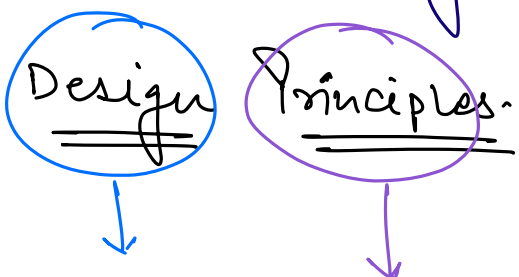
Agenda

- S : Single Responsibility Principle
- O : Open Close Principle
- L : liskov's Substitution Principle
- I : Interface Segregation Principle
- D : Dependency Inversion Principle

SOLID Design Principles.



⇒ Set of rules/guidelines defined in order to design software system that will have following properties.

1. Extensible
2. Maintainable
3. Understandable/Readable
4. Reusable
5. Modular.

DRY
⇒ Don't Repeat Yourself

Disclaimer: LLD is Subjective.

→ There's NO single right answer.

Design a BIRD.

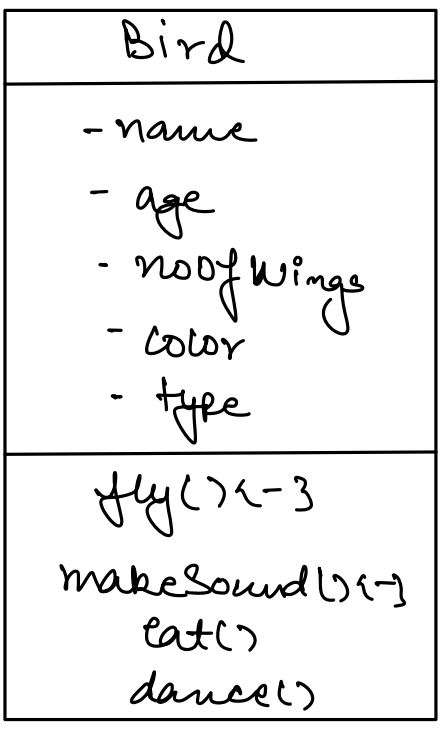
↳ Amazon Interviews.

⇒ Problem Statement

Build a Software System where we can store all the type of Birds.

⇒ Diversity of Birds.

V1



```
Bird b1 = new Bird();
b1.setName(—);
b1.setAge(—)
b1.setType("Crow");
```

⇒ b1.makeSound()

```
Bird b2 = new Bird();
b2.setName(—);
b2.setAge(—)
b2.setType("Pigeon");
```

⇒ b2.makeSound()

```
Void makeSound(type) {
    if(type == "Pigeon"){
        ——
        ——
        ——
    }
    else if(type == "Crow"){
        ——
        ——
        ——
        ——
        ——
        ——
    }
}
```

Problems with TOO MANY if-else conditions.

1. Understandability.
2. Difficult to test
3. Code duplicacy
4. No code reusability
5. Violates (S) of SOLID.

Single Responsibility Principle.

Every code unit (Class | Method | interface) in our codebase should have a single responsibility.

↓

There should be a single reason to change.

⇒ makeSound(—) is responsible for every bird to make a sound.

How to identify violation of SRP.

1. Method with too many if-else conditions.

⇒ Not always true.

⇒ Algorithm / Business logic.

```
CheckIfLeapYear(—){
    if(—){
        ——
    }
    else if(—){
        ——
    }
    ——
}
```

2. Monster Method.

↓

When a method does lot more things than what its name suggests.

```
saveToDatabase(User user){
    String query = "insert into users
                    ____
                    ____
                    ____";                    } (1)

    Database db = new DB()
    db.setURL(—);
    db.createConnect();                       } (2)

    db.execute(query)                         ] (3)
}
```

⇒

```
saveToDatabase(User user){
    String query = createQuery(—);
    CreateDBConnection()

    // Execute
}
```

Note: SRP ensures code reusability.

③ Commons | Utils.

→ Discouraged.

/utils
  | StringUtils.java
  | DateUtils.java
  | StudentUtils.java

SRP Summary

↓

Single Responsibility Principle.

① Too many if-else
② Monster method
③ Common | Utils.

8:30 Am.

Open Close Principle. (OCP)

⇒ Our codebase should be open for extension but closed for modification

extension → extensibility

⇒ Our codebase should be easily extensible but to add new feature we shouldn't require to change the existing code files.

adding new feature.

⇒ Rather than modifying the existing code we should try to add new code units.

⇒ Adding a new feature in our codebase should require very less changes in the existing codebase.

codebase → Project.

Bird
- name - age - noOfWings - color - type
fly() {-3 makeSound() {-} eat() dance()

⇒ Till now.

Sparrow, Crow, Pigeon

⇒ Add a new type of Bird

Peacock.

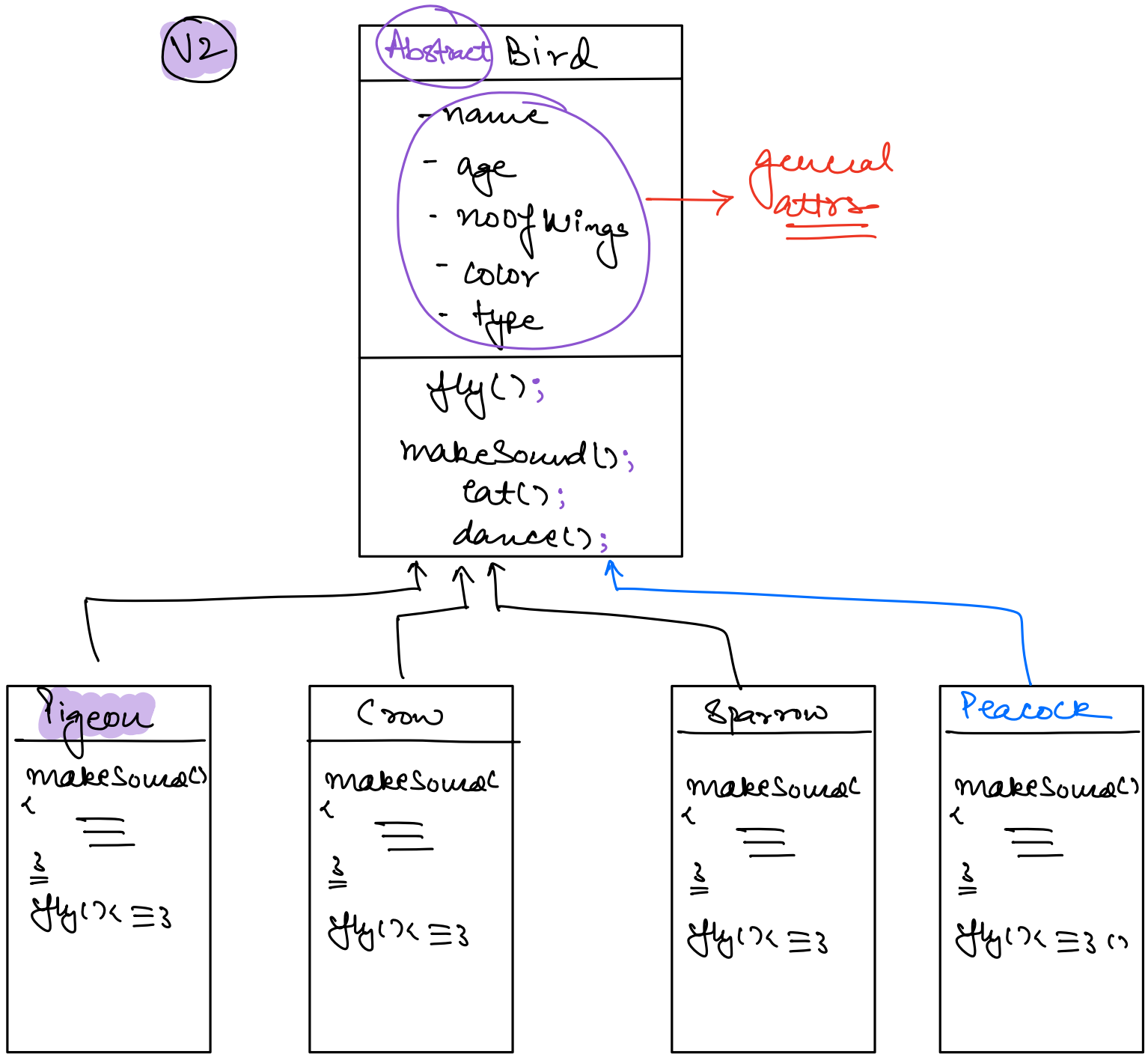
Why OCP ?

⇒

1. Existing code might start breaking.
2. Testing
   - Regression. (QA).

Sol4

Let the Bird class be only responsible for storing the general attributes/methods for Bird.



⇒ Too many if-else conditions. ✗

⇒ Req: Add a new type of Bird Peacock.

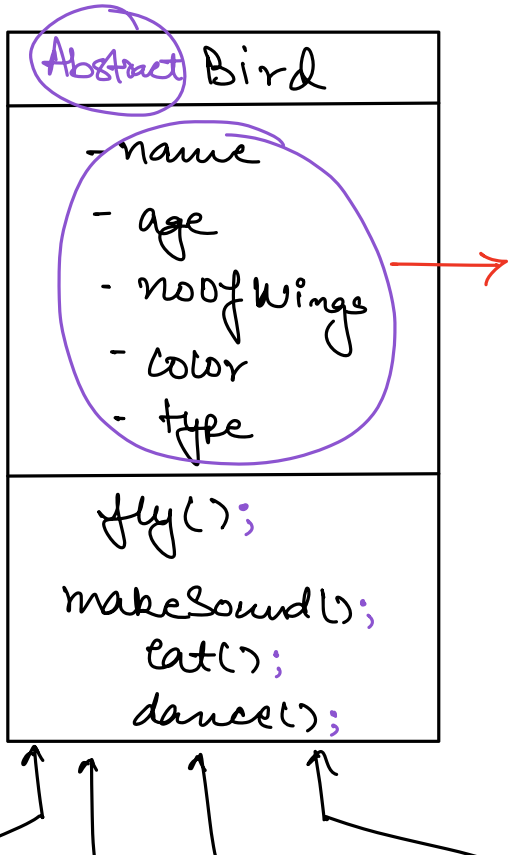
① To add a new Bird, we just have to create a new without making code changes in the existing codebases.

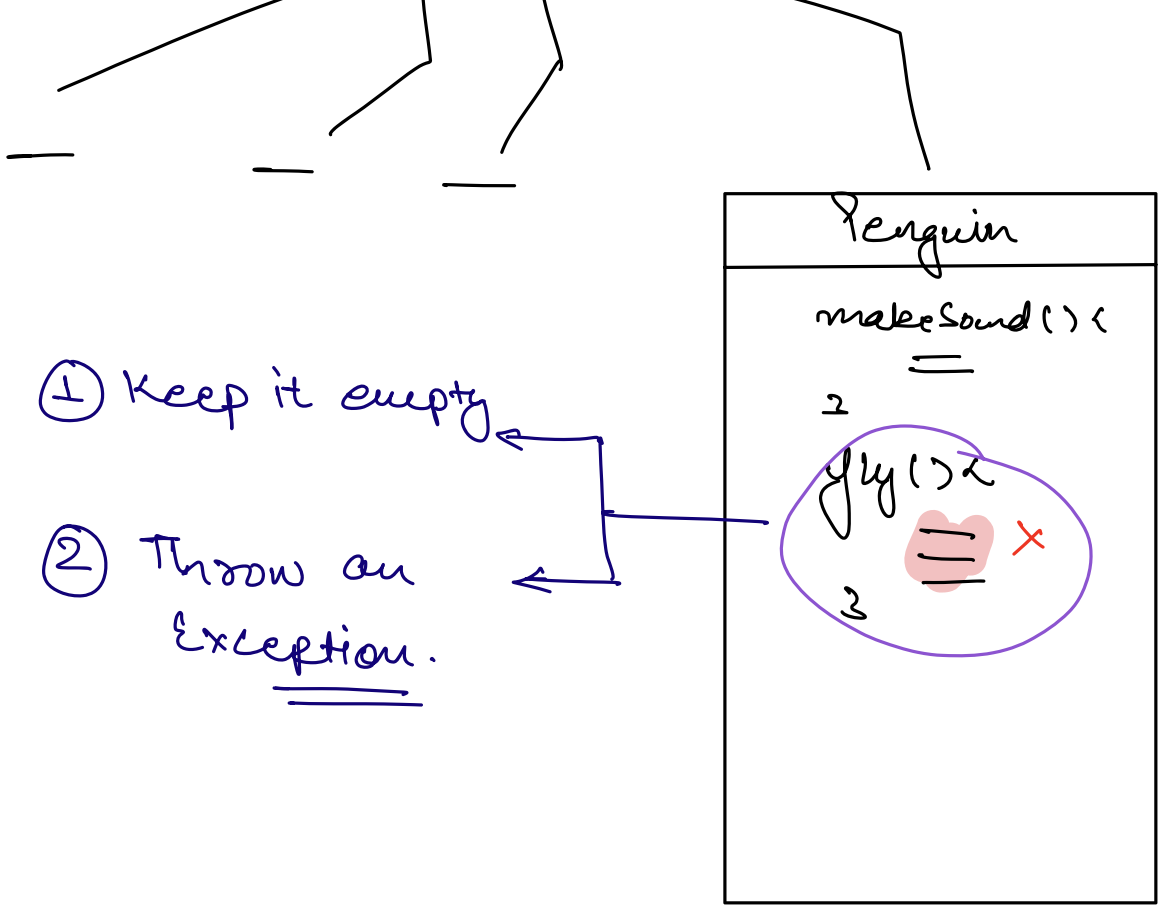
② Bird class is only responsible for general bird behaviours.

Requirement -

⇒ Add a new bird Penguin to the System.

Penguin → Can't fly





Client {

Bird b = new Penguin()

b.fly() ⇒ Unexpected behaviour.

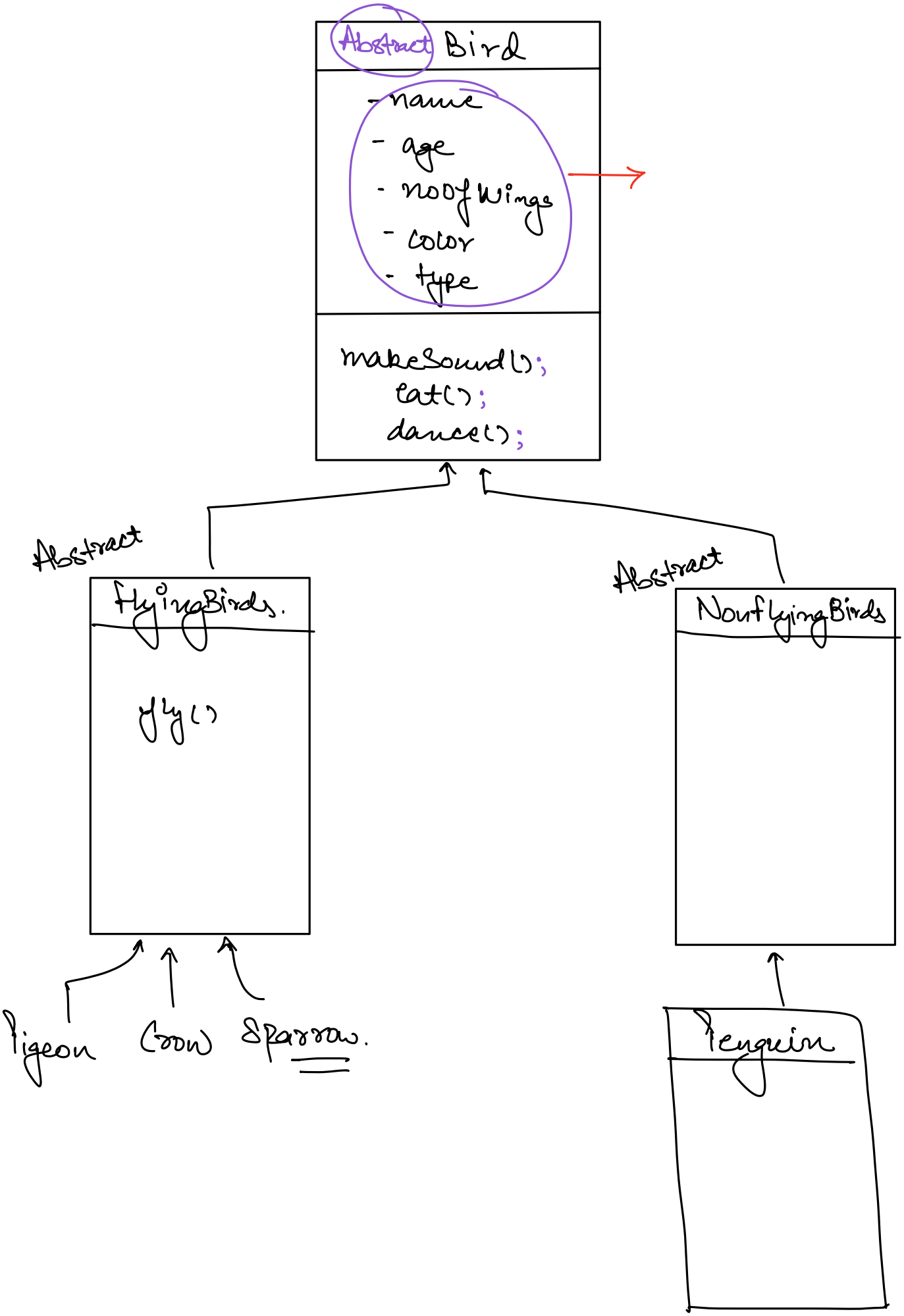
}

⇒ Client will get surprised.

⇒ Never give surprises to Client.

Ideal Soln

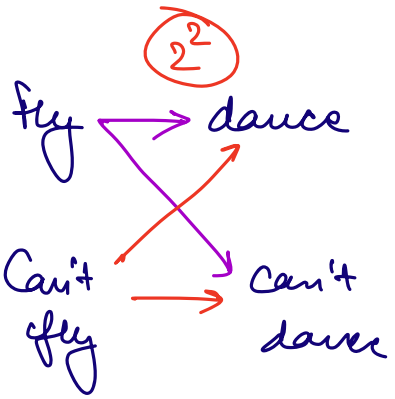
⇒ If a behavior is NOT supported by a particular type of object then that behavior shouldn't be available.

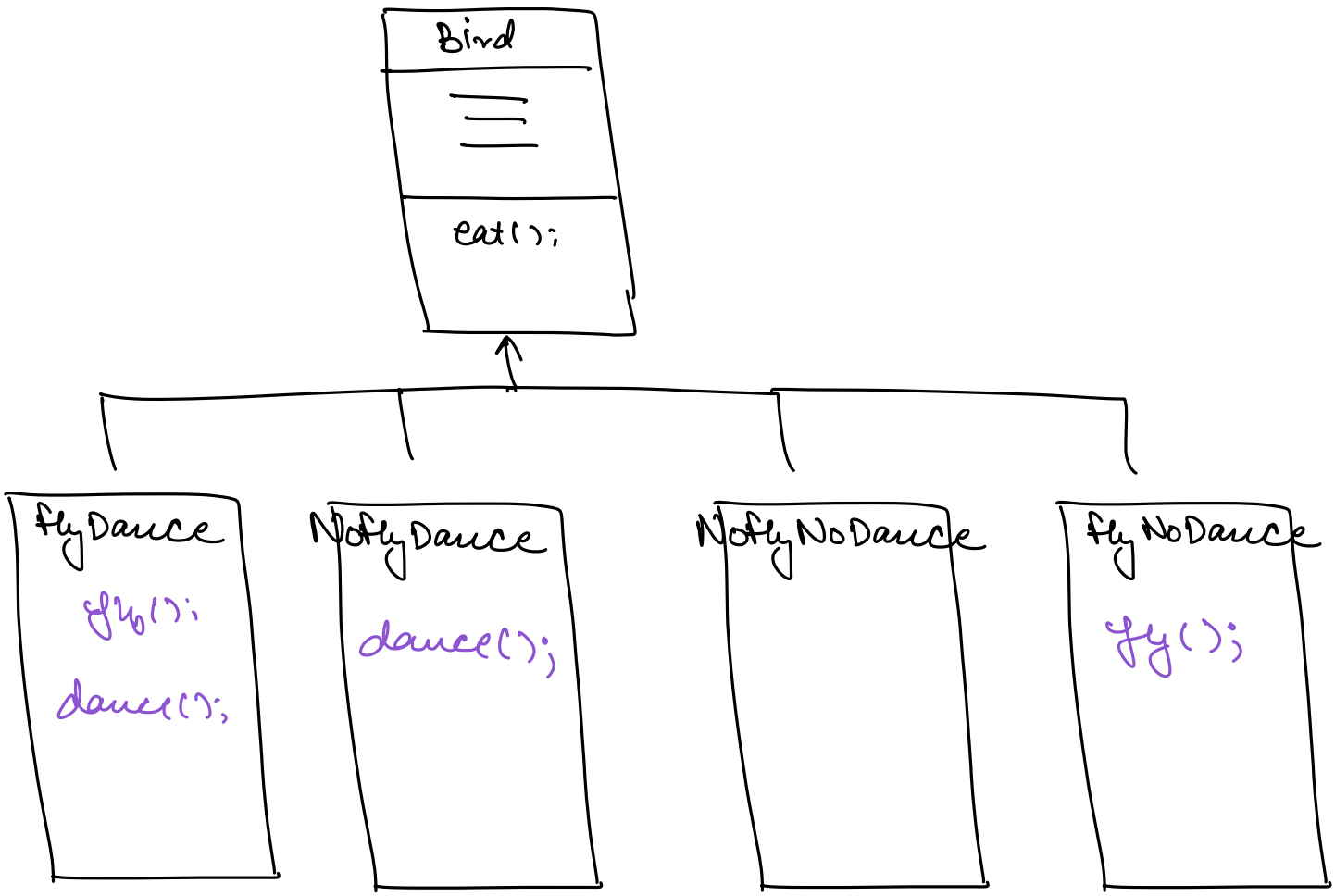

Abstract Bird
- name
- age
- noOfWings
- color
- type
makeSound();
eat();
dance();
Abstract
FlyingBirds.
fly()
Abstract
NonflyingBirds
Pigeon
Crow
Sparrow.
Penguin

printNames(List<Bird> birds)

⇒ fly | Can't fly

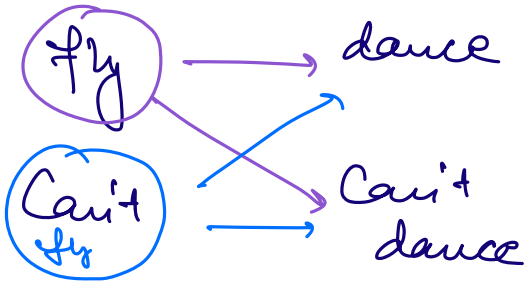
Dance | Can't Dance.

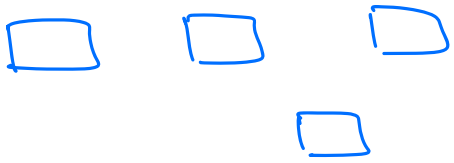




Problems.

⇒ Class Explosion : Too many classes.





1000 Birds

1000 class

⇒ Pigeon (P) = new Pigeon

P.fly();

Bird b = (——) new Pigeon;

b.fly(); ×

Agenda:

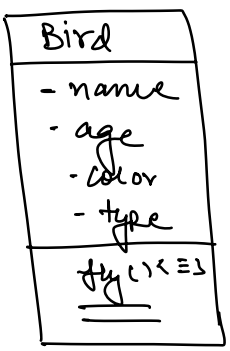
→ Recap

→ Liskov's Substitution

→ Interface Segregation

→ Dependency Inversion.

