Low Level Design:

Keerti Purswani Course

Lecture 1

** Contains Notes of what we study

As a developer, we want to write code such that any new joinee or new employee can pick it up and read it As if I left the company and he could understand the code. It is so well written.

My code should be

* Maintainable
* Extensible → Adding a new feature should not require massive changes
* Modular → Structed to maximize Cohesion and minimize Coupling

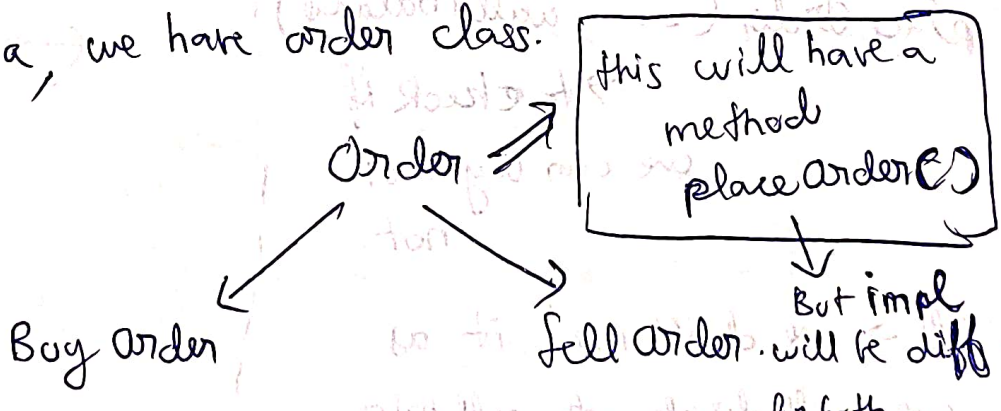
Encapsulation → Helps to encapsulate data members and data methods which helps in clear separation of code.

* Separation of Code
* Ensure Data Integrity
* Access Control → Private
  → Protected ⇒ (within same package, derived class, subclass)
  → Public.

Abstraction → Hiding the implementation, Just Displaying and presenting the possibilities we can provide.

Inheritance → Extending the functionalities of parent class and adding your own ontop of it. Making the code extendable

(Example) For Zerodha, we have order class.



Polymorphism → ~~Method Overloading~~

many forms.

→ we use this when we want multiple implementations of the same method.

2 types.

- → Compile time ⇒ Function overloading
- → Runtime ⇒ Function Overriding

Function Overloading

→ Same function Name, Different Signature

Example

Zerodha →

Place order.

to Buy → We will need to

place Order (—, walletBalance)

