SOLID

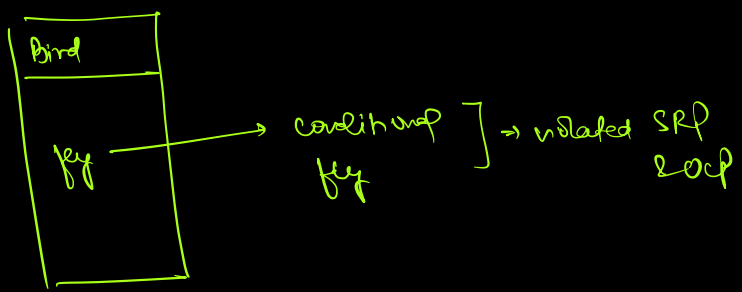
L - Liskov's substitution principle

I - Interface segregation

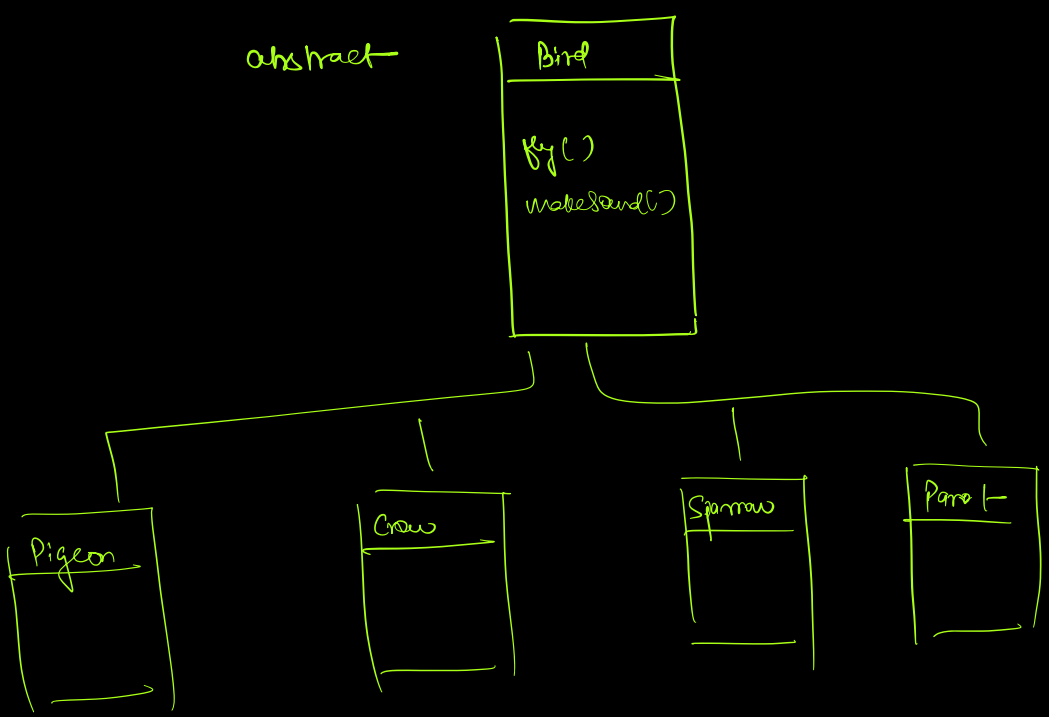
D - Dependendency Inversion.

⇒ Evolution of Bird :-

V0 →



V1) abstract



eg. ⇒ add a non-flyable bird Penguin

Problem Statement => Some birds demonstrate a behaviour while other birds don't demonstrate that behaviour

