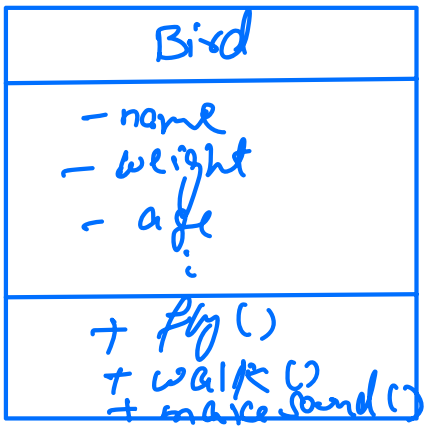
SOLID - 2

⊛ Agenda :-

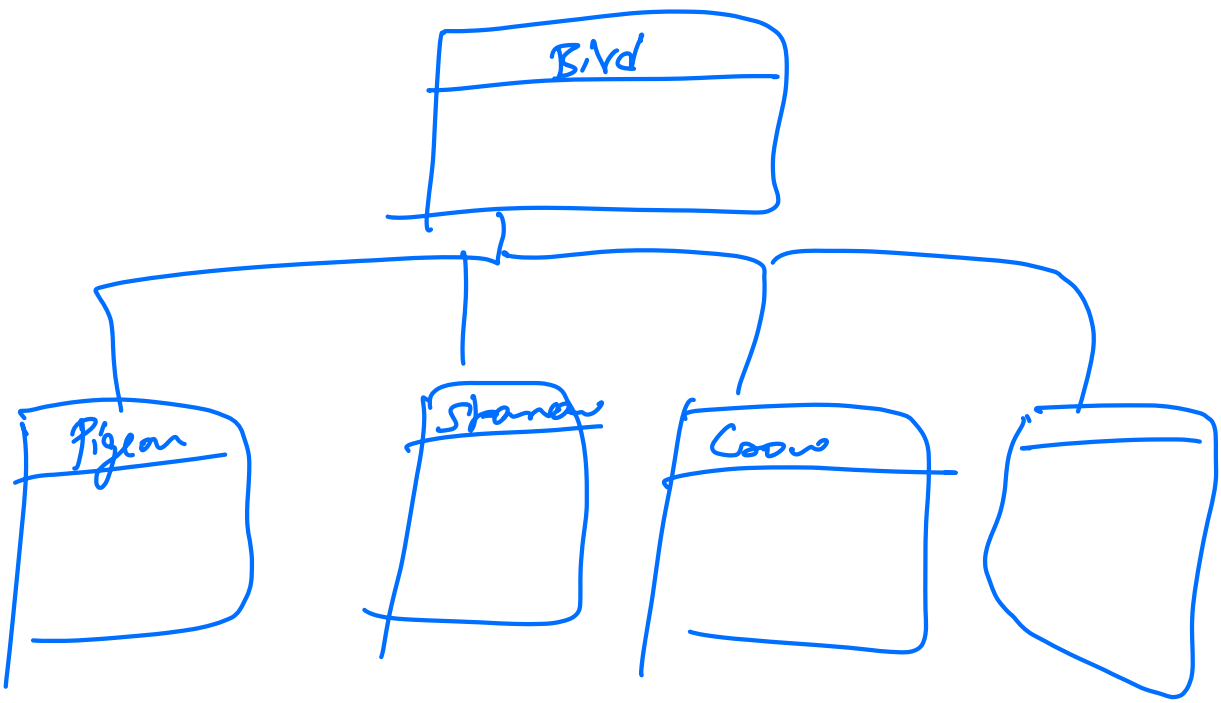
- ⊙ Liskov's Substitution Principle
- ⊙ Interface segregation Principle
- ⊙ Dependency Inversion Principle
- ⊙ Dependency Injection

⊛ Liskov's Substitution Principle (LSP) :-

V0



V1



SRP ✓
OCP ✓

Problem Statement :-

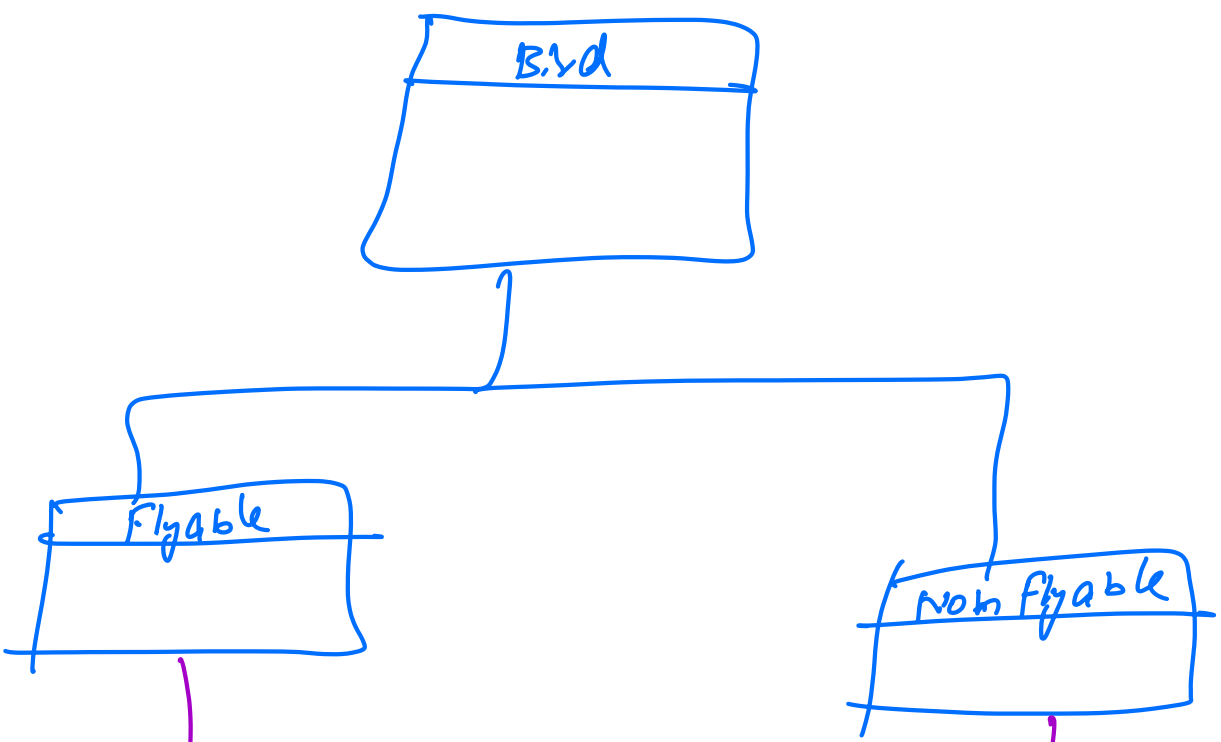
Some birds demonstrate a behaviour. while others don't.

