SOLID – 2

V0

Bird

≡

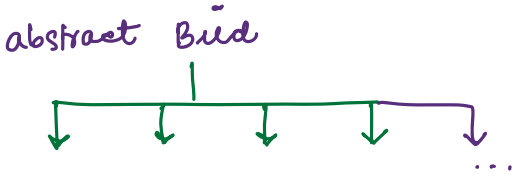
fly()

makeSound()

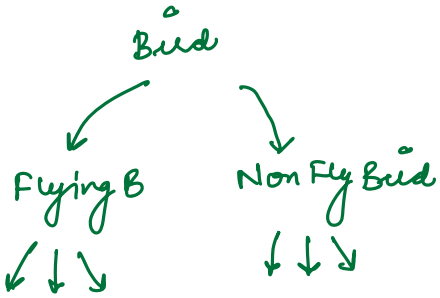
SRP ✗

OCP ✗

V1



V2



4

$2^4 = 16$

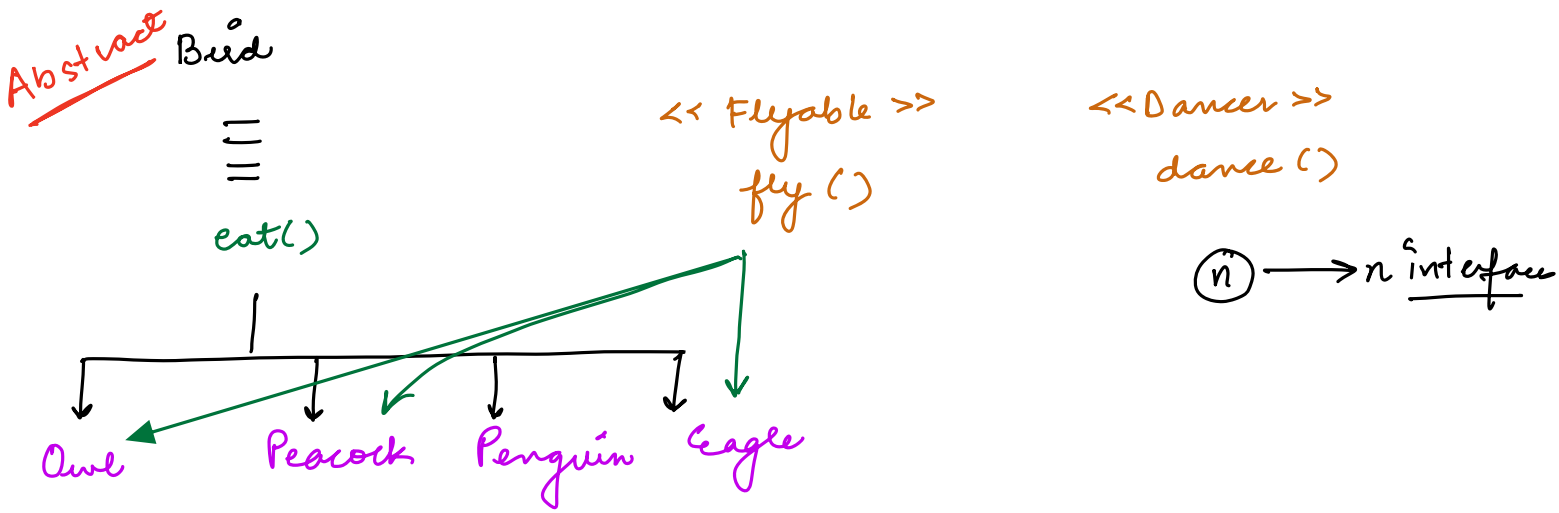
✓ ✗

Problem statement :- Some birds exhibit a particular behaviour and some don't

List < Birds who can fly > x

classes : entity

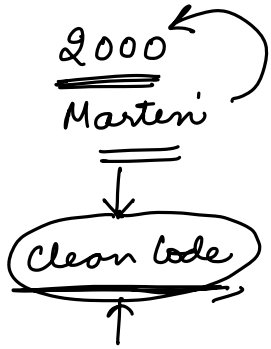
Interfaces : behaviour



Liskov's substitution Principle

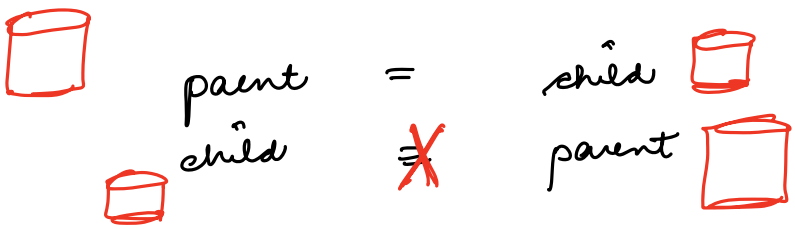
