Agenda

1. Liskov's Substitution Principle
2. Interface Segregation Principle
3. Dependency Inversion
   - → Dependency Injection
   - → Inversion of Control
4. Finish Design a Bird

```
class Bird {
    color
```

age
species

fly()
eat()
walk()

→ lot of if else in fly method

SRP and O/C

abstract ~~class~~ Bird

color
age
fly()
eat()
walk()

abstract

→ no implementation in the superclass

Hen
fly() {
}

Pigeon
fly() {
}

Peregrin f
fly() {
}

Sparrow
fly() {
}

Bird b = new Bird()

You can't create an object of
a abstract class

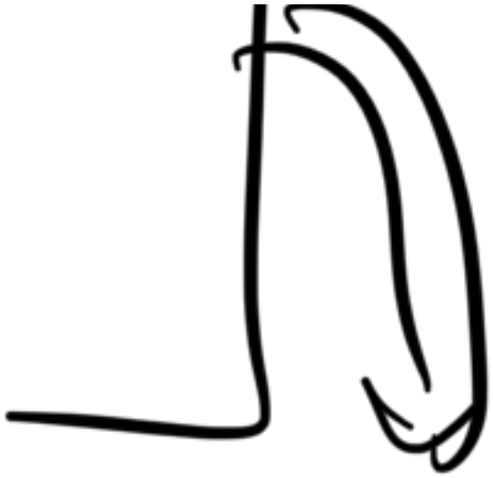
There were some birds that couldn't fly.

(1) Have an empty fly method

② Throw an exception

③ Returns a special value

④ Default implementation



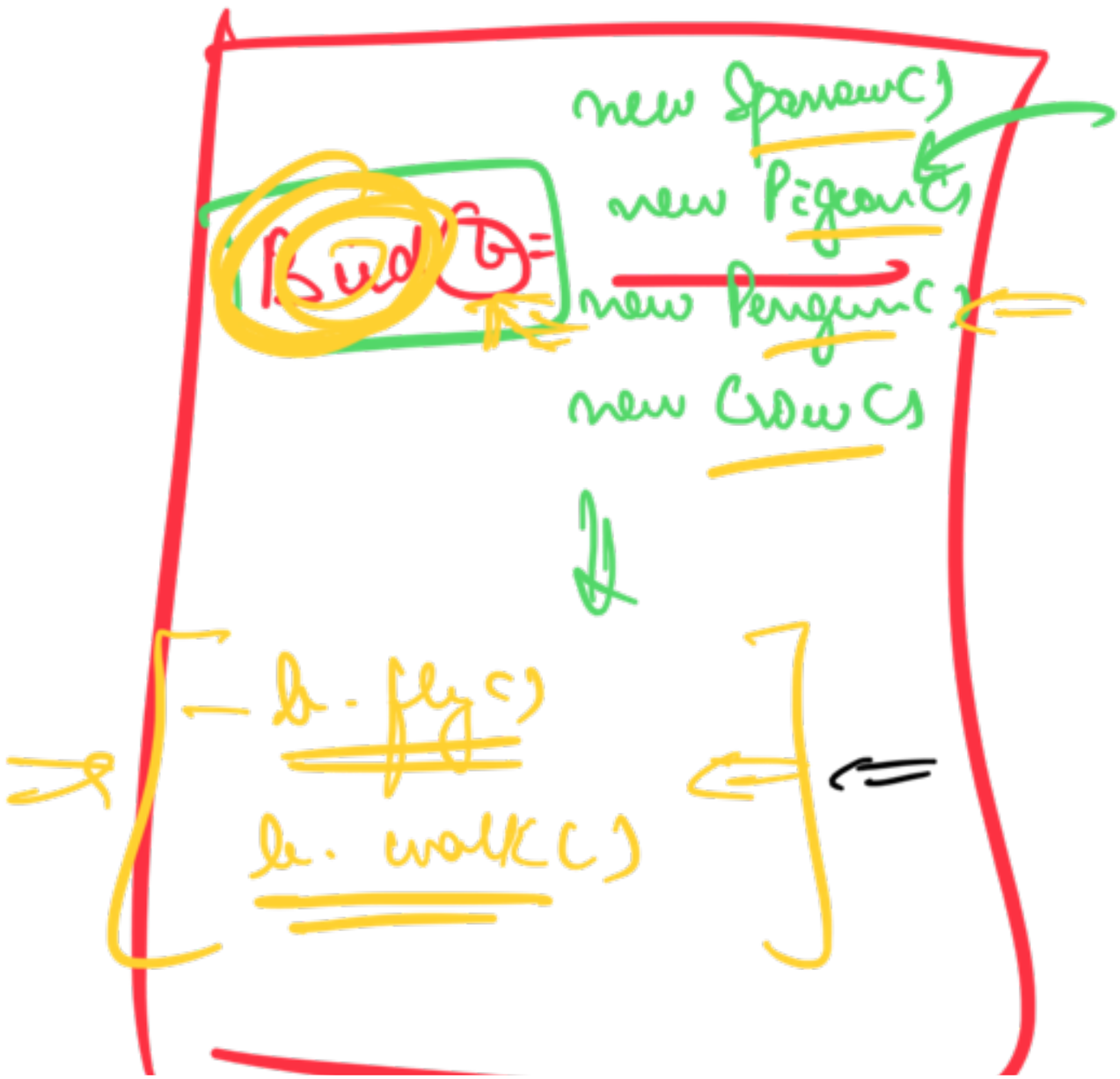
Client had to have if-else to check for a particular child's Special case.

