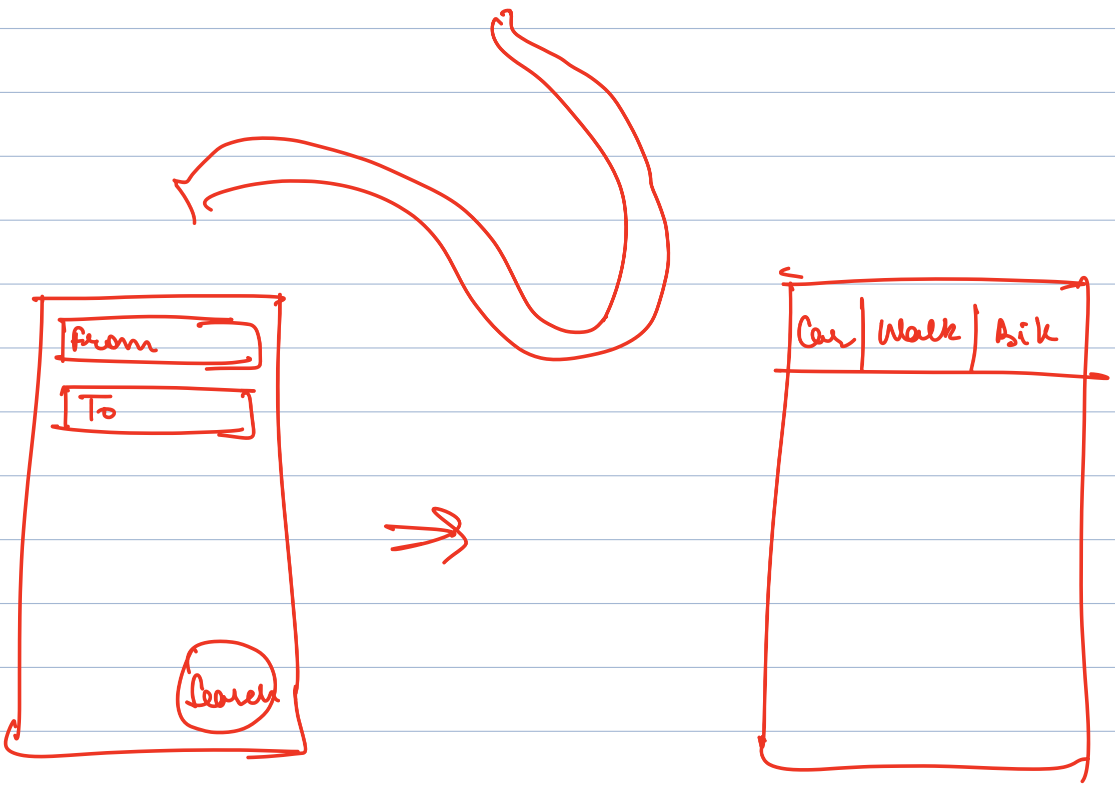
Behavioural Design Patterns

1. Strategy →
2. Observer
3. Command ⇒ Design BMS

Behavioural Design Patterns → Solve problems wrt implementing a specific complexity with a behaviour in the system.

