Agenda:

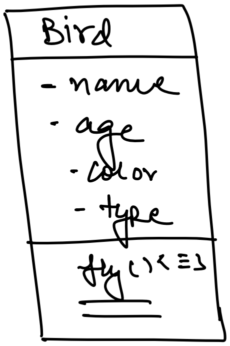
→ Recap

→ Liskov's Substitution

→ Interface Segregation

→ Dependency Inversion.

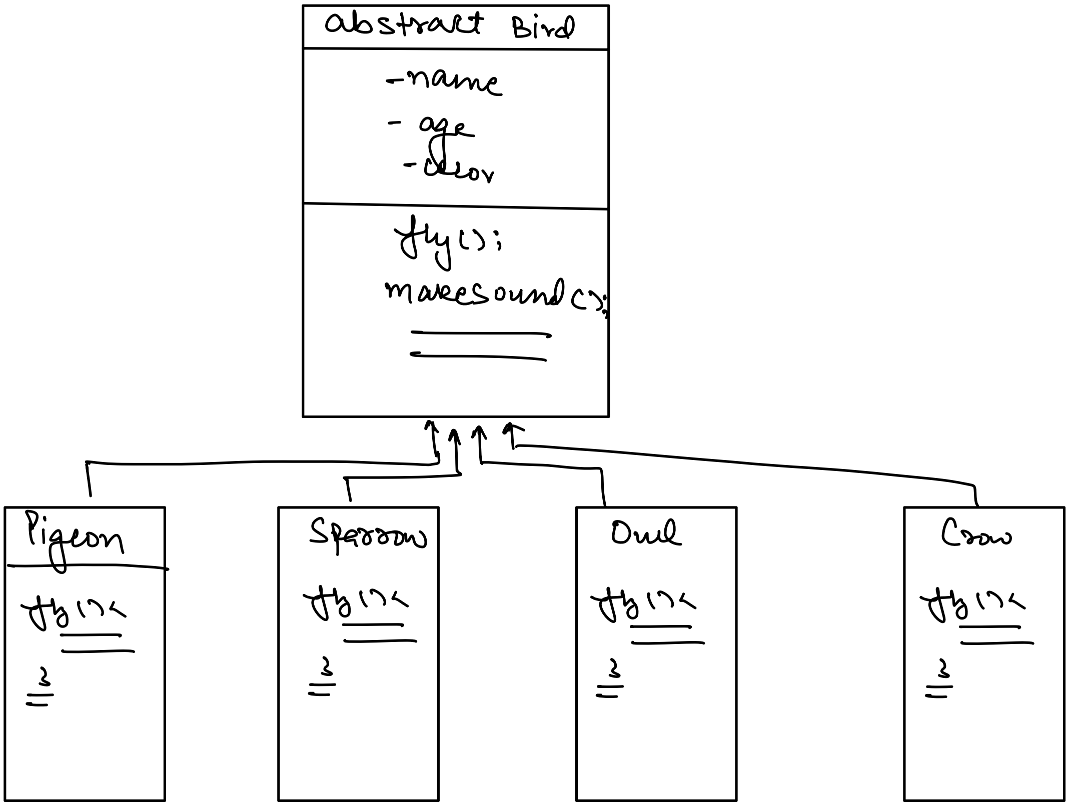
V1



SRP ×

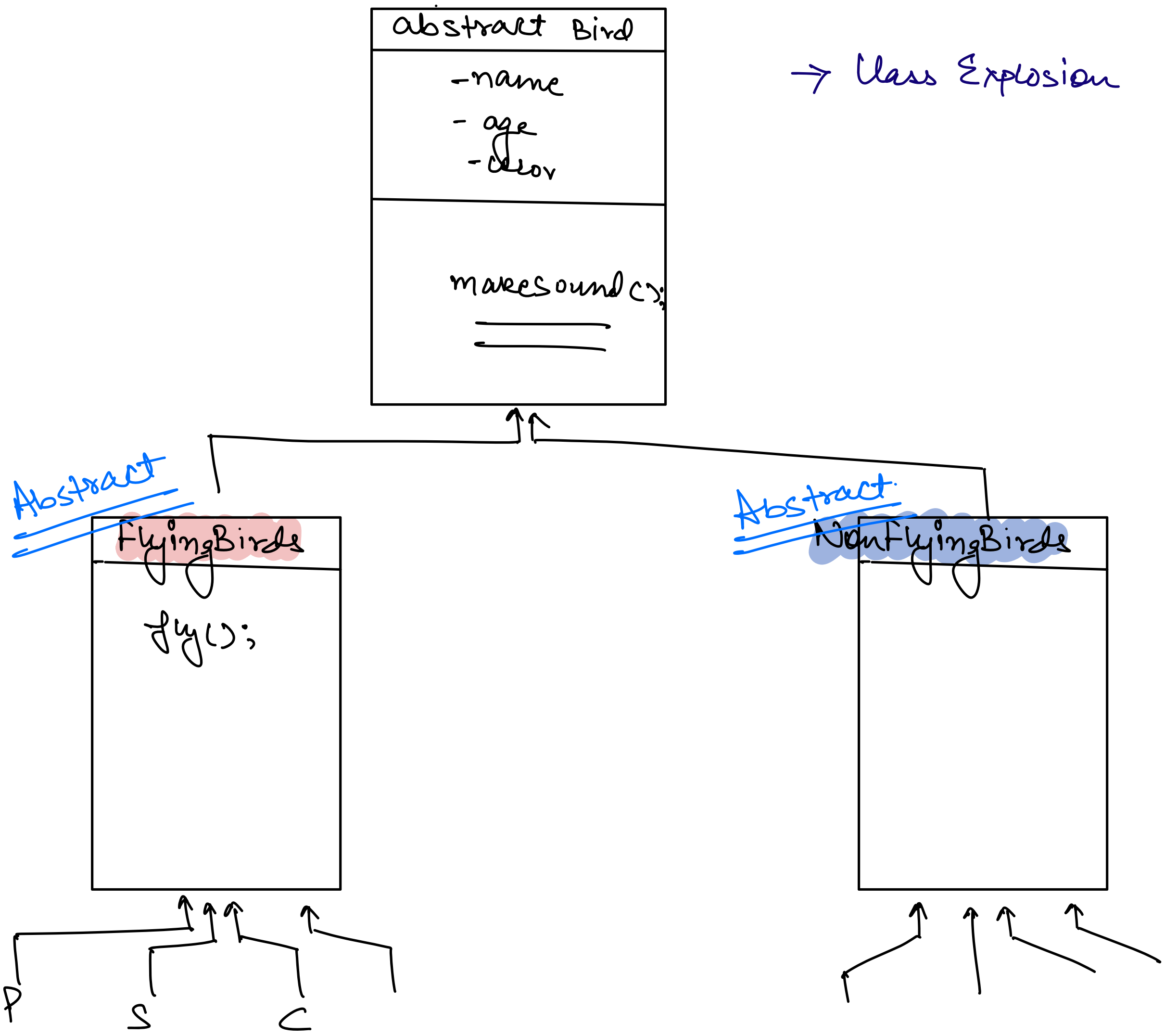
OCP ×

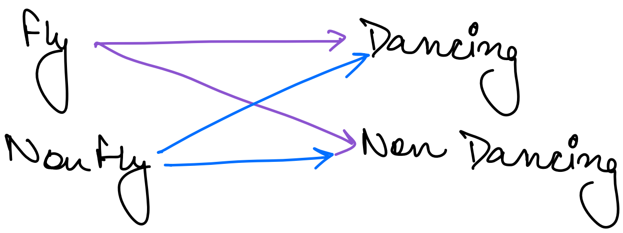
V2



Penguin => Can't ~~fly~~.

→ Class Explosion





List<FlyingBirds> ______

Problem Statement

⇒ Some birds demonstrate a behaviour while others are not.