> Object of any sub-class should be as-IS substitutable in the parent class without any code change.

All the child classes should behave as their parent.



Bird b = new Penguin();
new Peacock();
new Crow();
new Sparrow();
...

b.fly();



Interface segregation Principle

- Some Birds can fly
- Some Birds can dance
- All the birds that can fly can also dance & vice-versa.

Parents — 1 rule — 2 — child

Not.

300+

240 - 32

<< List >>
- add()
- remove()
- search()

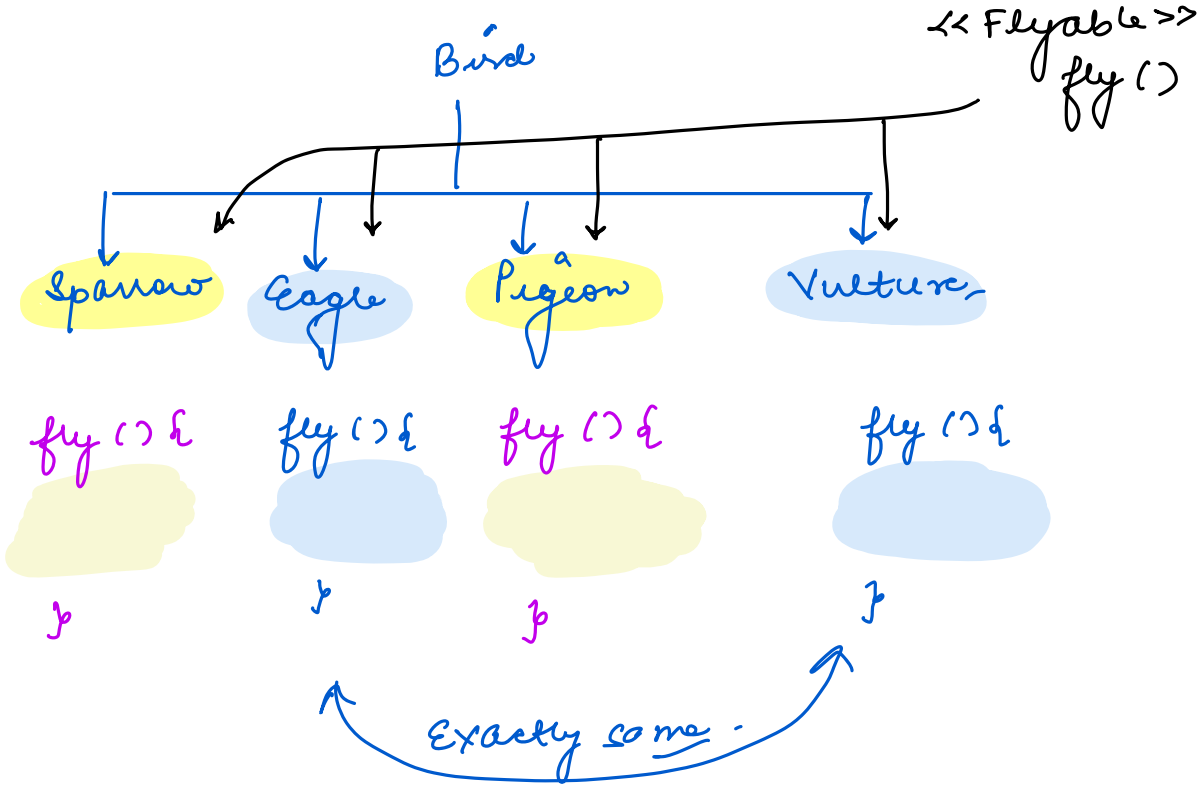
<< FlyDance >>
- fly()
- dance()

