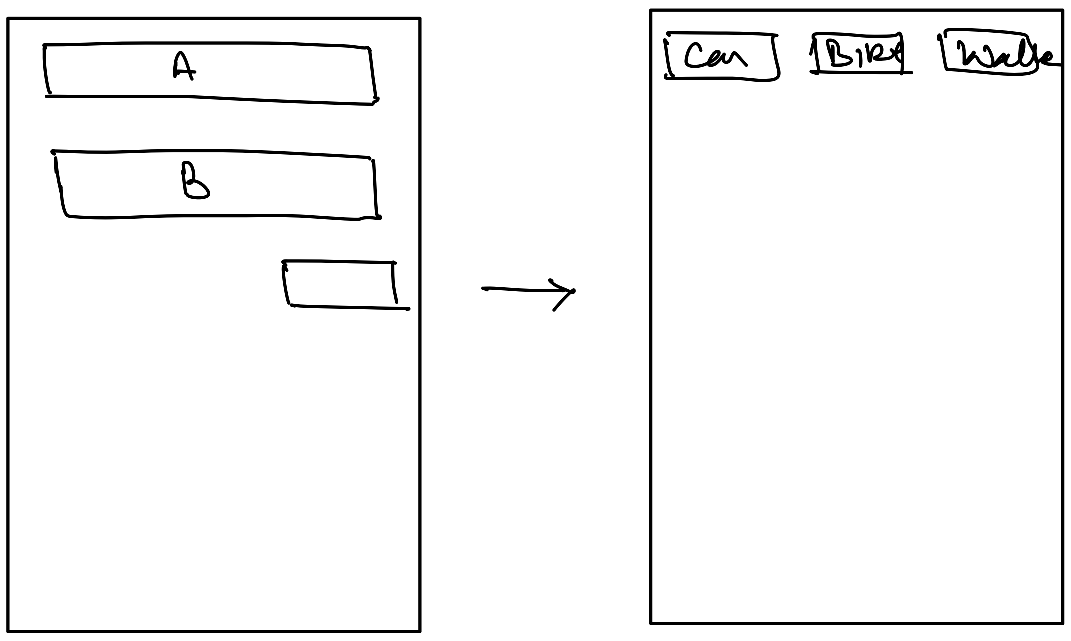
Behavioural Design Patterns.

↳ Related to behaviours | actions | methods.

Strategy Design Pattern

⇒ Google Maps.



⇒ Different mode of transportation.

⇒ When we search for a path on google maps it provides us multiple paths for different mode of transporations.

```
GoogleMaps {

    findPath(src, dest, mode) {
        if (mode == "Car") {
            ____
            ____
            ____
            ____
        }
        else if (mode == "Bike") {
            ____
            ____
            ____
            ____
        }
        else if (mode = ____ ) {
            ____
            ____
            ____
            ____
        }
}
```

Violates.