→ to check if we can buy it or not

to Sell → we don't need it as we will display the sell price only

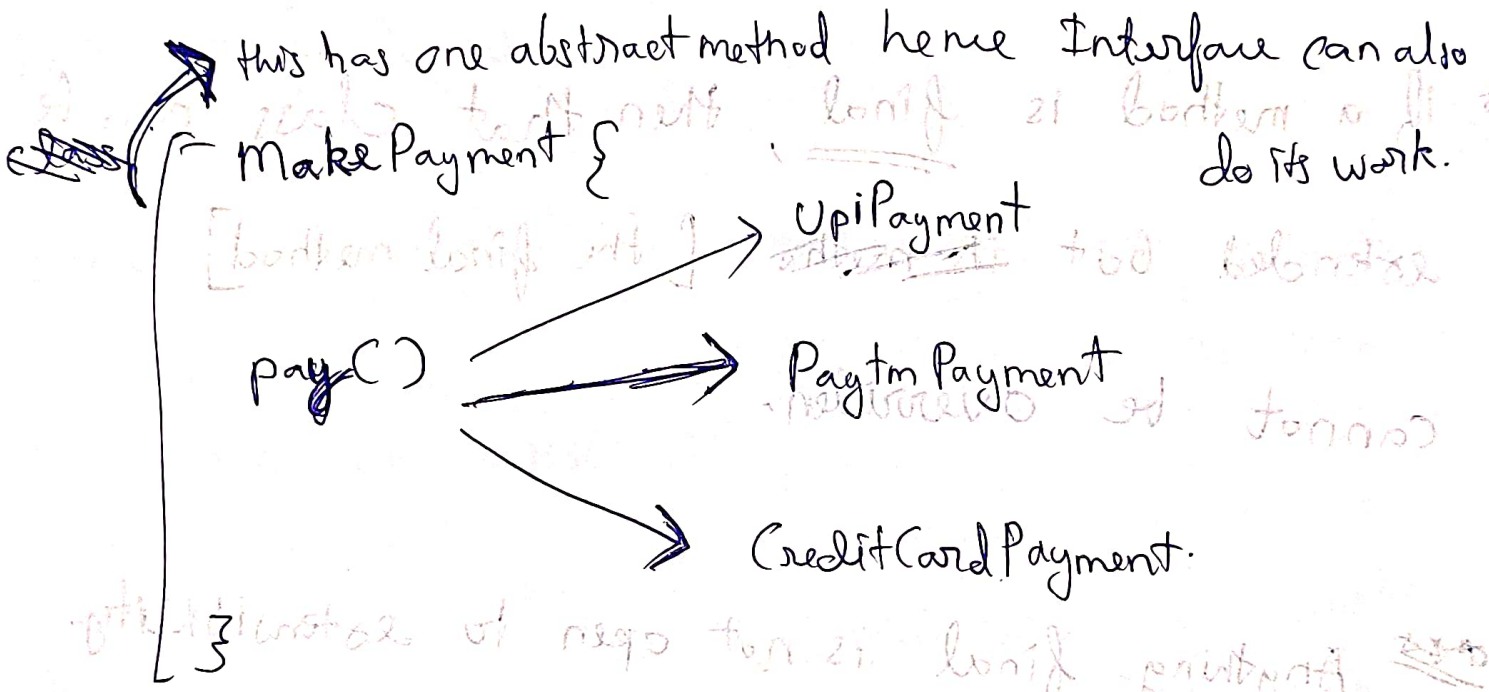
Function Overriding

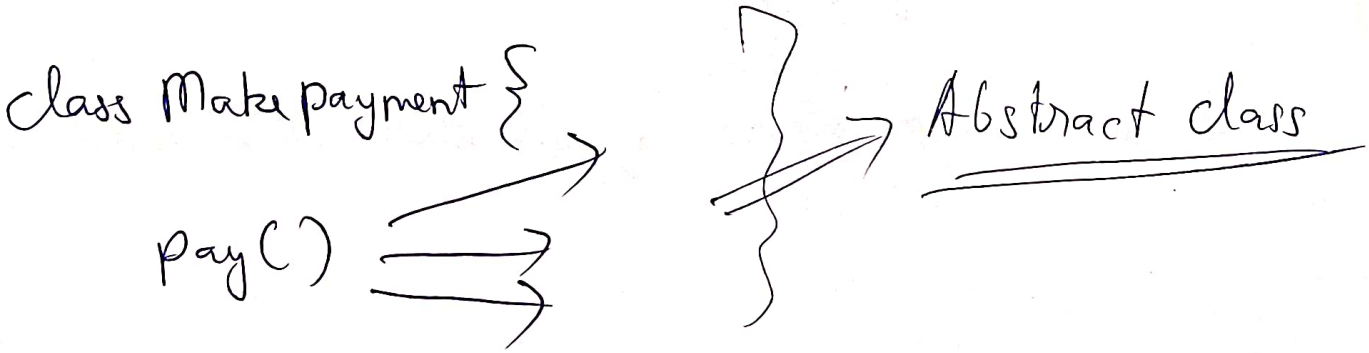
→ Same function Name, Different Impl

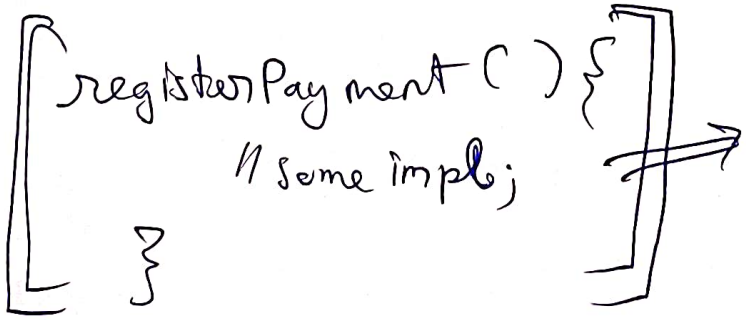
Abstract Class V/s Interface

Both have sub classes.

Lets take example of Payment in Swiggy, Zomato.







Imp Note

* If its a final class, then the class cannot be extended by any other sub class.

