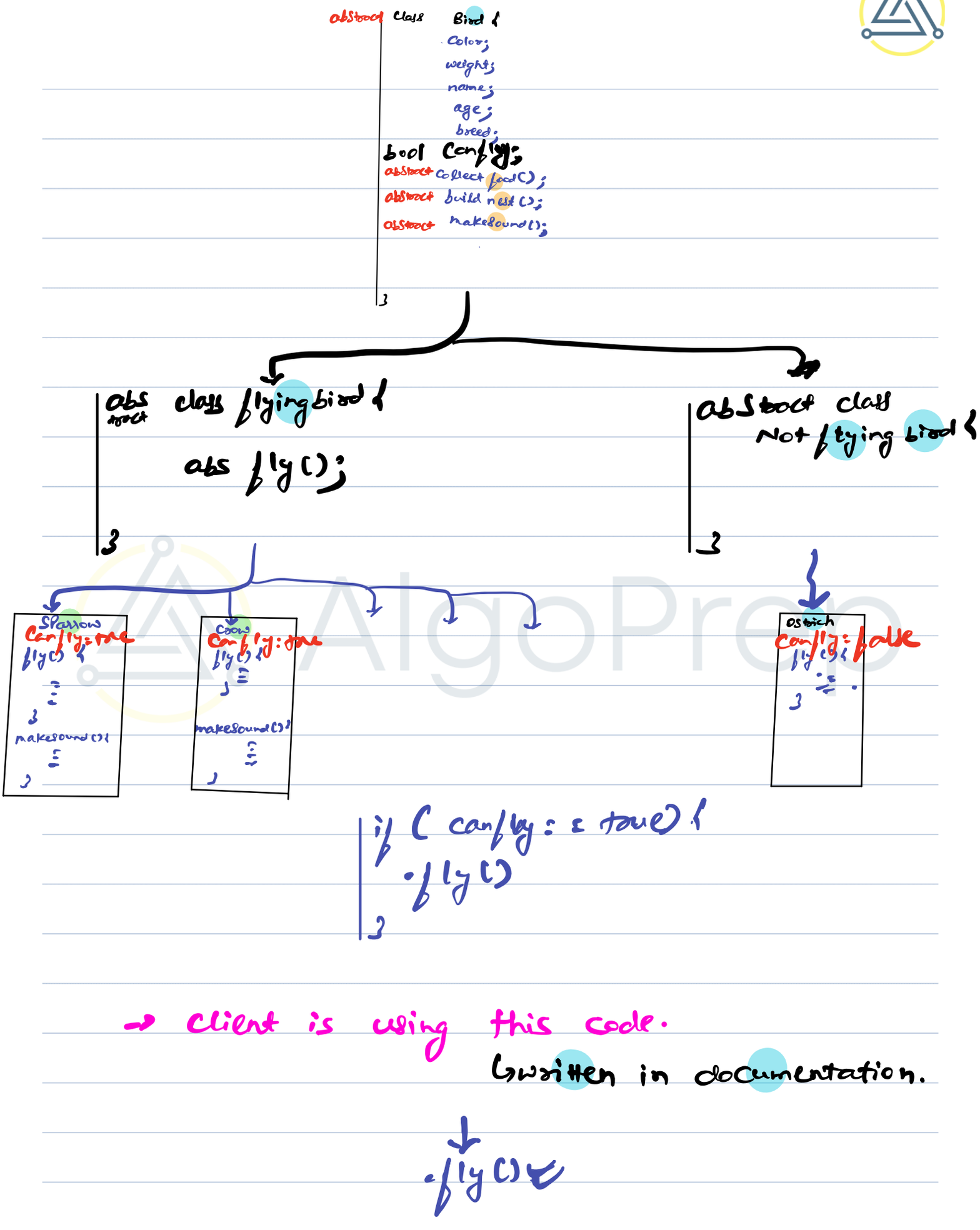
Today's agenda

- SOLID Principle

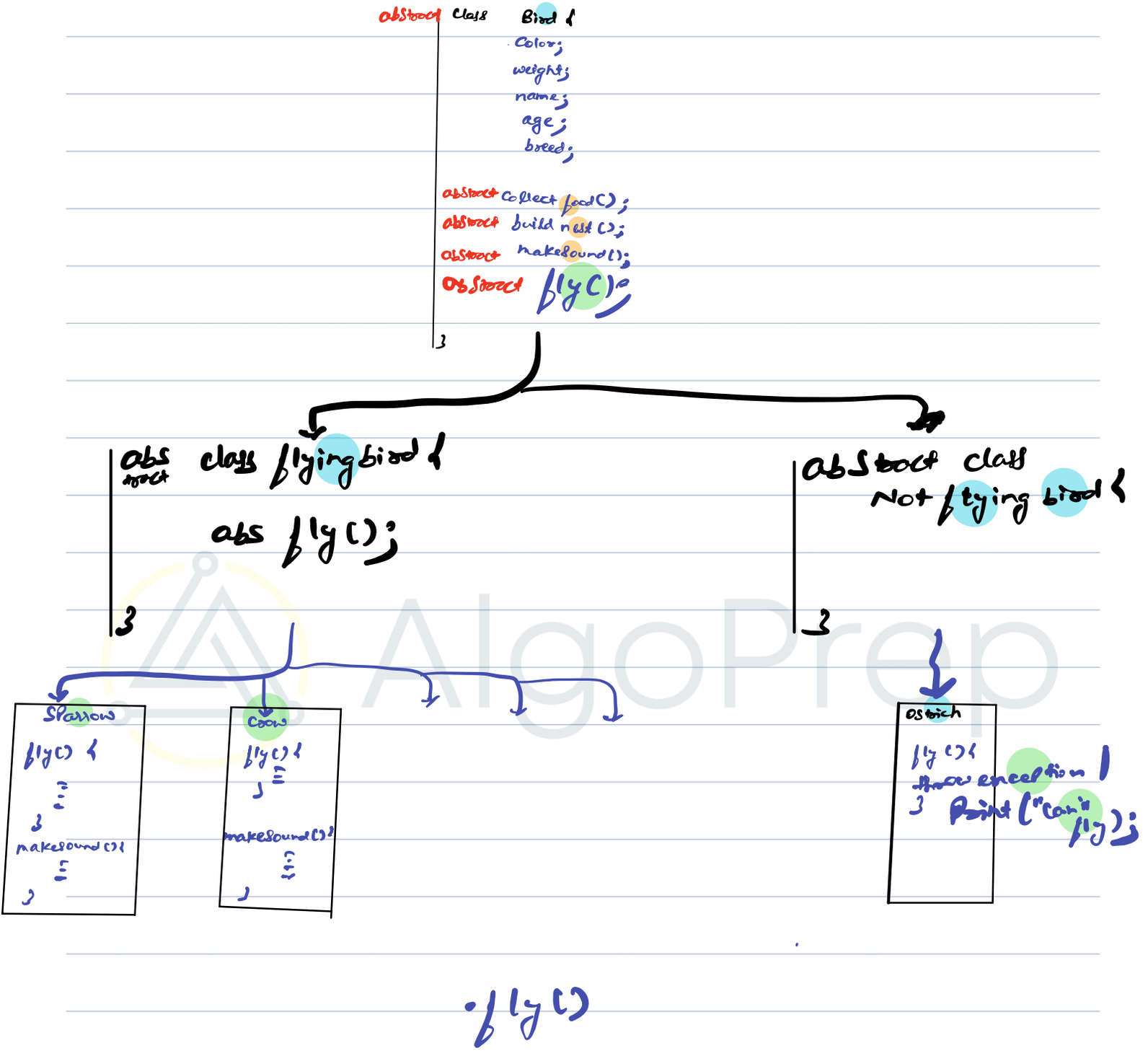


→ Client is using this code.

Written in documentation.

↓

fly() ✓


abstract class Bird {
color;
weight;
name;
age;
breed;
abstract collectFood();
abstract buildNest();
abstract makeSound();
abstract fly();
}
abstract class FlyingBird {
abs fly();
}
abstract class NotFlyingBird {
}
Sparrow
fly() {
}
makeSound() {
}
Crow
fly() {
}
makeSound() {
}
Ostrich
fly() {
throw exception
}
Bird("can't fly");

fly()

L → Liskov's Substitution Principle

↳ Child classes should do exactly what their Parents class expects them to do.

