1. Good Evening

2. We will begin at 9:10pm

3. Topic → 1. Adapter
2. Facade
3. Code Factory

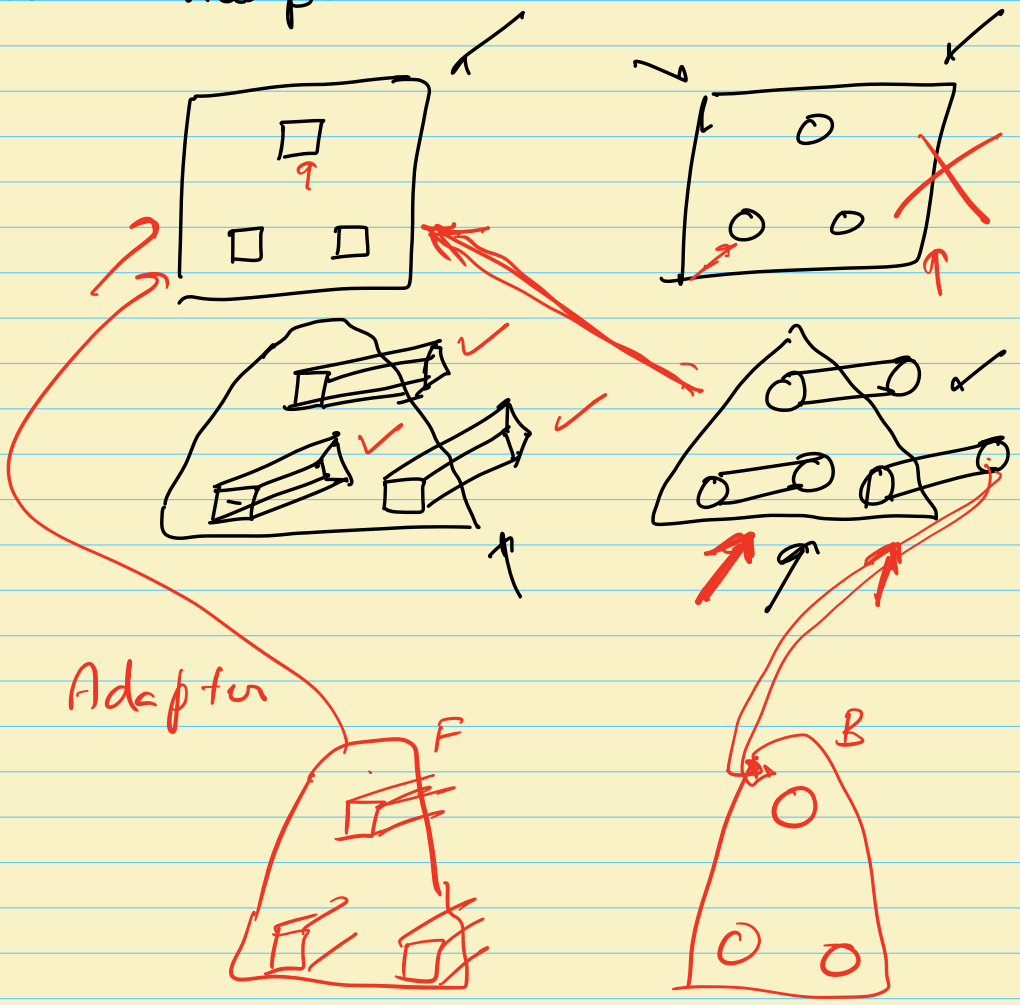
Agenda

1. Adapter Design Pattern
2. Facade Design Pattern
3. Code for Factory → Abstract Factory
Practical Factory

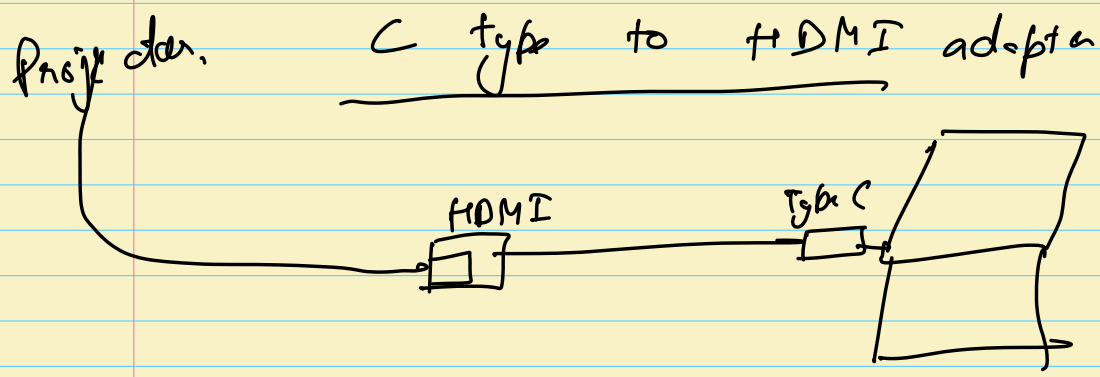
Adapter Design Pattern

Adapter

1. Power Adapter

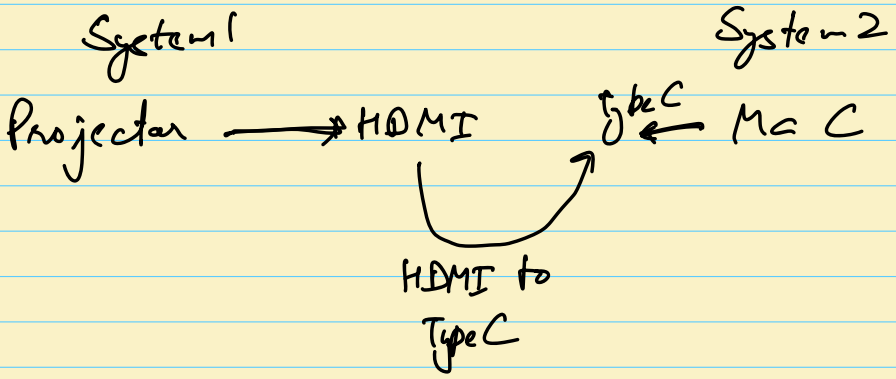


2. Mac [Type C]



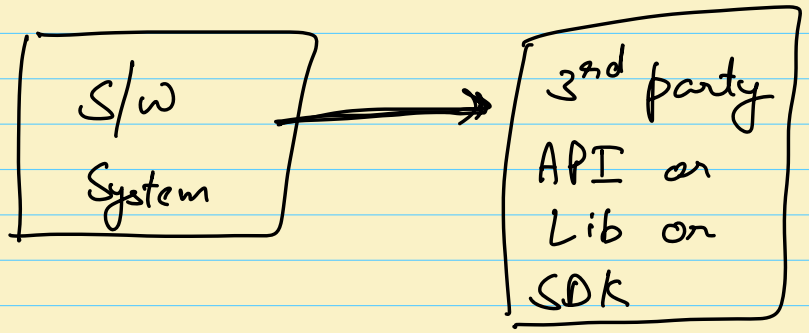
C type to USB

VGA to HDMI Adapter



☆ An intermediary that helps transform one form to another form.

