AGENDA

1.> strategy design Pattern

2.> Observer design Pattern

[Behavioural design Patterns]

start by 9:07 PM IST

*] BEHAVIOURAL DP:

These are mostly concerned with assignment of responsibility b/w objects.

i.e. How to handle special / edge cases for behaviour.

1.) STRATEGY DESIGN PATTERN:

Eg: Google Maps

search for (A) ⟶ (B)

different routes / modes of transport shown:

train ⟶ route (1)

bus ⟶ route (2)

SOMETIMES — for a Road : cars are Not allowed.

Bus Not allowed / one way etc....

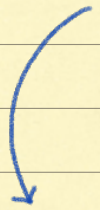
searching for routes:

(A) → (B)

we get multiple ways to travel

Each way has:

① Different route / ETA



if we write code-

```
class GoogleMaps {

  = findPath(source, destination, mode) {
      if (mode == car)
          getPathforCar()        // shortest
      else if (mode == bus)
          getPathforBus          // Bus
          ⋮
      else if (mode == metro)
  }

}
```

PROBLEM.... ?

① violates SRP / OCP

② supporting new mode is difficult

③ Making change requires full QA

Problem statement:

When multiple ways to do something, we often observe violation of SRP/OCP.

↳ End up using if/else...

↓

∴ Solution: use STRATEGY D.P.

*] OBSERVATION:

```
findPath(source, destination, mode){
    if (mode == car)
        getPathforCar()
    else if (mode == bus)
        getPathforBus
        ⋮
}
```

⟶ code for specific use cases → completely different / Independent.

Hence; Instead of having all behaviours in same method Implement them in separate class.

*] STEPS:

1:) for Every way to do Particular thing → create class

a:) carPathcalculator ✓
b:) BikePathcalculator ✓
c:) walkPathcalculator ✓

✗

↓

USAGE:

class GoogleMaps {
carPathcalculator c;
BikePathcalculator b;
walkPathcalculator w;

.....

