→ Structural Design Patterns.

- → Adapter.
- → Facade.

Structural Design Patterns.

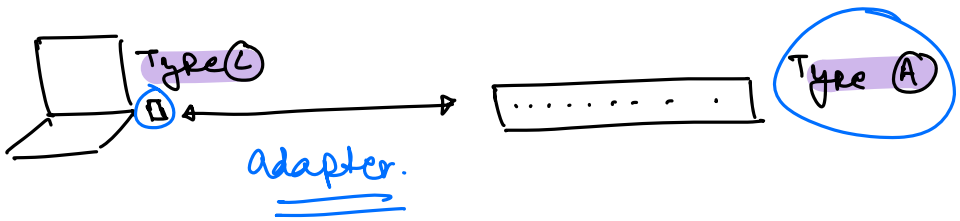
⇒ How to structure your codebase.

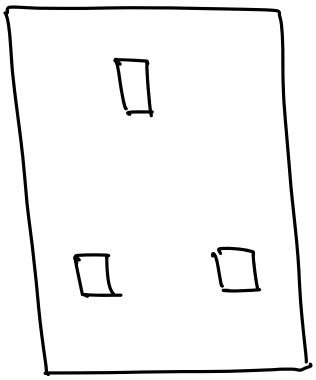
- → What all the classes you should have in the codebase.
- → How different classes are going to talk to each other.

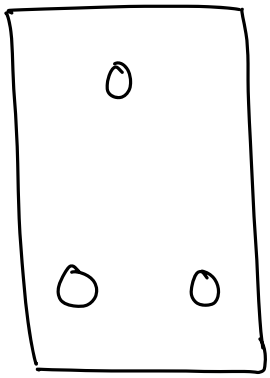
ADAPTER DESIGN PATTERN.

Power adapter.

Type C adapter for Macbook.







⇒ Adapter.

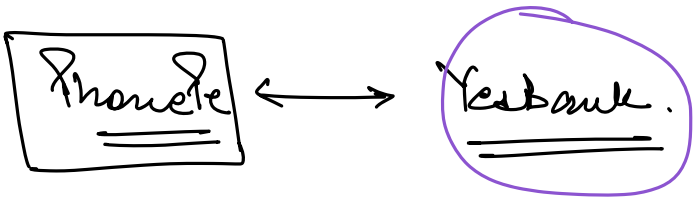
Intermediary layer that connects / transforms one form into another form.

HDMI ⟷ Type C

⇒ Apple developers need to talk to only one type of input i.e type C. They don't have to make their HW / SW compatible to all the different type of inputs, this responsibility conversion of one type into another type is given to Adapter.

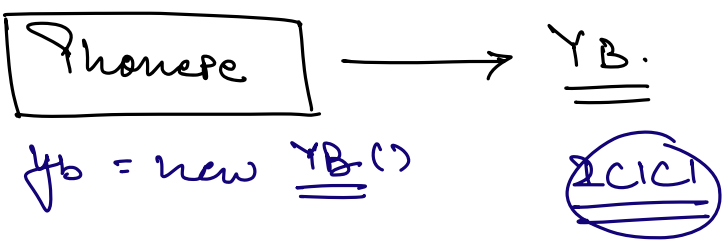
⇒ AWS / Cloud service.

⇒ Payment GW.



