LLD-2 SOLID Design Principles

Agenda

1. Liskov's Substitution
2. Interface Segregation
3. Dependency Inversion

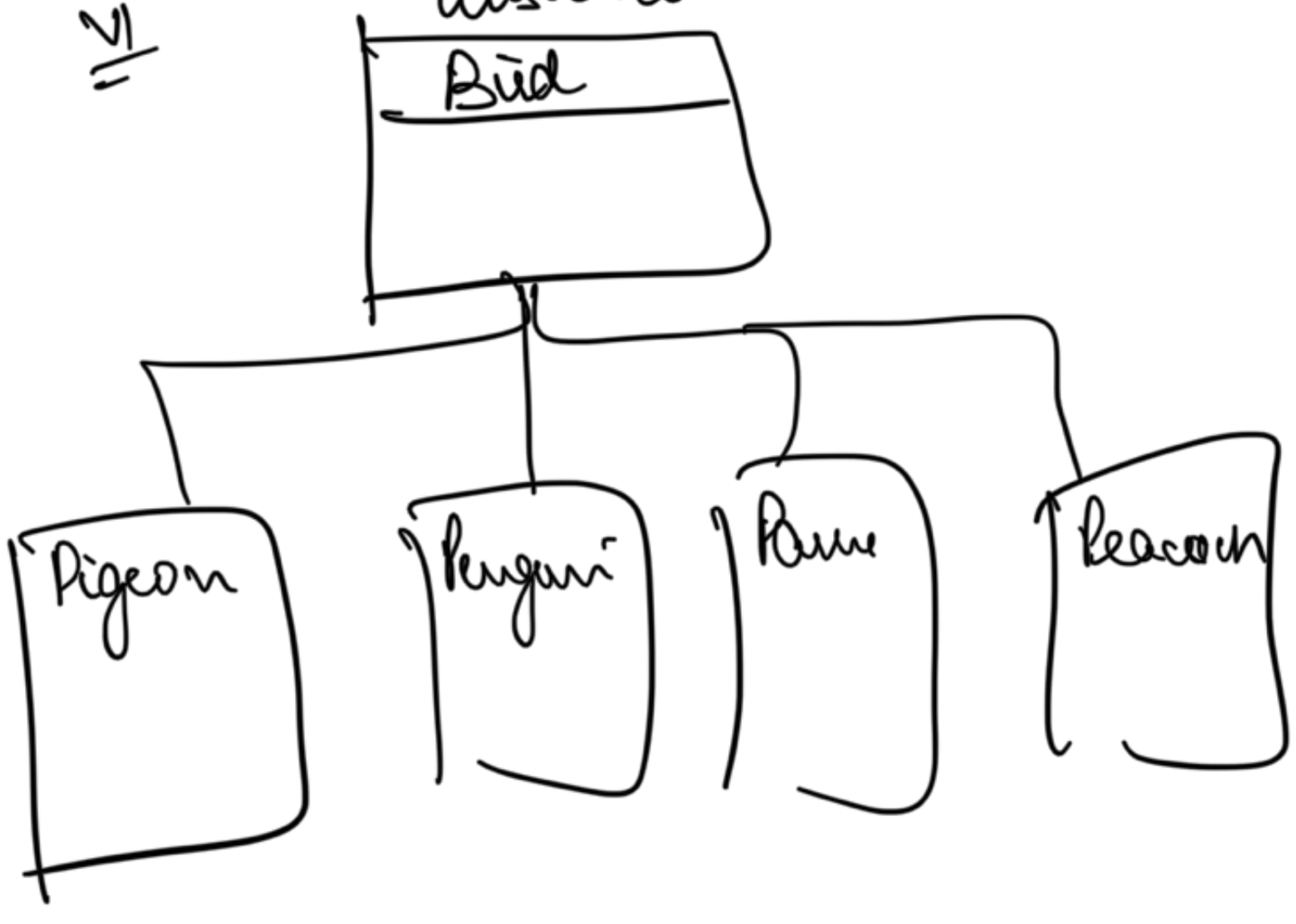
Design a Bid

VO

CAPX

O/C X

VI



SAP → ✓

O/C → ✓

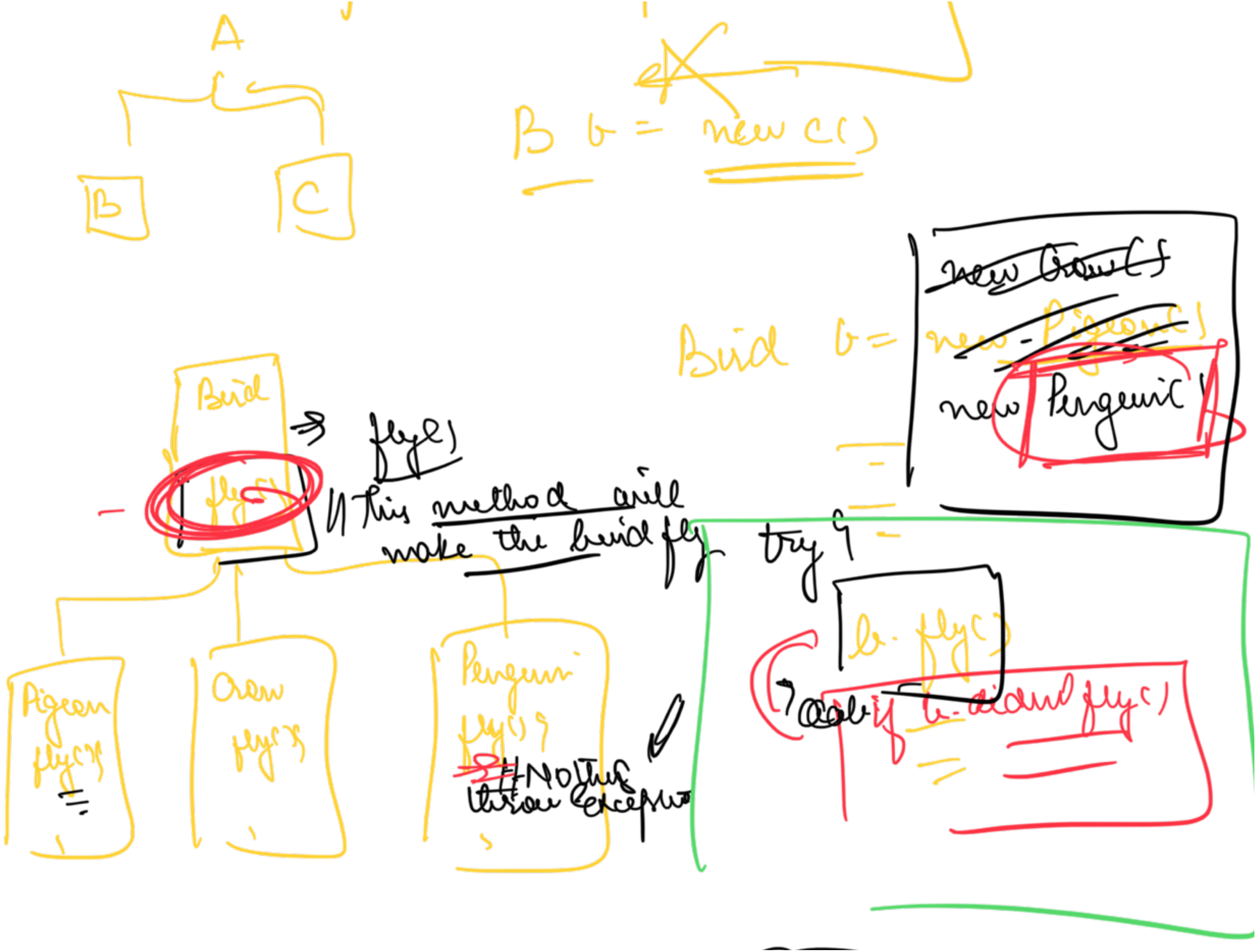
Every Birds can't fly ()

