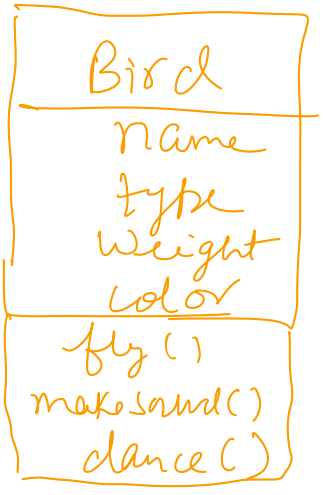
Agenda ⇒ ~~①~~ LID of SOLID

② Dependency Injection

Class starts at 9:05 PM

- L → Liskov Substitution principle
- I → Interface Segregation "
- D → Dependency Inversion "

V0



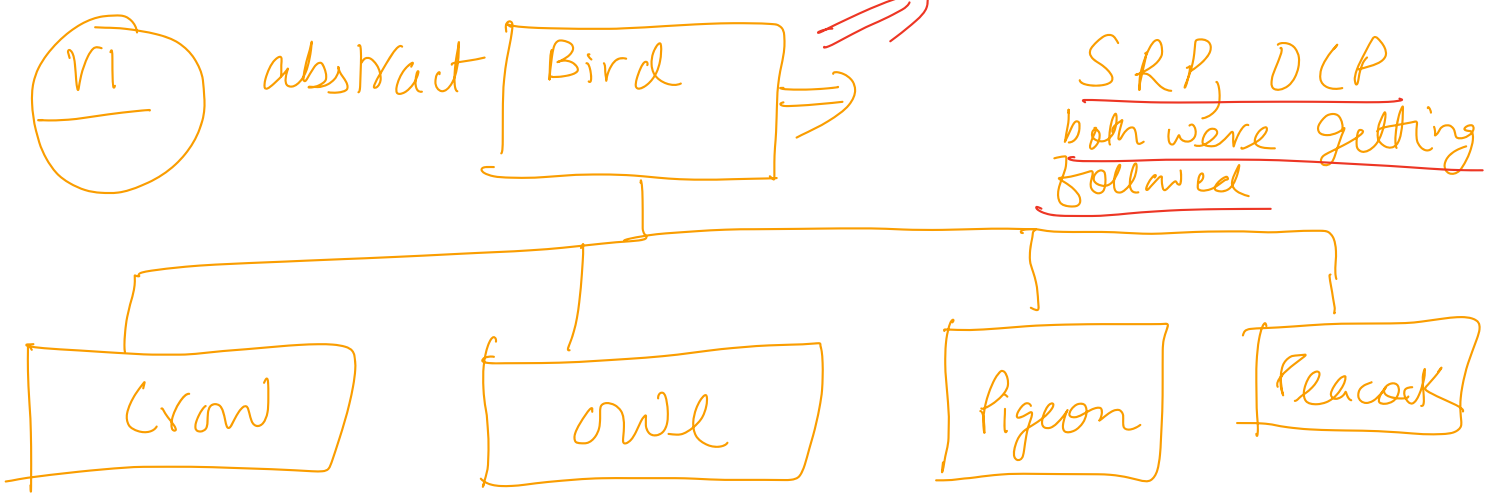
⇒ SRP, OCP

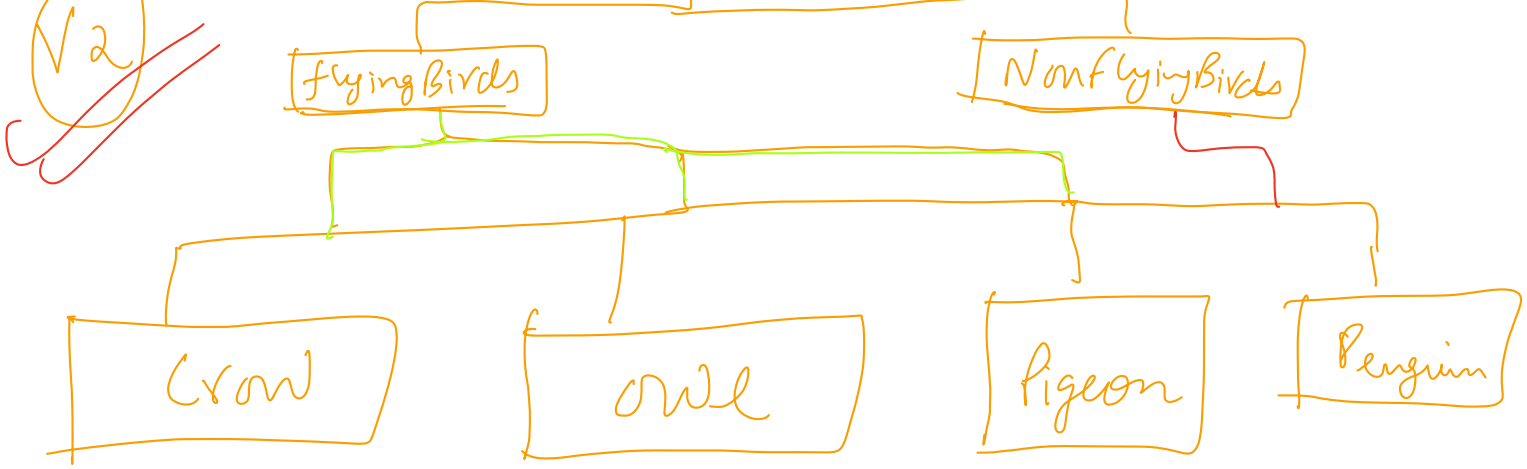
V1

abstract Bird

LSP is getting violated

SRP, OCP both were getting followed





⇒ class explosion would happen in the above solution.

Problem Statement ⇒ ① Some birds demonstrate a behaviour while other birds do not demonstrate

② I should be able to get list of flying birds. List<FlyingBirds>

① ① FlyingDancerBirds ② FlyingNonDancerBirds
③ NonFlyingDancerBirds ④ NonFlyingNonDancerBirds