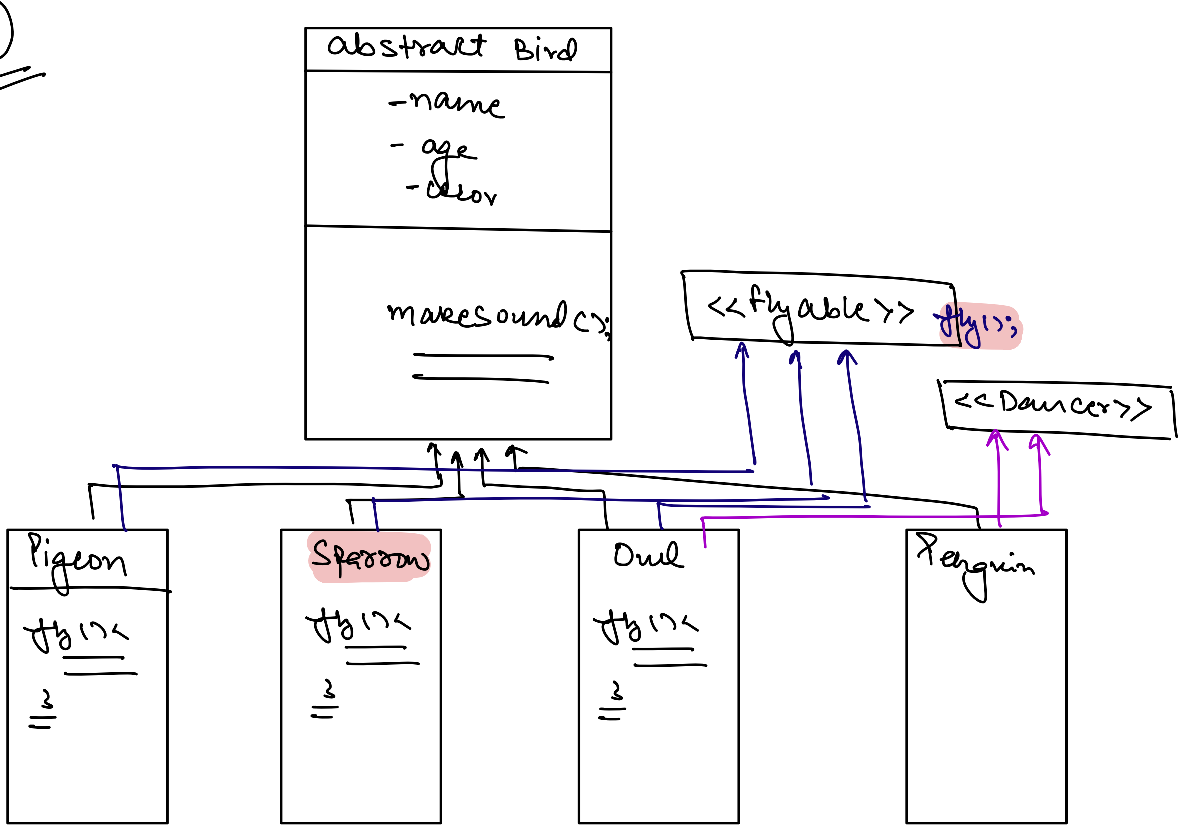
1. Only the birds having a behavior should have that method.

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.

⇒ Liskovs 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 = ne— Sparrow()

Class Sparrow extends Birds implements flyable {

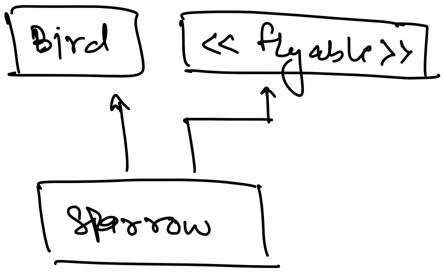
Bird b = Sparrow()

flyable f = Sparrow.

fly() {

}

}



```
Class Penguin extends Birds {

    ______
    ______

}
```

⇒ Interface Segregation Principle.