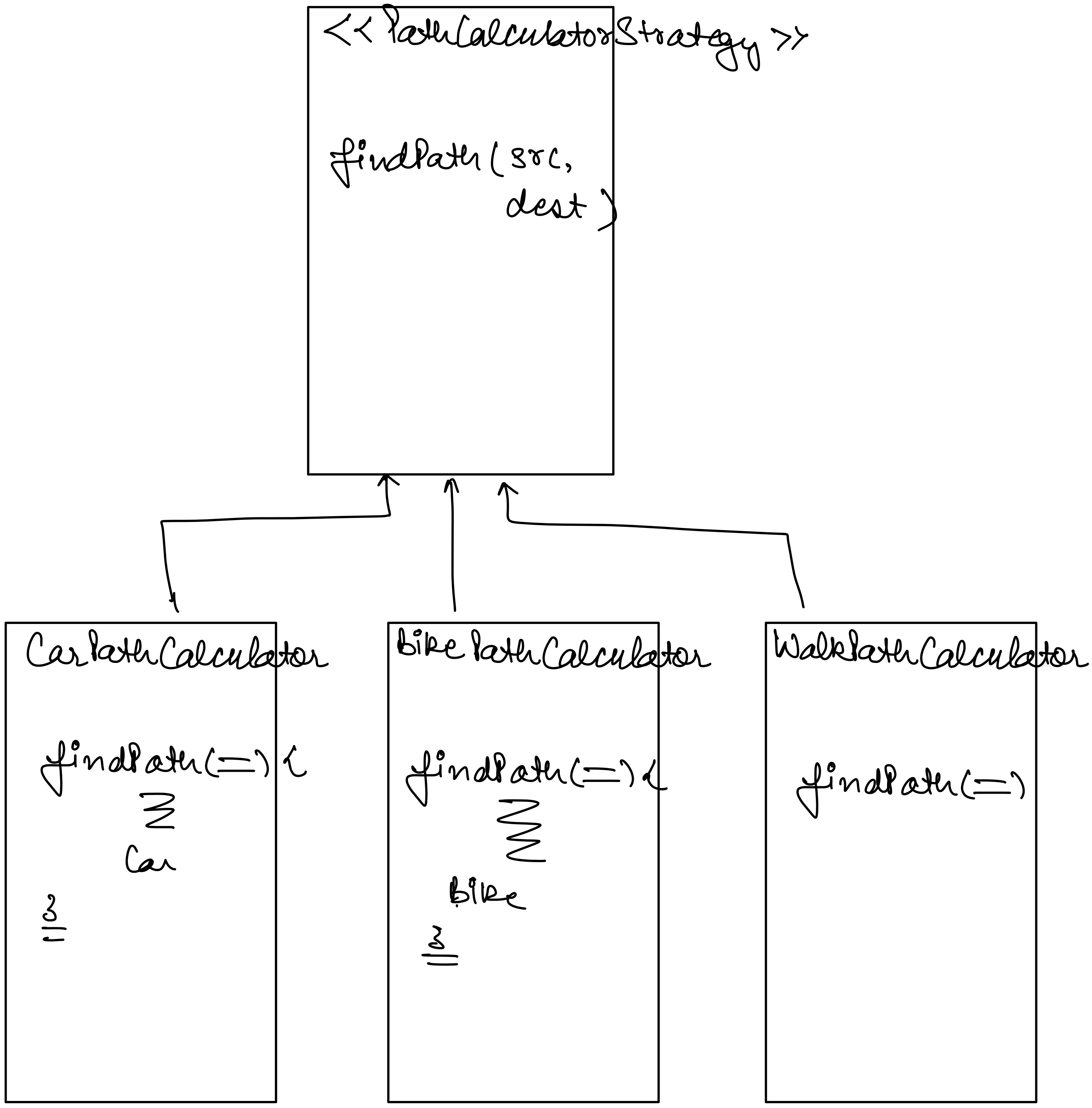
OCP ✗

SRP ✗

⇒ When we have multiple ways of doing something often we see the violation of design principles like SRP & OCP.

⇒ This can be solved using Strategy.

Rather than implementing these behaviours in a same method, move these into separate classes. for each way of doing something we'll create a new class.