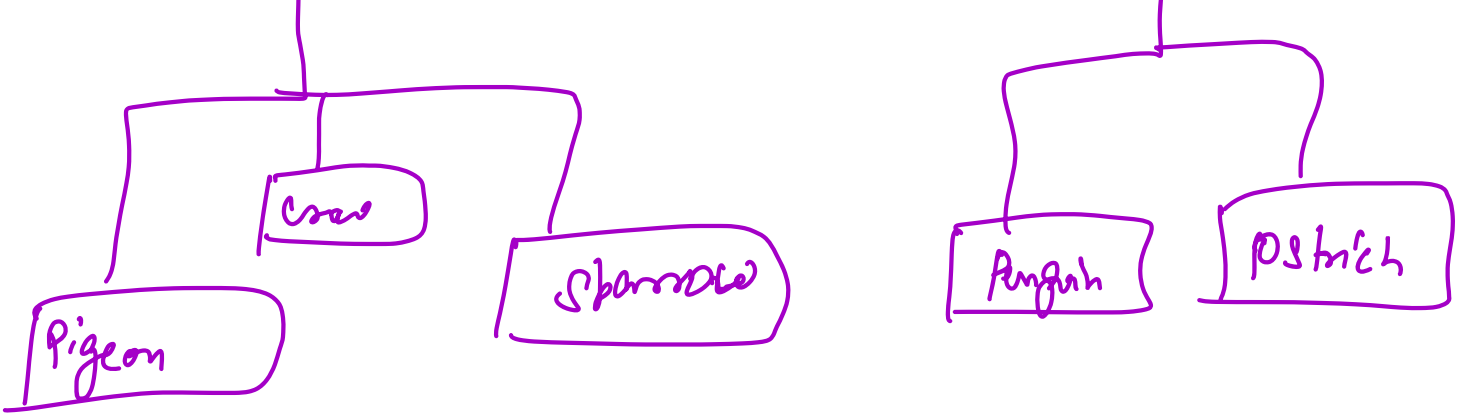
→ fly()

① What we want?

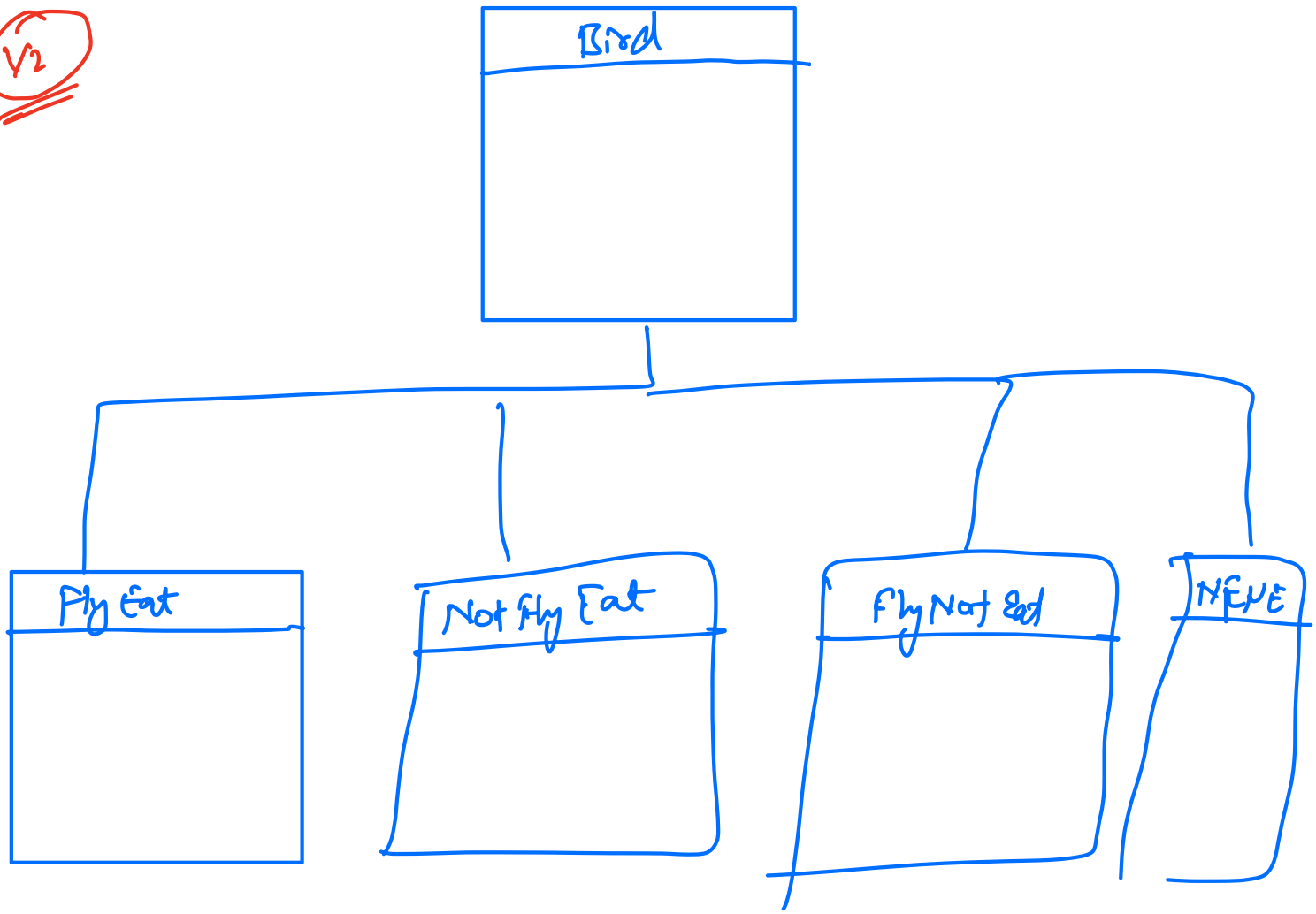
↳ only the birds which have that behaviour, should have that particular method.

→ 1. Create two child class ✗
2. Throw exception ✓
3. leave it blank.





V2



10 ← ~ X } 2 Combination

2^{10} = 1024 Classes.!