* if a method is final, then that class can be extended but [the final method] cannot be overriden.

* Anything final is not open to extensibility

SOLID

Best practices to code as a software engineers
Guidelines for designing.

S → Single Responsibility Principle

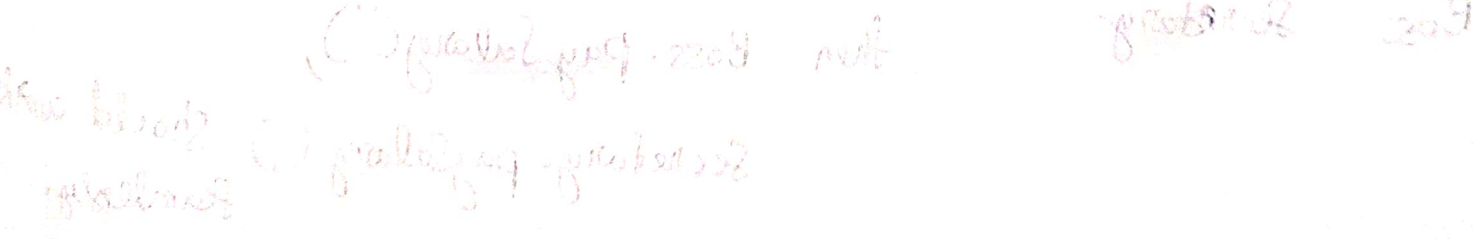
Example for Zomato.

```
class OrderValidation {

    // securityChecks()
    // Funds()
    // Permissions()

}
```

For security purpose, code has to change for it

Permission team will make changes in the same class.

3 teams will make changes on the same code base.

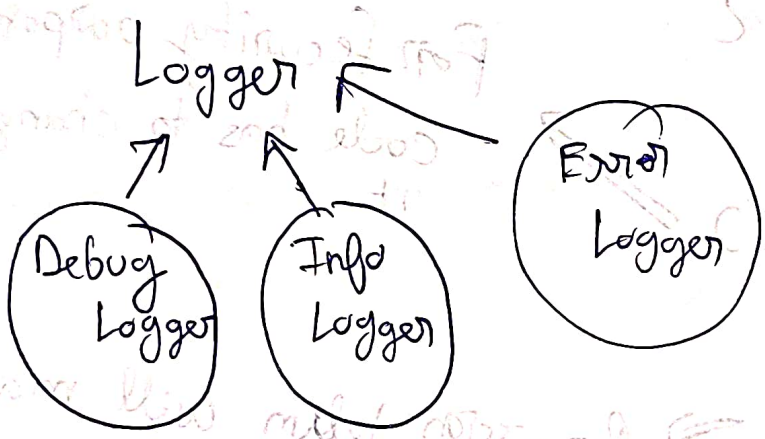


Question comes → where do we stop dividing it.
for single Responsibility

Thats where Requirements of the question come into play

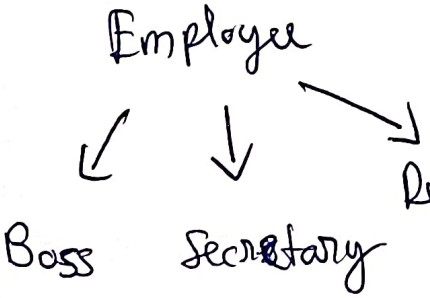
How extensible you want to make your code

O → Open for Extension, Closed for Modification



L → (Liskov's Substitution Principle)

We studied about inheritance right?



if anywhere in the code
employee.paySalary() is used
then Boss.paySalary(),
Secretary.paySalary() should work seamlessly

```
class Bird{
    public void fly(){
        Sout("I fly");
    }
}
```

```
class Sparrow extends Bird{
    @Override
    public void fly(){
    }
}
```

```
class Penguin extends Bird{
    @Override
    public void fly(){
        throw RuntimeException();
    }
}
```

This breaks Liskov's Substitution Principle

Behaviour broke,

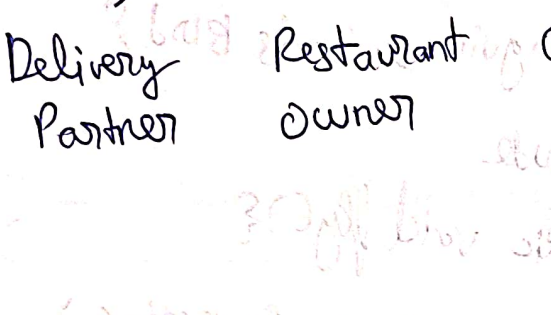
Runtime exception is thrown if

I do; penguin.fly() instead of bird.fly()

Base ⇒ (Replaced by) Derived (Subclass)

Should not change behaviour.

