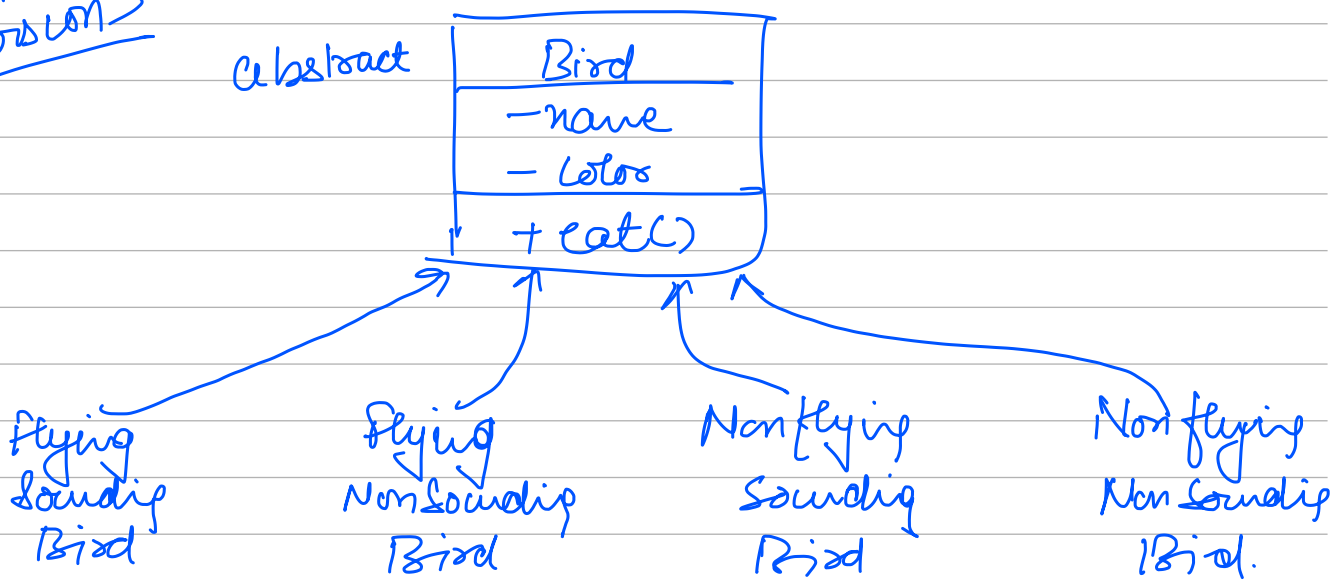
Agenda

Start @ 9:05 pm

1) Liskov's Substitution Principle
2) Interface Segregation Principle
3) Dependency Inversion Principle

Version 3



Problem statement

⇒ Some birds demonstrate some behaviors
& Some birds don't demonstrate some behaviors

