Agenda :-

Strategy

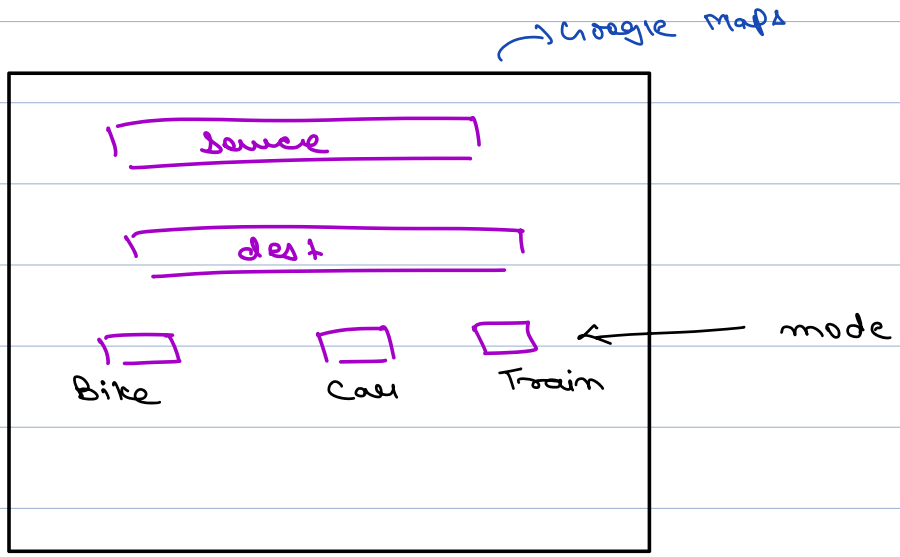
Observer

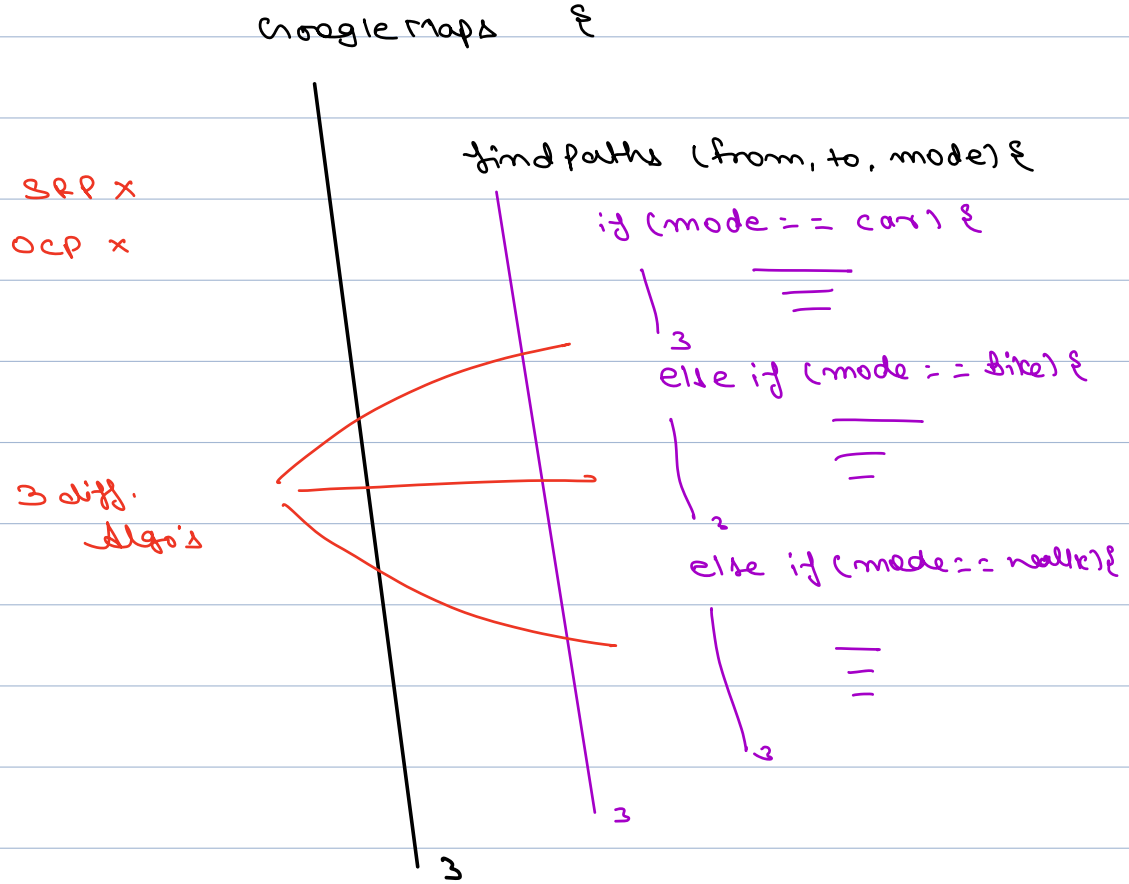
} Behavioural d.p.

