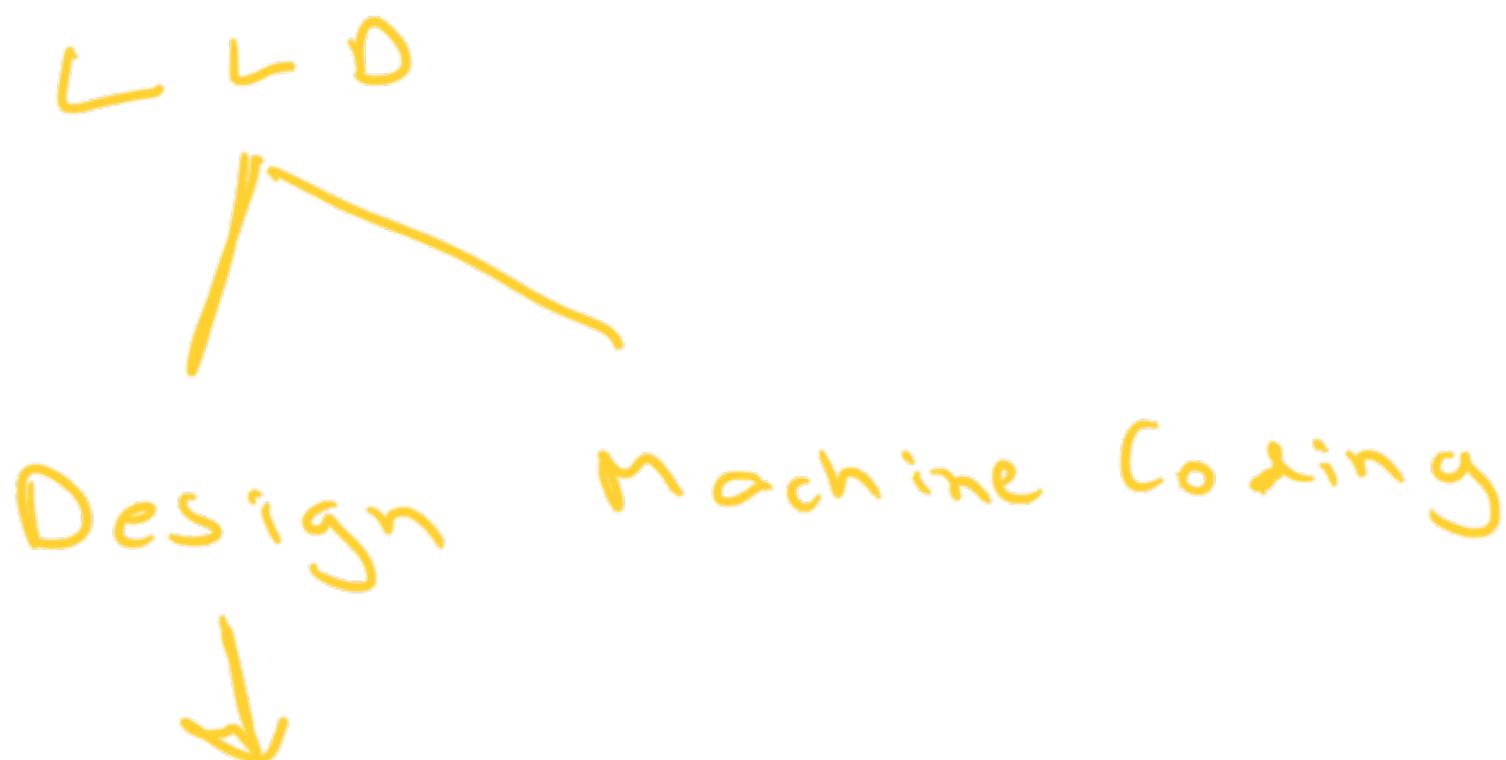


## LLD - Facade and Observer

- Facade
  - Observer
    - event driven
  - Strategy
- 



- entities
- class Diagram
- schema diagram

UML + schema diagram

Real world

→ Pen - Notebook

Games

→ Tic Tac Toe - Chess

→ Snakes & Ladders

Movies + Systems

Frontend

→ Parking lot

→ BNS

→ Email campaigns

Machine coding

- Splitwise

- Spring Boot

- JPA

---

Facade



Complex



→ Present a clean, simple  
interface to hide internal  
dragons

E-commerce - Flipkart Orders  
Inventory Service Inv.