→

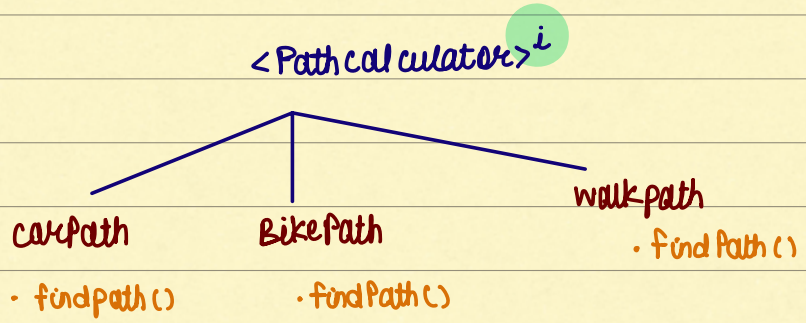
}

violates → D.I

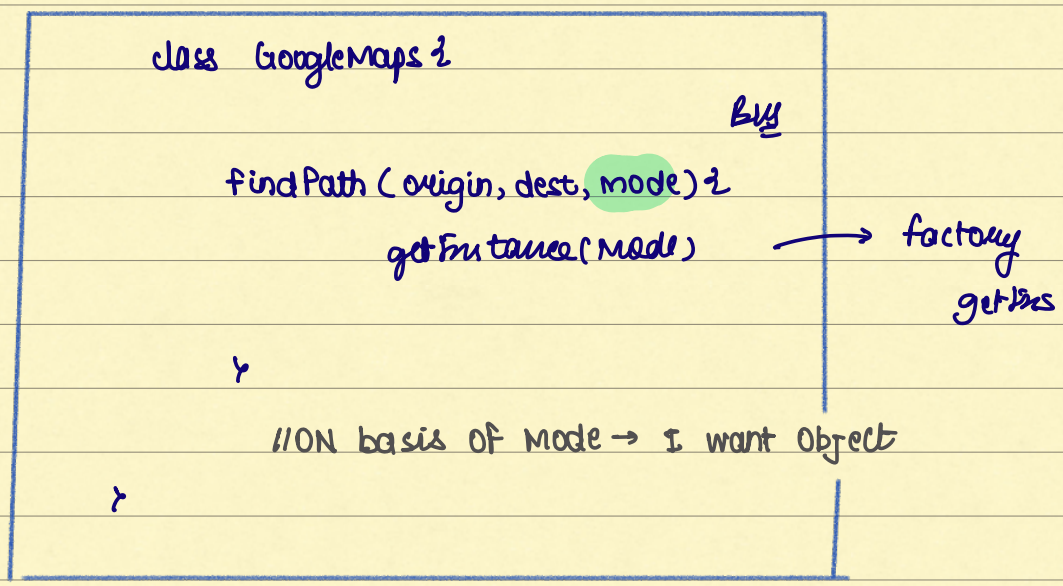
create Interface: ✓

Always code to Interface; rather than Implementation

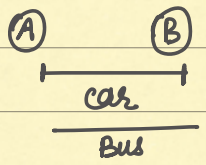
2.> create interface



3.> use the interface in Maps class



design Pattern :



Now, In GoogleMaps class :

```
class GoogleMaps {

                                    <modes>
    findPath (orgn, dest, mode) {
        getPathCalc(mode) - findPath()
    }
}
```

```
class PathCalculatorFactory {

    getPathCalculator(mode) {
        if mode == car
            return CarPathCalculator()
        ⋮
    }
}
```

Summarize: Multiple ways to do something

↳ use structural D.P.

strategy dp

(2) (3)

ASSIGNMENT/HW:

Rain Water Trapp. Problem

→ write code in All ways.

*] OBSERVER DESIGN PATTERN:

definition - useful when we are Interested in state of object & want to get Notified whenever there's any change.

eg:

eg: Amazon

Whenever order is placed → what happens:

Inventory check
user: Notification / Invoice

```
orderPlaced() {
    updateInventory()
    NotifyWareHouse()
    updateDelivery()
    triggerNotifications()
    .
    .
    .
}
```

facade

50 helper classes

PROBLEM

Observer dp

violates OCP / SRP

New Requirement: update details in Analytics system

would need changes

① In your existing code

② In Runtime, we validate many things-

eg: for Prime music/video

→ No warehouse update is Needed.

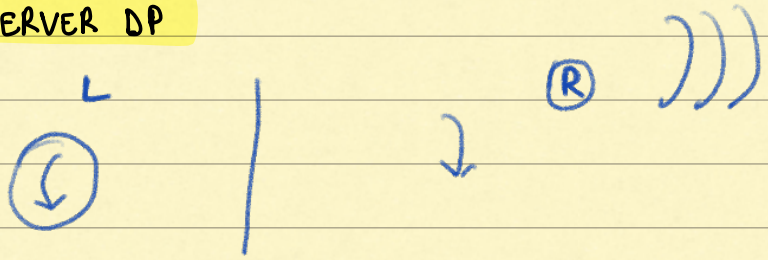
Using Above solution → becomes tricky.

Problem statement

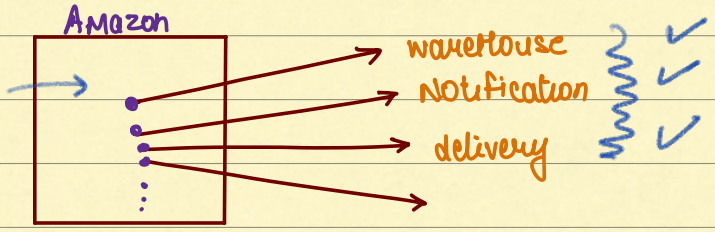
① When some event happens → we need to do lot of things everytime. [This becomes less scalable]

② we cannot modify things at Runtime during Any Event

SOLUTION: OBSERVER DP

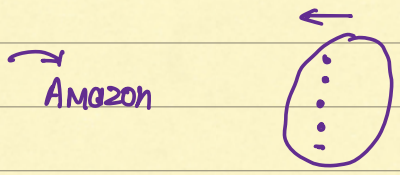


As of Now:



(1) class is responsible to trigger many flows

*] STEPS:



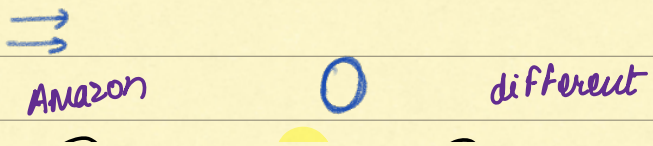
1.> In such cases, think in opposite way