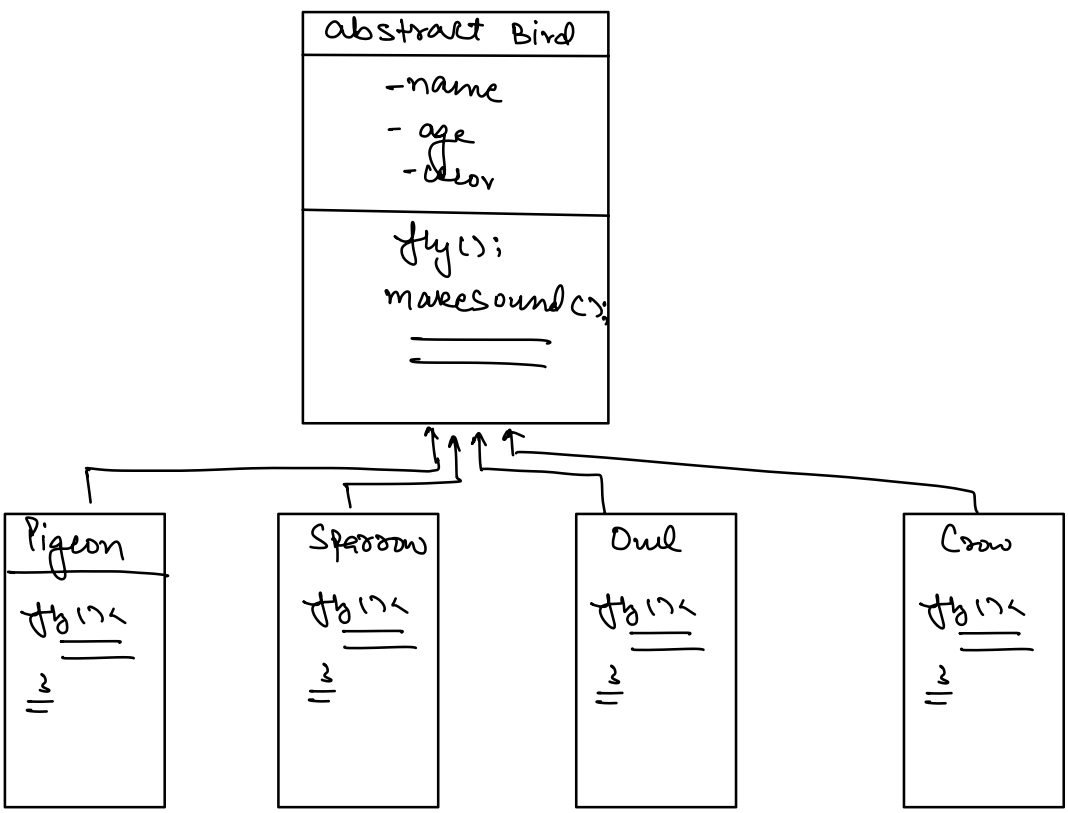
V1



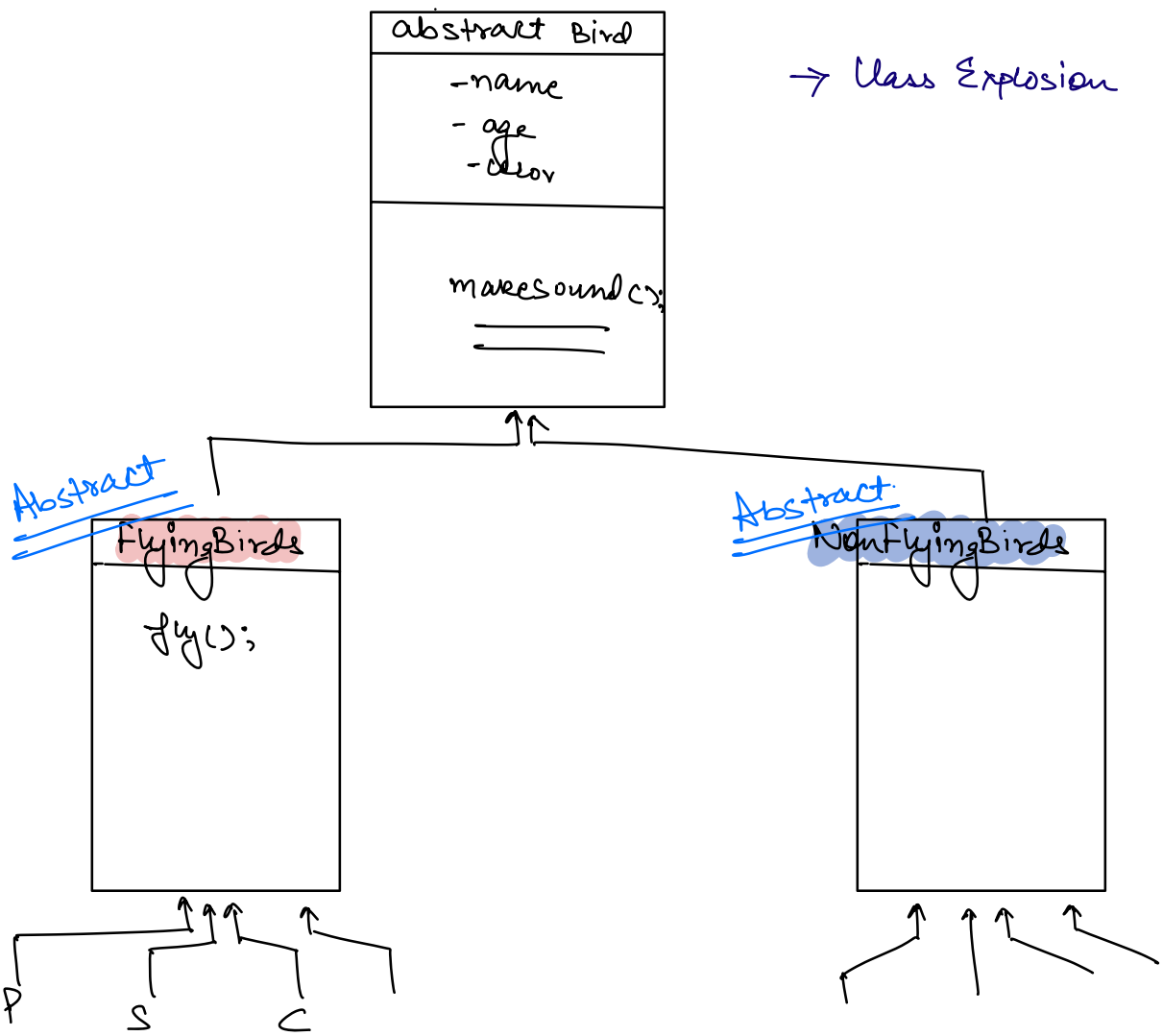
SRP ×

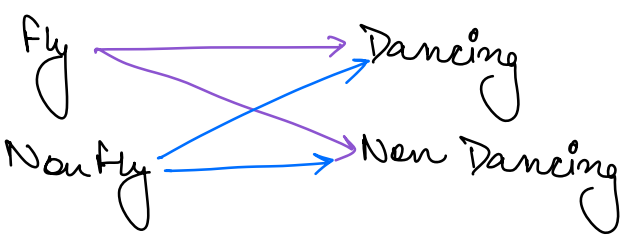
OCP ×

V2



Penguin => Can't fly.





List<FlyingBirds> ——

Problem Statement

⇒ Some birds demonstrate a behaviour while others are not.

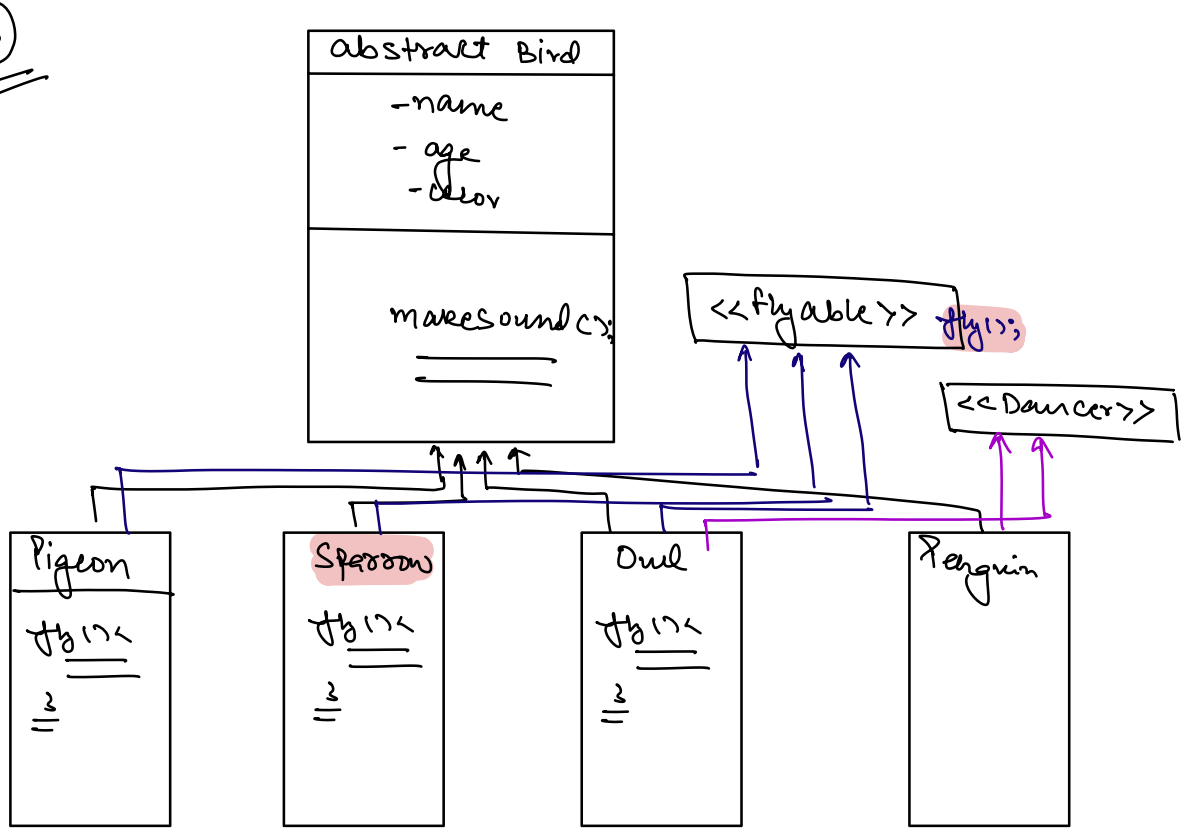
① Only the birds having a behaviour should have that method.

② We should be able to create a list of birds with a particular behaviour.

Classes. ⇒ Entity

Interface. ⇒ Behaviour.

(V3)



⇒ for fly behaviour, create an interface Flyable & the birds who can fly can implement the interface and others don't have.

⇒ Liskov Substitution Principle

Object of any child class should be as is substitutable in a variable of parent class without making any extra changes.

Bird b = new Pigeon()
new Sparrow()
new Penguin()

b.fly()

List<flyable> ← All flying Birds.

List<Dancers>

⇒ No special treatment to any child class.

Bird b = new Sparrow()

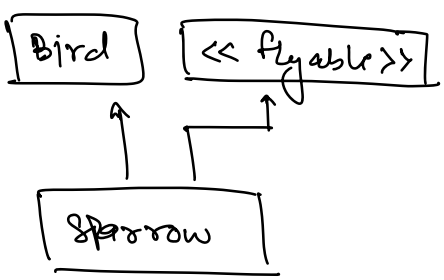
Class Sparrow extends Birds implements flyable {

Bird b = Sparrow()

flyable f = Sparrow.

fly() {

}



Class Penguin extends Birds {

}

3

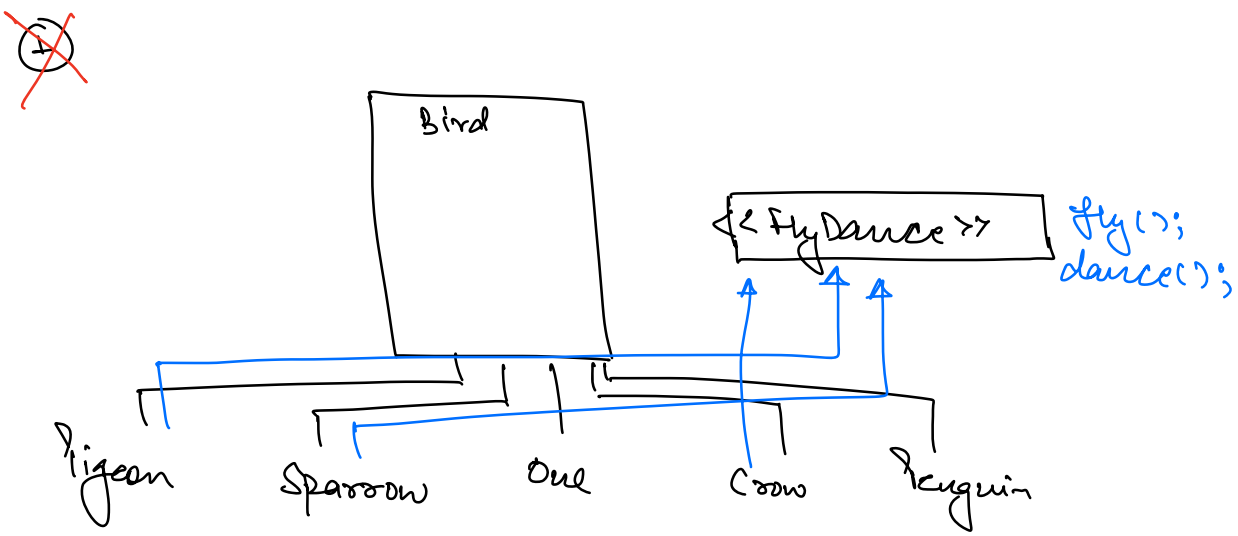
⇒ Interface Segregation Principle.

Req: Some birds can fly.

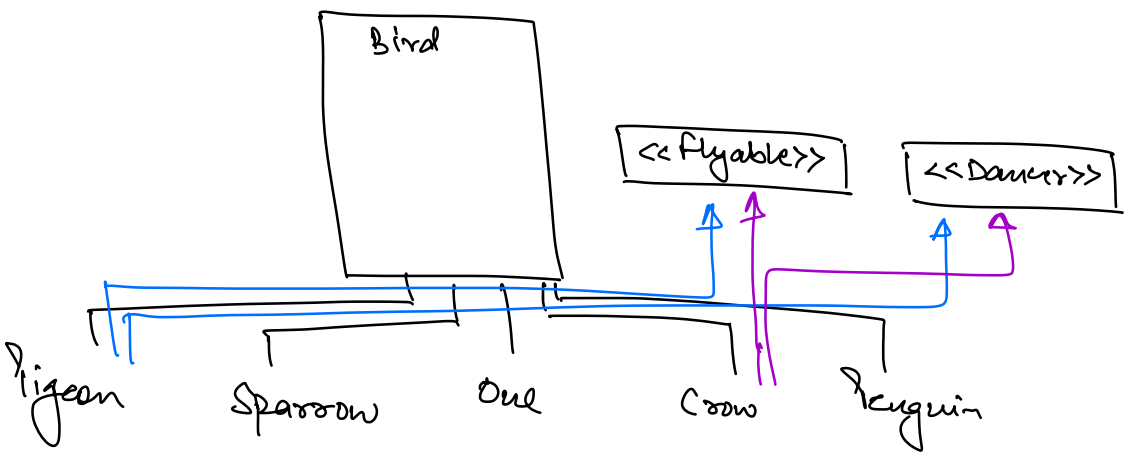
Some birds can dance.

All birds who can fly, they can dance as well & vice-versa.

Birds who can't fly they can't dance as well.



(2)



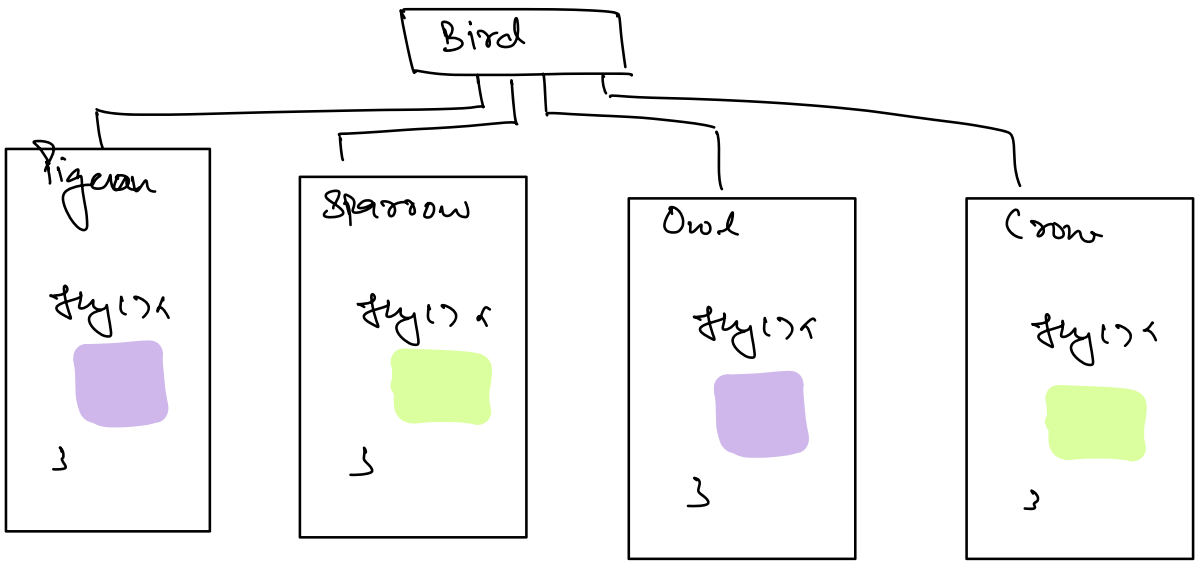
Interface Segregation Principle.

→ Interfaces should be as light as possible.
→ As less methods as possible.
→ Ideally Interfaces should have a single behaviour.

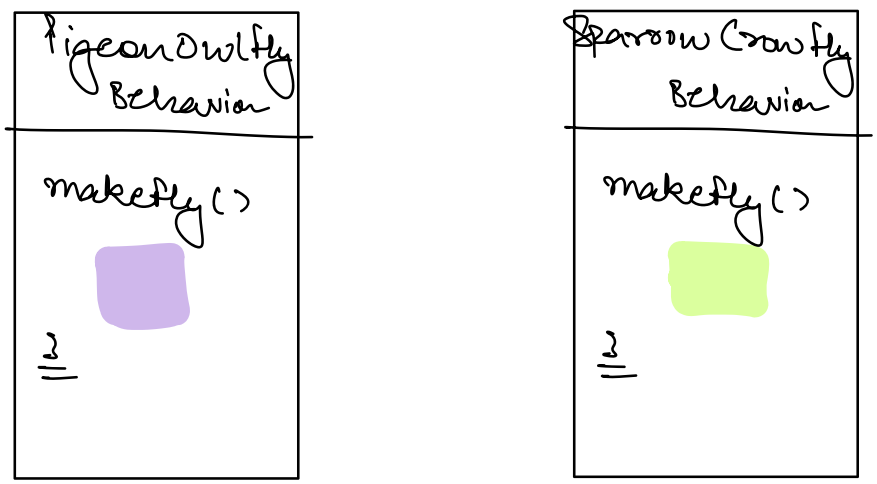
⇒ Functial Interface. ⇒ Interface with single method.

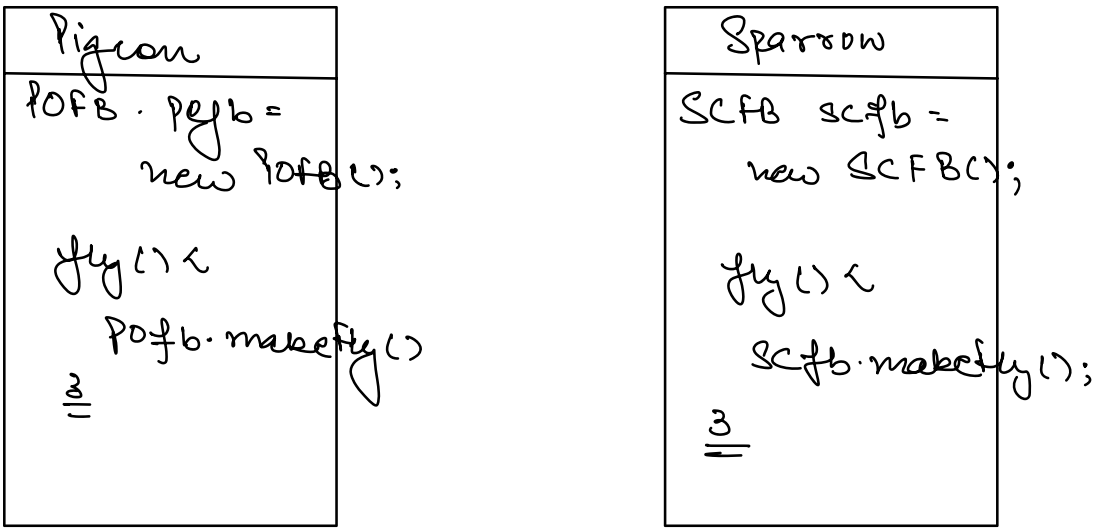
⇒ SRP on interface.

Dependency Inversion Principle.



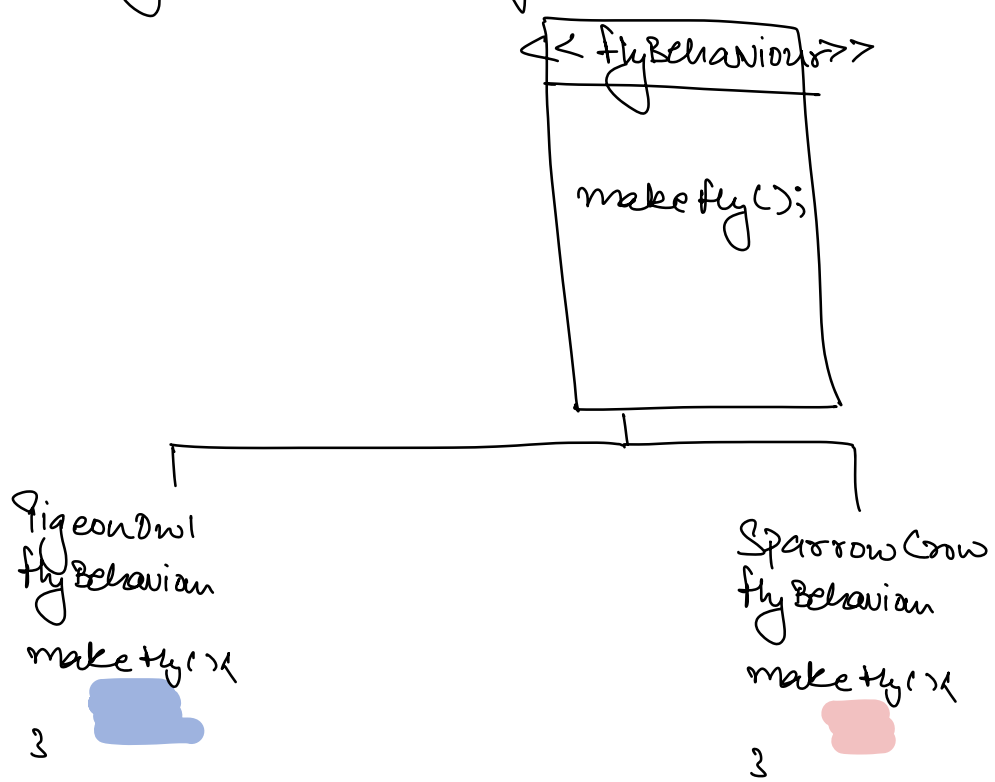
⇒ Code duplicacy

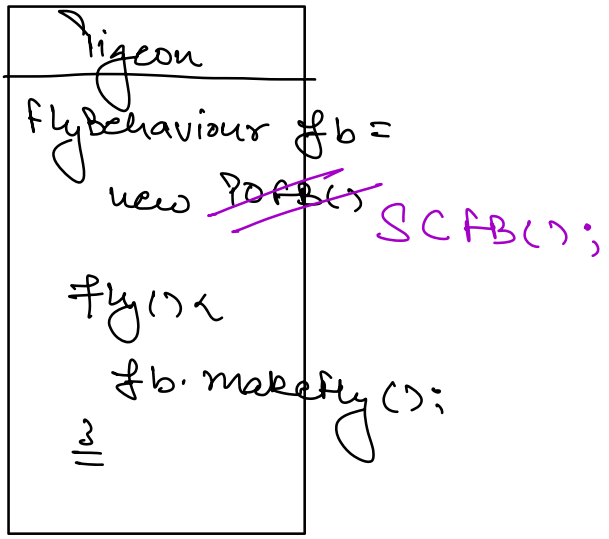




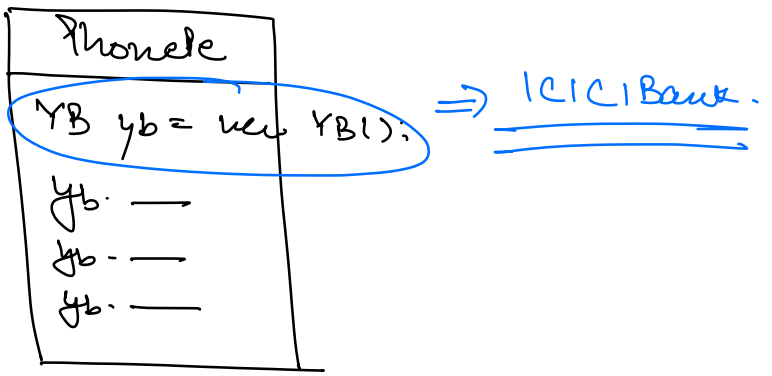
⇒ No Code duplicacy

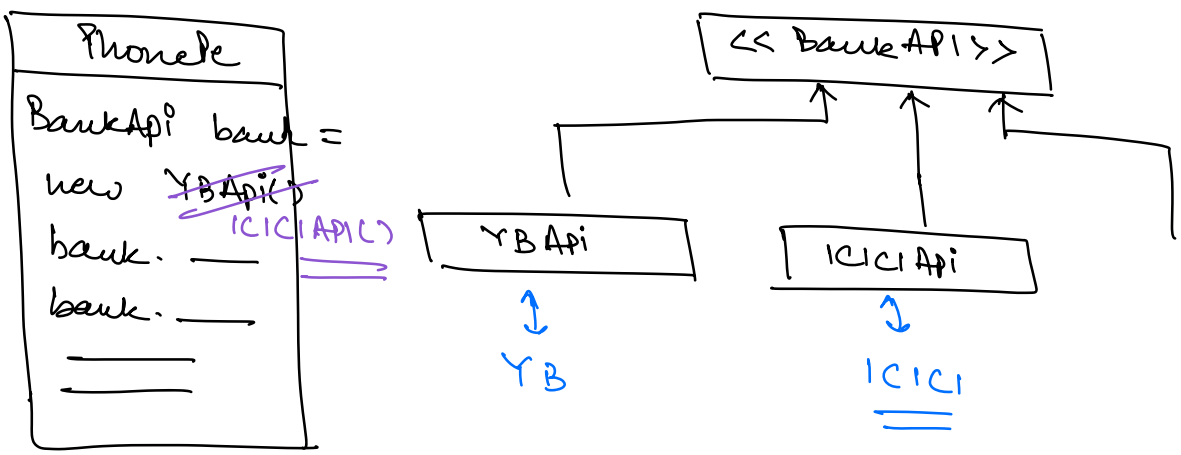
⇒ Program to interface, not to implementation




Pigeon
FlyBehaviour fb =
new ~~DoFB()~~ SCFB();
fly(){
fb.makefly();
}

⇒


Phonepe
YB yb = new YB();
⇒ ICICIBank.
yb.
yb.
yb.


Phonepe
BankApi bank =
new ~~YBApi()~~
ICICIApi()
bank.
bank.
<< BankAPI >>
YBApi
ICICIApi
YB
ICICI

Dependency Inversion

⇒ No 2 concrete classes should depend on each other directly, they should depend on each other via an interface.

X —— SOLID —— X

Dependency Injection.

→ It's NOT a part of SOLID.

→ If a class (A) is dependent on (B).

(B) ↓ ↙ Interface

Class

Class A {

B b = new ✕();

}

⇒ Pigeon {

FlyBehaviour fb = new POFB(); ✗

=

Pigeon {

FlyBehaviour fb;

Pigeon(FlyBehaviour obj) {

this.fb = obj;

Constructor

}

=

=

PigeonOwnFlyBehavior pofb = ——————;

Pigeon p = new Pigeon(pofb);

⇒ SpringBoot

⇒ Django.

⇒ Dependency Injection

⇒ No need to create an object of dependency on our own, instead let the user of the class create the dependency object.

⇒ Recap.

S : SRP
O : OCP
L : LSP
I : ISP
D : DIP.

——— * ———

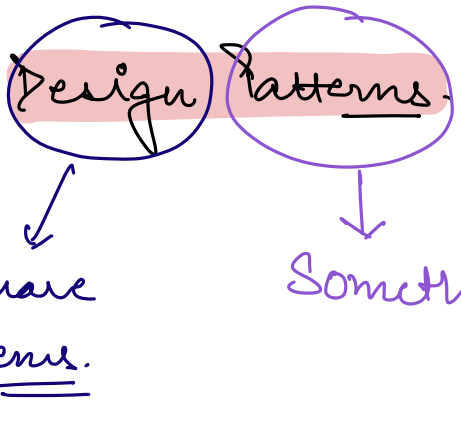
Agenda -

→ What are Design Patterns.

→ Type of Design Patterns.

→ Singleton

What are Design Patterns.



⇒ Well established solutions to commonly occuring usecases in Software System design.

⇒ Type of Design Patterns.

① Creational.

⇒ How to create an object ?

⇒ How many objects to be created ?

② Structural.

⇒ How a class should be structured?

⇒ What all the attrs a class should have?

⇒ How a class can interact with other class.

③ Behavioural.

⇒ Methods.

⇒ How to code a method?

CREATIONAL. DESIGN PATTERNS.

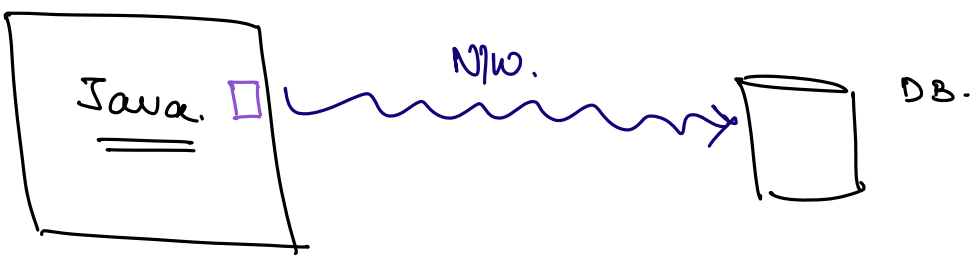
- Singleton
- Builder
- factory
- Prototype.