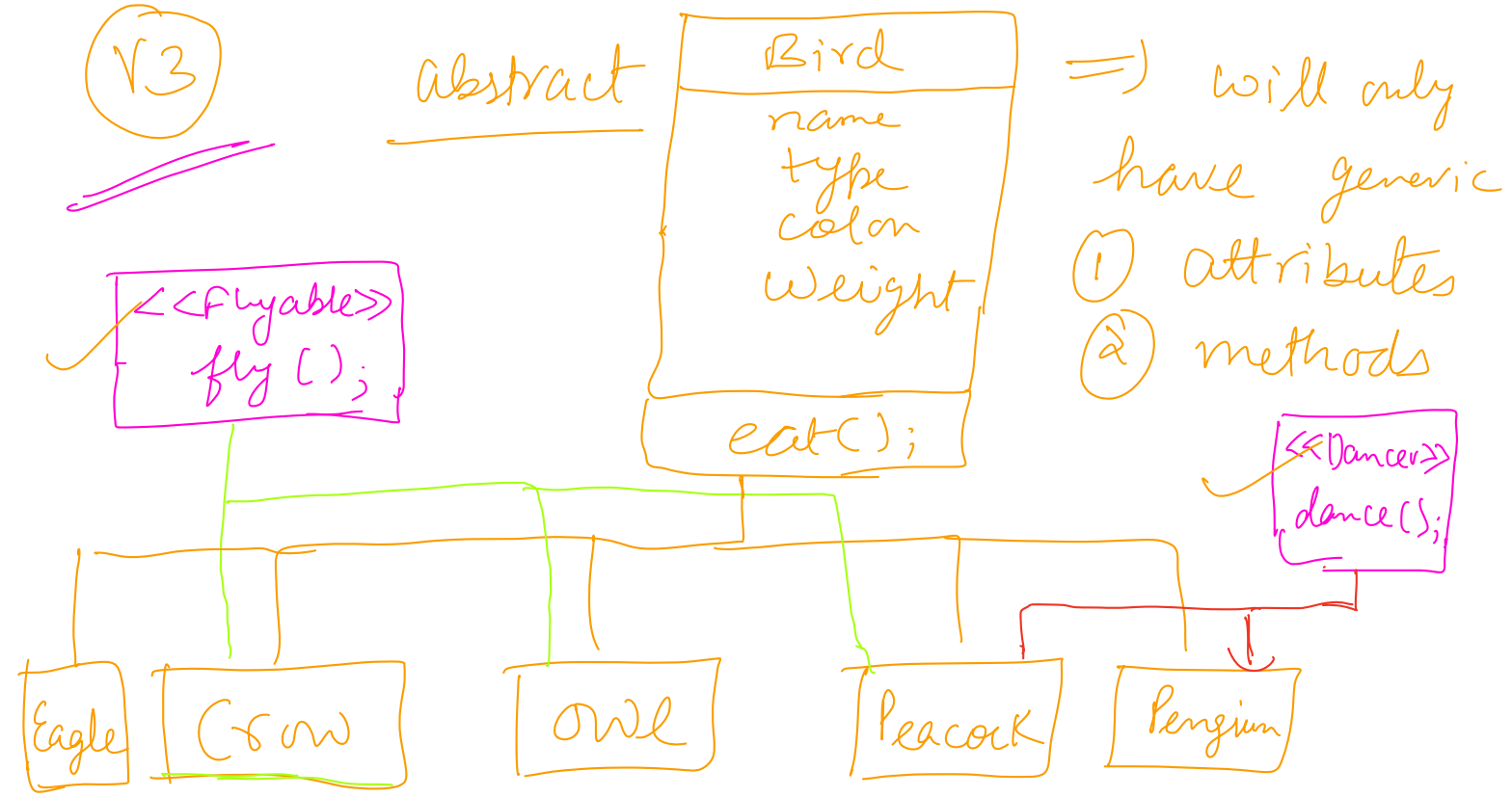
⇒ till now we were representing behaviours by classes which is not the ideal use case of classes

⇒ Behaviours should be represented by

Interfaces.