<< Flyable >>
- fly()

<< Dance >>
- dance()

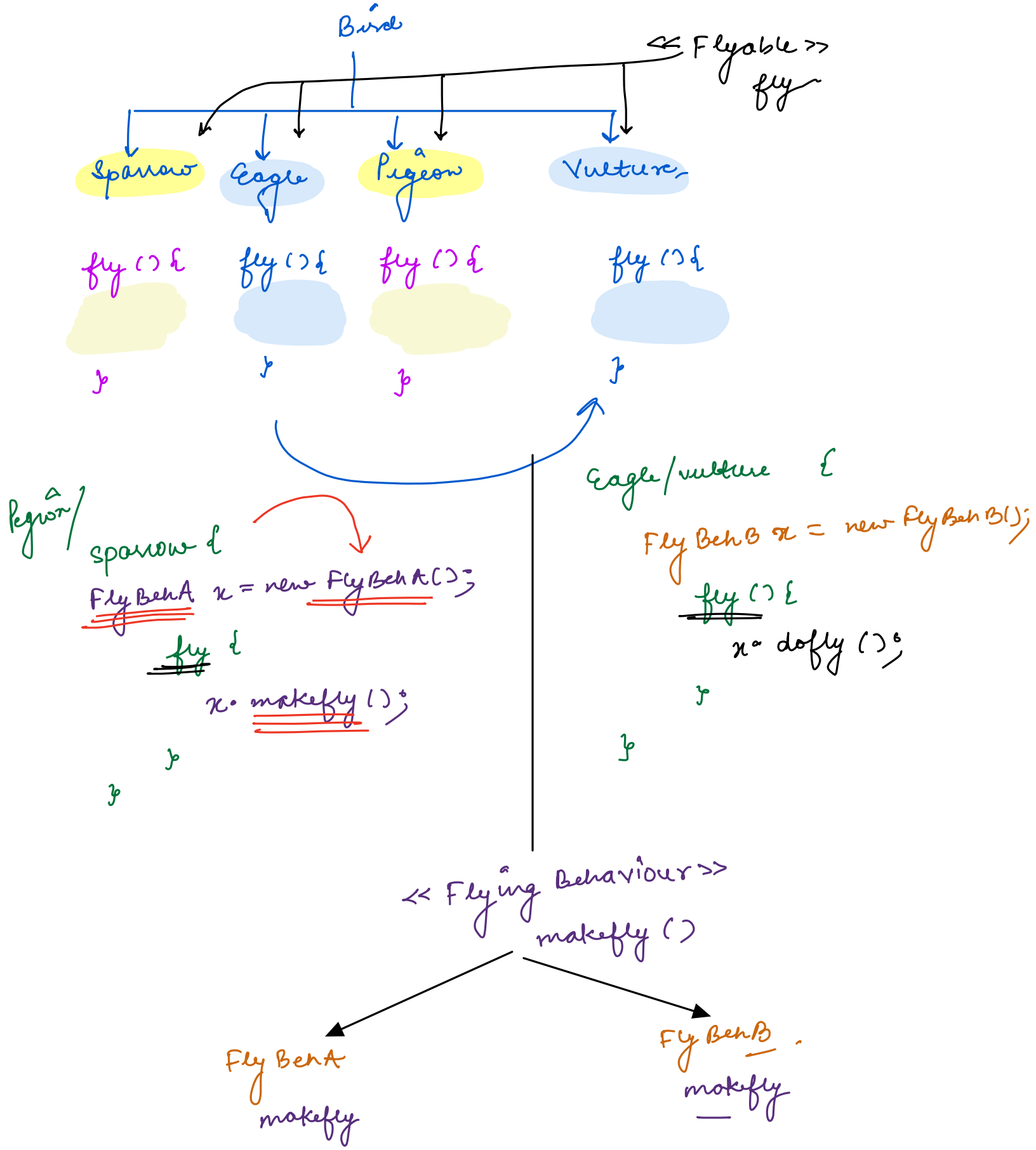
- Interfaces should be as light (→ less methods) as possible
- Ideally interfaces should have a single method.