→ how to implement a specific type of f^n

STRATEGY DESIGN PATTERN



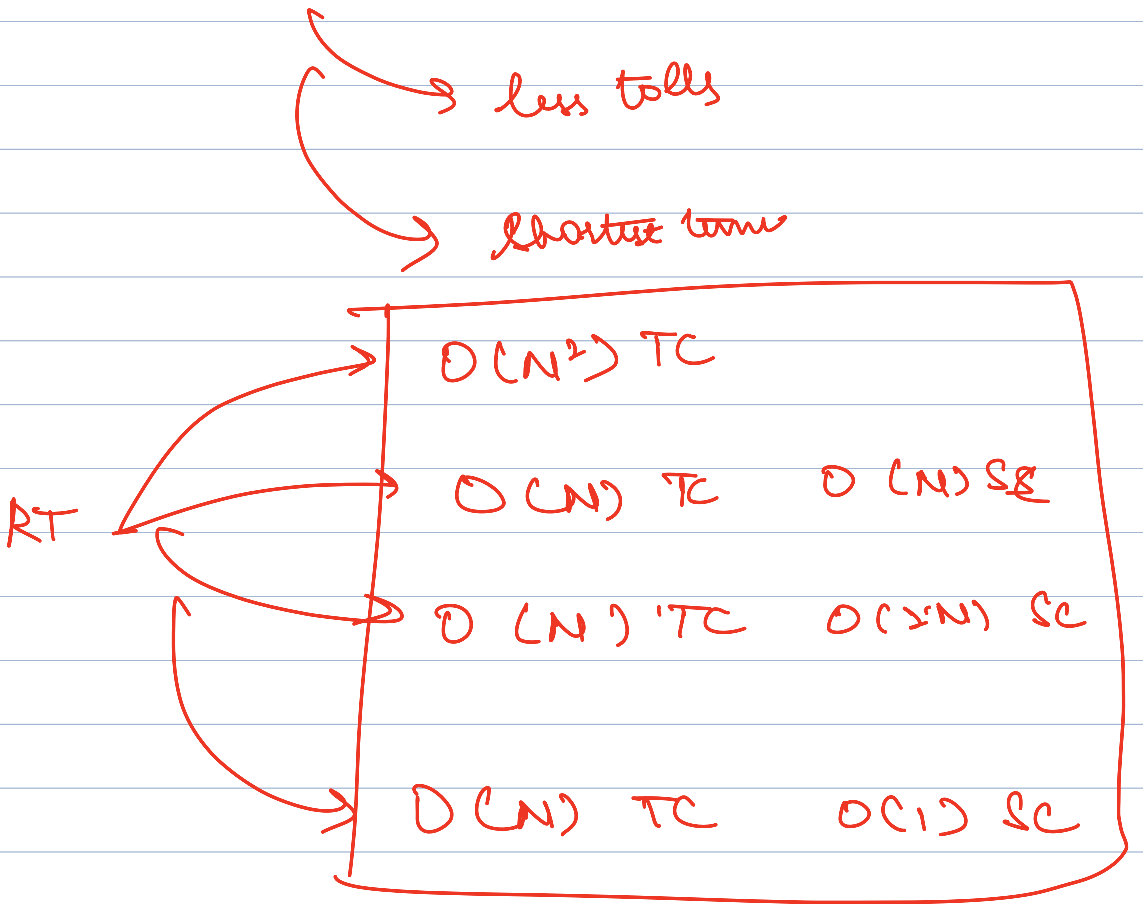
→ Avoid toll roulist
→ Show shortest path
  by dis
→ Shortest path by time

```
Googe Maps {

⇒   findPath (from, to, mode) {    Setting
        if (mode == car) {
            ===
        } elif (mode == public) {
            ===
        } else {
            ===
        } else if (mode == -
        }
    }
}
```

Violates SRP, OCP

⇒ Often there are Scenarios when there are multiple ways to execute a particular behaviour

$$\begin{tikzcd} & \text{train} \\ \text{findPath} \arrow[ur] \arrow[r] \arrow[dr] \arrow[ddr] & \text{car} \\ & \text{public} \\ & \text{walk} \end{tikzcd}$$