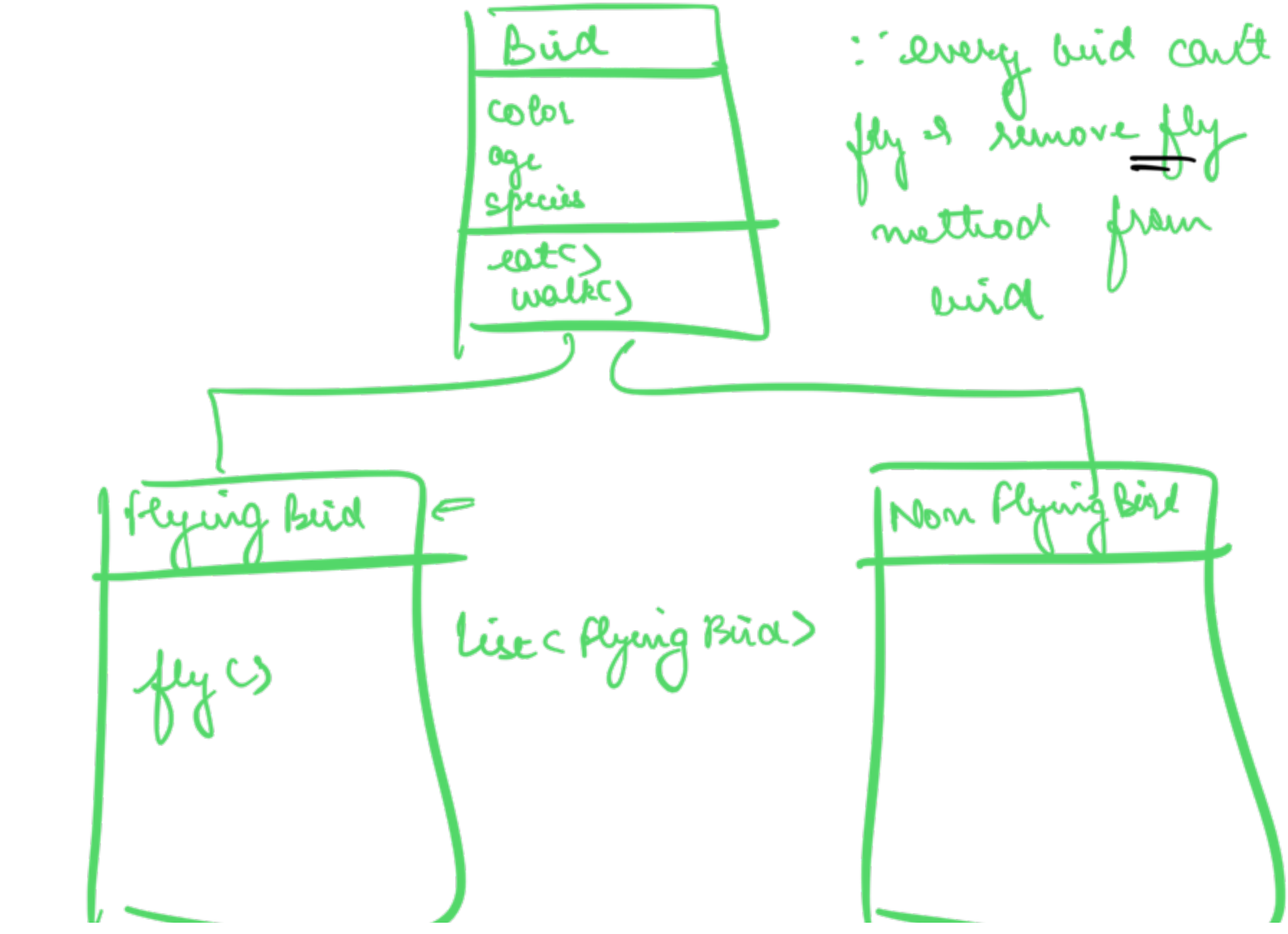
Liskov's Substitution Principle

In a variable of a parent class, I should be able to substitute an instance of any child class without the client having required to handle a special

client using Adapter

case (without requiring any change in the following code base)




Bird
color
age
species
eat()
walk()
: every bird can't fly & remove fly method from bird
Flying Bird
fly()
List<FlyingBird>
Non Flying Bird