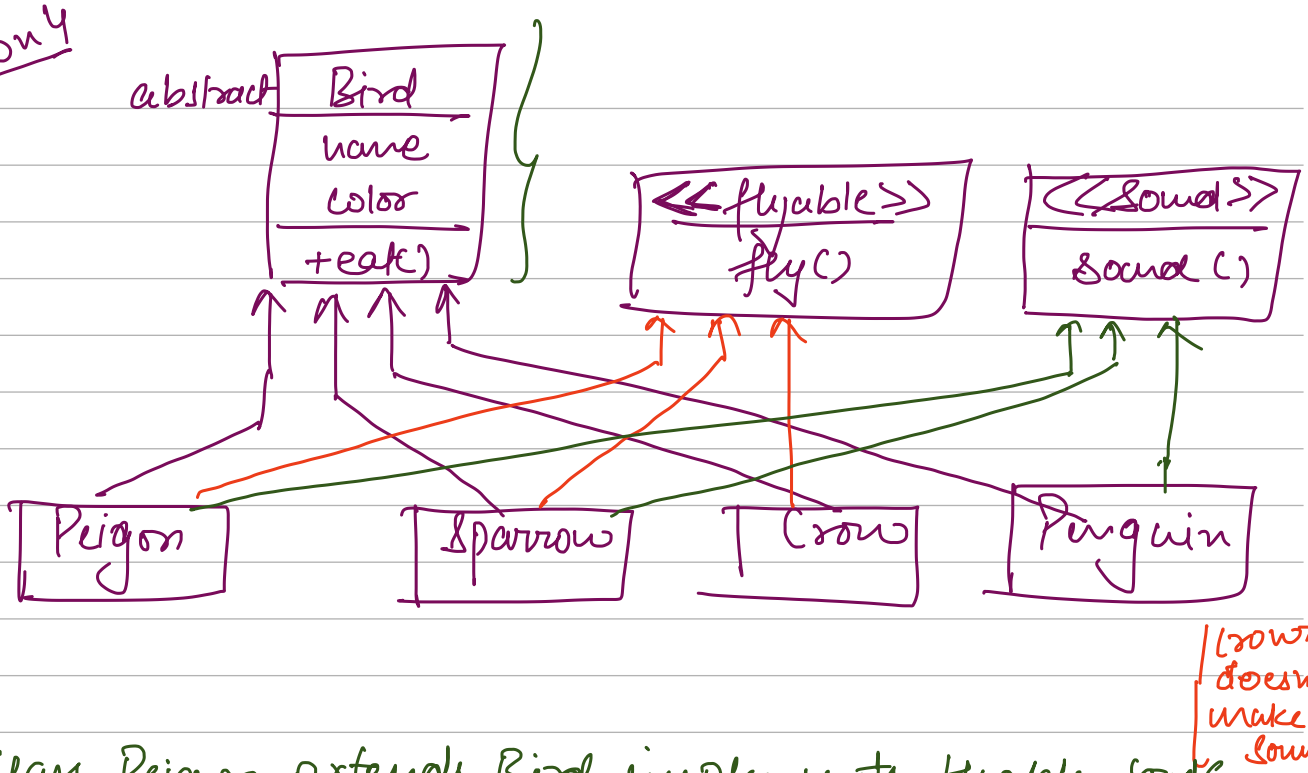
① Only the birds that demonstrate the behavior should have that method.

② We should be able to categories (create a list) based on behavior

Class ⇒ Blueprint of Entity

Interfaces ⇒ Blueprint of Behaviors

Version 4



Class Peigon extends Bird implements flyable, Sound{

eat() {—}

fly {—}

Sound {—}

}

Class penguin extends Bird implements Sound{

eat() {—}

Sound() {—}

}

→ No class explosion

→ List of all Birds — List<Birds>
List of all flyable Birds — List<flyable>
List of sound making Birds — List<Sound>

Liskov's Substitution Principle

Object of any child class should be as-is substitutable in the reference of parent class without making any change

```
Bird b = Any Bird Object
       = new Pugwin();
         new Pigeon();
```

eg

```
Bird b = new Pigeon();
         new Penguin();   // Version 2
                          // Version 4
try {
    b.fly();   // Version2
}
catch {
}
```

Code changes

2^N classes

No Child class should give any special meaning to its parent behaviour, better not have the behavior itself

<<flyable>>

```
{ Bird b = new Pugwin(); Pigeon();
  b.eat(); }
```

```
Sound b = new Pugwin
          new Crow
          new Pigeon
{ b.sound(); }
```