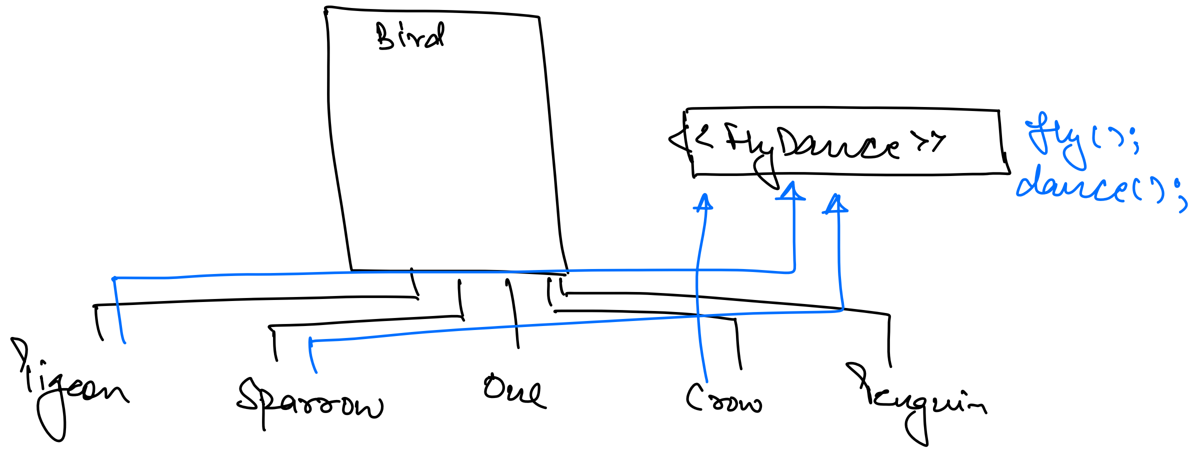
Req: Some birds can fly.

Some birds can dance.

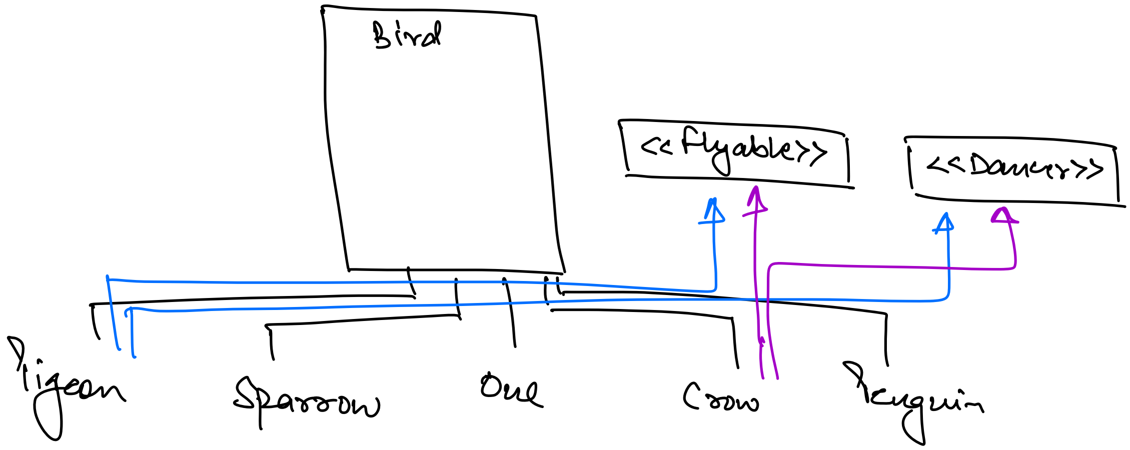
All birds who can fly, they can dance as well & vice-versa.

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

~~I~~



② ✓



Interface Segregation Principle.

→ Interfaces should be as light as possible..