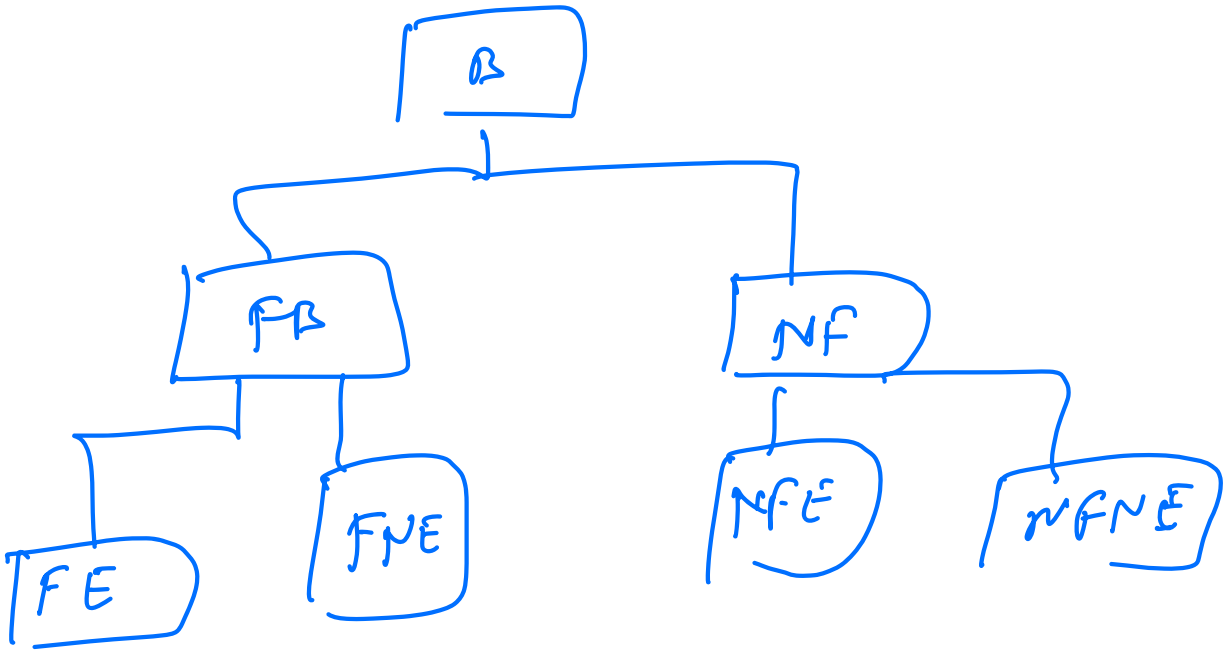
difficult to create these classes.

Problem 2:-

Let's say, someone wants to create a list of birds that can fly.

List < ? > birds = { }

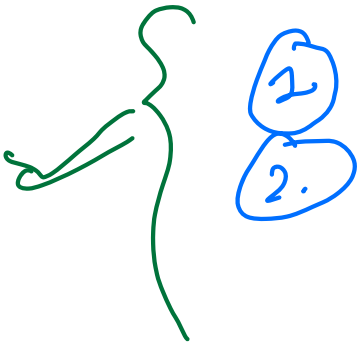
(IMPOSSIBLE!)



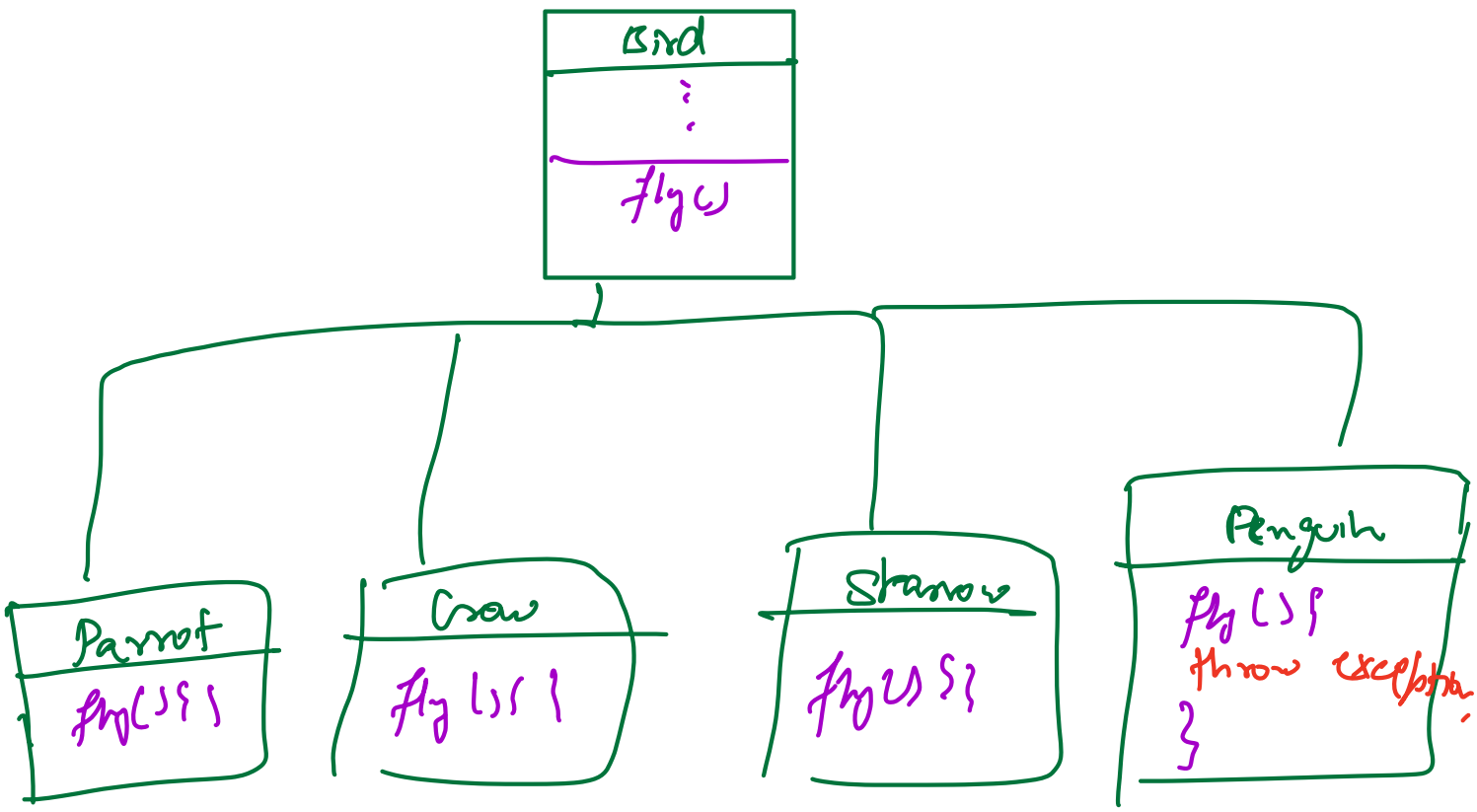
List<?> bird = { };

(Not Possible)