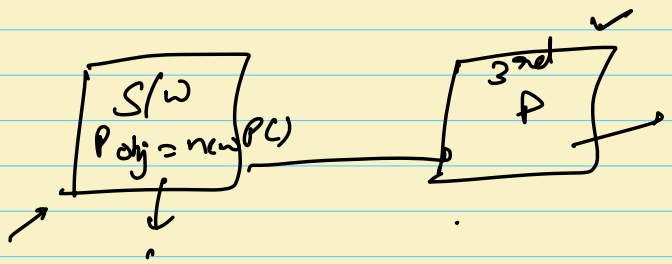
Scenario for S/W world



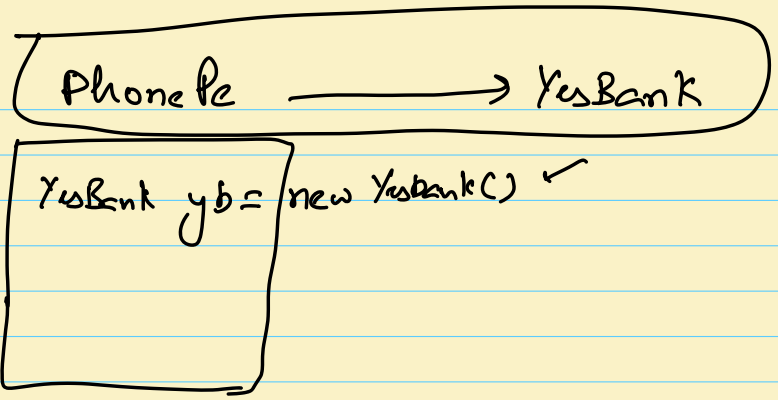
Possible Maintenance problem

1. We might want to switch to an alternate 3rd party API

2. API we are depending might have maintenance problems.



★ Adapter design pattern ensures one's codebase remains maintenable while using 3rd party API / Lib / SDKs



→ Violation of Dependency Inversion

High level modules should not directly depend on low level modules, instead they should be interacting via interfaces/abstractions.

PhonePe → YesBank

- ☆ Tight coupling
- ☆ Violation of [Dependency Inversion Principle]