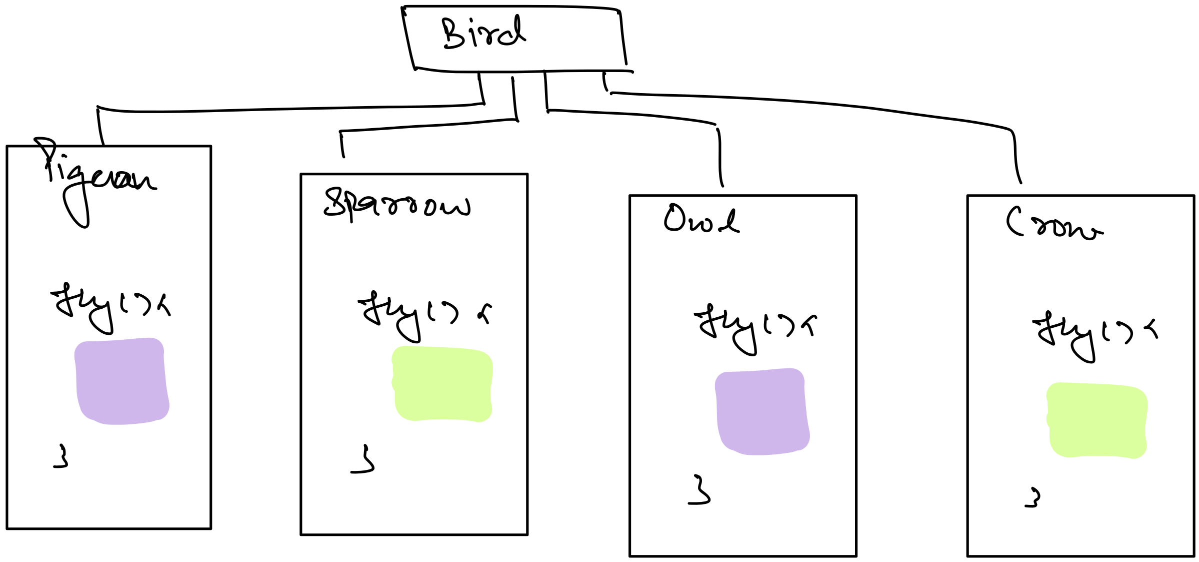
→ As less methods as possible.

→ Ideally interfaces should have a single behaviour.

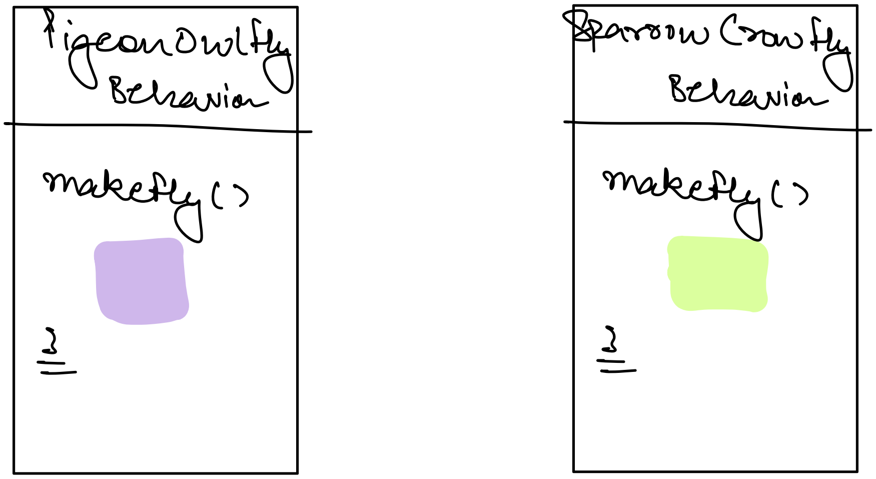
⇒ Functial Interface. ⇒ Interface with single method.

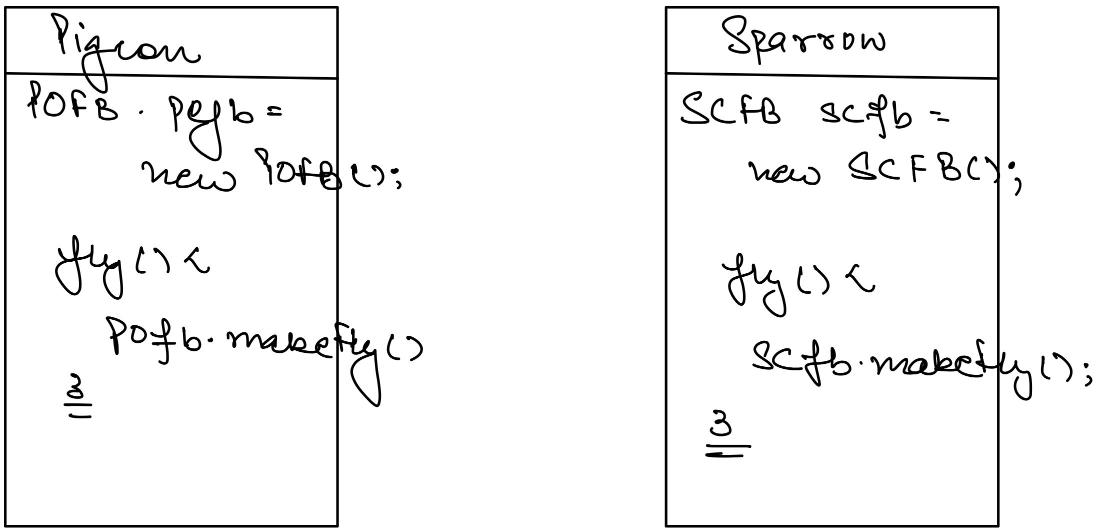
⇒ SRP on interface.