Problems :-

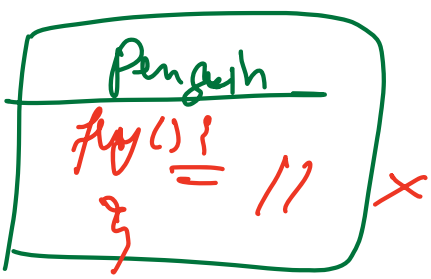


1. Class Explosion
2. Inability to recognise all the instances that demonstrate a particular behaviour.

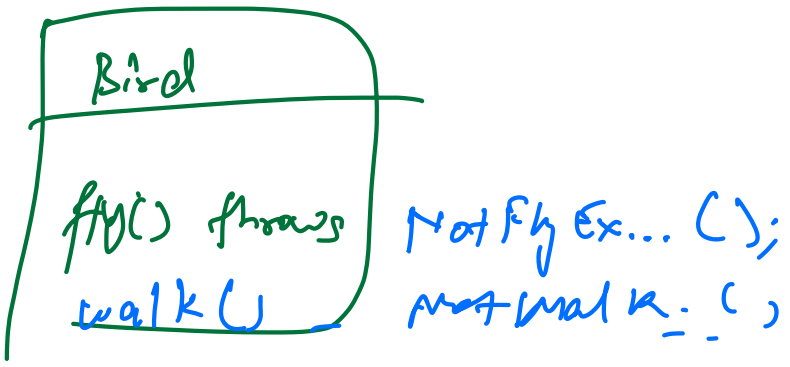


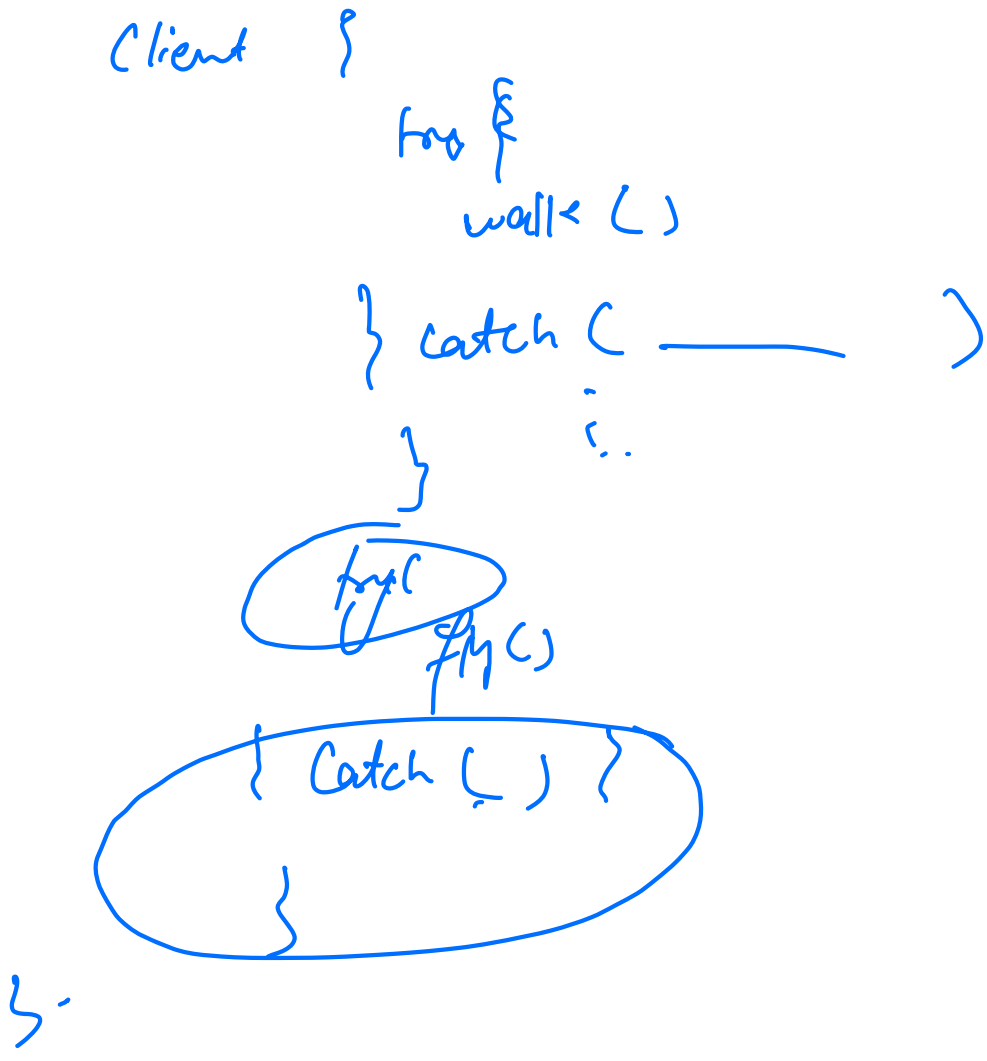
```
Client {
    psvm() {
        List<Bird> birds = { new Penguin() };
        for (Bird b : birds) {
            b.fly();
        }
    }
}
```

Try to minimise the surprises for your client :D



Liskov's Substitution Principle of SOLID's


Bird
fly() throws
walk()
NotFlyEx... ();
NotWalk...()


Client {
try {
walk()
} catch (
}
try
fly()
{ Catch ( ) }
}

⊛ Liskov's Substitution Principle (LSP):-

↳ Barbara Liskov.

↑ Mathematician

↳ Object of any Child class should be is-is substitutable in a variable of parent type without requiring any code changes.

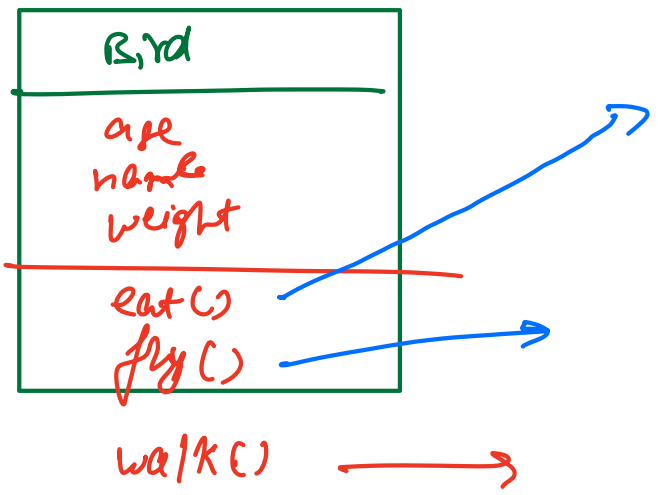
```
Bird bird = new Crow();
bird.fly()
```

① No special treatment should be given to child class.

```
Bird bird = new Penguin();
bird.fly();   ← ✗ LSP.
```

↑

Objects of child class should behave exactly how their Parents wants them to.

