Notes. Low Level Design (Keerthi Purswani)

(Lecture 2) → (Relationships)

Last Lecture we saw an example of a kind of relationship.

That was (is-a) relationship.

1) "is-a" Inheritance

Order → Buy order.

Piece → (Bishop, Queen, King)

Now while inheriting attributes from the parent class we will also have to see, that what are we adding to the derived class.

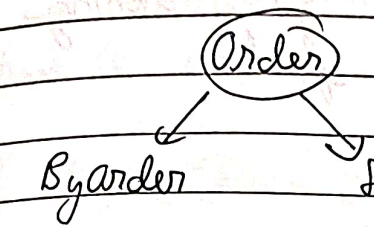
Like for Zerodha.

Order ⟹ { ≡

Buy order

In Buy order we have to add Buying Price, ~~Selling Price~~ etc attributes

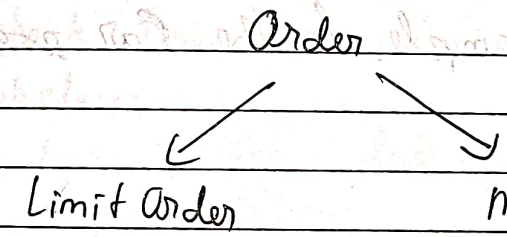
Keerti says "While coding out Zerodha



She was wondering what extra attributes are we adding. As it comes to us naturally to make this inheritance".

Hence we should not force inheritance everywhere.

In Zerodha again there is an option of.



Both will have different implementations.

Limit Order ⇒ Says, sell the stock at the price user has set. (Obviously lower that current price).

Market order → Says, sell the stock at the Market price.

Argument given by Keerti "She says we could just have a attribute type in both the order types and not need inheritance at all.".

"We should have good enough reasoning and we should not force feed".

2) "has-a"

Order has Delivery Address.
Order has Item/Product
Car has Engine → Car has (seat)
Game Chess has Pieces
Uni has Dept
Uni has Fest

Take the Uni example & the Car example.

(Uni example)

University has to have a Dept. Without Dept university cannot exist

By Dept can exist independently.
Fest can also exist independently

(Car example)

~~Engine cannot exist~~

Car cannot exist without Engine.

When we are talking about the "has-a" relationship, we have to check.

If ~~SuperClass~~ subclass can exist without a superClass.

If SuperClass can exist without a Sub Class.

(Simple Example)

We will go by the life cycle of each element.

We check how is the life cycle of SuperClass dependent on sub class.
And how is the life cycle of Sub class dependent on super class.

Person → heart
Person → bag

Person dies, heart also dies.
Person dies, bag can remain.
Bag dies, Person alive heart dies, Person dies.

(Product → Rating)

(Swiggy/Zomato) Example

Restaurant

↓

Menu

↓

Food Items

"has-a" relationship has 2 divisions.

"has-a" (Sometimes has-a itself is called composition)

↙ Aggregation

↘ Composition "part-of"

(Composition) (Part-of).

(Person - heart) ⇒ If person dies heart also dies.

Entire lifecycle of heart is related on Person.

Think of it like

"Heart is a part of the person." Without which it cannot exist.

("has-a") Aggregation. (Person - Bag) relationship.

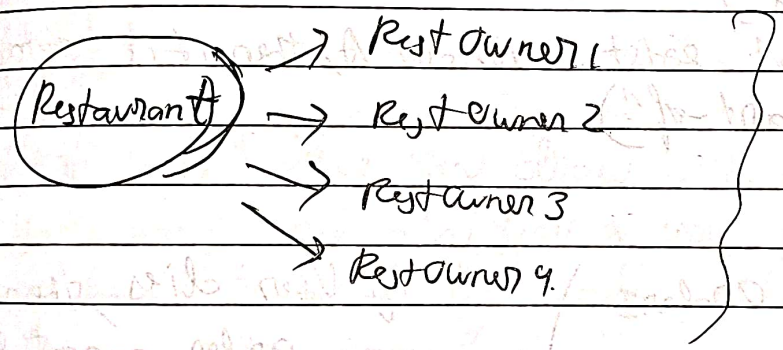
Bag can exist without Person as well.

(Another Example)

(Restaurant & Restaurant Owner)

a) What should be the relationship between them? Is it aggregation or composition.

b) We can say a Restaurant can have 5 Restaurant owners.