1) Only the birds having the behaviour should have the method

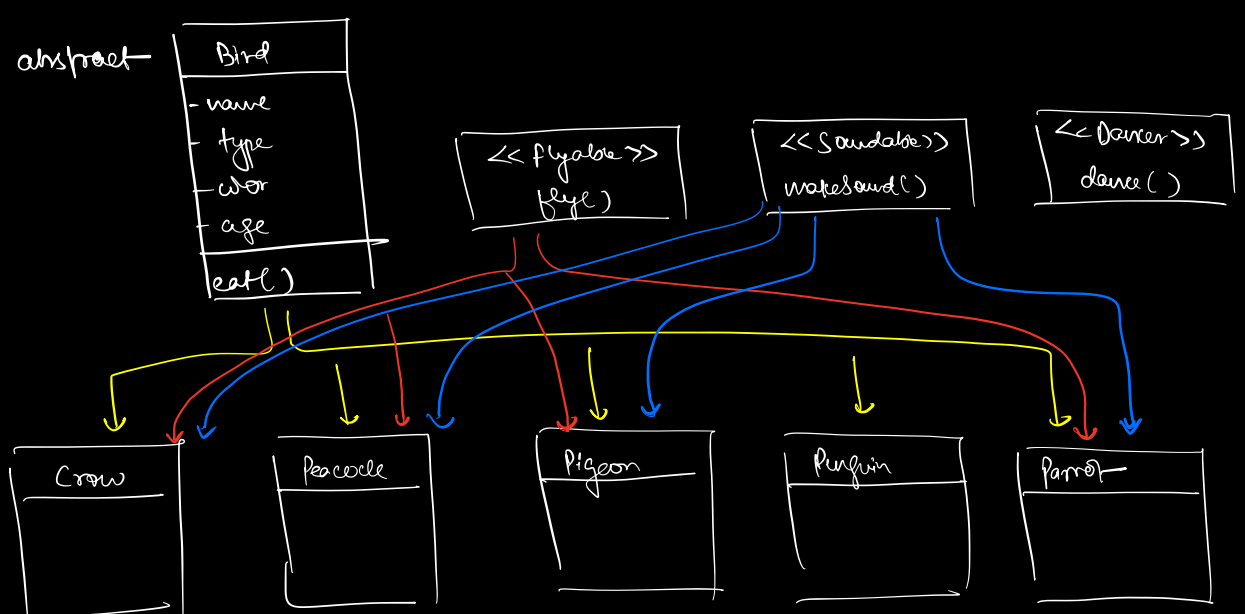
2) We should be able to group the birds that have that behaviour.

abstract behaviour → Interface

for specific behaviour we will create a interface and the bird which has that behaviour will impl it,

Other's won't

contains attr and behaviour common for all birds



Crow extends Bird
implements Flyable, Soundable

Peacock extends Bird
implements Flyable, Soundable, Dancer

Penguin extends Bird

Bird | Flyable | Soundable

```
class Crow extends Bird implement Flyable, Soundable {

    fly() {
        ≡≡≡
    }

    makeSound() {
        ≡≡≡
    }

    eat() {
        ≡≡≡
    }

}
```

[Crow doesn't have a dance as, it doesn't dance]

List<Flyable> flyBirds ← group of birds that can fly together

List<Soundable> soundableBirds → group of birds that can make sound.

: Liskov Substitution Principle :-

Object of any child class should be "as-is" substitutable in a variable of parent type without requiring any code change.

↳ a child obj should not get any special treatment to accomodate in a parent type variable

In v1 of Birds, the bird which could not needed a special treatment for fly() method

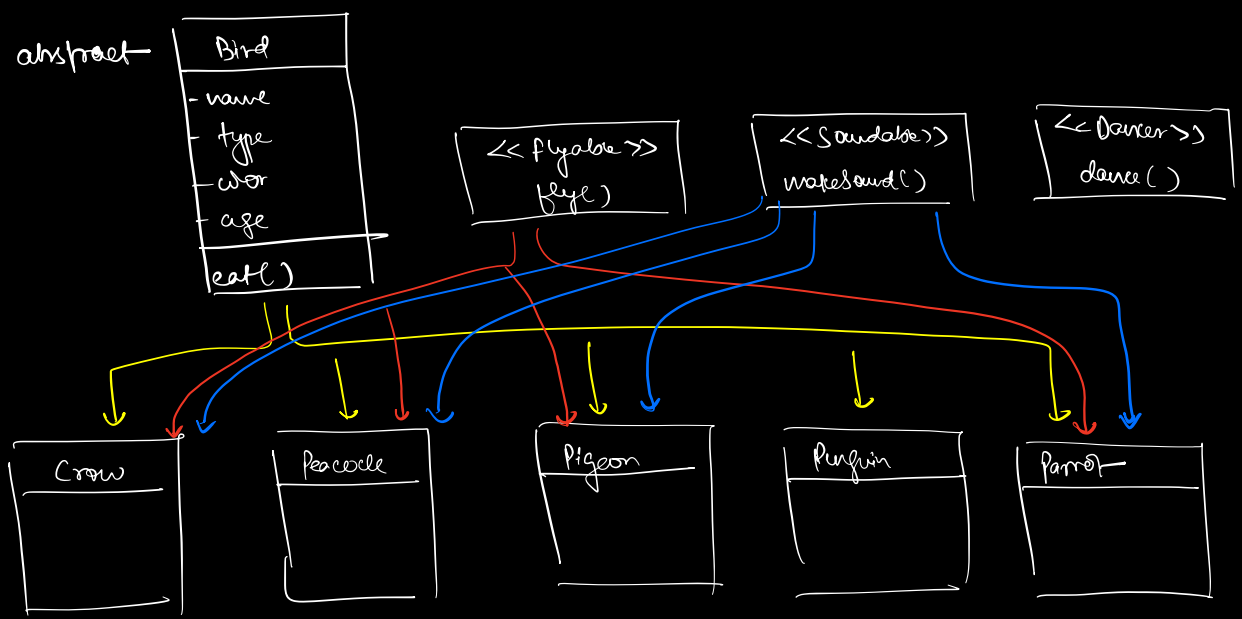
- either keep it empty
- throw exception