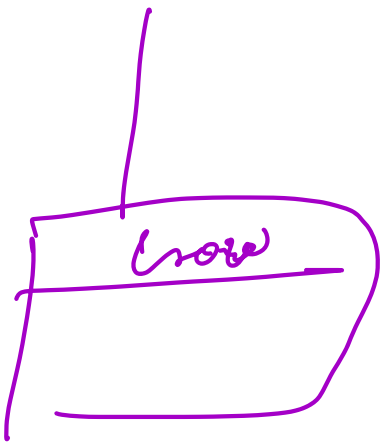


Documentation

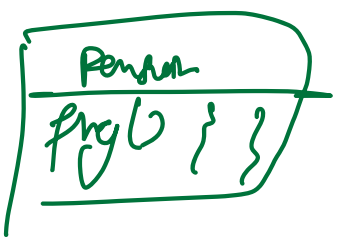
This method makes the bird eat food.

This method makes the bird fly.

This method makes the bird walk



crow.fly() ✓



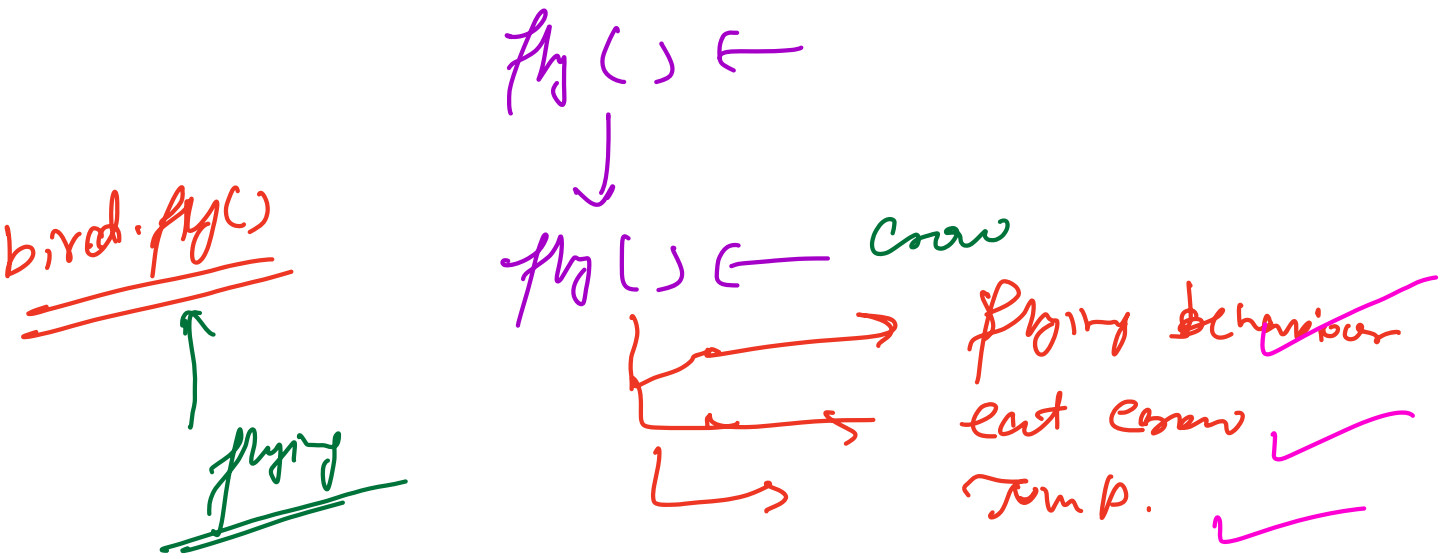
Penguin.fly() ✗

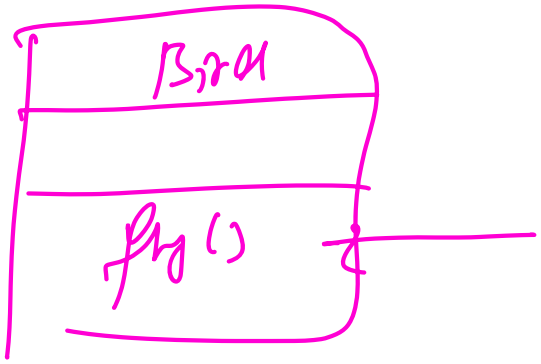
LSP

Bad!

↳ methods should be overridden to do the same thing differently, but not to do a different thing.

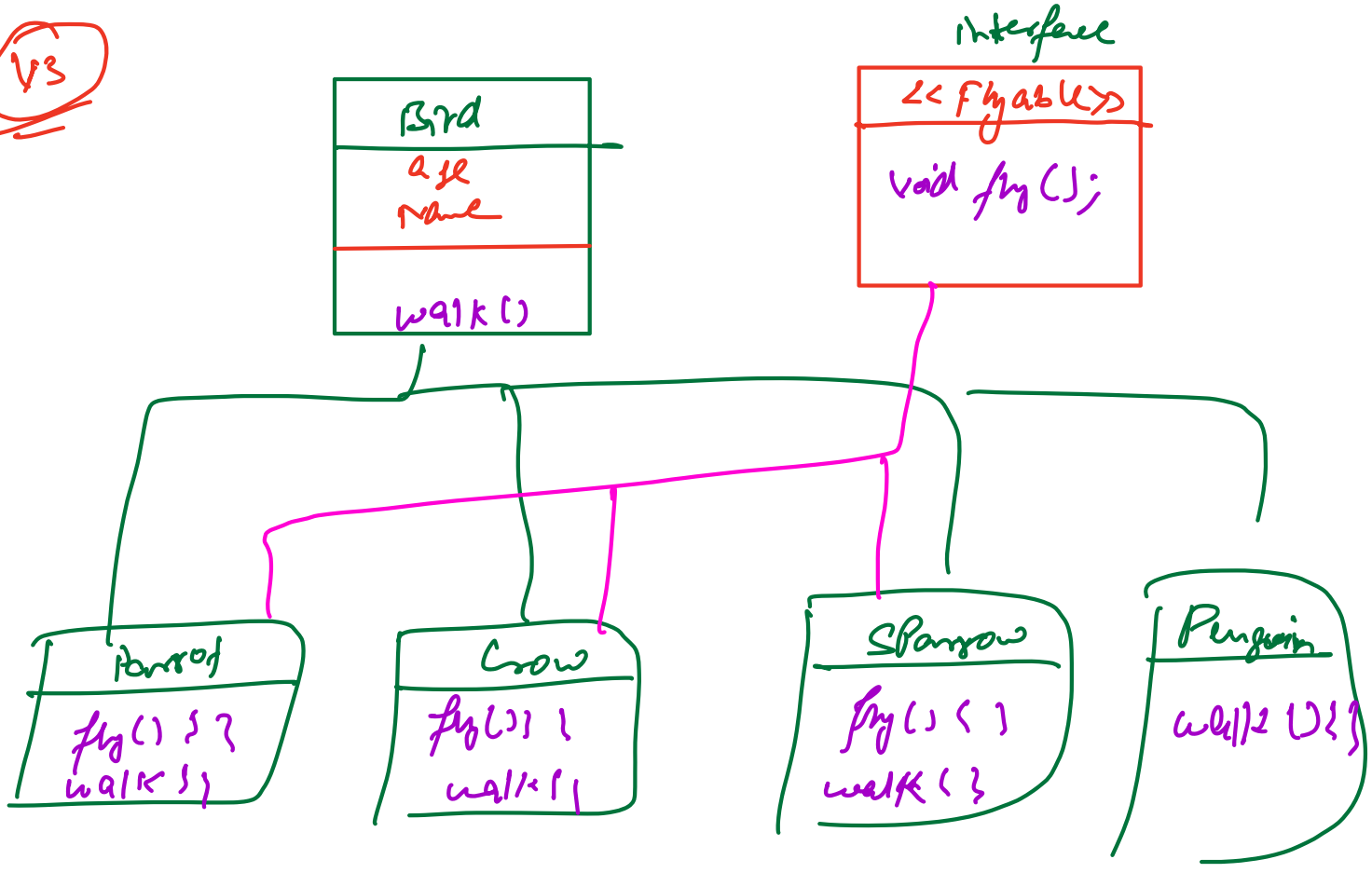
LSP → avoid surprises!