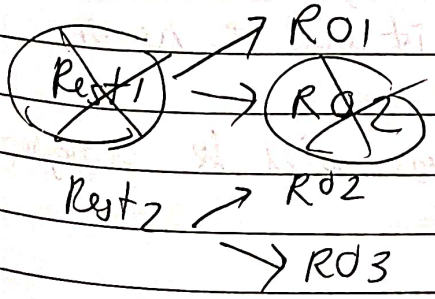

```
Rest {
    List<RestOwner>
}
```

Lets say we have

Rest1, Rest2

[RO1, RO2, RO3, RO4] → (Rest owner)

Object level



Now if Rest1 is closed. Does not mean RO2 should also go away.

Because we have to manage Rest2

Look for Aggregation ("has-a")
Composition ("part-of")

Look at Object level for determining the relationship.

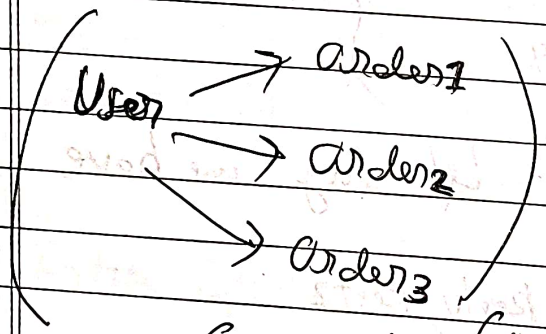
In simple terms. "Keerthi says

$A \xrightarrow{has} B.$

This can be either aggregation / composition.

Can B exist without A or not.
If B can exist without A, then it is aggregation. ("has-a")
If B cannot exist without A, then it is composition. ("part-of").

"



If User dies, then order cannot exist.

Whose order will it be?

Composition ("part-of")

" If there is not one-one, but one-many, or (many-many) then it will be never Composition and always will be aggregation."

(Example) : In terms of C++.

```
class Person{

    Heart heart;

    Person(){
        heart =  { Constructor is creating parameter }
    }

    ~Person(){
        delete   { Destructor is deleting it }
    }
}
```

(UML)

Veerti says " Once you draw UML diagram. It is very interesting and gives you a very good intuition on how to write the code and the coding becomes very easy."

(Inheritance)

A ——▷ B (A "is a" B)

Fruit ◁—— Avocado

Base ◁—— Derived

{Arrow is towards the parent}

Base
△
Derived

A ◇— B A "has a" B Aggregation.
B can exist without A.

A ◆— B A "has" B Composition.
B can't exist without A
(part of)

(***) Note: Arrow is towards the bigger class.
~~Inheritance~~

Inheritance A —▷ B (B inherits A, so B is the bigger class)

Aggregation A ◇— B A is the bigger class inside which B is the attribute. A is parent hypothetically

Composition A ◆— B A is the bigger class inside which B is a parameter. A is containing B. A is parenting.

(Arrow is towards the ~~bigger~~ Parent class Remember".)

PAGE EDGE

Date :.....

(Association) 1st type.

A —— B (A relates to B)

(A can call B, B can call A)

(Bidirectional)

(Association) 2nd type

A → B (Only A can call B
B cannot access A).
(Unidirectional)

(Example)

(User → Payment Gateway)

User can access Payment Gateway
But payment Gateway cannot access User.

*** Note : Driver & Ride.

If we keep

class Driver{

List<Ride> rides.

} ⇒ then it becomes rides.

then it becomes (has-a).

Keerti asks: "Can someone give example of bidirectional association".

To which one guy said: "

User — Rest ".

I was also assuming that its the right example. As Rest will need to contact User, in case order is late or some issue.

(Reason) But Keerti put a very valid argument.

Do you want to keep User class in Restaurant.

User is the basic class which has all sorts of important information which can be sensitive.

