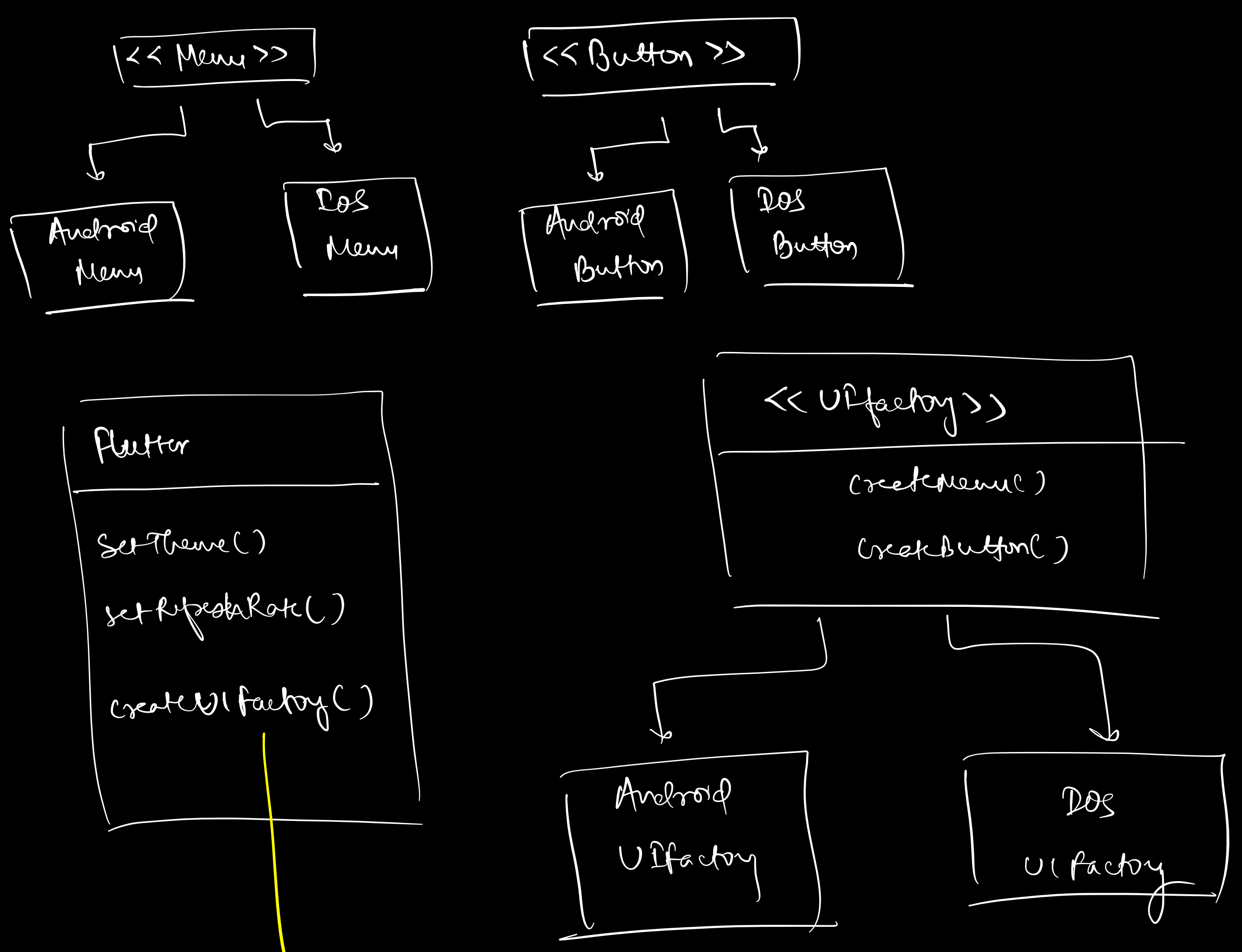
i) Practical Factory

ii) Adapter ←

iii) Facade ←

i) Practical Factory



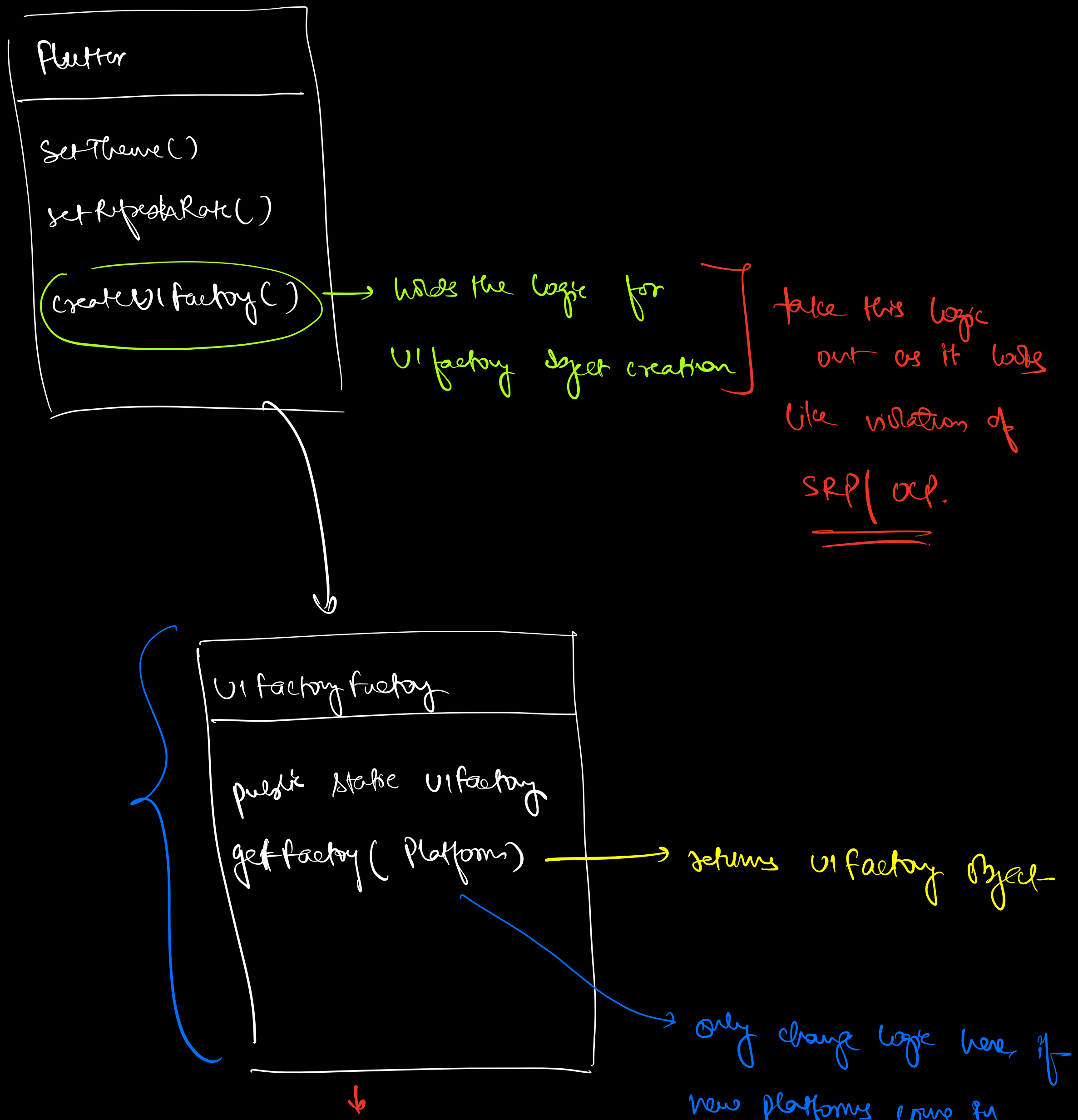
inside the logic for creating UI Factory depending on platform

lot of if/else

OCP violation? No

At times, we might have to write some code which might look like violation of SOLID principles but if they are part of the business logic / features of the application, then it wont be considered as violation.

Similarly in factory pattern, we are creating objects dependent on the user's choice/conditions, so we can't escape the use case of conditionals (either if/else or switch case) while implementing factory patter.


Flutter
setTheme()
setRefreshRate()
createUIFactory()
holds the logic for UI factory object creation
take this logic out as it looks like violation of SRP/OCP.
UIFactoryFactory
public static UIFactory getFactory(Platform)
returns UIFactory object
only change logic here if new platforms come in

* Most of the time, we will find this impl in production level appn

Creational DP → Singleton, Builder, Prototype | Registry, Factory | Abstract factory | Practical factory