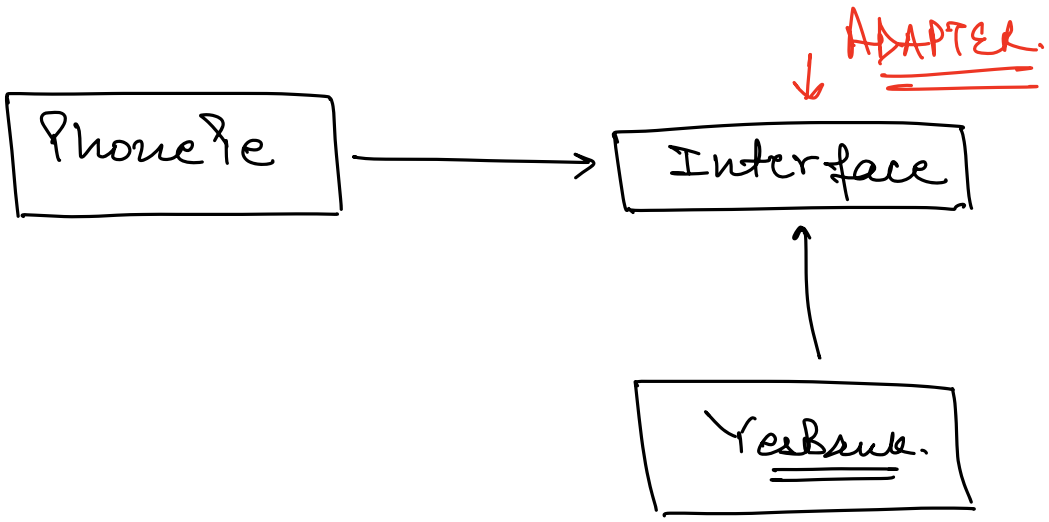
Problem Statement:

Our system should remain maintainable while talking to 3rd party API's.



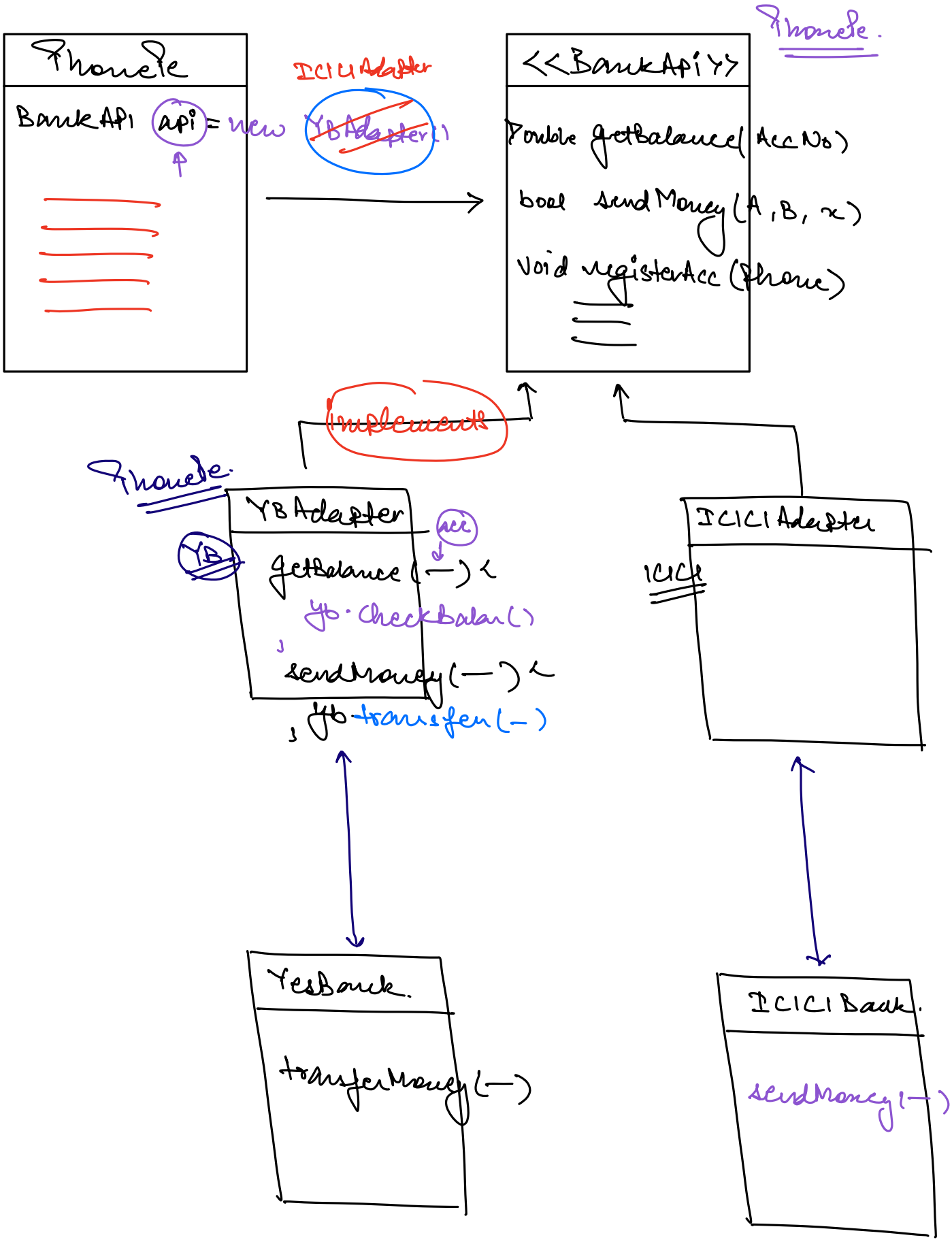
⇒ If your codebase is directly talking to 3rd party API's, it involves tight coupling b/w your codebase & 3rd party system. This affects maintainability of your codebase.