Create Order( ... ) {

{ → inventory. checkout() ;  
Payment. checkout() ;  
Recm.. checkGut() ; }

Create order →

- quantity check — Inventory Service
- update inventory — Payment Service
- generate payment link —
- tracking service —

- Analytics service -
- Recommendations -

## Testing

- mock all deeper der API
- easy

## Extensibility

- Code duplication

Controllen



Repository

Order Manager {

Create Order ()

S



OrderV1

OrderV2

} depends

---

Order Manager Impl

Order Processor procj

Create Order Cj

Processor . process Cj

3

3

→ P...

Testing ✓

Extensibility ✓

SRP

Face abc

Order Processor & SRF?

Inv inv

Pay inv

Process () &

inv. checkout();

analyze . truck();

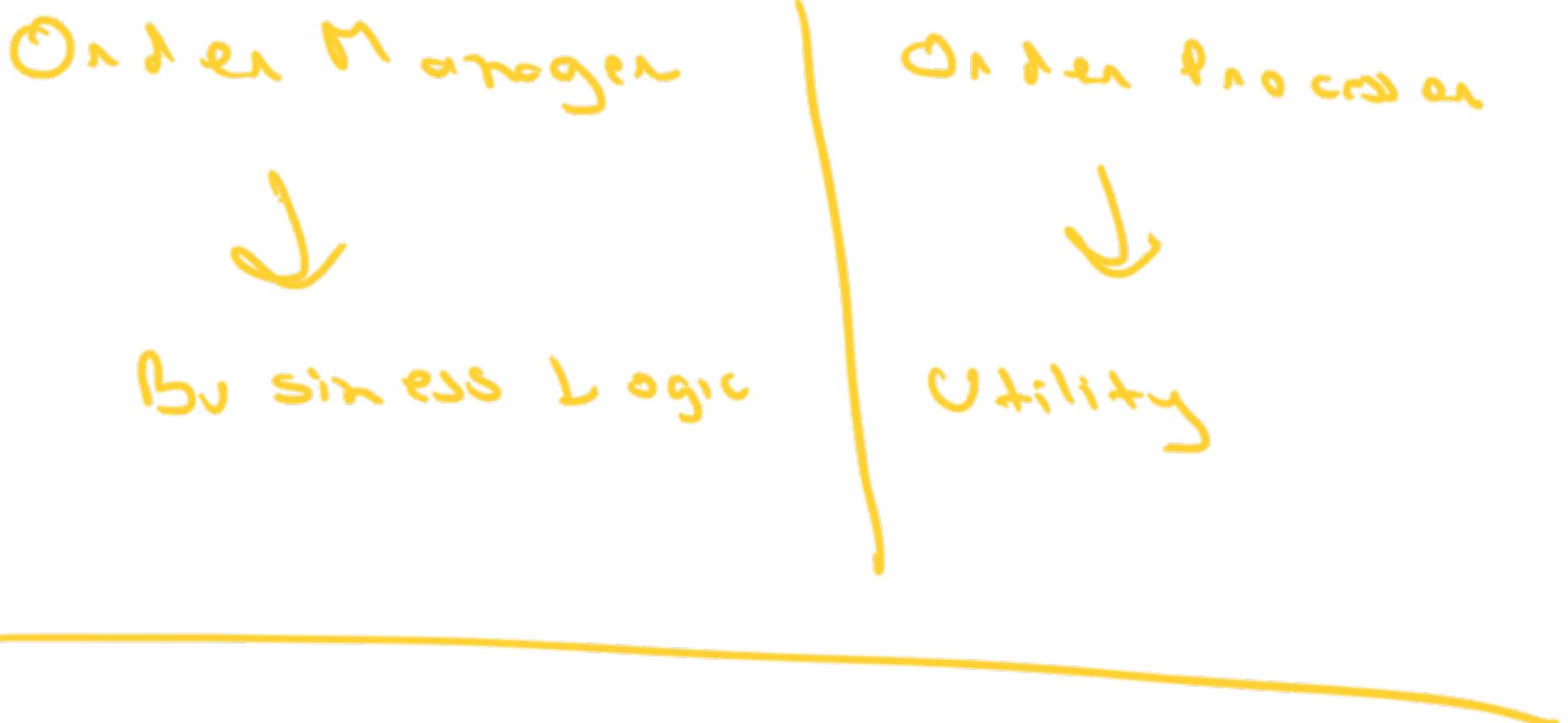
}

}

JDBC

Database. create (url)

---



Disadvantages -

- Good class
- Additional facade

Order

Order Manager Order Processor



Payment  
Processor.

Modification  
Processor.

Notification

notification. send(- -)

- Dot chess
- history
- slack / phone / email

## Synchronization

Synchronization

↳ user will wait





Sync Path

Req  $\Rightarrow$  Order Manager

Req Proc

Req Proc

Order Processor

$\rightarrow$  Queues

$\rightarrow$  Pub/Sub

P sync

## Facade

→ clean interface  
hiding complex  
Scenarios.

## Order Processor

- └→ Payment process.