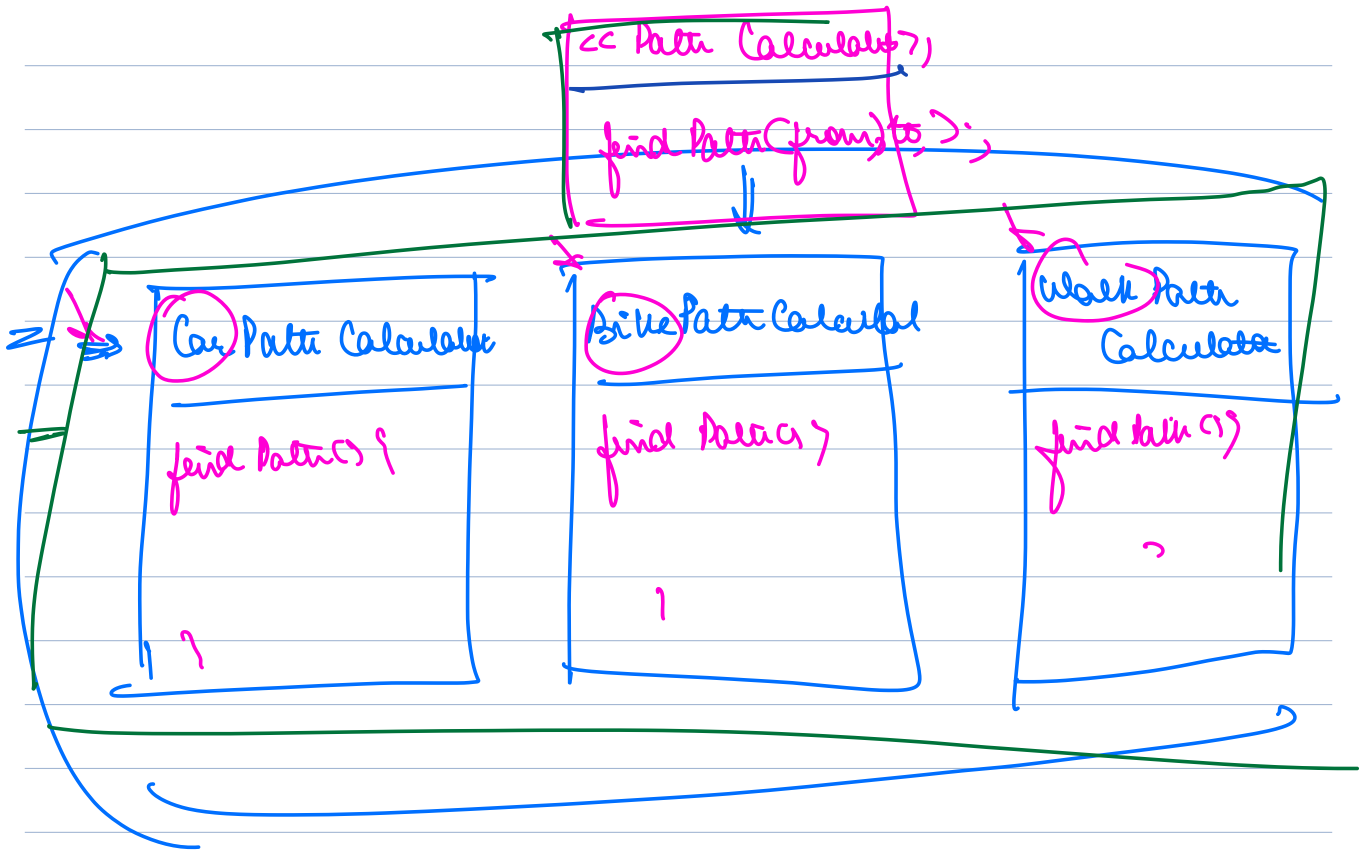


STRATEGY

→ Whenever there are multiple ways to do a particular behaviour.

① Rather than having code of each way of doing a behaviour within the same class, put the code in separate classes.
⇒ Create 1 class for each way of doing that particular thing.

