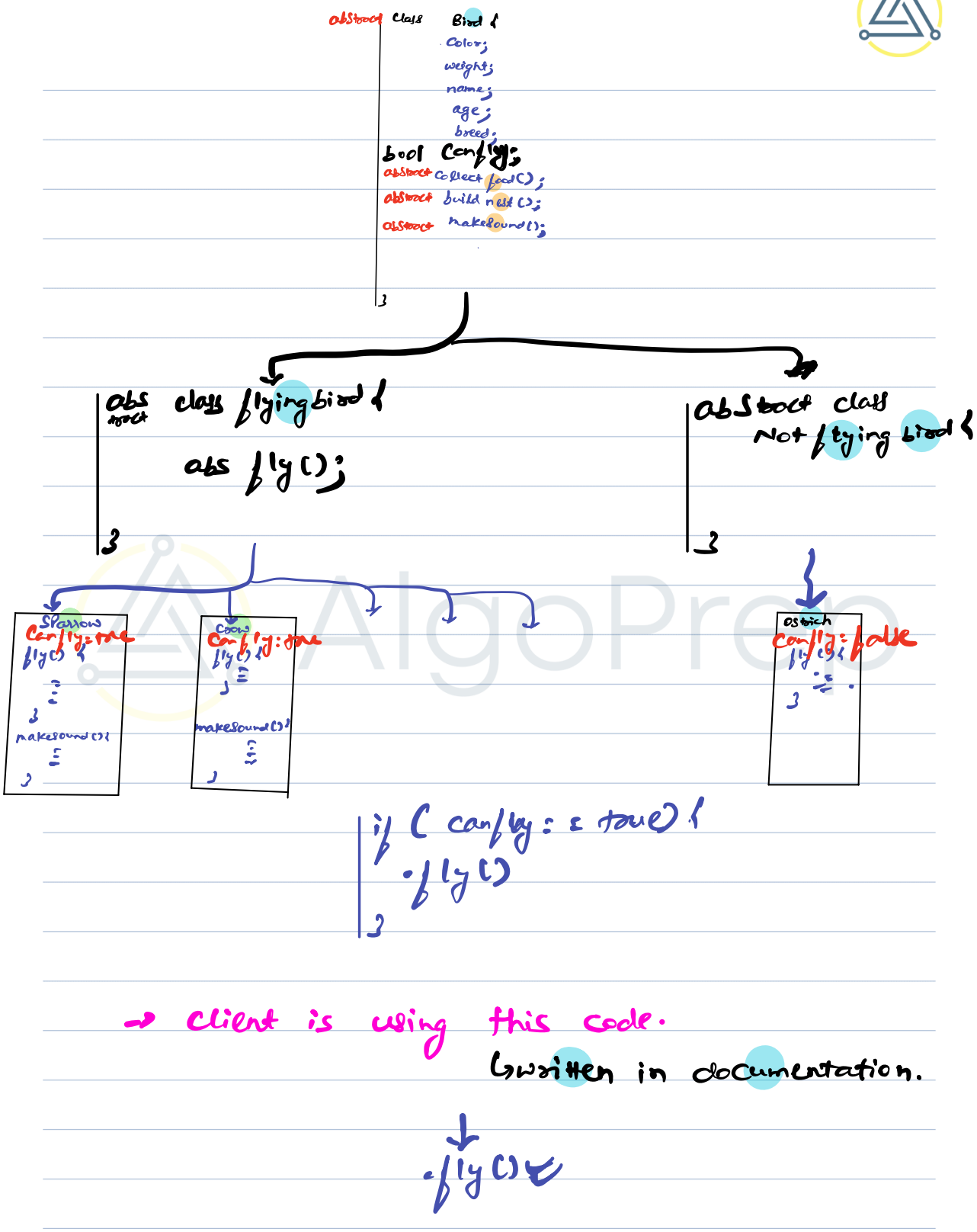
Today's Agenda

- SOLID Principle