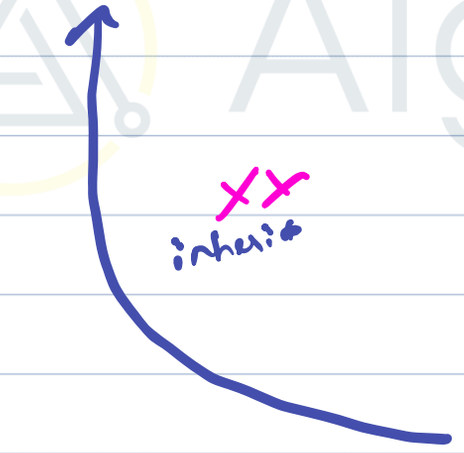
newspaper ← Paper Printing

abs Printer {

Print something on paper ← abs Print();

m1();
m2();

}



you want to do Commanline Printing

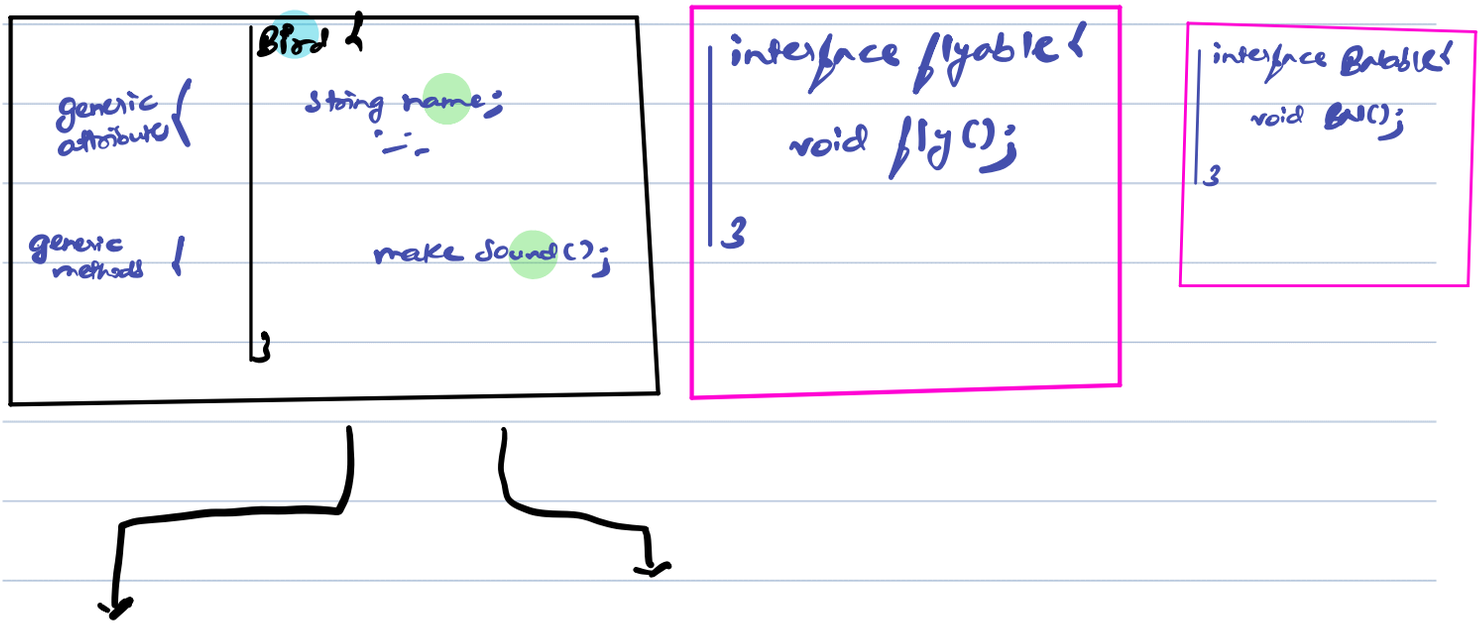
class CommandlinePrinting {

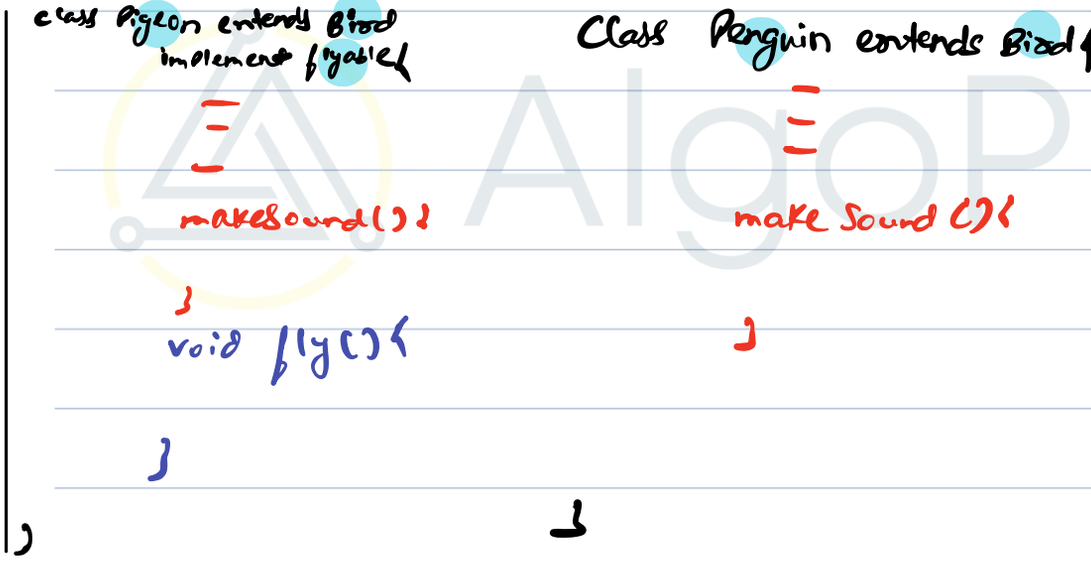
Print() {

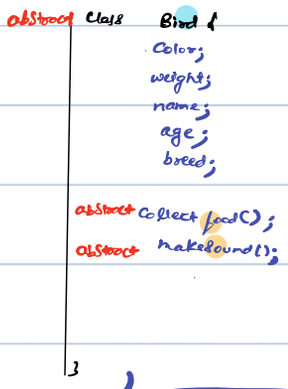
}

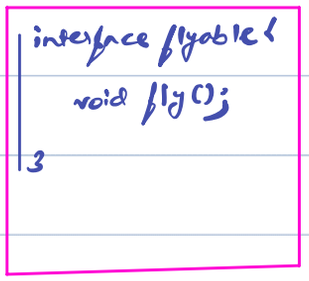
}

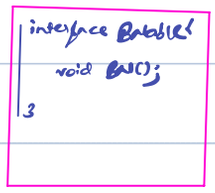
v4 → creating interfaces for optional method.











Class Crow extends Bird implements Flyable, Babble

Class Bird2 extends Bird implements Babble

Class Ostrich extends Bird implements Babble

-> 2 options

Can fly / Can't fly

Can Babble / Can't Babble

Can jump / Can't jump

Can eat / Can't eat

↓

2 interfaces

Requirement

All the birds that can fly can build nest and birds that can't fly can't build nest.

↳ Can we combine the 2 interfaces we took in last solution??

↳ yes

```
abstract class Bird {
    color;
    weight;
    name;
    age;
    breed;

    abstract collectFood();
    abstract makeSound();
}
```

```
interface FlyEnabled
    void fly();
    void enableFly();
}
```

1000s of bird

```
class Crow {
    extends Bird implements
        FlyEnable
        void fly() {
        }
}
```

↳ if you found a bird which can fly but can't build nest.

* Interface Segregation Principle.