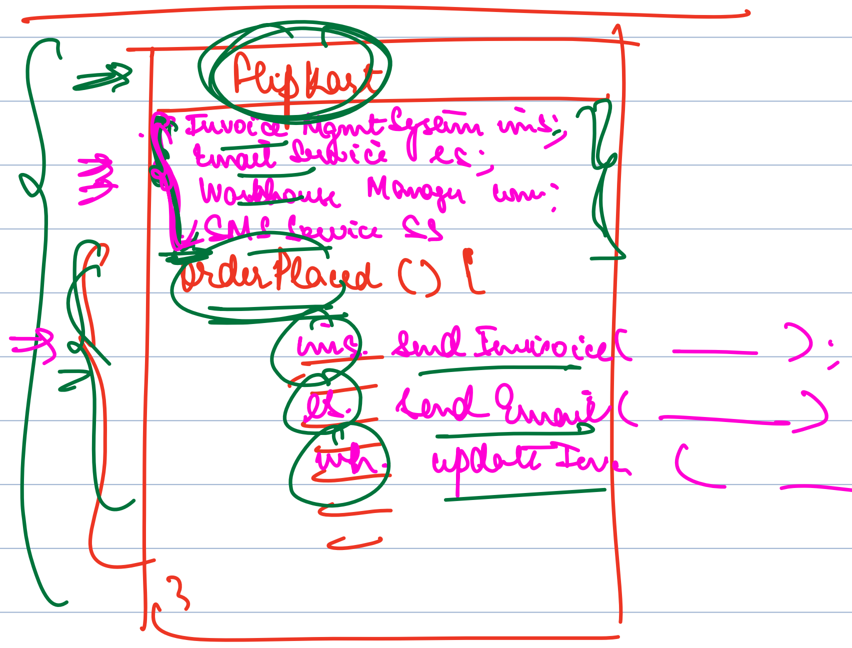
② Create an interface that has all common methods needed from each way.



```
Google Maps {

    findPath ( from , to, mode), {
        PathCalc pc = PCF. get PC for Mode (m);
        return pc. findPath (from, to);
    }
}
```

```
class PathCalculatorFactory {        // Practical Factory
                                         Design Pattern

static PathCalculator   findPathCalculatorForMode(Mode m){
        if (mode == CAR) {
            return new CarPathCalculator();
        }

    }
}
```

OBSERVER DESIGN PATTERN



```
Flipkart
    InvoiceMgmtSystem ims;
    EmailService es;
    Warehouse Manager wm;
    SMS Service SS
    orderPlaced () {
        ims. sendInvoice(  ____  );
        es. sendEmail(  ____  )
        wh. updateInven  (  ____  )
    }
```

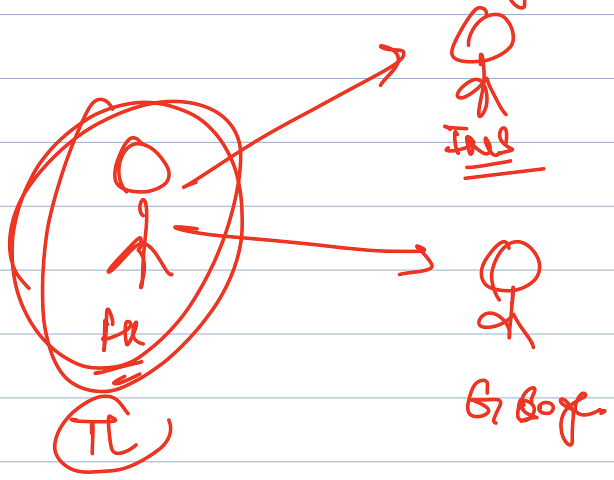
this method may involve/need to do a lot lot more things in future.