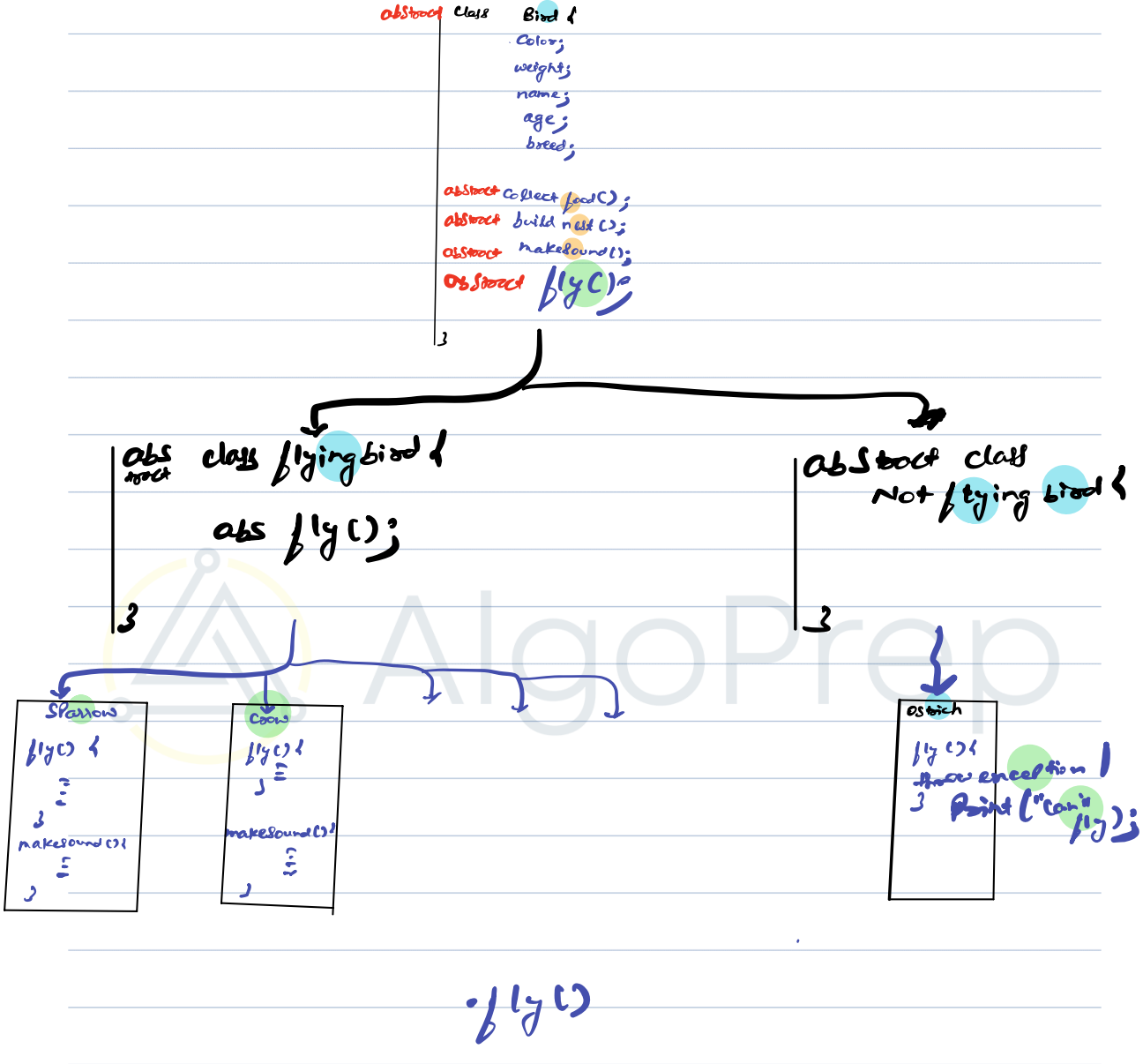


→ Client is using this code.

(written in documentation.