According to v1 of Birds

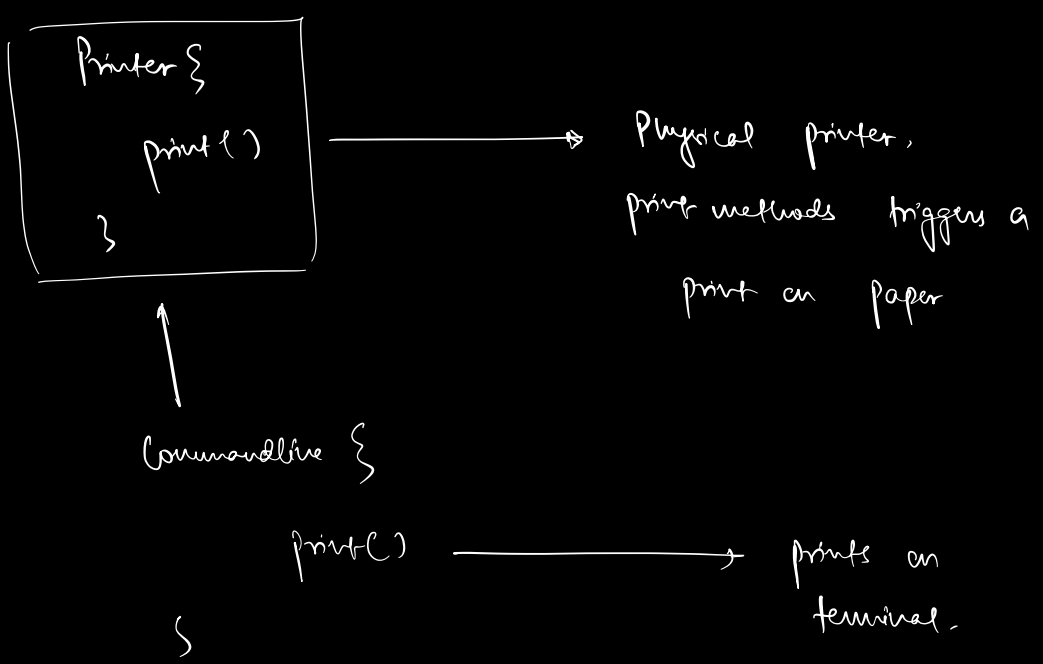
```
try {                                   → we need to add code to store child obj in parent type variable
    Bird b = new Pigeon()
             new Crow()
             new Penguin() → fly throws exception
} catch ( Exception thrown by fly() in Penguin ) {

}
```

* V1 of Birds wasn't following LSP.

