SOLID

Today
- S — Single Responsibility principle
- O — open close principle

Next class
- L — Liskov's substitution
- I — Interface segregation
- D — Dependency Inversion

SOLID design principles → Rules / Guidelines / Fundamental

better software design

1) Extensible
2) Maintable
3) Testable
4) Readable
5) Reusable
6) Modular
7) Robust

Design a BIRD

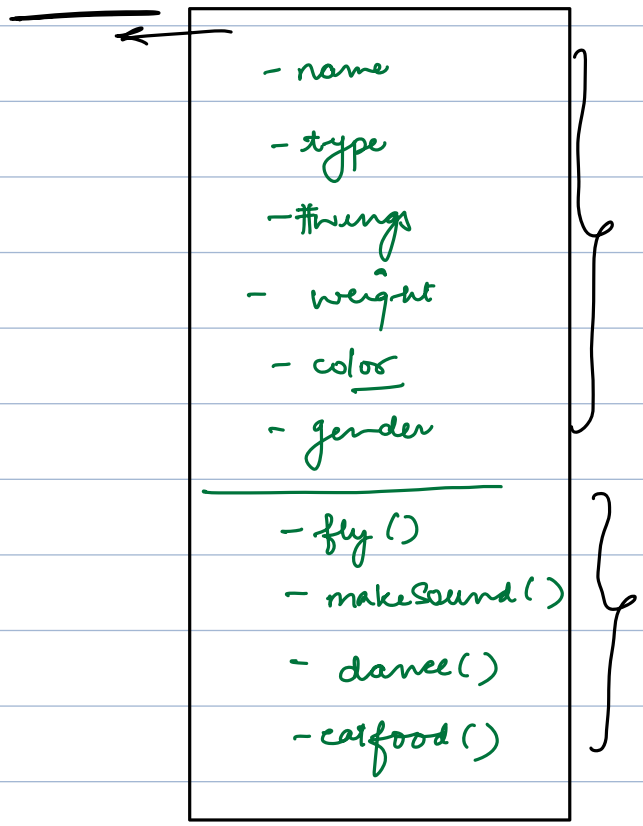
Req^n :- build a software system where you store the info & behaviour a bird

↓

Diversity of Birds

Bird

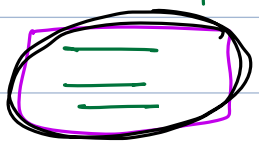
class Bird




```
Bird b1 = new Bird();
    b1.type = "sparrow"
    b1.wings = 6
        :
Bird b2 = new Bird()
    b2.type = "eagle"
        :
b1.fly()          b1.makesound()
b2.fly()          b2.makesound()
```

makeSound ( ) {

if (type == "sparrow")



else if (type == 'eagle")

else if 

else if ——

else :

}

(1) Understandable / Readable

(2) Difficult to test

(3) Code duplication

(4) Code is not Reusable

(5) Extensibility / Mai

(6) It violates (S) of SOLID

Single Responsibility Principle

→ Every code unit (class / Method / Package) in codebase should have exactly 1 responsibility.

↓

There should only be one reason to change the code.

makesound() — responsible for every bird's sound.

Bird class is also not following SRP.

LLD – subjective

Don't overengineer

How to identify violation of SRP

1. Method have lot of if-else.

checkleapyear()

✓ business logic

multiple if-else

take a input

```
if (input == 1)
{
    startdb
}
else if (input == 2)
{
    playmu
}
else
{
    shutdown
}
```

startdb()

independent

playmu()

shutdown()

(2) Monster method

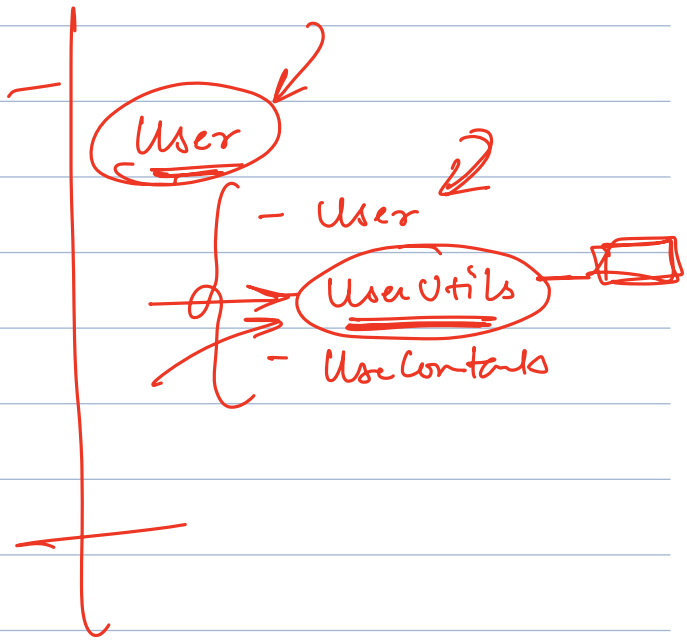
↓

when the method is doing more than what the method name suggests.

```
saveToDatabase ( User user, ~~Database db~~) {

① {  query = ' Insert ... ' ; }

② {  DataBase db = new DataBase();
     db.setUrl();
     db.makeConnect

③ {  db.execute( query, User);

}
```

Code Duplication → SRP

③ utilities / common



OCP :- open close principle

↓

codebase should be open for extensions

but closed for Modifications

easy to add new features → should n't require changes in existing code.