↓

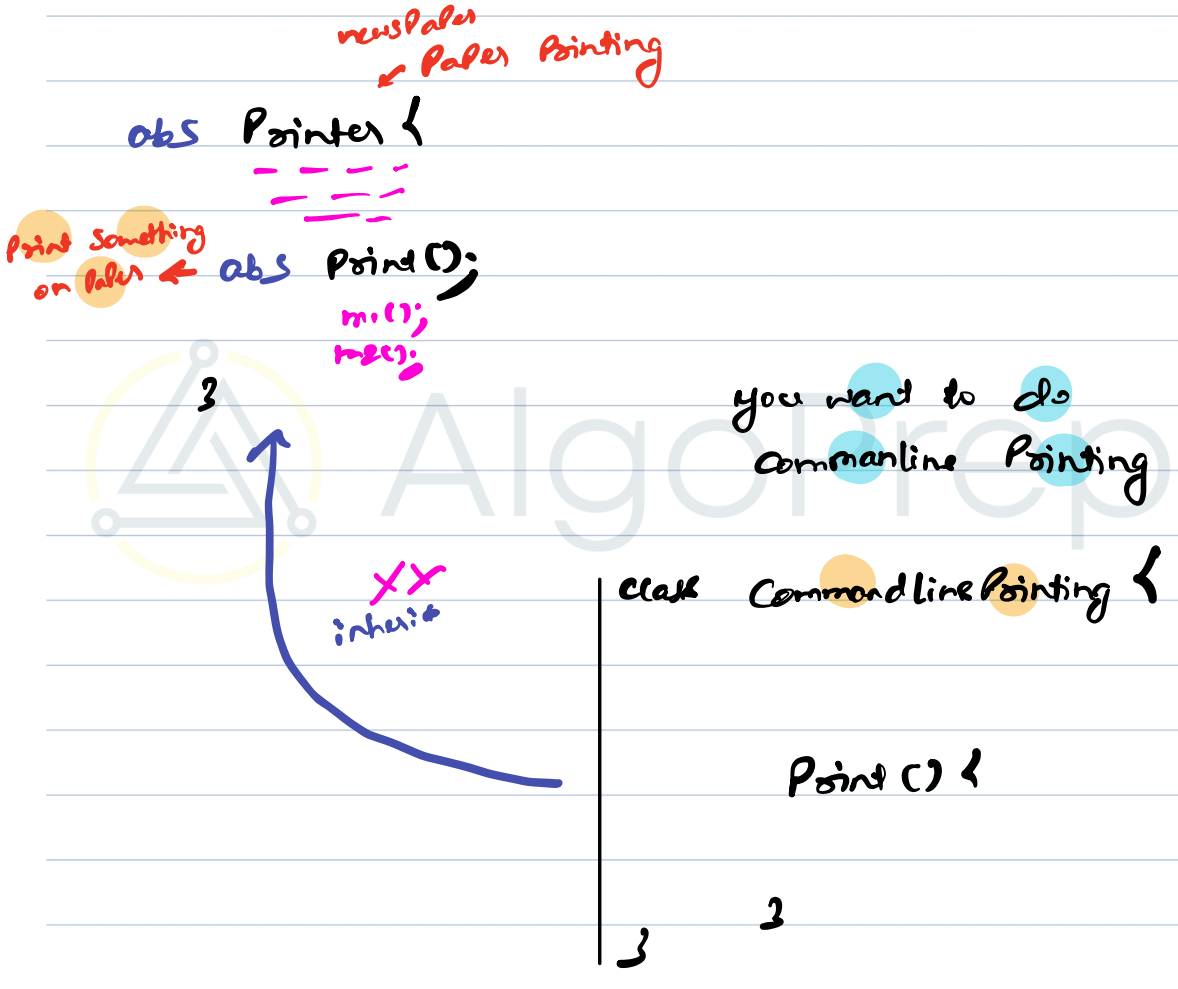
fly()


abstract class Bird {
Color;
weight;
name;
age;
breed;
abstract collect food();
abstract build nest();
abstract makeSound();
Abstract fly();
}
abs class flyingbird {
abs fly();
}
abstract class Not flying bird {
}
Sparrow
fly() {
}
makeSound() {
}
Cow
fly() {
}
makeSound() {
}
ostrich
fly() {
throw exception
}
Print("can't fly");