V3

abstract



⇒ will only have generic
① attributes
② methods

List<Flyable> ⇒

class Crow extends Bird implements Flyable {

eat() {

}

fly() {

}

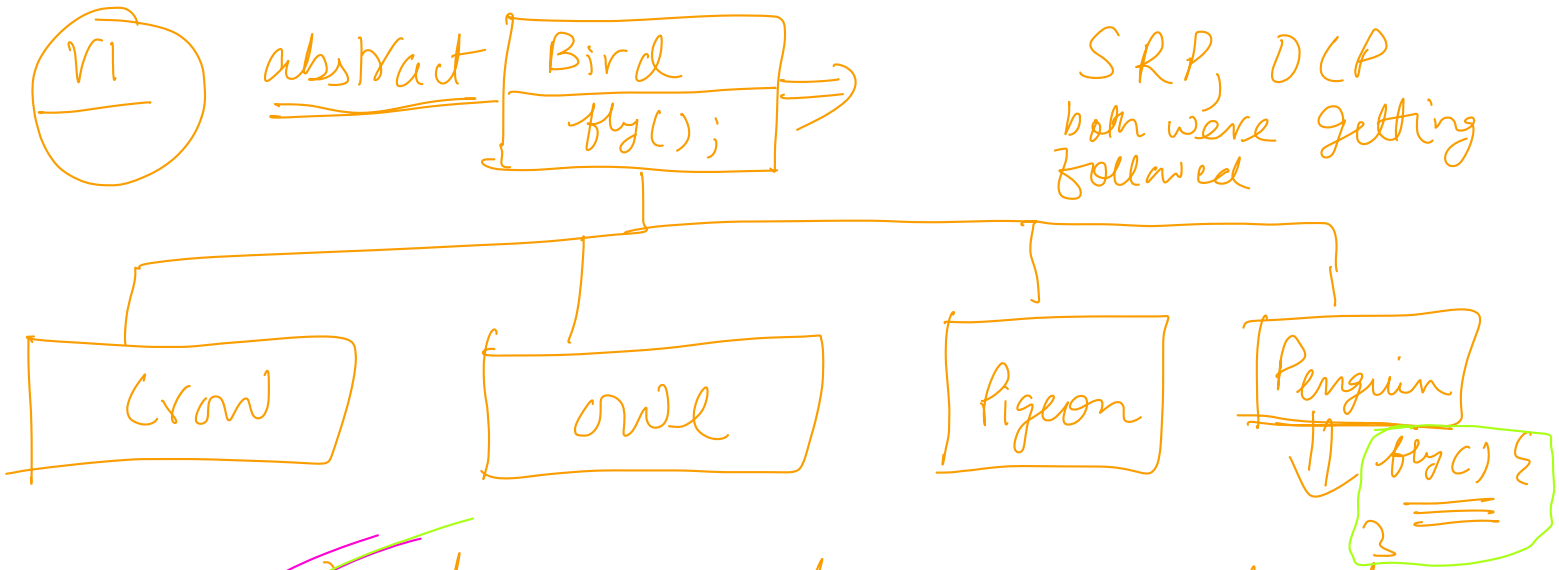
}

(3) Liskov Substitution Principle ⇒ Object of any child class should be substitutable in a variable of parent type without requiring any specific code change.

```
Bird b = new Bird();
       = new Crow();
       = new Penguin();
```

this Bird class might be getting called from other places as well, so the client can do b.fly() as well.

(V1)