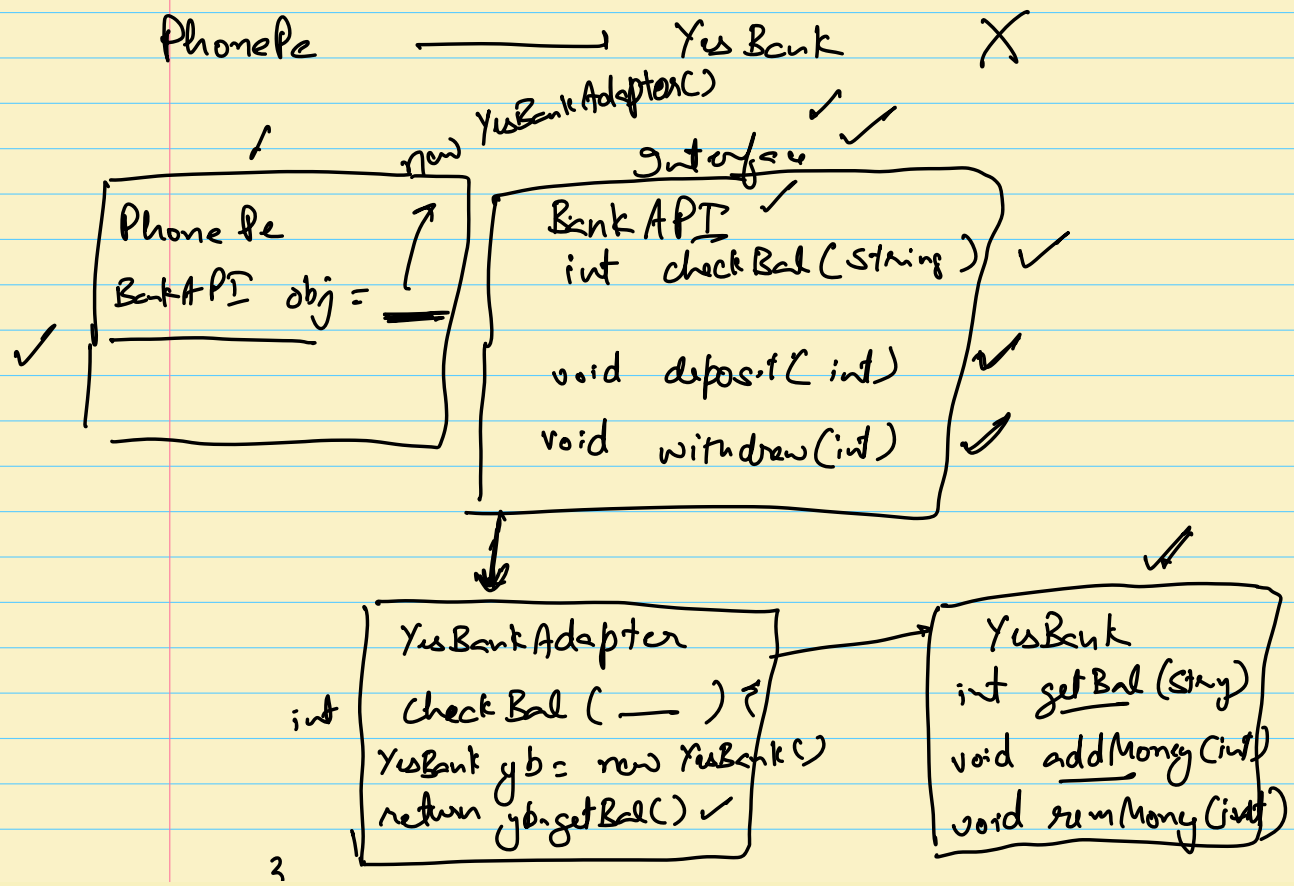
Client → 3rd party Lib ✗

Client → Interface ← 3rd party Lib

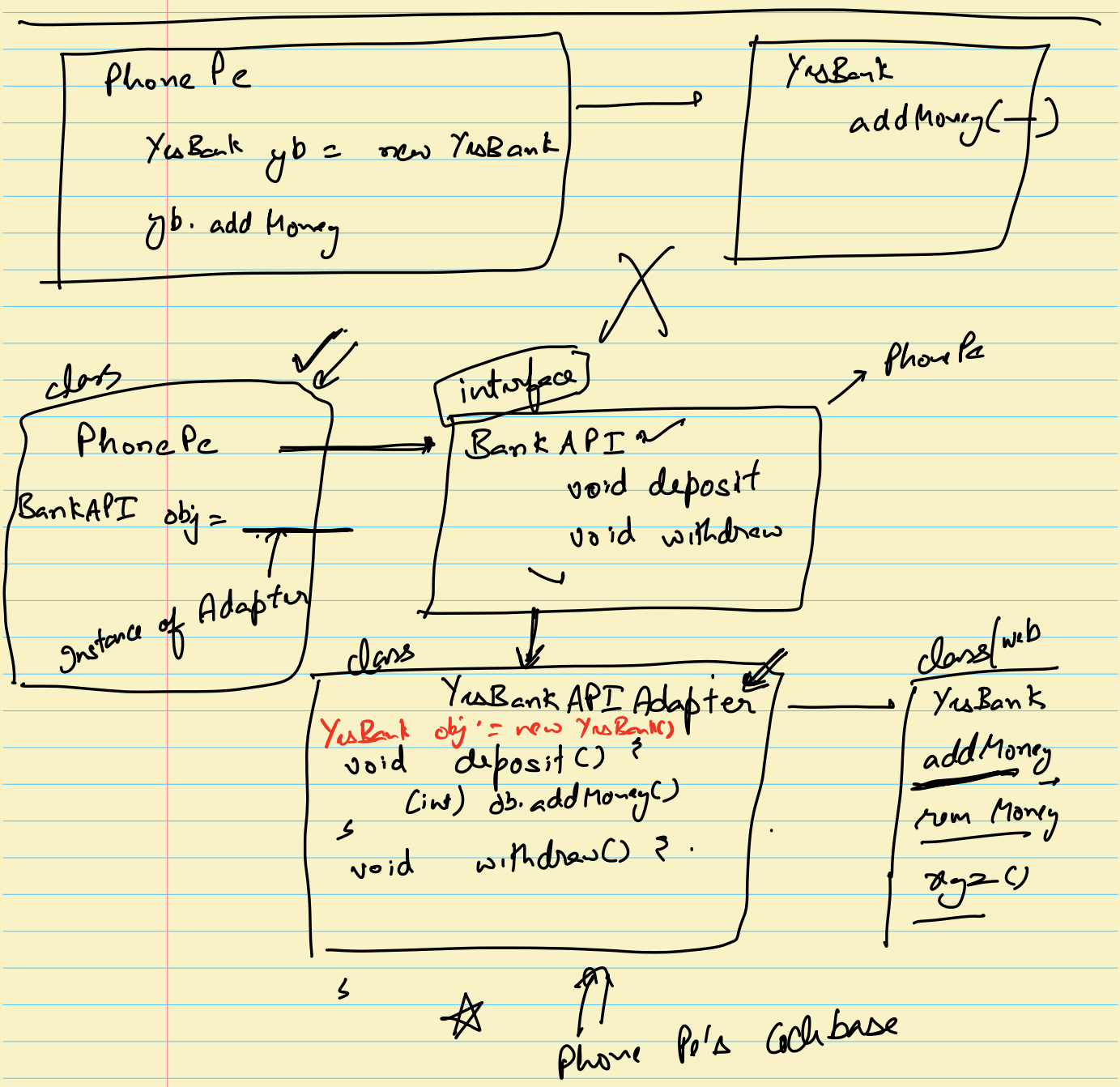
- ☆ Who implements the interface? 3rd party or Adapter
- ☆ Who consumes the interface? Client

How to use adapter?

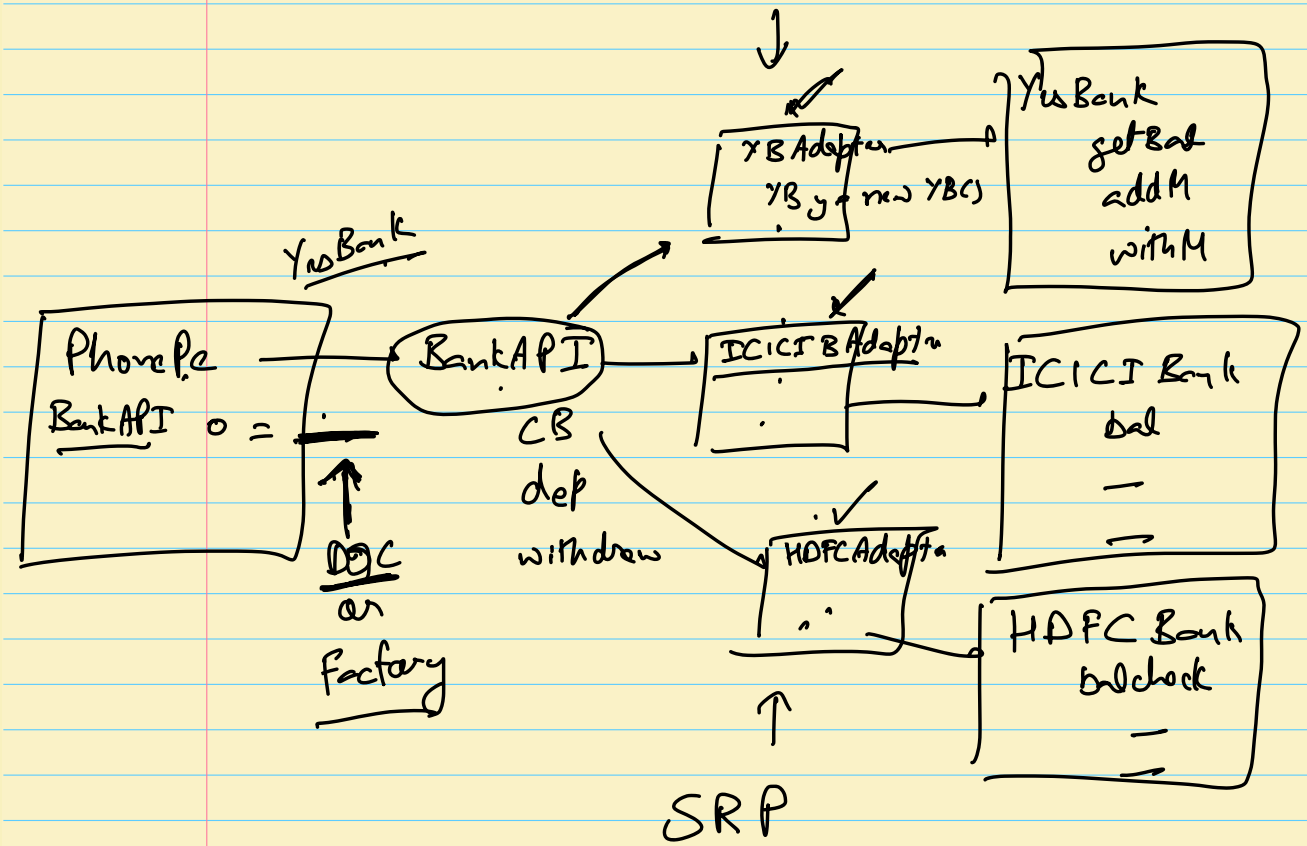
1. When connecting to 3rd party libraries, create an interface.
2. Create an adapter class implementing the interface, & calling the 3rd party.



```
void deposit(int x) {
    YesBank yb = new YesBank()
    yb.addMoney(x)
}
```