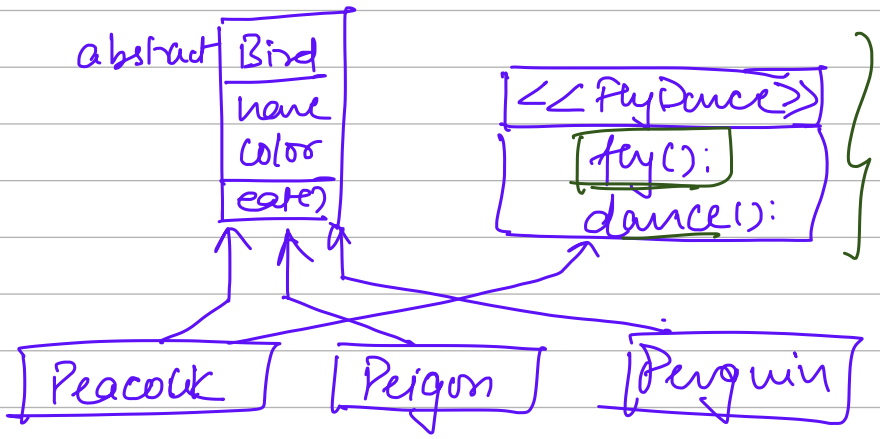
Interface Segregation Principle

Requirements

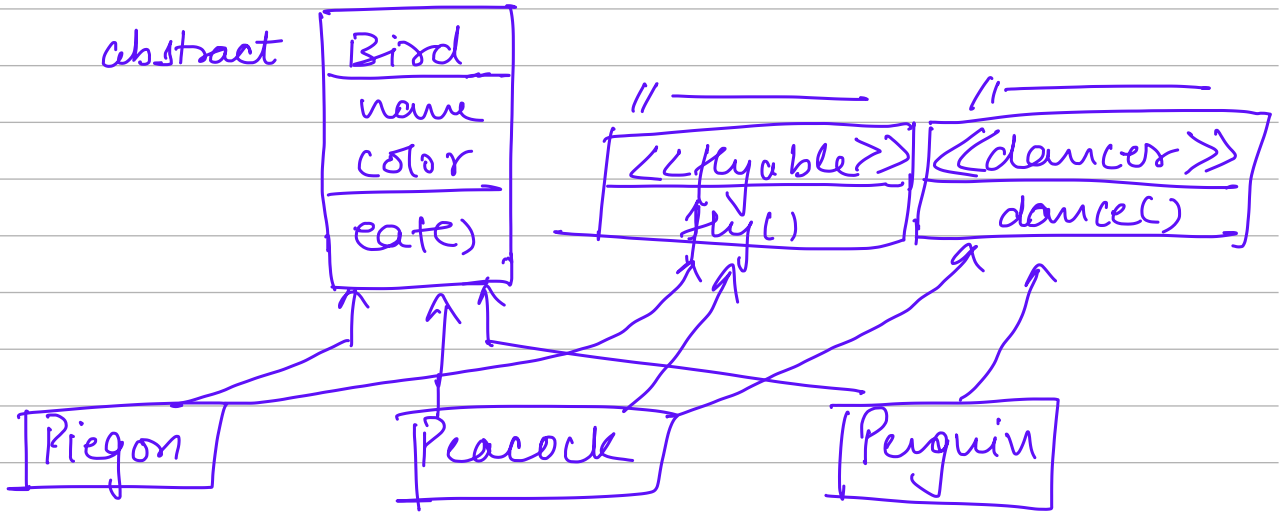
① Some birds can fly
② Some birds can dance
③ birds who fly can dance also and vice versa

→ Either a bird will fly & dance both or None

Soln ①



②



Interface Segregation Principle

1) Interface should be as light as possible

2) Each interface should have lesser number of functions, Ideally we should have only 1 function

   ⇒ Functional Interfaces

3) If more than 1 functions are logically connected then it is fine.

4) SRP for interfaces

Doubt

```
eating (Bird b){
    b.eat();
}
```

Version 4

```
flying (Bird b)
{
    b.fly();   X
}
```

```
class Pigeon extends Bird implements flyable
```

Version 2

```
flying (Bird b)
{
    try{
    b.fly();
    }catch
}
```

Penguin (flyable f)