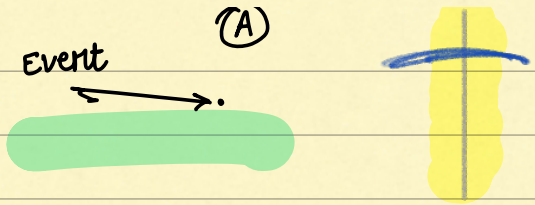
Multiple classes want to process something when Any Event happens.

basically: Many classes wait ON me

↳ HOW CLASSES ARE NOTIFIED ABOUT EVENT:

As of Now: No link / communication b/w ② independent classes





How would (B) know about Event

(P) (S)

*] TERMINOLOGIES:

(1) Publisher: classes that Generate the event (A)

(2) Subscriber: classes who would want to know Event has happened (CONSUME) (B/C/D....)

1 publisher for 1 Event; but Many subscribers

2.> create Method in Publisher

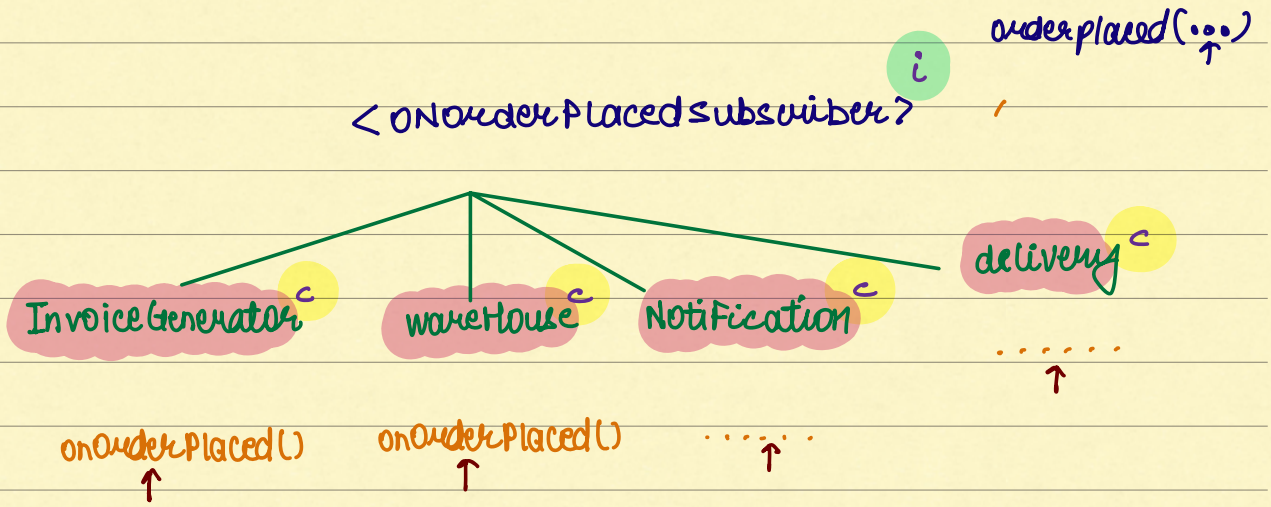
This Method allows subscribers to register themselves