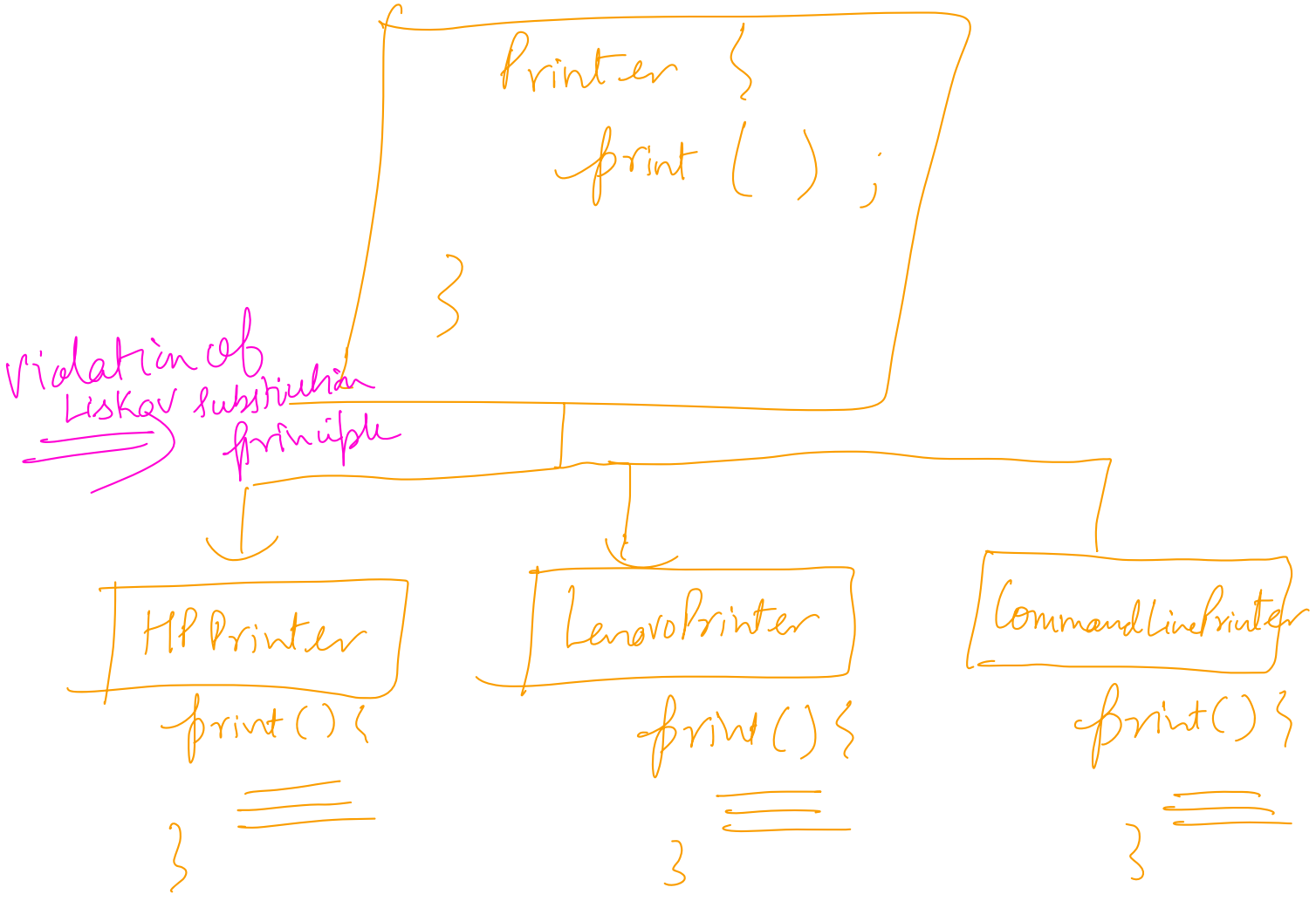
SRP, OCP both were getting followed

dummy implementation of fly() throw an exception.

```
Bird b = new Penguin();
```

b. fly() ⇒ exception ✗
dummy

⇒ All the child classes should behave as the parent expects it to behave.



✓ Printer p = getPrinterObject();
p.print();