↓

Comm b/w classes and objects

Strategy design Pattern.





Implement diff. classes for every algo.

carPathCal | BikePathCal | walkPathCal.

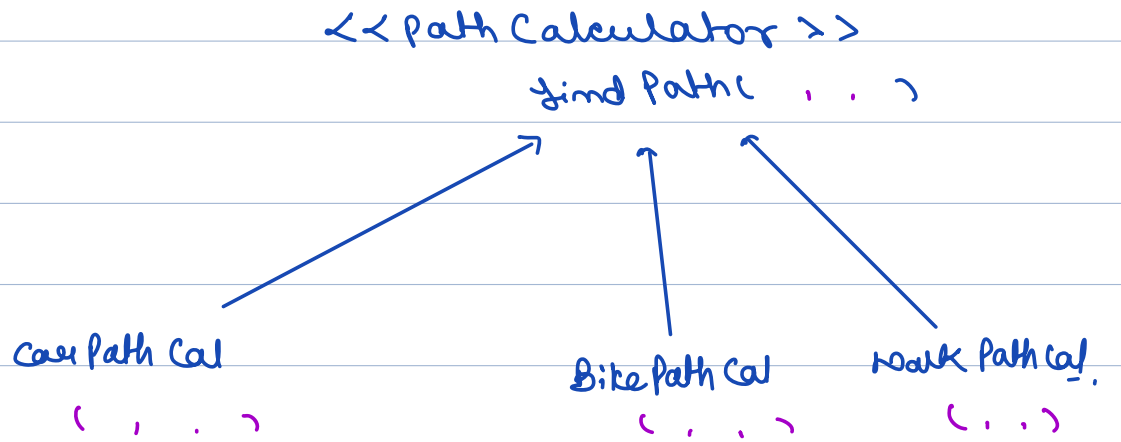
```
if (mode == 'car') {
    carPathCal n = new carPathCal();
    n.findPath();
}
```

DIP

violating dependency inversion principle

3

.
.
.