⇒ Whenever you are talking to a 3rd party API, you should talk to them via an interface.

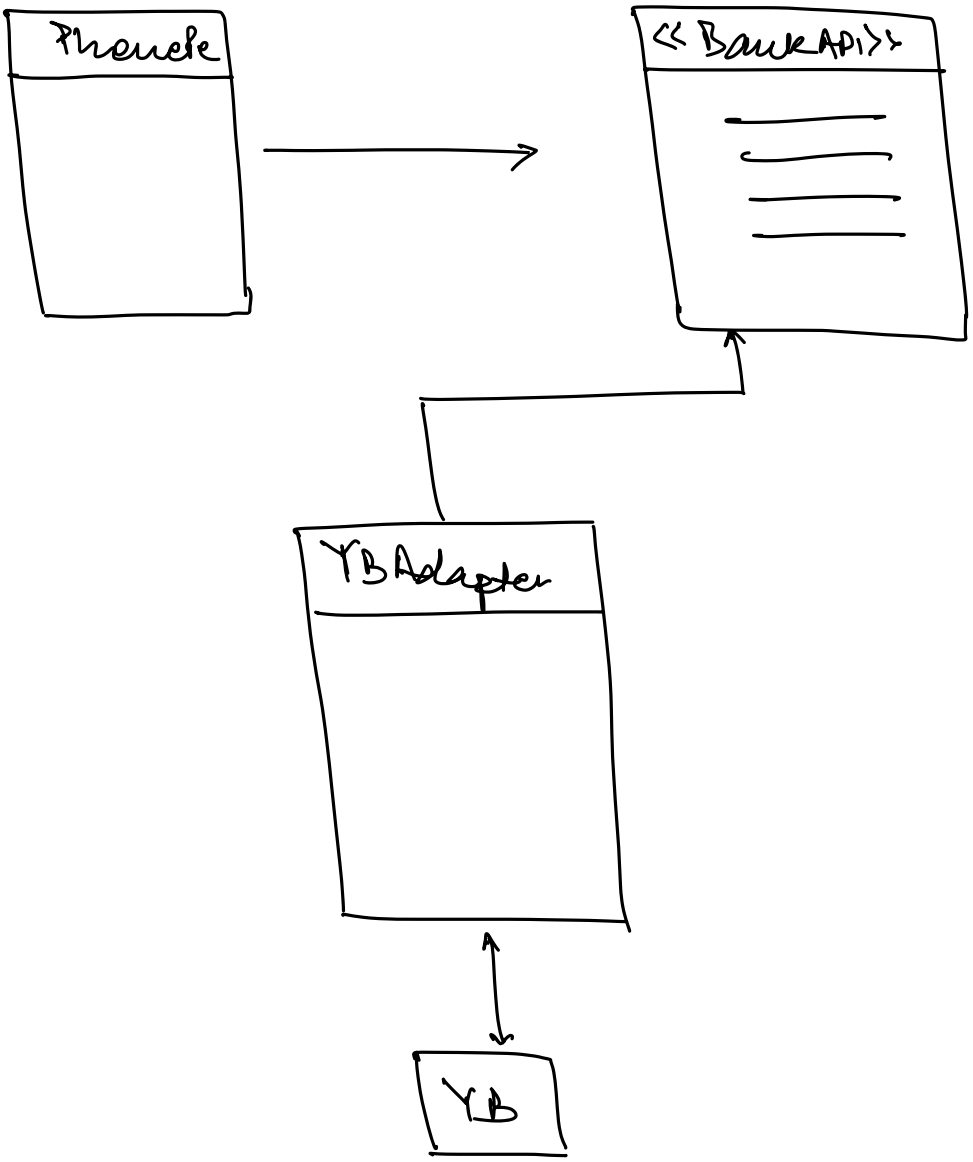


⇒ How to use Adapter Pattern.

① Whenever we need to talk to a 3rd party system, create an interface b/w our codebase & 3rd party.


Phonepe
BankApi api = new YBAdapter()
ICICIAdapter
Phonepe.
<<BankApiYY>>
Double getBalance(AccNo)
bool sendMoney(A, B, x)
Void registerAcc(Phone)
implement
Phonepe.
YBAdapter
YB
acc
getBalance(—){
Yb.checkBalan()
sendMoney(—){
Yb.transfer(—)
ICICIAdapter
ICICI
YesBank.
transferMoney(—)
ICICIBank.
sendMoney(—)

(2) As 3rd party will not implement this interface, create an Adapter class to implement the interface that uses the 3rd party API's behind the scenes.



FACADE DESIGN. PATTERN.