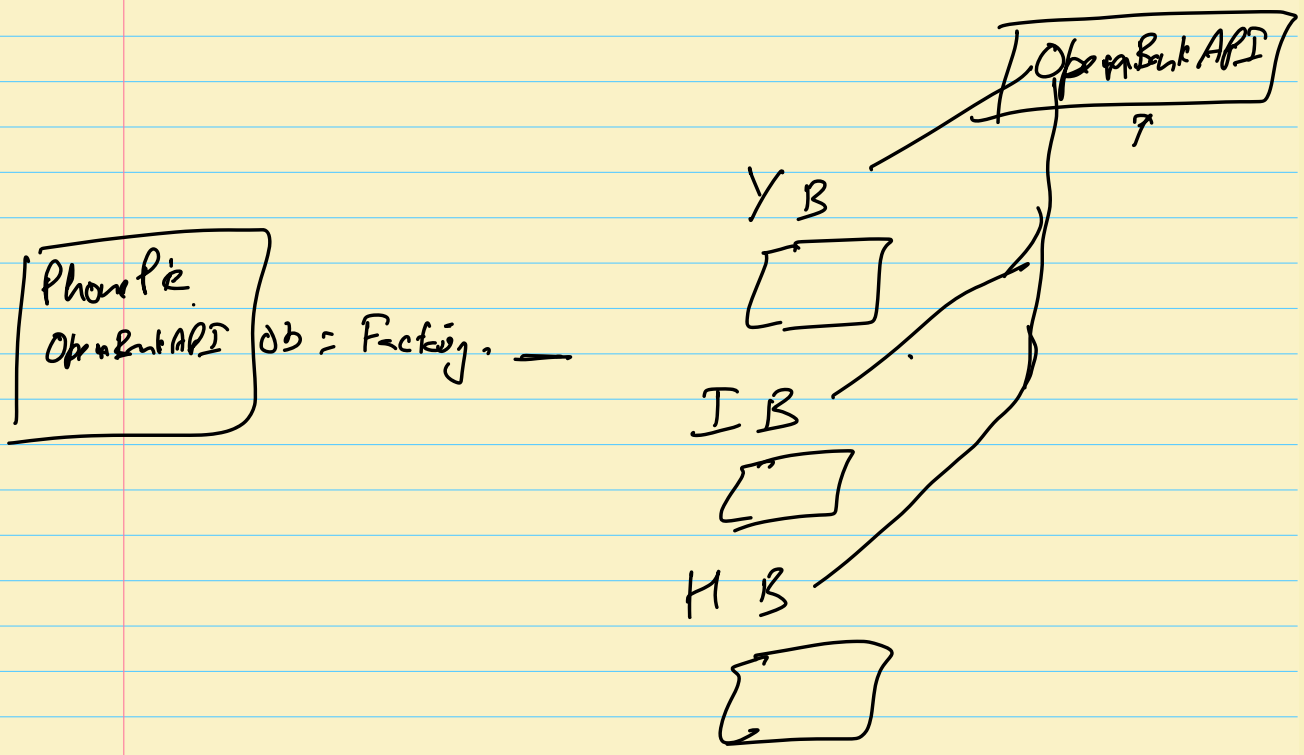


1. PhonePe wanted to use a 3rd party API
2. PhonePe wrote an interface to tell its requirements
3. Interface shared to vendors
4. Adapter written & 3rd party lib consumed