Dependency Invcusion Principle



class FlyBehA {
makefly() {
≡
}
}

class FlyBehB {
dofly {
≡
}
}


Bird
<< Flyable >>
fly
Sparrow
Eagle
Pigeon
Vulture
fly () {
}
Pigeon/
sparrow {
FlyBehA x = new FlyBehA();
fly {
x.makefly();
}
}
Eagle/vulture {
FlyBehB x = new FlyBehB();
fly () {
x.dofly();
}
}
<< Flying Behaviour >>
makefly()
FlyBehA
makefly
FlyBehB
makefly

Pigeon / sparrow {

FlyBehaviour x = new FlyBehA();

fly {

x.makefly();

}

}

Eagle / vulture {

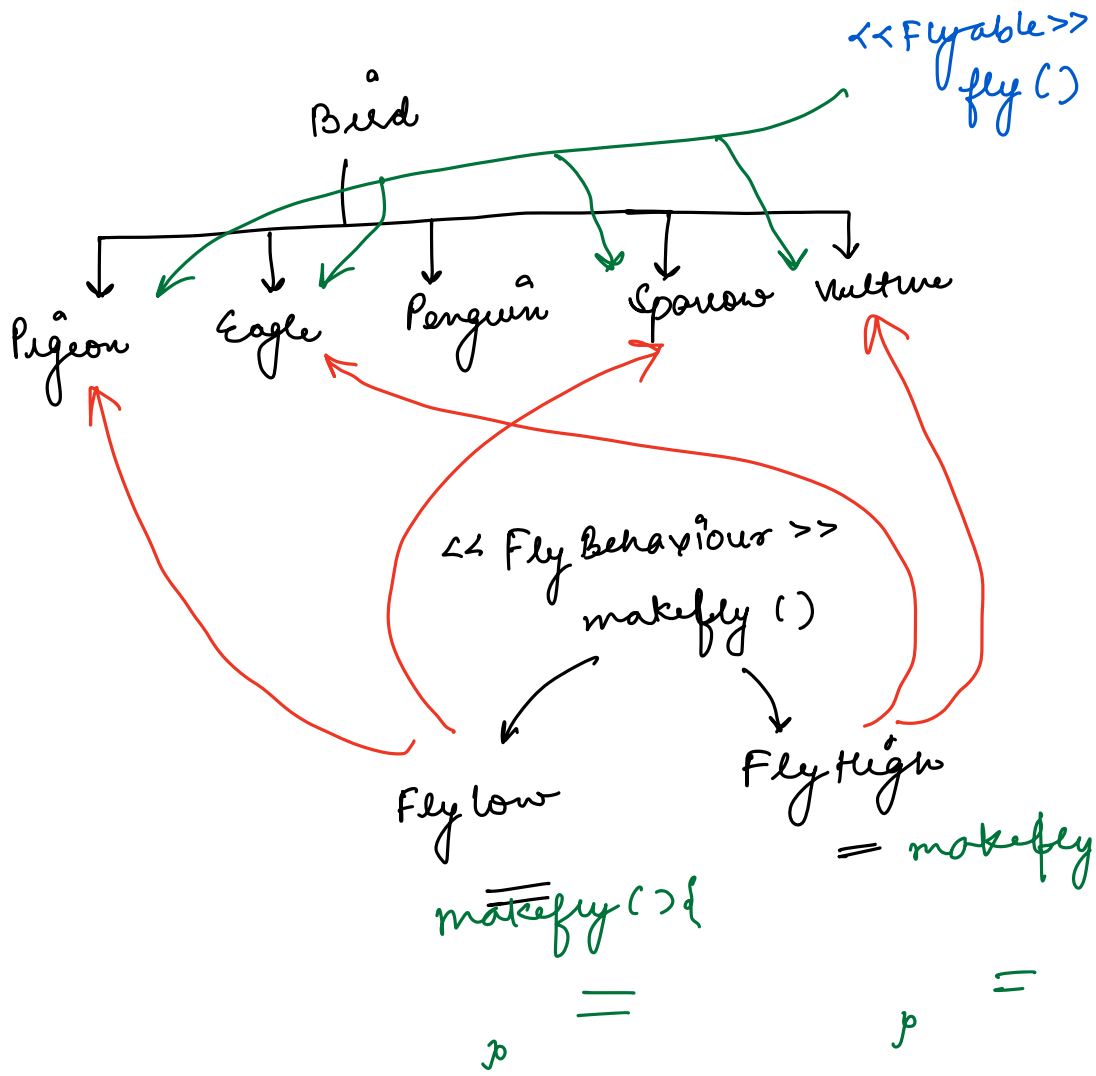
FlyBehaviour x = new FlyBehB();