: Structural Design Pattern :-

: How to structure your code base :-

i) what classes would be present

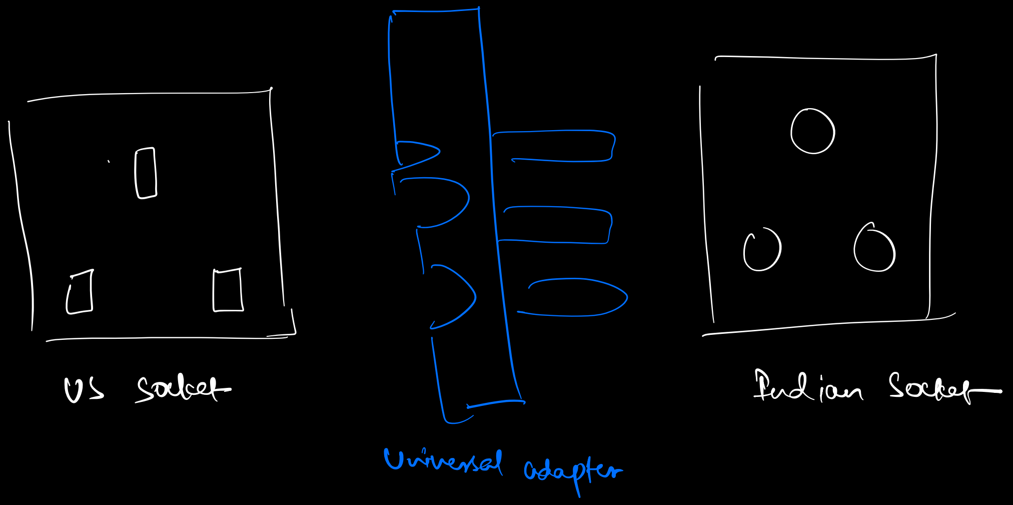
ii) attributes you need with every class

iii) how diff classes talk to each other

: ADAPTER DESIGN PATTERN :-

: Converter from one port to another → eg ⇒ Macbook Ports.

:



ADAPTER -> Intermediary layer that connects [transforms] one form to another form.

ex : HDMI -> USB C, or USB type A -> USB type C , US sockets to Indian sockets

- Assume :- apple wants to give all ports to users, for ex :- HDMI, type A, type C, magsafe, ethernet.

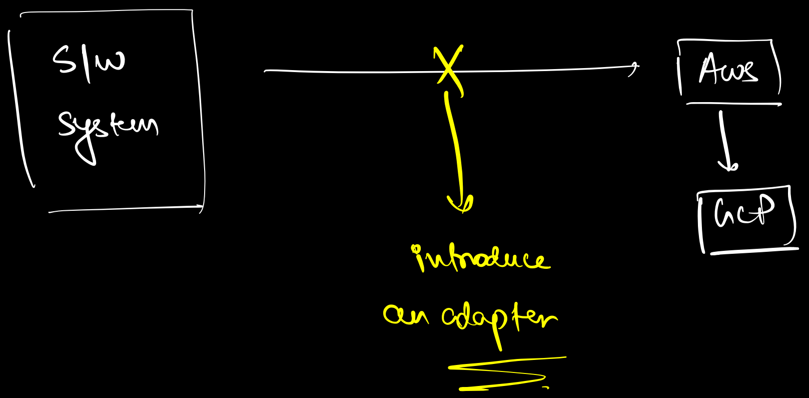
problems : i) need to provide hardware code to talk each port (OS needs connect with each port)

ii) complexity in hardware level

iii) increase size, weight and price

So, Apple developers choose to move forward with only USB C for most of the laptops, or USB C with HDMI for pro models, so their is a minimal need of code change/addition, and minimal effort in terms of design.

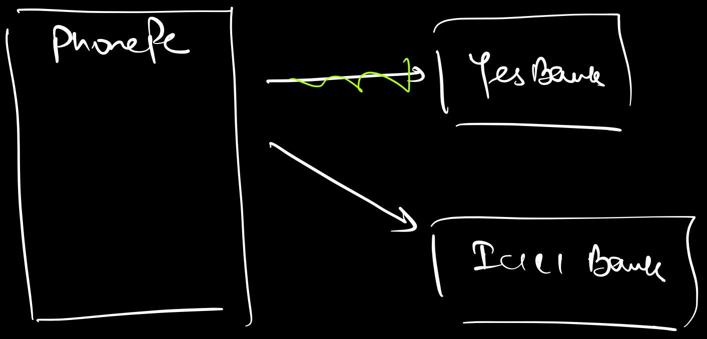
The responsibility of attaching other equipments to MacBook, like printer, Ethernet, musical devices etc. lies with the peripherals ie. Via using Adapters.



Problem Statement :-

We might want to change a 3rd party service provider in future. Examples -> Cloud platforms, Maps, SMS service providers, Payments, Billing, KYC etc.

Due to several reasons like pricing, performance, availabilility or maintanence of these 3rd party service we might need to switch between them.