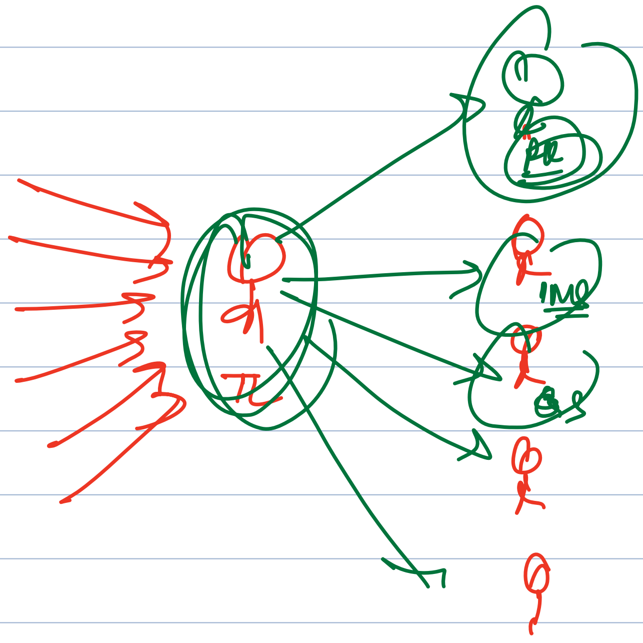
→ Whenever you want to do an additional thing on order placed, you will have to open FK class, update its CB.

$\boxed{\text{X Violates OCP}}$ ⇒

① I want to do something when an order is placed

② I dont want to update my CB to call that dependency to do that thing





Push V/S Pull Mechanism

① TL goes to eng to tell them the work ⇒ Violates OCP

② TL just sits in the office. Eng who want work come to office and ask for work.

Observer Design Pattern

→ Think of order placed as a special event

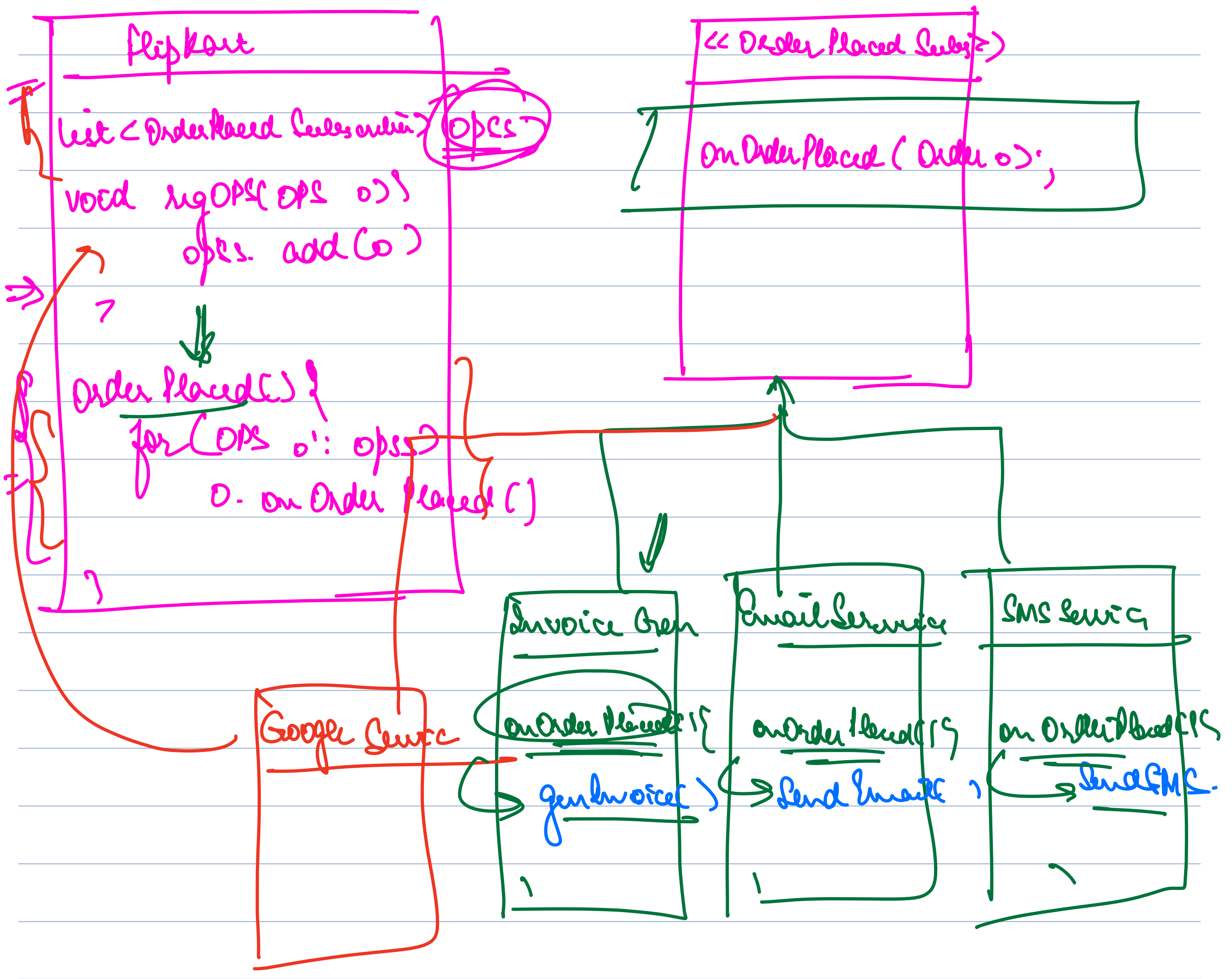
↳ a lot of services / dependencies are waiting for the event to happend to do what they want to do.