Liskov's Substitution Principle

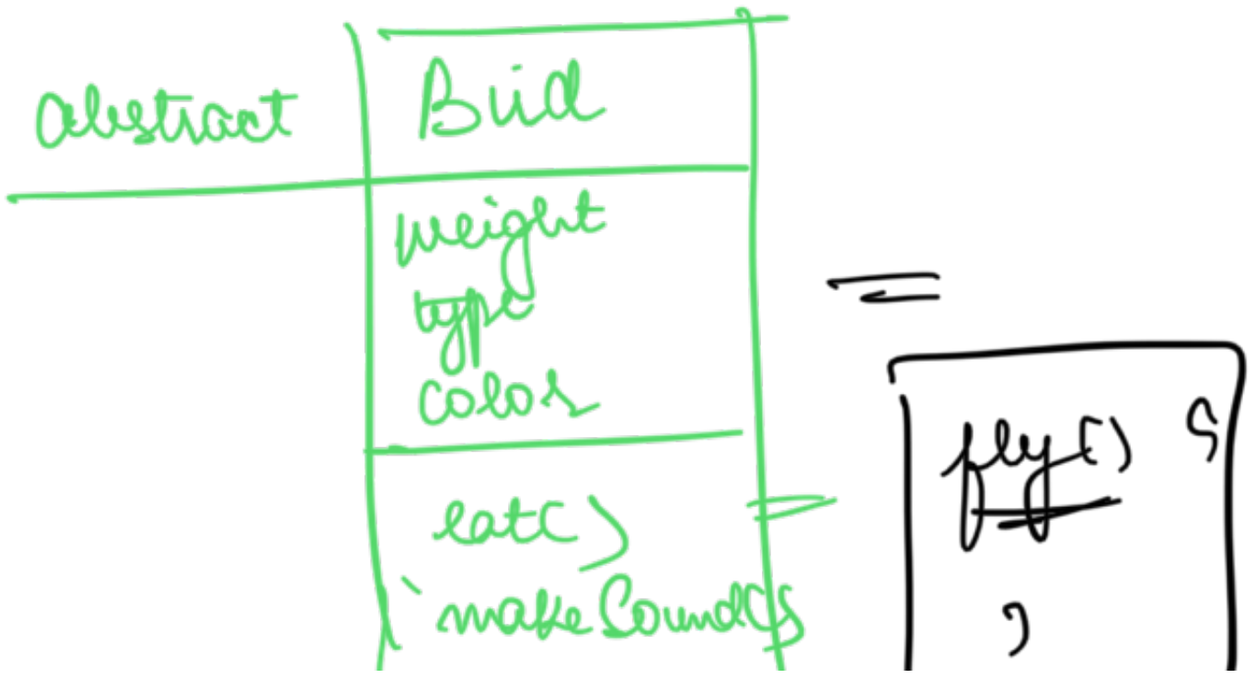
⇒ No child class should observe ANY SPECIAL TREATMENT

⇒ All objects of any child class should be substituted in a variable

of parent class type (w/o) requiring any code change in further code -


A
B
C
B b = new C()
Bird b =
new Crow()
new Pigeon()
new Penguin()
Bird
fly()
// This method will make the bird fly
Pigeon
fly(){}
Crow
fly(){}
Penguin
fly(){
// Nothing
throw Exception
try {
b.fly()
if b.canFly()

① Let's remove fly() method from Bird class and instead only have it in child classes that can fly:



Crow

fly() {

}

Pigeon

fly() {

}

Penguin

Owl

fly() {

}

Use Case : Get a list of all objects that can fly

List<Bird> l = List.of(

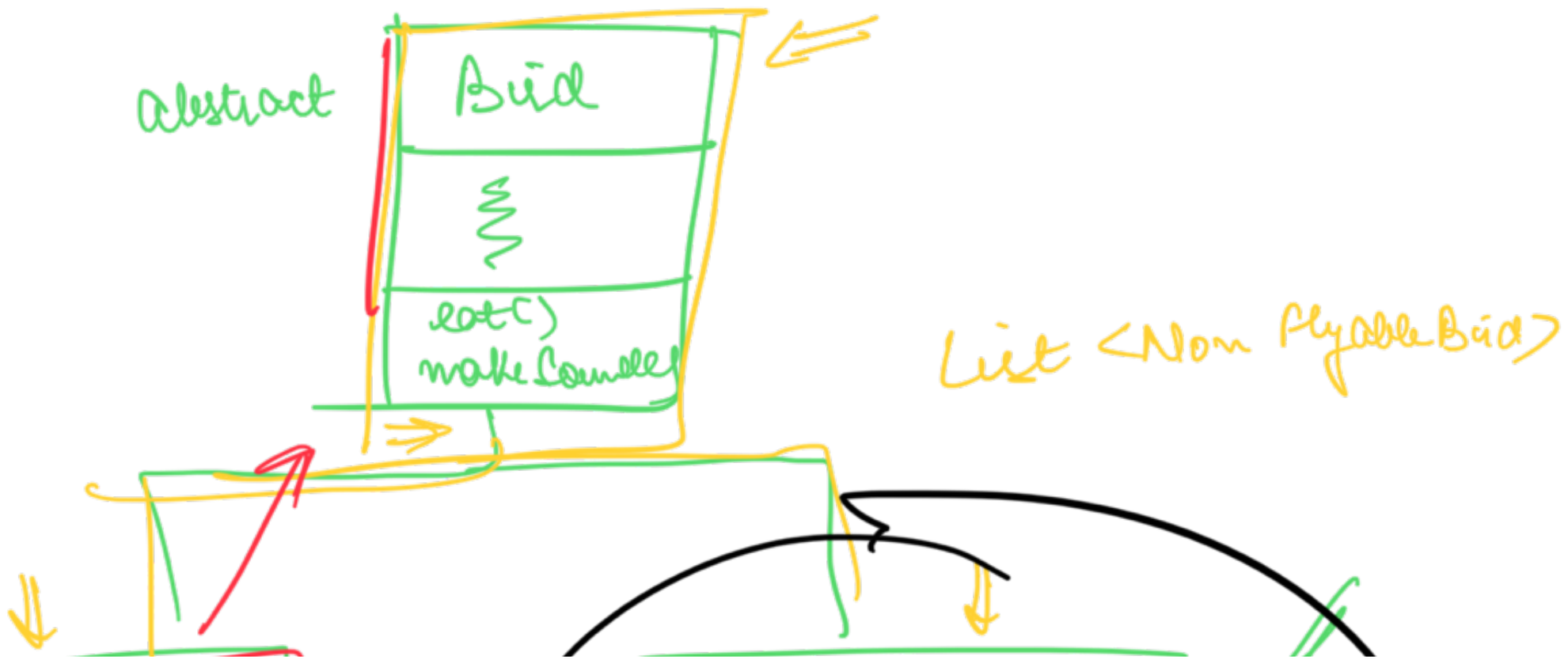
new Owl(),

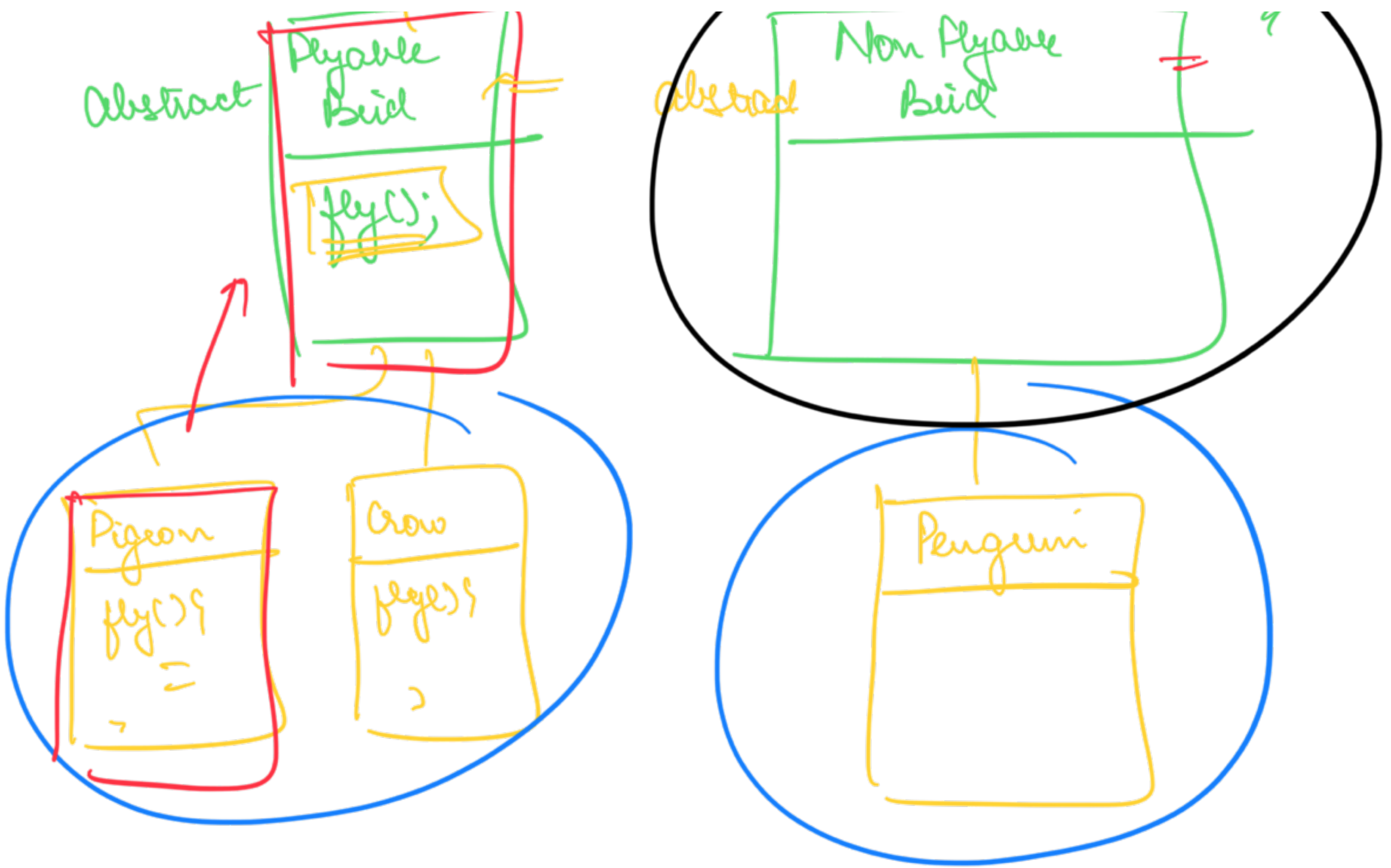
new Pigeons,

)

---

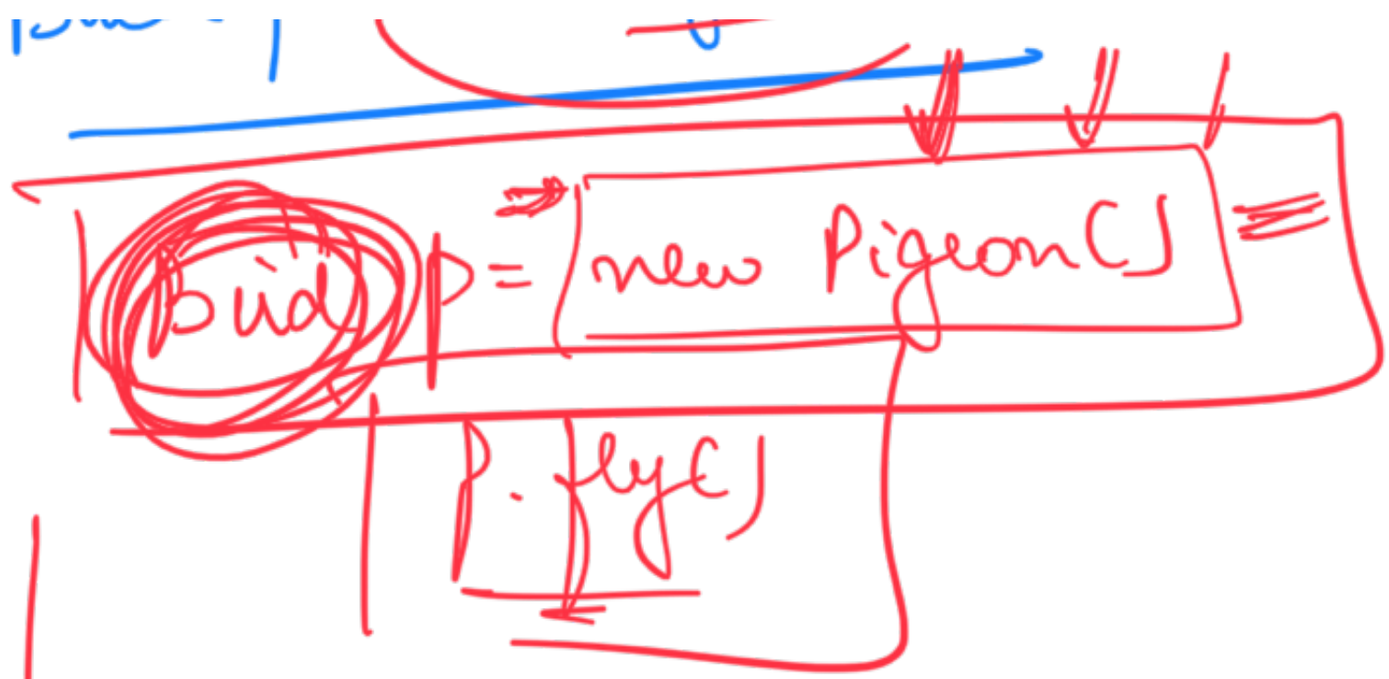
(2)