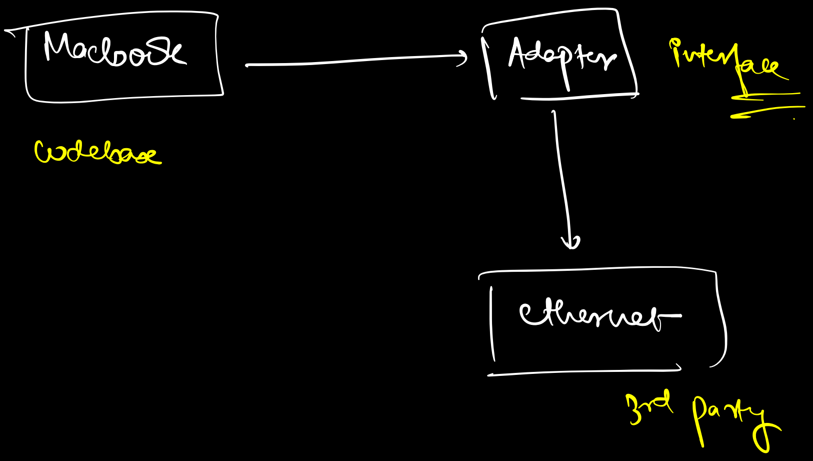


Adapter design pattern ensures that our codebase remains maintainable and relatably independent of our 3rd party dependencies, so that we can change/replace these dependencies easily whenever required.

3rd party dependencies -> APIs/SDKs/Libraries.

If out code base is directly talking to 3rd party dependencies, it involves a lot of tight coupling b/w our code base and the dependency, this can highly impact the maintainability.

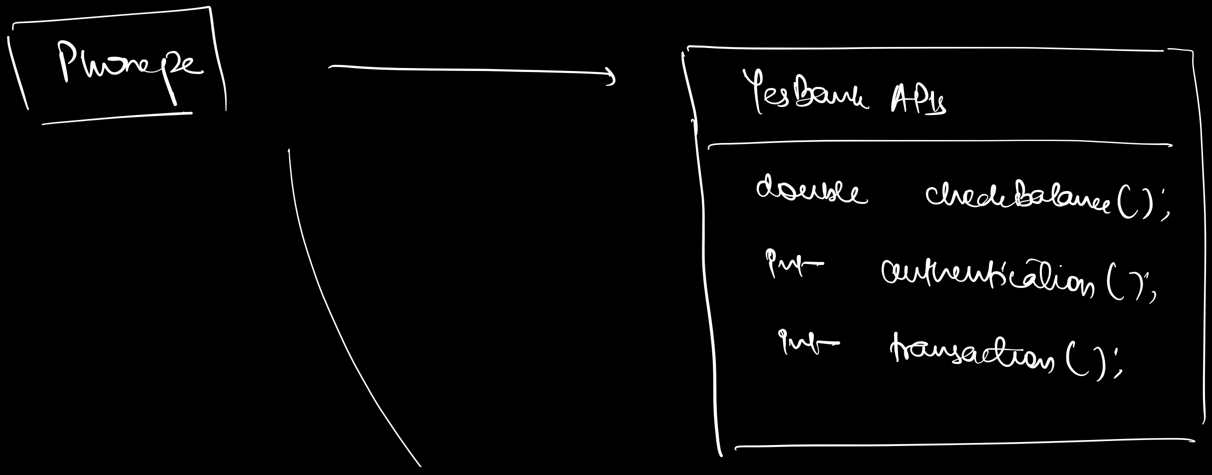
As a rule, always make sure whenever we are introducing a 3rd party dependency in our code base, we never connect to it directly, instead we use an interface in between, this interface acts as an adapter.