Hen

Pigeon

Crow

Penguin

Some Birds can fly

Some Birds can eat

Bird

Flyind And Eating Beid

Flyind No Eating Beia

Not Flying Eating Beid

NoC Flying NoE Eatin Bird

4 such fly

eat

walk

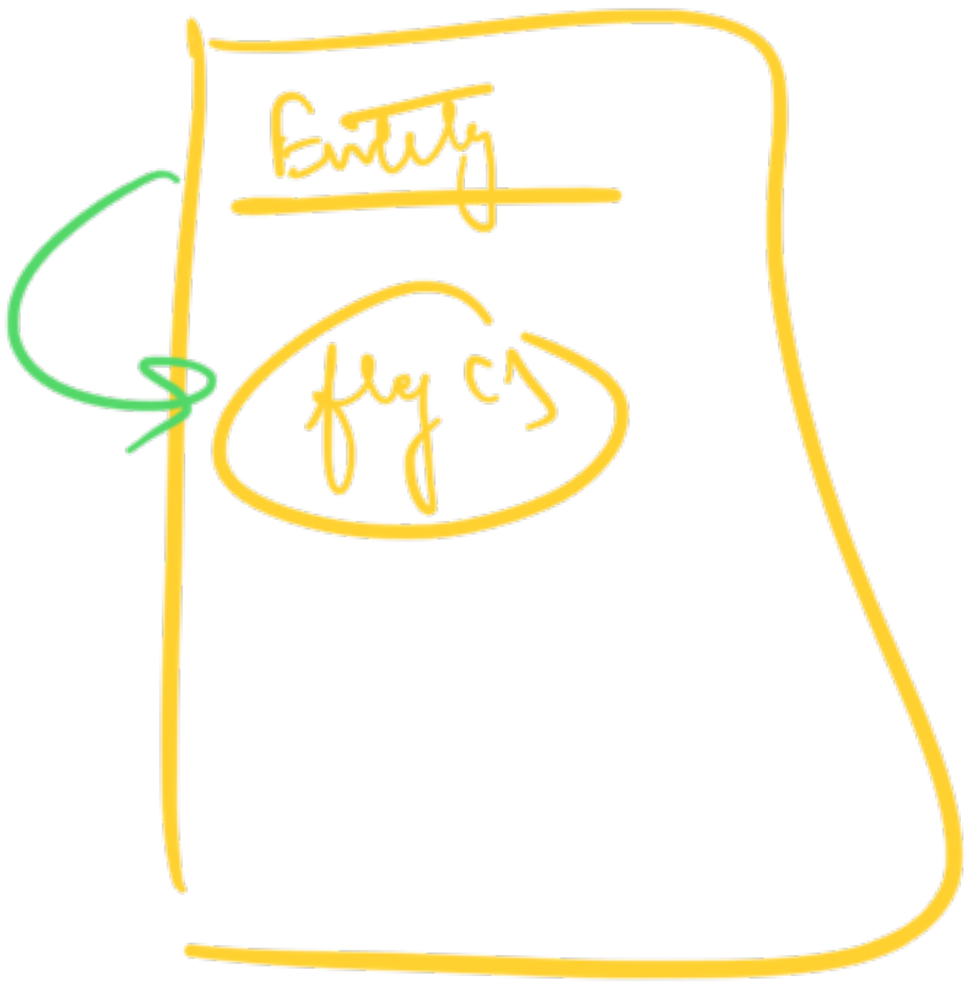
drink

$\rightarrow 2^4 \Rightarrow 16$

$10 \Rightarrow 2^{10} = 1024$

```
class Penguin extends Not_Flying_Bird {

}
```



Problem : If I want to get a list of all the bird who can fly?

List < Integer >

List < Bird >

List< >

flying Race (List<? > flies) {

}

Observations

① If a class supports a particular behaviour, then only it should have that method.

fly() => can fly

② I might have a need to get all the objects that demonstrate a particular behaviour.

=> flyingRace(List<? > fliers)

for (Flier? f : fliers)

f.fly()

3

Somehow need a way to represent behaviours

interfaces

abstract

Bird

color

share

eat

abstract

walk()

<<interface>>

Flier

fly()

<<interface>>

Eater

eat()

a class should have only those methods that are obvious behaviours of that entity. (every type of that entity will have that behaviour)

→ Use inheritance for entities Not for behaviour

DON'T USE INHERITANCE TO SIGNIFY BEHAVIOURAL DIFF

↓

Use interfaces instead

```
Class Sparrow extends Bird
```

implements Flies, Eatu {

flyO{

}

eatO {

}

}

Client {

Name of Interface

List<Flies> flies = {

new Sparrow(),

new Crow()

new Heart

class LEDSensor implements Waterproof {

}

Foot to fly

(1) is the method that requires these behaviours violating SRP

Eater e = new Sparrow()

Flier f = new Sparrow()

Eater Flier ef = new Sparrow();

SomeMethod(object)

object.fly()

object.eat()

I: Interface Segregation Principle

Every interface should be as light as possible.

→ It should have as less methods as possible

→ Don't make your interfaces thick.