Do lesser modifications if not completely avoidable

```
makesound() {
    if (   )
    else if(   )
    else if (   )
    else if (   )
    else (   ) {
    }
}
```

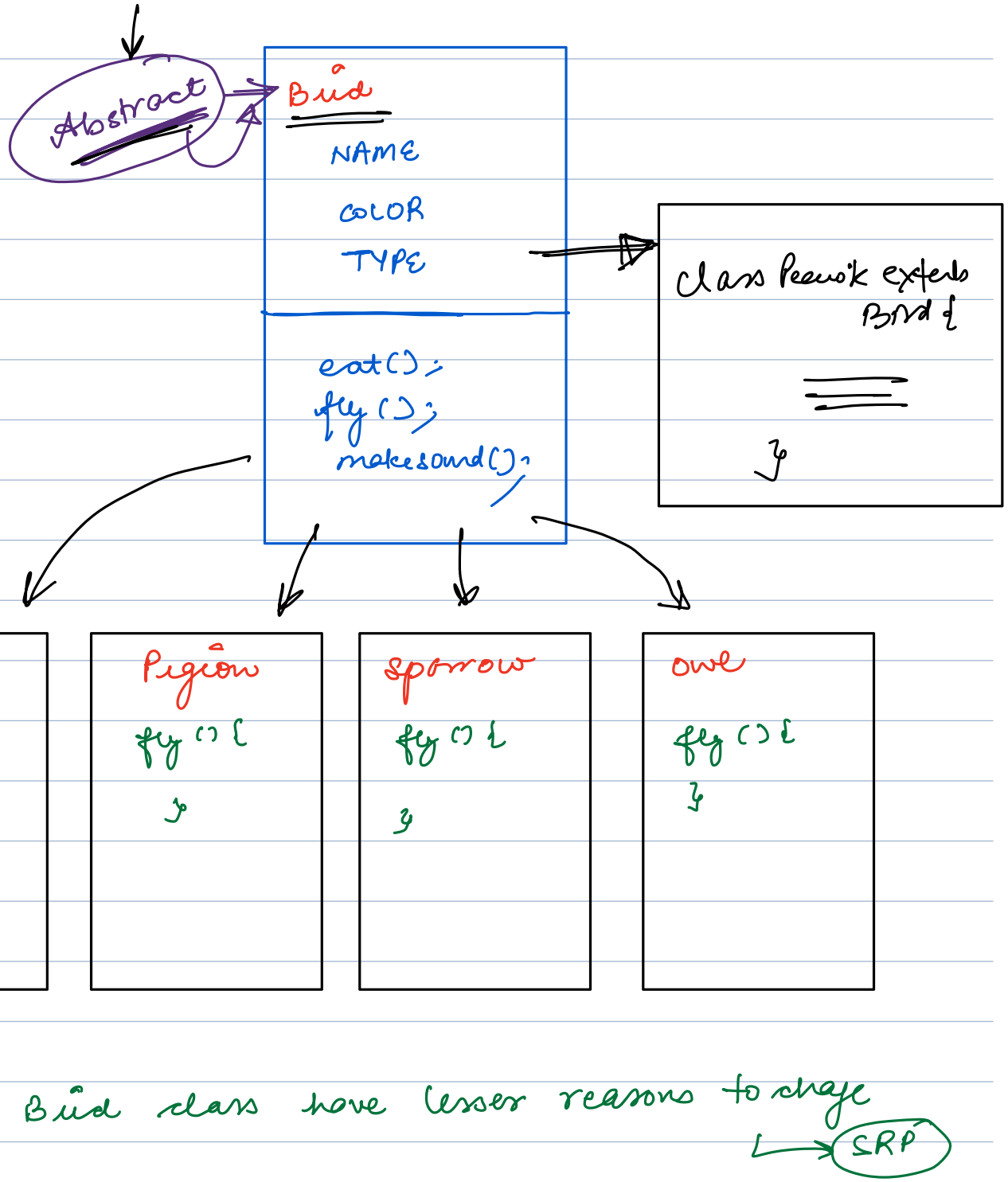
- sparrow
- crow
- Pegion
- owl
- Peacock