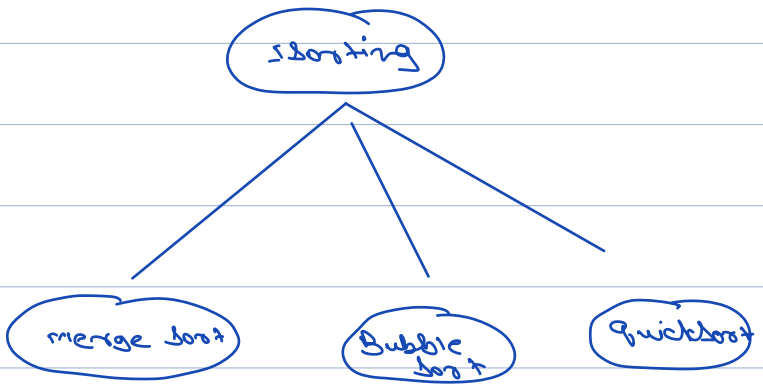


PathCalFactory {

static getPathCalculatorByMode(_) {

}

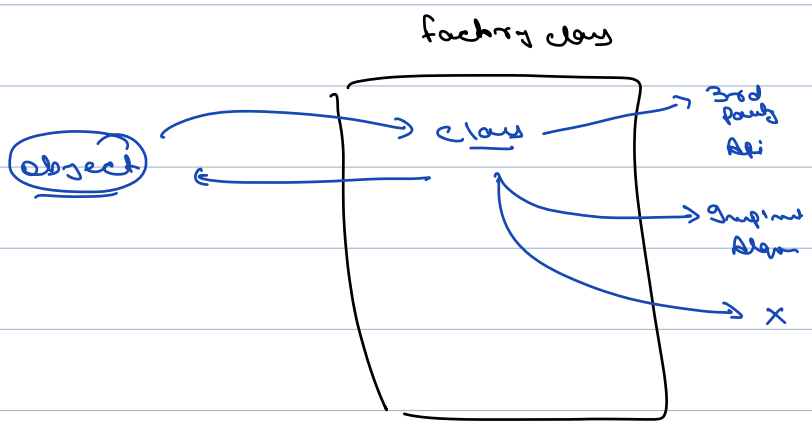
}



Strategy :- Rather than implementing, behaviour in a method, implant it in a seperate class.

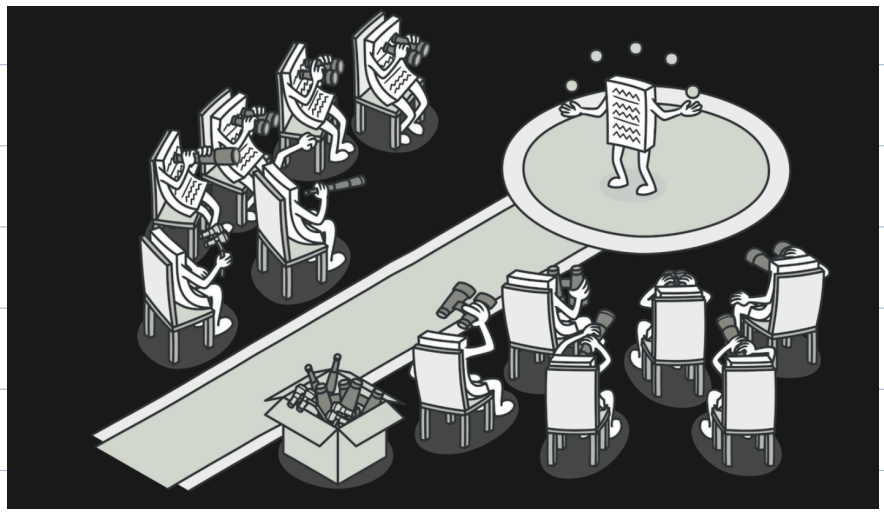
Strategy design pattern is one of the **behavioral design pattern**. Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

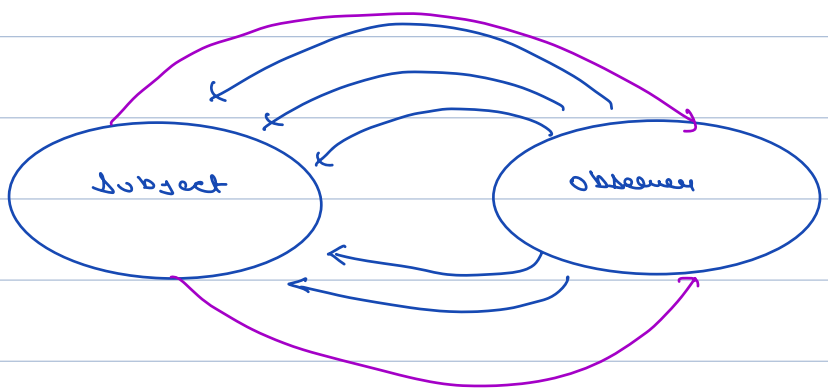
Strategy design pattern is one of the behavioral design pattern. Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime. It is based on the idea of encapsulating a family of algorithms into separate classes that implement a common interface.