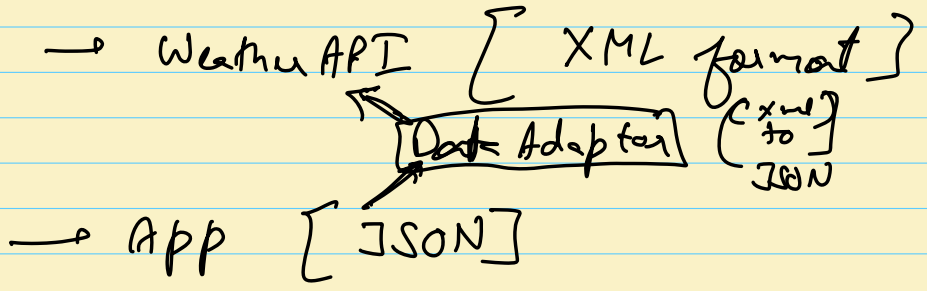


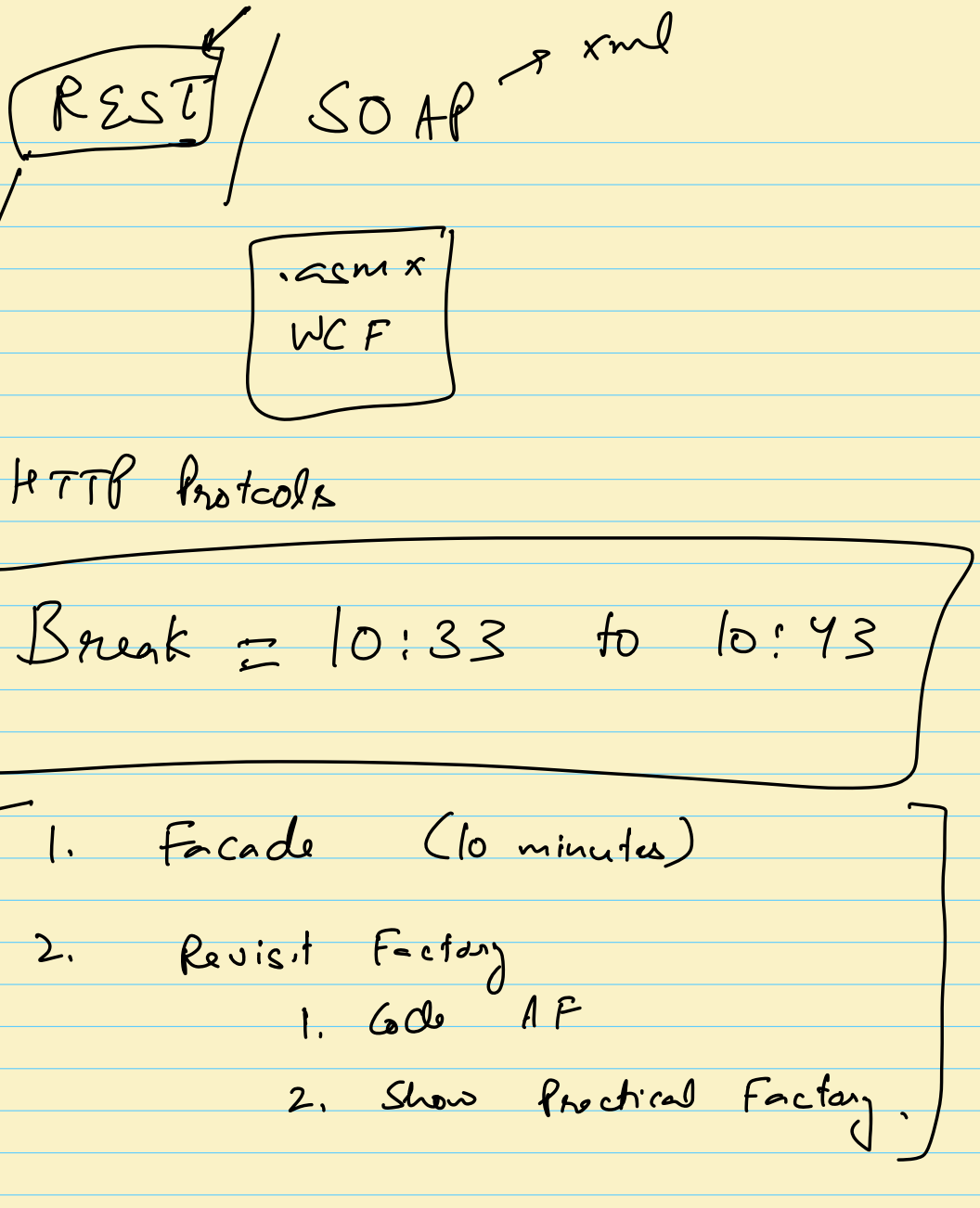
Code Adapter.