Its better to have User Contact Details class inside the Restaurant Class.. ⇓ (UserMetaData)

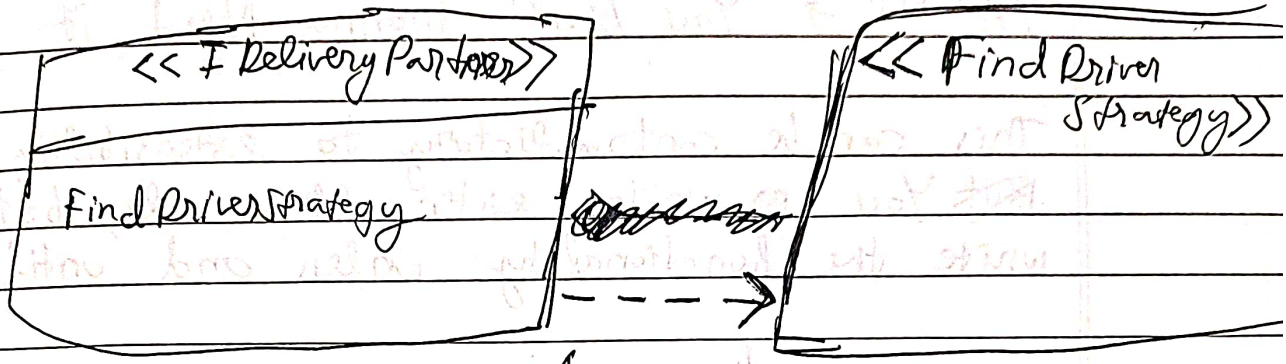
Implements.

A ----▷ B (implements) { A is implementing B interface }

A ——▷ B (extends) (A is extending B.)

B is parent, A is sub Class.

In UML diagram, anything w.r.t to interface we will have ---- line. Even if it's association.



As

only IDelivery Partner can call FindDriverStrategy

Tip from Keerti " In the interviews, its very tough to and time consuming to draw arrows and all,

So what she used to do. She would draw a line and just write on top of it, & convey to the interviewer whether its aggregation, composition & implements or etc."

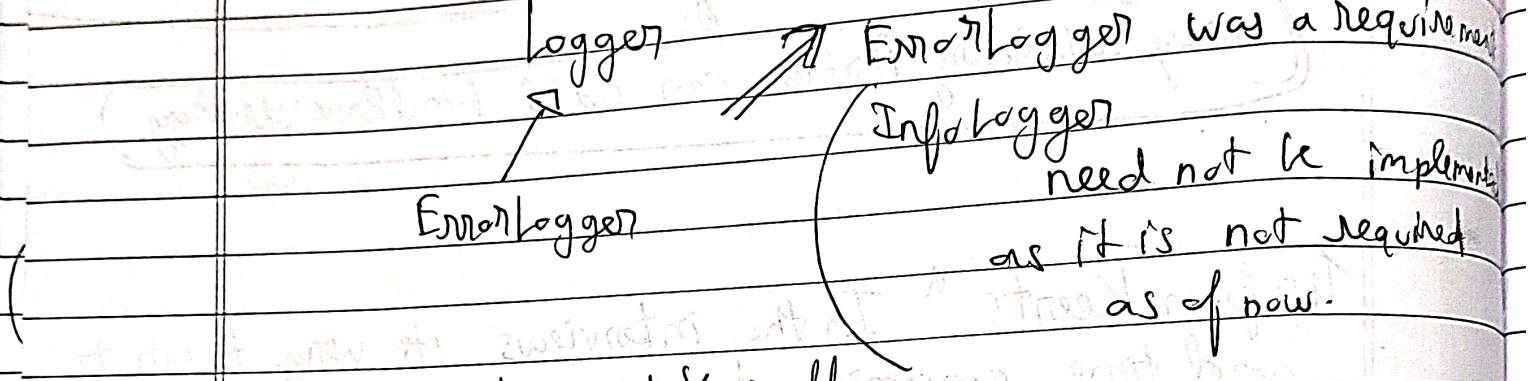
Helpful Guidance Coming up front →

Keerti said she will be discussing how to convert Requirements → Sequence Diagram and then to UML Diagram and all those things.

KISS → Keep it Simple Stupid

YAGNI → You Aren't Gonna Need it.

This can be contradictory to extensibility.
But You write extensible code, but don't write the functionality unless and until required.



DRY → Don't Repeat Yourself.

Design Patterns

Some common problems keep occurring while designing low level in software engineering.

So we don't reinvent the wheel. We use already inplace design pattern templates, created for this purpose.

1) Creational Design Pattern →
2) Structural Design Pattern →
3) Behavarial Design Pattern. →