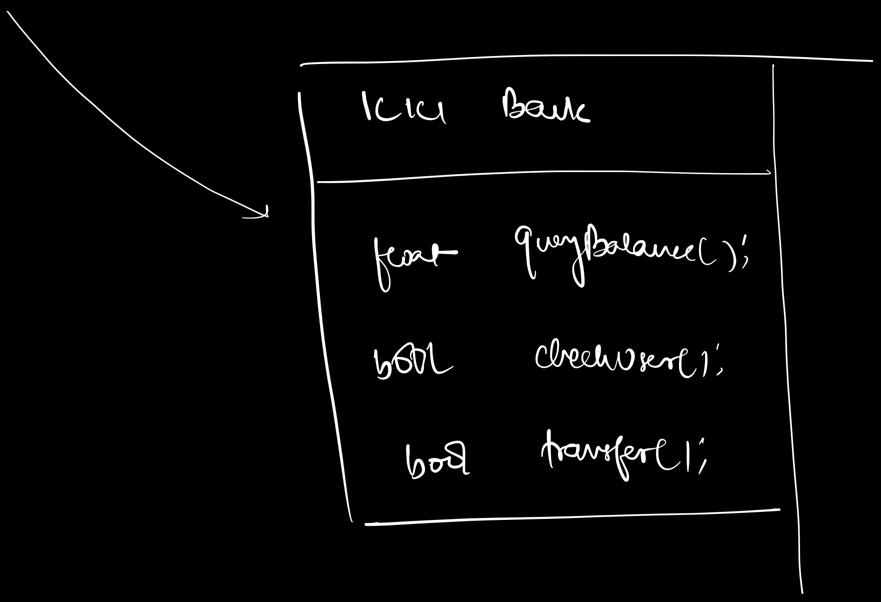


: How to use adapter pattern

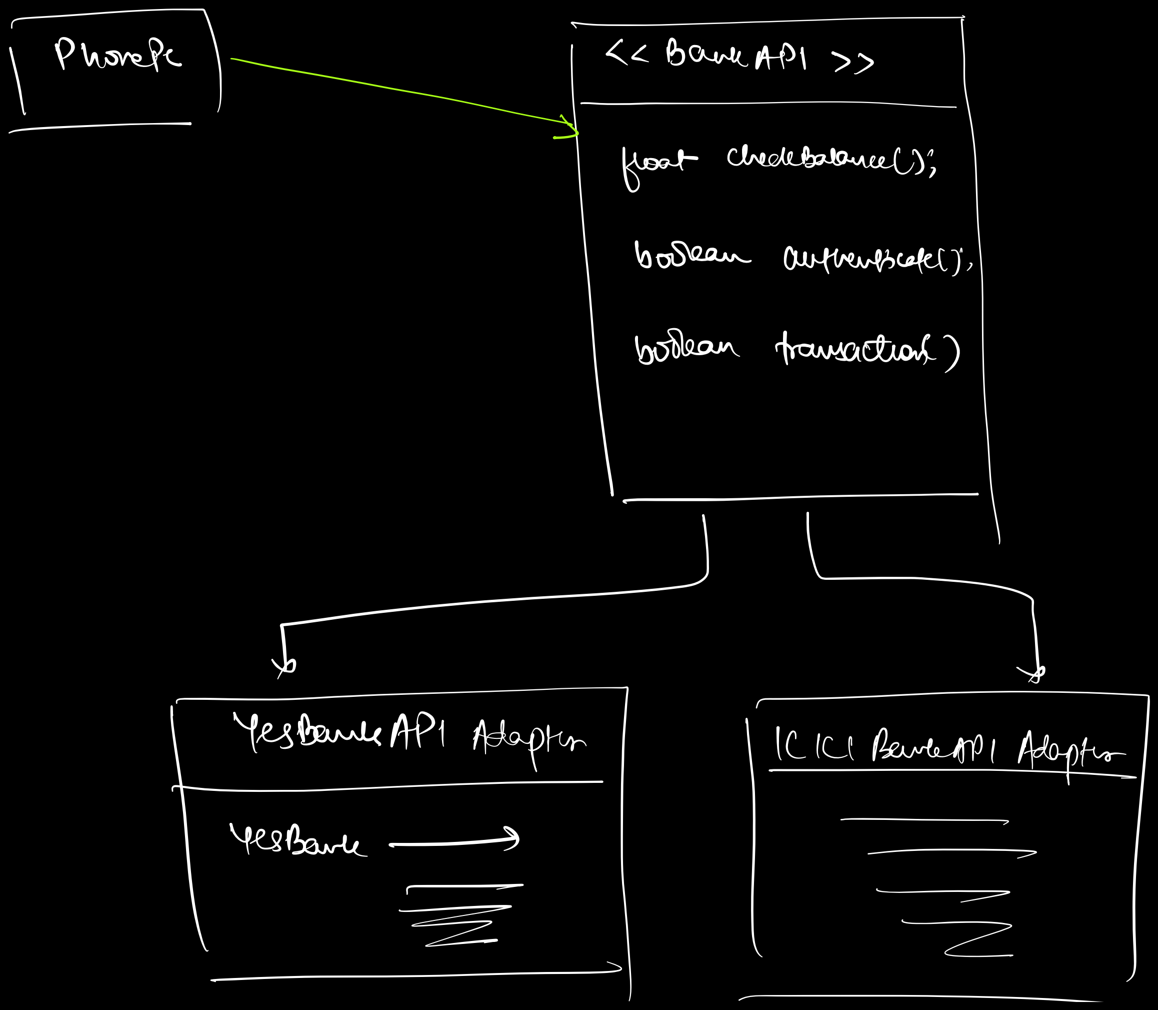
1) Create an interface to connect to a 3rd party dependency.

ii) Codebase & business logic should depend on the interface and not on the 3rd party directly





As the 3rd party dependencies, will not implement our interfaces ☹, we should create a wrapper class that implements the interface using the 3rd party dependencies and we dependent on the interface.