She took an example of Swiggy / Zomato



Lets say User has a method called `doKyc()`.

For Delivery Partner & Restaurant Owner, this example works.

But why Customer needs to do KYC().

So in Customer Class the method's behaviour is broken.

Which disObey's Liskov's Substitution Principle

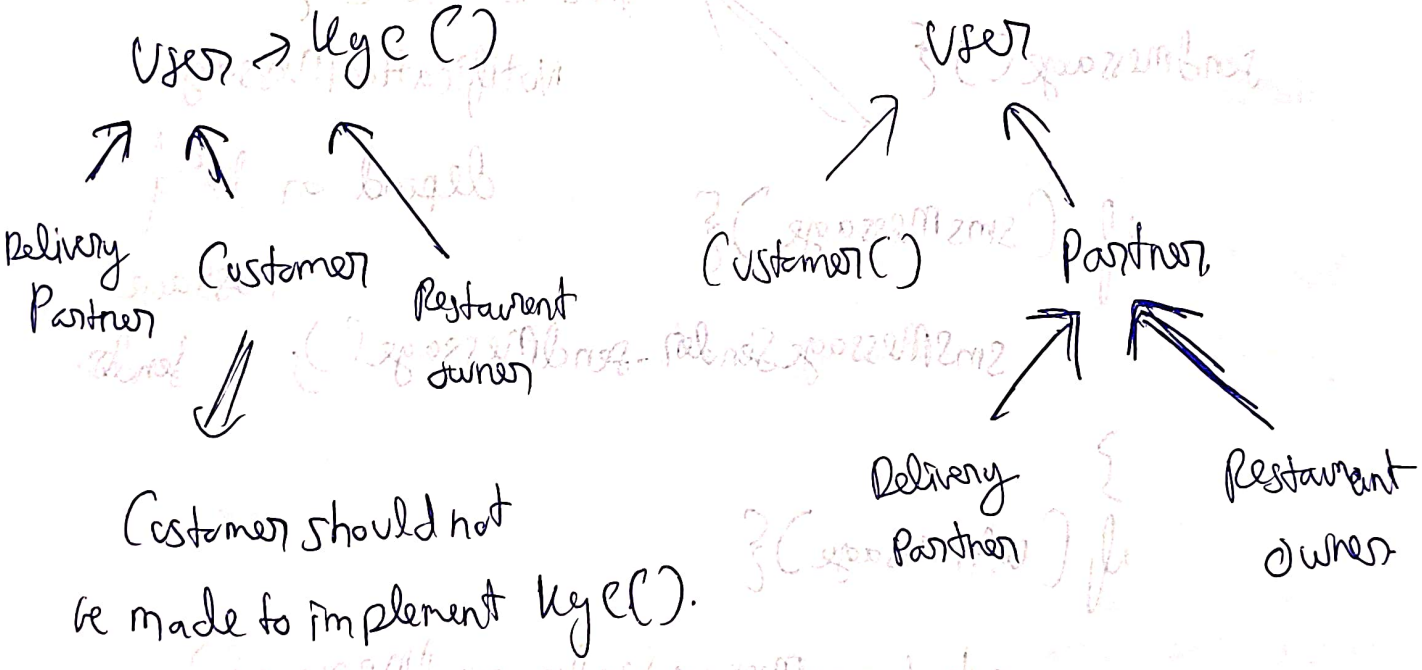
* Keerti says

"Lot of SOLID principles are interlinked with each other, If you ~~be~~ follow two or more of SOLID, You end up following all of the Principles!!

I → Interface Segregation Principle

This principles tells us to keep interfaces concrete and only have methods which are required and not methods which are not required.

Segregate Interfaces if its becoming too long



Keerti says

" In the next Lecture, we will see Composition is said to be better than Inheritance "

class B { → nesting class A {

A a;

}

}

D → Dependency Inversion Principle.