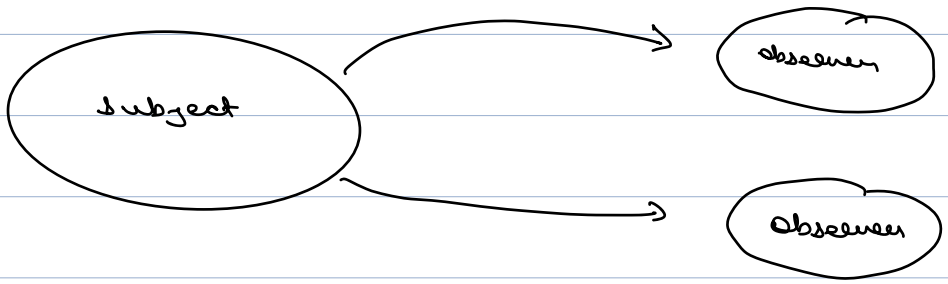


Observer pattern.

Polling

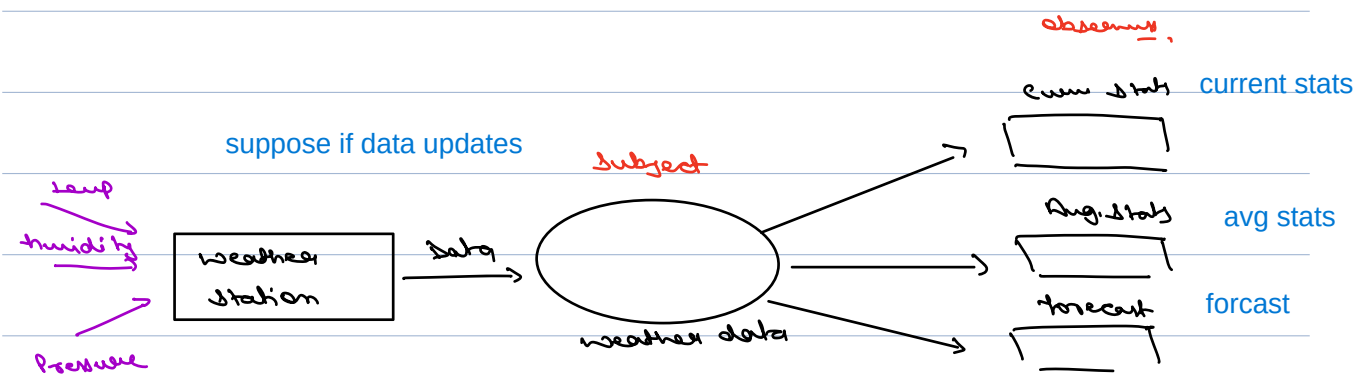




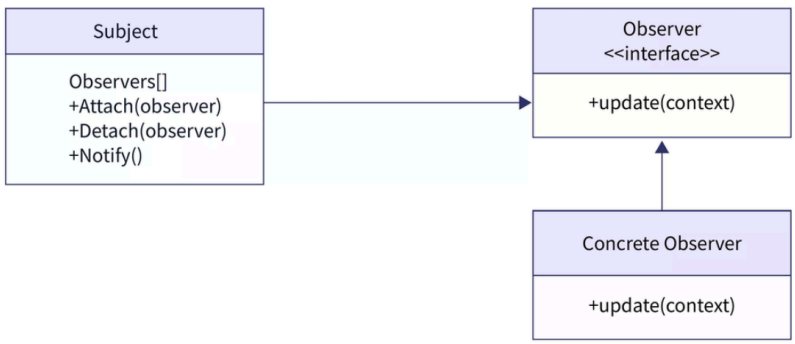


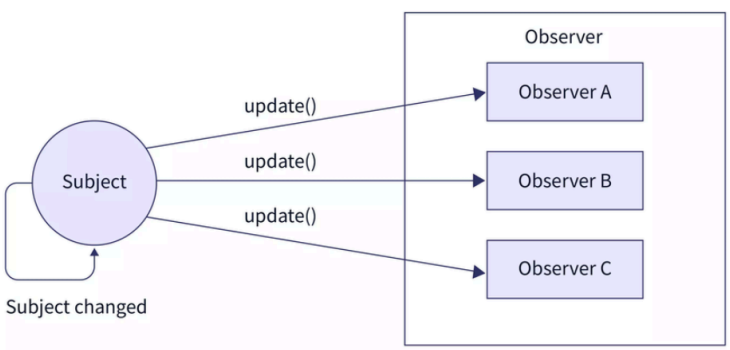
all these data should update

suppose if data updates



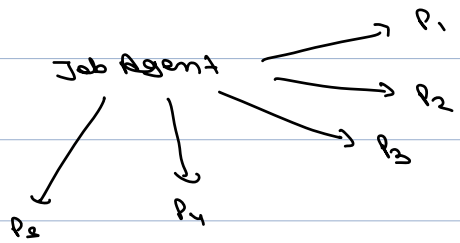
subjects can be multiple
and observers can be multiple.

