change, or a same type of reason.
Responsibility — a reason for change (an axis of change)
→ when req. change, that change will be manifest through a change in

Date:

Sun Mon Tue Wed Thu Fri Sat

responsibility among classes

→ If a class has more than one ~~reason~~ responsibility, then there will be coupling of responsibility & it will make the code fragile – easy to break.

Exposed behaviours / method of class are the responsibility of a class.

Jab bhi use change now → do not put it in if/else, make a seperate class & mention that use in the function of that class

Open - Close Principle

→ software entities (class, func, module etc) should be open for extension but close for modification

→ once in production or many classes are dependant on it, so don't modify it

→ defect nai to change it b/c needs to be fixed.

Open for modification → means that behavior of module can be extended

Close for modi → extending the behaviour of a modu'e doesn't ~~result~~ result in changes to the source or binary code of module

* maan concrete class bna'na is an extension, nai abstract class same rahany gi

Eg.

"An implementation is either following or violating the principle & Why?"

→ If requirement gets changed, we don't modify existing code or design. we write a new code / design.

→ Agr switch statement validation ko lkssay main deal rahi so its violating the OCP. If we're extending, its not violating

COPY

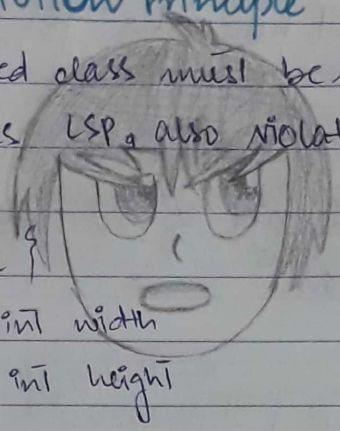
Date:

Sun Mon Tue Wed Thu Fri Sat

Liskov Substitution Principle

Subtypes / derived class must be substitutable in the base code.

→ What violates LSP, also violates OCP in background.



```
Class Rectangle {
    private int width
    private int height

public Rectangle (int h, int w)
    { width w ;
      height w ; }
public void setWidth (int w)
    { width w; }
public void setHeight (int h)
    { height = h; }

public int calArea ()
    { Area = width * height; }
```

```
public Square : Rectangle
    private int side;
public square (int s)
    { base (s,s) ; }
public setWidth (int w) {
    base.setWidth (w)
    base.setHeight (h) }
```

```
TestforsubsAbility (Rectangle r)
    { r.setHeight (h)
      r.setWidth (w)
      (calArea() == 20) ; }
```

↑ test case which is failing

(1,2)

```
Rectangle : myrectangle = new rectangle();
    { TestforsubAbility (myrectangle)
      Retangle : my square = new sq.(3)
      TestforsubAbility (mysquare); }
```

→ remove these two when you apply delegation

COPY

Implement kriya par test case fail hoa raha for the derived class so This is violating LSP.

→ Concrete class se extend kiya → this is a prob → DONOT DO THIS

* base class change nai hoti aur uski waja se derive class modify hoti hai

method overwrite krna pata hai.

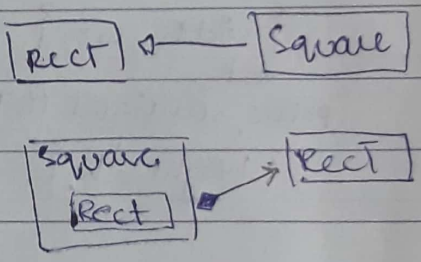
rules violate hoa rahay.

→ testcase agar koi 1 bhi aisa hoa jo fail ho so the design is not workable. no matter how many others are passing testcase.

Delegation is a process

```
class square {
    private rectangle r;

public square (int s)
    {r = new rectangle(s,s)}
```



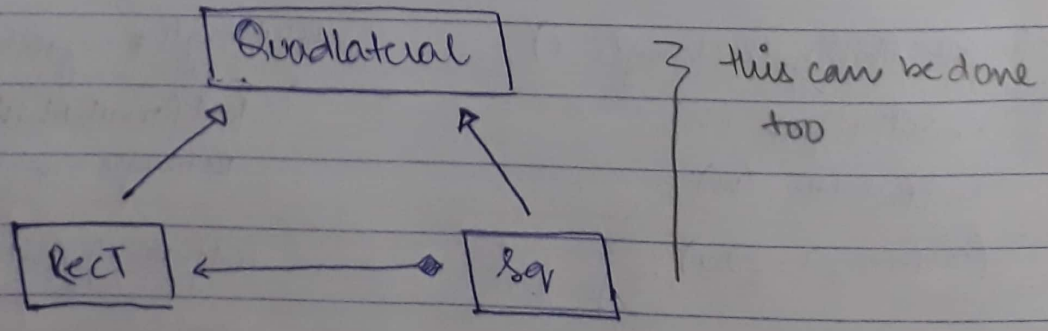
Advantage:

→ koi majood hai jo saray kam kr rahe hai.

