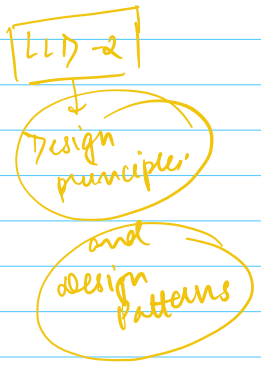
LLD → 1 ⇒ finished
Contest → coming weekend.

Agenda



S O L I D Principles.

- S → Single Responsibility Principle (SRP)
- O → Open-closed principle
- L → Liskov's Substitution Principle
- I → Interface Segregation Principle
- D → Dependency Inversion Principle

Principle:

Guidelines / Fundamentals

↓

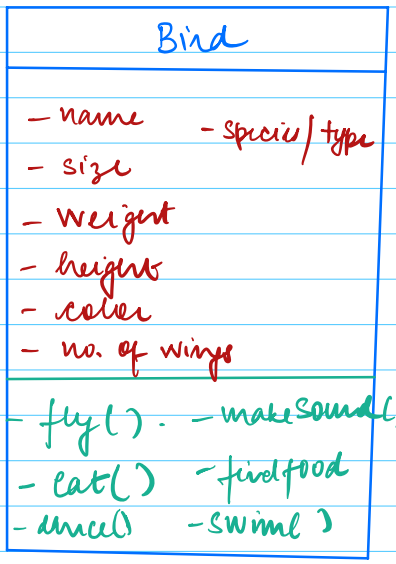
to help a SWE design better software systems

1) Maintainable → bugs fixed
2) Extensibility → easier to add on new features.
3) Reusable
4) Easily Testable → test coverage for codebase should be there
5) Modular:
6) Understandable → proper naming, indentation, comments

⇒ Design a bird

Requirements:

Build a system where we have to store information about different kinds of birds



Bird sparrow = new Bird()
sparrow.name = "sparrow"
sparrow.size = SMALL
sparrow.ht.
- "

Bird crow = new Bird()
crow.name = "crow"
crow.size = MEDIUM.
crow.ht.
- "

Attributes done

sparrow.makeSound()

emits diff sound

crow.makeSound()

emits diff sound

```
makeSound() {
    if (name == "crow")
        sout("kow kow")
    else if (name = "sparrow")
        sout("chi chi")
    else if (name == "pigeon")
        sout("gutar gu")
}
```

Problems with this code?

- → not maintainable
- → Error prone
- → Difficult to test
- → Not readable
- → Code duplication.

violating S of SOLID principles.

SINGLE RESPONSIBILITY PRINCIPLE (S.R.P)