SINGLETON.

⇒ Allows us to create only a single object for a particular class

Why Singleton?

→ When object creation is expensive.



⇒ Database Connection

↳ Creating DB connection is an expensive operation, so we can use Singleton Pattern.

⇒ Logging : One logger object would be sufficient to write logs across the application.

⇒ Note: Singleton pattern is preffered in case Object creation is expensive.

⇒ Shared resource.

⇒ Class DatabaseConnection {

```
String url;
int portNo;
String userName;
String password;
```

}

⇒ DBC dbc1 = new DBC()

DBC dbc2 = new DBC()

⇒ Till the time constructor is available then we can create any # of objects of a class.

⇒ Make the constructor private.

Class DatabaseConnection {

```
String url;
int portNo;
String userName;
String password;
private DatabaseConnection() {
```

}

}

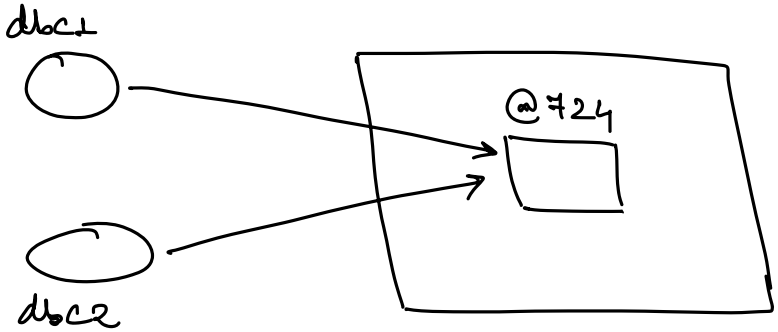
```
DBC dbc1 = new DBC()
```

⇒ We can't even create a single object of the class.

```
Class DatabaseConnection {
    String url;
    int portNo;
    String userName;
    String password;
    Private DatabaseConnection() {
    }
    Public static DBC getInstance() {
        return new DBC();
    }
}
```

⇒
```
DBC dbc1 = DBC.getInstance();
DBC dbc2 = DBC.getInstance();
```

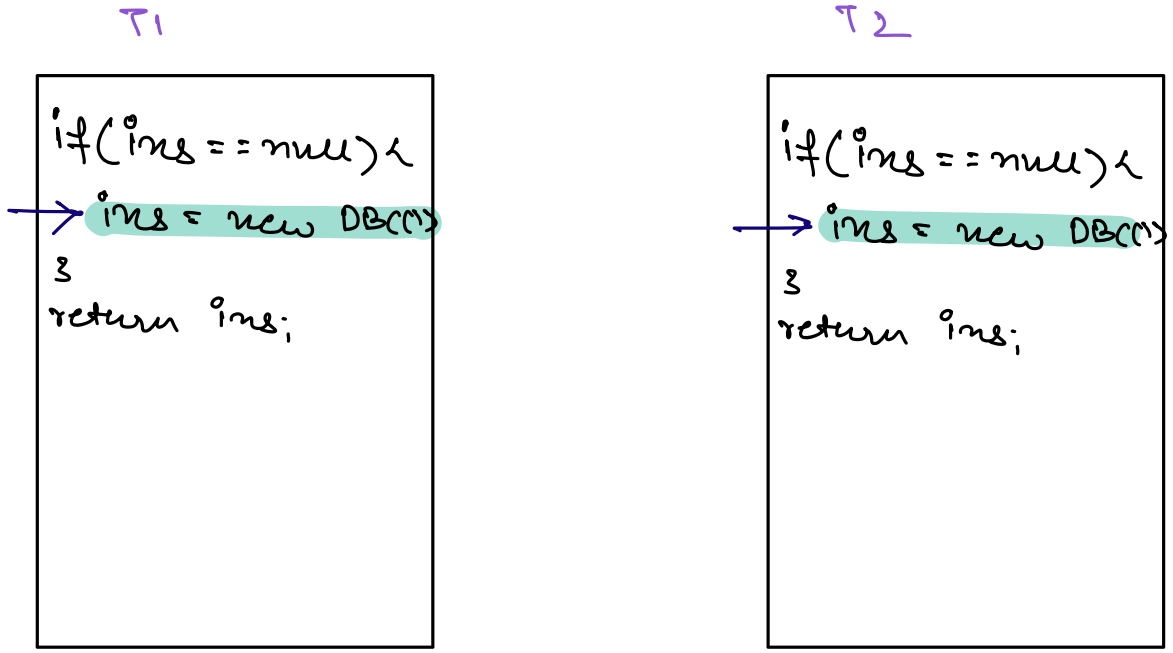
```
Class DatabaseConnection {
    Private Static DBC dbc = null;

    String url;
    int portNo;
    String userName;
    String password;
    Private DatabaseConnection() {
    }
    Public Static DBC getInstance() {
        if (dbc == null) {
            dbc = new DBC();
        }
        return dbc;
    }
}

DBC dbc1 = DBC.getInstance();
DBC dbc2 = DBC.getInstance();
```



Steps.

1. Make the constructor private.
2. Create a public static getInstance method.
3. Create a static instance of class.

⇒ MULTI-THREADING.



⇒ Above code won't work in case Multi threading environment.

(1) EAGER / EARLY INITIALIZATION.

```
Class DatabaseConnection {
    Private Static DBC dbc = new DBC();

    String url;
    int portNo;
    String userName;
    String password;
    Private DatabaseConnection() {
    }
    Public Static DBC getInstance() {
        return dbc;
    }
}
```

⇒ Lot of static attrs will increase the App load time.

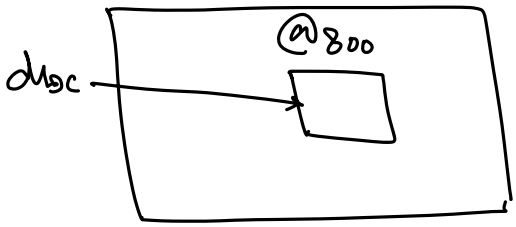
② LAZY.

→ Synchronized getInstance() method.

```
Class DatabaseConnection {
    Private Static DBC dbc = null;

    String url;
    int portNo;
    String userName;
    String password;
    Private DatabaseConnection() {
    }
    Static Synchronised
    Public DBC getInstance() {
        if (dbc == null) {
            dbc = new DBC();
        }
        return dbc;
    }
}
```

⇒ Slow performance.

Double Check Locking.




```
if (dbc == null){
    lock()
      if (dbc == null){
          dbc = new DBC();
      }
    unlock()
}
return dbc;
```

⇒ Best way to implement Singleton in Prod environment.

Steps.

1. Check without lock.
2. Take a lock.
3. Check again after lock.

⇒ Usecase of Singleton.

↳ Only 1 object.

⇒ Concurrency. ⇒ T1, T2, T3 → Commit.

⇒ BUILDER DESIGN PATTERN.

Class with lot of attributes.

```
Class Student {
    String name;
    int age;
    String batch
    double gpa;
    String univName;
    int gradYear;
    String phoneNo;
}
```

```
Student s = new Student()
s.setName(—);
s.setAge(—);
s.setGpa(—)
```

We want to validate the object before its creation.

Validations.

1. Students should have gradYear < 2020.
2. Phone no. should be valid.

⇒ No Student object should be created without checking validations.

```
class Student {
    String name;
    int age;
    String batch
    double gpa;
    String univName;
    int gradYear;
    String phoneNo;
}
```

```
Student(name, age,
        batch, gpa,
        univName,
        gradYear, phoneNo){
    if (gradYear >= 2021) {
        throw ——————
    }
    ——
    ——
    ——
}
```

```
psvm(){
    Student st = new Student("Kailash", 25,
        "Morning", 84.26, . . .
        ------);
}
```

Issues.

1. Difficult to understand.
2. Prone to Errors.

Class Student {

```
Student(name) {
    this.name = name
}
Student(name, age) {
    this.name = name
}
Student(name, univName) {
    this.name = name
}
Student(name, batch) {
}
}
```

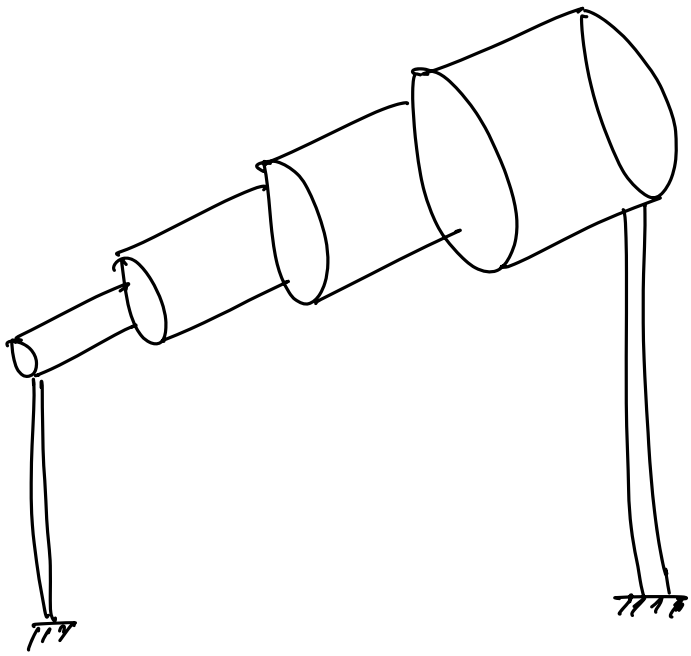
⇒ Too Constructors with same method signatures are not even allowed.

⇒ Issue :

Too many constructors.

⇒ 2^N

⇒ Telescopic Constructors.



```
Student (name){
    this.name = name;
}

Student (name, univName){
    this(name);
    this.univName = univName;
}
```

⇒ Class Student {

```
    Student (param) { attrs ⇒
    }
}
```

⇒ Map:

Some data structure that can allow us to pass multiple attrs ⇒

name: ____
age: ____
batch: ____

HashMap<String, Object> map.

```
Class Student {
    ------
    ------
    ------
    ------
    ------
    ------

    Student (Map<String, object> map) {
        this.name = (String) map.get("name");
        this.age = (Integer) map.get("age");
        ------
        ------
    }
```

→ Type Casting

⇒ It can lead to some Runtime Exceptions.

```
psvm() {

    Map<String, Object> map = ----------;

    map.put("nama", "Shorabh");
    ----------  ----------
    ----------  ----------
    ----------  ----------

    Student st = new Student(map);
}
```

Typo.

→ No compile time Check on Attr names.

Something which is like a Map (it allows us to store Attr names with their values) and also provides compile time safety over the attribute names & attribute type.

map.name = ~~45~~

map.name = " ✗ "

Builder.

Class ~~Helper~~ {

String name;
int age;
String batch
double gpa;
String univName;
int gradYear;
String phoneNo;

}

Helper helper = new Helper();
helper.setName(45);
helper.setGradYear (___);

psvm().

Class Student {

No type casting.

Student(Helper helper) { → helper.name

// Validations.

this.name = helper.name;

this.age = helper.age;

}

}

⇒ BUILDER.

Allows us to create an object where we have

(1) Class with too many attributes.

(2) Validate before object creation.

Agenda.

→ Prototype

→ Registry.

Creational.

PROTOTYPE DESIGN PATTERN.

Problem Statement

→ Given an object of a class.

→ We need to create a copy of that object.

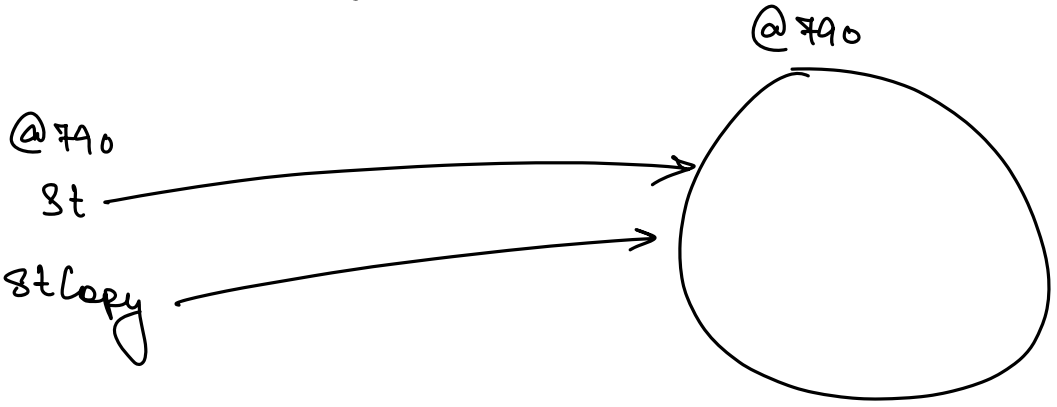
{Create a new object with exact same attrs}

```
psvm() {
    Student st = new Student();
    st.name = ____
    st.age = ____
    st.psp = ____
    ____
    ____
```

```
    Student stCopy = new Student();
    stCopy.name = st.name;
    stCopy.age = st.age;
    ____
    ____
    ____
```

3

Student stCopy = st ; ✗

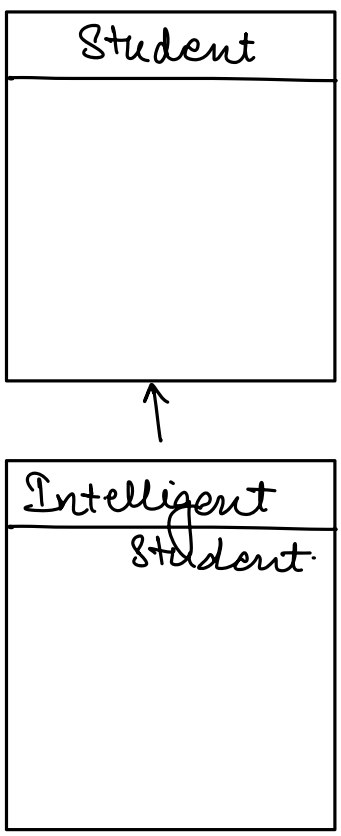


Issue

→ Client needs to know the complete details of Student object.

→ Student might have some private attrs that client might not be able to access.

→



Student ↓ ↓ IS

```
Student original = ____

Student copy = ?

if (original.instanceOf(Student)){
    copy = new Student();
}
else{
    copy = new IS();
}
```

⇒ OCP. Violation.

Copy Constructor.

```
Class Student {

    Student ( Student st ) {
        this.name = st.name;
        this.age = st.age;
    }
}
```

```
Student st = new Student();
Student stCopy = new Student(st);
```

⇒ Class IntelligentStudent extends Student {

```
    IntelligentStudent (IS st) {
        this.name = st.name;
        this.age = st.age;
    }
}
```

⇒

Student original = ——— ← IS().
↓ Student

Student copy = ?

```
if (original.instanceOf(Student)){
        copy = new Student(original);
}
else{
        copy = new IS(original);
}
```

} OCP X.

⇒ If client wants to create a copy of an object then having the copy logic on client side is prone to Errors.

⇒ Ideal Sol^{n} can be that if client outsources the copy logic to the object itself

⇒ Student {

```
Student Copy(){
        Student stCopy = new Student();
        stCopy.name = this.name;

        return copy;
}
}
```

```
main(){
    Student original = new Student();
    Student copy = original.copy();
}
```

⇒ Intelligent Student :

```
Class IntelligentStudent extends Student{
    Student Clone.
    Student Copy(){
        Is stCopy = new Is();
        stCopy.name = this.name;
        return copy;
    }
}
```

```
Student original = ____________;
                   ↓ IntelligentStudent
Student copy = original.copy();
               Is
```

⇒ All the child class must also override the copy method.

Student original = ———— ; ← new Student()

Student copy = original.copy()

Prototype Design Pattern.

↓

Sample.

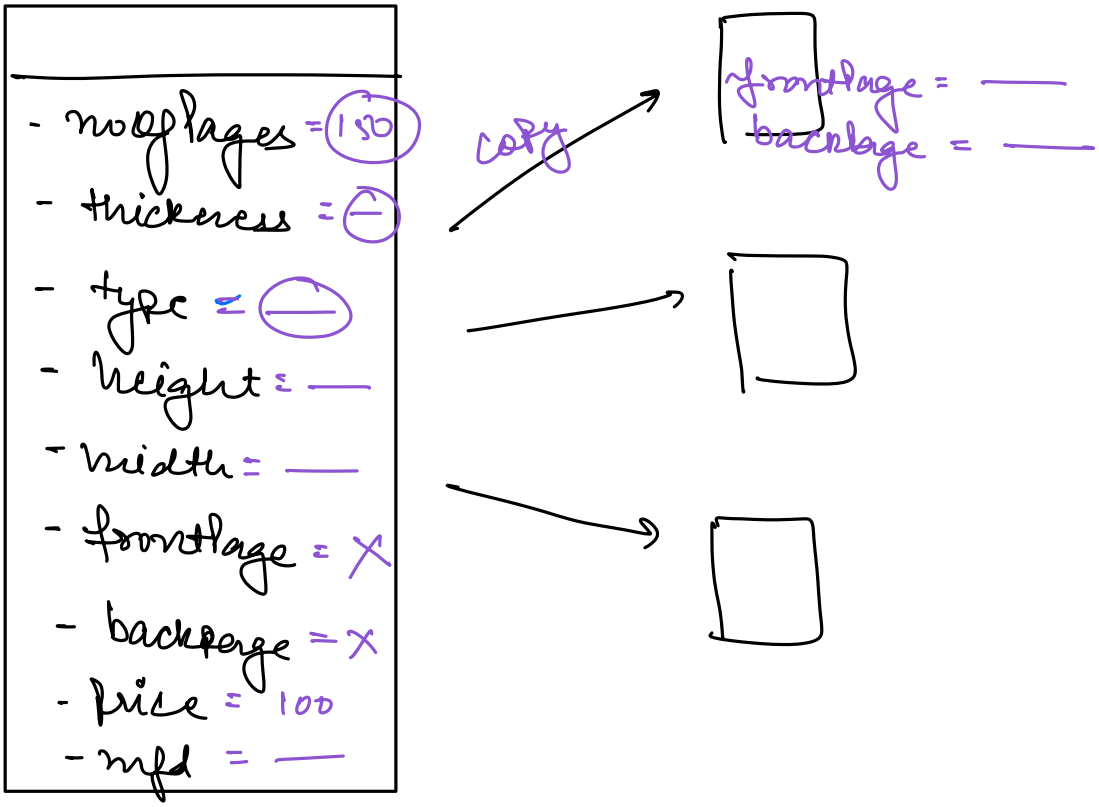
⇒ Classmate Notebooks.

Notebook
- noOfPages ✓
- thickness ✓
- type ✓
- height ✓
- width ✓
- frontPage ✗
- backPage ✗
- price ✓
- mfd ✓

Requirement:

Create 10000 notebooks of type plain & with 150 pages each.

Sample Object



Prototype Design Pattern.

⇒ Often there are scenarios where we don't want to create an object from scratch, rather we prefer creating an object from some existing template & changing only the required fields.

⇒ Class Student {

name

age

batch

psp

avgbatchpsp

Contest

address

CompanyName;

}

```
Student vishnu = new Student();
vishnu.name = ———
vishnu.age = ———
vishnu.batch = "Mar22 MWF9
vishnu.psp = 80
vishnu.avgbatchpsp = 75.0
```

Way 1

```
Student umes = new Student();
umes.name = ———
umes.age = ———
umes.batch = "Mar22 MWF9
umes.psp = 90
umes.avgbatchpsp = 75.0
```

Way 2

Registry

```
Student prototype = ____
prototype.avgBatchPsp = 75.0;
prototype.batch = "May22 MWF";
prototype.contest = "____";
```

```
Student umes = prototype.copy();
umes.name = ____
umes.age = ____
umes.psp = 90
```

⇒ Steps.

1. In the class for which we want to create prototypes, declare a method copy() | clone() to create copy object.

Note: All the child class should Override the copy method

2. Create the prototype objects & store them in Registry

→ Map<String, Object>

3. Create the copy object from prototypes stored in Registry whenever required.

→ Class to store prototype objects.

Agenda.

↳ factory, Abstract factory & Practical factory.

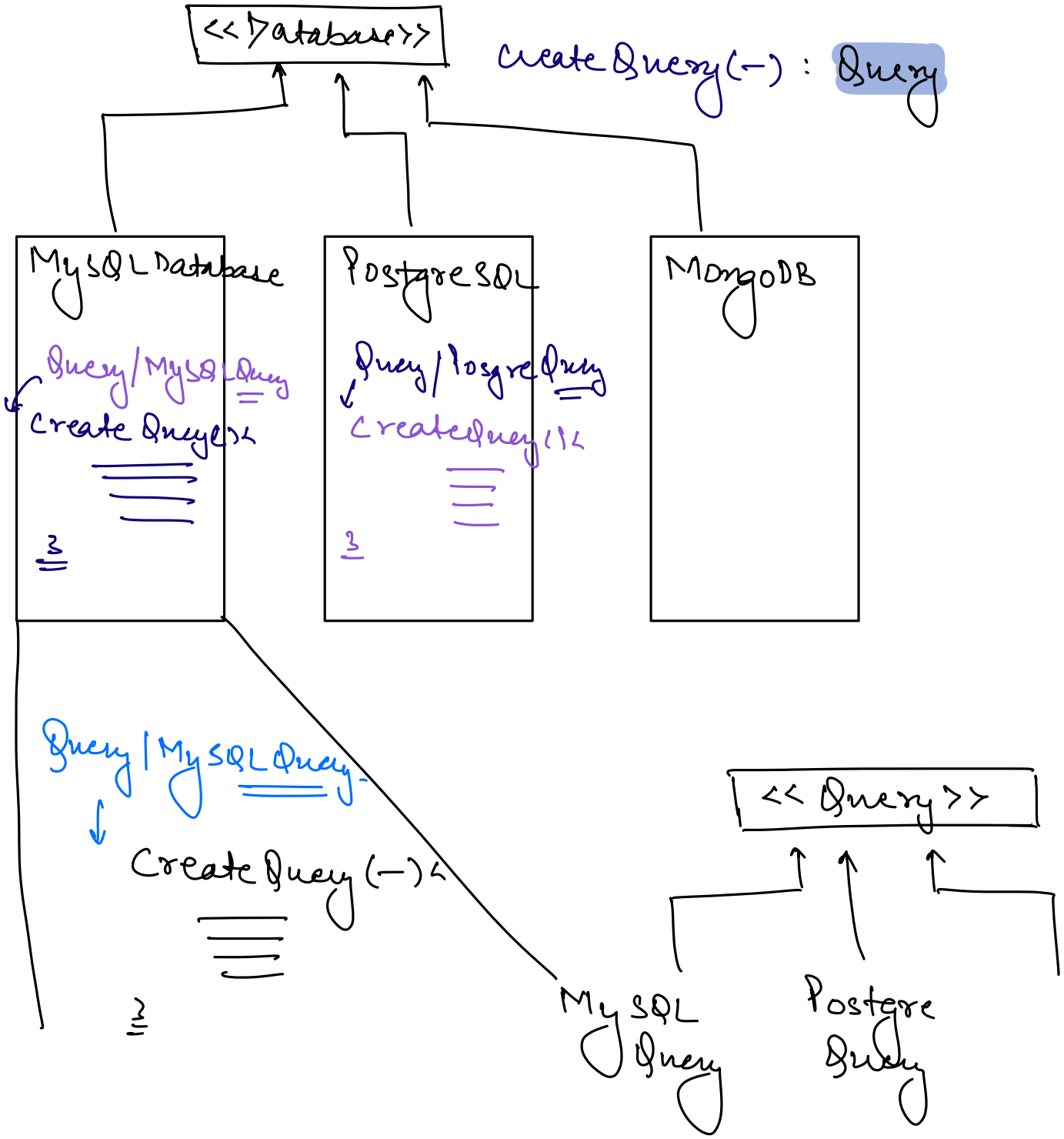
#

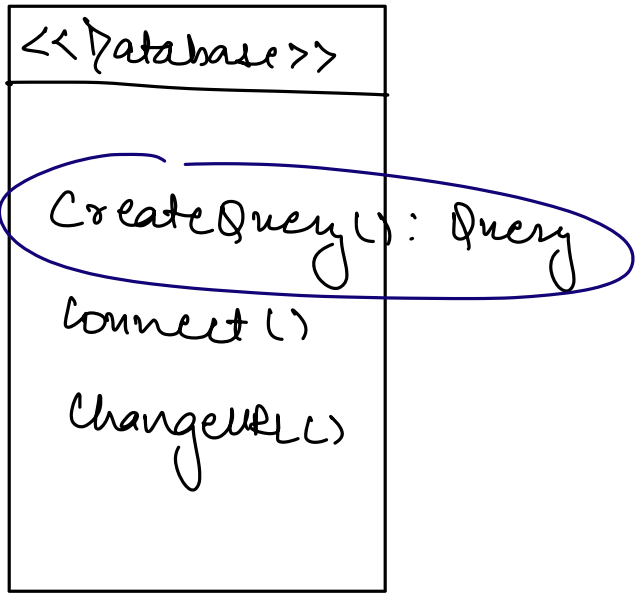
```
Class UserService {
    Database db;
    Query q = db.createQuery("—");
    q.execute();
}

deleteUser(—) {
}
```

MySQL.
PostgreSQL.
MongoDB.

⇒ If Database is a Normal class then it will violate Dependency Inversion Principle

⇒ Ideally Database should be an interface, so that it would be easier for us to change the underlying object in DB reference.


<<Database>>
createQuery(-) : Query
MySQL Database
PostgreSQL
MongoDB
Query/MySQLQuery
createQuery()<
Query/PostgreQuery
createQuery()<
Query/MySQLQuery
createQuery(-)<
<<Query>>
MySQL Query
Postgre Query



Purpose of Create Query Method.

=> It should return an object of corresponding Query.

=> Factory Method.

⇒ User Service {

Database db;

Query q;

createQuery() {

if (db is MySQL) {

q = mySQL query()

}

else if (db is Posgre) {

q = Postgre Query()

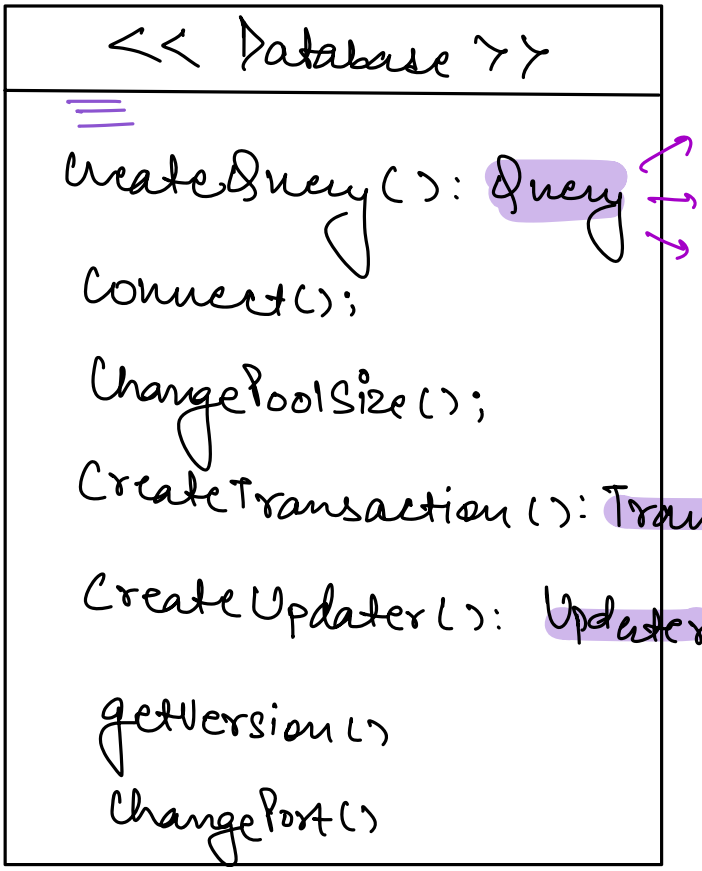
}

}

}

OCP / SRP.