// interfaces.

V3



1. only the sub classes that represent the behaviors should have associated method. —

Class Crow extends Bird implements Flyable {

fly() { }

walk { }

}

10 ₹ $2^{10} = 1024$

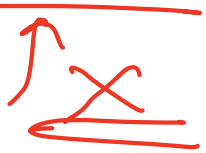
10 = Class Explosion

X

2. I should be able to create a list of birds that represents particulars behaviors..

```
List<Flyable> birds = { new Eagle(); ____ };
```

new Penguin()



```
for (Flyable b : birds) {
    b.fly();
    b.walk();
}
```

```
List<Bird> b = ____________;
```

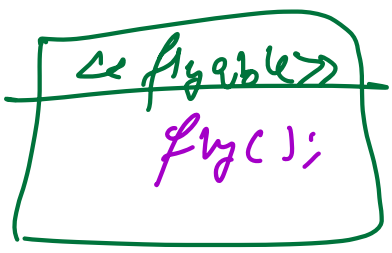
```
for (Bird b : birds) {
    Flyable.(Flyable) b;
    f.fly();
}
```

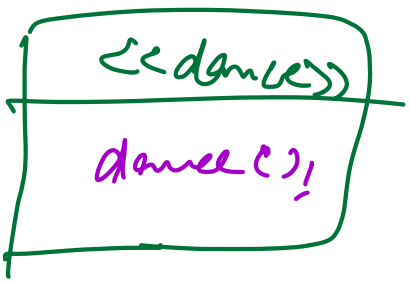
```
if (b instanceof Flyable) {
    ((Flyable)b).fly();
}
```

Break till
08:27 AM IST

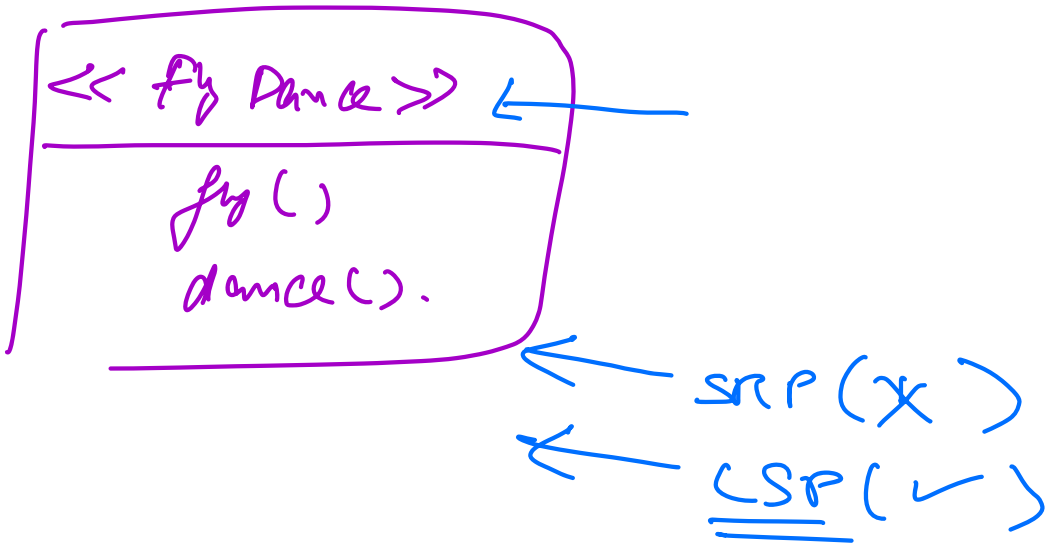
(*) Interface Segregation Principle:-

↳ Some birds can fly
↳ Some birds can dance.
↳ All the birds who can fly can dance & vice-versa.



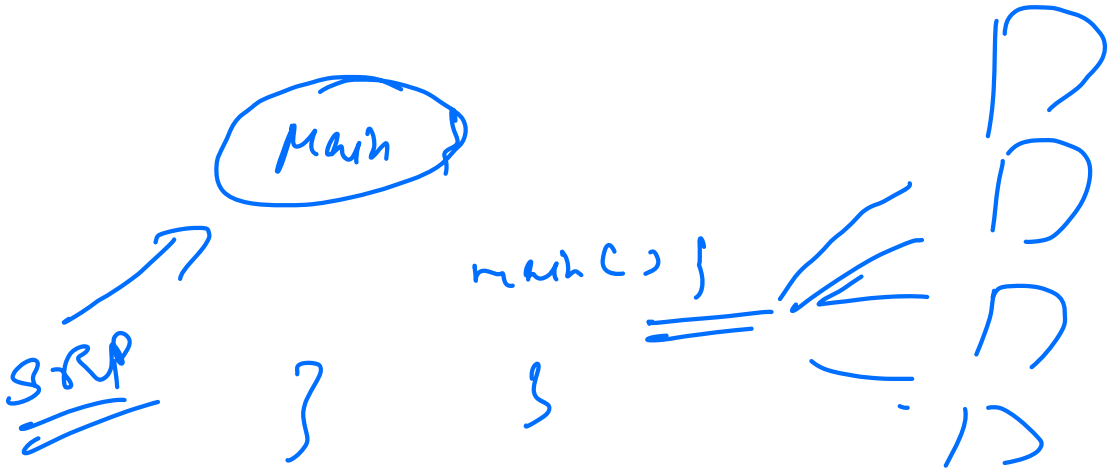


1.