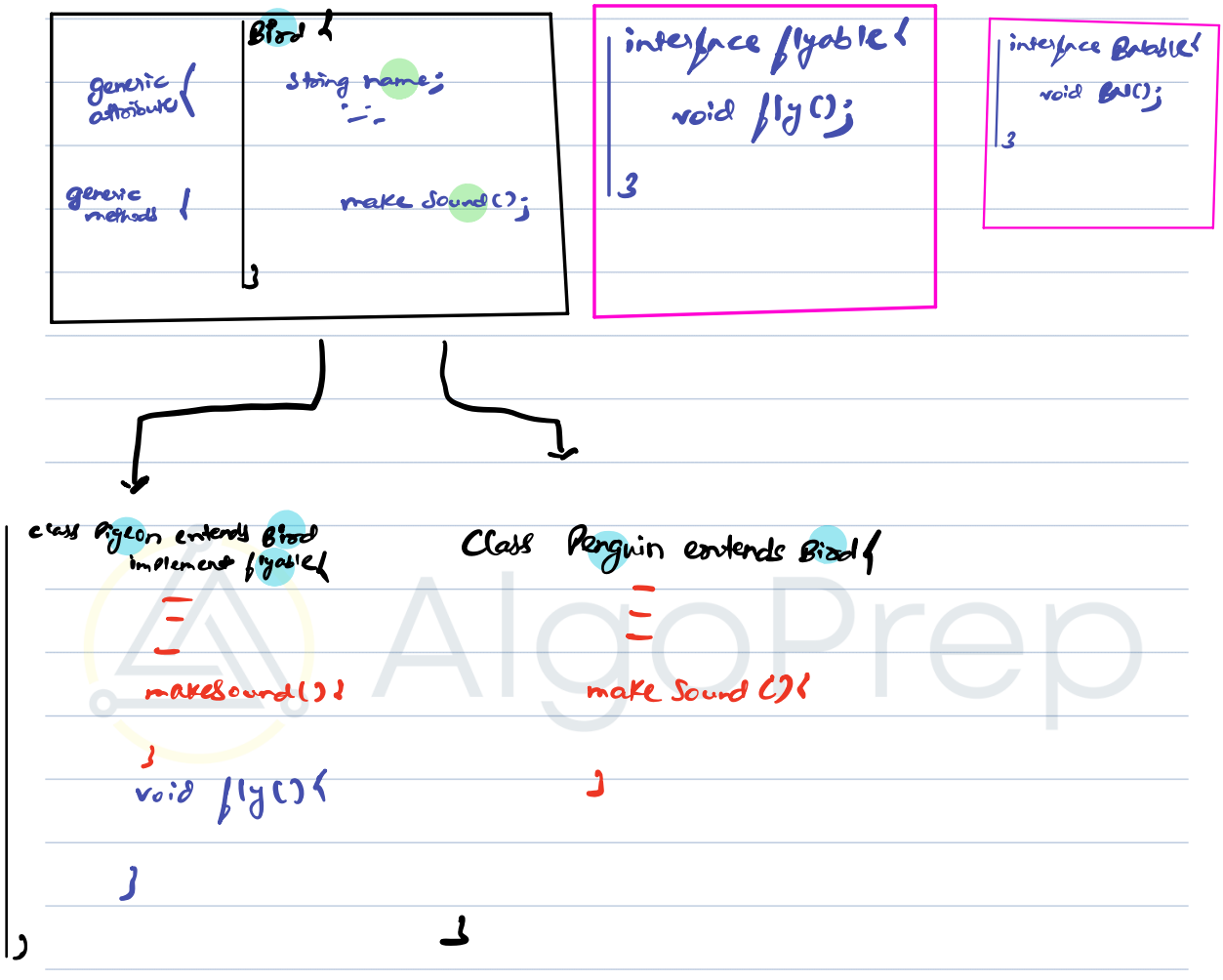
fly()

L → Liskov's Substitution Principle

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



v4 → creating interfaces for optional method.