⇒

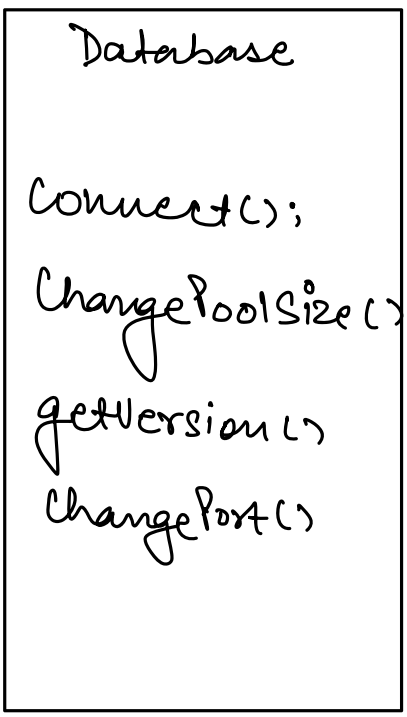


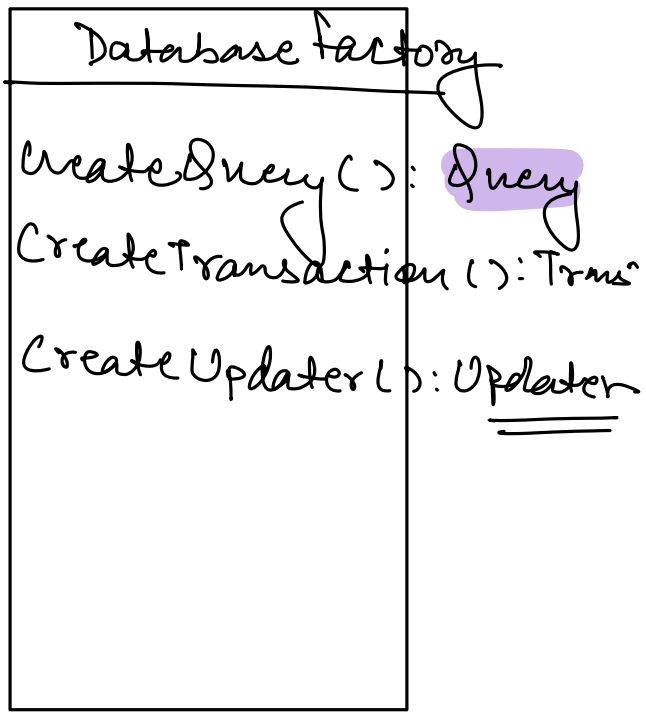
Responsibilities.

→ Attributes

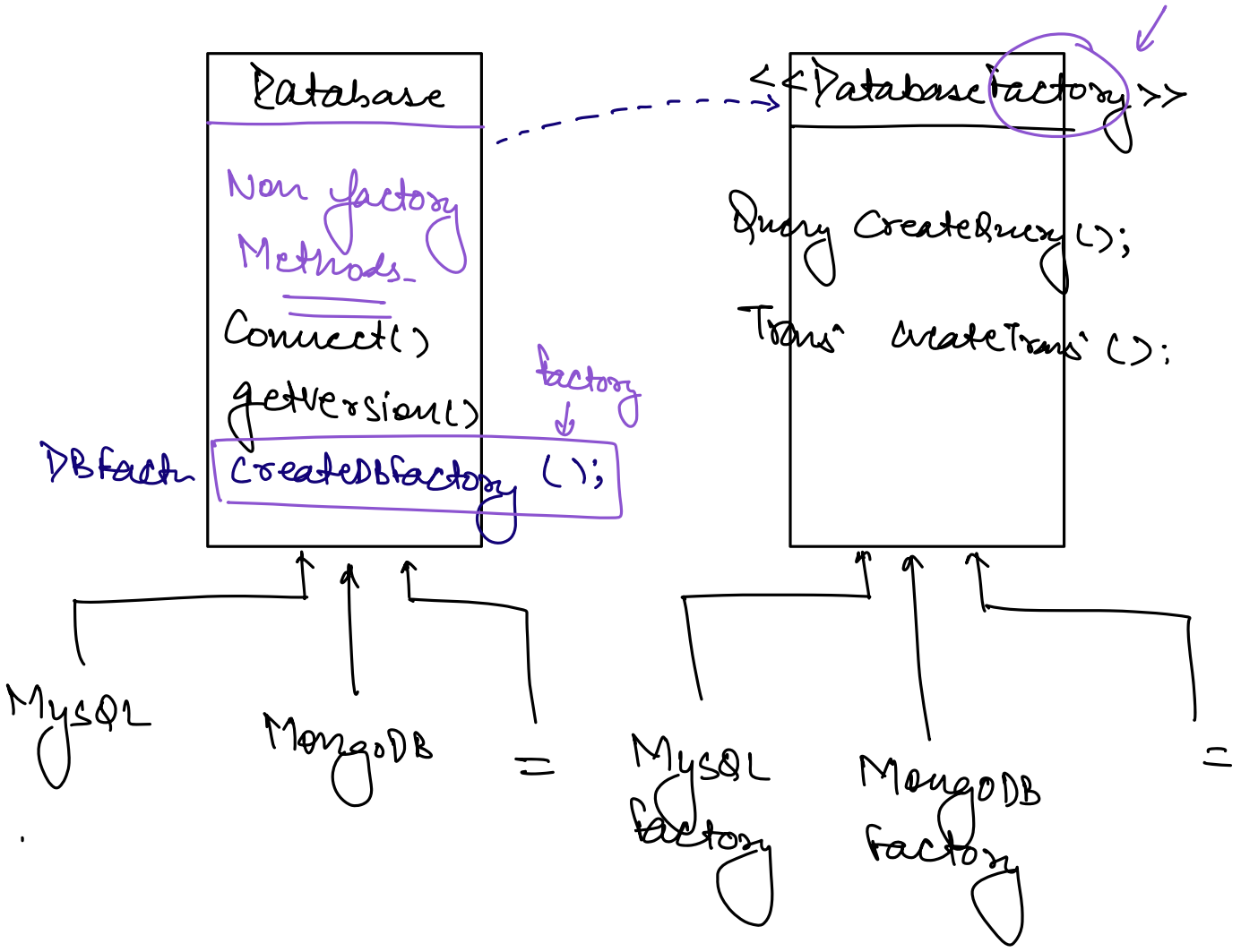
→ factory & New factory methods.

⇒ SRP is being violated.





⇒ Abstract factory.



UserService {

Database db; ← MySQL

DatabaseFactory dbf;

if (db is MySQL) {

dbf = MySQLfactory ();

}

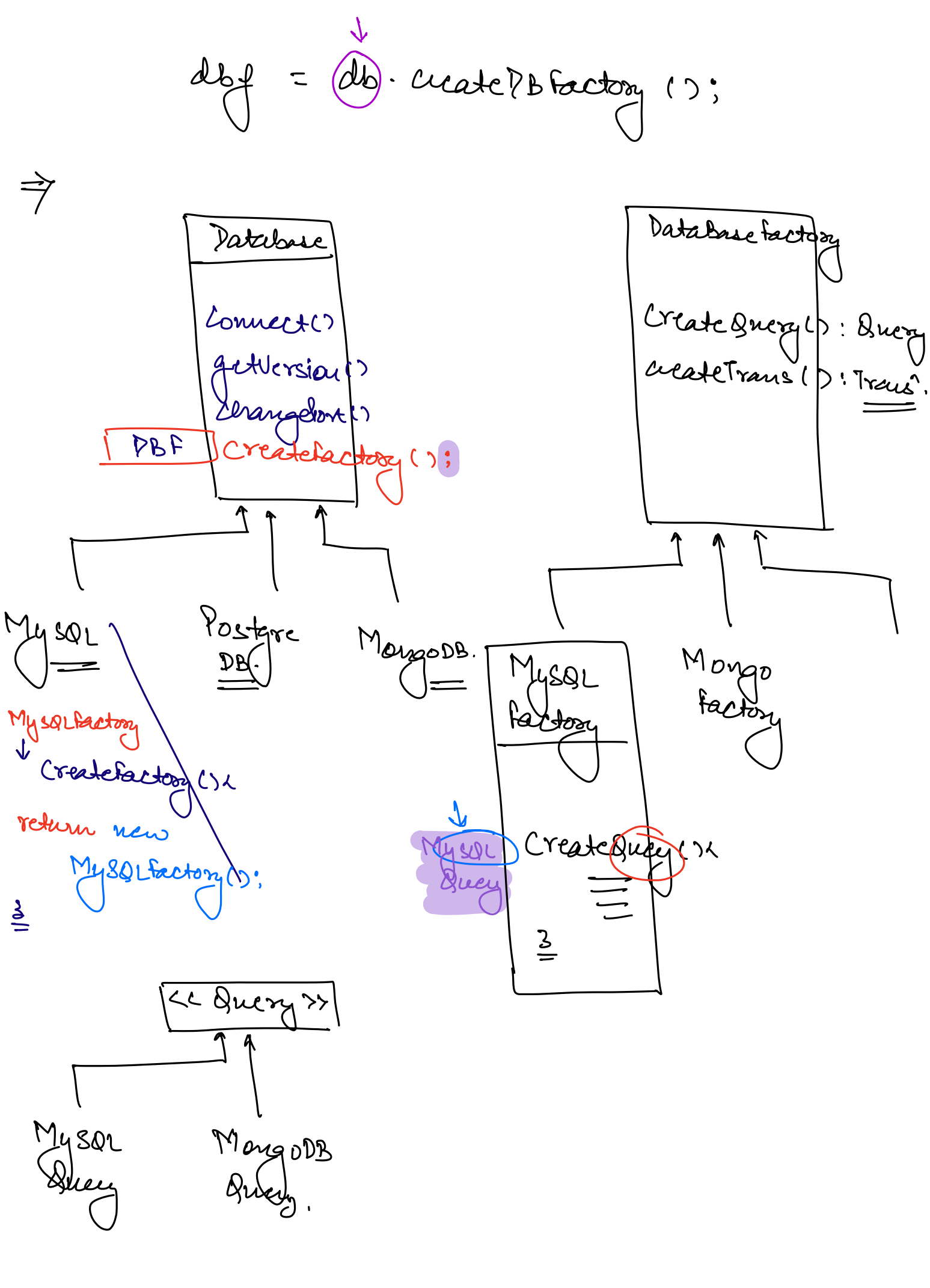
else if (db is MongoDB) {

dbf = MongoDBfactory ();

}

}

×


dbf = db.createDBFactory();
Database
Connect()
getVersion()
changeDbt()
DBF
CreateFactory();
DatabaseFactory
CreateQuery() : Query
createTrans() : Trans.
MySQL
Postgre DB.
MongoDB.
MySQLfactory
CreateFactory(){
return new
MySQLFactory();
}
MySQL factory
Mongo factory
MySQL Query
CreateQuery(){
}
<< Query >>
MySQL Query
MongoDB Query.

UserService {

Database db = new MySQLDB();

DatabaseFactory dbf = db.createFactory();

CreateUser (——) {

Query q = dbf.createQuery()

MySQLQuery.

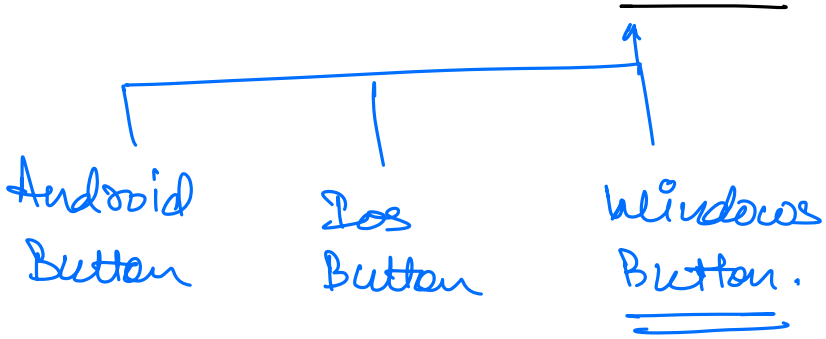
}

=

⇒ UI libraries.

Cross Platform frameworks

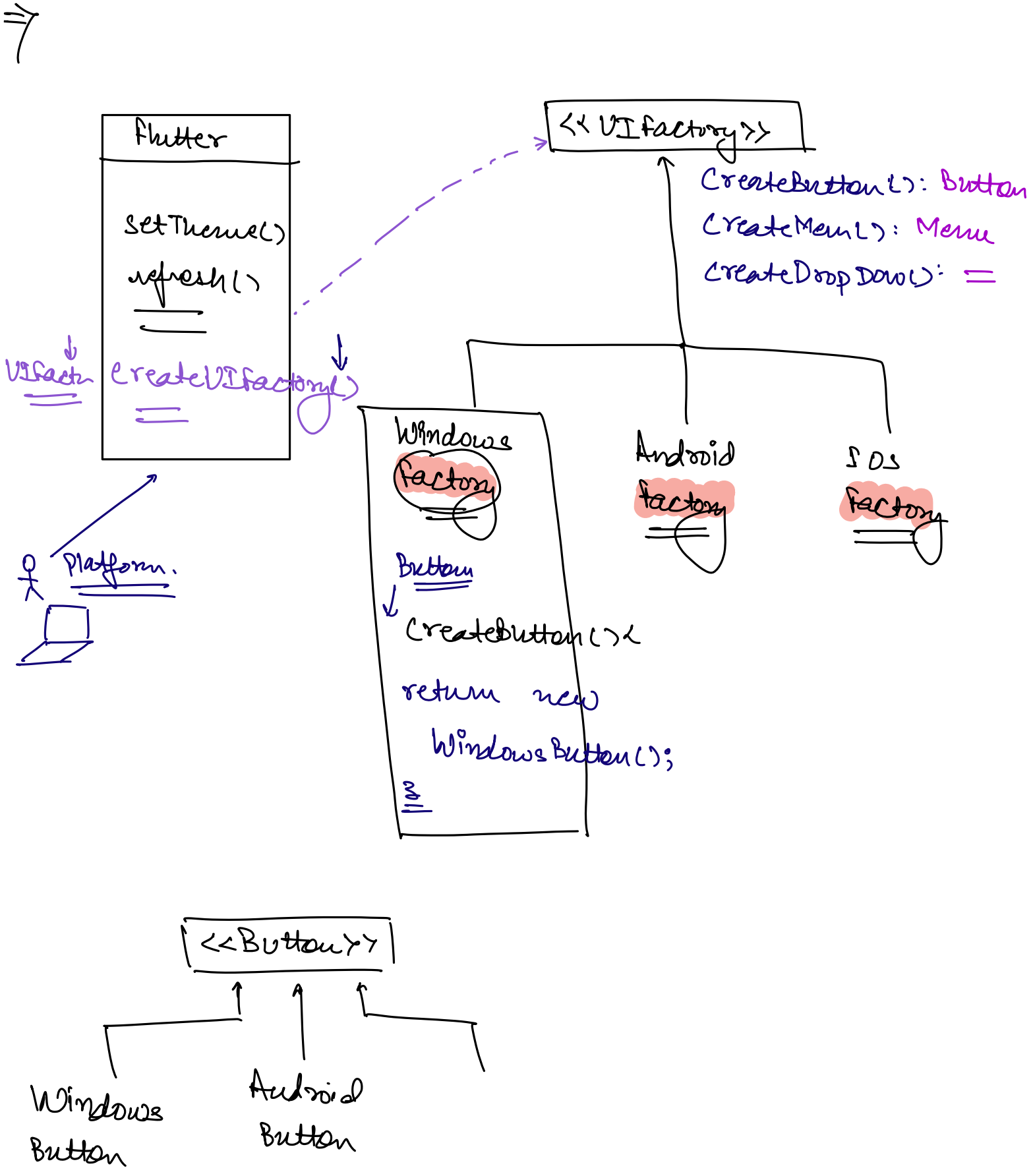
CreateButton() ⇒ Button



Flutter.

⇒ Flutter ← SRP / OCP. ✗

```
CreateButton() {
    if (Platform == "Android") {
        return AndroidButton()
    }
    if (Platform == "MacOs") {
        return MacOsButton()
    }
    ————
    ————
    ————
    ————
}
CreateMenu() {
    ————
    ————  if-else
    ————
}
CreateDropDown() {
    ————
    ————
    ————
    ————
}
}
```


Flutter
setTheme()
refresh()
VIfactory
createVIfactory()
<< VIfactory >>
CreateButton(): Button
CreateMenu(): Menu
CreateDropDown():
Windows factory
Android factory
IOS factory
Button
CreateButton(){
return new
WindowsButton();
}
Platform.
<<Button>>
Windows Button
Android Button

Factory: Anything that allows us to create the object of corresponding types.

Abstract factory: When we have lot of factory methods, we move all the factory methods to a new interface.

Practical factory

Move the logic of factory object creations to a new class.

→ Structural Design Patterns.

- → Adapter.
- → Facade.

Structural Design Patterns.

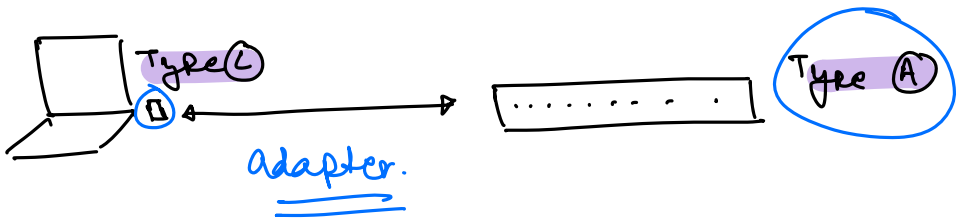
⇒ How to structure your codebase.

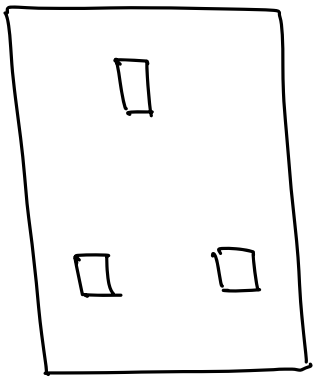
- → What all the classes you should have in the codebase.
- → How different classes are going to talk to each other.

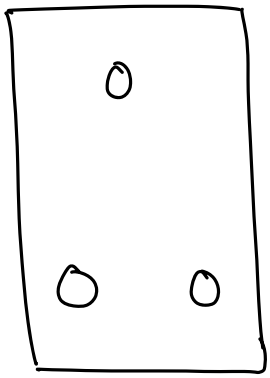
ADAPTER DESIGN PATTERN.

Power adapter.

Type C adapter for Macbook.







⇒ Adapter.

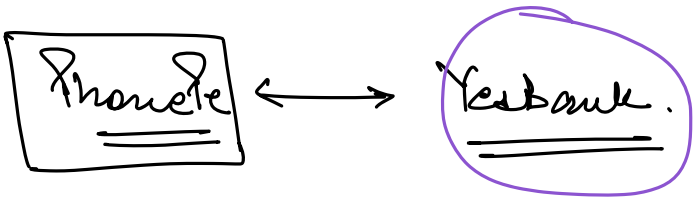
Intermediary layer that connects / transforms one form into another form.

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.

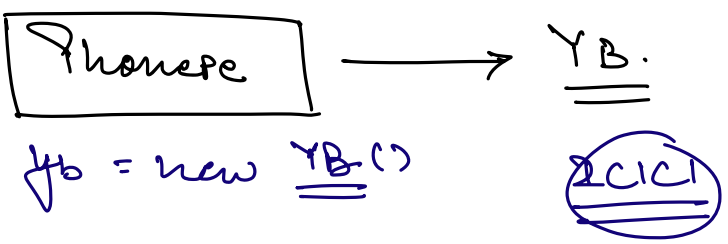
⇒ AWS / Cloud service.

⇒ Payment GW.



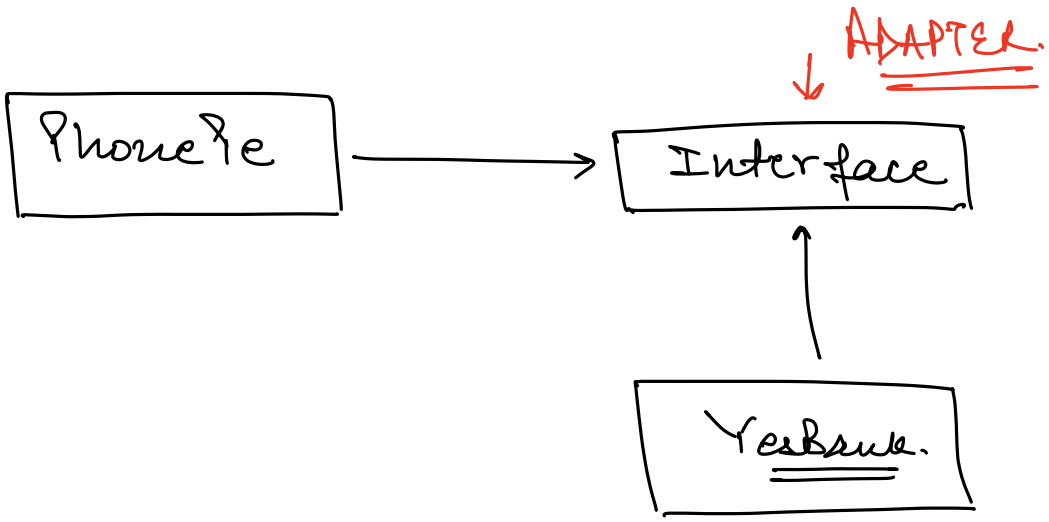
Problem Statement:

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



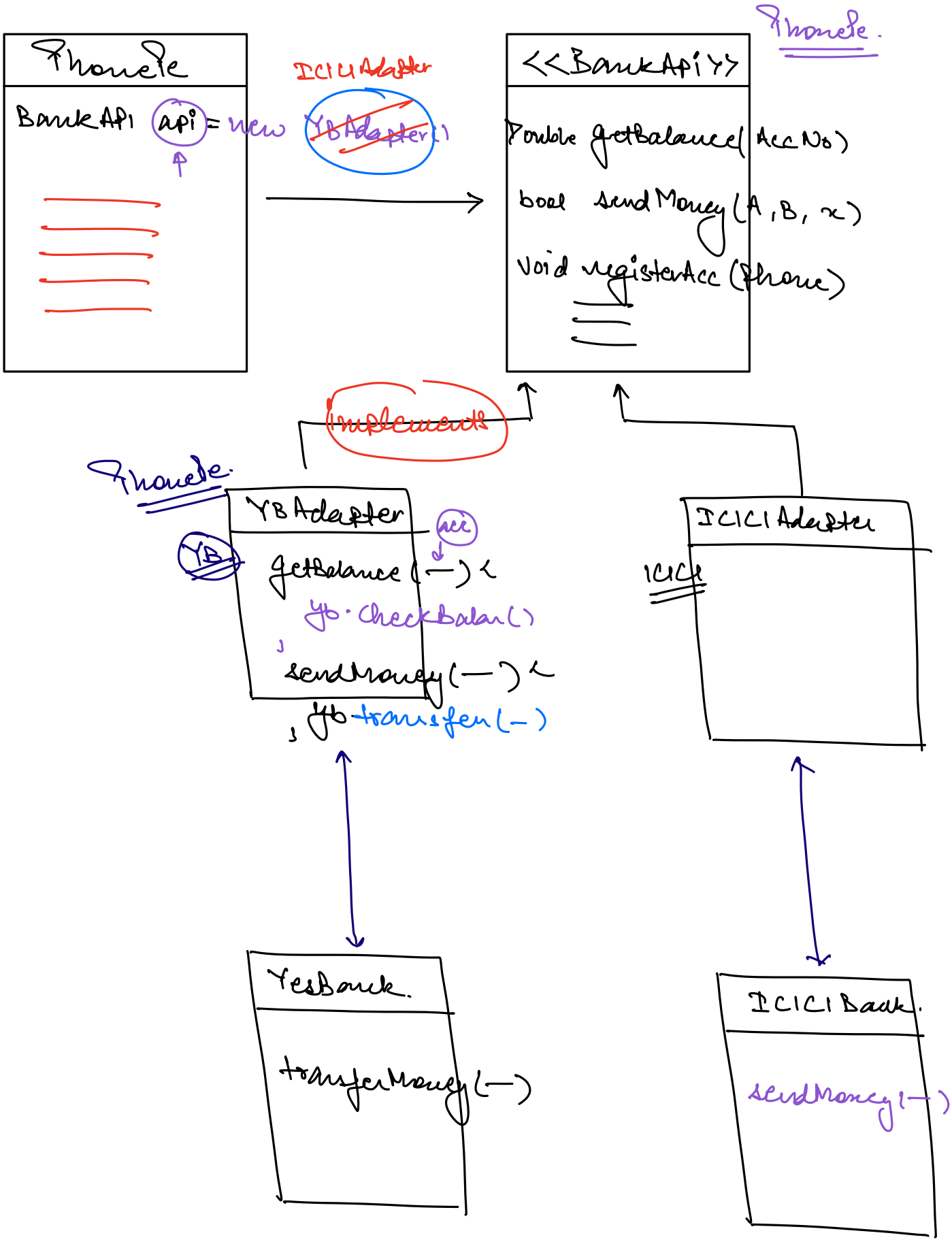
⇒ If your codebase is directly talking to 3rd party API's, it involves tight coupling b/w your codebase & 3rd party system. This affects maintainability of your codebase.