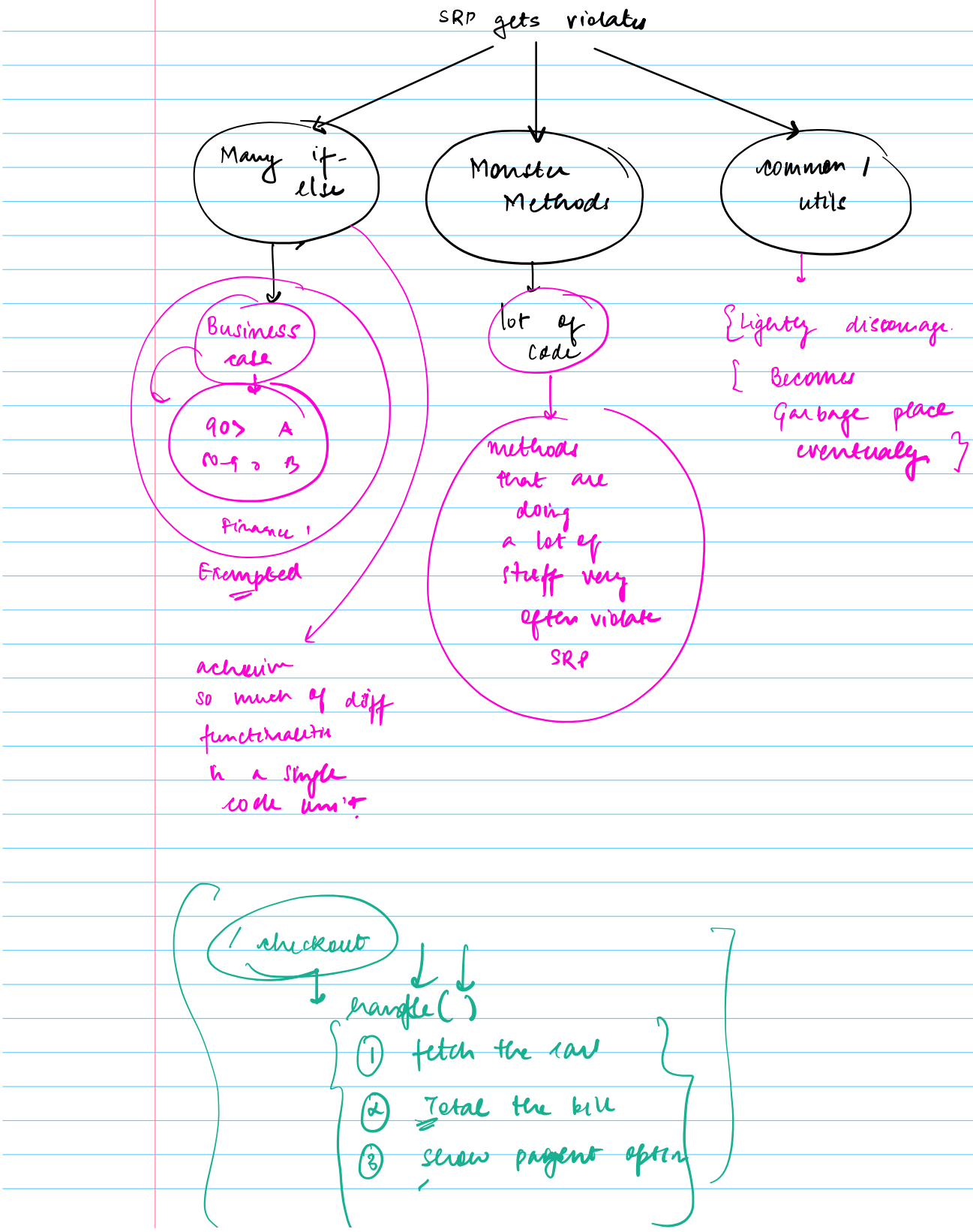
⇒ Every code unit (method / class / package) in our codebase should have exactly 1 responsibility.

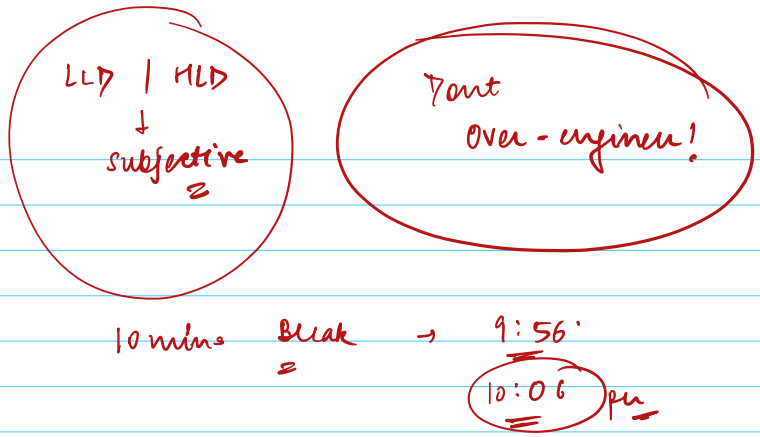
OR

⇒ There should be exactly 1 reason to change the code

⇒ makeSound()
↓
only responsible for how EVERY bird is making sound

class Bird
↳ holds attributes and methods for every kind of bird.