→ jo methods chahiye sirf wohi expose krwayein.

* Delegation is a type of composition ; composition is a generic term.

→ Rect mein already saara kam hua wa hai. Hum sirf ab oneline se methods call krwa rahay jo humein chahiye.



Date:

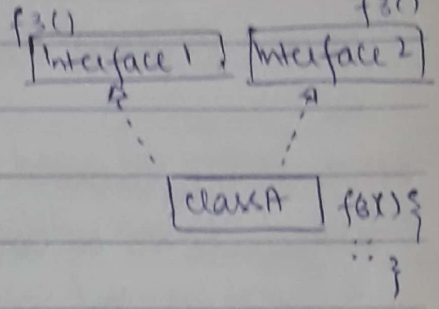
Sun Mon Tue Wed Thu Fri Sat

Interface Segregation Principle

→ Interface belongs to class, not hierarchy.
ait interface ki implementation ait hei class mein kra chahtay hain.

→ we want to realize an interface & not inherit.

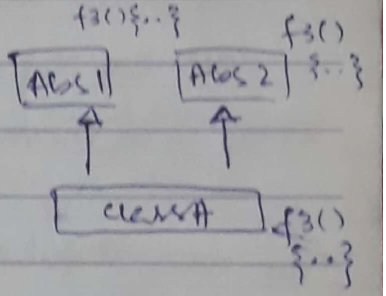
→ all methods should be implemented



→ restriction is that method implementation is not there in interface. It's implemented in class A.
(kiska hai, pata implement nai kia wo)

→ If we make 2 abstract class instead of interface, so functions implemented in the abstract class & if we call func, so which one will be called?

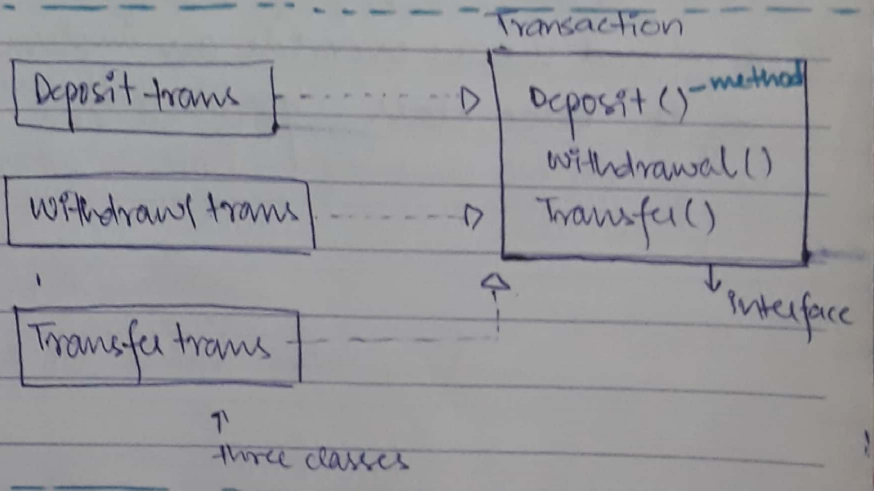
* SRP is being violated. bc 2 set of func are being implemented.



→ Interface is not Obj Oriented ; it is used to avoid undesirable behaviour.

→ ISP ka purpose is that if we've made a big interface (sara kam us (all methods) mein daal diya), so

Agar deposit trans. avail krein ge tou transaction interface sab methods dega par ap usmein se withdraw aur transfer implement nai kr saktay; to why add it all in 1 interface —?



• Iss say interface se realization hei na krein jis se ap method implement nai kr saktay.

→ Solution: break the interface ; if ISP is being violated

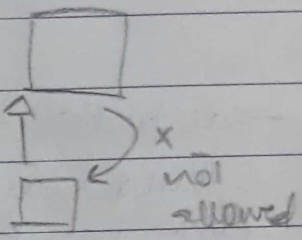
* It is imp to know the purpose.

Date:

Sun Mon Tue Wed Thu Fri Sat

Dependency Inversion Principle

→ high level module should not depend on other (low level) modules → hierarchy
→ Abstraction should not depend upon details
→ base class should not ~~depend~~ call any func of its subclass
↓
should depend on abstraction



① • No variable should hold a pointer or reference to a concrete class
↓ abs class ka reference banayein aur usmein concrete class ka obj daal dein
• obj is in memory; The obj written in code is reference

② no class should derive from concrete class
→ b/c concrete class can change → too much changes krni pdein gi
→ can derive from abstract class
→ If there's an additional field, add that in constructor parameter.
→ class ki state frequently modify nai honi chahiye (agr hoga to kisi method ke through)

③ No method should override an implemented method of any of its base class process