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

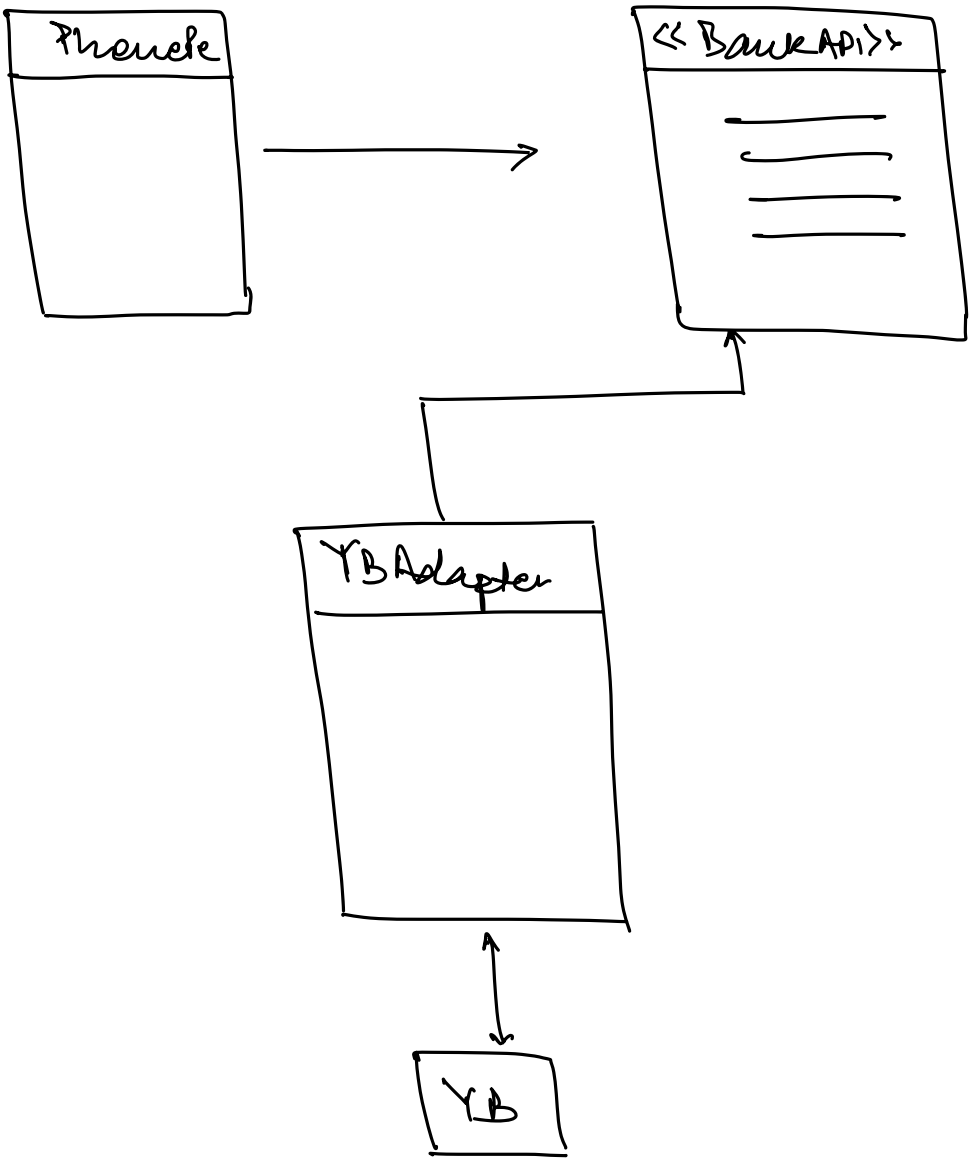


⇒ How to use Adapter Pattern.

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.

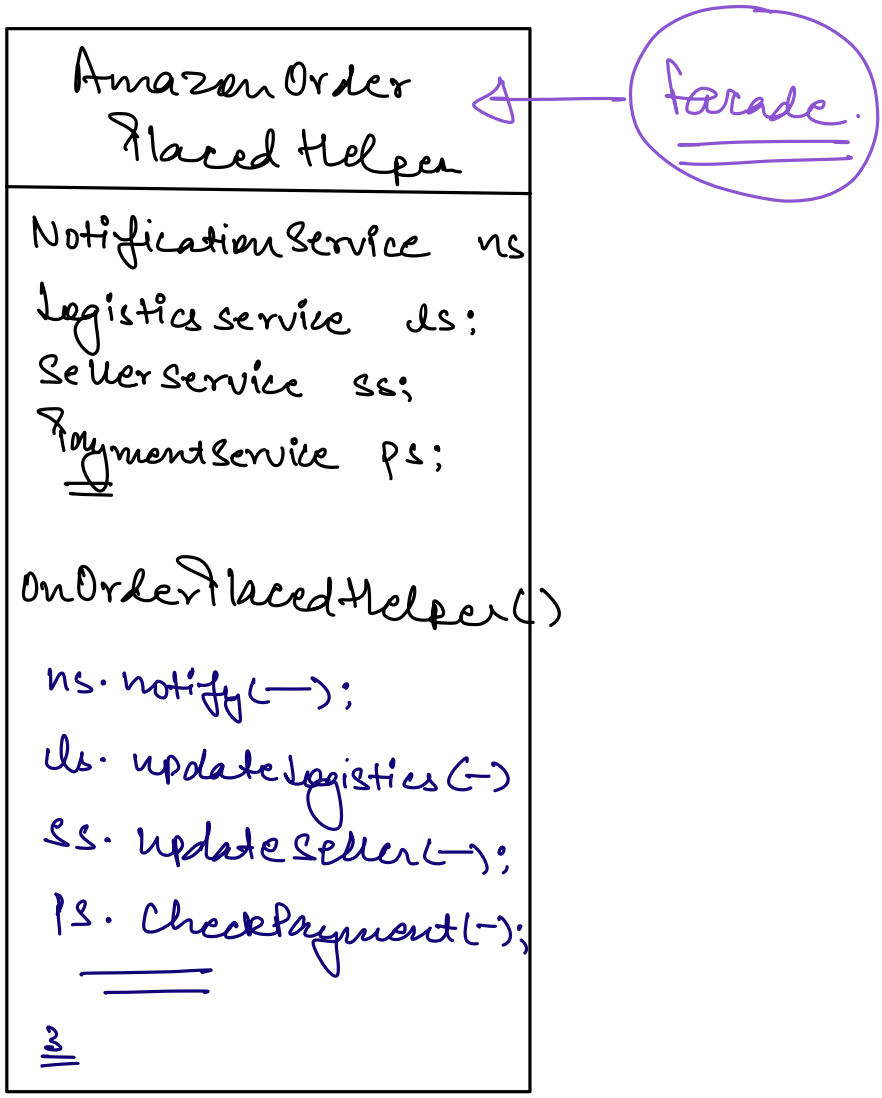
↓

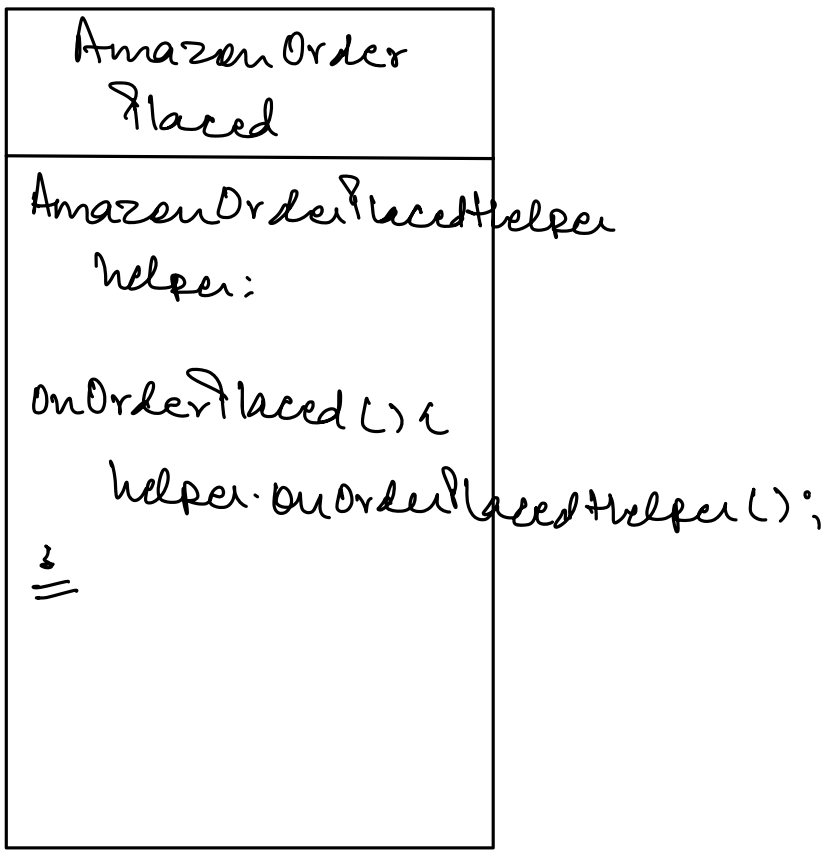
front | Boundary of building

⇒ 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(—);
}
 	
}

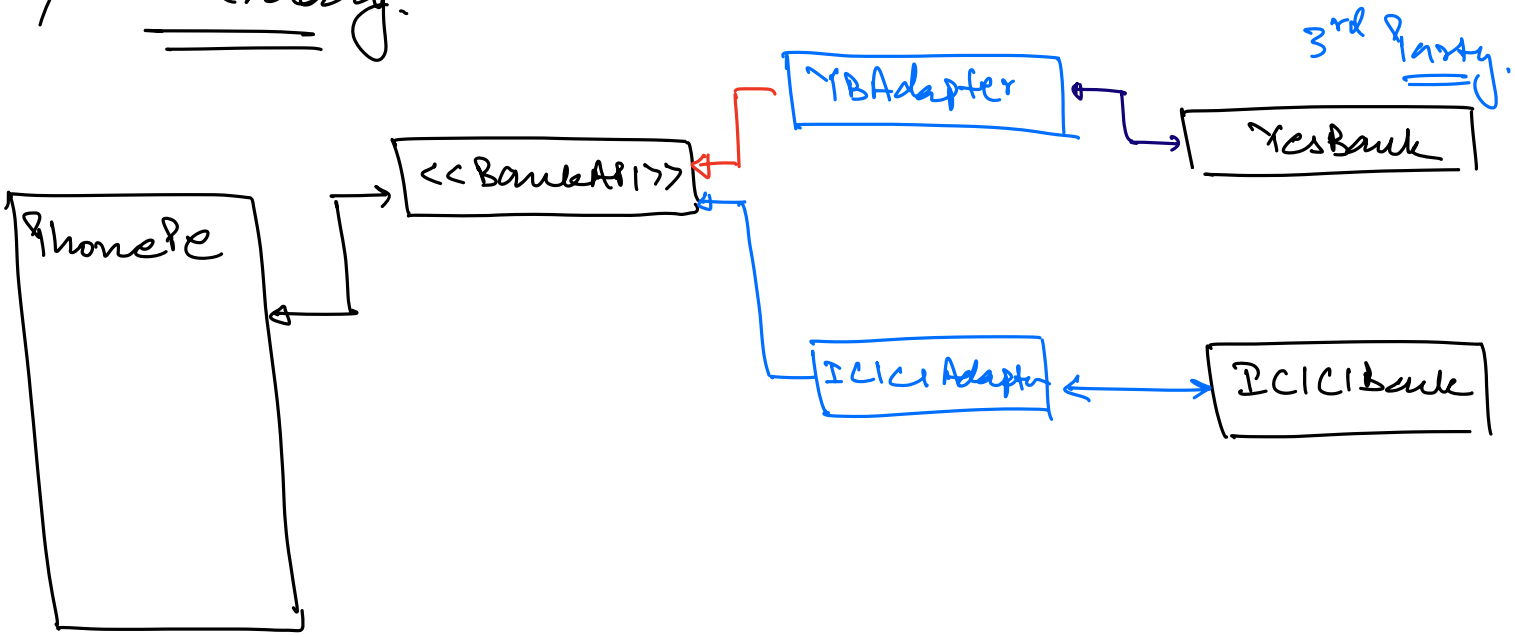

AmazonOrder PlacedHelper
NotificationService ns
LogisticsService ls;
SellerService ss;
PaymentService ps;
OnOrderPlacedHelper()
ns.notify(—);
ls.updateLogistics(—)
ss.updateSeller(—);
ps.CheckPayment(—);
}
facade.


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.

Agenda.

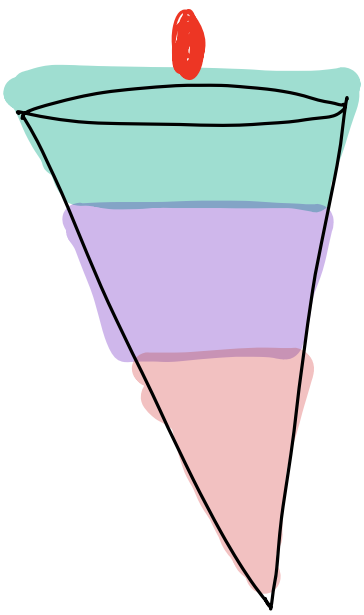
- Decorator
- Flyweight.

⇒ DECORATOR PATTERN.

↳ Icecream Parlour. | Pizza | Coffee M/C.

⇒ ICECREAM.

↳ Custom ice creams.



→ App^n that takes orders for ice creams.

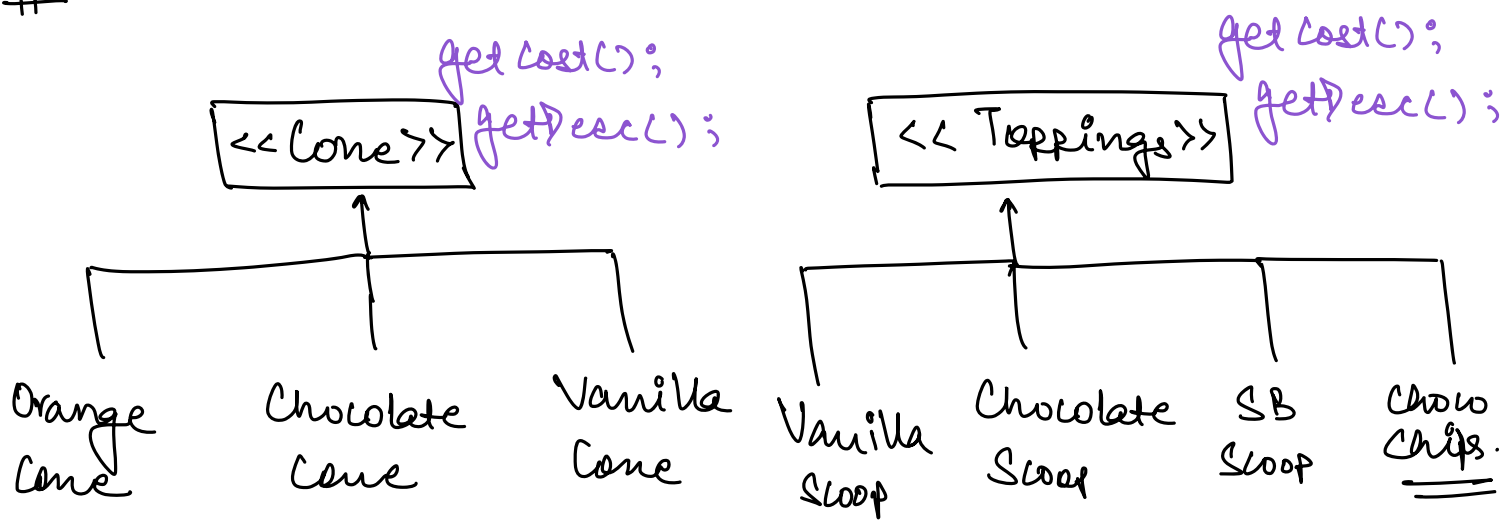
→ Custom config.

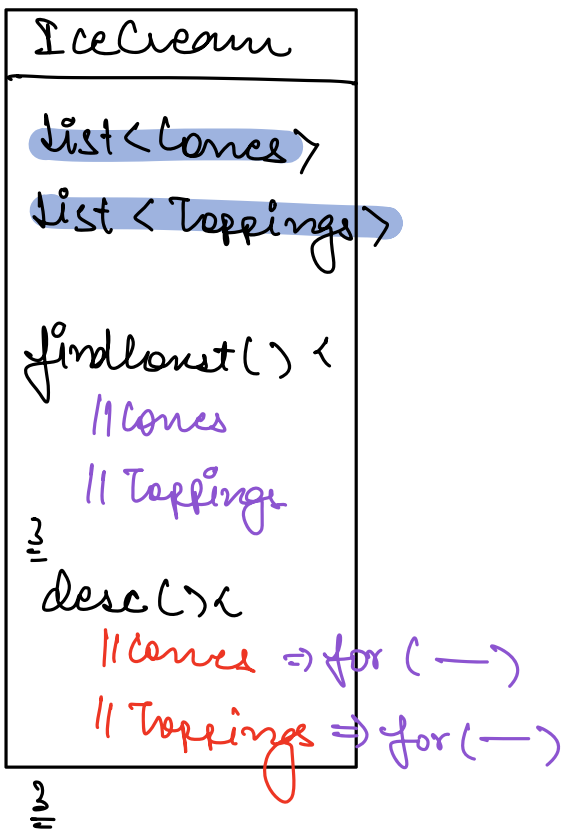
⇒ Build ice cream.

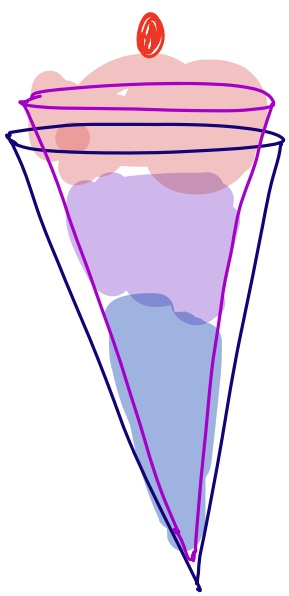
⇒ get the cost of the ice cream.

⇒ get Description of the ice cream.

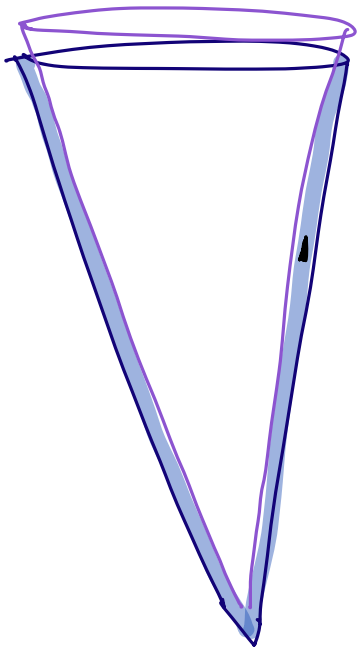
#


<<Cone>>
getCost();
getDesc();
Orange Cone
Chocolate Cone
Vanilla Cone
<< Toppings>>
getCost();
getDesc();
Vanilla Scoop
Chocolate Scoop
SB Scoop
Choco Chips.


IceCream
List<Cones>
List<Toppings>
findCost(){
// Cones
// Toppings
}
desc(){
// Cones => for(—)
// Toppings => for(—)
}

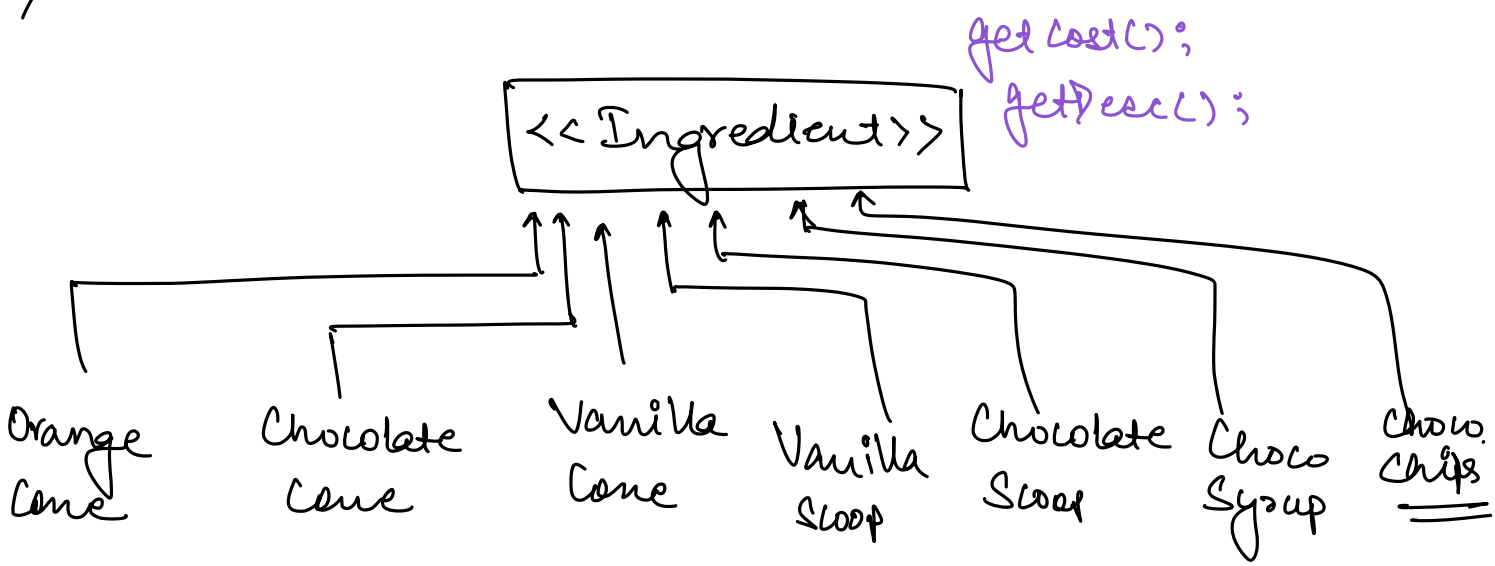


⇒

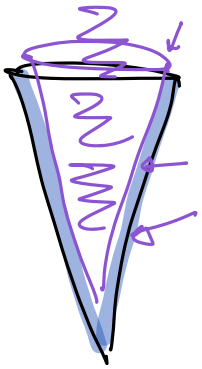


⇒ Sequence of ingredients can't be manged.

⇒



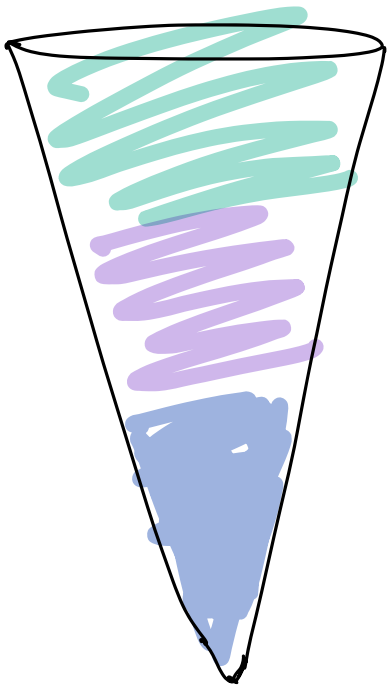

```
getCost(){
    return 20;
}

getDesc(){
    return "Choco
            Cone"
}
```



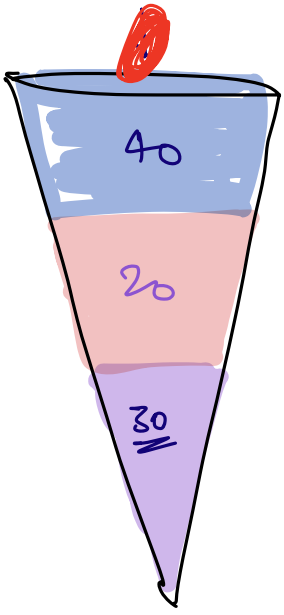
```
IceCream
-----------------------------
List<Ingredient>

getCost(){
    for(ingredients){
        cost += ing.getCost;
    }
}
getDesc(){
    for(ingredients){
        desc += ing.getDesc();
    }
}
```

```
ingredients.add(orangeCone)
ingredients.add(ChocoSyrup);
ingredients.add(ChocoCone);
```

Decorator.



⇒ Right from the beginning we had an icecream, & then we can keep on adding multiple ingredients on top of it.