p is ① HP Printer object ⇒ print to Physical paper

② LenovoPrinterObject ⇒ print to physical

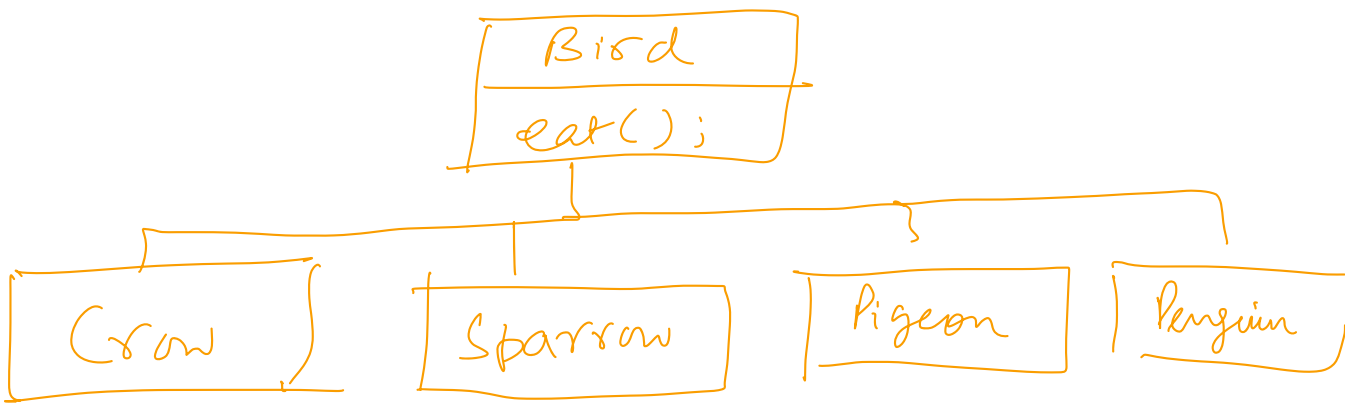
(3) CommandLinePrinter object ⇒ Proper output will be printed to cmd line

⇒ Wherever you are implementing a method of parent class, then do what the parent class expects the method to do.