Dependency Inversion Principle.



⇒ Code duplicacy



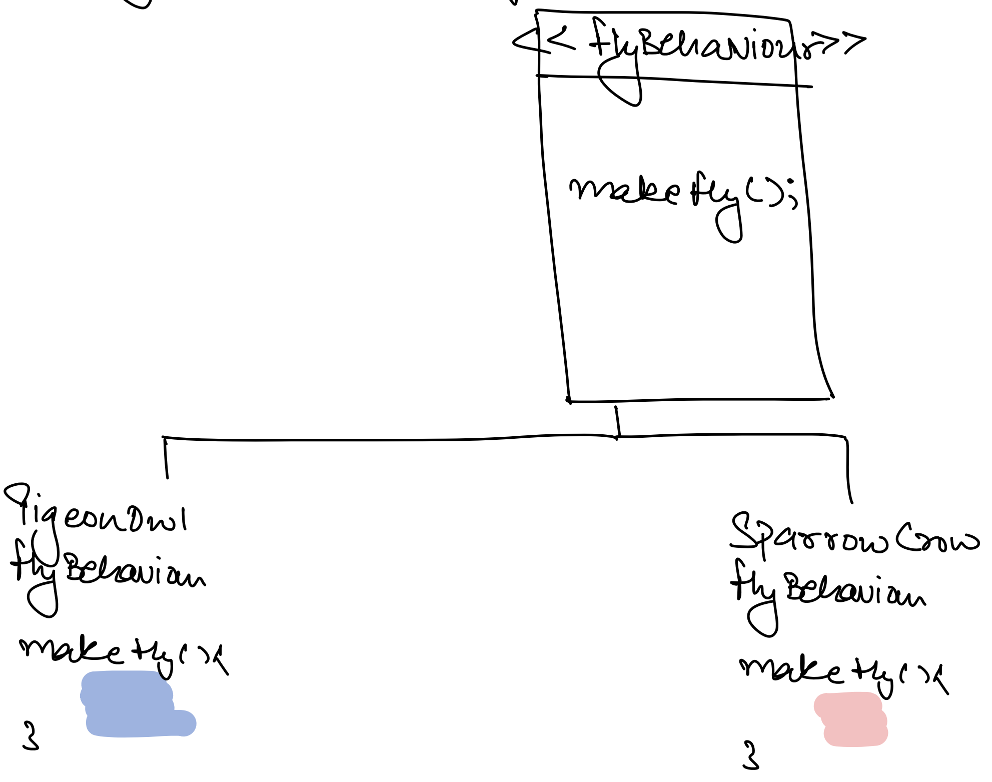


```
Pigeon
POFB · pofb =
      new POFB();

fly() {
   pofb.makefly()
}
```

```
Sparrow
SCFB scfb =
   new SCFB();

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

⇒ No Code duplicacy

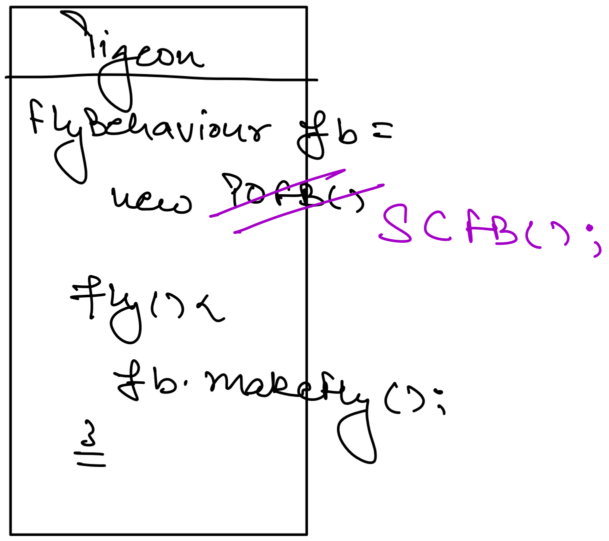
⇒ Program to interface, not to implementation