2.

Class → fly, dance;



ISP

↳ ⊙ Interfaces should be as slight as possible.

⊙ Interfaces should have as less method as possible.

⊙ Ideally, interface should have only one method.

↳ Functional Interfaces.

⊙ Interface can have more than one method only iff all of the methods are logically related to each other.

⊙ ISP is also known as SRP for interfaces.

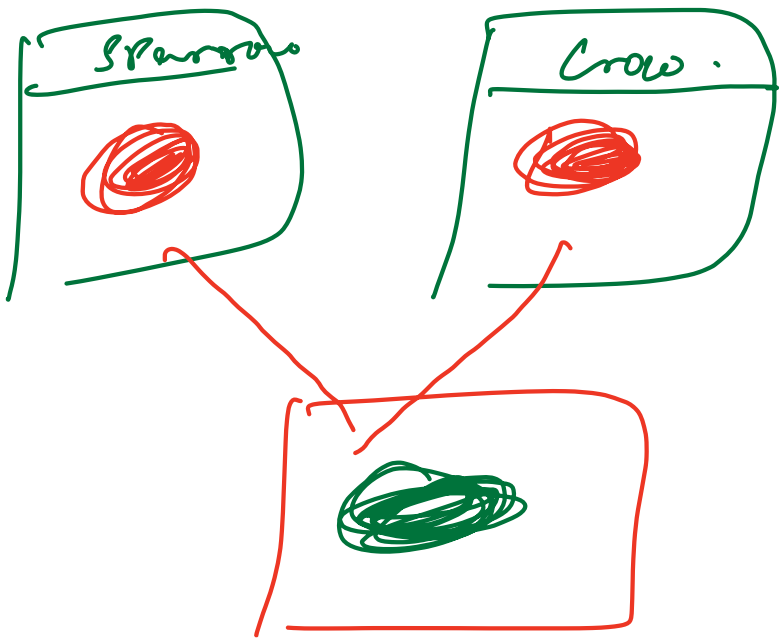
⊙ Does every bird fly in a different way?

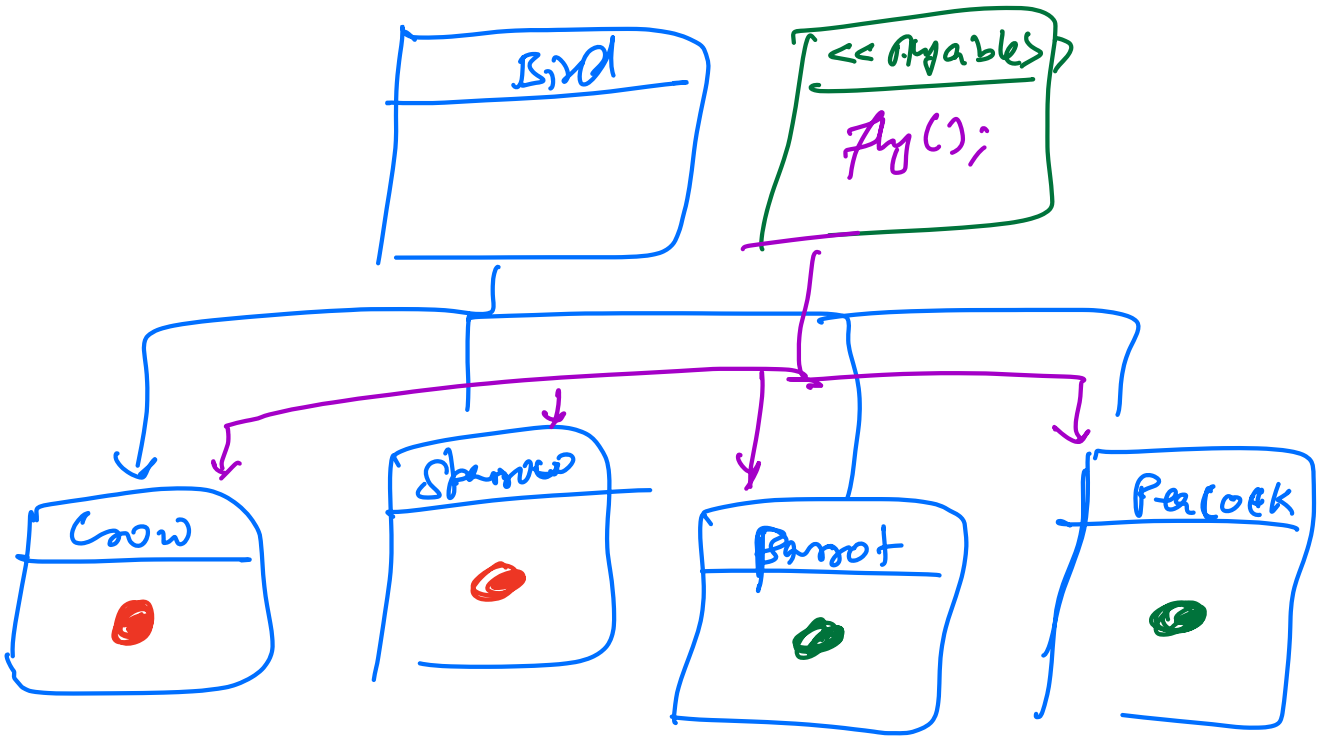
→ No!

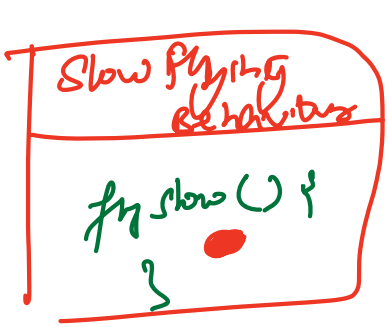
→ Sparrow and Crow flies in same way.

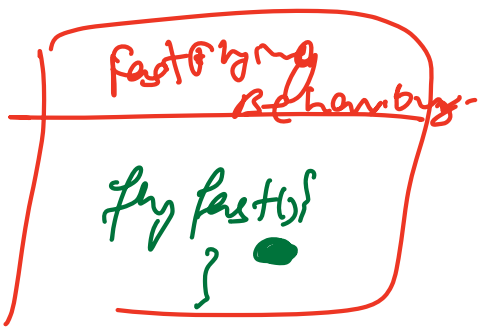
→ Parrot and Peacock fly in same way.

Code Duplicacy!