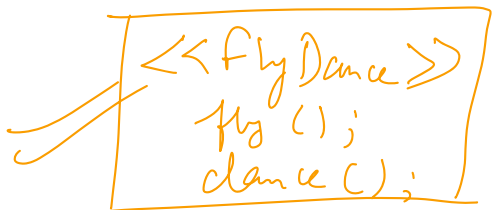
(4) Interface Segregation ⇒ let us say I have a requirement

(a) Either a bird can fly & dance

(b) Or birds cannot fly & cannot dance.



Option 1 ⇒



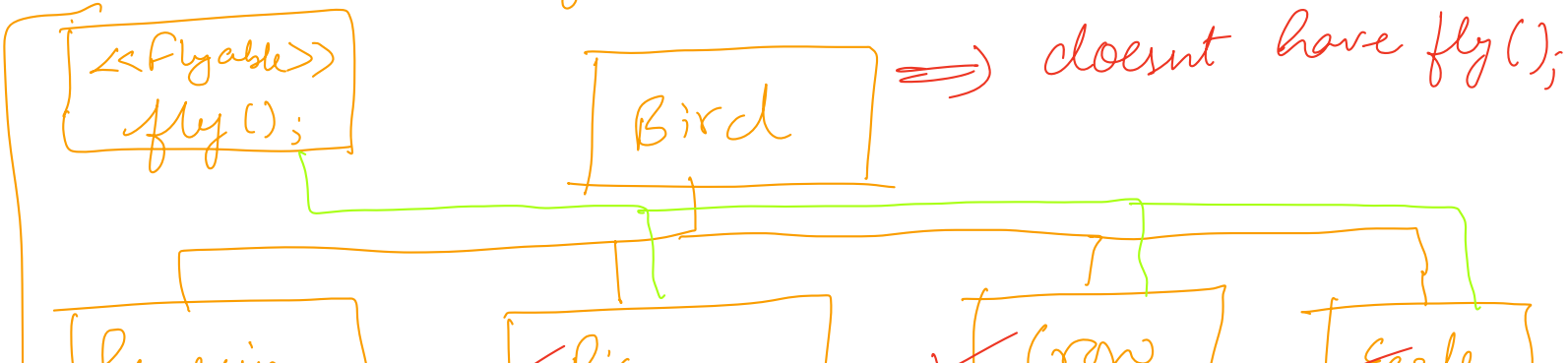
Option 2 → <<Flyable>> fly(); <<Dancer>> dance();

Option 2 is better here because in the future the requirement can change and it can happen that there are birds who can fly but cannot dance and vice-versa.

Interface Segregation ⇒ Try to keep your interfaces as light as possible, basically have minimum methods in it.

⇒ An interface with only 1 method is called as functional interface

(5) Dependency Inversion principle



Penguin Pigeon Eagle

fly() {
Ⓐ [redacted] makeFly();
}

fly() {
makeFly();
[redacted]
}

class X {
do() {
[redacted]
}}

instead of having duplicate code in Ⓐ & Ⓑ, fly method of pigeon & eagle, I can just call the do() method from there