Production Level Code → Github

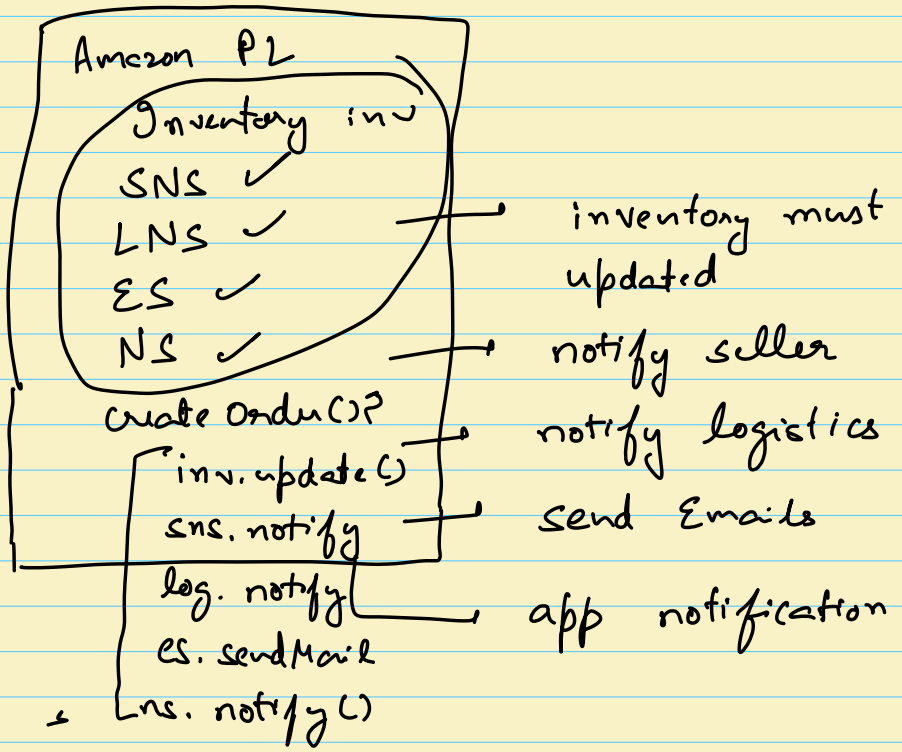


☆ Example of Adapter



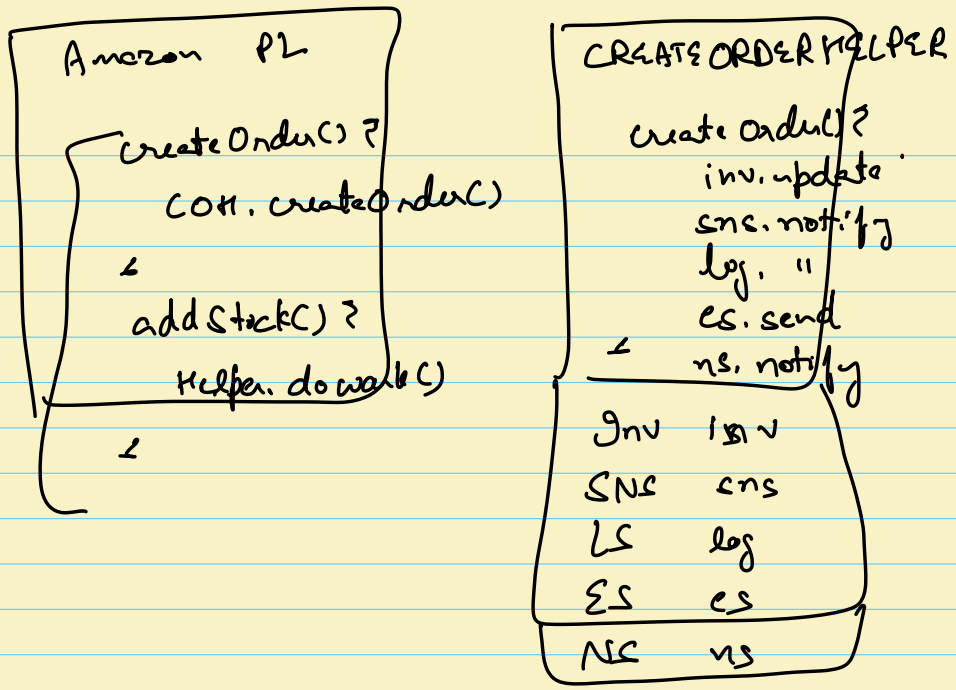

REST / SOAP → xml
.asmx
WCF
HTTP Protcols
Break = 10:33 to 10:43
1. Facade (10 minutes)
2. Revisit Factory
1. Code AF
2. Show Practical Factory.

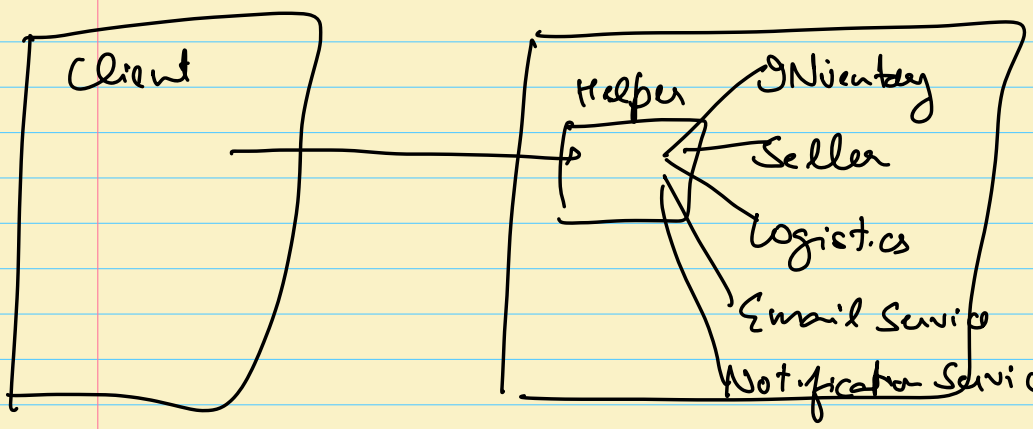
Facade Design Pattern



func() ?

}





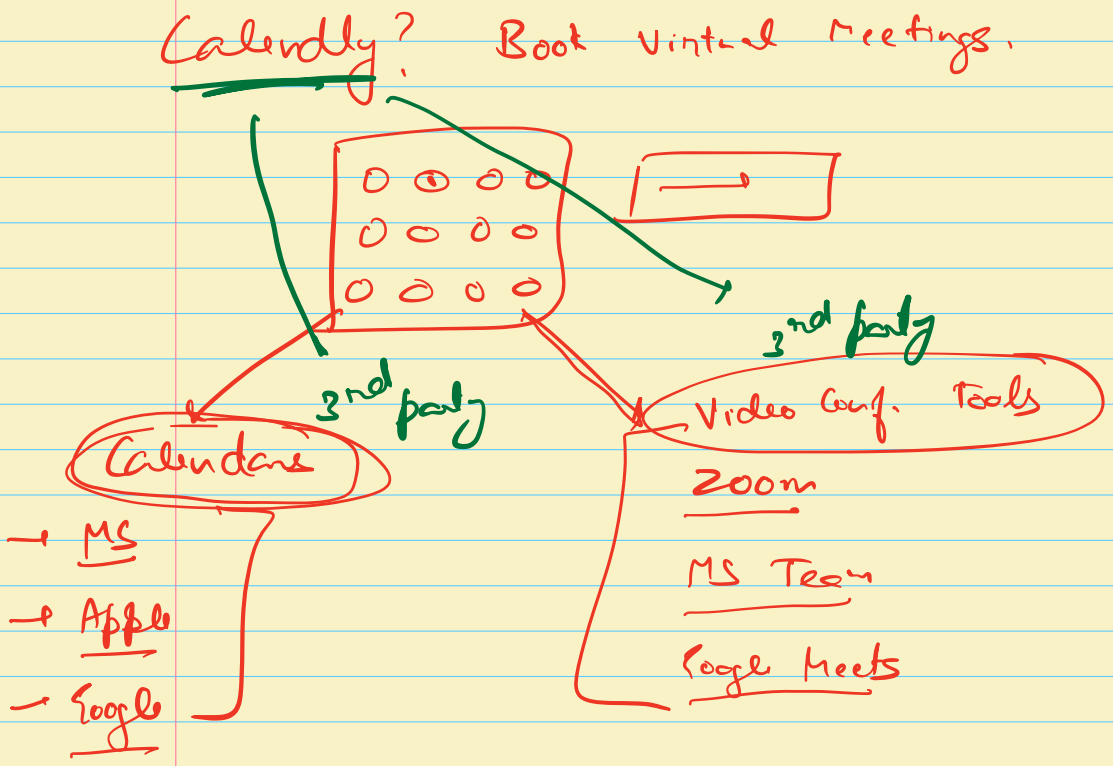
☆ Facade → Door

→ Provides a cleaner view (way to use) a complex system of many modules/services

→ A layer added btw client & complex system, to spare the client from complexity. Making calls convenient for the client.