- └→ Analytics Engine

# Behavioral design patterns

→ how best to implement

common behaviors

Strategy  
↗

→ functionality

Bind → Flying Behavior

Strategy → switching algo. attribute

Observer →

Observer



→ continuously goto the stone

→ wasteful

→ send everyone an email  
→ wasteful

→ subscriber register = mailing  
list of subscribers → send an email

Observe

↳ notify interested interfaces

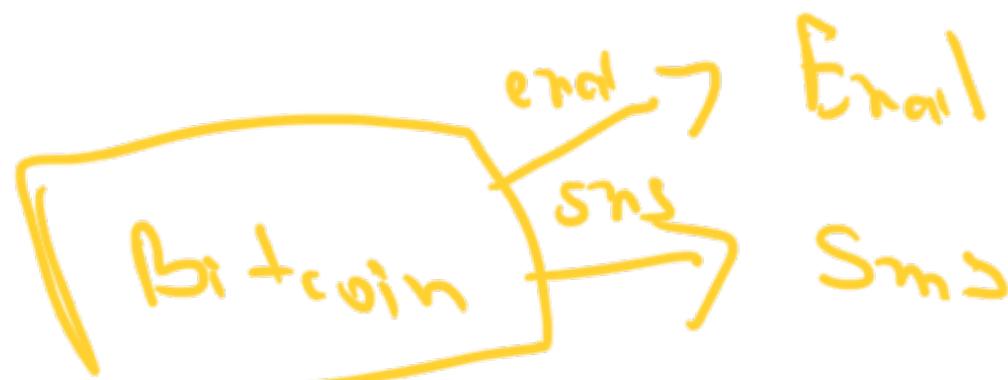
— price of bit coin ↗

- email, tweet ↴

→ email  
→ tweet  
→ stock  
→ SNS

Bitcoin Manager Impl ↴  
Bit coin bit coin; TweetService  
Email Service email;  
⇒ public void setPriceC int){  
    bit coin .setPrice();  
    email. send();  
    + tweet. send();  
↳

OCP  
SRP



②

## Polling



if price != previousPrice  
Send a Notification

Bitcoin Poller 2

Bitcoin Manager  
previous

Tweet  
Email } notification

```
poll() {  
    1 sec.  
    mongo.getPrice()  
    if (price != previous){  
        send notifications  
    }  
}
```

→ wasteful

→ period = lmin  
 - 1sec  
 - 1day

bit coin → deeper dev.

→ dependency register with Observable  
Publisher

→ when state changes, publisher  
calls each Observable — En oil  
↓ — Trust  
subscriber.

→ En oil → subscribe

→ Bit coin price changes

→ Publisher → send a message

to your subscriber.