When to use adapter ⇒ when we have any 3rd party dependency

: Case study:

Calendly allows us to schedule meetings

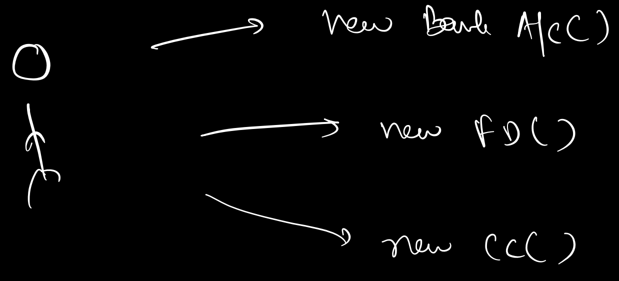
- 3rd parties
  - Gmeet
  - Teams
  - Zoom
  - etc

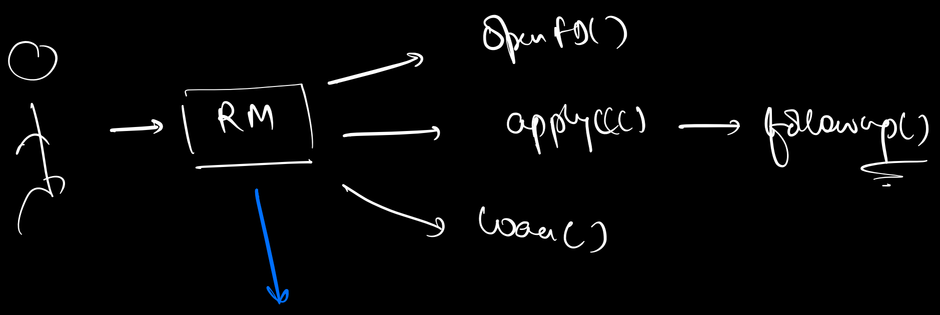
Cal.com ⇒ Open source ⇒ uses adapter for 3rd party

: Facade Design Pattern :

↳ boundary | outside view

to provide a cleaner view of a complex environment





Acts as a facade to simplify your banking exp., makes it easy for you in complex structure

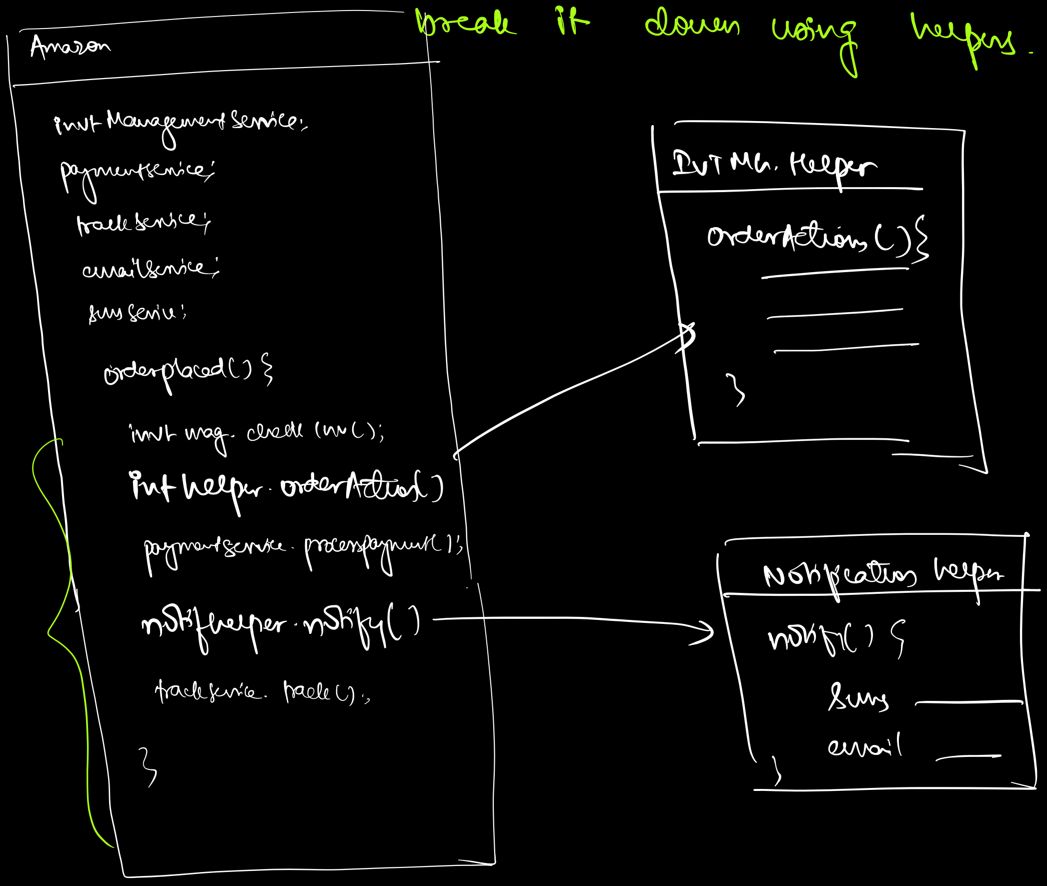
```
Amazon
------------------------------
invt ManagementService;
paymentService;
trackService;
emailService;
sms Service;

orderplaced() {
    invt mag. checkInv();
    invt mag. update inv();
    invtmag. checkLocationforproduct();
    invtmag. notifywarehouse();
    paymentService. processpayment();
    smsService. sendNotif();
    emailService. sendNotif();
    trackService. track();
}
```