V.2)



LSP ✓ SRP ✓ OCP ✓



```
Printer p = new CommandLinePrinter();
p.print();
```

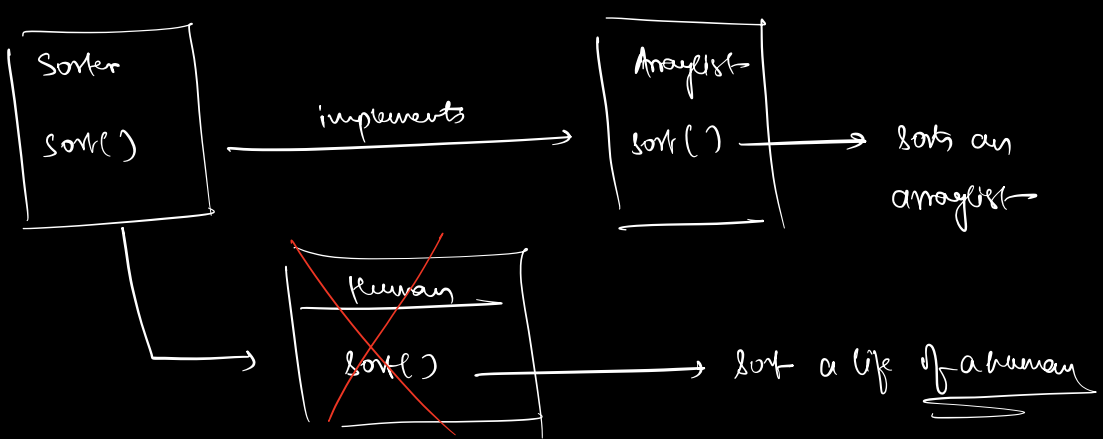
```
Runnable {
    run()
}
```

→ running a thread, deals with multi-threading

```
Person implement Run {
    run() {
        person is running
    }
}
```