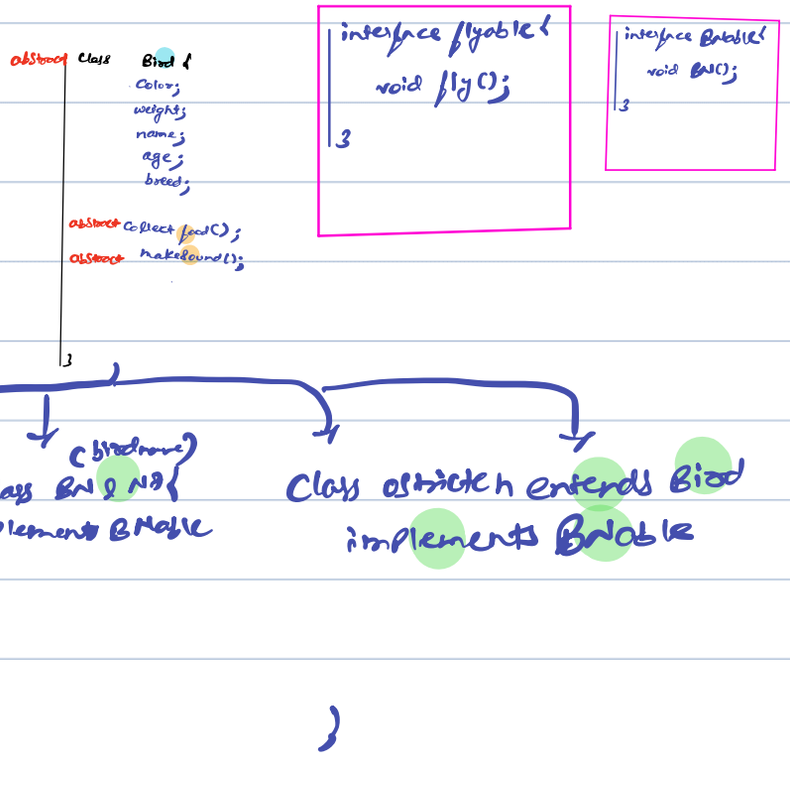
↳ interfaces should be as light as possible.

# of methods in interface should be minimum possible. (ideally 1)

if interface has more than 1 method, they should be related to each other.

→ Interface segregation principle is equivalent to SRP for interfaces.

Correct Solution even with ↳ considering above requirement.



example with interface having more than 1 method:

↓

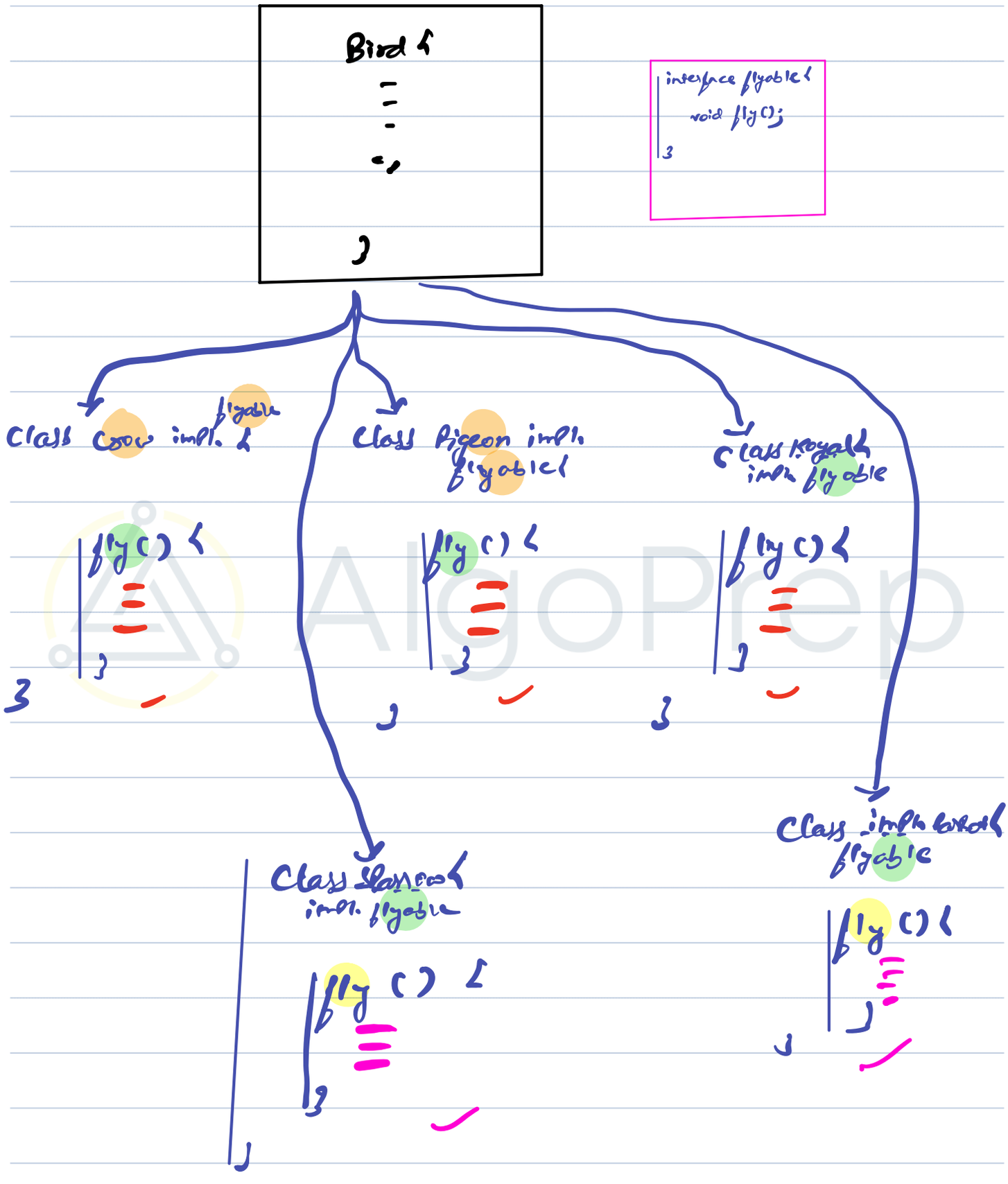
```
Interface Stack {

    Push();
    Pop();
    Size();
    top();

}
```