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

    abstract collect food();
    abstract makeSound();
}
```

```
interface flyable {
    void fly();
}
```

```
interface Bwable {
    void Bw();
}
```

```
class crow
extends Bird implements
flyable, Bwable {

}
```

```
class Bw2 N2 {      (birdname)
implement BMable

}
```

```
class ostricteh extends Bird {
implements Bwable

}
```

→ 20 options

C7	C6w	Can jump	Can eat	— · —
Can't a	Can't BW	Can't jump	Can't eat	

↓

20 interfaces

Requirement

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

↳ 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 flyEnable
    void fly();
    void BNable();
}
```

1000s of bird

```
class crow {
    extends Bird implements
    flyEnable
        void fly() {
    }
}
```

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

Interface Segregation Principle.

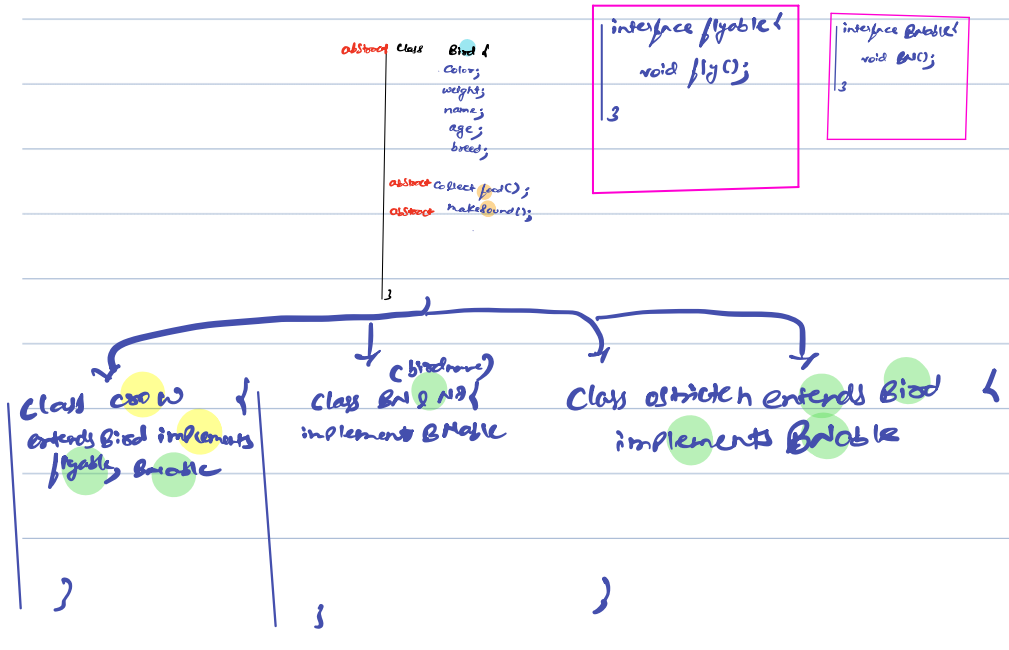
↳ interfaces should be as light as possible.