→ physical act of running | exercise

LSP => don't override things that don't go together logically

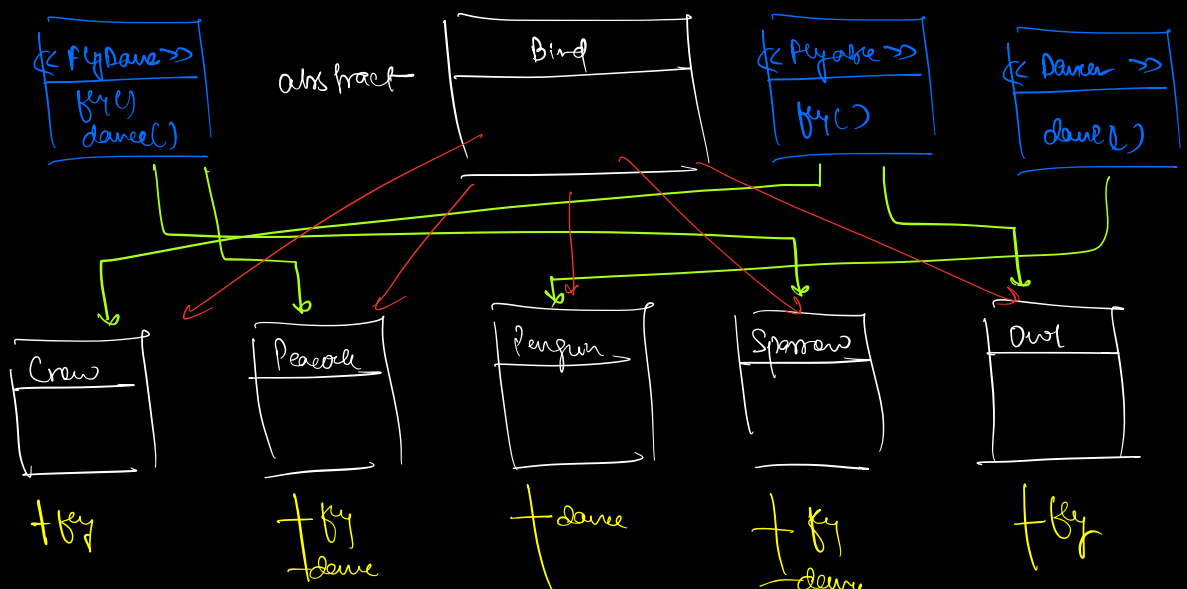


: Interface Segregation Principle :-

→ Some birds fly

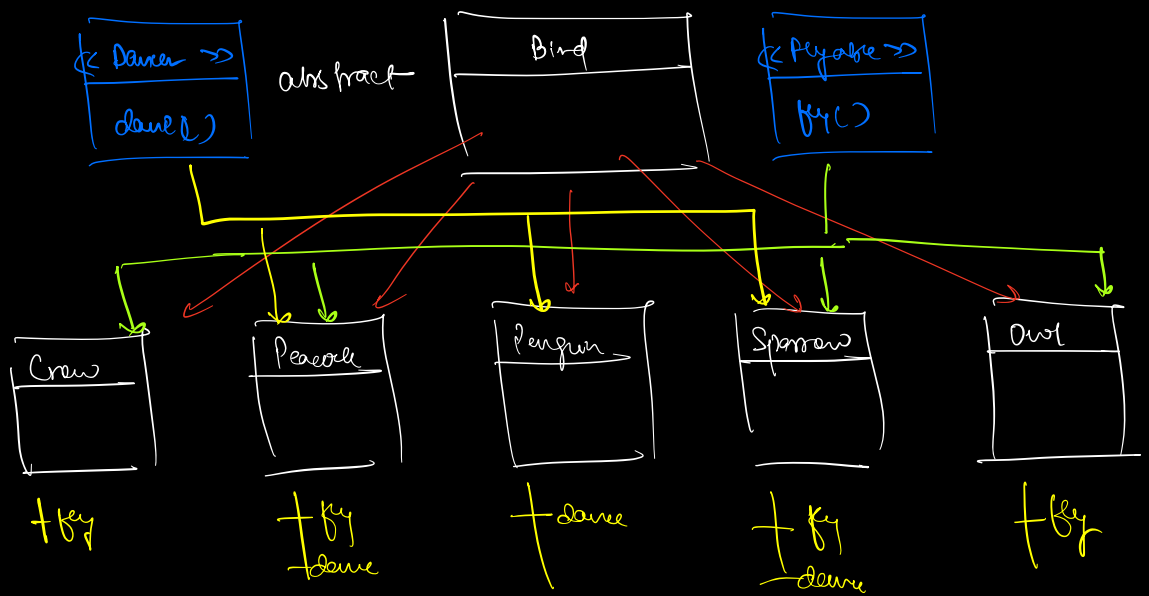
→ Some birds can dance

→ Some birds can fly as well as dance



↓

① Peacock extends Bird
implement FlyDance;



↓

Peacock extends Bird

(1) imply Flyable, Dancer

DSP :-

→ Interfaces should be as light as possible

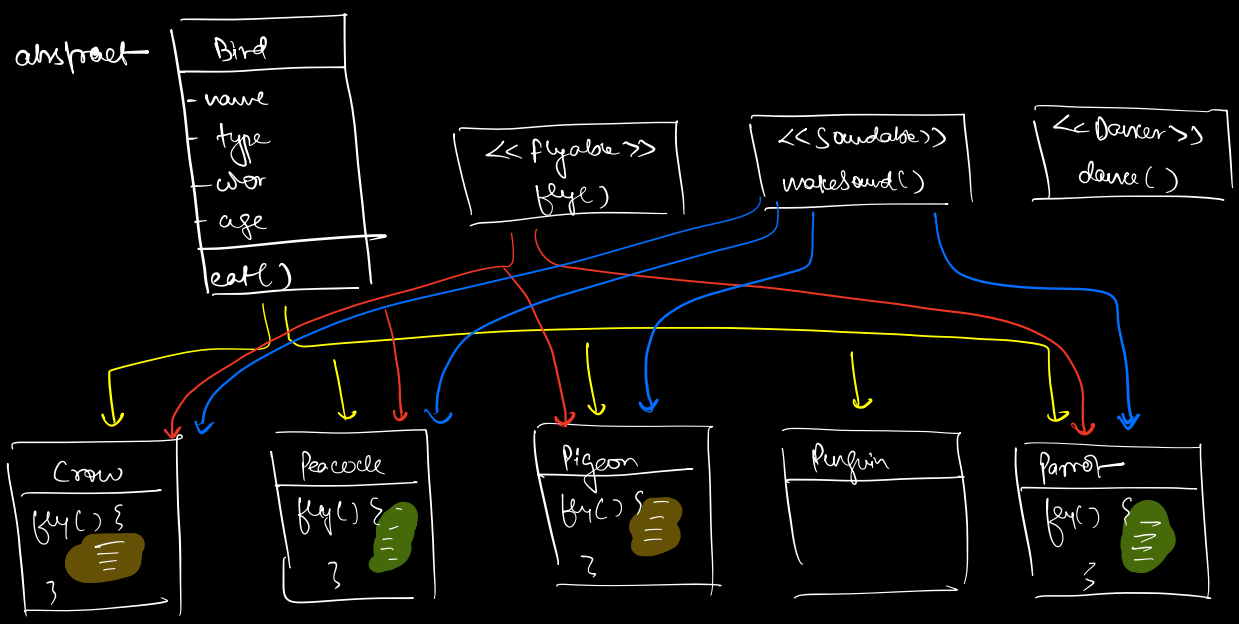
→ As less methods as possible

→ Ideally should have only 1 method.