Pigeon p = new Pigeon();

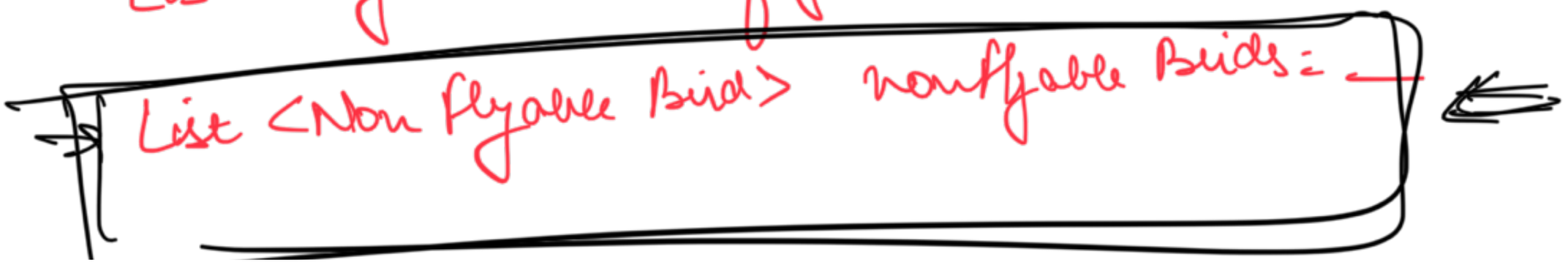
Bird p = new Pigeon();



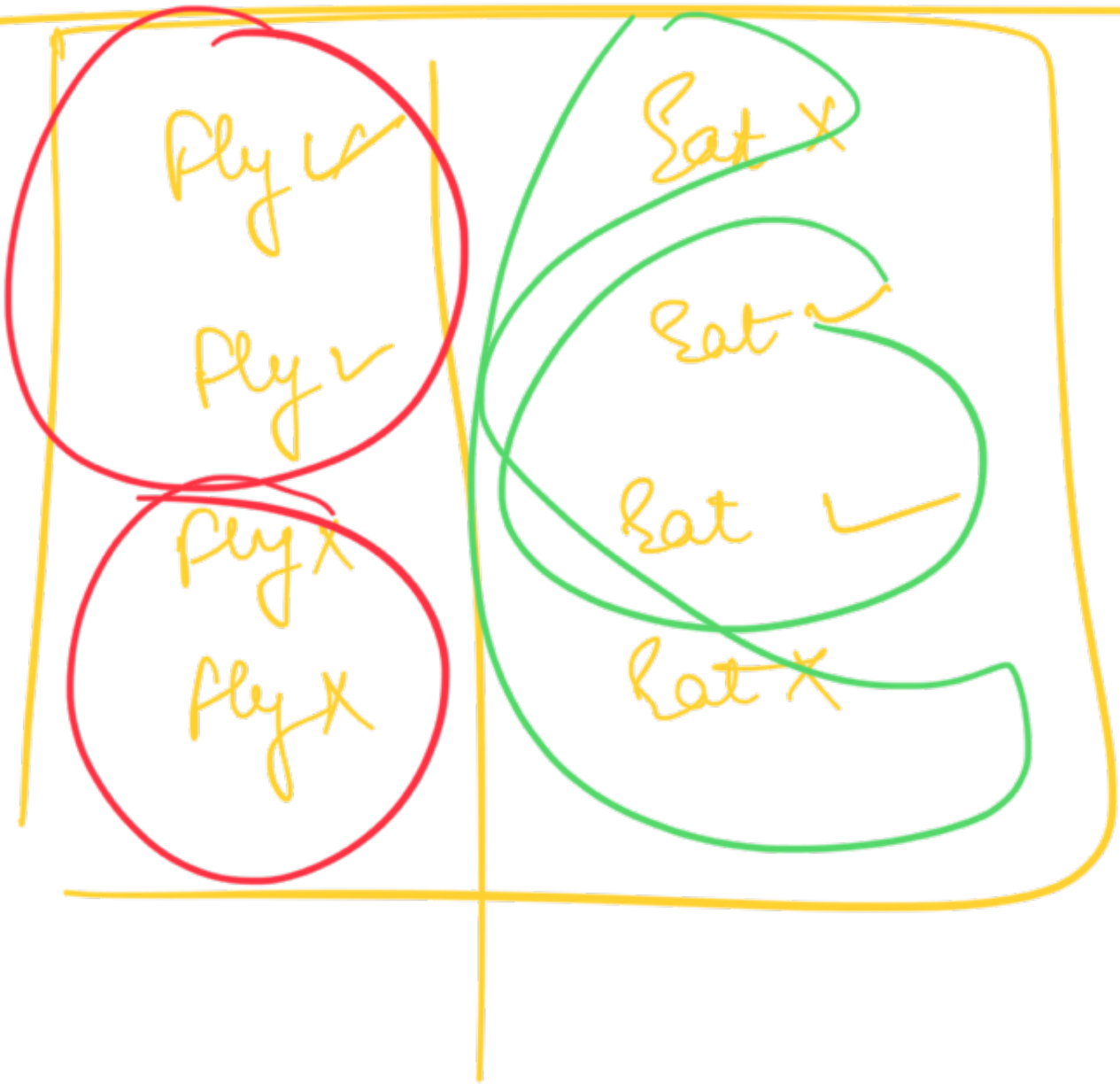
C IS-A A

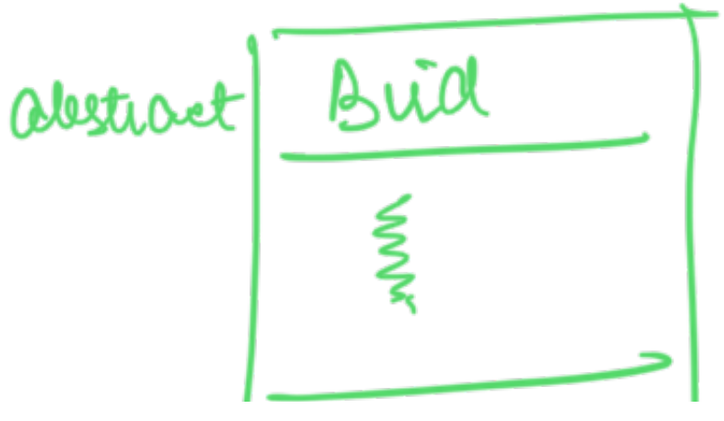
List <Flyable Bird> flyableBirds = ——————