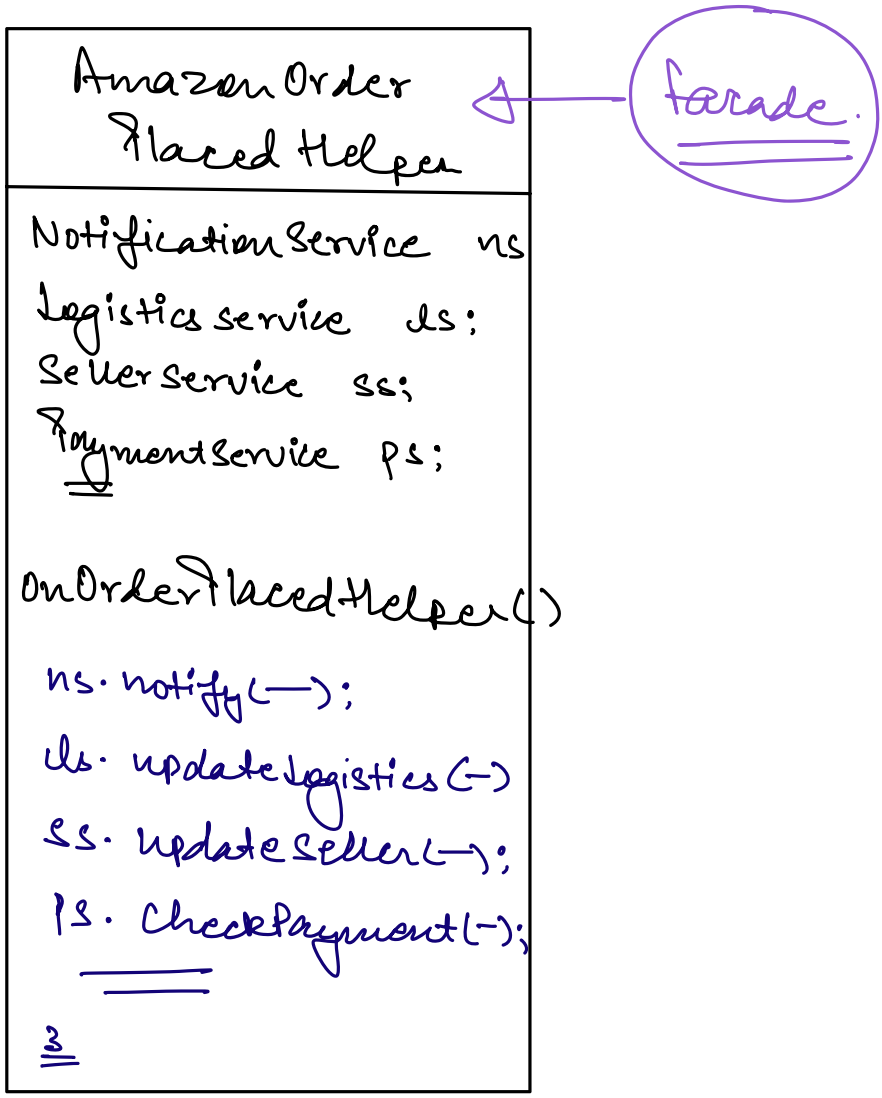
↓

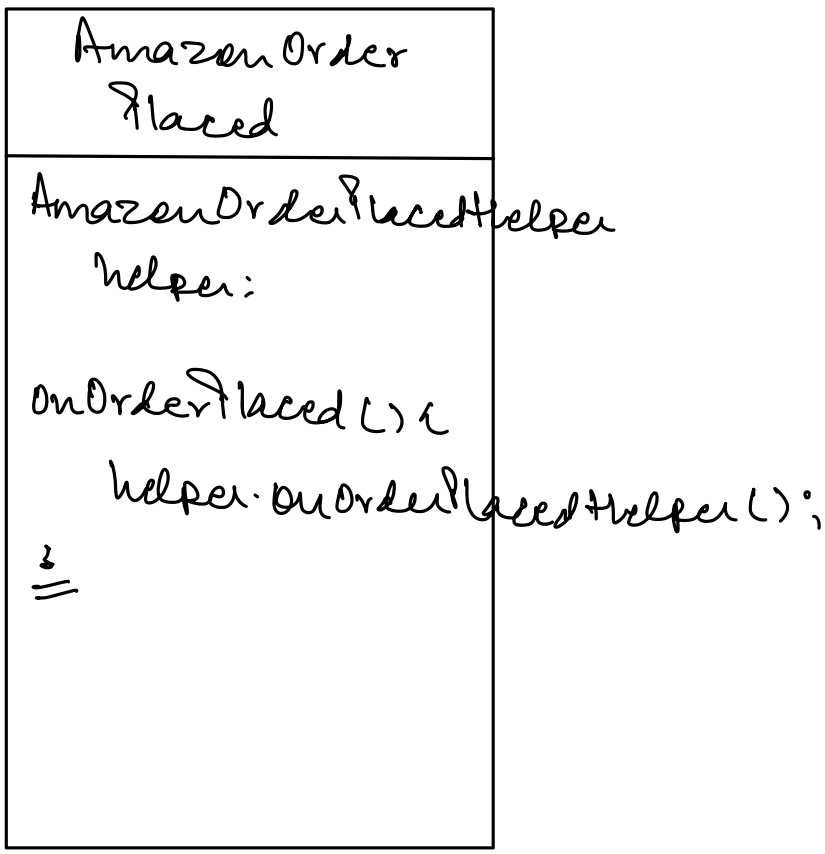
front | Boundary of building

⇒ Provides easy to understand representation to a complex environment.