SRP gets violated
Many if-else
Monster Methods
common / utils
Business case
90> A
Finance
Exempted
lot of code
methods that are doing a lot of stuff very often violate SRP
{Lightly discourage.
Becomes Garbage place eventually.}
achieving so much of diff functionalities in a single code unit
/ checkout
handle()
① fetch the cart
② Total the bill
③ screw payment option



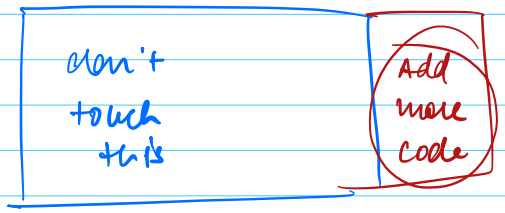
10 mins Break → 9:56
10:06 pm

OPEN CLOSED PRINCIPLE (OCP)

=) code should be open for extension but closed for modification

=) makes more extensible

=) Adding new features means adding code and not modifying old code



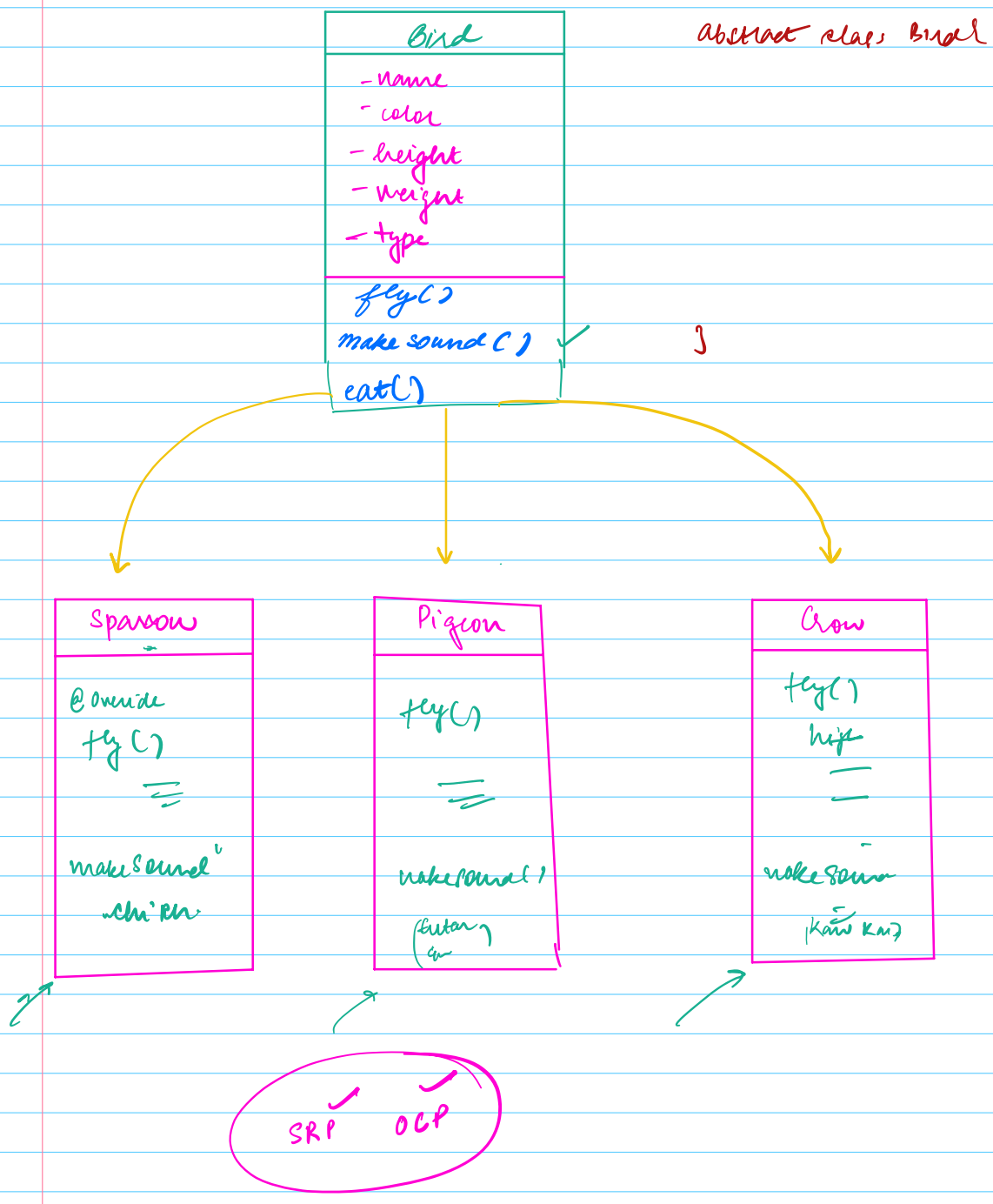
Adding new features =) Minimal modification