# 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.



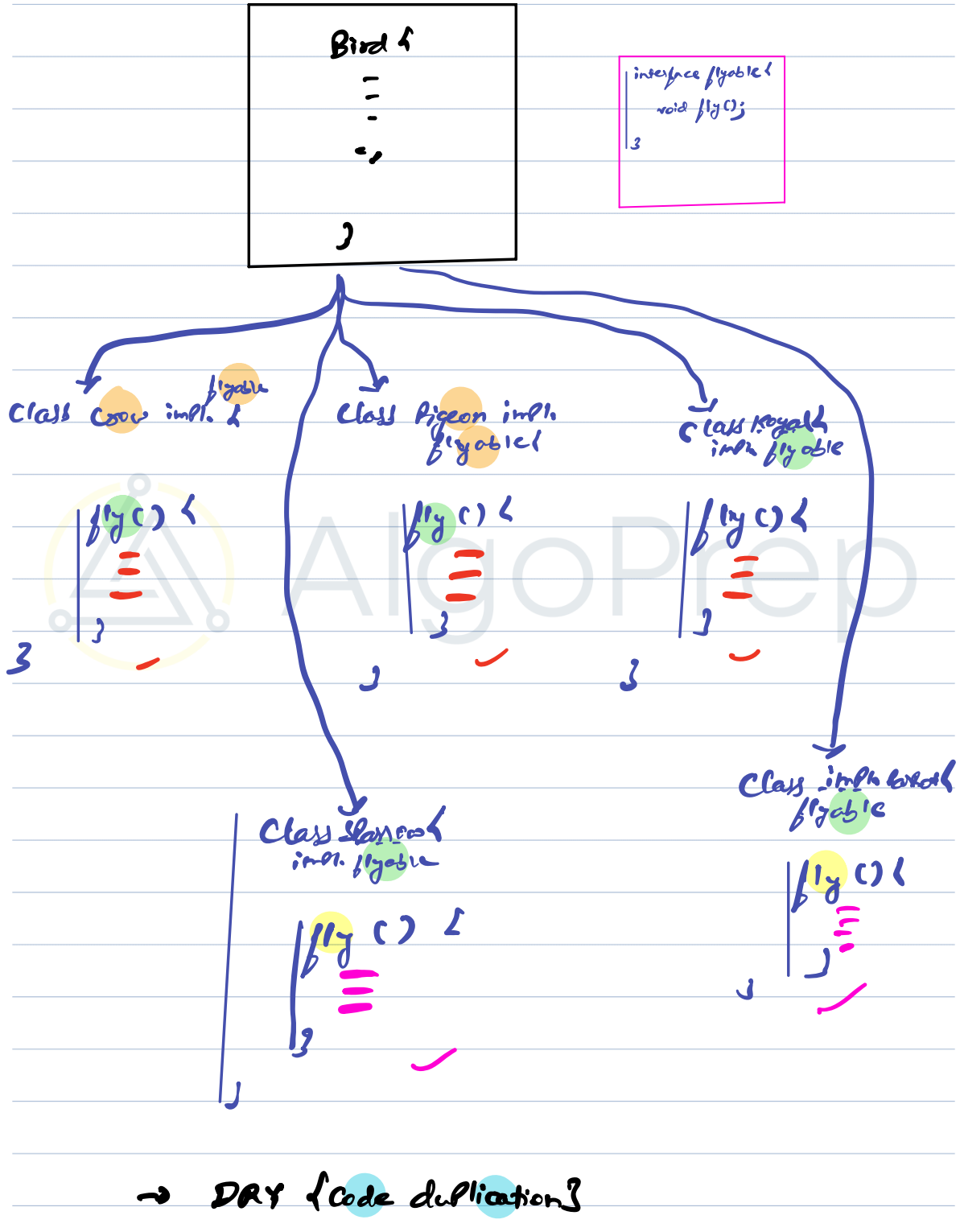
example with interface having more than 1 method:

↓

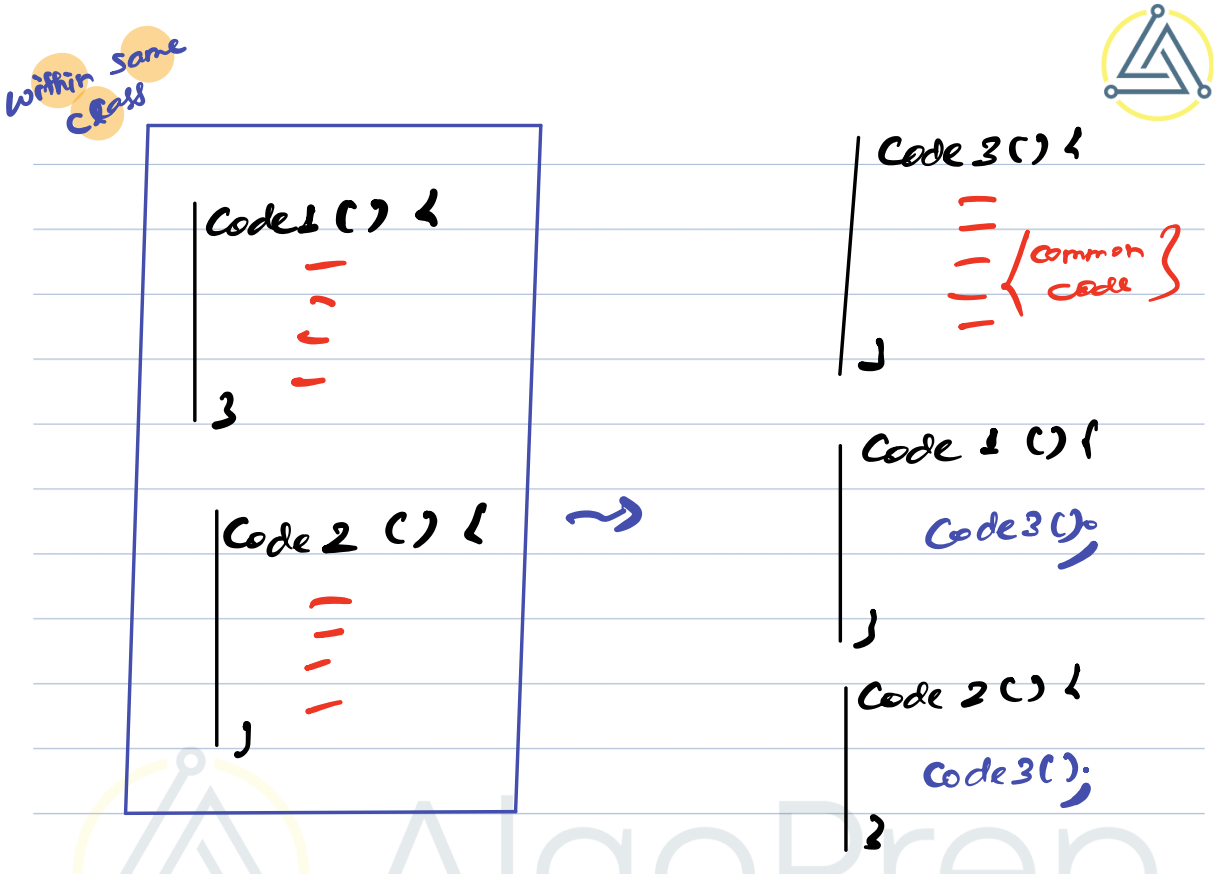
```
Interface Stack {

    Push();
    Pop();
    Size();
    top();
}
```