⇒ 20 Rs. + 30 Rs. + 20 + 40 + 5

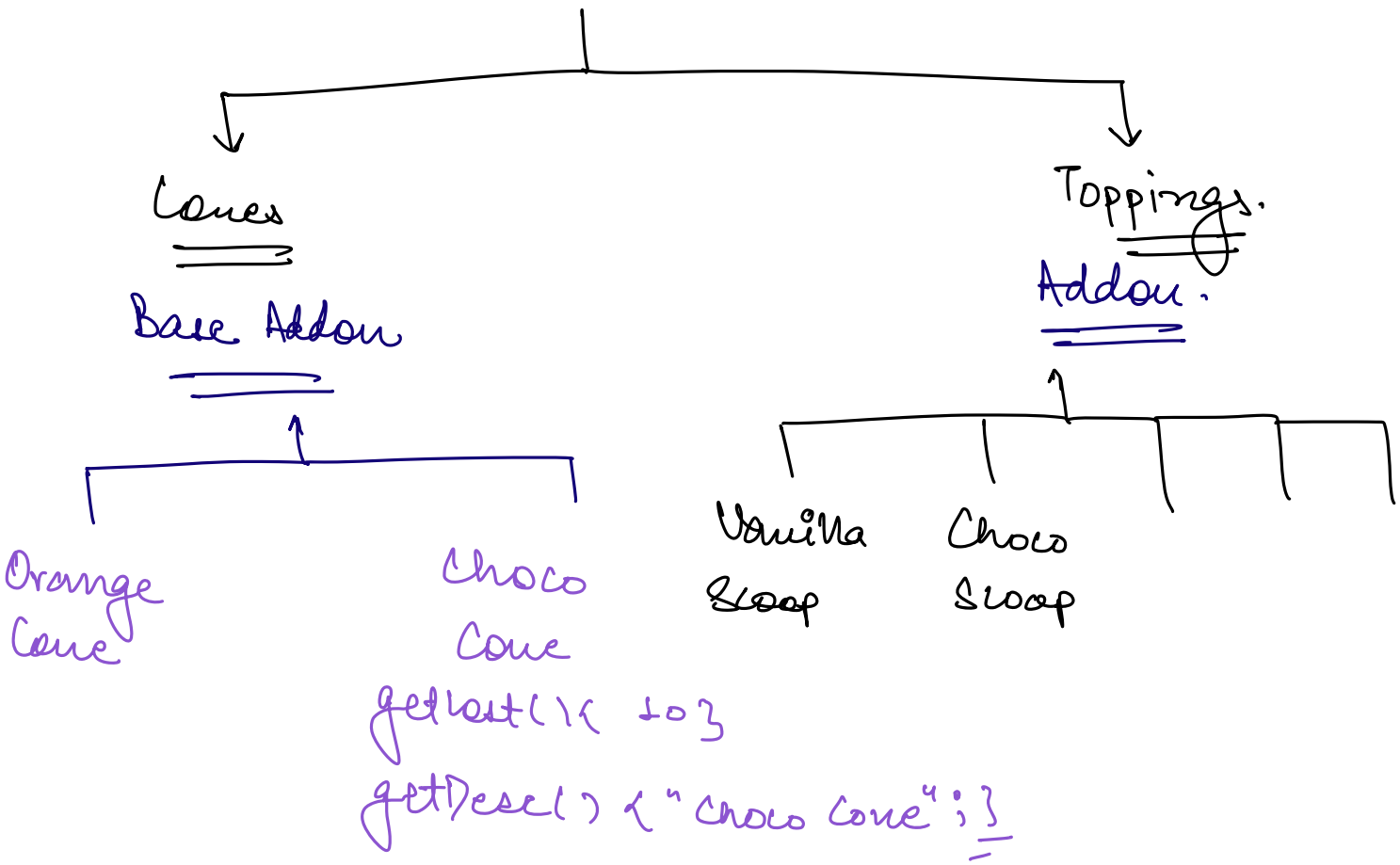
"Cone" + "Scoop-1" + "SB Scoop"

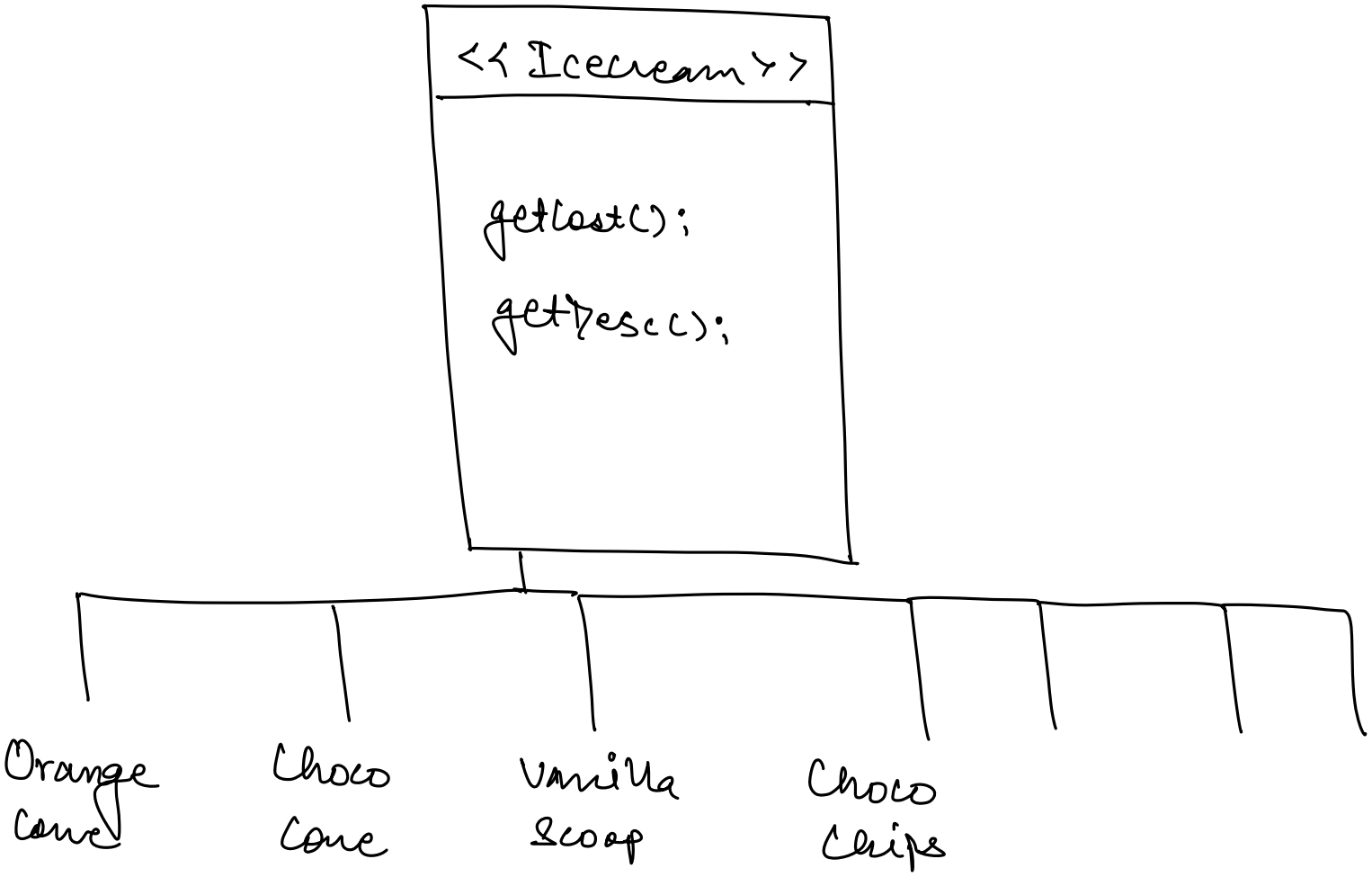
+ "BB Scoop" + "Cherry"

Step 1 :

Creat an interface that is going to represent the entity that we are constructing.

Step 2 : There are 2 kind of ingredients.






```
getCost(){
    10;
}
getDesc(){
    "OC"
}
```

```
getCost(){
    20.
}
getDesc(){
    "CC"
}
```

```
getCost(){
    30.
}
getDesc(){
    "VS"
}
```

. . . . .

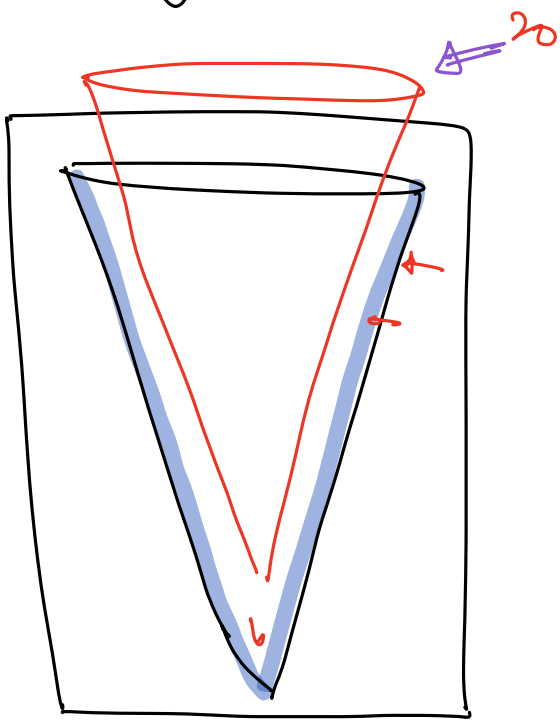
Base Addon ⇒ can be the first item in the icecream (or) can also be added as an addon on an existing icecream.

① Base Addon is the first component in the DE.

getCost() ⇒ Cost of the item itself.

getDesc ⇒ Name of the item itself.

② Base Addon, we are adding on top of an existing icecream.



cost = getCost() + 20.

Desc = getDesc() + "OC".

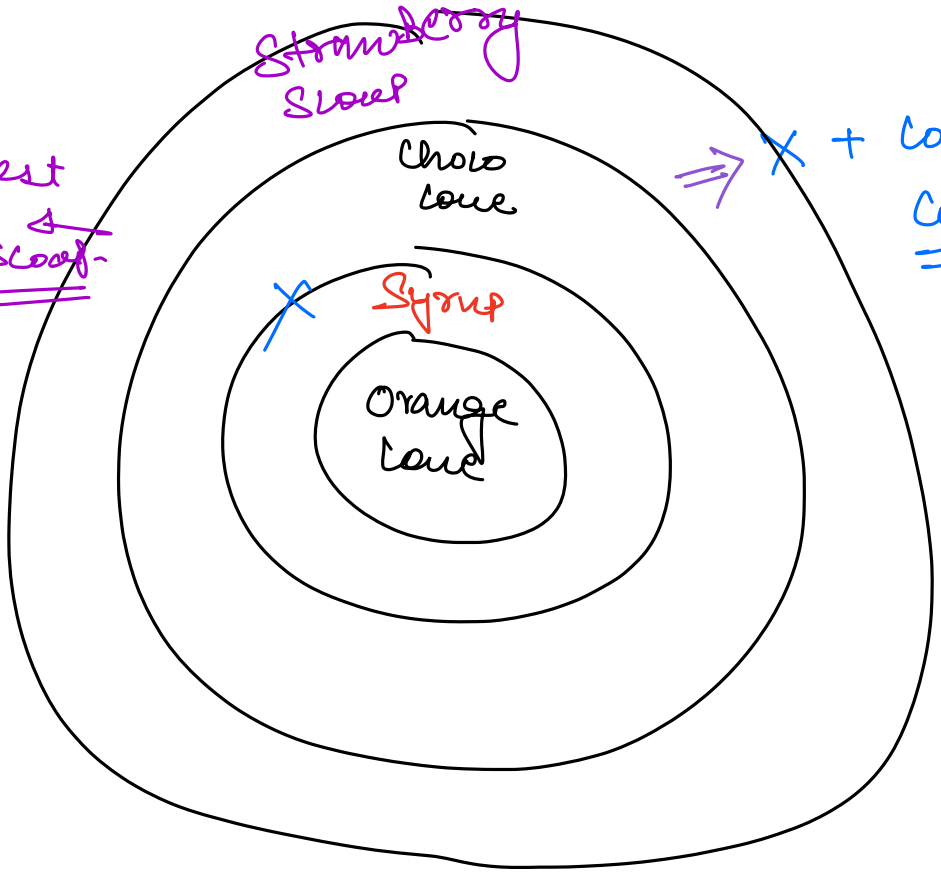
Addon.

we are adding on top of an existing icecream.

cost = getCost() + 20.

Desc = getDesc() + "OC".

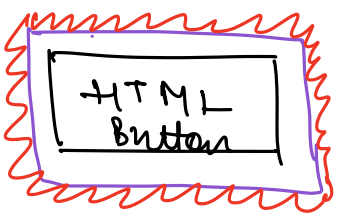
⇒



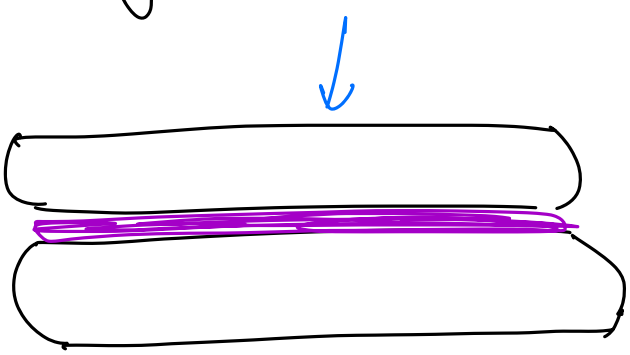
Y + cost of SB Scoop.

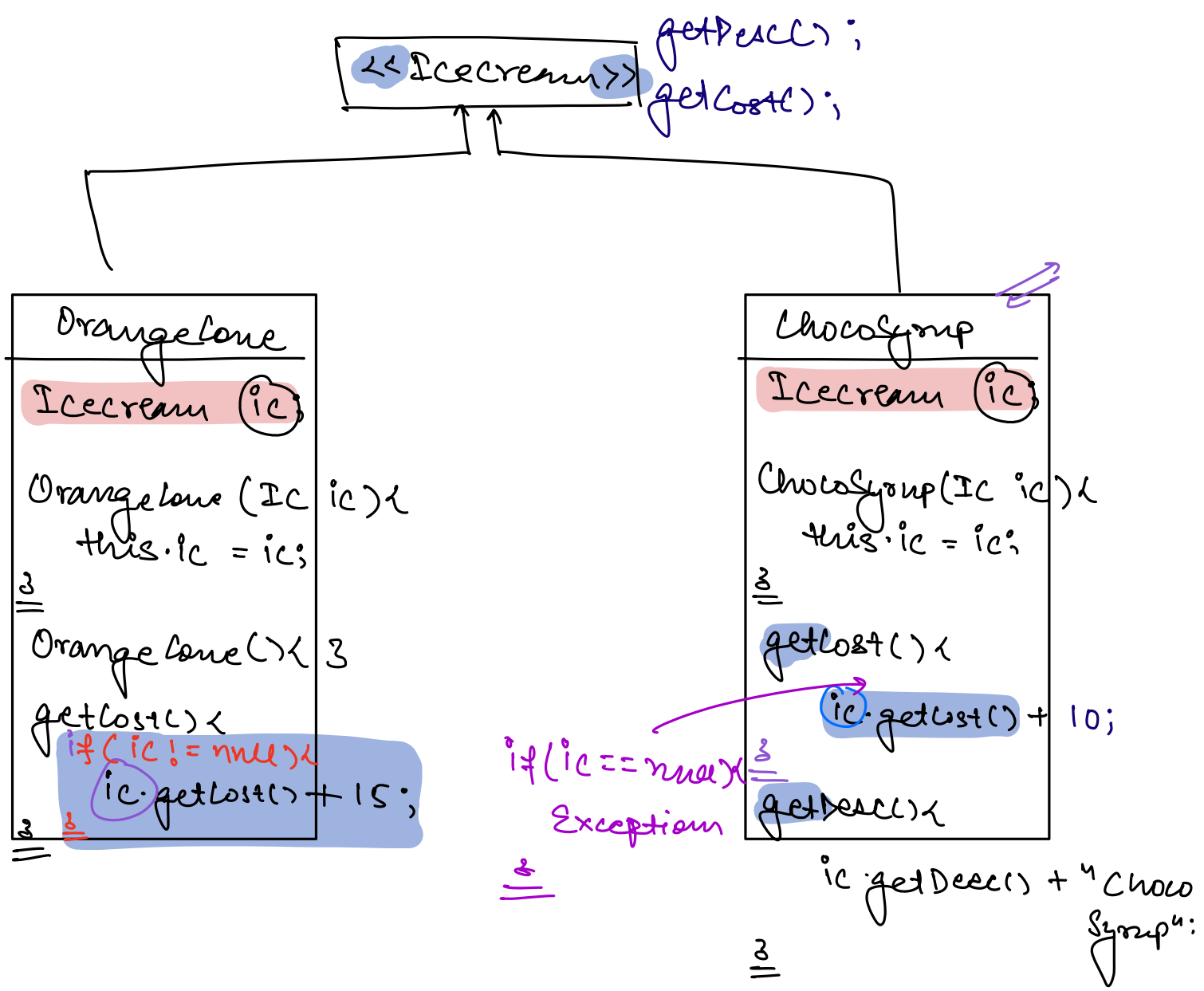
⇒ X + cost of Choco Cone. = (Y)

⇒

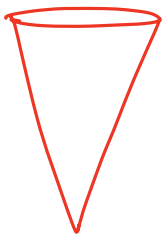


⇒ Pizza | Subway






```
Icecream ic =
        new VanillaScoop(
            new ChocoLove(
                new ChocoSyrup(
                    new OrangeLove()
                )
            )
        );
```



⇒

ic.getCost() ⇒ 95

↳ VanillaScoop.getCost()

↓

ChocoCone.getCost() + 50;

↓

ChocoSyrup.getCost + 25;

↓

OC.getCost() + 15

↓

10

IceCream ic = new OrangeCone();

ic = new ChocoSyrup(ic)

ic = new ChocoCone(ic);

ic = new VanillaScoop(ic);

Decorator-

⇒ When we want to keep adding the properties at run time.

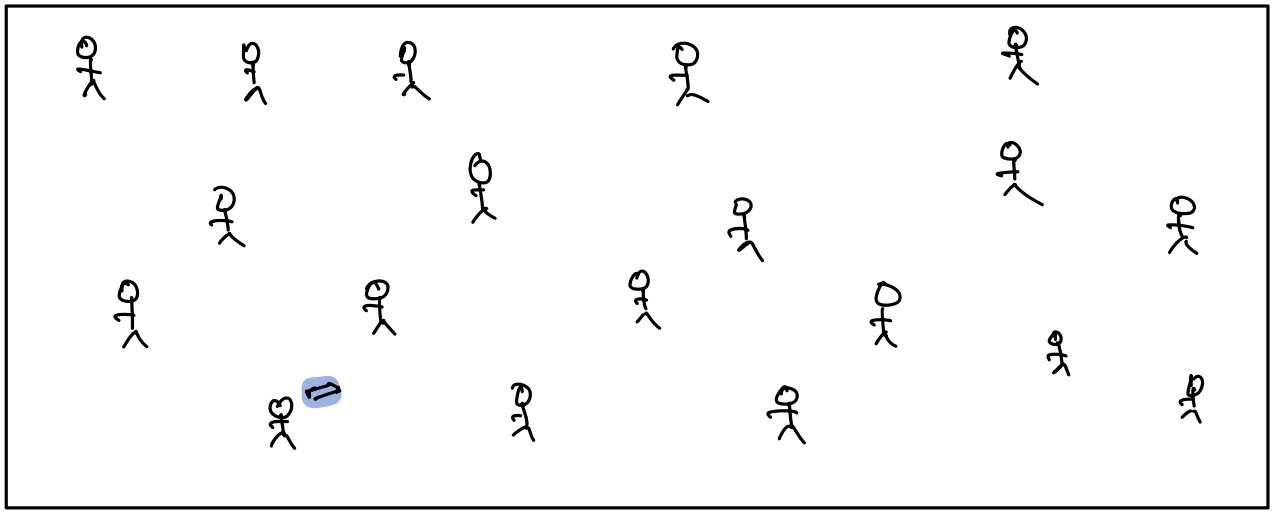
FLYWEIGHT.

⇒ Building some game application.

↓

Multiplayer.

Eg: Pubg.

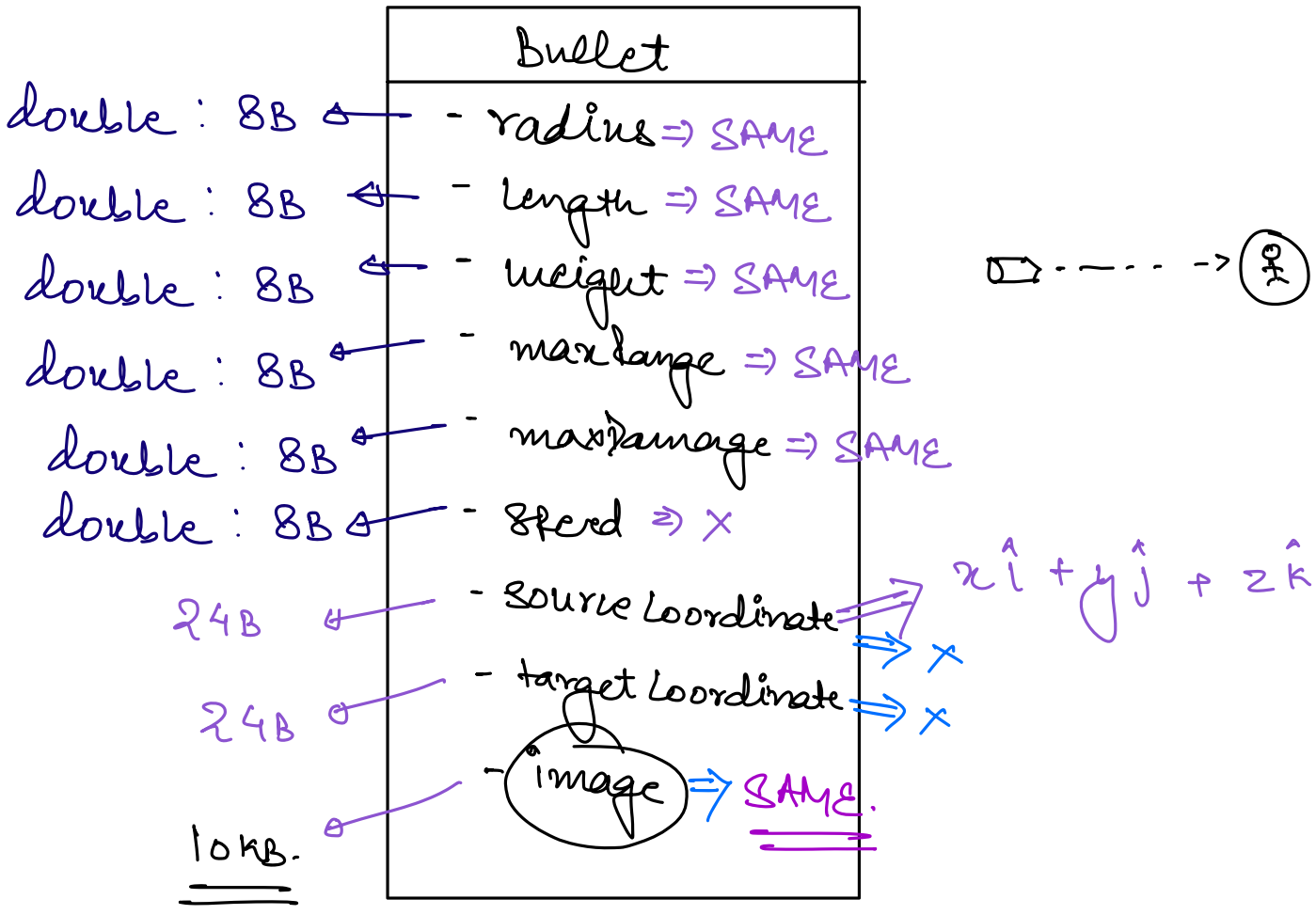


⇒ At the start, game loads the data for each player.

⇒ Complete state of the game is downloaded at start time & after that all the events will be transferred to other players

100 player × 10^3 Bullets.

⇒ 100,000 bullets.



Size of 1 Bullet obj = 96B + 10KB

≈ 10KB

100,000 Bullets ≈ 10KB × 10^5

10×10^5 KB

10×10^2 MB.

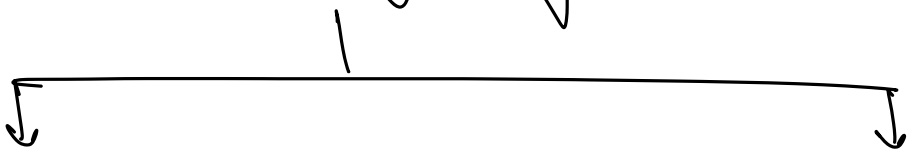
1GB.

⇒ 10 type of Bullets.

1 type ⇒ 10,000

⇒ Bullet Objects have lot of common attributes

⇒ There are 2 type of properties.



⇒ Same for all the object

⇒ Not same for all the object.

Flyweight.

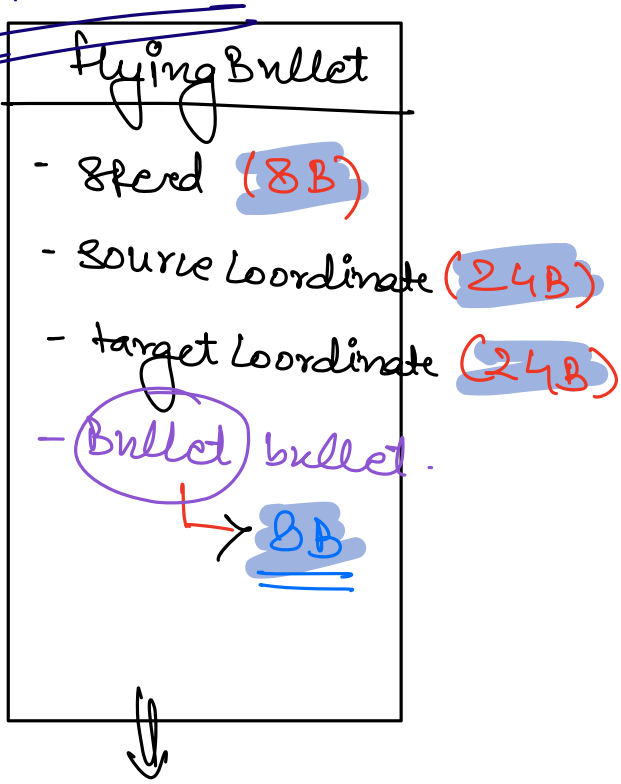
If we have a class whose objects contains both intrinsic & extrinsic properties & lot of memory is being consumed because of this, consider using flyweight.

⇒ Divide the class into (2)

→ Class with only intrinsic

→ Class with only extrinsic.

Extrinsic.



10^5 Bullets

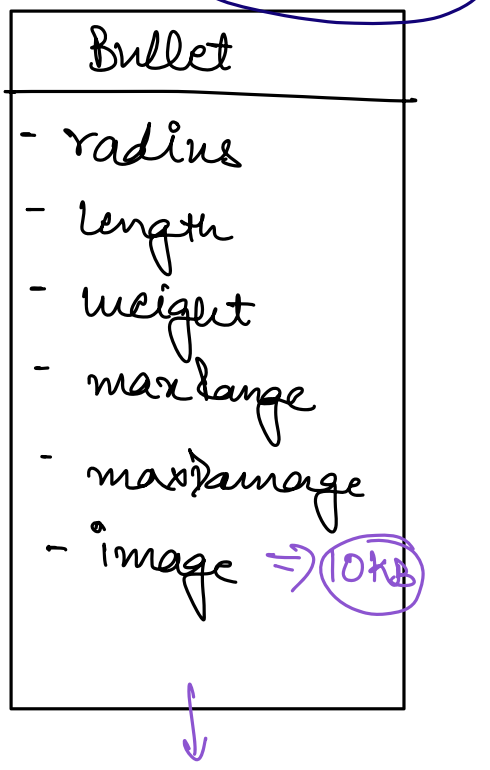
Size of 1 Obj = 64B.

total size = $64B \times 10^5$

$= 64 \times 10^2$ KB.

$= 6.4$ MB.

intrinsic



1 object.

Size = 10 KB.

Total space = 6.4 MB + 10 KB.

≈ 7 MB.

*

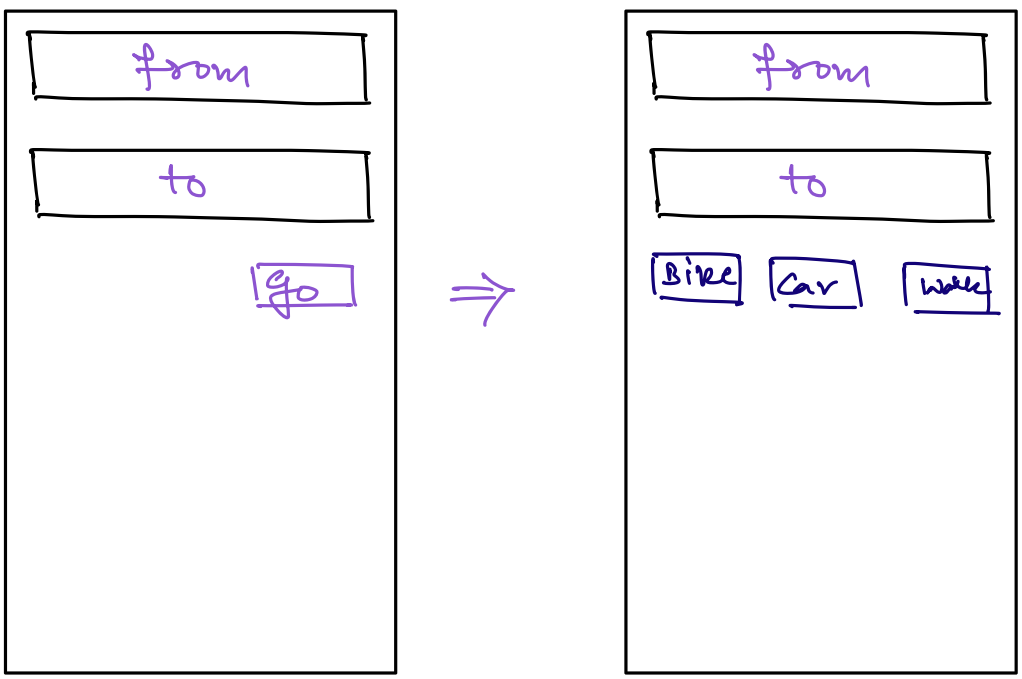
Behavioural Design Patterns.

↳ Actions / Methods.

⇒ There are some special kind of behaviours that needs to be implemented for an entity.