Bird design make sound() violating

v2

SRP and OCP should be followed

abstract class → Bird.



Now, to add a Bird →
We will not modify any of existing code/.

We will create a new class.

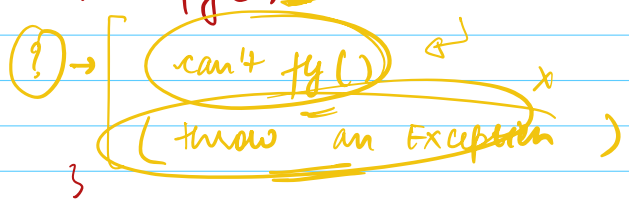
Introduce → Penguin

Bird

```
class Zoo {
    List<Bird> birds = {
        new Sparrow();
        new Pigeon()
        new Crow()
    }
    birds.stream().map(bird -> bird.fly()).toList()
}
```

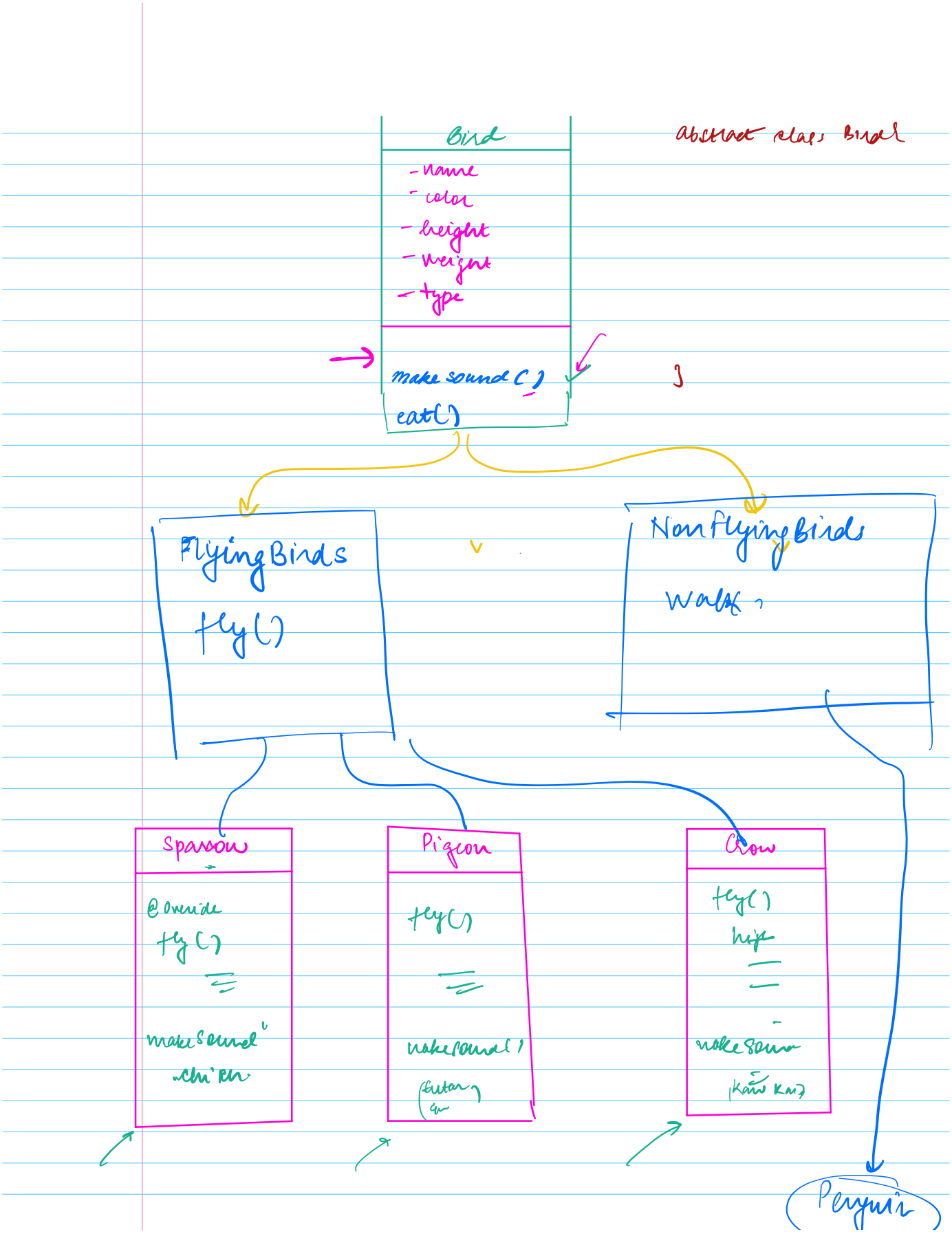
class Penguin extends Bird {

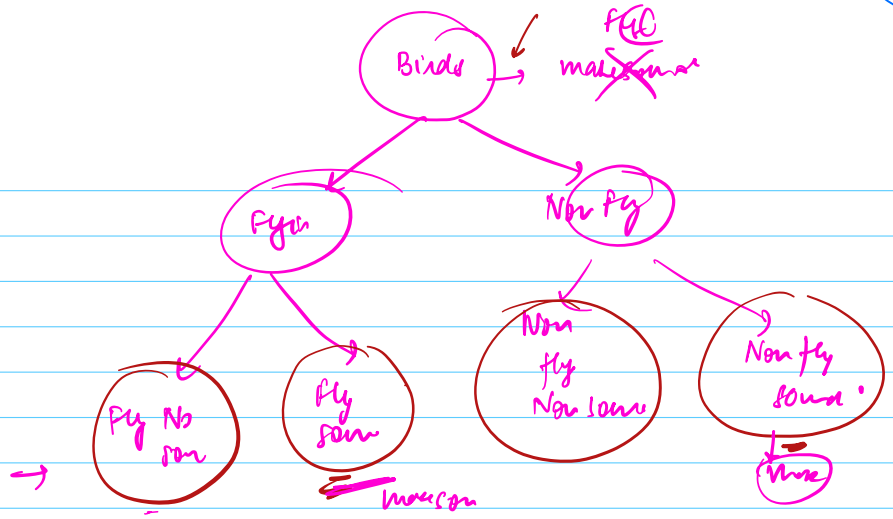
@Override
void fly() {



}

}


Bird
- name
- color
- height
- weight
- type
makeSound()
eat()
abstract class Bird
FlyingBirds
fly()
NonFlyingBirds
Walk
Sparrow
@Override
fly()
Pigeon
fly()
Crow
fly()
Penguin



Problem (1) → class Explosion

N behaviours

2^N → classes

(2) → can I have a common class on which I could call • makeSound()

birds.stream().map(bird → bird.makeSound())

List<?> = • makeSound

Fowy Fly → List<FlyingBirds>

• fly()

List < Fly sound + NonFly sound >

X . makeSound ()

LSP
↓