The couple dependent actions when one action  
triggers multiple reactions

Observer

→ Ecommerce → Triggers

→ Type ORM



Subscribers

→ and so on

✓ event driven



---

6:09 - 6:12



---

Steps

✓ redressing

① Create an observable interface



Subject

publisher

bitc om Manager.

interface



Observable S

register C<sub>i</sub>j

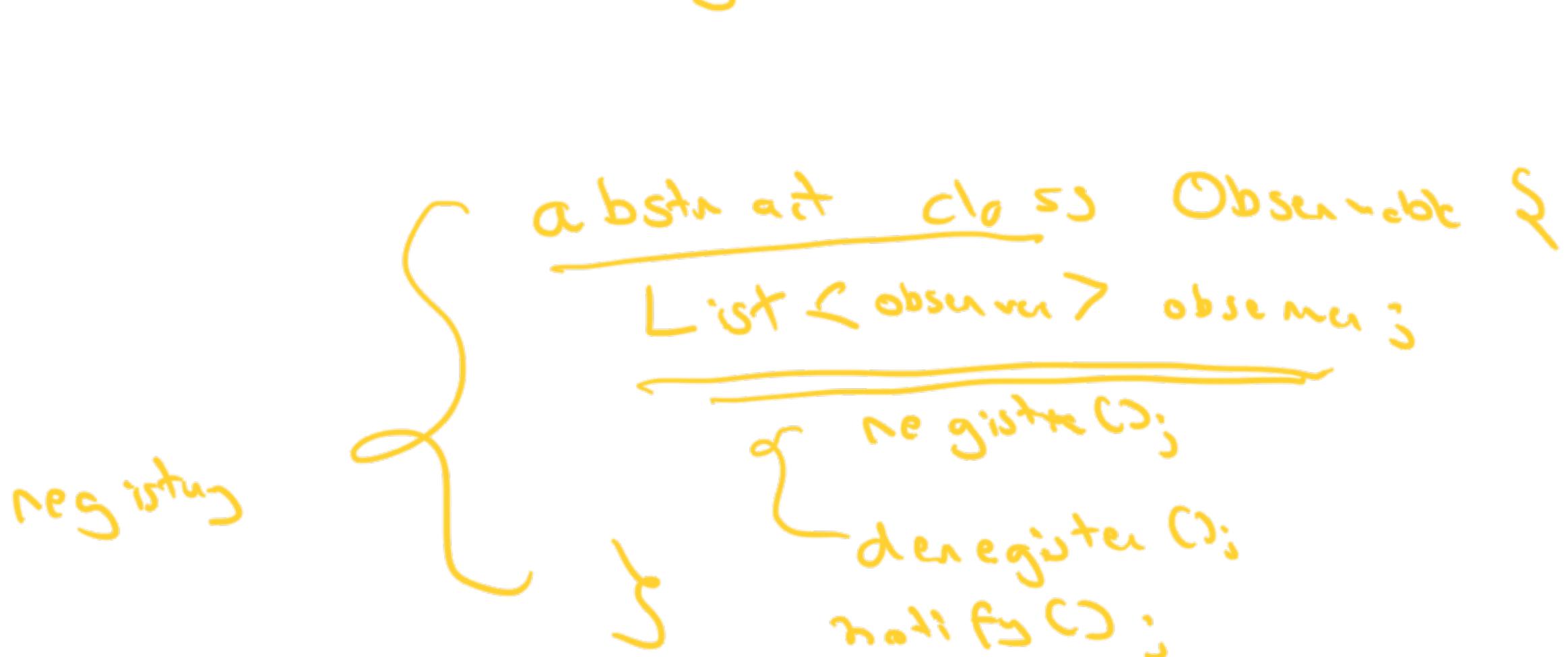
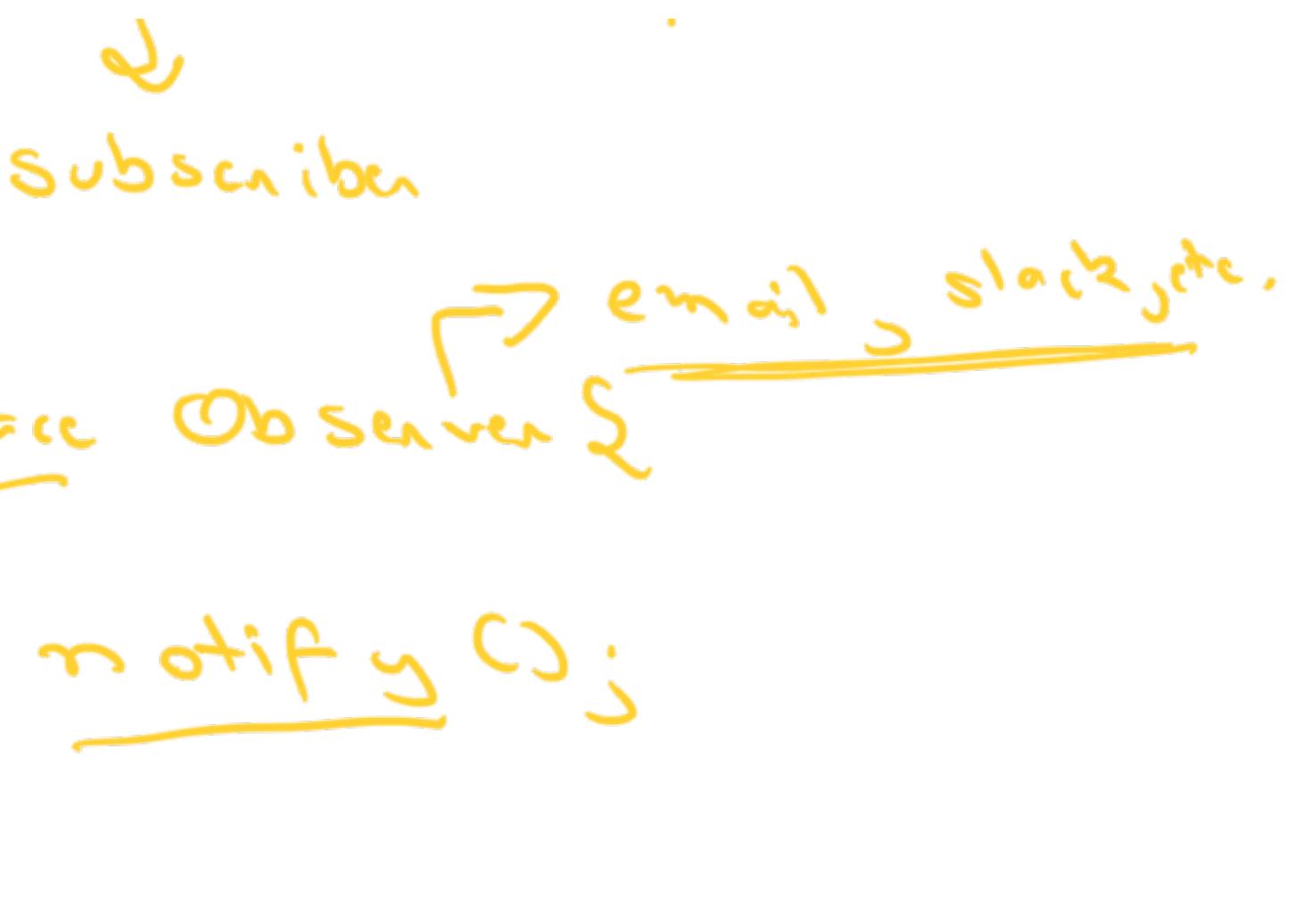
deregister C<sub>i</sub>j