> Every interface should have a

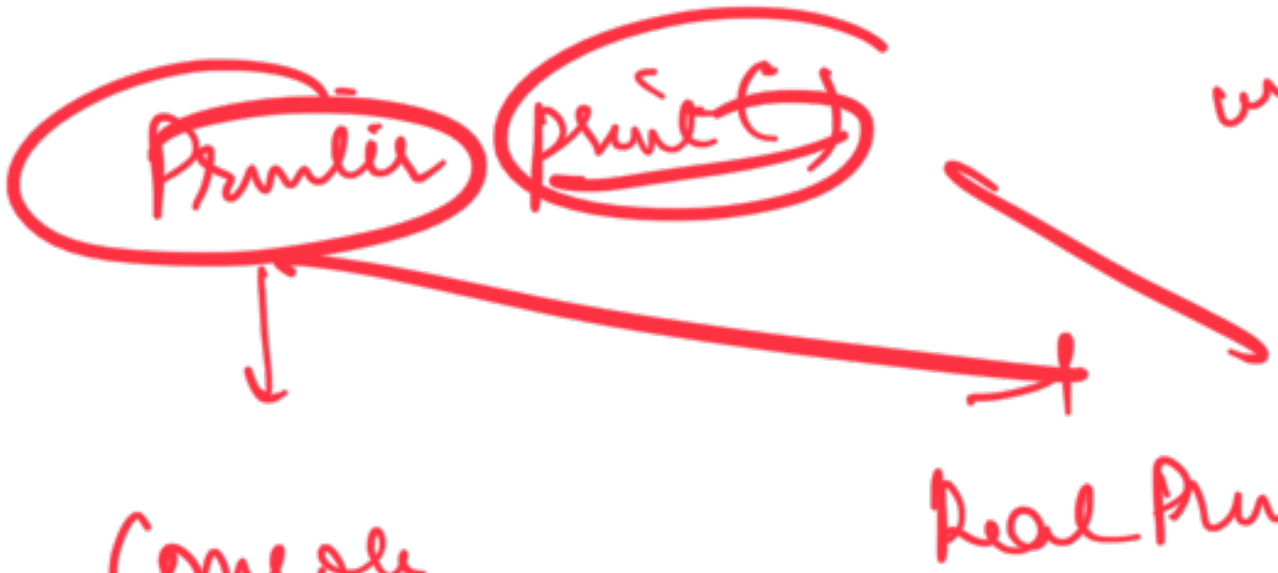
Single Responsibility

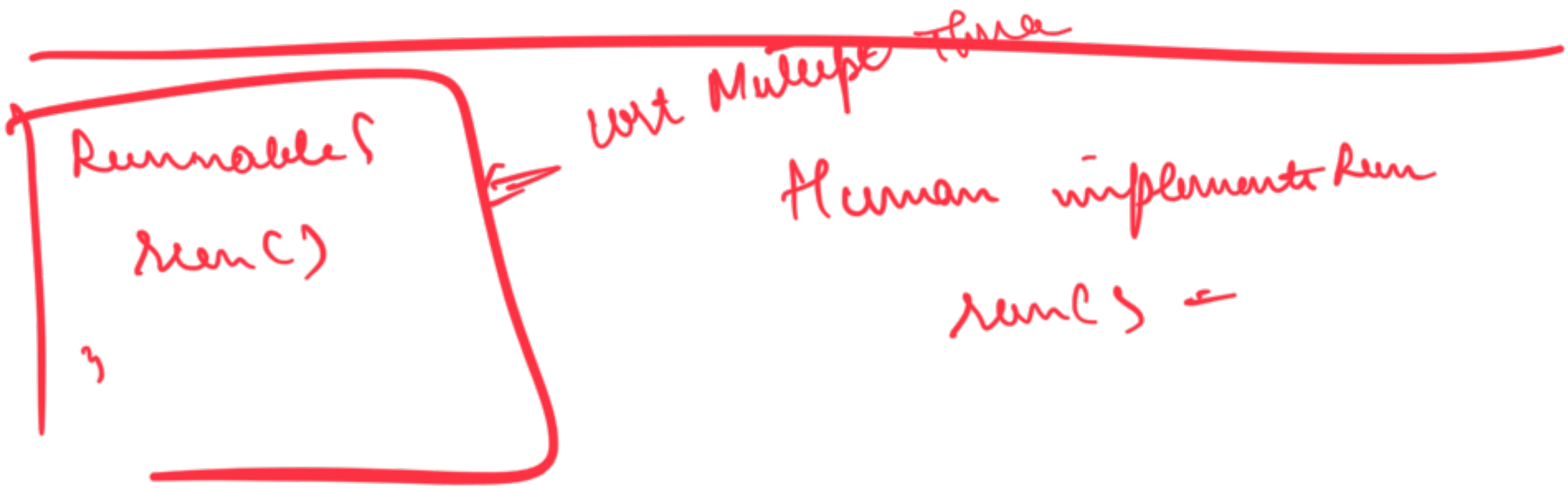
Lambda Functions

↳ Interfaces with only one method

List → 8 Methods

Only have interface with more than 1 method if the methods are logically linked





D: Dependency Inversion Principle =

Abstract de

No 2 concrete ~~class~~ should be directly connected to each other.

They should ~~always~~ depend on each other via an interface / abstract class

---

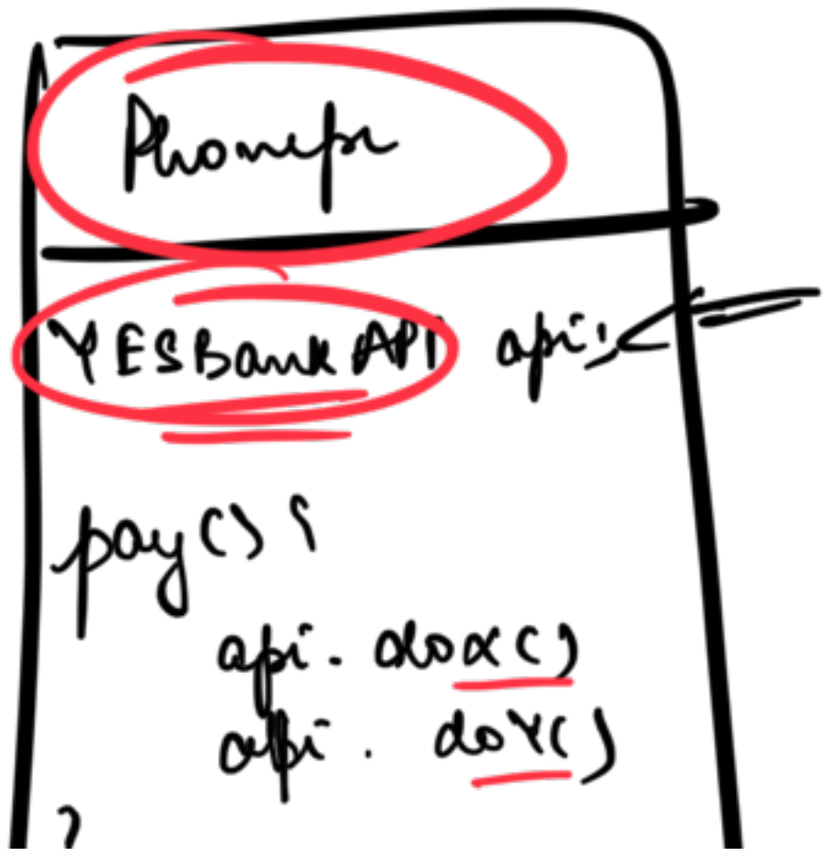
Never code to implementation (concrete class)