=) Currently, all the work that needs to be when an order is placed, is done via a single method inside a single class.

=) This makes it very complex → introduce a facade

Facade pattern: when something becomes too complex, break it down using helpers.



```
Amazon

int ManagementService;
paymentservice;
tradeservice;
emailservice;
sms service;

orderplaced() {
    invt mng. check Inv();
    Int helper. orderAction()
    paymentservice. processpayment();
    notifhelper. notify()
    tradeservice. trade();
}
```

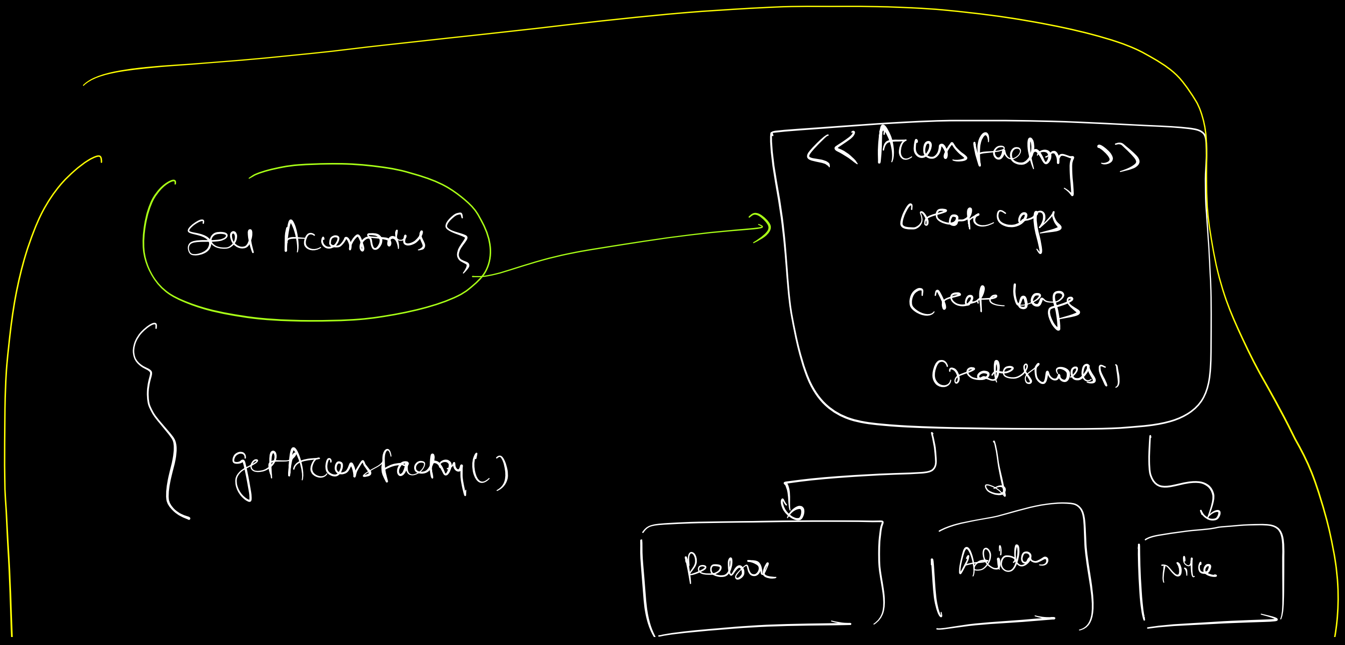
```
InvT Mn. Helper

orderActions(){

}
```

```
Notification helper

notify() {
    sms
    email
}
```



```
Sell Accessories {

getAccessfactory()
```