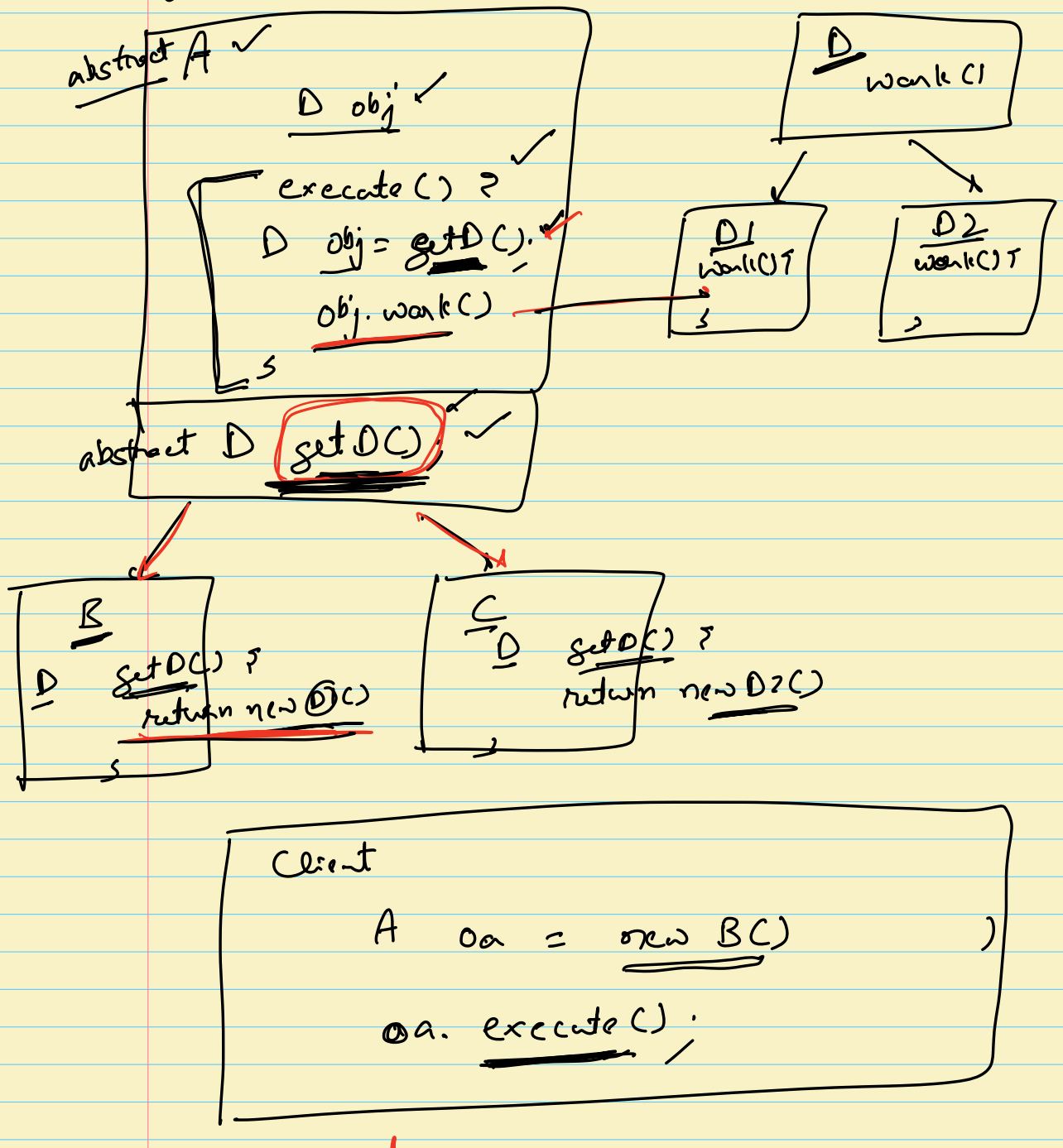
☆ More art, less science

☆ Production Level Code of a real adapter in a system.

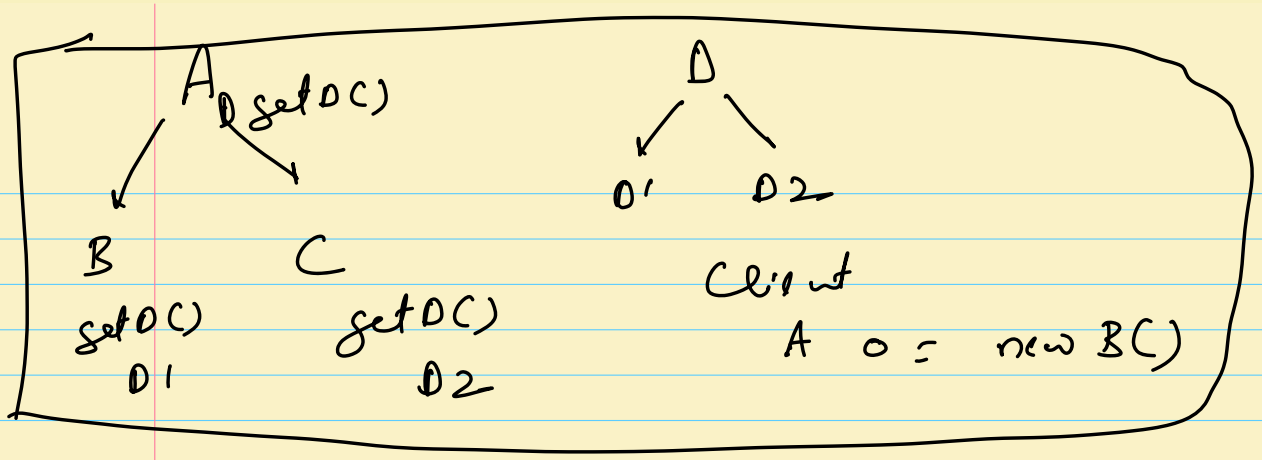


Coding Factory

Factory Method



Factory Method →



Abstract Factory

A

private Factory f;

execute() ?

D o = f.setDC()

o.work();

}

void setFactory(Factory f)?

this.f = f

D

work

D1

work()?

D2

work()?