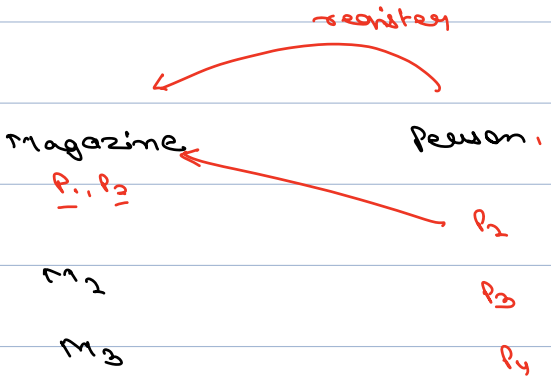


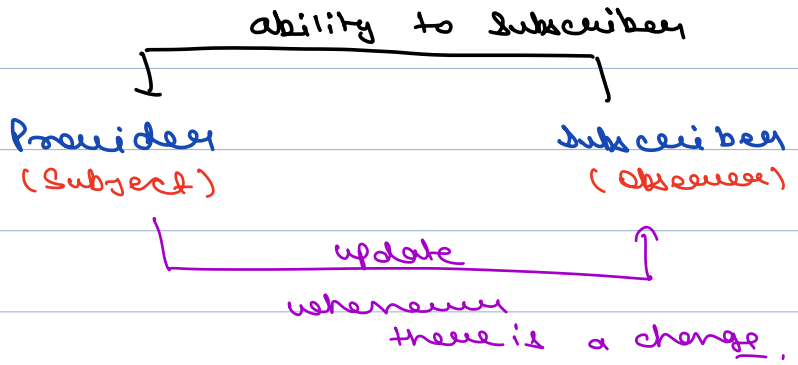


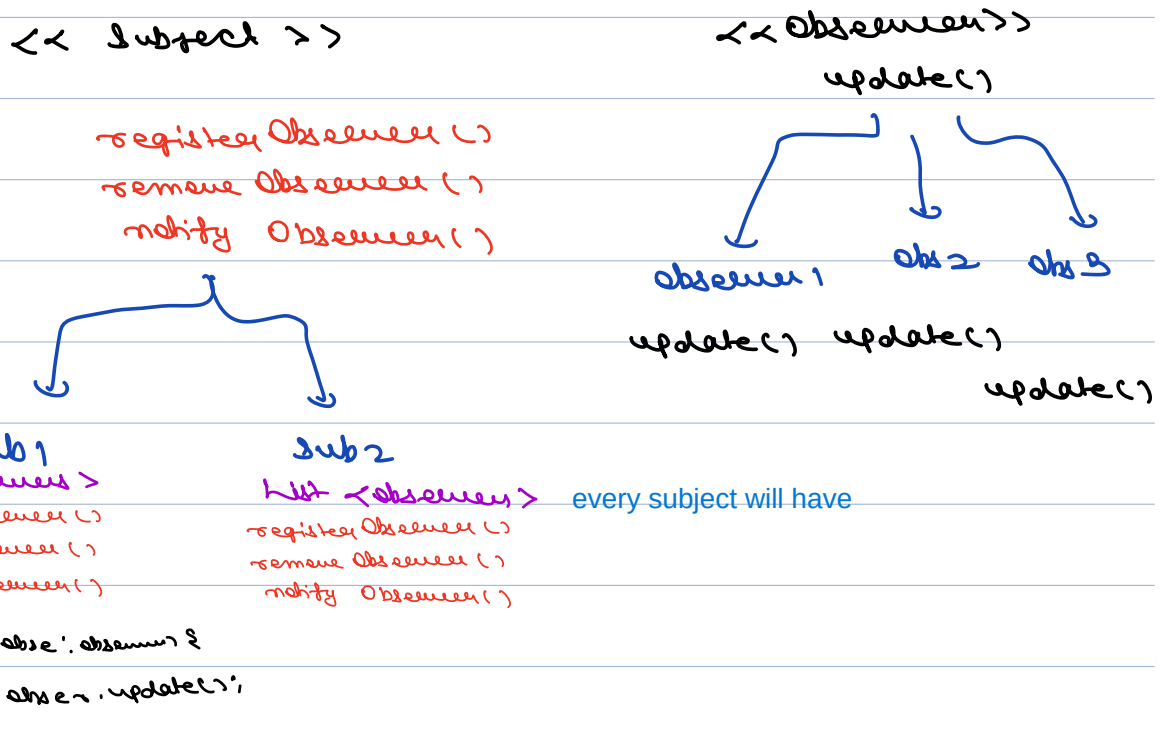
if it gets any job

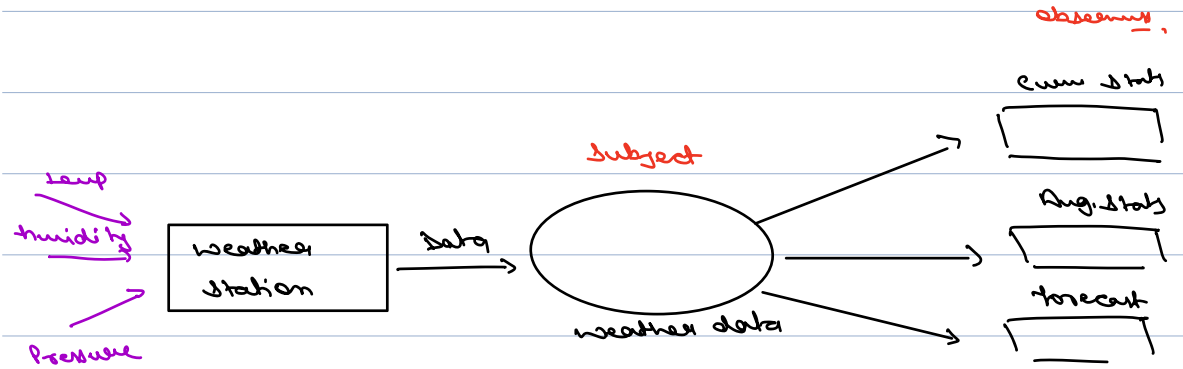
all the user who are interested in these kind of job should get updated.






ability to subscribers
Provider
(Subject)
Subscribers
(Observer)
update
whenever
there is a change.


<< Subject >>
registerObserver()
removeObserver()
notifyObserver()
<<Observer>>
update()
Observer 1
Obs 2
Obs 3
update()
update()
update()
Sub 1
List<Observer>
registerObserver()
removeObserver()
notifyObserver()
Sub 2
List<Observer>
registerObserver()
removeObserver()
notifyObserver()
every subject will have
remove
for(obs: observers){
obs.update();
}


temp
humidity
pressure
weather station
data
subject
weather data
observe
curr. stats
Avg. stats
forecast

The object that has some interesting state is often called *subject*, but since it's also going to notify other objects about the changes to its state, we'll call it *publisher*. All other objects that want to track changes to the publisher's state are called *subscribers*.

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

↑ weather data

<<Subject>>

registerObserver()
removeObserver()
notify()