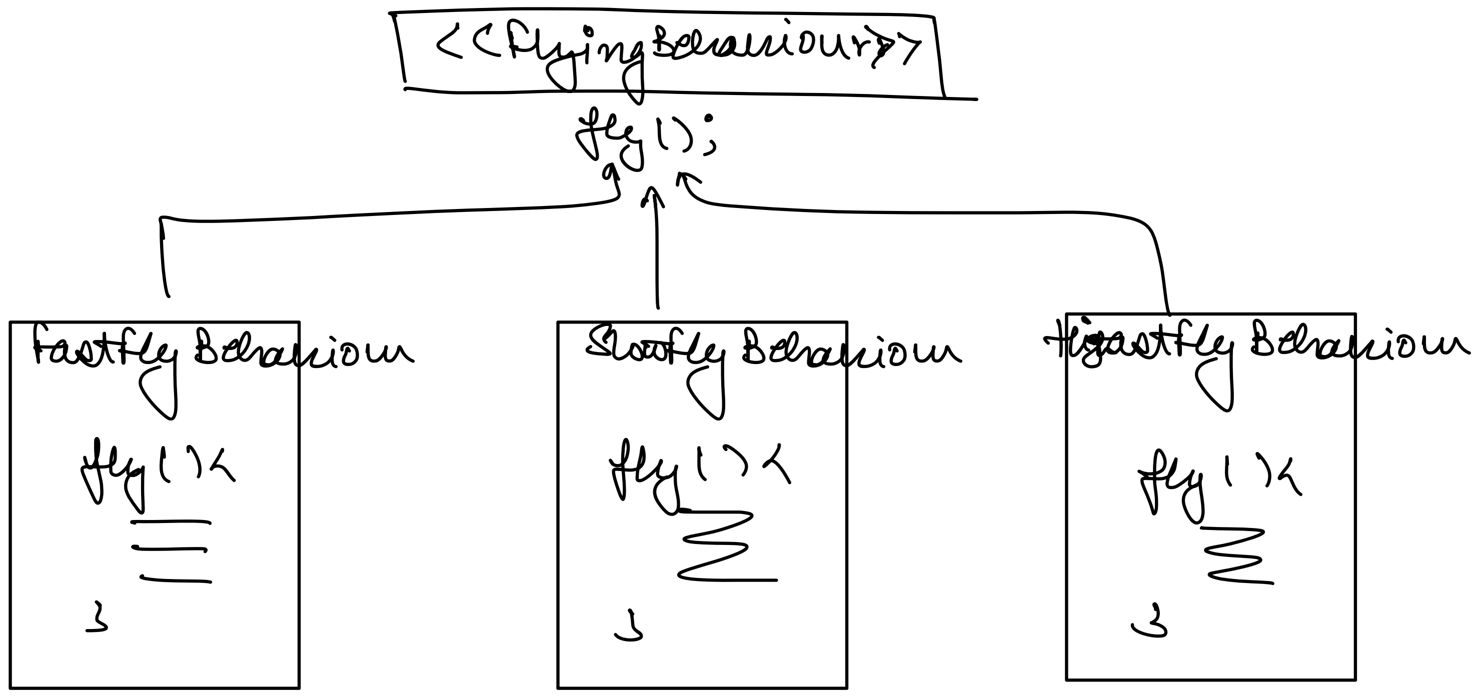

<< PathCalculatorStrategy >>
findPath(src, dest)
CarPathCalculator
findPath(=){
Car
}
BikePathCalculator
findPath(=){
Bike
}
WalkPathCalculator
findPath(=)

Client → GoogleMaps.

```
GoogleMaps.
PathCalculator pc;

findPath(source, dest,
                mode){
    pc = PCF.getPCforMode(mode);
    pc.findPath(src, dest);
}
```

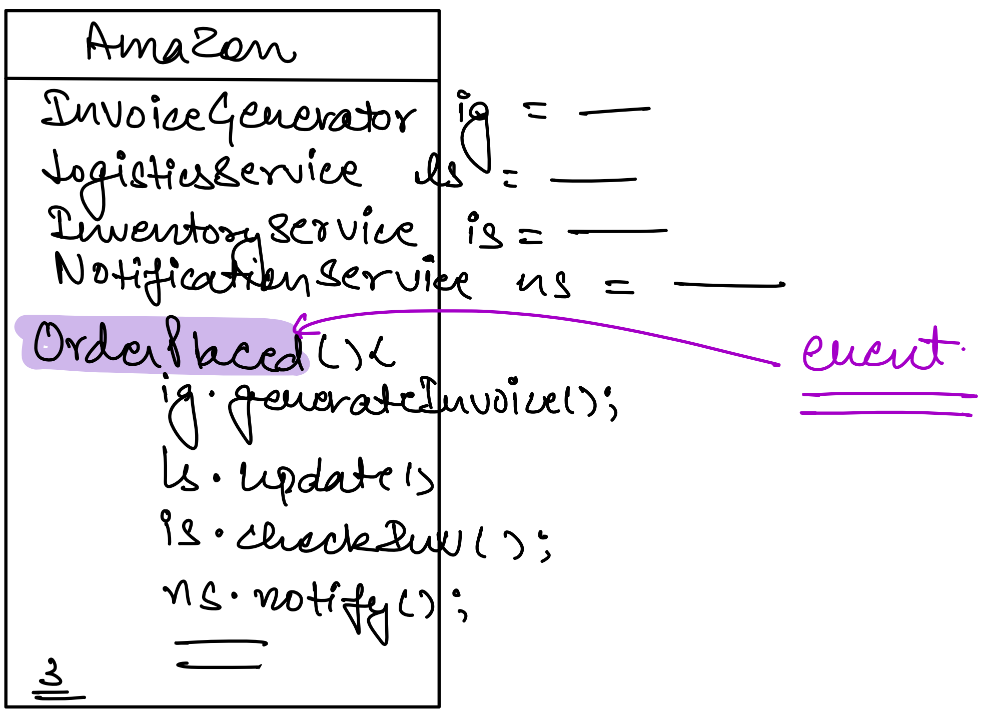
```
PathCalculatorFactory
Static { CPC cpc = new CPC();
         BPC bpc = new BPC(); }

getPathCalculatorforMode(mode){
        if(mode == Car){
            return cpc;
        }
        else if(mode == "Bike"){
            return bpc;
        }
}
```