Always code to interface

Animal

Flier

Eater



refund() {
  api.doIt()
}

PhonePe

BankAPI api;
payU() {
  api.AC()
  api.BC()
}
refund() {
  api.CC()
}

YesBankAPI implements BankAPI

ICICIBankAPI implements BankAPI

Concrete class

PhonePe

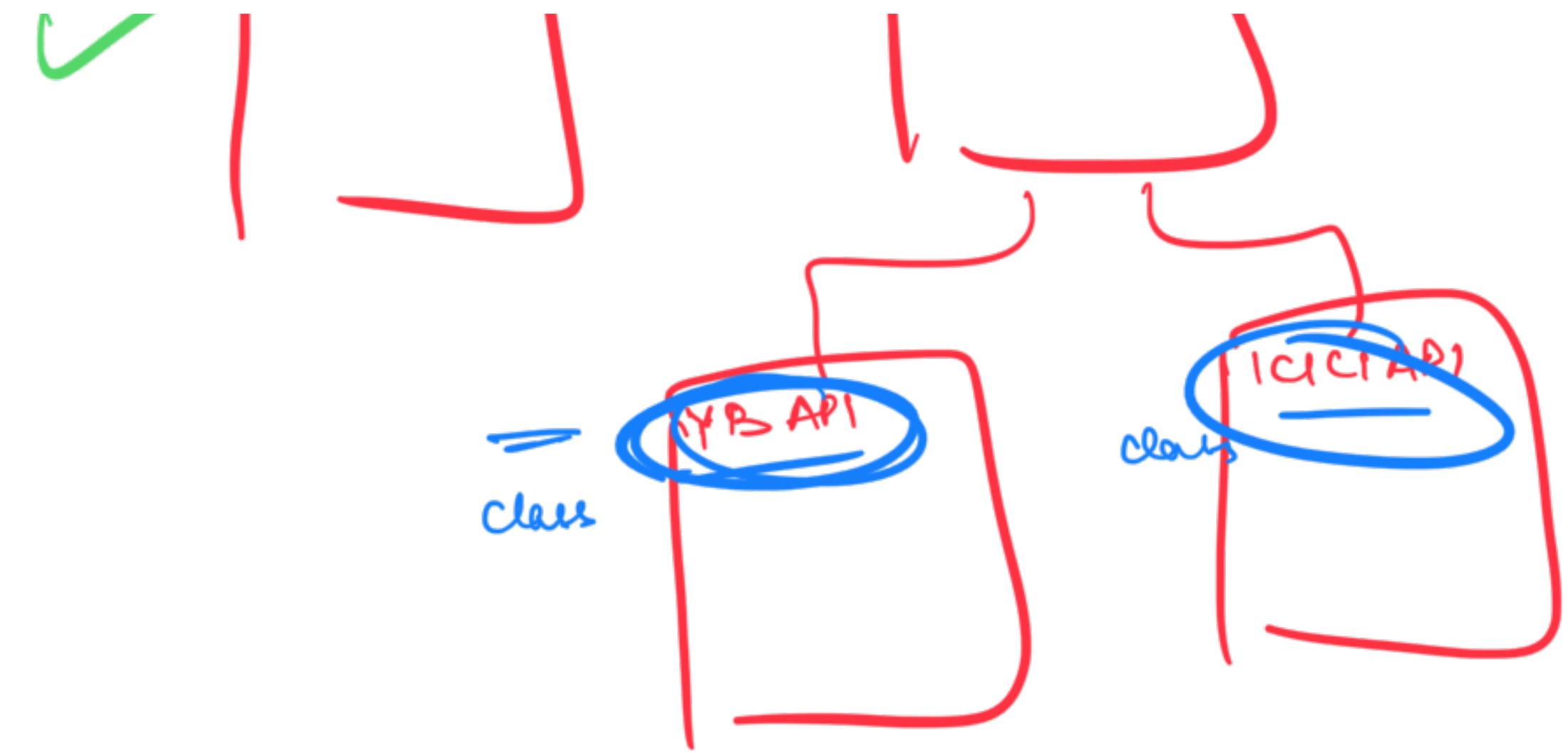
HDFCAPI

<< interface >> <- abstract class

Phone Pe

Bank