1. Strategy
2. Observer.

STRATEGY.



⇒ When we search path from point (A) to (B) on Google Maps, it suggests us different path based on mode of transportation we are using.

⇒ GoogleMap {

findPath (from, to, mode) {

SRP ×

OCP ×

if (mode == Car) {

CPC = new CPC();

cpc.findPath(—);

}

else if (mode == Bike) {

BPC = new BPC();

bpc.findPath();

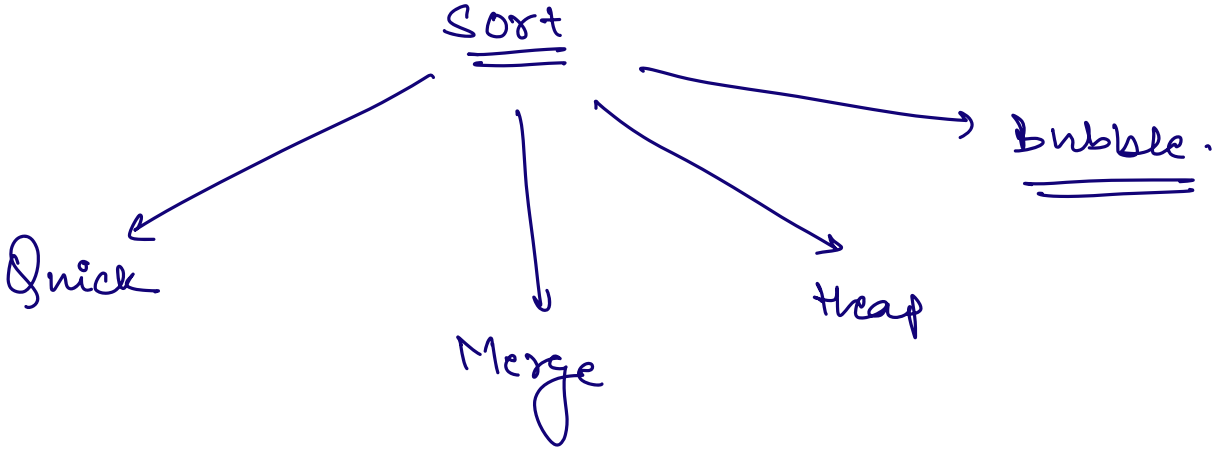
}

else if (mode == —) {

}

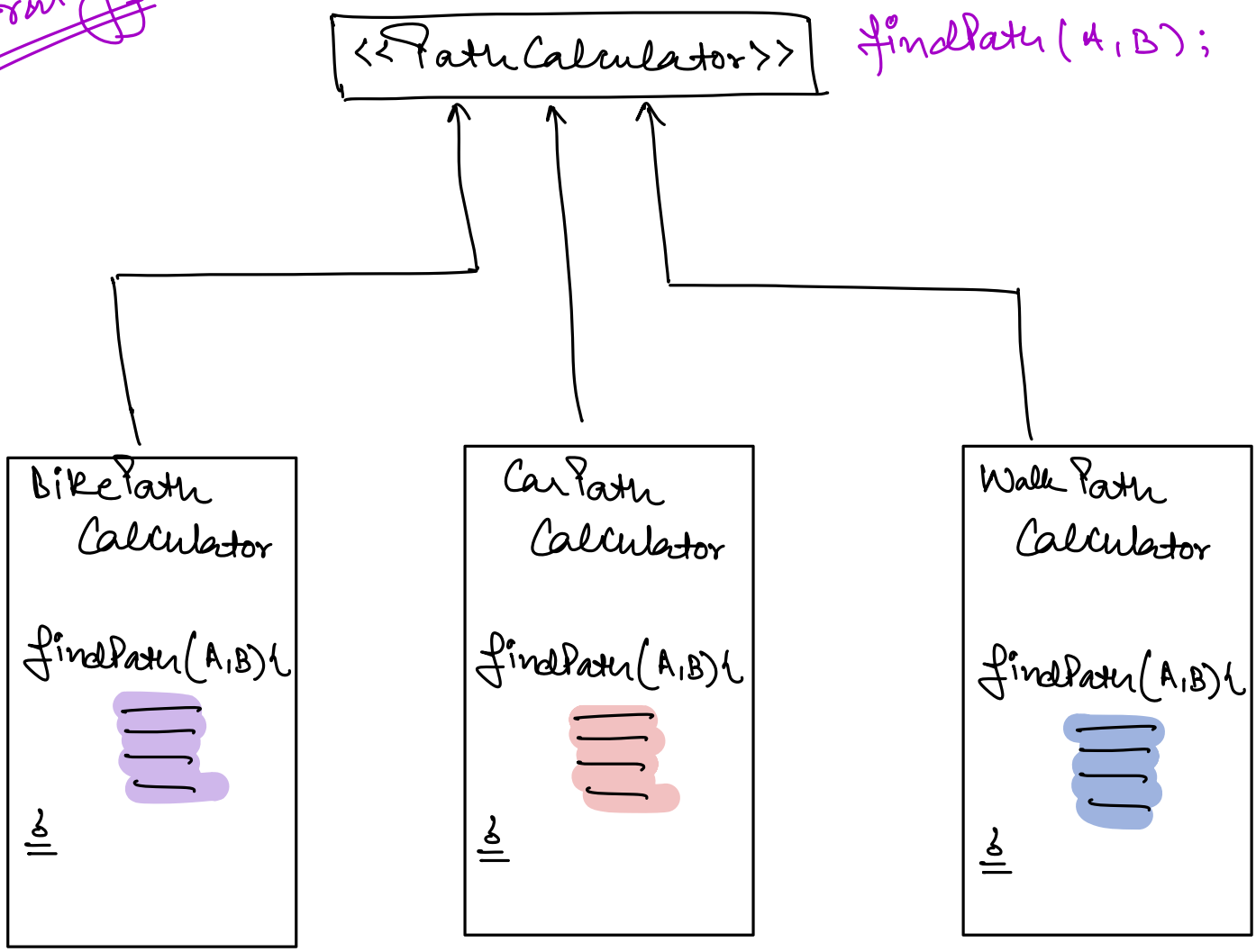
}

⇒ When there are multiple ways of doing something.



⇒ Often we see violation of SRP | OCP if there multiple ways of implementing a method because of multiple if-else conditions.

Strategy



```
GoogleMap
-----------------------------------
PathCalculator PC;

findPath(A, B, mode){
  if(mode == Car){
      PC = new CPC();
  }
  else if (mode == "Bike"){
      PC = new BPC();
  }
}
```

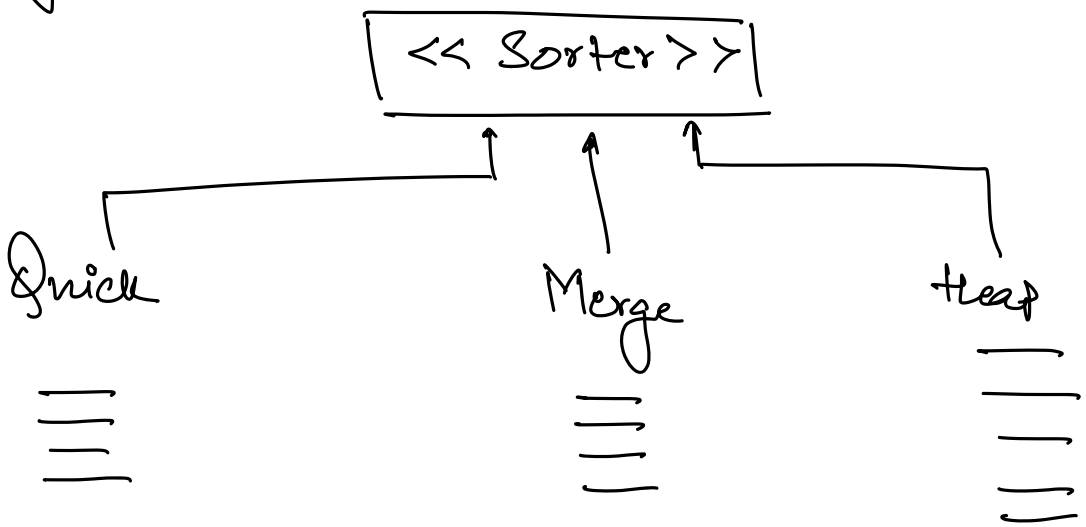
OCP ×

SRP ×

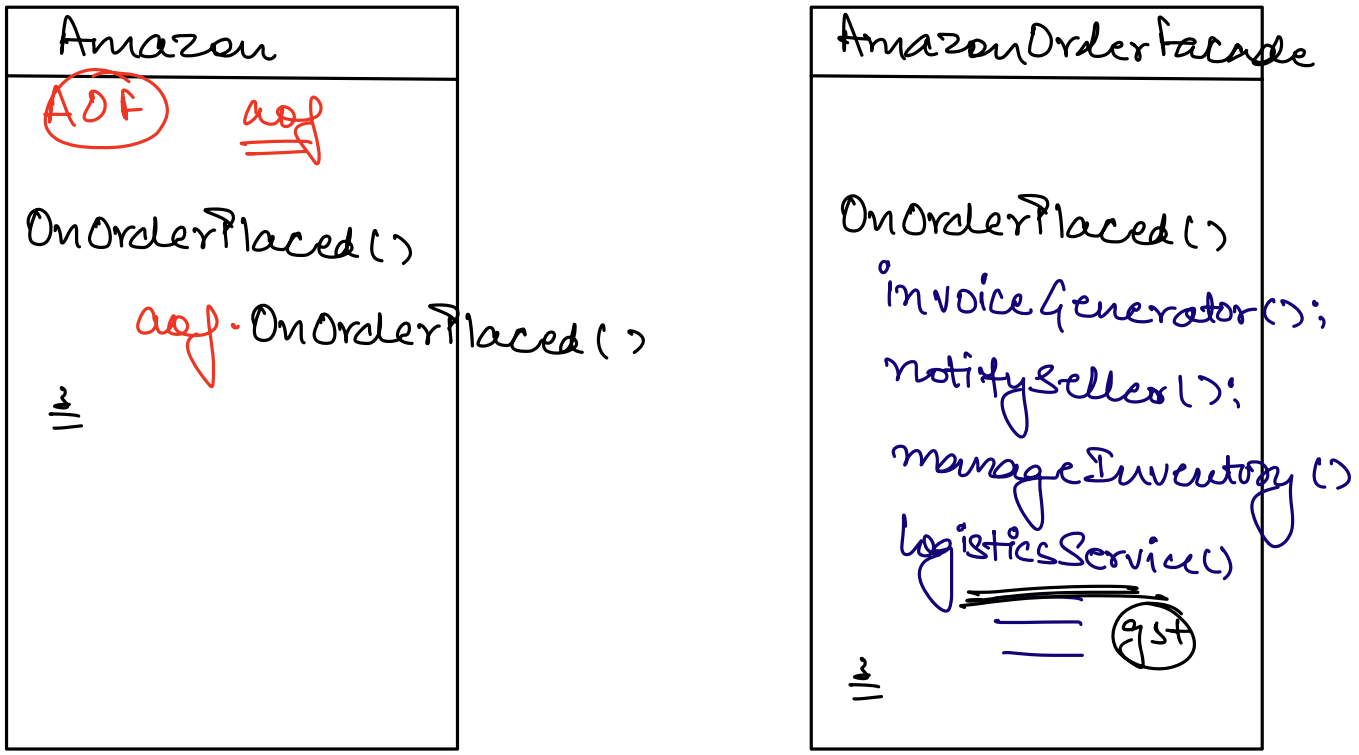
```
PathCalculatorFactory
-----------------------------------

Static:
getPathCalculatorForMode(String mode){
  if(mode == Car){
      PC = new CPC();
  }
  else if (mode == "Bike"){
      PC = new BPC();
  }
}
```

```
GoogleMap
-----------------------------
PathCalculator PC;

findPath(A, B, mode){
    PC = PCF.getPCforMode(mode);
    PC.findPath(A, B);
}
```

Sorting



⇒ Observer.

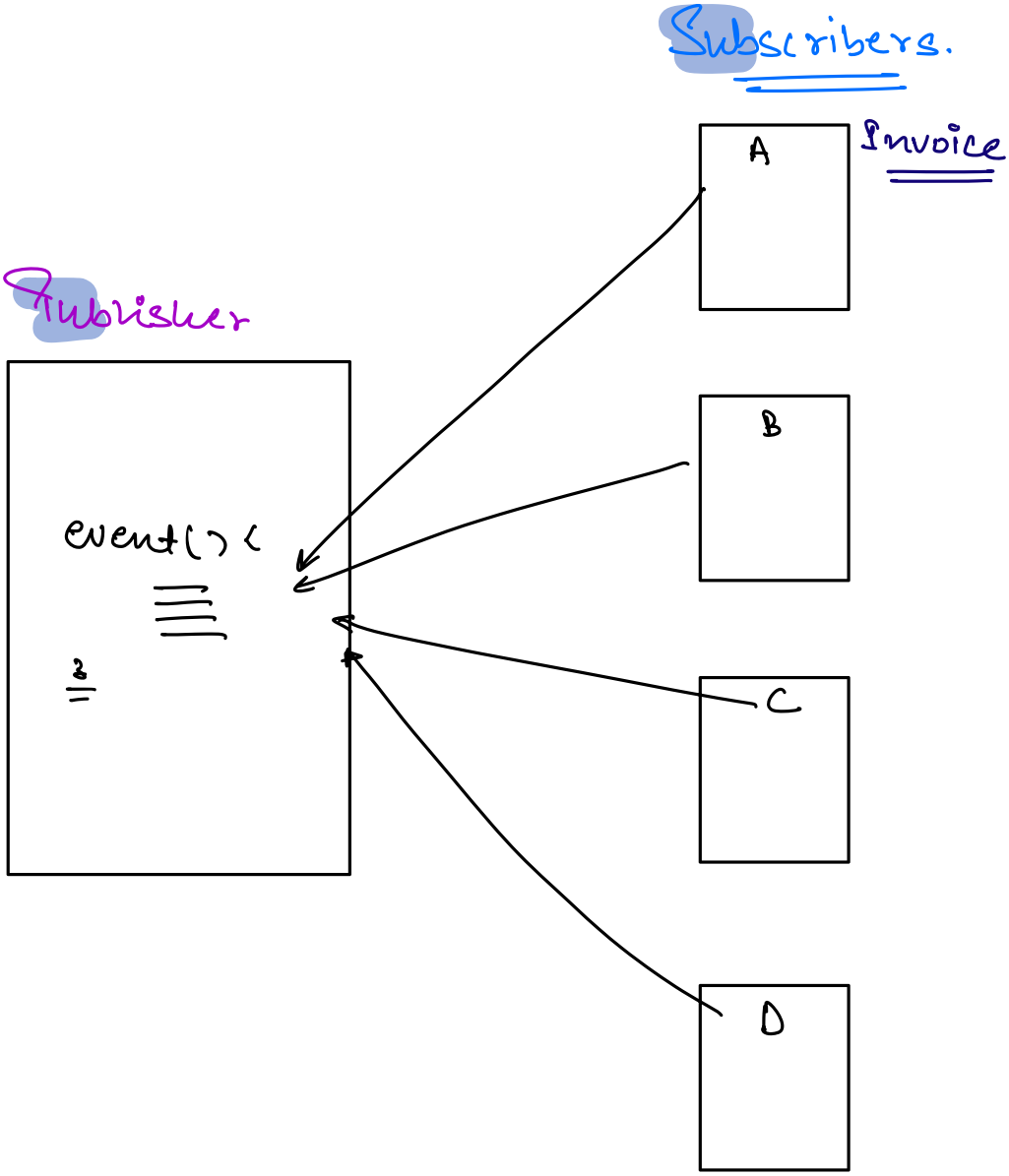


Problem Statement

⇒ When some event happens, we might want to do a lot of things internally. Every time we want to add/remove something from the list of actions, we'll have to do the code changes.

⇒ We can't add/remove actions at runtime.

OBSERVER.



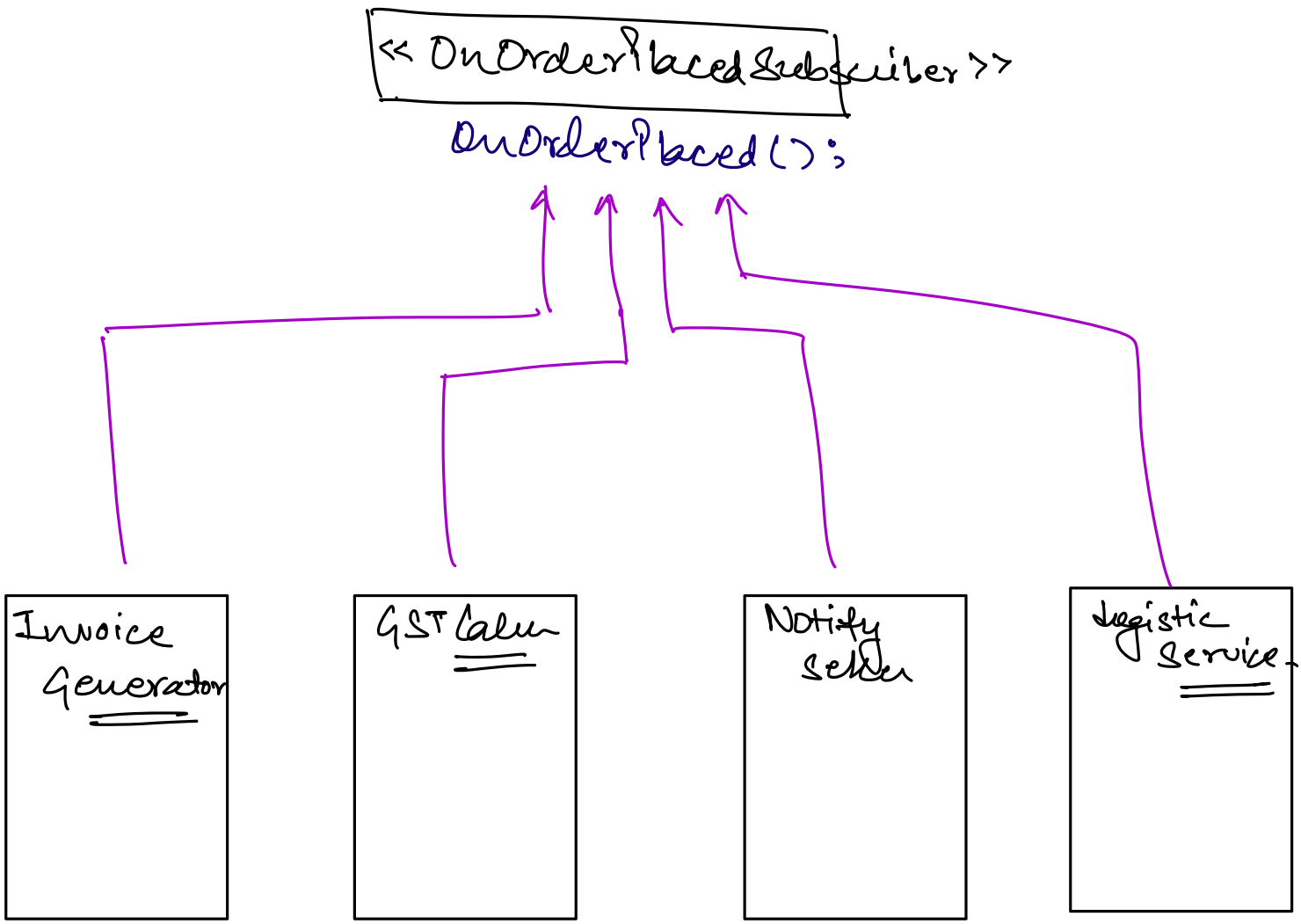
⇒ Event Driven Architecture.

⇒ When an event gets published, multiple Subcribers would like to get executed.

1. Create different classes for each subscriber.

2. Make all the subscribers implement a common interface.

3. Publisher should provide a functionality to add/remove a subscriber for the event.



AmazonOrderPlaced

List<OnOrderPlacedSubscriber> subscribers;

Static

addSubscriber(OOPS —>){

subscribers.add(↓)

}

OnOrderPlaced(){

for(Subs sub : subscribers){

sub.onOrderPlaced();

}

}

⇒

InvoiceGenerator.

InvoiceGenerator(){

AmazonOnOrderPlaced.subscribe(this);

}

——— * ———

Implementation.

Case Studies : Tic Tac Toe
Parking lot
Book My Show
Splitwise.

} LLD-3.

⇒ LLD-1 (language) ⇒ Mock Interview

LLD-2 & 3 ⇒ Mock Interview

Agenda -

⇒ UML Diagrams. → Class Diagram
→ Use Case Diagram.

UML Diagrams.

Communication.

Clients ⇒ Gather requirements

Managers
- Product Managers
- Engineering Managers

} Design Discussions / Approvals.

Business } Requirements.

Ways to communicate?

⇒ Words.

↳ Misunderstanding / Ambiguity.

⇒ Explaining the design using only words isn't good enough because there can be lot of ambiguity.

⇒ Picture.

Pros.

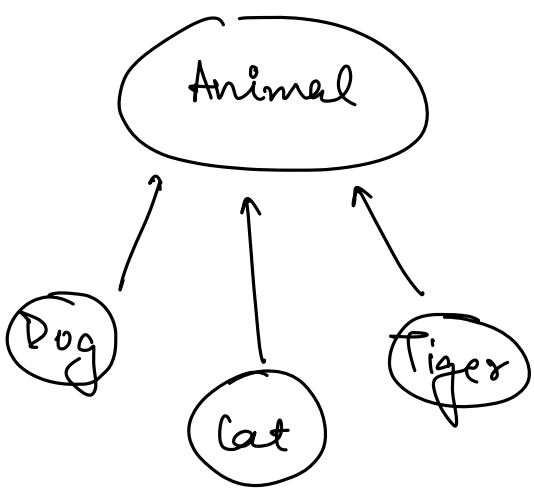
↳ less ambiguous.

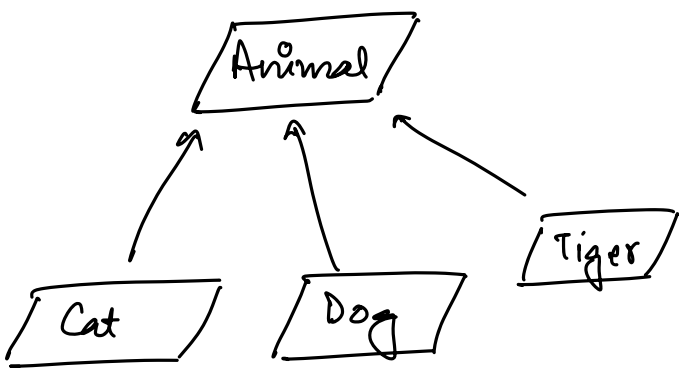
↳ Easy to understand.

↳ Easier to visualize.

Cons.

↳ No Standardization.



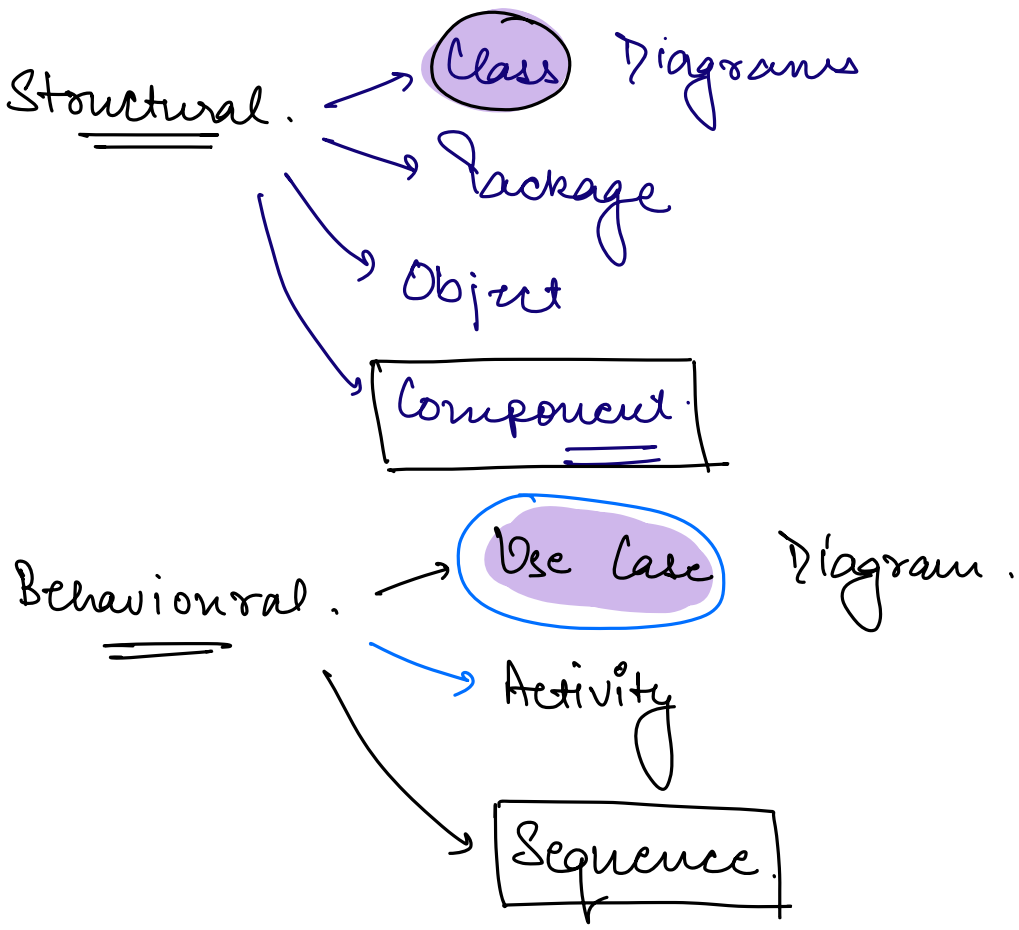


UML (Unified Modeling language).

⇒ Standard way of representing our low level design.

2 type of UML diagrams.

- Structural ⇒ How our codebase looks like?
- Behaviourals.
  - ↳ Methods / Actions.



Use Case

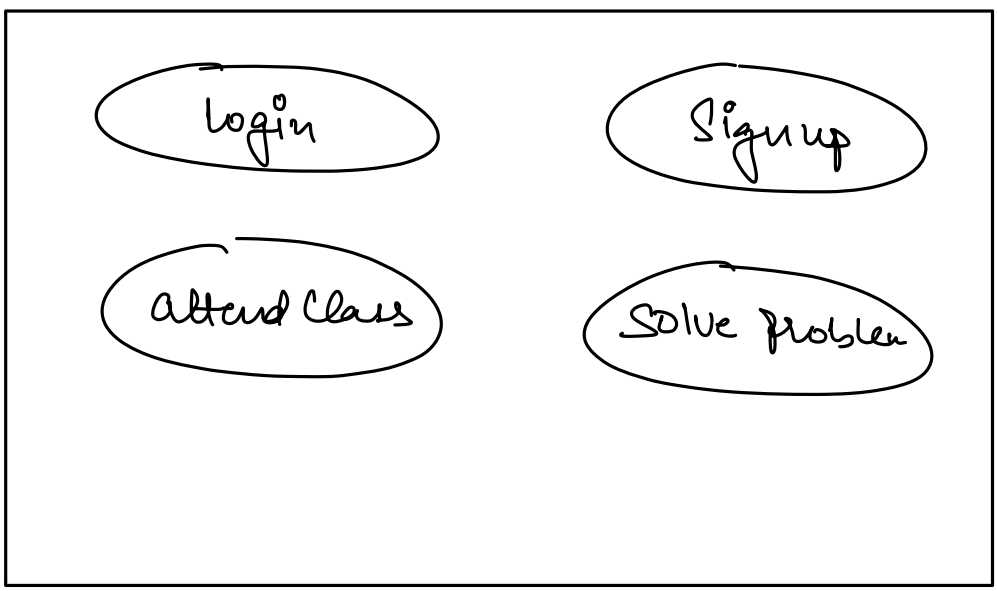
⇒ Behavioural.