List <Non Flyable Bird> nonFlyable Birds = ——



Some Birds





2^10 = 1024

10

eat, fly

4 such all

2^4=16

make Sound()

Flyable Eating Birds

fly()

eat()

Flyable Non Eating Birds

fly()

Non flyable Eating Birds

eat()

Non Flyable Non Eating Birds

Pigeon

Crow

Sparrow

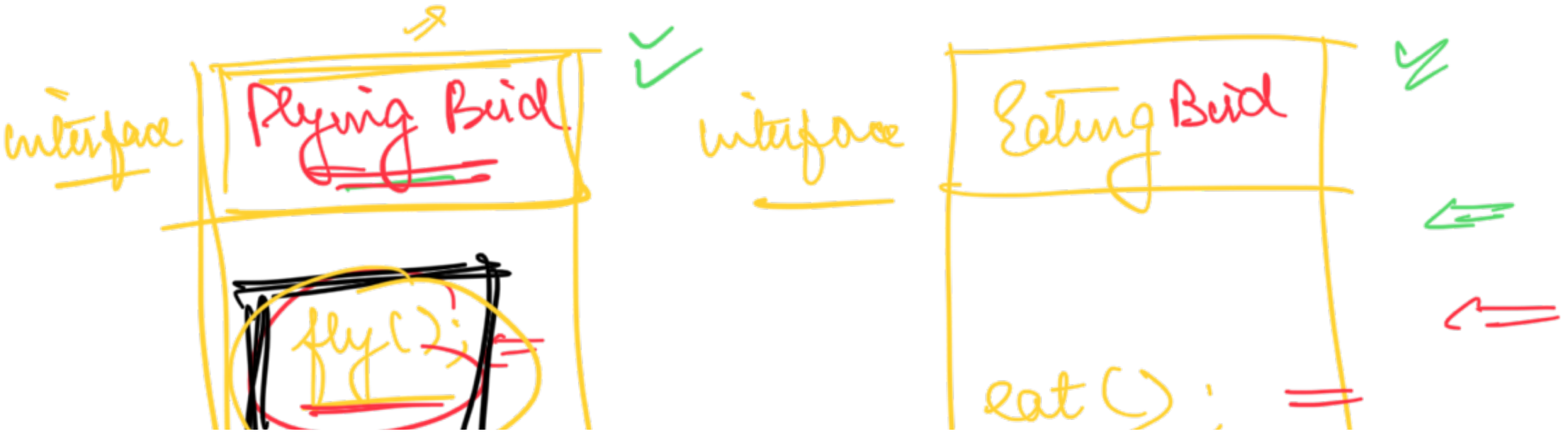
Penguin

Toy bird

Problems:

1. Class Explosion
2. Not possible to get a list of all the birds that can fly

Interfaces : Blueprint of behaviours



abstract

Bird

makeSound()

Human

Pigeon implements Human

Pigeon

fly()

eat()

Penguin

eat()

Sparrow

fly()

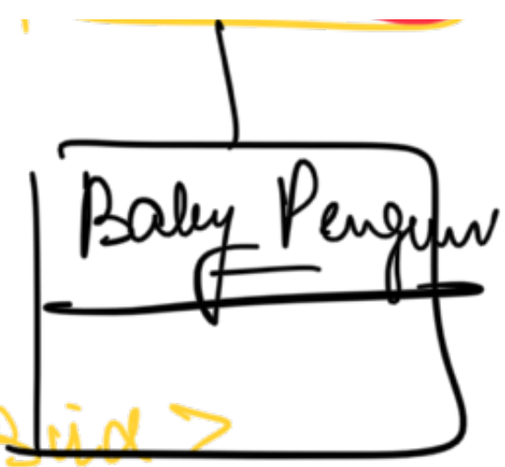
eat()

Owl

fly()

eat()

Toy



List < Bird >

List < Flying Bird > ⇒ List . of (

new Pigeon(),

new Sparrow()

new Baby Penguin

)

List < Eating Bird > →

```
for (SoaringBird e: birds)
    e.eat()
```

SRP for interface

I: Interface Segregation Principle

→ Interfaces should be as light as possible

→ Don't have Chick/ generalized interface.