Factory

D setDC()

F1

D setDC()?

return new D1()

F2

D setDC()?

return new D2()

Client

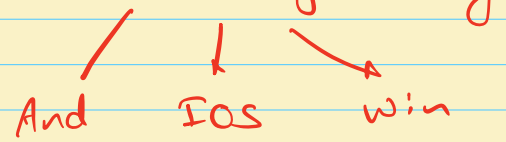
A o = new A();

o.setFactory(new F1());

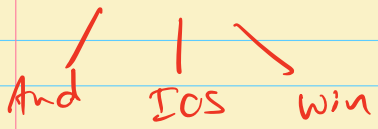
o.execute();

Homework

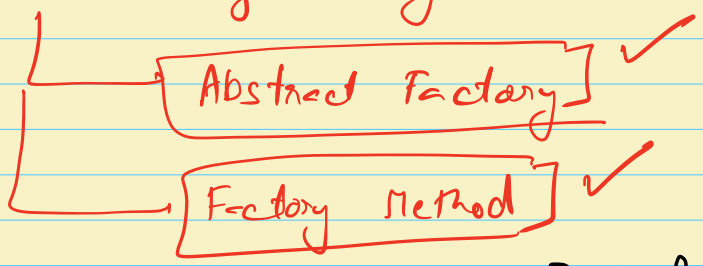
- The FlutterPage story



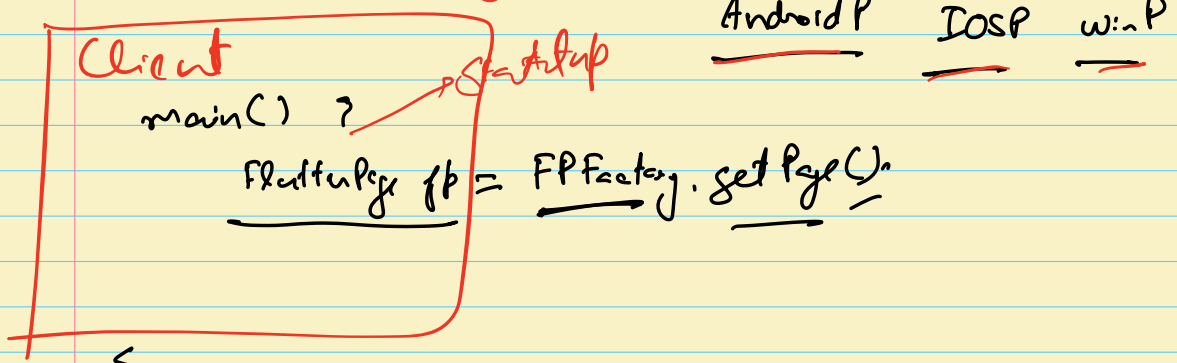
Button Menu Ddl



- Code FlutterPage story



Practical Factory



class FP Factory { → Registry

static FlatPage getPage() {

if (platform == "Android")
return new AndroidPage()
else if (platform == "IOS")
return new IOSPage()
else if ( ) {

}

}

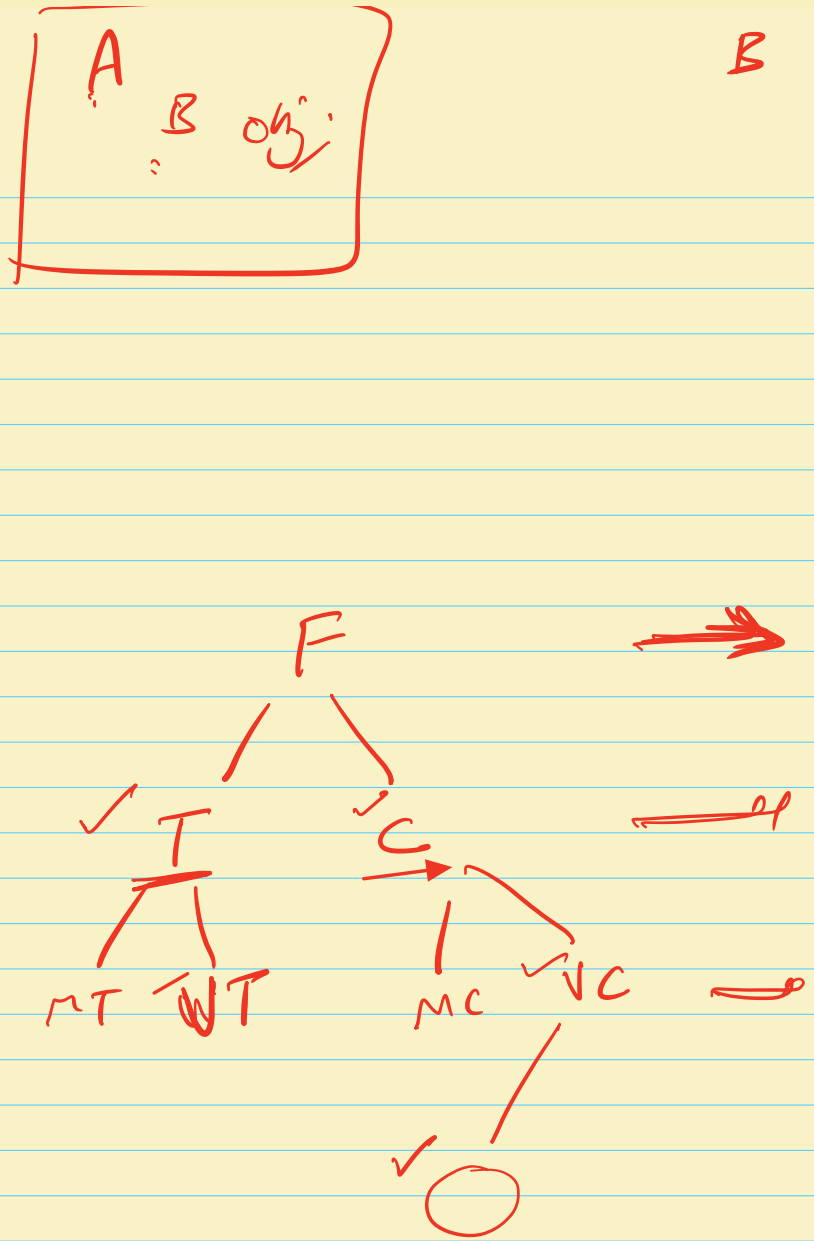
OCP violation?

SRP

FP getPage()
return map.get(platform)

Codebase

- Adapter
- Facade
- FM vs AF vs PF


A
B
B
F
T
C
MT
NT
MC
NC