● → flySlow

● → flyFast.

```
Class Crow extends Bird
                implements Flyable {

    SFB sfb = new SFB();

    fly() {
        sfb.flySlow();
    }
}
```

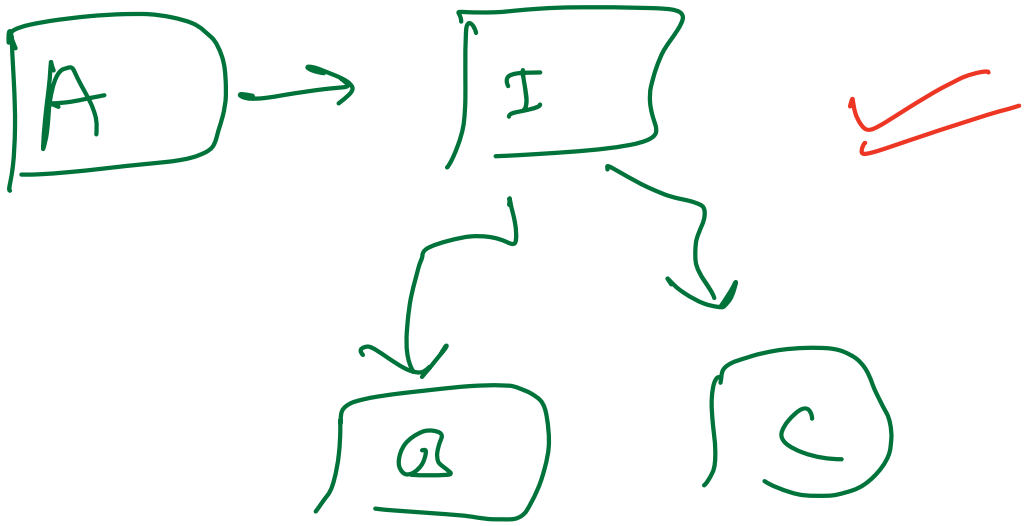
Always code to interface, not to an implementation

DIP → No two concrete classes should depend directly upon each other.

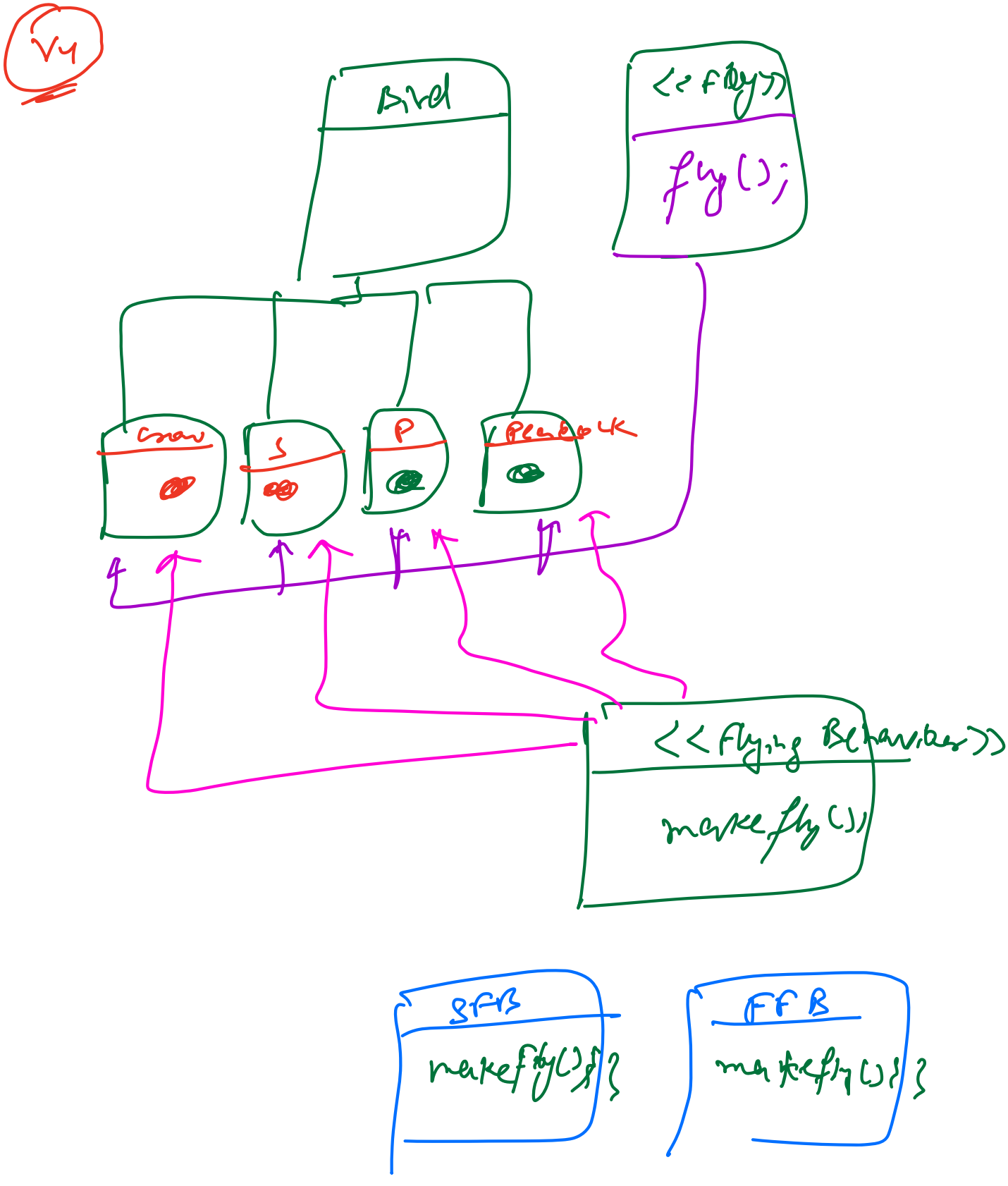
Dependency Inversion Principle.

⊙ They should depend upon interfaces.

Bad !



Better


V4
Bird
<< Fly >>
fly();
Crow
S
P
Peacock
<< Flying Behavior >>
makeFly();
SFB
makeFly(){}
FFB
makeFly(){}

```
Class Crow extends Bird
                implements Flyable {

    FlyingBehaviour fb = new SFB();

    fly() {
        fb.makefly();
    }
}
```

(*) Dependency Injection :-

↳ NOT part of SOLID.

↳ No need to create an Object of dependency self.

↳ Instead, let the creator of your class pass the dependency.

↳ Constructor

↳ setter

.

vs

```
Class Crow extends Bird
                implements Flyable {

    FlyingBehaviour fb;

    Crow (FlyingBehaviour fb) {
        this.fb = fb;
    }

    fly() {
        fb.flySlow();
    }
}
```

Benefit of Pependency Injection.

↳ Testing becomes easy.

↳ loose coupling.

H/w

Draw V0 → V5 of Bird by yourself.