→ Make interfaces as specific as possible

(Have very less #of

ideally 1 method/ interface

↳ functional interfaces

↳ Lambda fn

Runnable

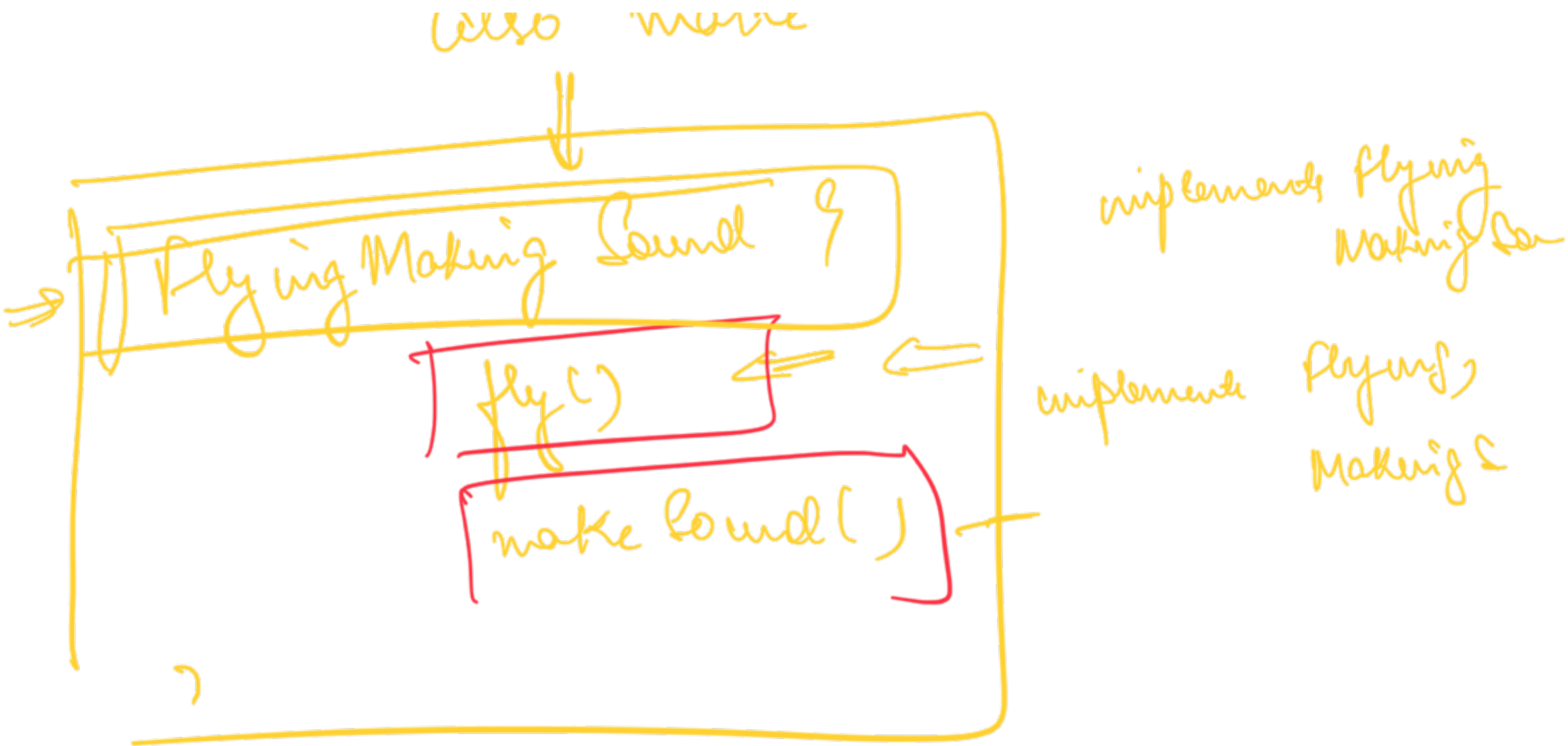
- run()

Callable

→ call

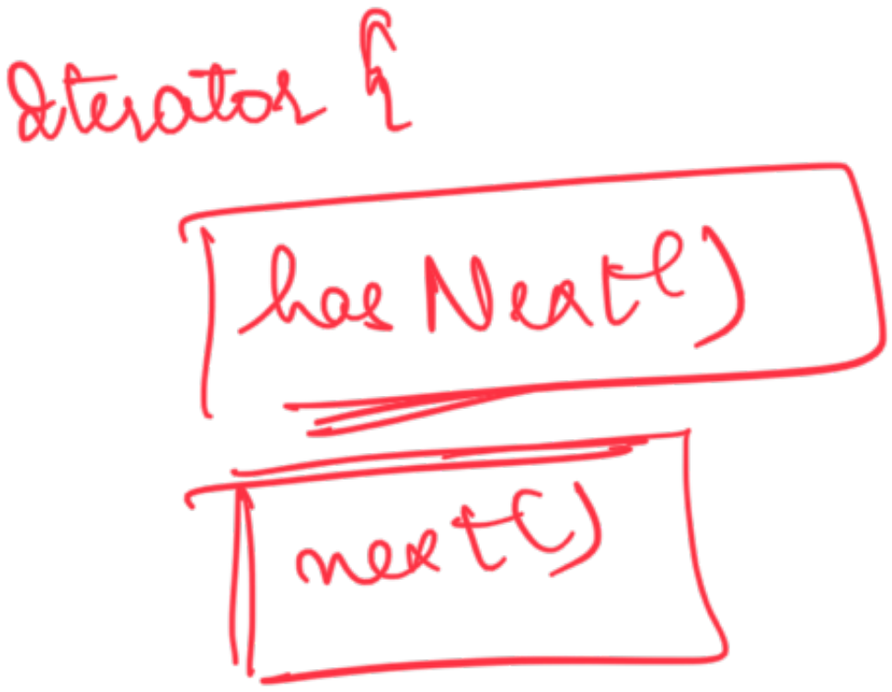
(interfaces with a single fn)

Use case: Every bird who can fly can also walk and



⇒ An interface should have more than one method only and only if both of are similar and

guaranteed to be present together



Eating
fly()

Playing
eat()

Making
Sound

Flying Eating Bird

Pigeon

By default
fly()
eat()

FlyingEatingBird b = new Pigeon();

FlyingBird b = new Pigeon();

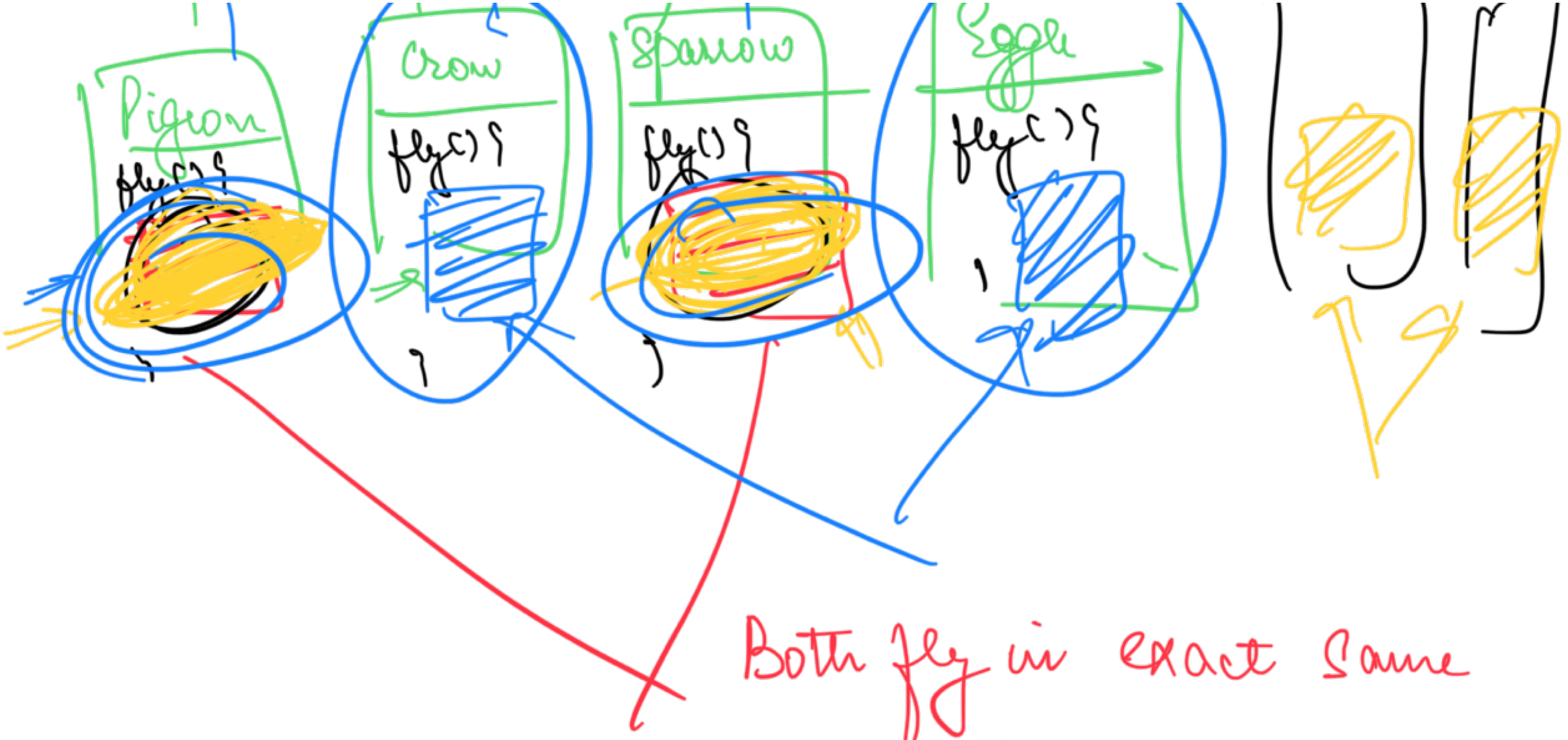
interface FlyingEatingBird extends FlyingBird, EatingBird