⇒ Multiple ways of doing something.



Strategy : Multiple ways of doing something.

Observer Design Pattern.



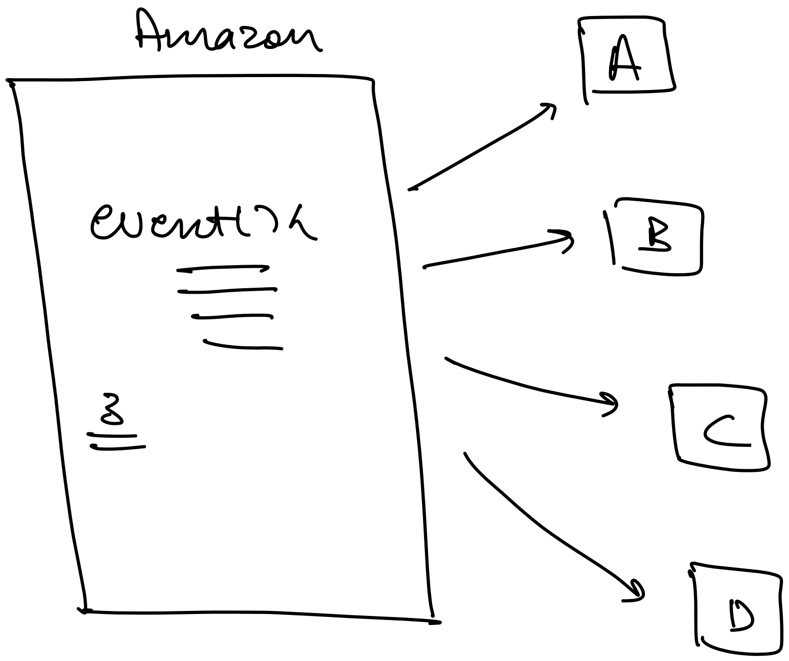
```
Amazon
InvoiceGenerator ig = ___
LogisticsService ls = ___
InventoryService is = ___
NotificationService ns = ___
OrderPlaced(){
    ig.generateInvoice();
    ls.update()
    is.checkInv();
    ns.notify();
    ___
3
```

Problem Statement

When an event happens, we might want to do lot of things i.e to add/remove some actions that we need to perform. In this case we'll have to do code changes and thus compilation of Applⁿ would be required.