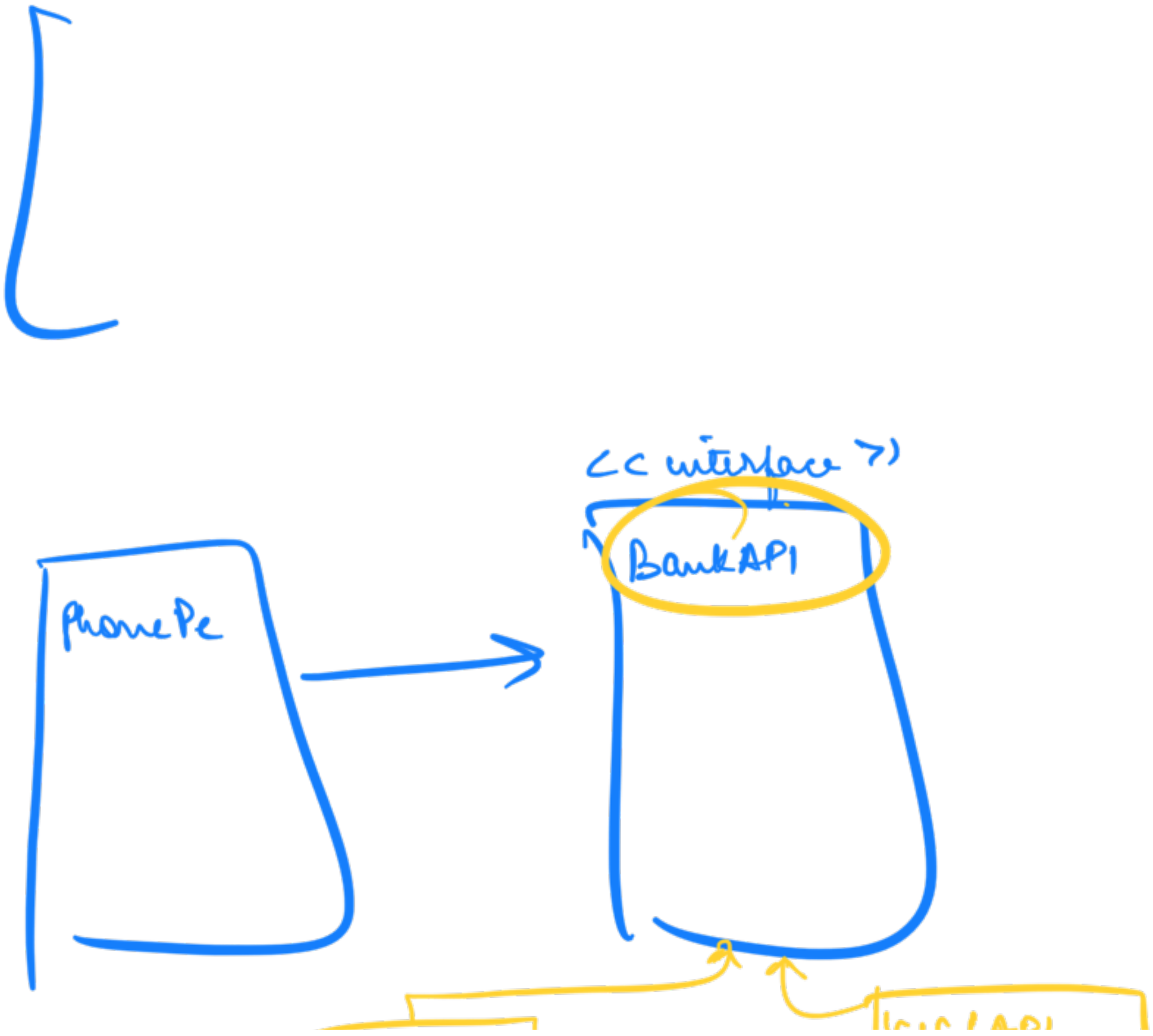
Bank API



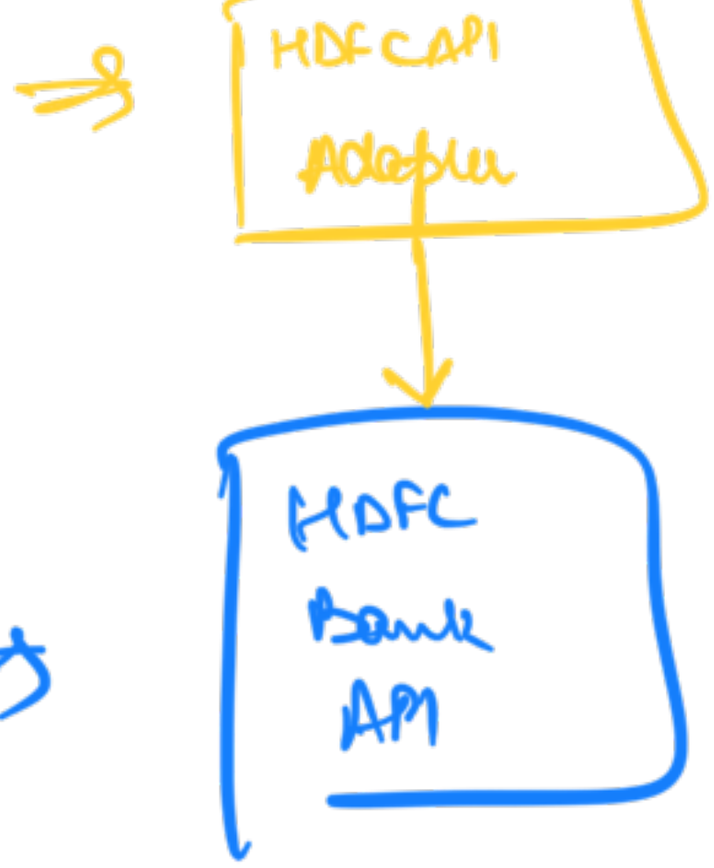
List<Integer> i = new ArrayList<>();

ArrayList<Integer> i = new ArrayList<>();

⇒ SOLID DP are violated


PhonePe
BankAPI

Wrappers created by people



classes that talk

Dependency: when 1 classes uses other class to support its behavior

class A {

B b;