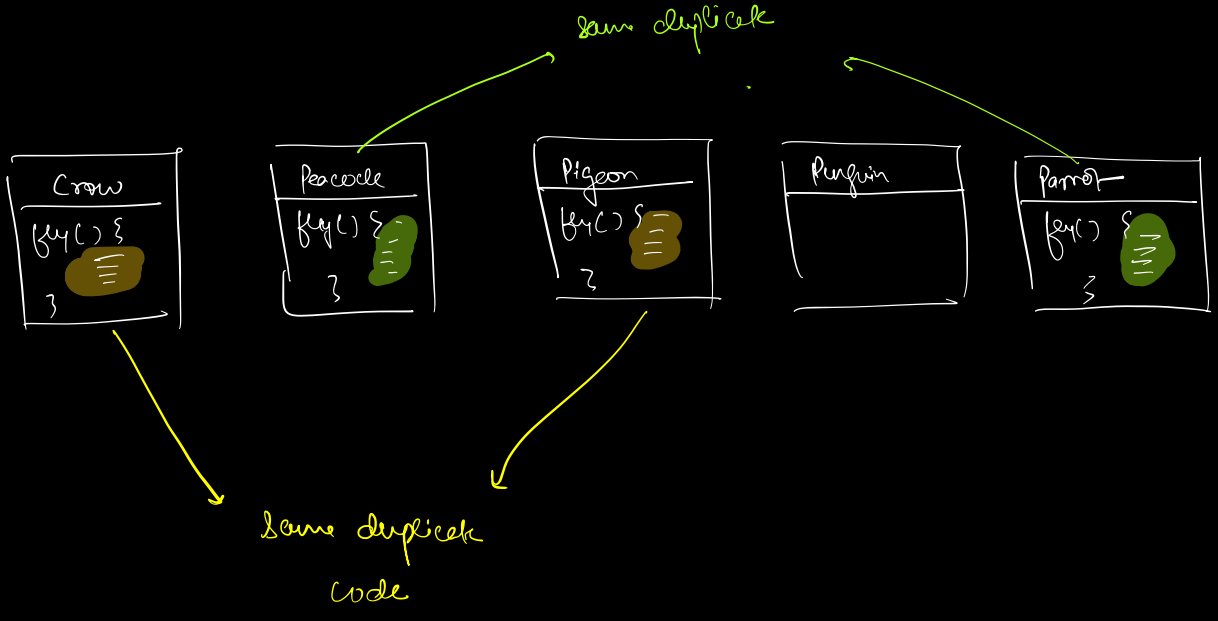
[⇒ methods in an interface, logically go together]

SRP → interfaces

=> Dependency Inversion Principle :-



Reg
1) Crow and Pigeon fly in exact same way
2) Peacock and Parrot fly in exact same way