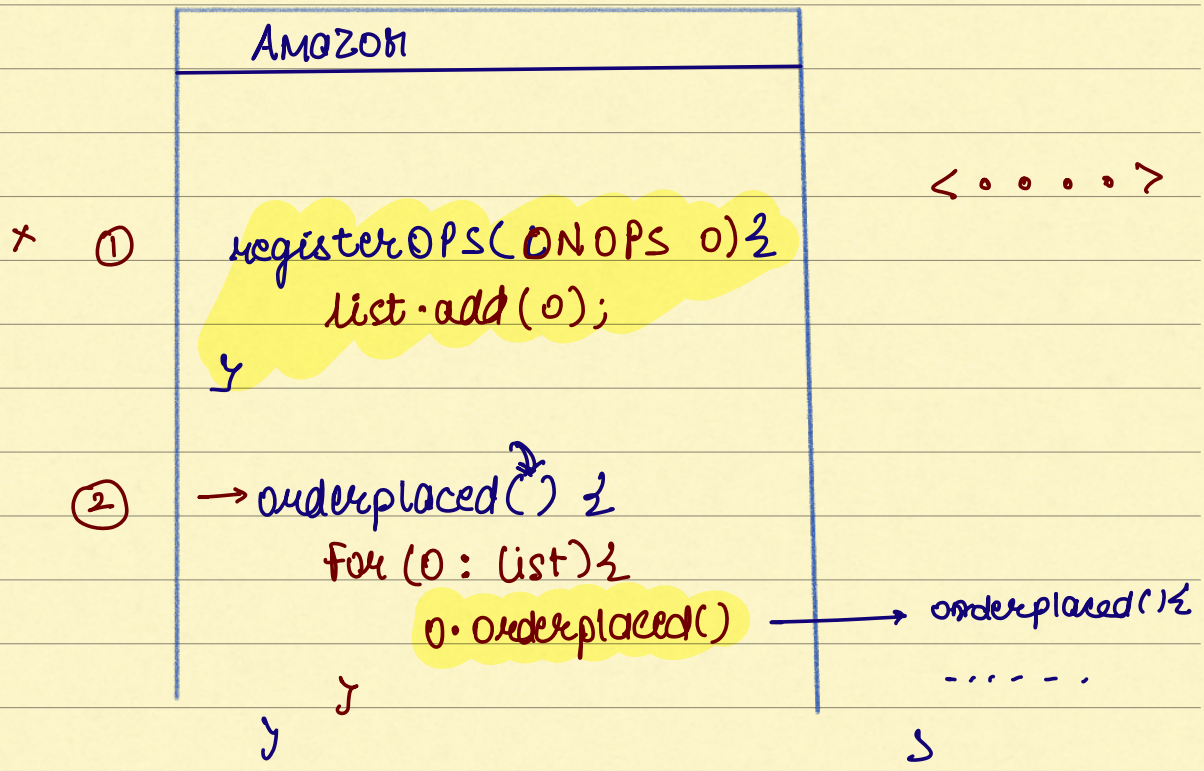
```
class Amazon {
    List<Ops> list;

    registerOrderPlaceSubscribers(i)
        ops.add(_);
}

orderplaced(){
    for(...){
    }
}
```

What should be passed ??

}

→ create interface & Implement in All subscribers.





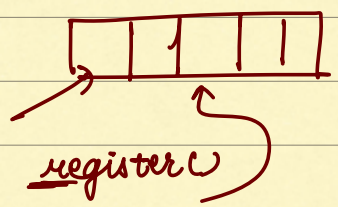
registerPlaceOrderSubscriber (orderPlacedSubscriber)

*) Advantages :

1.> Adding New subscriber is Easy

Similarly; → Method should be added for

→ unregisterOrderPlacedSubs ()



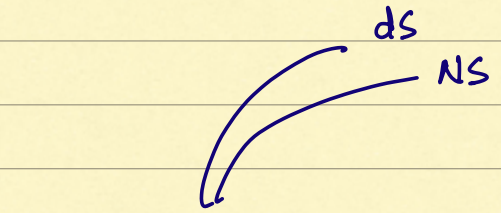
3.> Registering subscriber

Q.) when subscriber should register themselves ?

↓

Mostly, when Object is created.

eg:



InvoiceGenerator Implements OPS {

IG() {

amazon.registerOPS(this);

}

IG(.....) {

amazon.register(this);

}

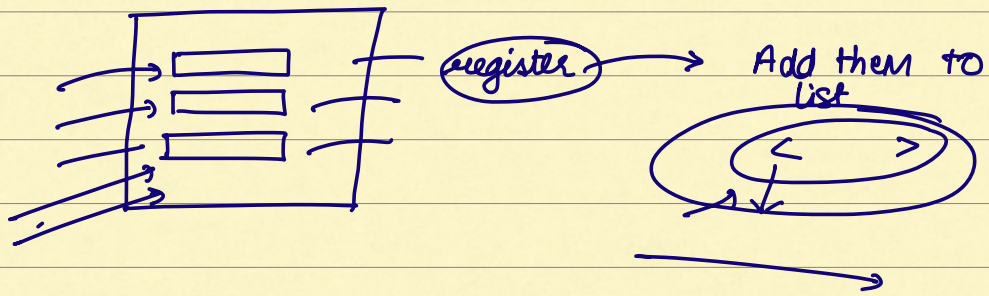
orderplaced(...) {

}

}

EXTRA:

Can we Add / Remove subscriber on Runtime ?