A → B

Adapter Design Pattern

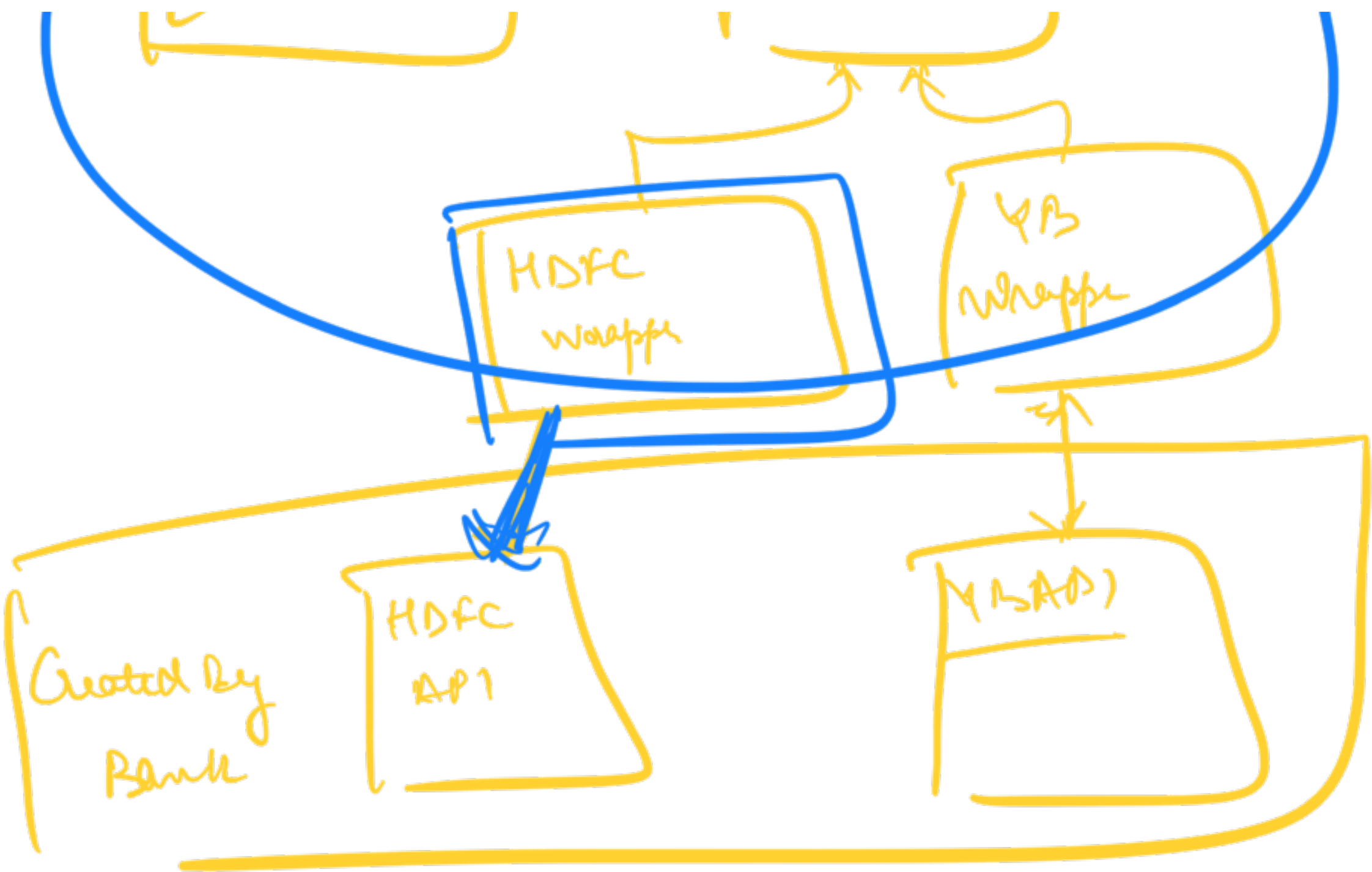
PhonePe

PhonePe

Bank API

interface

payes



HDFC Wrapper implements Bank API's

HDFC wrapper implementation

= HDFC API api = new HDFC API()

pay() {

api.doA()

api.doB()

}

}

App

Class

Outside

service

User Service

3

class User Service {

PostgreSQL DB db = new PostgreSQL DB

fetch() {

  db.getDetails()

db.getDetails();

}

}

fetch()

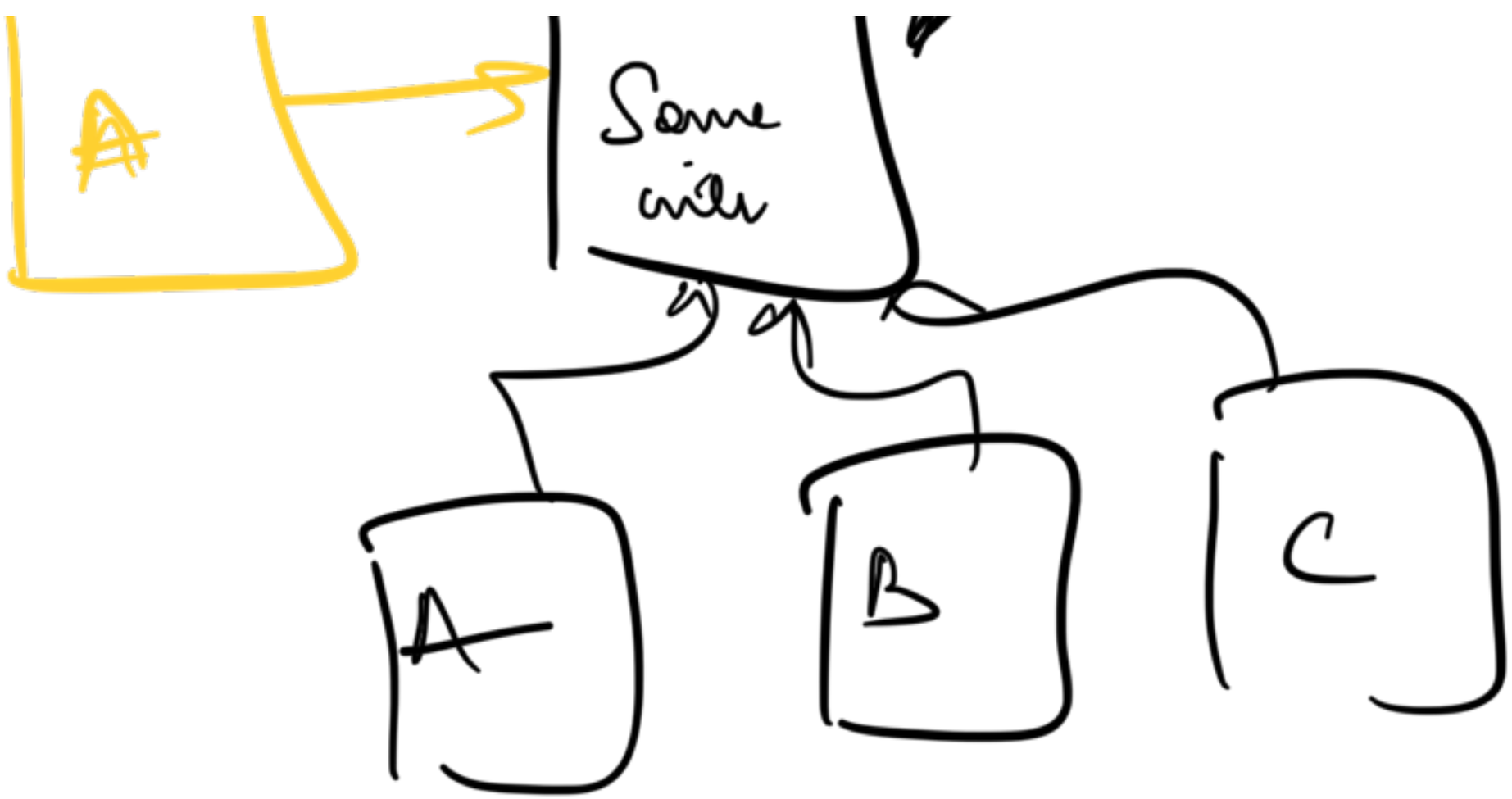
MongoDB

PostgreSQLDB().