⇒ Actions / Methods.

⇒ Different features / functionalities (Use cases.) that are supported by our system, along with their users.

5 Components.

1) System Boundary. ⇒ Represents the scope of the System.



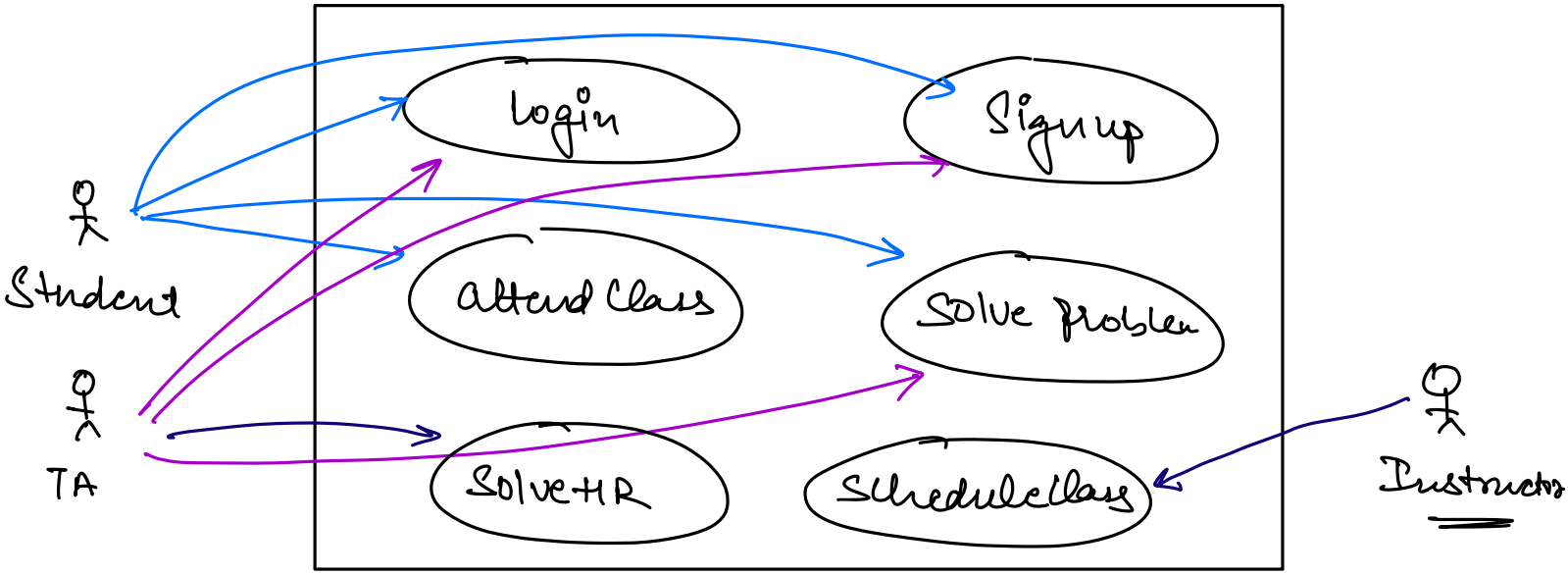
2) Use cases.

- ↳ functionalities / methods.
- ↳ Oval Shape.

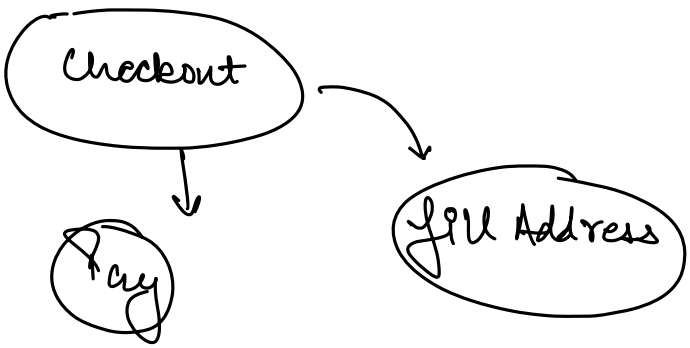
③ Actors.

⇒ Users of the system.

⇒



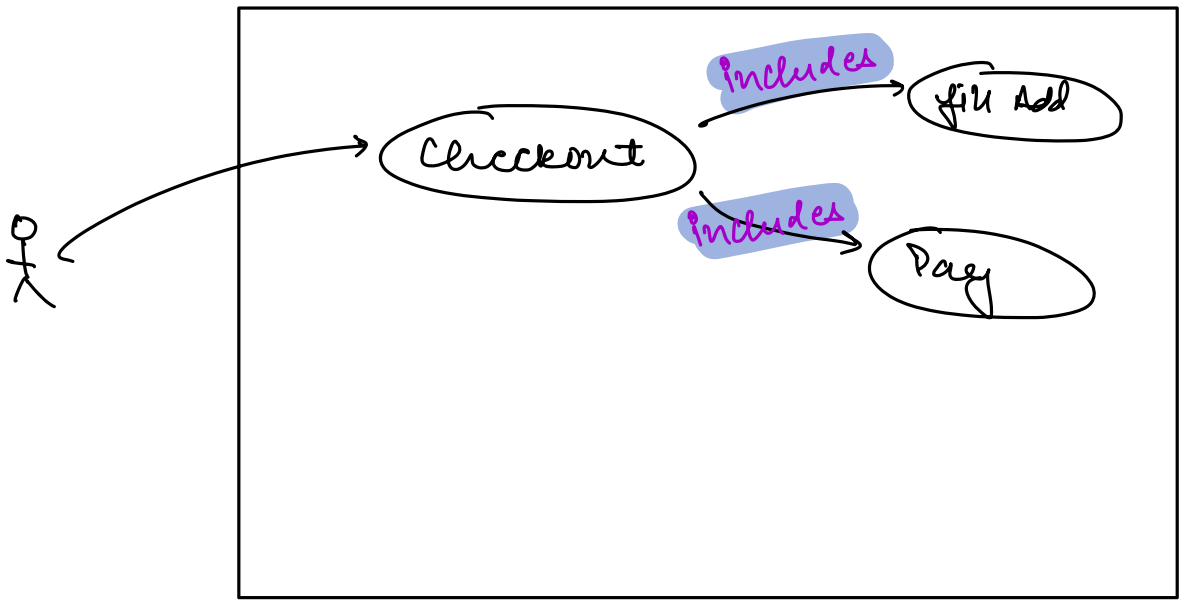
④ Includes.



Checkout includes

→ Pay

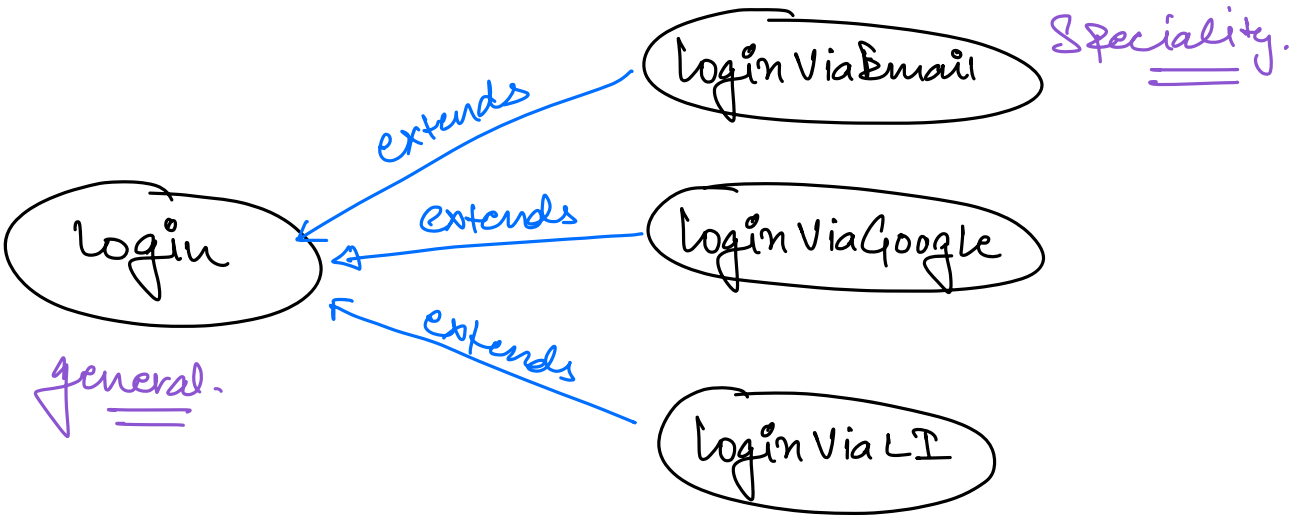
→ fill address.



To complete Checkout, user will have to fill address & pay

(5) Extends.

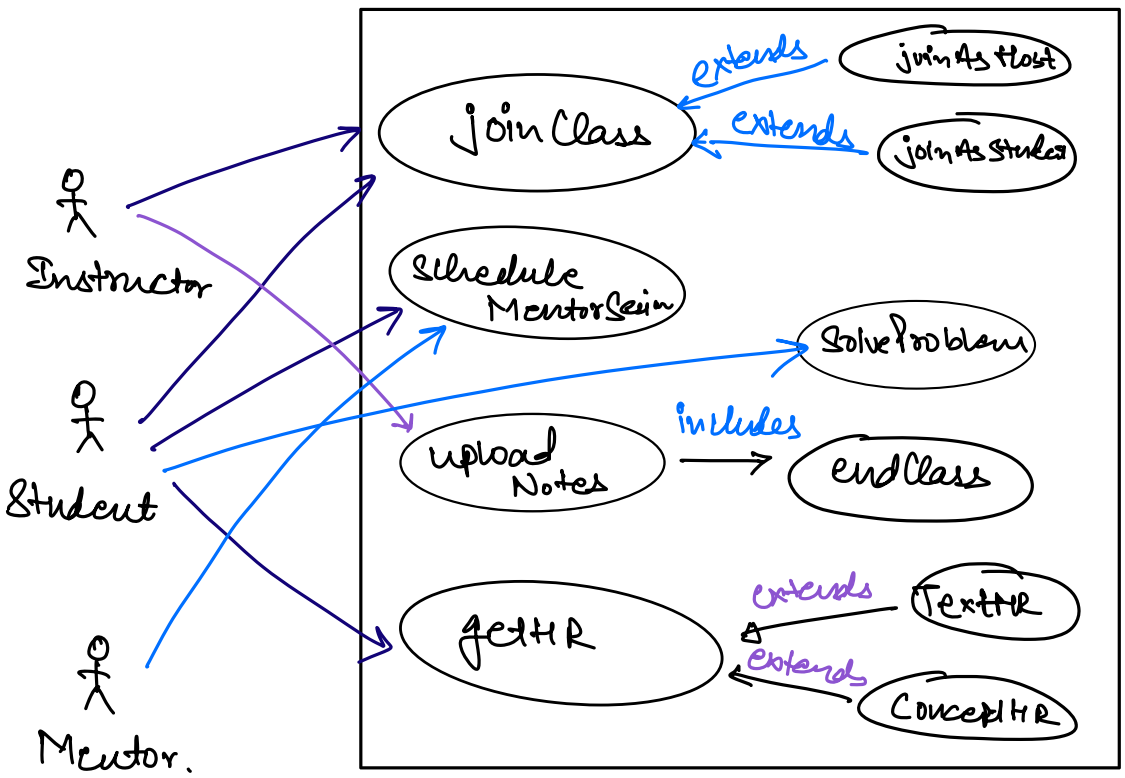
If one feature has multiple variants.



Assignment.

⇒ Draw a use case diagram [SCALER].

1) At least 5 features
2) At least 2 actors.
3) Includes
4) Extends.



Class Diagram.

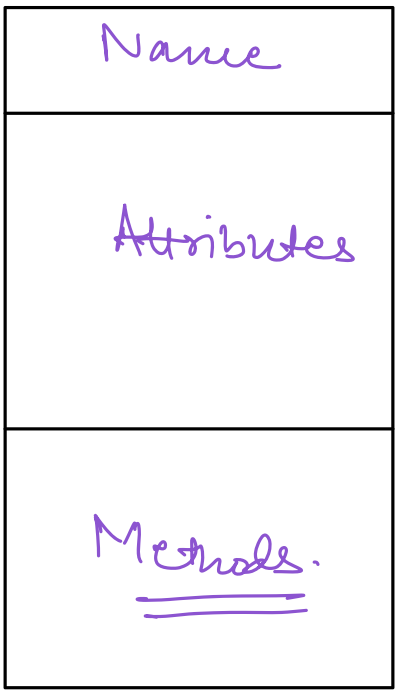
⇒ Used to represent entities in the system & their relations.

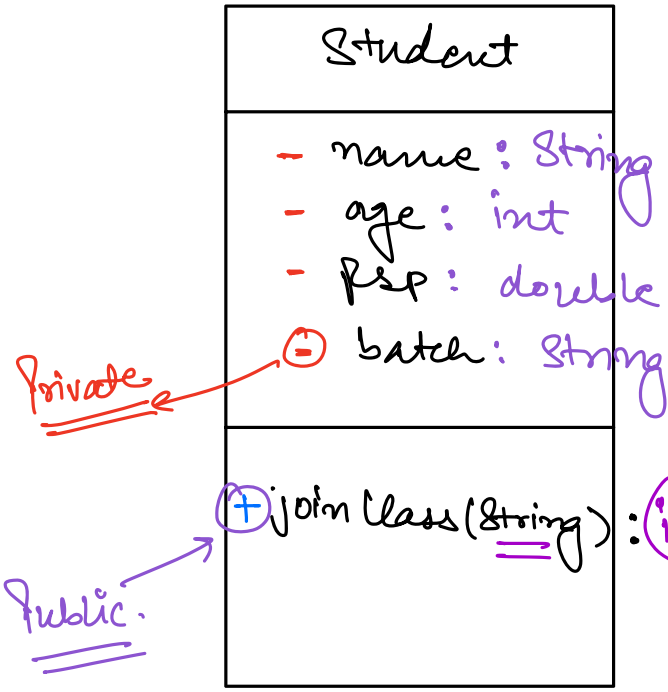
- Class
- Interfaces
- Abstract Class.

⇒ Inheritance relation.

⇒ Class implementing an interface.

⇒ Class





access Modifier — Datatype — attr.

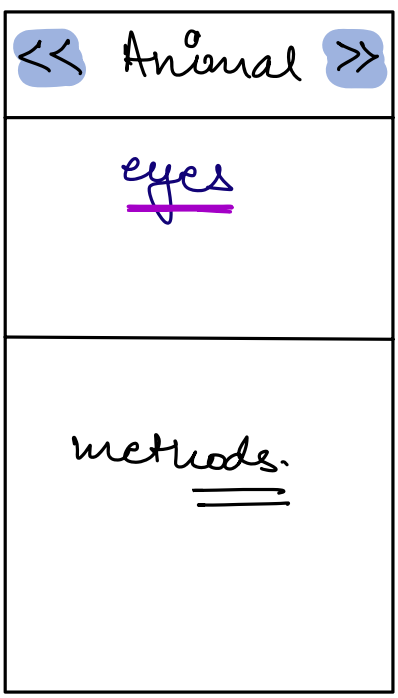
↳ Public : +

↳ Private : −

Protected : #

return type of fun.

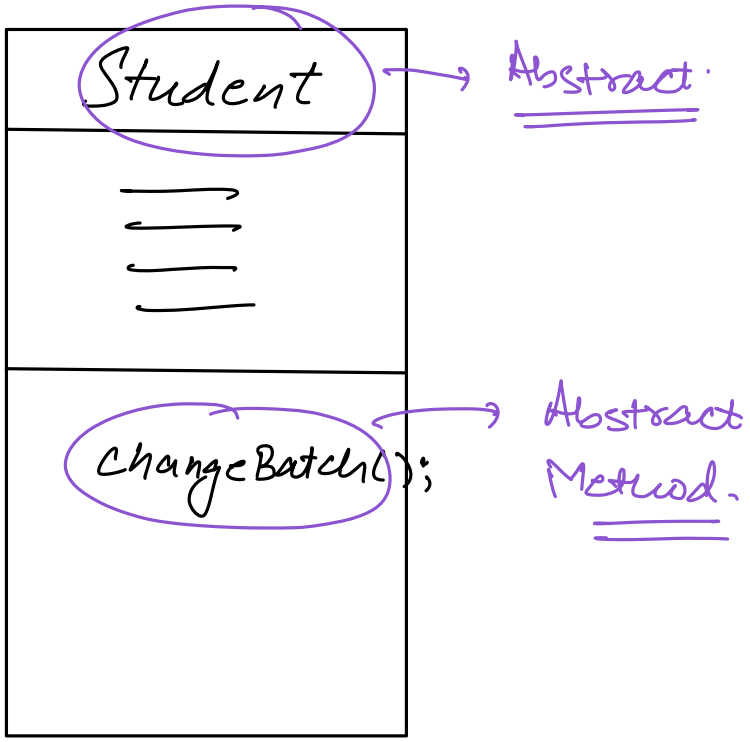
⇒ Interfaces.



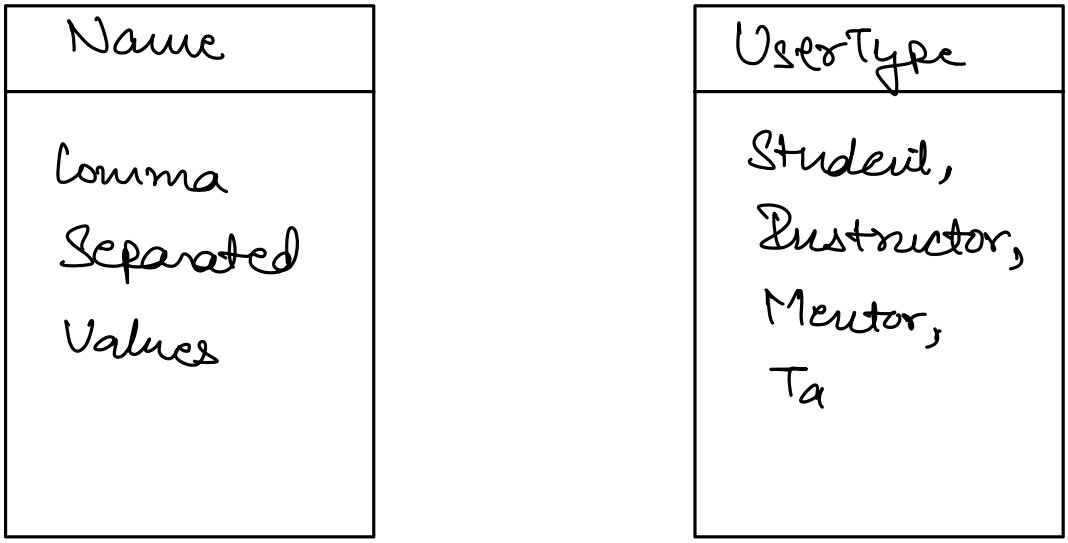
⇒ Static keyword is represented using an underline.

Abstract Class.

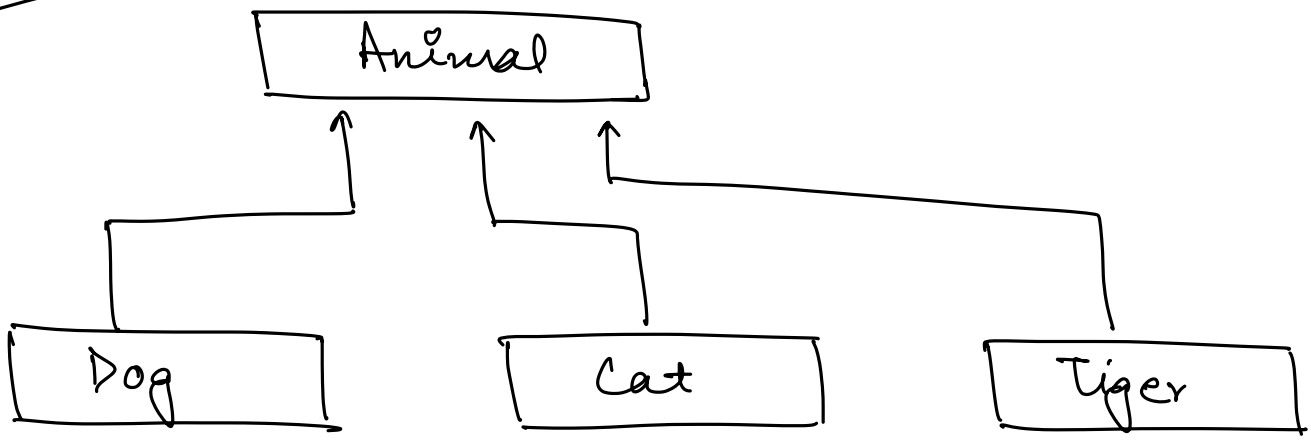
Exactly same as normal class the only diff is to write the class name in italics.

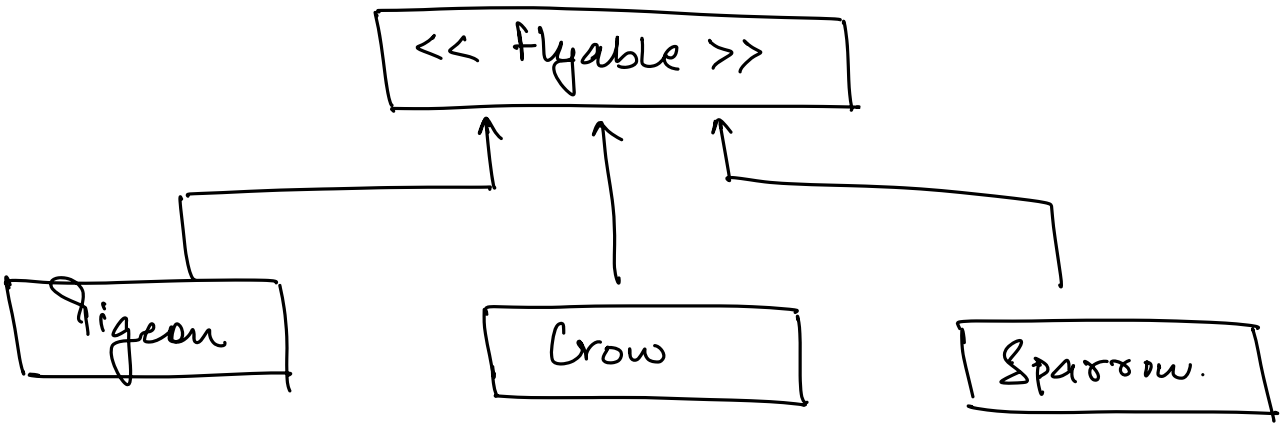


Enum

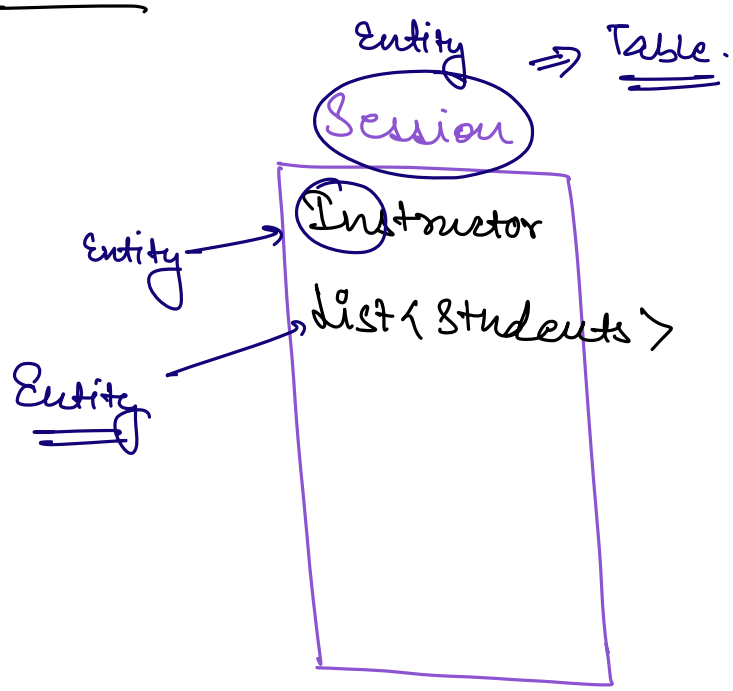


IS-A.

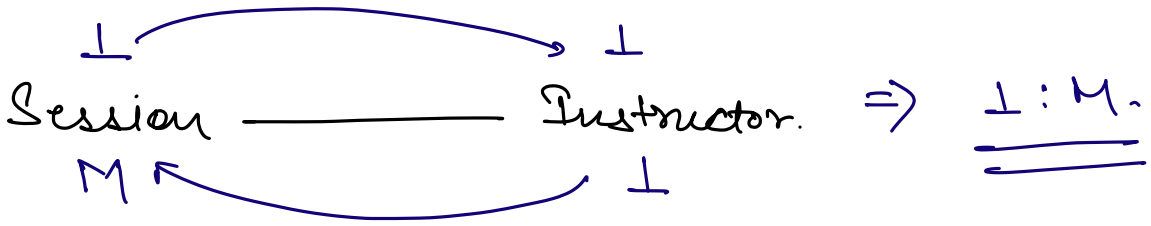




HAS-A.



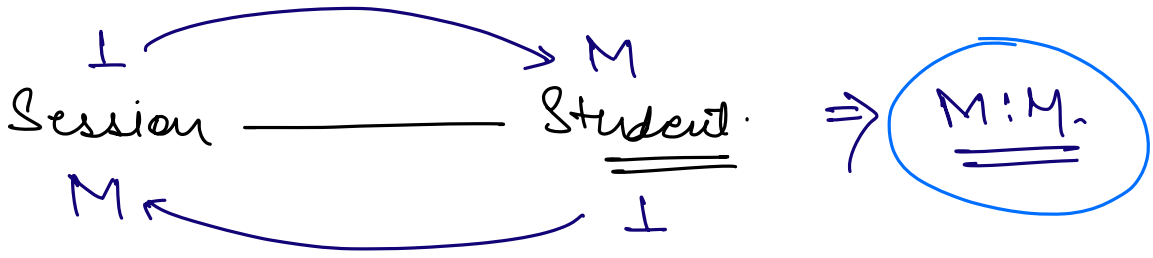
Session HAS-A Instructor



⇒ 1 : M.

⇒ Id of (1) side on (M) side.

⇒ Session table will have instructor id.



⇒ M : M.

⇒ Mapping table.

Student_sessions

St_id	Session_id

———— * ————

LLD-3.