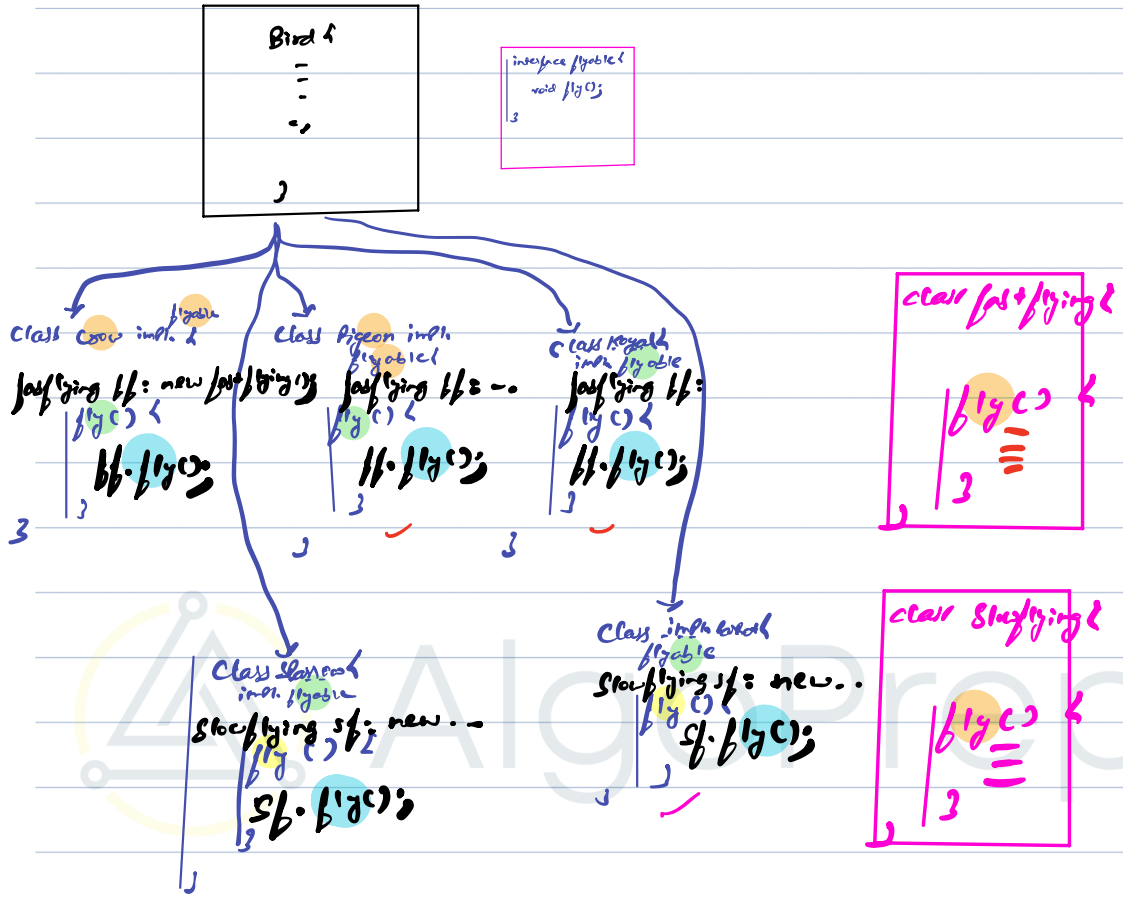

→ functional Interface: interfaces with only 1 method.

Break till 9:25 Pm



→ DRY {Code duplication}

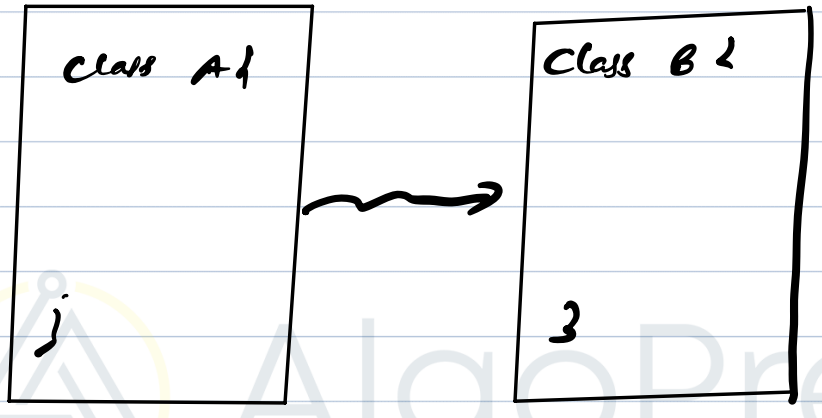

within same class
Code1 () {
}
Code2 () {
}
Code3 () {
common code
}
Code 1 () {
Code3();
}
Code 2 () {
Code3();
}



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.