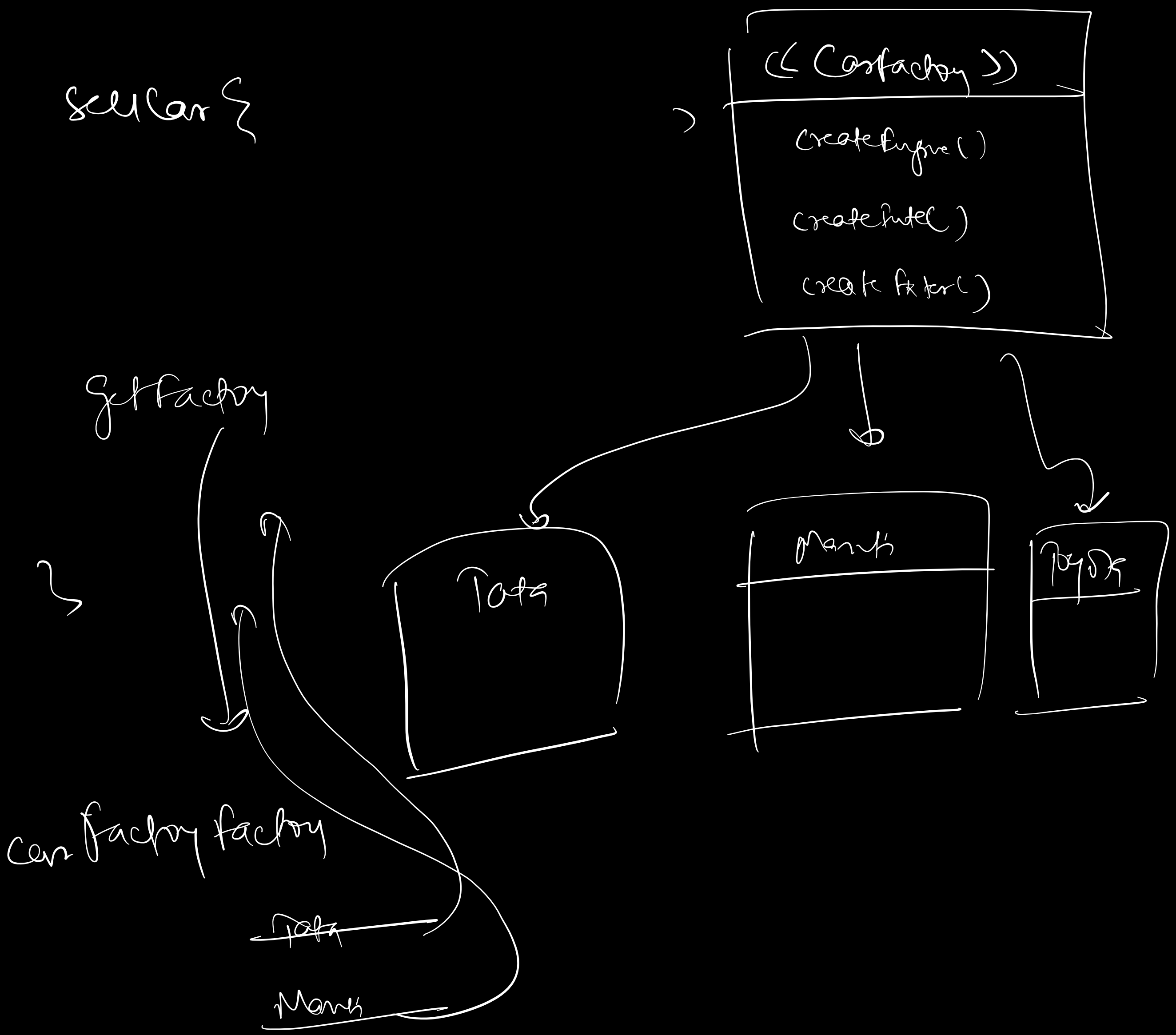
Access factory factory {

____

____

____

}


sellCar {
getFactory
}
<< CarFactory >>
createEngine()
createWheel()
createFilter()
Tata
Maruti
Toyota
car factory factory
~~Tata~~
Maruti