fly() {

x.makefly();

}

}



makefly() {

=

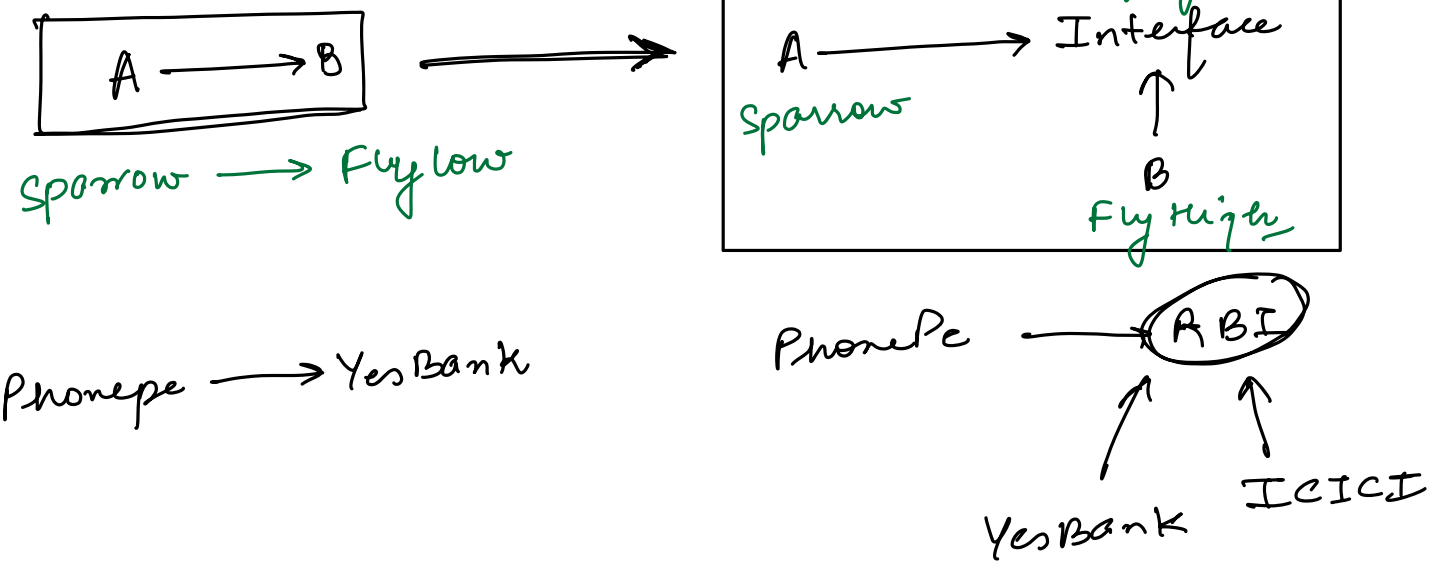
}

makefly() {

=

}

— No 2 classes should be directly dependent on each other, instead they should be dependent via interface



MCQ's — classes — weekend — reattempt
→ PSP ↑ Backend

List < ? extends Flyable > birds ;