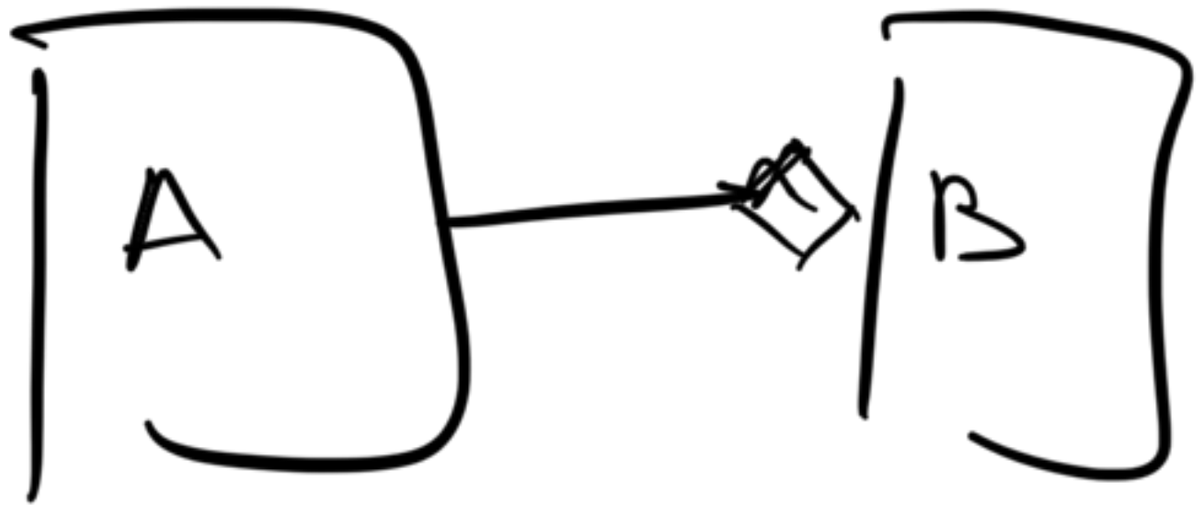
class UserService {

Database db = new MySQLDB()

}



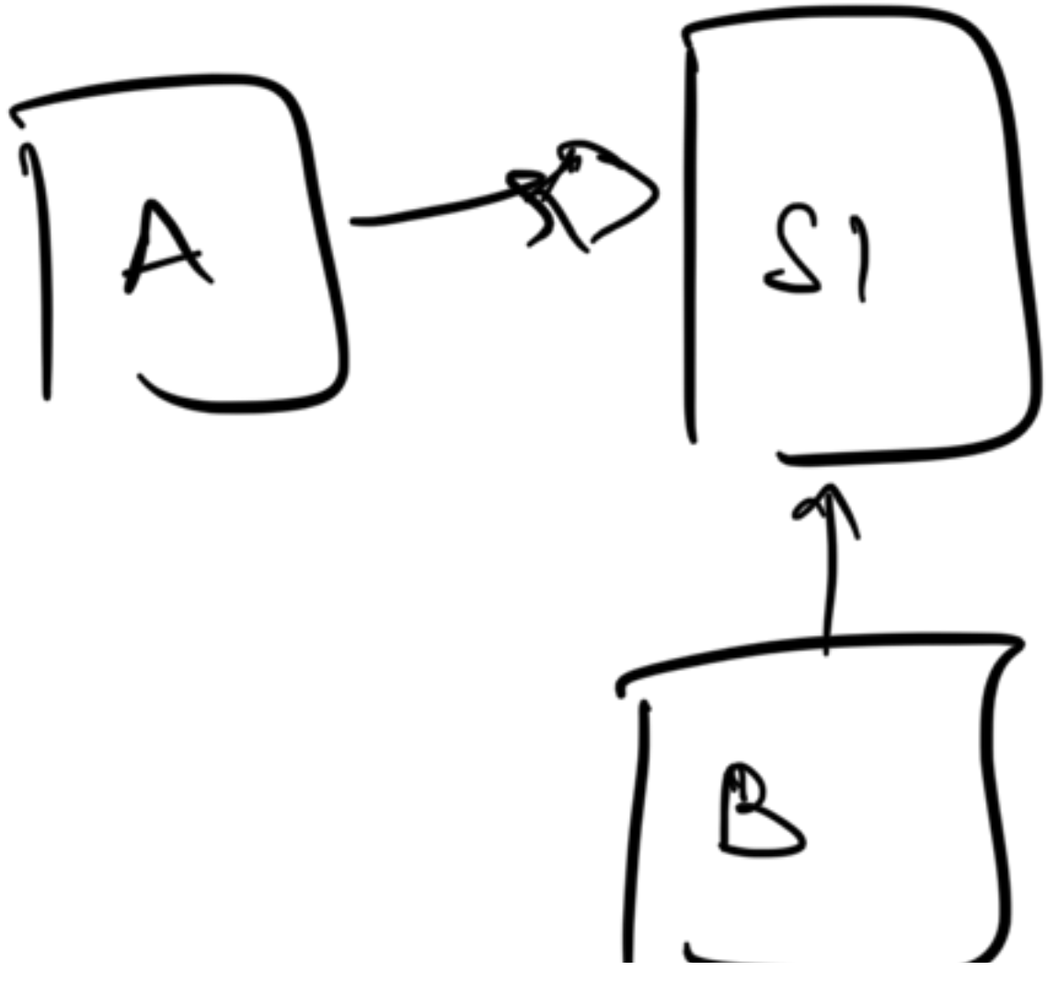
MySQL DB implements DBMS



Class A {

B bi;

}



class B implements S1 {}

class A {

S1 si = new B();

}

In your CB if you have any third party dependencies never depend directly on them

class User Service {

Database db = new PostgreSQLDB()

User

Dependency Injection

Classes should never create instances of their dependencies themselves

It should take the dependency via its constructor

Class UserService {

  Database db;

  → UserService (Database db)

    this.db = db;

}

Client {

  UserService us = new UserService(

    new MySQL

Client {

PhonePe pp = new PhonePe (

new HDFC Wrapper:

}

Spring Boot

Django

Angular

→ Any framework that supports Dependency Injection

Config : url

IoC

bank API = ~~Properties~~

Yes

when your framework does DI

Loose Coupling

"Classes not directly connected to each other but via an interface.

ENV

db = MySQL

List < Player > ls = List.of (

new Sparrow(),

new Crow(),

new Pigeon(),

)

HDFC Wale

HDFC Wrapper Gripbonds BankAPIs

HDFC
API

HDFCBankAPI api;

payC)}

}