AmazonOrderPlaced

NotificationService ns;
LogisticsService ls;
SellerService ss;
PaymentService ps;
 	
OnOrderPlaced() {
ns.notify(—);
ls.updateLogistics(—)
ss.updateSeller(—);
ps.CheckPayment(—);
}
 	
}

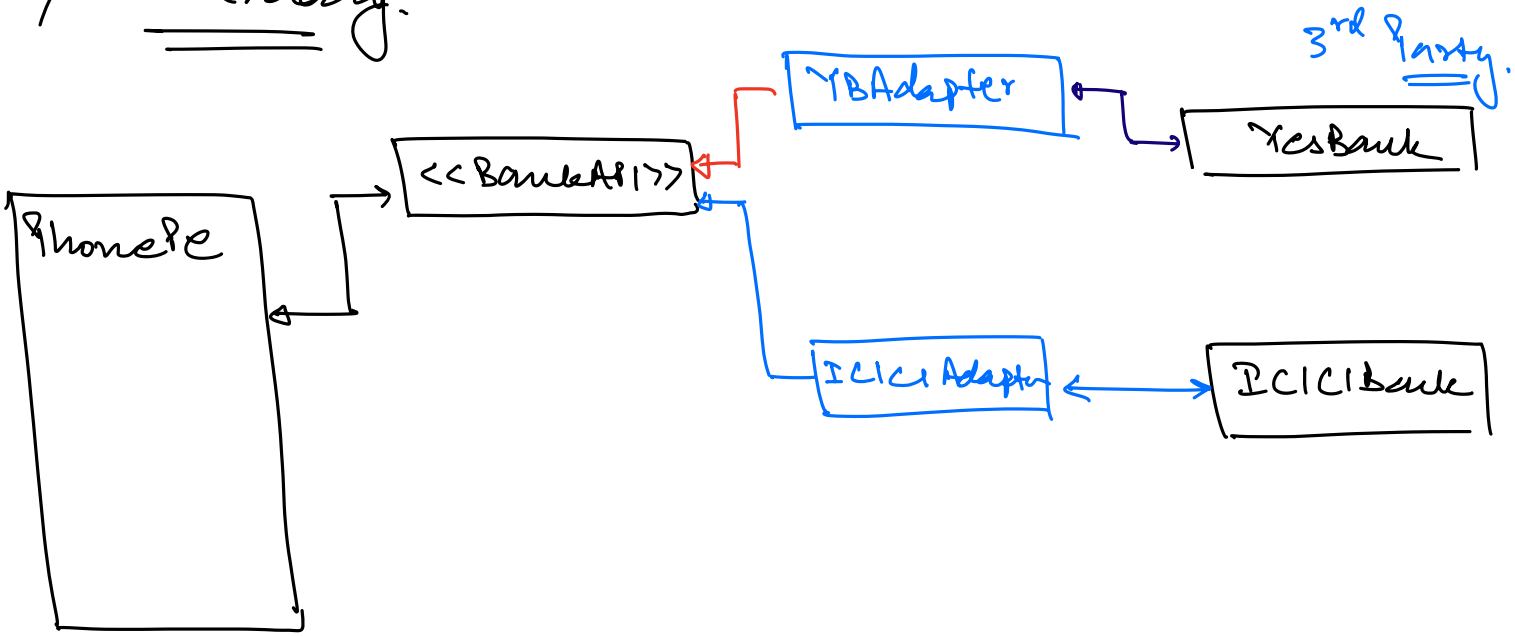

AmazonOrder PlacedHelper
NotificationService ns
LogisticsService ls;
SellerService ss;
PaymentService ps;
OnOrderPlacedHelper()
ns.notify(—);
ls.updateLogistics(—)
ss.updateSeller(—);
ps.CheckPayment(—);
}
facade.


AmazonOrder Placed
AmazonOrderPlacedHelper helper;
OnOrderPlaced(){
helper.OnOrderPlacedHelper();
}

FACADE.

→ Whenever you find a method / class that is doing too much of work, instead of doing that work within the main class, create a **Helper / facade** to do that work.

⇒ Summary.



facade.

⇒ Creating helper / facade classes to make our code look cleaner.