Phonpe

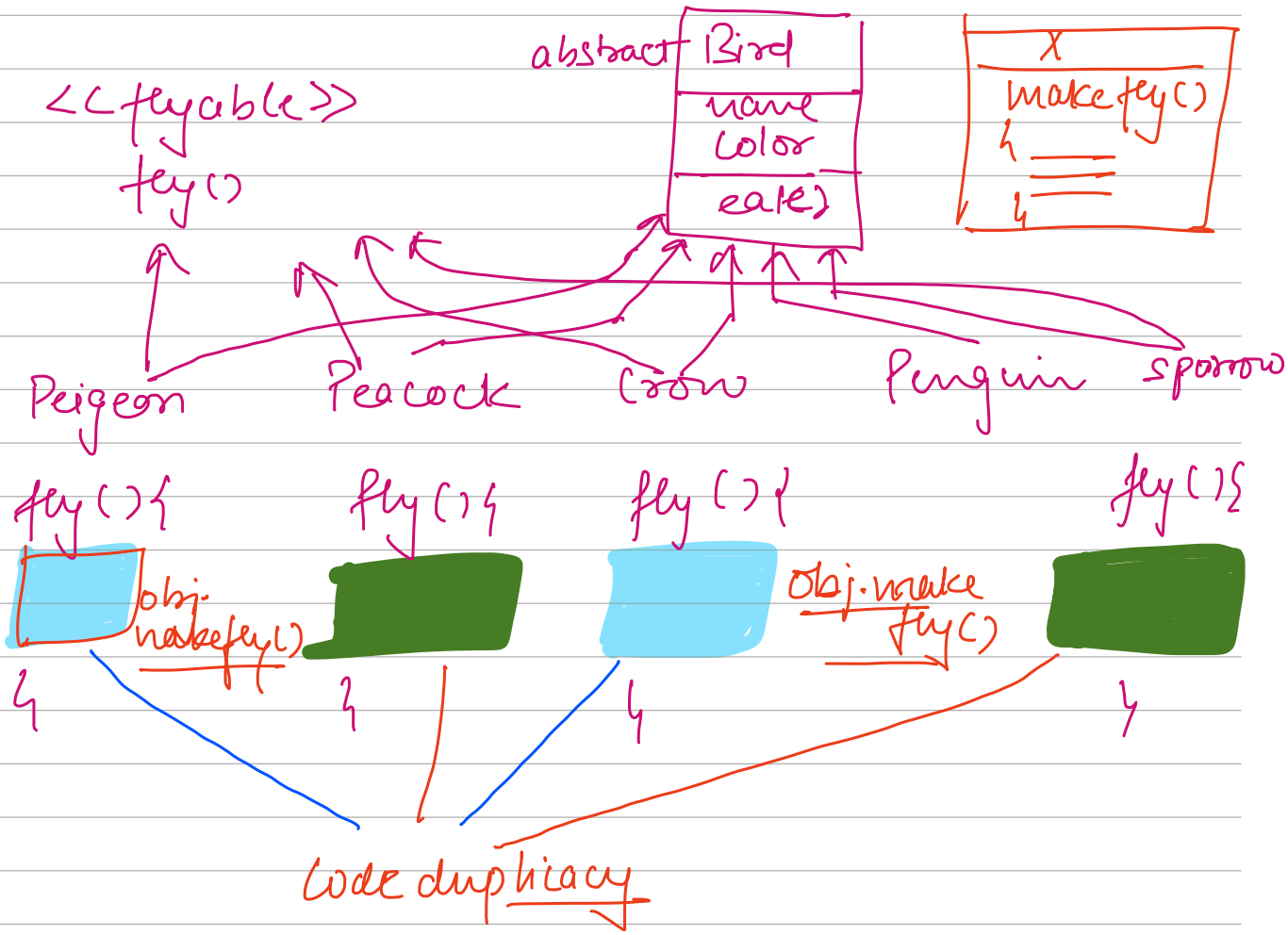
RBI Api → Interface

ICICI Bank | → HDFC Bank

boolean transfer( ) → IO behaviour

String

Dependency Inversion Principle



Requirement

1. Peigon and Crow fly in same way
2. Peacock and Sparrow fly in same way

Soln

Peigon Crow Fly Behavior ⟷ Peacock Sparrow fly Bh

makefly() { —

}

Peigon {

PCFB fb = new PCFB

fly() {

fb.makefly()

}

}

Peacock {

PSFB fb = new PSFB();

fly() {

fb.makefly();

}

}

↓

Peigon dependent on class PCFB

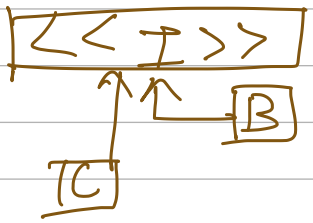
DI Principle: No 2 Concrete classes should depend on each other directly, Rather they should depend on each other via interface