class Pigeon implements

Bird Example

There are some birds that fly in a similar way.

⇒ Code Duplication amongst those classes



Both fly in exact same way

⇒ Code Duplication

Design Patterns : Strategy Design Pattern

Pigeon Sparrow
Flying Behavio

makeFly() {

}

Crow Eagle
Flying Behavio

goFly() {

}

Common to
crow b
Eagle

Flying

fly()

Bird

Pigeon

fly() {

PSFB = new PSFB()

PSFB.makeFlye)

Sparrow

fly() {

PSFB = new PSFB()

PSFB.makeFly

Crow

fly() {

CEFB = new CEFB()

cefb.goFly()

CEFB = new CEFB()

cefb.goFlye)

Code Duplication ⇒

Pigeon extends Bird implements Flying {

Pigeon Sparrow Flying Behaviour :-

CEFB = new CEFB

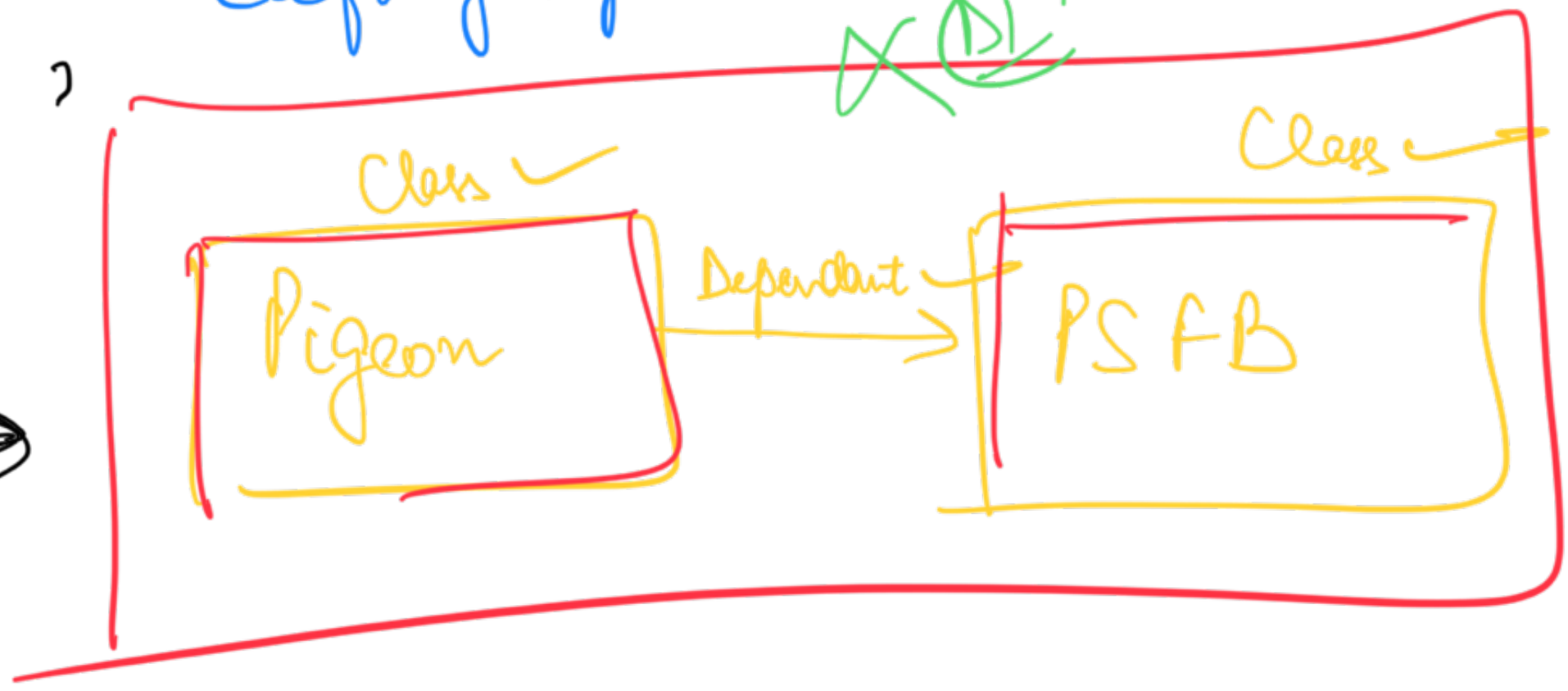
fly() {

CEFB.goFly

PSFB.makeFly()