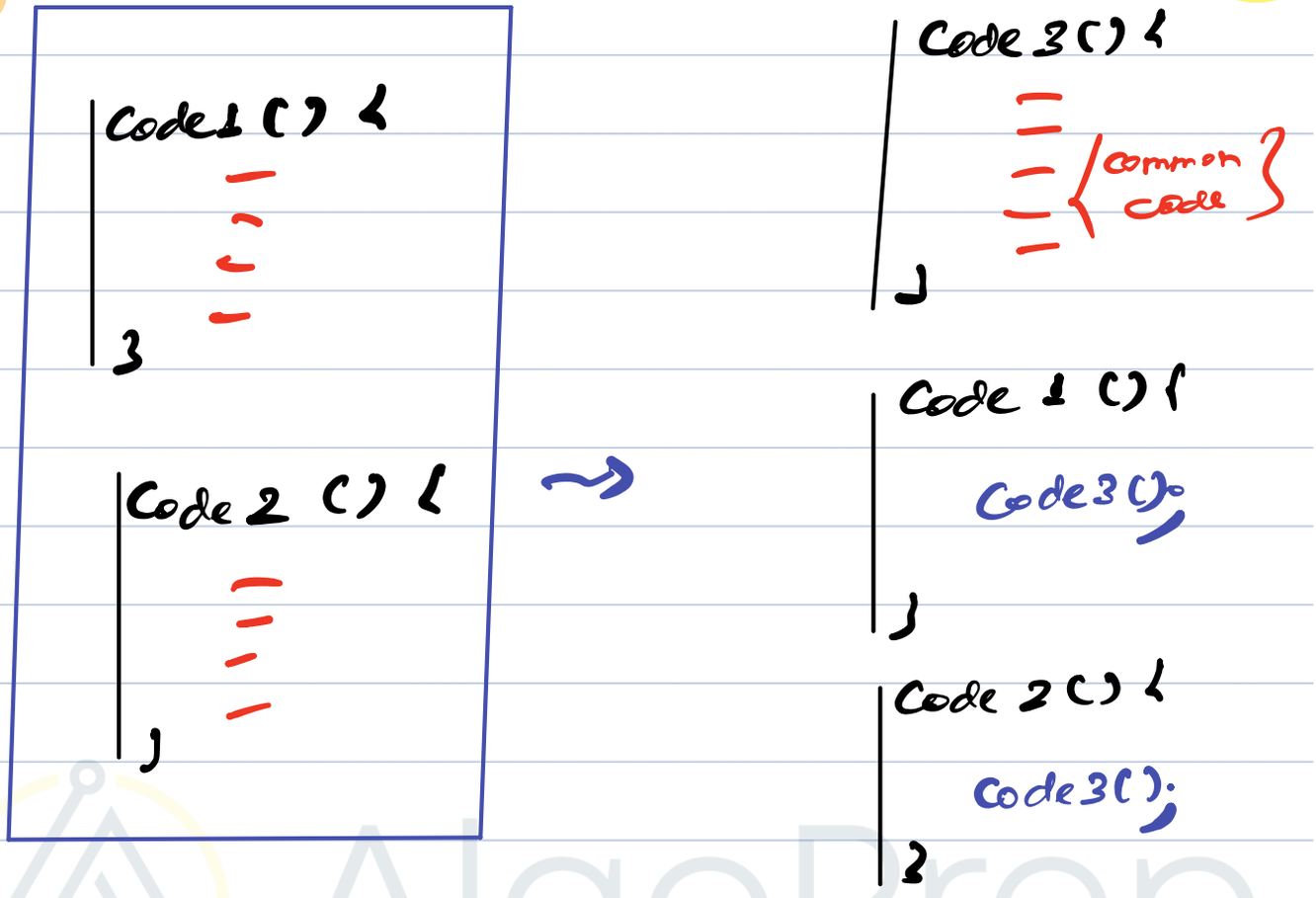
→ functional Interface: interfaces with only 1 method.