** Main aim of this principle is to have loose Coupling

Example: Lets say we have a NotificationMessage

```
Class. NotificationMessage{

    sendMessage(){

        if (smsMessage){
            smsMessageSender.sendMessage();
        }
        if (waMessage){
            whatsappMessageSender.sendMessage();
        }
        if(telegram){
            ...
        }
    }
```

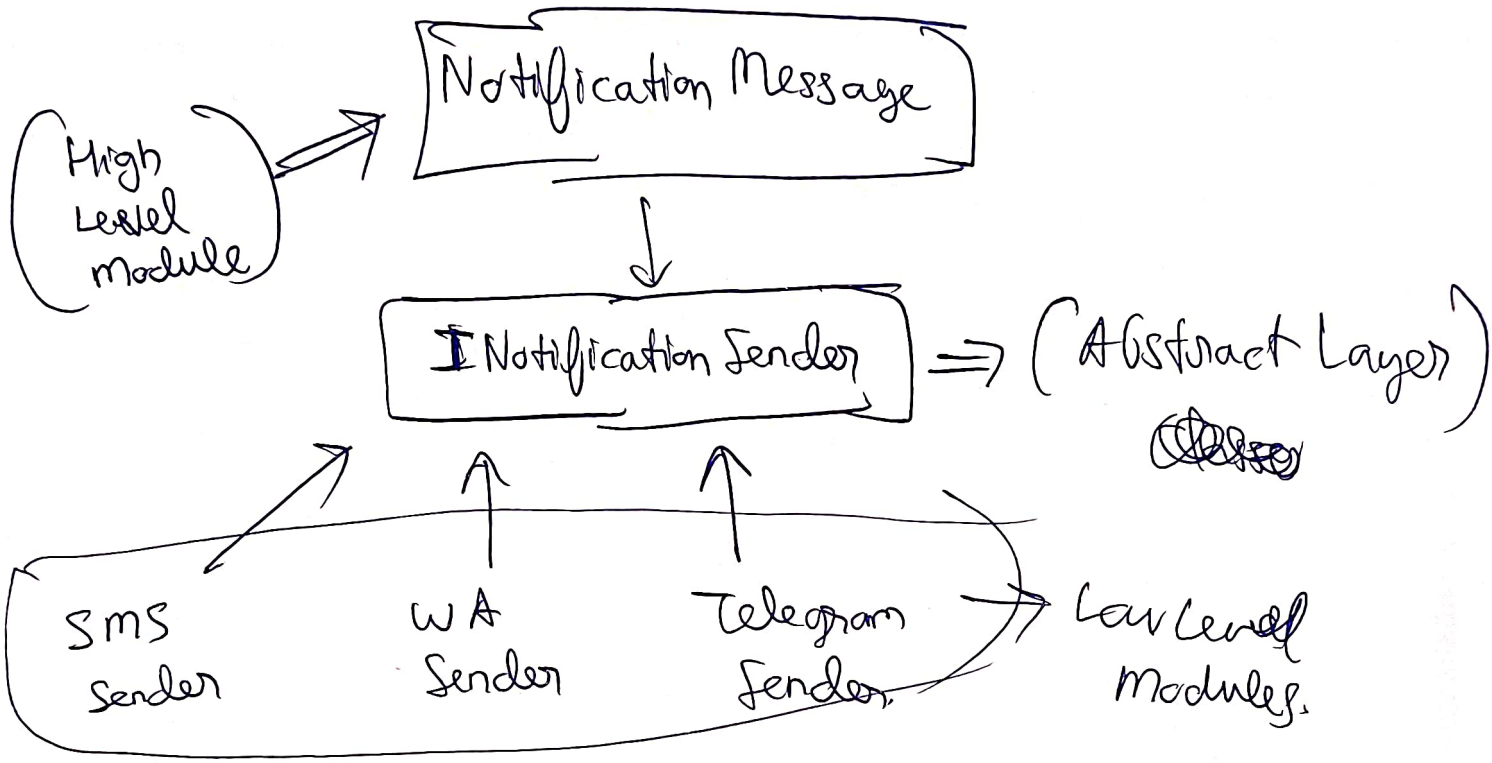
this is making NotificationMessage depend on lot of MessageSender.

** tomorrow if a new messageSender something new comes we will have to add it to the class

(Breaking openclosed principle)

(To Simplify this?)

We identify that NotificationMessage is a high-level module, inside which all the message Senders are present



Funda

High Level Module, should not depend on low level module. Instead there should be an abstraction layer in between.

Its Loosening the Coupling - Any Concrete Sender being deleted won't cause much of an issue