}

}

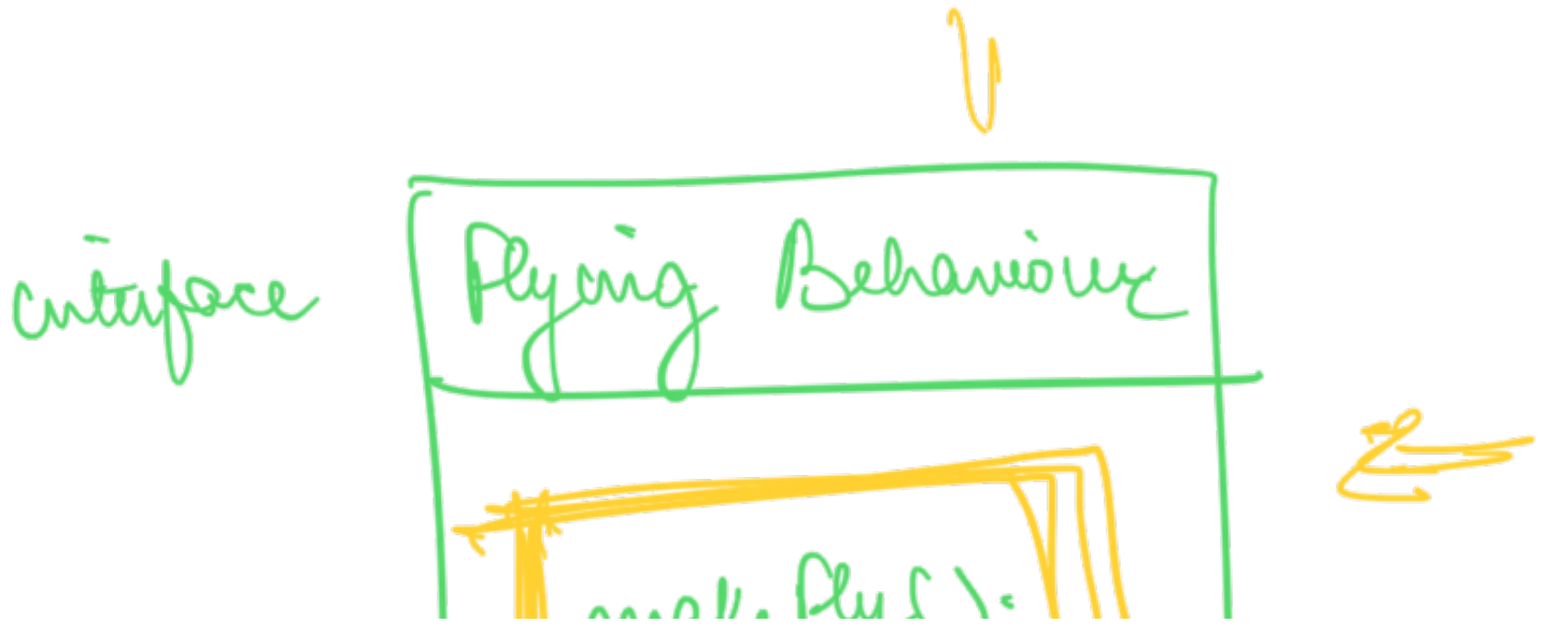


(D) ⇒

Dependency Inversion Principle

No 2 concrete classes should directly depend on each other

They should depend on each via an interface in between.



PSFB

makeFly()

}

CEFB

makeFly()

}

class Pigeon extends Bird implements Flying {

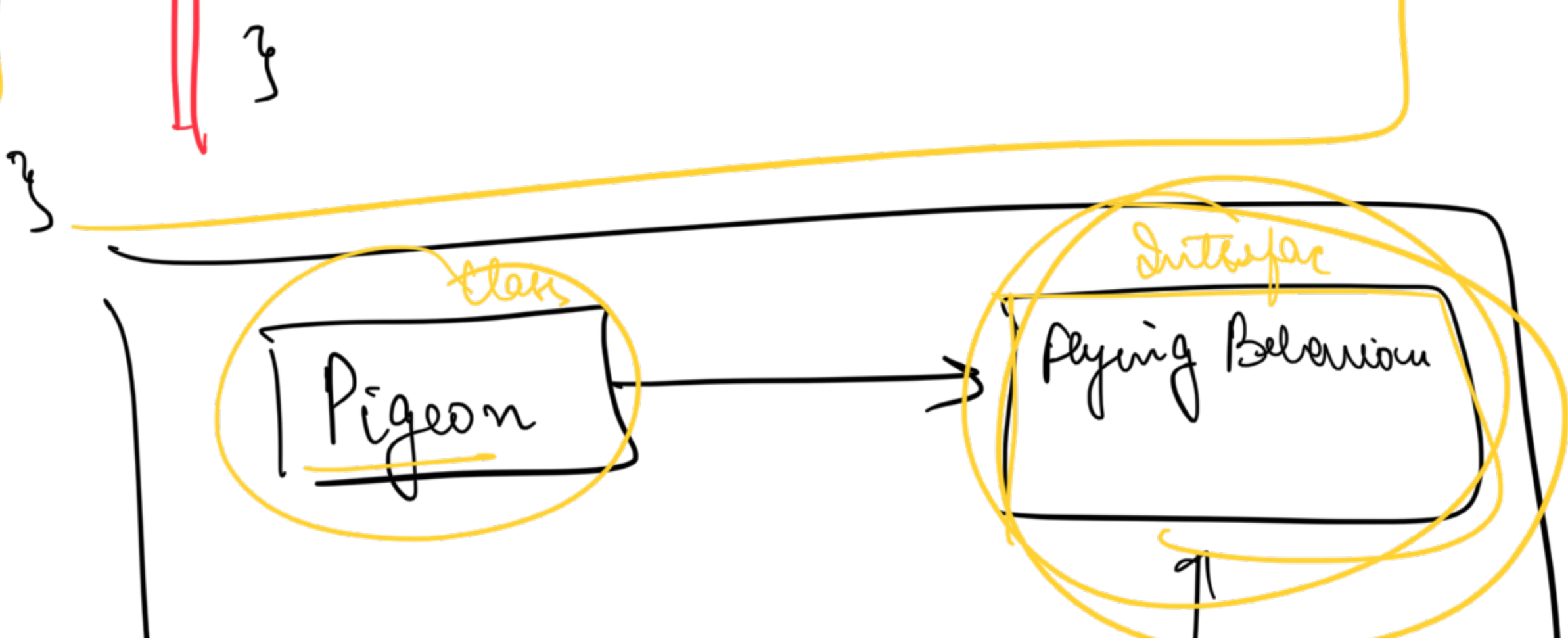
FlyingBehaviour fb = new FSFBC()

fly() {

fb.makeFly();

}

}



PSFB

Clas

Build

b = new Pizza()

b.fly()

Build

②

Bu

3

4

Dependency Injection

- class Pigeon extends Bird implements Flying {

FlyingBehaviour fb;

Pigeon (FlyingBehaviour fb) {

this.fb = fb;

}

fly() {

this.fb.makeFly();

}

Client {

Pigeon p = new Pigeon(

new PSFBC)

);

Pigeon p = new Pigeon(

new CEFBC)

);

}

Implement Tax Calculator

① Revise all DP

② Implement the final code of

Assgn

"Design a Bird"

Monday

Req

① 5 Birds:

Pigeon

Crow

Sparrow

Ostrich

Penguin

② Penguin & Ostrich don't fly.

(3) Crow and Sparrow fly in the same way.

(4) Every bird can eat() and make sound()

When a new feature has to be added, it shouldn't require you to do much changes in already existing code

instead it should mostly involve the addition of
new classes / methods