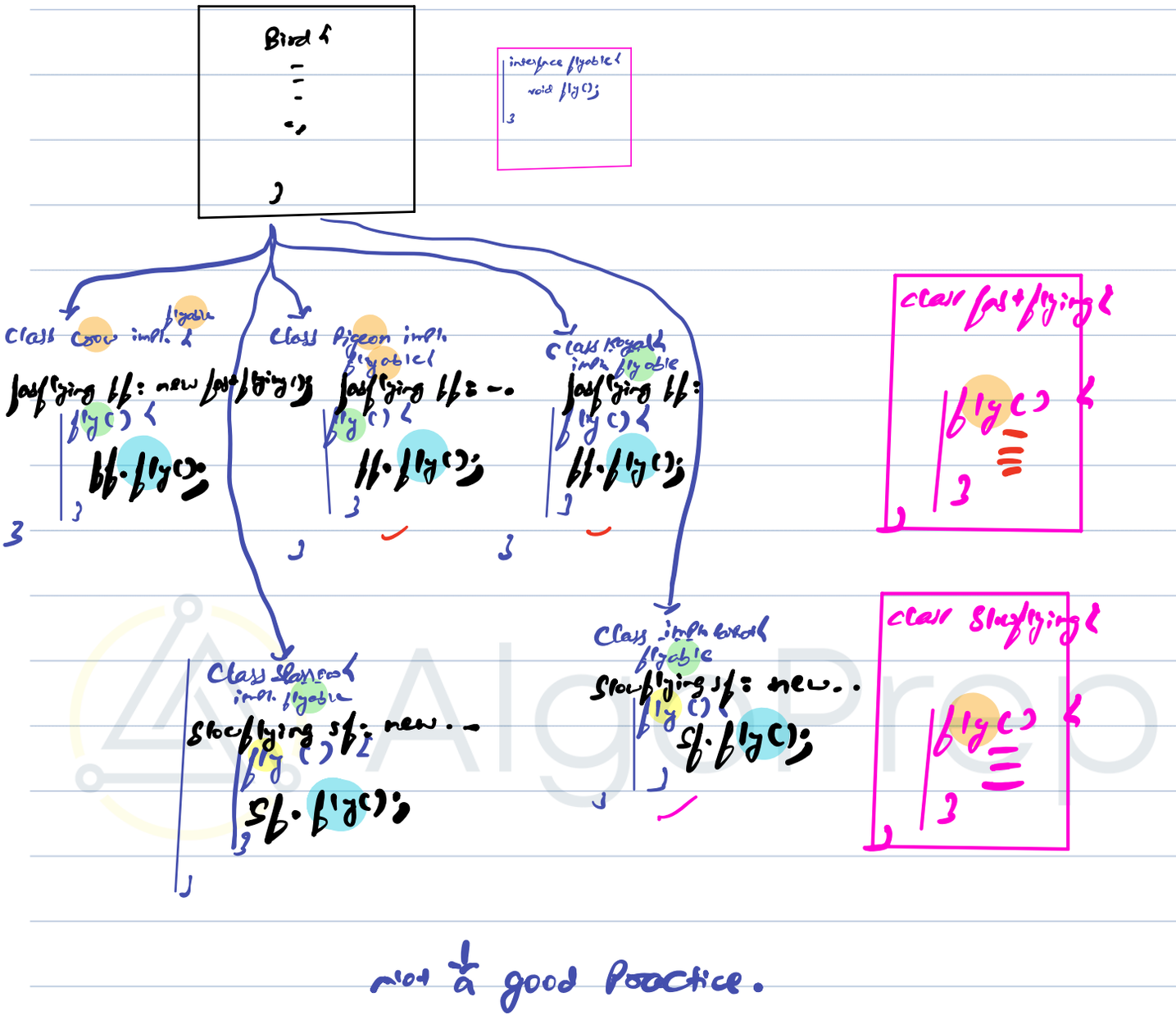
Break till 9:25 Pm



→ DRY {Code duplication}

within same class

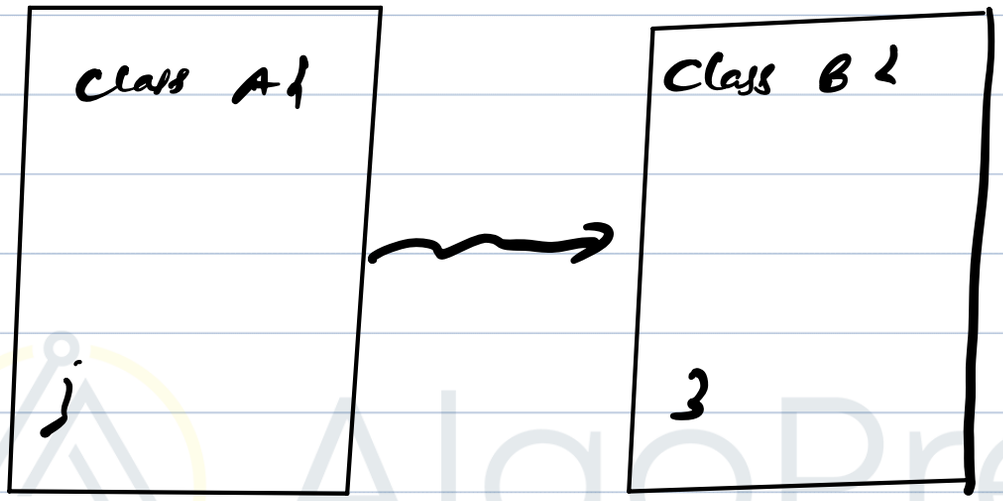




not a good Practice.

* D: Dependency inversion Principle.

↳ No 2 class should ideally depend on each other. They should depend via an interface.



↳ Not ideal.

yesBank();

→ Paytm → ~~Paytm~~ ~~Payments~~ ~~bank~~

Class PaytmPaymentsBank {
- transfermoney();
- fetchBalance();
.
.
.
}

PaytmPaymentsBank PPB = new PaytmPaymentsBank();

PPB.transfermoney();
PPB.fetchbalance();

yesBank yb = new yesBank();

yb.transfermoney();
yb.fetchbalance();
}

→ Dependency inversion principle

include

given by RBI

interface Bank {
transfermoney();
fetchbalance();
}

yesBank()

→ Bank b = new ~~Paytm~~~~PaymentsBank~~();
b.transfermoney();
b.fetchbalance();

→ Bank b = new ~~Paytm~~~~PaymentsBank~~();
new yesBank();

following dependency inversion.