1) Testing
2) Regression

when because of a new addition something which was working breakes

Bird class is responsible for generic details and not specific

V2



* Bird class have lesser reasons to change → SRP

Penguin → extends Bird

flying

① fly()
{
// leave it empty
}

② fly()
{ Throw an exception BirdCantFly()
}

Bird b = new Penguin();
b.fly();

Bird can't fly

⇒ { Reduce surprises for clients }

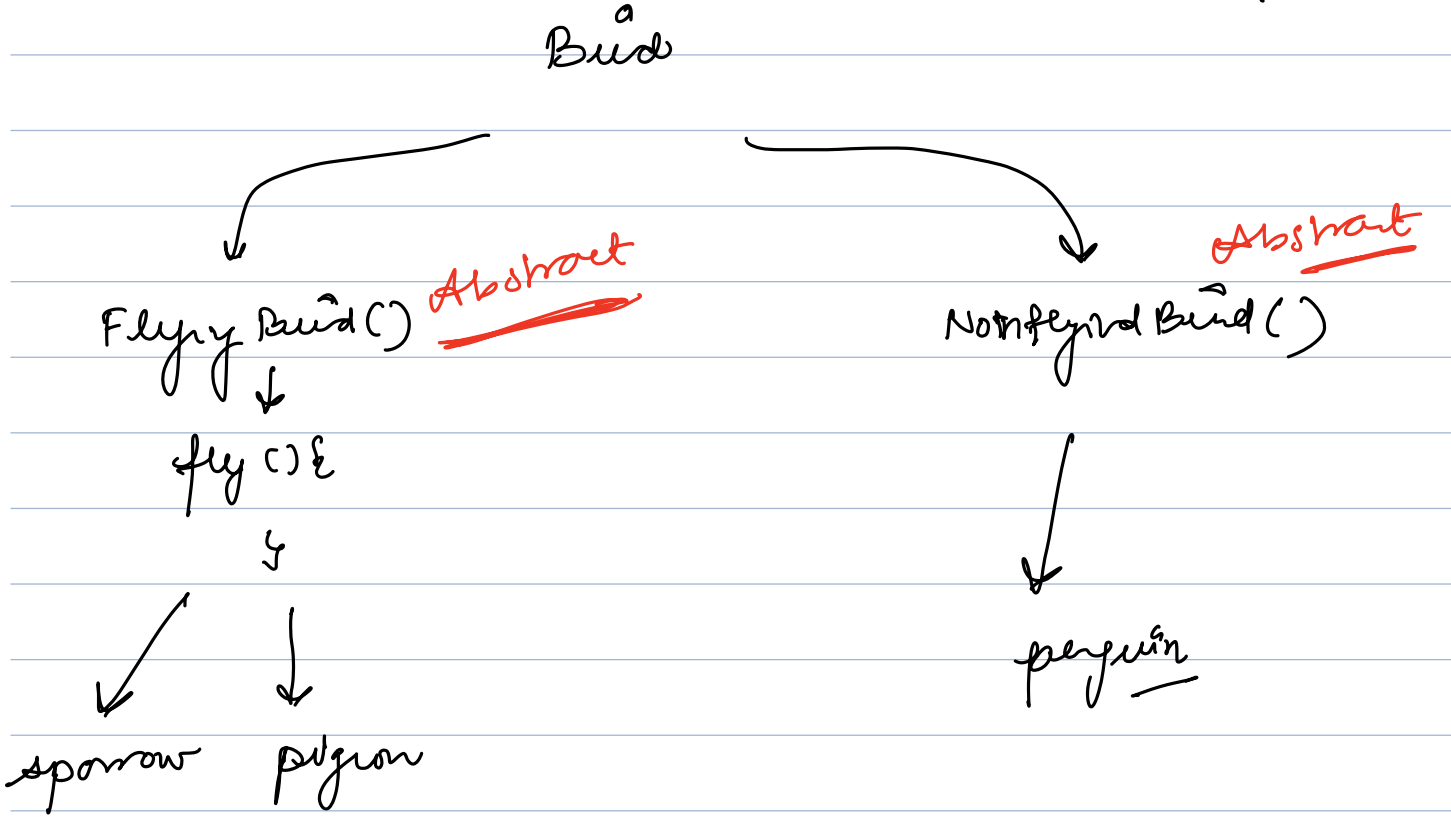
Ubereats — surprise — Paytm

person
1) tried pay first
↓ Error
2) second
↳ " X "

if (response == ERROR)
don't get order to
cv
else { order successful }

ideal :- If an entity doesn't support behaviour, it should not have method.

Some Birds can fly and some can't



Some birds can make sound some can't;

SF

CS, F

S, CF

CS, CF

N attributes → 2^n ⇒ class explosion

DP

Case studies

SRP

OCP

→ solve

Problem

→ W