```
<< flyBehaviour >>
makefly();
```

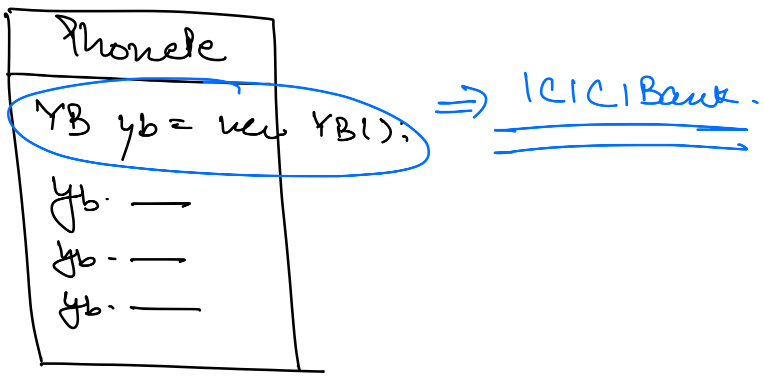
```
PigeonOwl
flyBehaviour
makefly(){
}
```

```
SparrowCrow
flyBehaviour
makefly(){
}
```

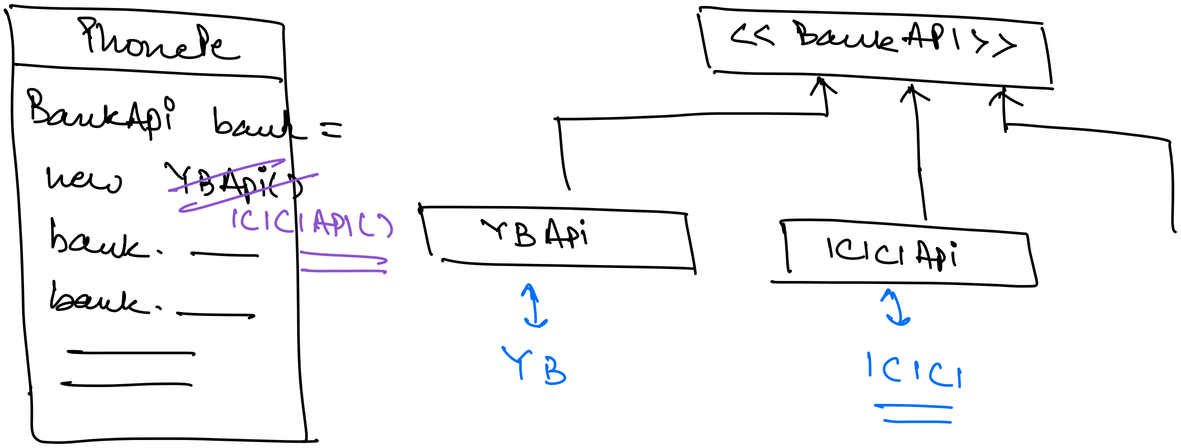


```
FlyBehaviour fb =
    new ~~POFB()~~ SCFB();

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

⇒



```
YB yb = new YB();   ⇒ ICICIBank.
  yb. ___
  yb. ___
  yb. ___
```



```
BankApi bank =
  new ~~YBApi()~~ ICICIAPI()
  bank. ___
  bank. ___
  ______
  ______
```

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.

Class ↓ ↙ Interface

→ If a class (A) is dependent on (B).

```
Class A {
      B  b = new ✗()
      ──
      ──
}
```

```
⇒   Pigeon {
          FlyBehaviour fb =  new  POFB();  ✗
          ________
     }

     Pigeon {
          FlyBehaviour fb;
Constructor { Pigeon( FlyBehaviour  obj){
                   this.fb = obj;
              }

          ________
          ________
     }

     PigeonOwlFlyBehavior  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.

——— * ———