① Rather than us calling the dependencies' methods directly when the event happens, we should allow a way for dependencies to register themselves with us.

Object

```
Class Flipkart {
    List<OrderPlacedSubscriber> orderPlacedSubscribers;

    → void registerOrderPlacedSubscriber(OPS ) {
          ops.add(ops)
    }

    void orderPlaced() {

        for ( subscriber: orderPlacedSubscribers)
              subscriber.notify ()

        }
    }
}
```

② Create a common "Subscriber" interface with a method, which is implemented by all classes who want to do something when an order is placed.



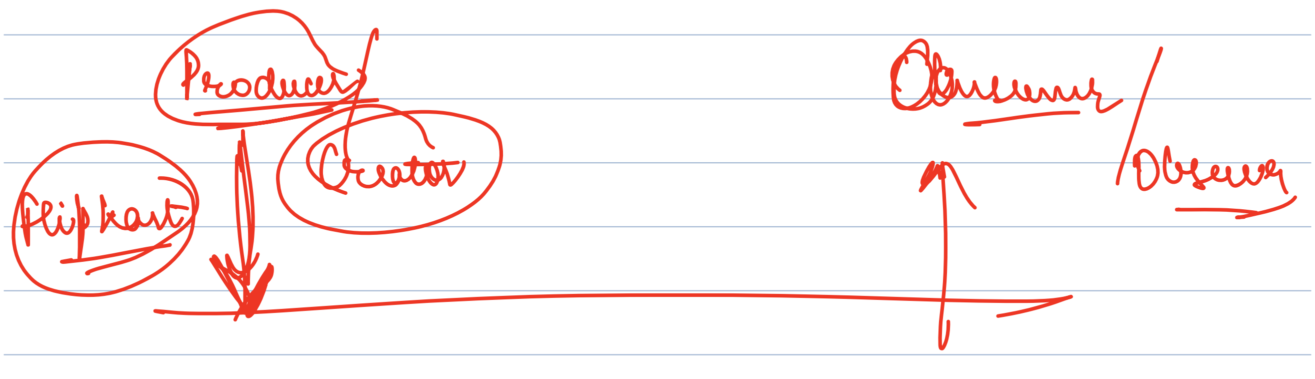
```
List<OrderPlacedSubscriber> OPSs
void regOPS(OPS o){
    opss.add(o)
}

OrderPlaced(){
    for(OPS o : opss)
        o.onOrderPlaced()
}
```

```
onOrderPlaced(Order o);
```

```
onOrderPlaced(){
    genInvoice()
}
```

```
onOrderPlaced(){
    sendEmail()
}
```

```
onOrderPlaced(){
    sendSMS.
}
```


Producer
Creator
Flipkart
Consumer
/Observer