PigeonEagleFlyingBehaviour {
makeFly() {
[redacted]
}}

class Pigeon { ⟹ violating

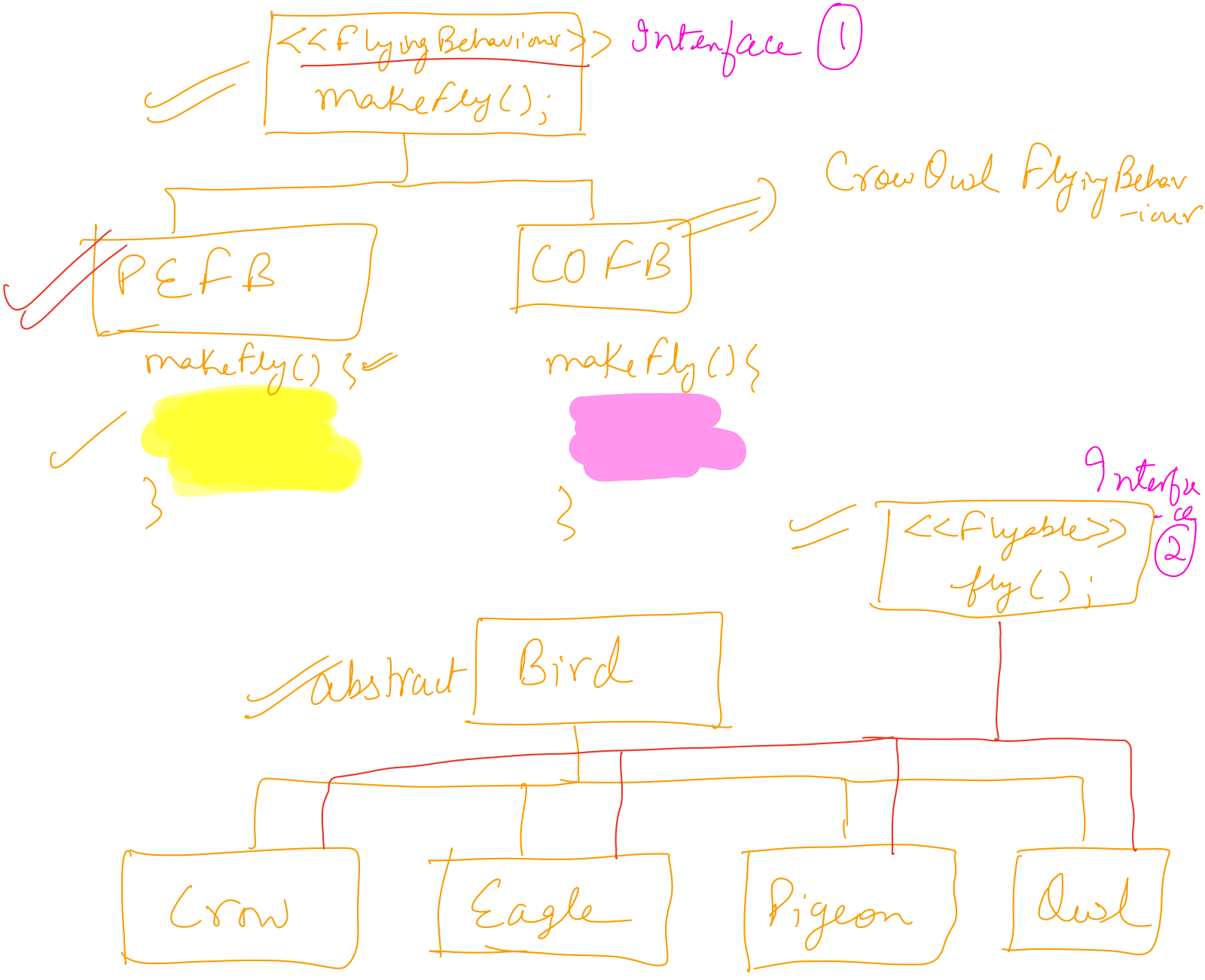
PEFB pefb = new PEFB();