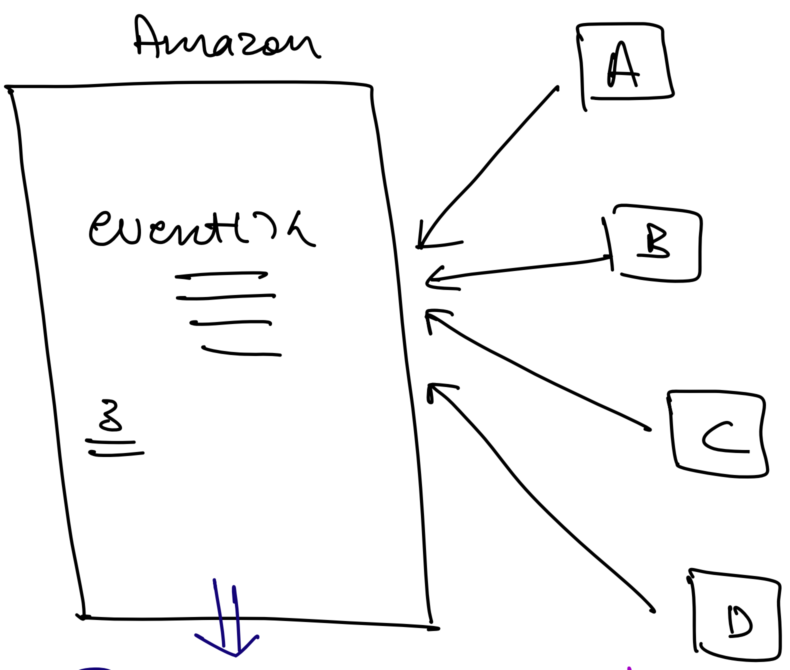
⇒ We can't add/remove functionalities at runtime.

Observer



We want to execute A, B, C, D if event is triggered

⇒ Observer says reverse the dependencies.



A, B, C, D wants to execute themselves if event happens.

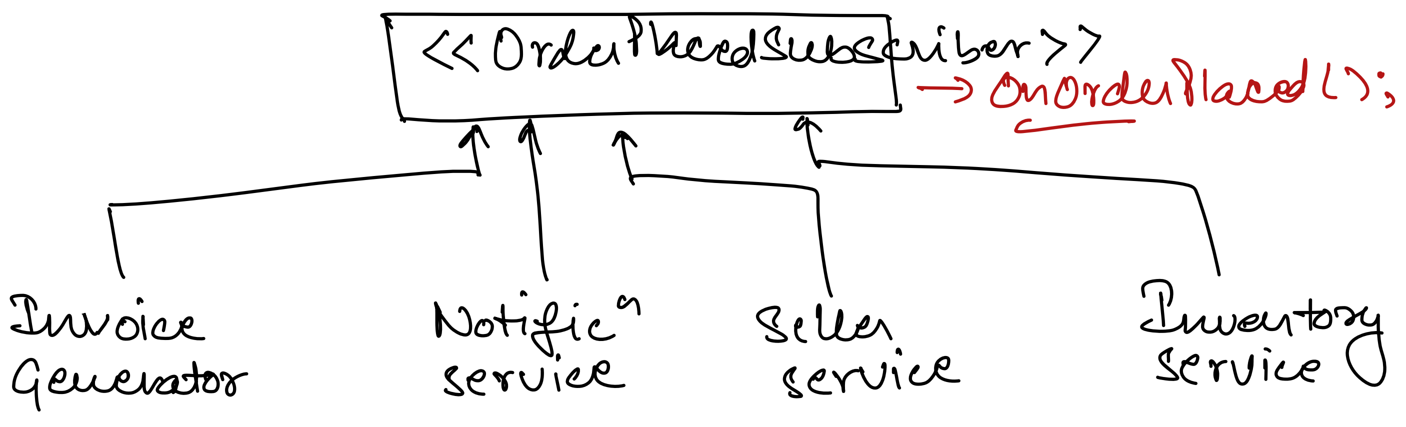
Publisher.

(event happens)

Subscribers.

→ Create a method in publisher that allows a subscriber to register itself

```
Amazon  → Publisher.
List<OPS> subscribers;

registerForOrderPlaced(OPS ops){
    subscribers.add(OPS);
}

OrderPlaced(){
    for(subscribers){
        subs.onOrderPlaced()
    }
}
```



```
InvoiceGenerator(){
    registerForOrderPlaced(this)
}
```

Invoice Generator object is registering itself for the event.