(P) (P) (S)

Amazon 2

// ① access all subscribers —
② Publish

list<OrderplacedSubscriber> list;

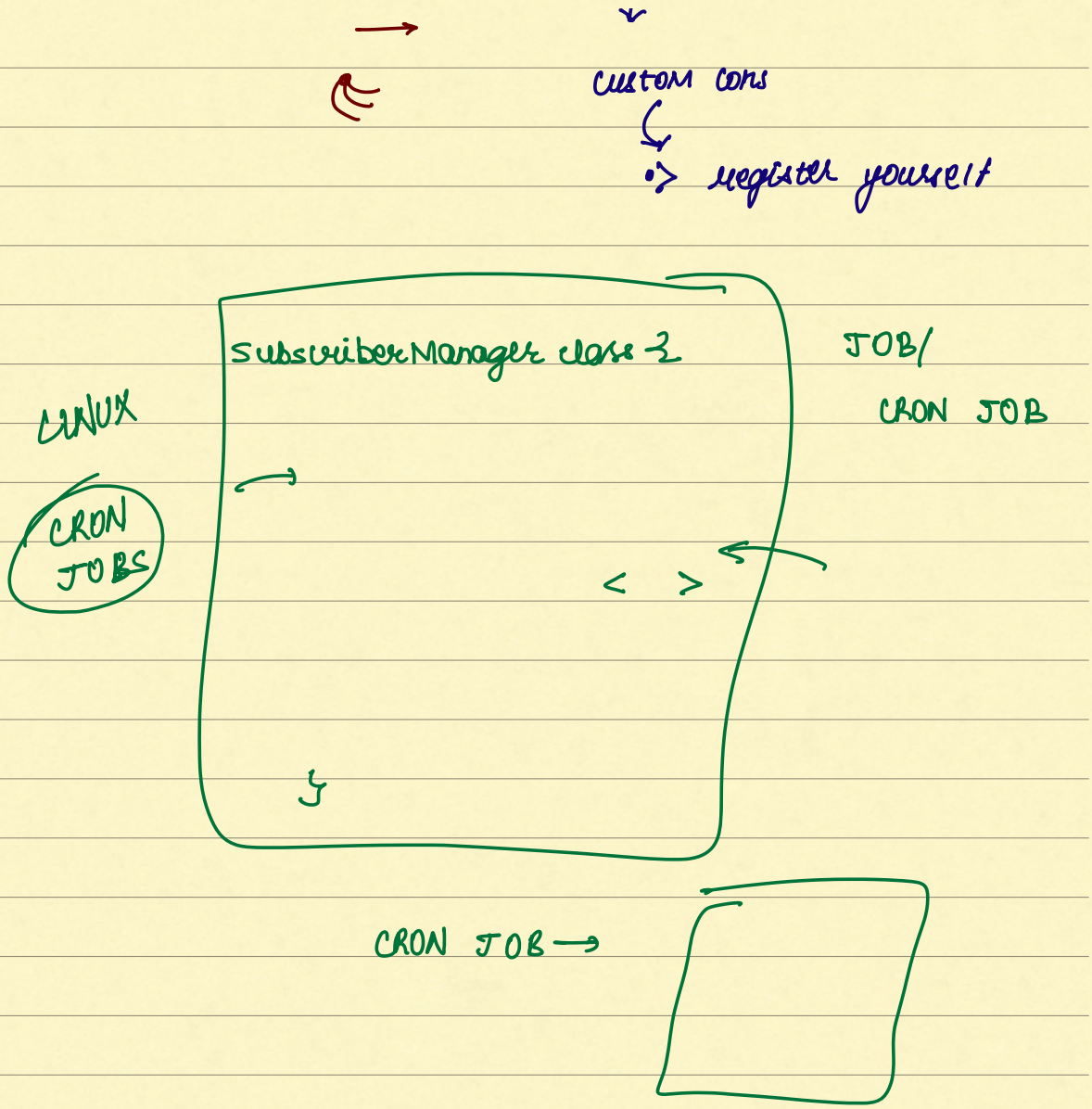
registerOrderPS(ON orderp subs. s) {
list.add(ops);
}

orderplaced() {

}

Invoice Gent .. Impl OPS

client xx
→ IG ig = new IGC(); // xx


custom cons
=> register yourself
SubscriberManager class {
JOB/
CRON JOB
LINUX
CRON JOBS
< >
}
CRON JOB →