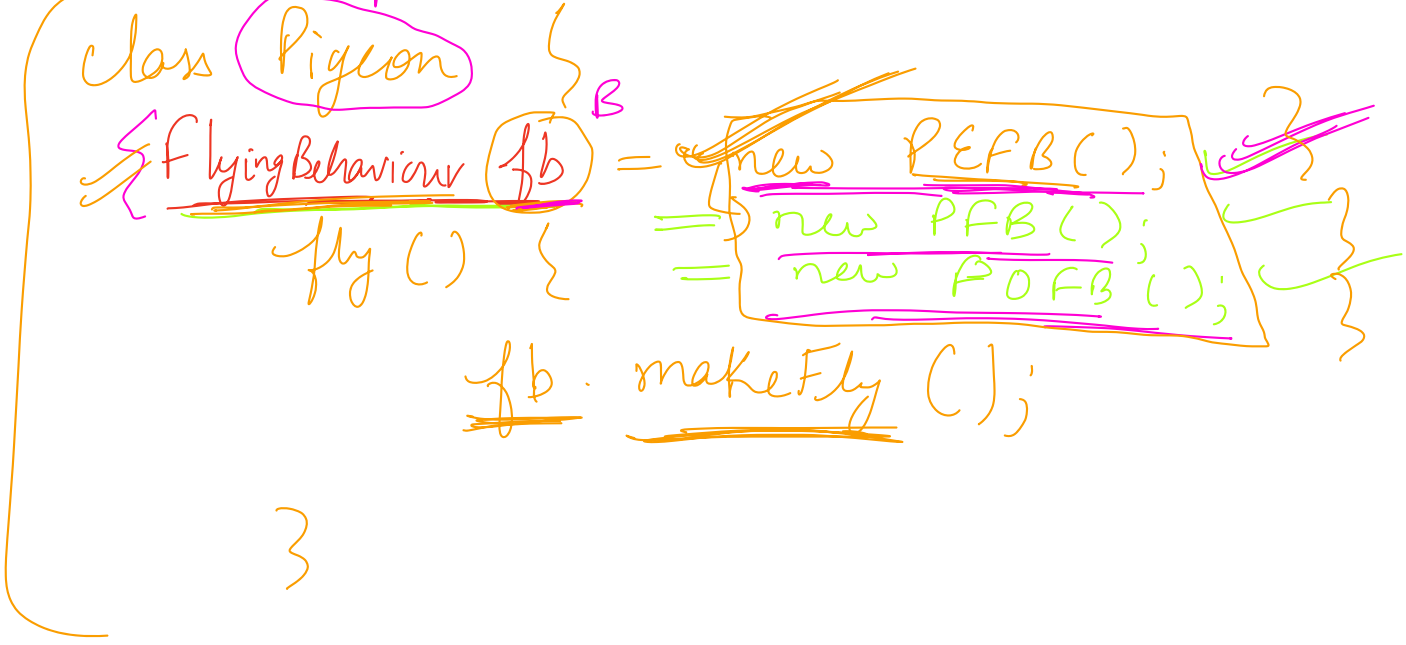
fly() {

pefb.makeFly();

}

⇒ DI says that no two concrete classes should directly depend on each other





All 5 SOLID principles are being followed now.

⟹ { DI says do not code to an implementation, code to an interface.

⟹ Dependency Injection ⟹ [A] → [B]

{ If a class A has an attribute of class B, then A should not be creating object of B itself, rather the caller/client of A should create an object of B and pass it to A.

class Pigeon {

FlyingBehaviour fb; }

Pigeon (FlyingBehaviour behaviour) {

this.fb = behaviour;

}

fly() {

fb.makeFly();

}

⟹ And this can be done via constructor.

```
Pigeon p = new Pigeon ( new PEFB() );
```

⟹ It is called dependency injection because we dont have to create the object of dependency ourselves, rather it is injected into me via the constructor.

Benefits of Dependency Injection

① My codebase wont be dependent on any concrete class now.

(2) It makes the testing of code very easy.

Pigeon p = ~~new~~ Pigeon (new PEFB());

= ~~new~~ Pigeon (new POFB());

= ~~new~~ Pigeon (new PFB());

Bird b = getObjectOfBird();

⇒ owl.fly()

⇒ pigeon.fly()

⇒ penguin.fly()