Let

②

Create your observer interface

→ interested



③

Concrete Observable

bit coin Manager extends Observable

set Price () {

↳ notify ()

↳

④

Concrete Observable

class EmailService implements Observable {

→ notify () 2  
|| send a animal  
3 3 ~~animal~~

## Angular - Observables

- promise
- Redux Store
- React - observe
  - in the section - obsrv.

## Factory

- don't want to use subclasses
- why → OCP
- maintenance



⇒ Simple factory

→ Create a static method to access type

→ Based on type, create instances,

→ createBtn (BtnType) {  
switch (type) {  
} → P: return P button () ;  
}

→ O/C

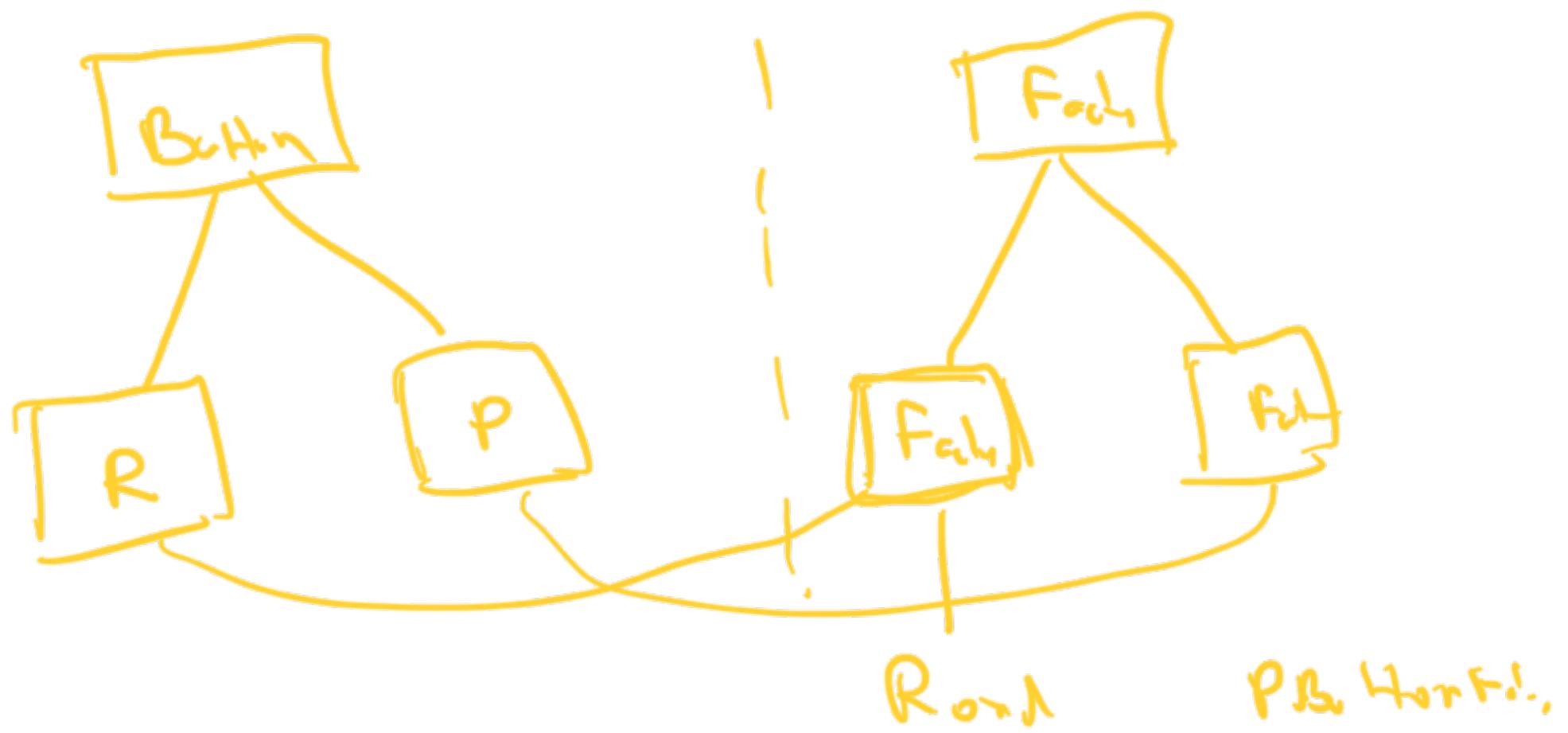
→ maintainable

→ readability

Btn btn = SimpleFactory.  
CreateBtn [ Round ]  
↳ Programming glo interfaces

One class → multiple objects  
<sup>inst cases</sup>

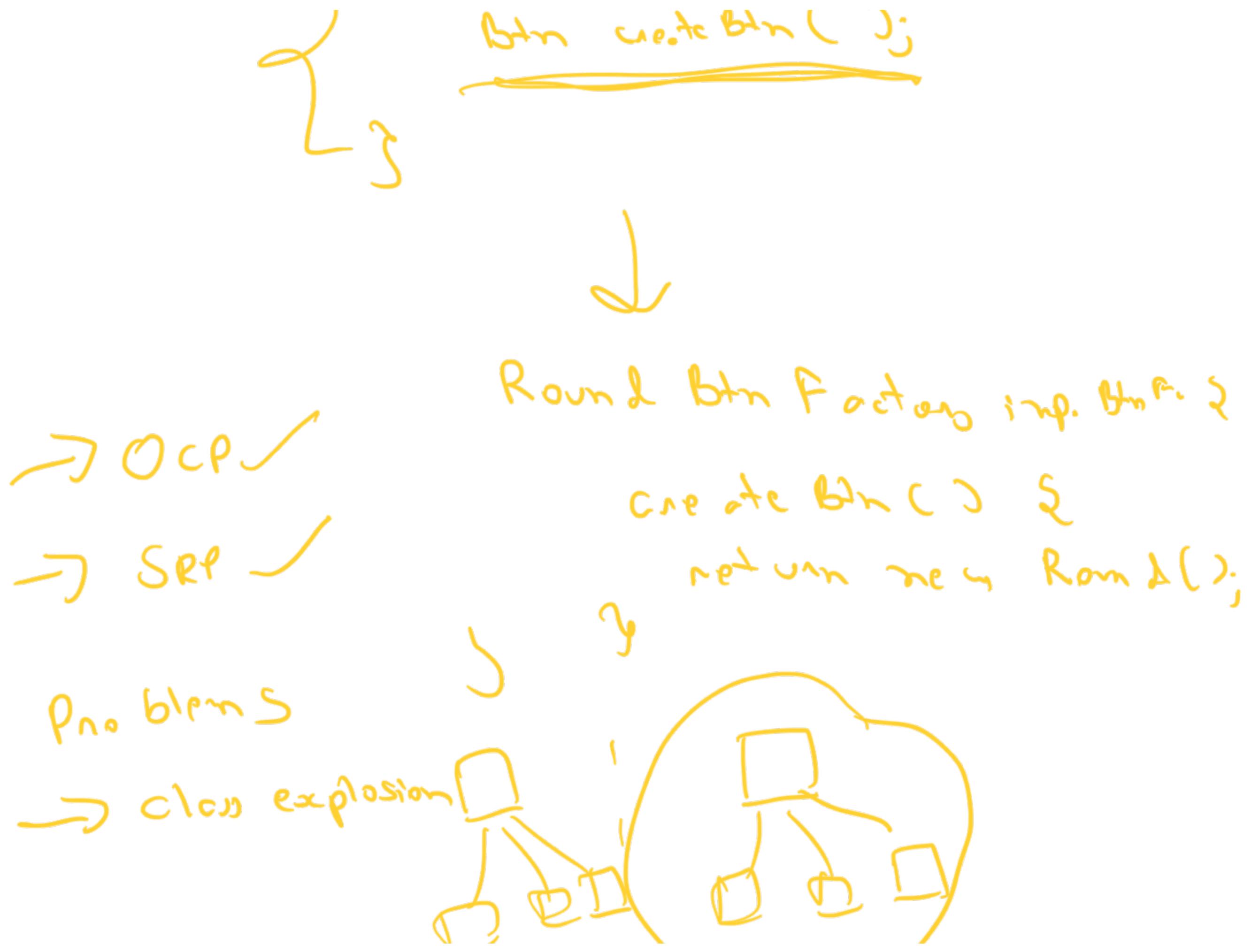
On class  $\Rightarrow$  ~~one~~<sup>instace</sup> of one class



# Factory interface

Tractor

factory  
method



## Abstract factory

- relate set of products
- family of prod.





Abstract factory → factory of factories

Theme factory {

create Btr();

create Check box();