A a {

B b = new B() ⇒ A depend B X

}

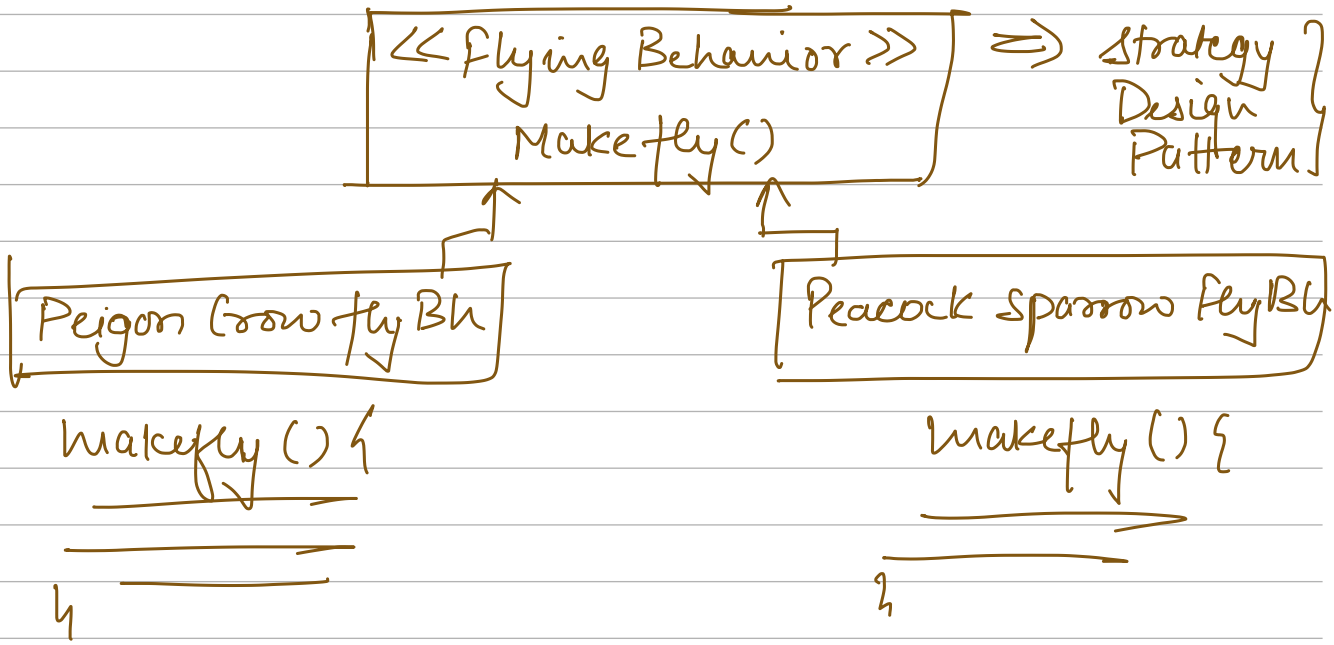
instead



A a {

I i = new B(); new C();

}



⇒ Strategy Design Pattern

Pigeon{

- FlyingBh fb = ~~new PCFB();~~ XFB();

⇒ Code is more Maintainable

fly()

fb.makefly()

Dependency Injection

→ Not a part of SOLID

→ If class A is dependent on B

A{

Bb;

}

① A{

Bb = new B();

}

② A {

B b;

A (B obj)

{ this.b = obj; }

}

{ Constructor Injection

DB Connection

(MongoDB)
Cassandra
MySQL
Dynamo DB

Dependency Injection: No need to create dependency by yourself. rather pass the dependency object via const.

A {

I i;

A (I obj) {

this.i = obj;

}

}

<< I >>

B

[C]

My Codebase isn't depending on any concrete class because we are not creating an object it.

Spring boot Framework

→ @ autowise

IOC → Inversion of Control

Dependency Injection

```
A {
    B b;
    A(B obj) {
        this.b = obj;
    }
}
```

[DI voilating]

```
Piegon {
    fly() {
        FlyingBehaviour = ( — )
    }
}
```

```
Piegon {
    Private FlyBh fb;
    Piegon(FlyBh fb)
    { this.fb = fb)
    }
    fly() {
        fb.makefly();
    }
}
```