SOln

CrowPigeonFlyBehaviour {

void makefly() {

}

PeacockParrotFlyBehaviour {

int doFly() {

}

Crow

```
CFFB cffb =
new CFFB();
fly() {
  cffb.makefly()
}
```

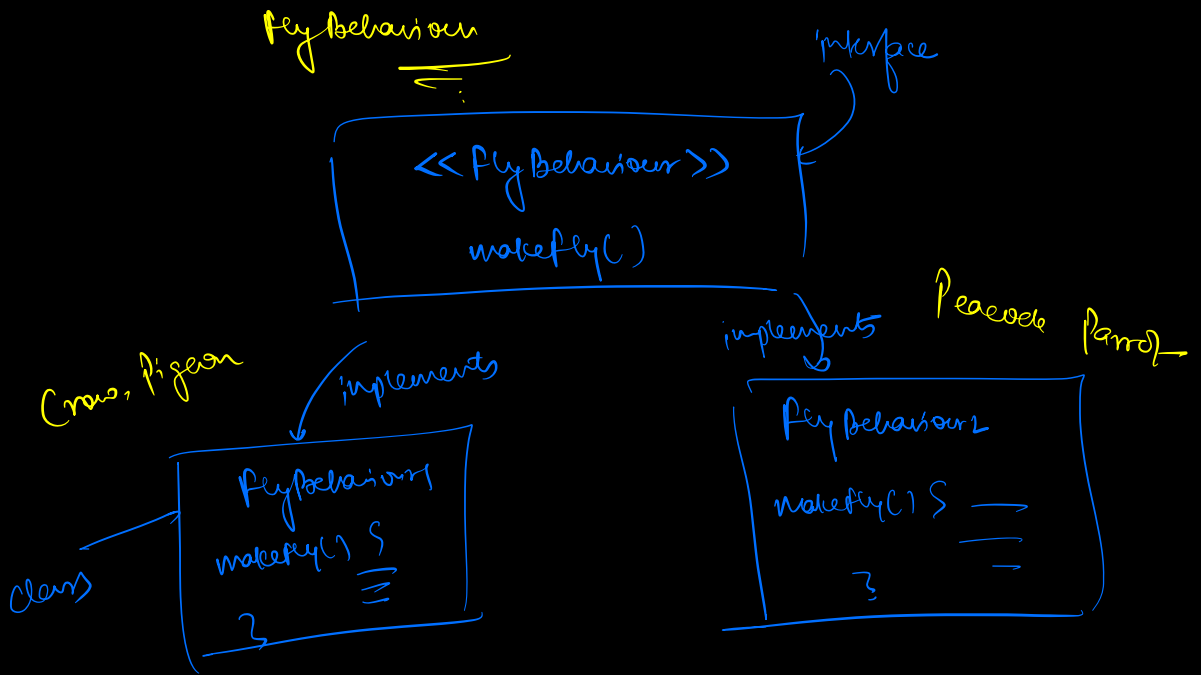
Pigeon

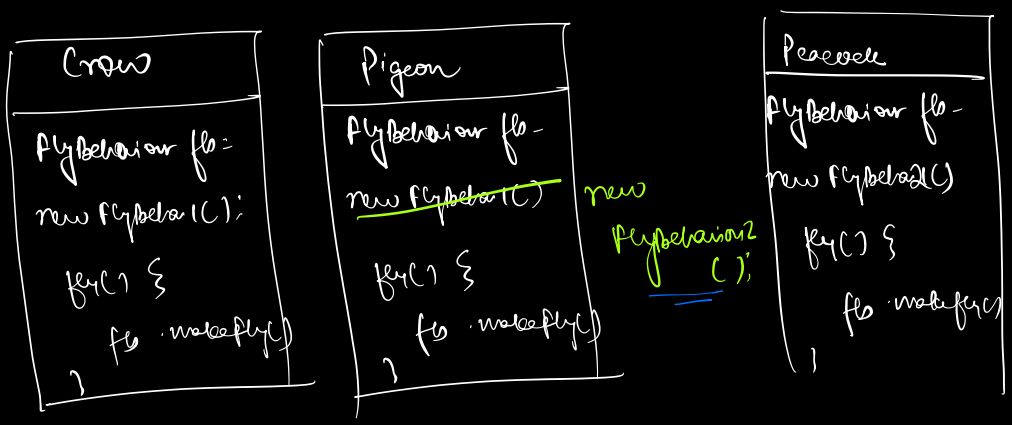
```
CFFB cffb =
new CFFB();
fly() {
  cffb.makefly()
}
```

Peacock

```
PPFB ppfb =
new PPFB()
fly() {
  ppfb.doFly()
}
```

depending upon concrete impl of FlyBehaviour





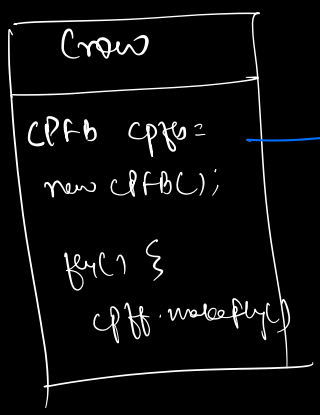
DI principle :- No 2 concrete classes should directly depend on each other, they should depend on each other via interfaces.

** Never depend on a specific company, only depend on having the ~~a~~ fly

⇒ make them loosely coupled

Dependency injection

Doubts



class Crow {

CPFb cpfb;

public Crow() {

cpfb = new CPFB();

}